Noname manuscript No.
(will be inserted by the editor)

Building on the Diamonds between Theories: Theory Presentation Combinators

Jacques Carette · Russell O'Connor · Yasmine Sharoda

Received: date / Accepted: date

Abstract To build large libraries of mathematics, it seems more promising to take advantage of the inherent structure of mathematical theories. Various theory presentation combinators have been proposed, some have been implemented in specification systems but not in interactive theorem provers. Even in systems that implement such features, they seem under-used in their own standard libraries.

Inspired by combinators originating in Clear and Specware and their descendents (both direct and intellectual), we present variants of these combinators optimized for building libraries in the setting of interactive theorem provers over dependent type theories. The main technical contribution is that our combinators draw their power from the inherent structure already present in the *category of contexts* associated to a dependently typed language. We have also implemented the system.

Keywords Mechanized mathematics, theories, combinators, dependent types

1 Introduction

The usefulness of a mechanized mathematics system relies on the availability of a large library of mathematical knowledge, built on top of sound foundations. While sound foundations contain many interesting intellectual challenges, building a large library seems a daunting task simply because of its sheer volume. However, as has been documented [14, 18, 37], there is a tremendous amount of redundancy in existing libraries. Thus there is some hope that by designing a good meta-language, we can reduce the effort needed to build a library of mathematics.

J. Carette
Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada,
E-mail: carette@mcmaster.ca ORCiD: 0000-0001-8993-9804

R. O'Connor
Blockstream.com,
E-mail: roconnor@theorem.ca

Y. Sharoda
Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada,
E-mail: sharodym@mcmaster.ca

Our aim is to build tools that allow library developers to take advantage of commonalities in mathematics so as to build a large, rich library for end users, whilst expending much less actual development effort than in the past. Our means are not in themselves new: combinators for combining theories, with a clear categorical semantics. The design space is large and complex and has been explored for decades. We survey the related work in Section 8. What is new is that we leverage the surrounding structure already present in dependent type theories, to help us decide which combinators to focus on as they are, in some sense, already present. Using combinators in the setting of interactive theorem proving for Martin-Löf Type Theory is also new. Lastly, we squarely target system developers rather than casual users.

These combinators, as well as creating new theories, also build a graph of theories connected by theory morphisms. These make it easy to transport results between theories, thereby increasing automation [30]. This also represents the continuation of our work on *High Level Theories* [13] and *Biform Theories* [15] through building a network of theories, leveraging what we learned through previous experiments [14]. We have done several prototype implementations, with three prototypes [19,3,67] that really allowed us to better understand the design space. We also have a solid version [16] that is publicly available[1].

1.1 The Context

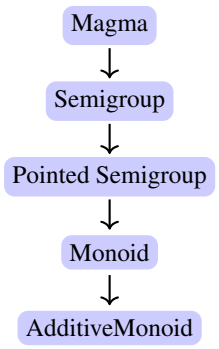

Fig. 1 Theories

Why would the development of mathematical libraries be different than other software, i.e. why would there be effort saving techniques that are generally not worth seeking [9]? Because we know since Whitehead's 1898 text "A treatise on universal algebra" [79] that significant parts of mathematics have a lot of structure, structure which we can take advantage of. The Jipsen's [50] list of 342 structures is both impressively large and can still be greatly extended. Another beautiful source of structure in a theory graph is that of *modal logics*; Halleck's web pages on Logic System Interrelationships [40] is quite eye opening.

Figure 1 is a common picture of the situation, where arrows denote extensions. Strict inclusions at the level of presentations is only part of the structure: a `Ring` actually contains two isomorphic copies of `Monoid`, where the isomorphism is given by a simple *renaming*. There are further commonalities to take advantage of, which we will explain later. However, the natural structure is not linear but full of diamonds, as in Figure 2. In Computer Science, this is known as *multiple inheritance* and the diamonds in inheritance graphs are much feared, giving rise to *The Diamond Problem* [7,26,81], or fork-join inheritance [68]. In our setting, we will find that these diamonds are a blessing rather than a curse, because they give theory-level sharing information, rather than being related to dynamic-dispatch. Tom Hales [39, point # 9] is unhappy that Lean does not permit this (neither do Coq [77], Agda [60] or Idris [8]).

But is there sufficient structure outside of universal algebra, i.e. single-sorted equational theories, to make it worthwhile to develop significant infrastructure to leverage that structure? There is: generalized algebraic theories [21] are a rich source, which encompasses categories, bicategories, functors, etc. as examples.

[1] `https://github.com/ysharoda/tog`

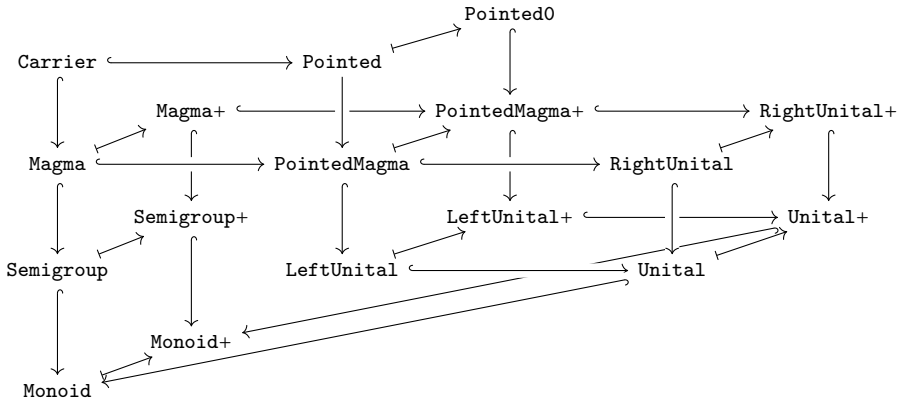

Fig. 2 Structure of the algebraic hierarchy up to Monoids

We note that *in practice*, when mathematicians are *using* theories rather than developing new ones, they tend to work in a rather "flat" namespace [13]. An analogy: someone working in Group Theory will unconsciously assume the availability of all concepts from a standard textbook, with their usual names and meanings. As their goal is to get some work done, whatever structure system builders have decided to use to construct their system should not *leak* into the application domain. They may not be aware of the existence of pointed semigroups, nor should that awareness be forced upon them. Thus we need features that "flatten" a theory hierachy for some classes of end users. On the other hand, some application domains do rely on the "structure of theories", so we cannot unilaterally hide this structure from all users either.

1.2 Contributions

This paper is a substantial rewrite of [20], where a variant of the *category of contexts* was used as our setting for theory presentations. There we presented a simple term language for building theories, along with two (compatible) categorical semantics, one in terms of objects, another in terms of morphisms. By using "tiny theories", this allowed reuse and modularity. We emphasized names, as the objects we are dealing with are syntactic and ultimately meant for human consumption. We also emphasized morphisms: while this is categorically obvious, nevertheless the current literature is largely object-centric.

We extend the work in multiple ways. We pay much closer attention to the structure already present in the categorical semantics of dependent type theories. In particular, we extend our semantics to a fibration of generalized extensions over contexts. This is not straightforward: not clobbering users' names prevents us from having a cloven fibration without a renaming policy. But once this machinery is in place, this allows us to build presentations by lifting morphisms over embeddings, a very powerful mechanism for defining new presentations. There are obstacles to taking the "obvious" categorical solutions: for example, having *all* pullbacks would require that the underlying type theory have subset types, which is something we do not want to force. This is why we insist on having users provide an explicit renaming, so that they remain in control of the names of their concepts. Furthermore, equivalence of terms needs to be checked when constructing mediating morphisms, which

in some settings may have implications for the decidability of typechecking. We also give complete algorithms as well as a type system.

While we are far from the first to provide such combinators, our requirements are sufficiently different than previous work, that we arrive at a different solution in this large design space, driven by what we believe to be an elegant semantics.

1.3 Plan of paper

We motivate our requirements through concrete examples in Section 2. Section 3 lays out the basic (operational) theory, with concrete algorithms. The theoretical foundations of our work, the fibered category of contexts, is presented in full detail in Section 4, along with the motivation for why we chose to present our semantics categorically. This allow us in Section 5 to formalize a language for theory presentation combinators and present a type system for it. We close with some examples, discussion, related work and conclusions in Sections 6–9.

2 Motivation

The motivation for combinators is clear and we will not repeat it. As we want slightly different behaviour than other systems, we give a quick introduction to each, informally. Here, we use an informal syntax which should be understandable to someone with a background in mathematics and type theory; section 5 will formalize everything. We highlight the issues that arise with the "intuitive" combinators when we try to use them at scale. This helps to establish our requirements for a sound solution. This coherent semantics (developed in Sections 3 and 4) will then lead us to rebuild our formal language, including its syntax, in Section 5. After introducing the combinators, we proceed to present some of our design decisions.

Our perspective is that of *system builders*. In particular, it is imperative that we respect the syntactic choices of users, even when these choices are not necessarily semantically relevant. In other words, for theory presentations, de Bruijn indices (for example) are unacceptable, but so are generated names.

2.1 Overview of Combinators

Note that we use the term "combinator" where others might have used "construction". And indeed, what we present below are *algorithmic constructions* that take a *presentation*, perhaps encoded as an algebraic data type, and produce another *presentation* in the same language.

2.1.1 Extension

The simplest situation is where the presentation of one theory is included, verbatim, in another. Concretely, consider `Monoid` and `CommutativeMonoid`, which differ only by a `commutative` axiom. Thus, given `Monoid`, it would be much more economical to define

$$\texttt{CommutativeMonoid} \triangleq \texttt{Monoid}\ \textbf{extended\_by}\ \{\texttt{commutative} : \forall x, y : U . x \circ y = y \circ x\}$$

Note that some systems implement this literally, where `Monoid` is *include*d in `CommutativeMonoid`. While this can be implemented by just appending a new axiom, semantic validity requires checking that the new name is indeed new and that the new type (here the commutativity axiom) is well-typed in the context of the previous definitions.

2.1.2 Renaming

From an end-user perspective, our `CommutativeMonoid` has one flaw: these are frequently written *additively* rather than multiplicatively. Let us call a commutative monoid written additively an `AdditiveCommMonoid`. Thus we would like to say

$$\texttt{AdditiveCommMonoid} \triangleq \texttt{CommutativeMonoid}[\ \circ \mapsto +,\ e \mapsto 0]$$

But how are `AdditiveCommMonoid` and `CommutativeMonoid` related? Traditionally, these are regarded as *equal*, for semantic reasons. However, since we are dealing with presentations, we wish to regard them as *isomorphic* rather than equal[2] . While working up to explicit isomorphism is a minor inconvenience for the semantics, this enables us to respect user choices in names.

Note that many theorem proving systems do not have a renaming facility and indeed their libraries often contain both additive and multiplicative monoids that are inequivalent (for example, Lean's library does this as of September 2020). This makes using the Little Theories method [30] less useful.

2.1.3 Combination

Using these features, starting from `Group` we might write

$$\texttt{CommutativeGroup} \triangleq \texttt{Group}\ \textbf{extended\_by}\ \{\texttt{commutative} : \forall x, y : U . x \circ y = y \circ x\}$$

which is problematic: we lose the relationship that every commutative group is also a commutative monoid, and introduce needless duplication. In other words, we reduce our ability to transport results "for free" to other theories and must prove that these results transport, even though the morphism involved is (essentially) the identity. We need a feature to express sharing. Taking a cue from previous work, we might want to say

$$\texttt{CommutativeGroup} \triangleq \textbf{combine}\ \texttt{CommutativeMonoid}, \texttt{Group}\ \textbf{over}\ \texttt{Monoid}$$

This can be read as saying that `Group` and `CommutativeMonoid` are both "extensions" of `Monoid` where `CommutativeGroup` is formed by the union (amalgamated sum) of those extensions. In other words, by **over**, we mean to have a single copy of `Monoid`, to which we add the extensions necessary for obtaining `CommutativeMonoid` and `Group`. This implicitly assumes that our two `Monoid` extensions are meant to be orthogonal, in some suitable sense.

Unfortunately, while this "works" to build a sizeable library (say of the order of 500 concepts), it is nevertheless brittle. By `combine`, we really mean *pushout*. But a pushout is an operation defined on 2 morphisms (and implicitly 3 objects); our syntax gives the 3 objects and leaves the morphisms implicit. Can we infer the morphisms and prove that they are uniquely determined? Unfortunately not: these morphisms are (in general) impossible to infer, especially in the presence of renaming. As mentioned previously, there are two distinct morphisms from `Monoid` to `Ring`, with neither being "better" or somehow more canonical

[2] Univalent Foundations [78] does not change this, as we **can** distinguish the two, *as presentations*.

than the other. In other words, even though our goal is to produce *theory presentations*, using pushouts as a fundamental building block gives us no choice but to *take morphisms seriously*.

2.1.4 Mixin

There is one last annoying situation:

$$\texttt{LeftUnital} \triangleq \texttt{PointedMagma}\ \textbf{extended\_by}\ \{\texttt{leftIdentity} : \forall x : U.e \circ x = x\}$$
$$\texttt{RightUnital} \triangleq \texttt{PointedMagma}\ \textbf{extended\_by}\ \{\texttt{rightIdentity} : \forall x : U.x \circ e = x\}$$
$$\texttt{Unital} \triangleq \textbf{combine}\ \texttt{LeftUnital}, \texttt{RightUnital}\ \textbf{over}\ \texttt{PointedMagma}$$

The type of `rightIdentity` arises from flipping the arguments to $\circ$. Can we capture this transformation and automate it? We will call this "mixin" as this construction bears a strong resemblance to that of mixins in programming languages with traits.

2.2 Morphisms

Our constructions also describe how the symbols of the source theory can be mapped into expressions of the target theory. For extensions, this is an injective map. In other words,

$$\texttt{CommutativeMonoid} \triangleq \texttt{Monoid}\ \textbf{extended\_by}\ \{\texttt{commutative} : \forall x, y : U.x \circ y = y \circ x\}$$

lets us see a `Monoid` inside a `CommutativeMonoid`. More explicitly, this means a definition of all symbols all symbols of `Monoid` in terms of those in `CommutativeMonoid`:

$$\begin{aligned}\texttt{MtoCM} \triangleq [&\texttt{U} \mapsto \texttt{U}, \circ \mapsto \circ, \texttt{e} \mapsto \texttt{e},\\ &\texttt{right\_identity} \mapsto \texttt{right\_identity}, \texttt{left\_identity} \mapsto \texttt{left\_identity},\\ &\texttt{associative} \mapsto \texttt{associative}] : \texttt{Monoid} \Rightarrow \texttt{CommutativeMonoid}\end{aligned}$$

which is a tedious way of writing out the identity morphism. *Display maps* [48] express the same idea but focus on what is to be *dropped*: $\texttt{MtoCM} \triangleq \delta_{\texttt{commutative}}$.

A *renaming* is a *bijection* on names.

Combine Combinations create morphisms too but unfortunately choosing names for symbols in the resulting theory (when there are clashes) can be a problem: there are simple situations where there is no canonical name for some of the objects in the result. For example, take the presentation of `Carrier`, aka $\{U : \texttt{Type}\}$ and the morphisms induced by the renamings $U \mapsto V$ and $U \mapsto W$; while the result will necessarily be isomorphic to `Carrier`, there is no canonical choice of name for the end result. This is one problem we must solve. Figure 3 illustrates the issue. Thus we need to

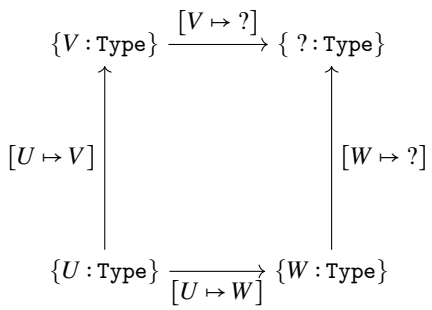

Fig. 3 The need for choosing names when combining theories.

compute *amalgamated sums* and not simply syntactic union. As the names U, V and W are assumed to be meaningful to the user, we do not wish to gensym a *fresh* name to replace the ? in Figure 3. This would give a valid pushout but it is not a good idea to invent names for concepts that ought be be both meaningful and allow further compositional extensions.

These 3 combinators suffice to build a fairly sizable library [14]. Extensions are used to introduce new symbols and concepts, renames to ensure the "usual" name is used in context, and then larger theories are built up using combine. For instance, our library defines a *binary operator* only once, and renamed to the usual names used in context in mathematics.

General morphisms We can give morphisms explicitly as well. These can be written as a sequence of assignments of valid terms of Q for each symbol of P, of the right type. For example, one can witness that the additive naturals numbers form a monoid (i.e. a morphism from `Monoid` to `Nat`) by showing how to define the symbols of `Monoid` in terms of those of `Nat`:

$$\texttt{view Nat as Monoid via } [U \mapsto \mathbb{N}, \circ \mapsto +_{\mathbb{N}}, e \mapsto 0, \ldots] \tag{1}$$

where we elide the proofs. Note that while the above looks like a rename, it is not as the right hand sides are defined, well-typed terms. When a general morphism is valid in the empty context, it is akin to *instances* in Haskell, where a Haskell `class` is akin to a theory presentation. A similar analogy holds for signatures and structures in Ocaml and Standard ML, as well as traits and instances in Scala.

General morphisms do not have to be to the empty context. For example, we can have a morphism from `Magma` to itself which maps the binary operation to its opposite:

$$\left\{ \begin{array}{l} U : \texttt{Type} \\ \_\circ\_ : U \to U \to U \end{array} \right\} \xrightarrow{[U \mapsto U, \circ \mapsto \lambda xy \cdot y \circ x]} \left\{ \begin{array}{l} U : \texttt{Type} \\ \_\circ\_ : U \to U \to U \end{array} \right\} \tag{2}$$

2.3 Little Theories

An important observation is that *contexts* of a type theory (or a logic) contain the same information as a *theory presentation*. Given a context, theorems about specific structures can be constructed by transport along theory morphisms [30]. For example, in the context of the definition of `Monoid`, we can prove that the identity element, e, is unique:

$$\forall e' : U. \left(\left(\forall x. e' \circ x = x \right) \vee \left(\forall x. x \circ e' = x \right) \right) \to e' = e$$

In order to apply this theorem in other contexts, we can provide a theory morphism from one presentation to another. There are two natural morphisms from `Monoid` to `Semiring`, induced by the renamings $[\circ \mapsto +,\ e \mapsto 0]$ and $[\circ \mapsto \times,\ e \mapsto 0]$. Both of these can be used to transport our example theorem to prove that 0 and 1 are the unique identities of their respective associated binary operations.

There are also many more morphisms; for example, we could send $(\circ)$ to $\lambda x, y : U. y \circ x$. Thus, in general, we cannot infer morphisms as there are simply too many non-canonical choices. We also do not want to have to write out all morphisms in explicit detail, as the example of $\delta_{\texttt{commutative}}$ shows.

2.4 Tiny Theories

Previous experiments [14] convinced us that for scaling it was best to use *tiny* theories, i.e. adding a single concept in at a time. This is useful both for defining pure signatures (presentations with no axioms) as well as when defining properties such as commutativity. First, one defines the smallest typing context in which the property can be stated, then use combine (and appropriate renames) to insert it into larger theories. Figure 2 is representative of the actual kinds of graphs we get when using *tiny theories* systematically.

Note that we did not need general morphisms: the ones we needed were combinations of morphisms induced by extensions and renamings.

Nevertheless, whether tiny theories are used or not, when we want to work with a `Group`, we do not really care about the details of how the library developers constructed it. In particular, we might never have heard of `Monoid` or `Semigroup`, never mind `Magma`, yet the user should still be able to use and understand `Group`. This is not the case for other approaches, such as the hierarchy in Agda [1], Lean [25] or Coq [34,72]. Furthermore, if library developers change their mind, this should not cause any downstream problems. When one of the authors added `Magma` to the Algebra hierarchy of Agda, this caused incompatibilities between versions of the standard library. Similar problems happened in the Haskell ecosystem when the `Functor-Applicative-Monad` proposal [80] was adopted, which introduced `Applicative` in between `Functor` and `Monad`.

2.5 Induced Requirements

We assemble the issues documented above into a set of requirements for combinators.

First, we need to have a setting in which extensions, renamings and combinations make sense. We will need to pay close attention to names, both to allow user control of names and prevent accidental collisions. To be able to maintain human-readable names for all concepts, we will put the burden on the *library developers* to come up with a reasonable naming scheme, rather than to necessarily push that issue onto end users. Symbol choice carries a lot of *intentional* and *contextual* information which is commonly used in mathematical practice.

As different kinds of morphisms induced by name permutations, extenstions, etc have different properties, tracking these can greatly simplify their use and re-use. We need a lightweight syntax where easily inferred information can be omitted in common usage. For example, we earlier tried to provide an explicit base theory over which `combine` can work; this works whenever there is a unique injection from the base theory into the extensions. While this is frequently the case, this is not always so. Thus we need to be able to be explicit about which injection we mean. A common workaround [65,66] is to use long and/or qualified names that "induce" injections more often, but this still does not scale. *Much worse*, this has the effect of leaking the details of how a presentation was constructed into the names of the symbols of the new presentation, which prevents later refinements, as all these names would change. As far as we can tell, any automatic naming policy will suffer from this problem, which is why we insist on having the library developers explicitly deal with name clashes. In practice few renamings are needed, so if we have special syntax for "no renaming needed" consistent with the renaming case, the resulting system should not be too burdensome to use.

We can summarize our requirements as follows:

- Names and symbols associated to concepts should all be user-disambiguated,

- It should be possible to use the usual mathematical symbols for concepts,
- Concepts should be defined once in the minimal context necessary for their definition and transported to their use site,
- Identical concepts with different names (because of change of context) should be automatically recognized as "the same" concept,
- The choices made by library developers of the means of construction of theory presentations should be invisible to end-users,
- Changing the means of construction of a theory should remain invisible,
- Constructions should be re-usable (whenever possible),
- Meta-properties of morphisms should not be forgotten by the system.

3 Basic Semantics

We present needed definitions from dependent type theory and start to develop the categorical semantics of theory presentation combinators. The principal reason to use category theory is that previous work on categorical semantics of dependent type theory has essentially established that the structure we need for our combinators to "work" is already present there. More precisely, that the category of contexts has enough pullbacks and supports suitable fibrations [48,75].

Recall the basic observation first made in Section 2.3, that theory presentations and contexts of a dependent type theory are the same: a list of symbol-type pairs, where the types of latter symbols may depend on earlier symbols to be well-defined. In this section, we will use "presentation" and "context" interchangeably.

Presentations depend on a background dependent type theory but are agnostic as to many of the internal details of that theory. We outline what we require, which is standard for such type theories:

- An infinite set of variable names $\mathbb{V}$.
- A typing judgement for terms s of type σ in a context Γ which we write $\Gamma \vdash s : \sigma$.
- A kinding judgement for types σ of kind κ in a context Γ which we write $\Gamma \vdash \sigma : \kappa : \Box$.
- A definitional equality (a.k.a. convertibility) judgement of terms s_1 of type σ_1 and s_2 of type σ_2 in a context Γ, which we write $\Gamma \vdash s_1 : \sigma_1 \equiv s_2 : \sigma_2$. We will write $\Gamma \vdash s_1 \equiv s_2 : \sigma$ to denote $\Gamma \vdash s_1 : \sigma \equiv s_2 : \sigma$.
- A notion of substitution on terms. Given a list of variable assignments $\wr x_i \mapsto s_i \int_{i<n}$ and an expression e we write $e[x_i \mapsto s_i]_{i<n}$ for the term e after simultaneous substitution of variables $\{x_i\}_{i<n}$ by the corresponding term in the assignment.

We use the meta-variable v and w to denote lists of assignments and its application to a term e by $e[v]$.

3.1 Theory Presentations

A theory presentation is a well-typed list of declarations. Figure 4 gives the formation rules. We use $|\Gamma|$ to denote the set of variables names of a well-formed context Γ:

$$|\varnothing| = \varnothing \qquad |\Gamma\ ;\ x : \sigma| = |\Gamma| \cup \{x\}$$

$$\frac{}{\varnothing\ \texttt{ctx}} \qquad \frac{\Gamma\ \texttt{ctx} \qquad \sigma \notin |\Gamma| \qquad \Gamma \vdash \kappa : \square}{(\Gamma\ ;\ \sigma : \kappa)\ \texttt{ctx}} \qquad \frac{\Gamma\ \texttt{ctx} \qquad x \notin |\Gamma| \qquad \Gamma \vdash \sigma : \kappa : \square}{(\Gamma\ ;\ x : \sigma)\ \texttt{ctx}}$$

Fig. 4 Formation rules for contexts

3.2 Morphisms (Views)

As outlined in Section 2.2, a morphism from a theory presentation Γ to a theory presentation Δ is an assignment of well-typed Δ-expression to each declaration of Γ. The assigments transport well-typed terms in the context Γ to well-typed terms in Δ, by substitution. Figure 5 gives the formation rules.

$$\frac{\Delta\ \texttt{ctx}}{[] : \varnothing \to \Delta} \qquad \frac{(\Gamma\ ;\ x : \sigma)\ \texttt{ctx} \qquad [v] : \Gamma \to \Delta \qquad \Delta \vdash r : \sigma[v]}{[v, x \mapsto r] : (\Gamma\ ;\ x : \sigma) \to \Delta}$$

Fig. 5 Formation rules for morphisms (substitutions).

There is a subtle but important distinction between assignments $\wr v \int$ and morphisms, $[v] : \Gamma \to \Delta$: morphisms are typed and thus Γ and Δ are integral to the definition, while the same assignment may occur in different morphisms.

Since most morphisms allow us to "view" one presentation inside another, we will frequently refer to the morphisms as *views*.

3.2.1 Inclusions, Renames and Embeddings

In Section 2.1.1, we define a construction to extend a presentation with new fields and in Section 2.1.2, renamings were defined. Section 2.2, details how these correspond to morphisms acting solely on names, being respectively an inclusion and a bijection.

We thus define an *embedding* to be a special kind of morphism, which we denote

$$\tilde{\pi} : \Gamma \to \Delta$$

where we require that $\tilde{\pi}$ is the morphism induced by $\pi : \mathbb{V} \to \mathbb{V}$ where:

- $\pi : \mathbb{V} \to \mathbb{V}$ has *finite support* (i.e. outside of a finite subset of $\mathbb{V}$, $\pi(v) = v$).
- π is a bijection,
- π^{-1} restricts to an injective function $\pi^{-1} : |\Delta| \to |\Gamma|$.

The $\tilde{}$ on $\tilde{\pi}$ is a reminder that while π is function on names, $\tilde{\pi}$ is a morphism with special properties. Note that inclusions are embeddings, with empty support [64]. Embeddings thus include both inclusions and renamings.

We make a further, important assumption about the host type system: the types involved must be *intensionally* equal, after the renaming has been applied. These morphisms are "extremely syntactic" and require no normalization beyond α-equivalence.

As we have mentioned previously, `Ring` is an extension of `Monoid` in two different ways and hence both embeddings cannot be inclusions. Inclusions will not be special in our formalism, other than being a case of an embedding. We draw attention to them here as many other systems make inclusions play a very special role. As we will see later, it is instead *display maps* which here hold that special role.

3.2.2 Composition

Given two morphisms $[v] : \Gamma \to \Delta$ and $[w] : \Delta \to \Phi$, we can compose them $[v];[w] : \Gamma \to \Phi$. If $v \triangleq [a \mapsto r_a]_{a \in |\Gamma|}$ then the composite morphism is

$$[v];[w] \triangleq [a \mapsto r_a[w]]_{a \in |\Gamma|}$$

That this gives a well-defined notion of composition and is associative is standard [21, 48, 75].

3.2.3 Equivalence of Morphisms

Two morphisms with the same domain and codomain, $[u], [v] : \Gamma \to \Delta$ are *equivalent* if $\Delta \vdash r_a : (\sigma_a[u]) \equiv s_a : (\sigma_a[v])$ where

$$\Gamma \triangleq [a : \sigma_a]_{a \in |\Gamma|}$$
$$u \triangleq [a := r_a]_{a \in |\Gamma|}$$
$$v \triangleq [a := s_a]_{a \in |\Gamma|}$$

We work in the setting of Setoid-enriched categories, as these are constructively simpler to deal with [46]. The above are used to define the equivalence relation on morphism in the definition of our category.

3.2.4 The category of theory presentations

The preceeding gives the necessary ingredients to define the *category of theory presentations* $\mathbb{P}$, with theory presentations as objects and views as morphisms. The identity inclusions are the identity morphisms.

Note that in [20], we worked with $\mathbb{C} = \mathbb{P}^{op}$, which is traditionally called the *category of contexts*, and is more often used in categorical logic [21, 48, 75, 63]. In our setting and as is common in the context of specifications (see for example [11, 70, 24] amongst many others), we prefer to take our intuition from *textual inclusion* rather than *models*. Nevertheless, for the *semantics*, we too will use $\mathbb{C}$, as this not only simplifies certain arguments, it also makes our work easier to compare to that in categorical logic.

3.3 Combinators

Given theory presentations, embedding and views, we can can now define presentation and view combinators. In fact, all combinators in this section will end up working in tandem on presentations and views. They allow us to create new presentations/views from old ones, in a more convenient manner than building everything by hand.

The constructions (operational semantics) will be spelled out in full detail, and are directly implementable. In the next section, we will give them a categorical semantics; we make a few inline remarks here to help the reader understand why we choose a particular construction.

3.3.1 Renaming

Given a presentation Γ and an injective renaming function $\pi : |\Gamma| \to \mathbb{V}$ we can construct a new theory presentation Δ by renaming Γ's symbols: we will denote this action of π on Γ by $\pi \cdot \Gamma$. We also construct an embedding $\tilde{\pi} : \Gamma \to \pi \cdot \Gamma$ which provides a translation from Γ to the constructed presentation $\pi \cdot \Gamma$. For this construction as a whole, we use the notation

$$\mathfrak{R}(\Gamma, \pi : |\Gamma| \to \mathbb{V}) \triangleq \left\{ \begin{array}{l} \texttt{pres} = \pi \cdot \Gamma \\ \texttt{embed} = \tilde{\pi} : \Gamma \to \pi \cdot \Gamma \end{array} \right\}$$

The rename function used in Section 2.1.2, produces the presentation `AdditiveCommMonoid` and the embedding $\tilde{\pi}$: `CommutativeMonoid` $\to$ `AdditiveCommMonoid`

3.3.2 Extend

Given a theory presentation Γ, a name a that does not occur in Γ and a well formed type σ of some kind κ, (i.e. $\Gamma \vdash \sigma : \kappa : \Box$) we can construct a new theory presentation $\Delta \triangleq \Gamma; a : \sigma$ and the embedding $\tilde{\mathsf{id}} : \Gamma \to \Delta$. More generally, given sequences of fresh names, types and kinds, $\{a_i\}_{i<n}$, $\{\sigma_i\}_{i<n}$ and $\{\kappa_i\}_{i<n}$ we can define a sequence of theory presentations $\Gamma_0 \triangleq \Gamma$ and $\Gamma_{i+1} \triangleq \Gamma_i; a_i : \sigma_i$ so long as $\Gamma_i \vdash \sigma_i : \kappa_i : \Box$. Given such a sequence we construct a new theory presentation $\Delta \triangleq \Gamma_n$ with the embedding $\tilde{\mathsf{id}} : \Gamma \to \Delta$.

As Δ is the concatenation of Γ with $\{a_i : \sigma_i : \kappa_i\}_{i<n}$, we will use $\Gamma \rtimes \Delta^+$ to denote the target of this view whenever the components of Δ^+ are clear from context. Δ^+ is *rarely a valid presentation*, as it usually depends on Γ. This is why we use an asymmetric symbol $\rtimes$. Nevertheless, there is an obvious extension of renamings to these presentation fragments, and we use the same notation for these.

Note that general embeddings $\tilde{\pi} : \Gamma \to \Delta$ as defined in §3.2.1 can be decomposed into a renaming composed with an $\rtimes$, in other words $\tilde{\pi} : \Gamma \to \Delta = \tilde{u}; \tilde{\mathsf{id}}$ where $\tilde{u} : \Gamma \to \pi \cdot \Gamma$ and $\tilde{\mathsf{id}} : \pi \cdot \Gamma \to \pi \cdot \Gamma \rtimes \Delta^+$. We will also write these as $\Gamma[u]_\Delta$ when we do not wish to focus on the pieces of the embedding.

Embeddings which are inclusions are traditionally called *display maps* in $\mathbb{C} = \mathbb{P}^{op}$ and our $\tilde{\mathsf{id}} : \Gamma \to (\Gamma; a : \sigma)$ in $\mathbb{P}$ is denoted by $\hat{a} : (\Gamma; a : \sigma) \twoheadrightarrow \Gamma$ in $\mathbb{C}$ [75] and δ_a in [48].

For notational convenience, we encode the construction above as an explicit function to a *record* containing two fields, `pres` (for presentation) and `embed` (for embedding).

$$\mathfrak{E}\left(\Gamma, \Delta^+\right) \triangleq \left\{ \begin{array}{l} \texttt{pres} = \Gamma \rtimes \Delta^+ \\ \texttt{embed} = \tilde{\mathsf{id}} : \Gamma \to \Gamma \rtimes \Delta^+ \end{array} \right\}$$

where $\Delta^+ = \{a_i : \sigma_i : \kappa_i\}_{i<n}$. Section 2.1.1 gives an example of an extension with Γ = `Monoid` and Δ^+ being the declaration of the `commutative` axiom.

3.3.3 Combine

To combine two embeddings $[u_\Delta] : \Gamma \to \Delta$ and $[u_\Phi] : \Gamma \to \Phi$ to give a presentation Ξ, we need to make sure the results agree on Γ (to avoid cases like the one in Figure 3). To insure this, we ask that two injective renaming functions $\pi_\Delta : |\Delta| \to \mathbb{V}$ and $\pi_\Phi : |\Phi| \to \mathbb{V}$ satisfying

$$\pi_\Delta(x) = \pi_\Phi(y) \Leftrightarrow \exists z \in |\Gamma| \,.\, x = z[u_\Delta] \wedge y = z[u_\Phi] \tag{3}$$

are also provided. π_Δ and π_Φ will be used to give a unique name to the components of Γ, for example the carrier U in Figure 3.

Suppose that the two embeddings decompose as $\Delta = \Gamma[u_\Delta] \rtimes \Delta^+$ and $\Phi = \Gamma[u_\Phi] \rtimes \Phi^+$. Denote by u_Δ the action on $|\Gamma|$ of $[u_\Delta] : \Gamma \to \Delta$, and by u_Φ the action on $|\Gamma|$ of $[u_\Phi] : \Gamma \to \Phi$. Define

$$\Xi \triangleq \Xi_0 \rtimes (\Xi_\Delta \cup \Xi_\Phi)$$

where

$$\begin{aligned} \Xi_0 &\triangleq (u_\Delta ; \pi_\Delta) \cdot \Gamma \\ \Xi_\Delta &\triangleq \pi_\Delta \cdot \Delta^+ \\ \Xi_\Phi &\triangleq \pi_\Phi \cdot \Phi^+ \end{aligned}$$

Condition 3 is defined to ensure that $\Xi_0 \equiv (u_\Phi ; \pi_\Phi) \cdot \Gamma$ is also true. Similarly, by construction, $\Xi_0 \rtimes (\Xi_\Delta \rtimes \Xi_\Phi)$ is equivalent to $\Xi_0 \rtimes (\Xi_\Phi \rtimes \Xi_\Delta)$; we denote this equivalence class[3] of views by $\Xi_0 \rtimes (\Xi_\Delta \cup \Xi_\Phi)$.

The combination operation also provides embeddings $[v_\Delta] : \Delta \to \Xi$ and $[v_\Phi] : \Phi \to \Xi$ where $[v_\Delta] \triangleq \tilde{\pi}_\Delta$ and $[v_\Phi] \triangleq \tilde{\pi}_\Phi$. A calculation shows that $[u_\Delta];[v_\Delta]$ is equal to $[u_\Phi];[v_\Phi]$ (and not just equivalent); we denote this joint morphism $[uv] : \Gamma \to \Xi$. Furthermore, `combine` provides a set of mediating views from the constructed theory presentation Ξ. Suppose we are given views $[w_\Delta] : \Delta \to \Omega$ and $[w_\Phi] : \Phi \to \Omega$ such that the the composed views $[u_\Delta];[w_\Delta] : \Gamma \to \Omega$ and $[u_\Phi];[w_\Phi] : \Gamma \to \Omega$ are equivalent. We can combine $[w_\Delta]$ and $[w_\Phi]$ into a mediating view $[w_\Xi] : \Xi \to \Omega$ where

$$[w_\Xi] \triangleq [\pi_\Delta(x) := x[w_\Delta]]_{x \in |\Delta|} \cup [\pi_\Phi(y) := y[w_\Phi]]_{y \in |\Phi|}.$$

This union is well defined since if $\pi_\Delta(x) = \pi_\Phi(y)$ then there exists z such that $x = z[u_\Delta]$ and $y = z[u_\Phi]$, in which case $x[w_\Delta] = z[u_\Delta][w_\Delta]$ and $y[w_\Phi] = z[u_\Phi][w_\Phi]$ are equivalent since by assumption $[u_\Delta];[w_\Delta]$ and $[u_\Phi];[w_\Phi]$ are equivalent. It is also worthwhile noticing that this construction is symmetric in Δ and Φ.

For this construction, we use the following notation, where we use the symbols as defined above (omitting type information for notational clarity), and λ is from the meta-theory.

$$\mathfrak{C}(u_\Delta, u_\Phi, \pi_\Delta, \pi_\Phi) \triangleq \begin{Bmatrix} \texttt{pres} = \Xi_0 \rtimes (\Xi_\Delta \cup \Xi_\Phi) \\ \texttt{embed}_\Delta = [v_\Delta] : \Delta \to \Xi \\ \texttt{embed}_\Phi = [v_\Phi] : \Phi \to \Xi \\ \texttt{diag} = [uv] : \Gamma \to \Xi \\ \texttt{mediate} = \lambda\ w_\Delta\ w_\Phi\ .\ w_\Xi \end{Bmatrix}$$

The attentive reader will have noticed that we have painstakingly constructed an explicit *pushout* in $\mathbb{P}$. There are two reasons to do this: first, we need to be this explicit if we wish to be able to implement such an operation. And second, we do not want an arbitrary pushout, because we do not wish to work up to isomorphism as that would "mess up" the names. This is why we need user-provided injective renamings π_Δ and π_Φ to deal with potential name clashes. If we worked up to isomorphism, these renamings would not be needed, as they can

[3] In practice, theory presentations are rendered (printed, serialized) using a topological sort where ties are broken alphabetically, so as to be construction-order indedepent.

always be manufactured by the system but then these are no longer necessarily related to the users' names. Alternatively, if we use *long names* based on the (names of the) views, the method used to construct the presentations and views "leaks" into the names of the results, which we also consider undesirable.

Example Consider combining the two embeddings $[u_\Delta] : \texttt{Magma} \to \texttt{Semigroup} = \tilde{\mathsf{id}}$ and $[u_\Phi] : \texttt{Magma} \to \texttt{AddMagma} = [\texttt{U} \mapsto \texttt{U}, \circ \mapsto +]$; in the above, this makes $\Gamma = \texttt{Magma}$, $\Delta = \texttt{Semigroup}$ and $\Phi = \texttt{AddMagma}$. Choosing $\pi_\Delta = \pi_\phi = \mathsf{id}$ does not satisfy condition 3. The problem is that the two embeddings $[u_\Delta]$ and $[u_\Phi]$ disagree on the name of the binary operation. Thus the user must provide a renaming; for example, the user might choose + and define `AddSemigroup`, by using $\pi_\Delta = [\circ \mapsto +, \texttt{associativity\_}\circ \mapsto \texttt{associativity\_}+]$; π_ϕ can remain id. (We would prefer for the user to only need to specify $[\circ \to +]$ and for the system to infer $[\texttt{associativity\_}\circ \mapsto \texttt{associativity\_}+]$ but we leave this for future work). The algorithm then proceeds to compute the expected `AddSemigroup` and accompanying embeddings.

3.3.4 Mixin

Given a view $[u_\Delta] : \Gamma \to \Delta$, an embedding $[u_\Phi] : \Gamma \to \Phi$ and two disjoint injective renaming functions $\pi_\Delta : |\Delta| \to \mathbb{V}$ and $\pi_\Phi : |\Phi| \to \mathbb{V}$, where the embedding Φ decomposes as $\Phi = \Gamma[u_\Phi] \rtimes \Phi^+$, we can mixin the view into the embedding, constructing a new theory presentation Ξ. We define $\Xi \triangleq \Xi_1 \rtimes \Xi_2$ where we need a new renaming π'_Φ:

$$\pi'_{\Phi^+}(y) \triangleq \begin{cases} z[u_\Delta][x \mapsto \pi_\Delta(x)]_{x \in |\Delta|} & \text{when there is a } z \in |\Gamma| \text{ such that } z[u_\Phi] = y \\ \pi_{\Phi^+}(y) & \text{when } y \in \left|\Phi^+\right| \end{cases}$$
$$\Xi_1 \triangleq \pi_\Delta \cdot \Delta$$
$$\Xi_2 \triangleq \pi'_\Phi \cdot \Phi^+$$

The mixin also provides an embedding $[v_\Delta] : \Delta \to \Xi$ and a view $[v_\Phi] : \Phi \to \Xi$, defined as

$$[v_\Delta] \triangleq \tilde{\pi}_\Delta$$
$$[v_\Phi] \triangleq \tilde{\pi}'_\Phi$$

By definition of embedding, there is no $z \in |\Gamma|$ that is mapped into Φ^+ by $[u_\Phi]$. The definition of π'_Φ is arranged such that $[u_\Delta];[v_\Delta]$ is equal to $[u_\Phi];[v_\Phi]$ (and not just equivalent); so we can denote this joint morphism by $[uv] : \Gamma \to \Xi$. In other words, in a mixin, by only allowing renaming of the *new components* in Φ^+, we insure commutativity *on the nose* rather than just up to isomorphism.

Mixins also provide a set of mediating views from the constructed theory presentation Ξ. Suppose we are given the views $[w_\Delta] : \Delta \to \Omega$ and$[w_\Phi] : \Phi \to \Omega$ such that the composed views $[u_\Delta];[w_\Delta] : \Gamma \to \Omega$ and $[u_\Phi];[w_\Phi] : \Gamma \to \Omega$ are equivalent. We can combine $[w_\Delta]$ and $[w_\Phi]$ into the mediating view $[w_\Xi] : \Xi \to \Omega$ defined as

$$[w_\Xi] \triangleq [\pi_\Delta(x) \mapsto x[w_\Delta]]_{x \in |\Delta|} \cup [\pi'_\Phi(y) \mapsto y[w_\Phi]]_{y \in |\Phi^+|}.$$

For mixin, again using the symbols as above, we denote the construction results as

$$\mathfrak{M}\left(u_\Delta, u_\Phi, \pi_\Delta, \pi_\Phi\right) \triangleq \left\{ \begin{array}{r} \texttt{pres} = \Xi_1 \rtimes \Xi_2 \\ \texttt{embed}_\Delta = \left[v_\Delta\right] : \Delta \to \Xi \\ \texttt{view}_\Phi = \left[v_\Phi\right] : \Phi \to \Xi \\ \texttt{diag} = \left[uv\right] : \Gamma \to \Xi \\ \texttt{mediate} = \lambda\ w_\Delta\ w_\Phi\ .\ w_\Xi \end{array} \right\}$$

Symbolically the above is very similar to what was done in combine and indeed we are constructing all of the data for a specific pushout. However in this case the results are not symmetric, as seen from the details of the construction of Ξ_1 and Ξ_2, which stems from the fact that in this case $\left[v_\Phi\right]$ is an arbitrary view rather than an embedding. The `Flip` view of Section 2.2 is an example.

4 Categorical Semantics

We delve more deeply into the semantics of `combine` and `mixin`. The categorical interpretation (in $\mathbb{C}$) of `combine` is unsurprising: pullback. But `mixin` is more complex: it is a Cartesian lifting in a suitable fibration. But we also obtain that our semantics is *total*, even for mixin. Key is that the algorithm for `mixin` in the previous section produces a unique *syntactic* representation of the results. Many combinators with well-defined categorical semantics (such as unrestricted `mixin`) do not have this property: while they have models, these models cannot be written down.

At first glance, the definitions of `combine` and `mixin` may appear ad hoc and overly complicated. This is because, in practice, the renaming functions π_Δ and π_Φ are frequently the *identity*. The main reason for this is that mathematical vernacular uses a lot of rigid conventions, such as usually naming an associative, commutative, invertible operator which possesses a unit +, the unit is named 0, backward composition is $\circ$, forward composition is ;, and so on. But the usual notation of lattices is different than that of semirings, even though they share a similar ancestry, so that renamings are clearly necessary at some point.

While our primary interest is in theory presentations, the bulk of the categorical work in this area has been done on the category of contexts, which is the opposite category. To be consistent with the existing literature, we will give our categorical semantics in terms of $\mathbb{C} = \mathbb{P}^{op}$. Thus if $\left[v\right] : \Gamma \to \Delta$ is a view, then a corresponding morphism, v, exists from context Δ to context Γ. We will write such morphisms as $v : \Delta \twoheadrightarrow \Gamma$ when we are considering the category of contexts, with composition as before.

4.1 Semantics

The category of contexts forms the base category for a fibration. The fibered category $\mathbb{E}$ is the category of context extensions. The objects of $\mathbb{E}$ are embeddings of contexts. We write such objects as $u : \Delta \rightarrowtail \Gamma$ where Γ is the base and Δ is the extended context. The notation is to remind the reader that the morphisms are display maps (i.e. that Γ is a strict prefix of Δ).

A morphism between two embeddings is a pair of views forming a commutative square with the embeddings. Thus given embeddings $u_2 : \Delta_2 \rightarrowtail \Gamma_2$ and $u_1 : \Delta_1 \rightarrowtail \Gamma_1$, a morphism between these consists of two morphisms $v_\Delta : \Delta_2 \twoheadrightarrow \Delta_1$ and $v_\Gamma : \Gamma_2 \twoheadrightarrow \Gamma_1$ from $\mathbb{C}$ such that $v_\Gamma ; u_1 = u_2 ; v_\Delta : \Gamma_2 \twoheadrightarrow \Delta_1$. When we need to be very precise, we write such a morphism as

$$\begin{array}{ccc} & \overset{v_\Delta}{} & \\ \Delta_2 & \twoheadrightarrow & \Delta_1 \\ u_2 \downarrow & & \downarrow u_1 \\ \Gamma_2 & \twoheadrightarrow & \Gamma_1 \\ & \underset{v_\Gamma}{} & \end{array}$$

. We will write $v_\Gamma^\Delta : u_2 \Mapsto u_1$ whenever the rest of the information can be inferred from context. When given a specific morphism in $\mathbb{E}$, we will use the notation $e^{\Mapsto}$.

A fibration of $\mathbb{E}$ over $\mathbb{C}$ is defined by giving a suitable functor from $\mathbb{E}$ to $\mathbb{C}$. Our "base" functor sends an embedding $e : \Delta \rightarrowtail \Gamma$ to Γ and sends a morphism $v_\Gamma^\Delta : u_2 \Mapsto u_1$ to its base morphism $v_\Gamma : \Gamma_2 \twoheadrightarrow \Gamma_1$.

Theorem 1 *This base fibration is a Cartesian fibration.*

This theorem, in slightly different form, can be found in [48] and [75]. We give a full proof here because we want to make the link with our `mixin` construction explicit. We use the results of §3.3 directly.

Proof Suppose $u_\Delta : \Delta \twoheadrightarrow \Gamma$ is a morphism in $\mathbb{C}$, and $u_\Phi : \Phi \rightarrowtail \Gamma$ is an object of $\mathbb{E}$ in the fiber of Γ (i.e. an embedding). We need to construct a Cartesian lifting of u_Δ, which is a Cartesian morphism of $\mathbb{E}$ over u_Δ. The components of the `mixin` construction are exactly the ingredients we need to create this Cartesian lifting. Let $\pi_\Delta : |\Delta| \to \mathbb{V}$ and $\pi'_\Phi : |\Phi^+| \to \mathbb{V}$ be two disjoint injective renaming functions. Note that such π_Δ and π_Φ always exist because $\mathbb{V}$ is infinite while $|\Delta|$ and $|\Phi^+|$ are finite. Let

$$\mathfrak{M}(u_\Delta, u_\Phi, \pi_\Delta, \pi_\Phi) \triangleq \left\{ \begin{array}{r} \texttt{pres} = \Xi \\ \texttt{embed}_\Delta = v_\Delta : \Xi \to \Delta \\ \texttt{view}_\Phi = v_\Phi : \Xi \to \Phi \\ \texttt{diag} = uv : \Xi \to \Phi \\ \texttt{mediate} = \lambda\ w_\Delta\ w_\Phi\ .\ w_\Xi \end{array} \right\}$$

where we recall that we are now working in $\mathbb{C}$, the opposite of $\mathbb{P}$ and thus the direction of the morphisms is flipped. Then $e^{\Mapsto} \triangleq$

$$\begin{array}{ccc} & \overset{v_\Phi}{} & \\ \Xi & \twoheadrightarrow & \Phi \\ v_\Delta \downarrow & & \downarrow u_\Phi \\ \Delta & \twoheadrightarrow & \Gamma \\ & \underset{u_\Delta}{} & \end{array}$$

is a morphism of $\mathbb{E}$ which is a Cartesian lift of u_Δ.

Firstly, to see that $e^{\Mapsto}$ is in fact a morphism of $\mathbb{E}$, we note that $[v_\Delta] : \Delta \to \Xi$ is an embedding, so $v_\Delta : \Xi \rightarrowtail \Delta$ is an object of $\mathbb{E}$. Next we need to show that $v_\Delta ; u_\Delta = v_\Phi ; u_\Phi$. Let $z \in |\Gamma|$. Then $z[u_\Delta][v_\Delta] = z[u_\Delta][x \mapsto \pi_\Delta(x)]$ by definition of v_Δ. On the other hand, $z[u_\Phi][v_\Phi] = z[u_\Phi][y \mapsto \pi'_\Phi(y)]_{y \in |\Phi|}$ by definition of v_Φ. However, $z[u_\Phi]$ is a variable since u_Φ is an embedding and by definition $\pi'_\Phi(z[u_\Phi]) = z[u_\Delta][x \mapsto \pi_\Delta(x)]_{x \in |\Delta|}$ so that we have $z[u_\Delta][v_\Delta] = z[u_\Phi][v_\Phi]$ as required.

Secondly we need to see that $e^{\Mapsto}$ is a Cartesian lift of $u_\Delta : \Delta \twoheadrightarrow \Gamma$. We need to show that for any morphism $f^{\Mapsto} \triangleq$

$$\begin{array}{ccc} & \overset{w_\Phi}{} & \\ \Omega & \twoheadrightarrow & \Phi \\ w_\Psi \downarrow & & \downarrow u_\Phi \\ \Psi & \twoheadrightarrow & \Gamma \\ & \underset{w_\Gamma}{} & \end{array}$$

from $\mathbb{E}$ and any arrow $w_0 : \Psi \twoheadrightarrow \Delta$ from $\mathbb{C}$ such

that $w_\Gamma = w_0; u_\Delta : \Psi \twoheadrightarrow \Gamma$, there is a unique mediating morphism $h^{\models} \triangleq w_\Psi$ from the square

$$\begin{array}{ccc} \Omega & \xrightarrow{w_\Xi} & \Xi \\ \downarrow & & \downarrow v_\Delta \\ \Psi & \xrightarrow[w_0]{} & \Delta \end{array}$$

$\mathbb{E}$ such that

$$h^{\models} \,;\, e^{\models} = f^{\models} \tag{4}$$

To show that such an $h^{\models}$ exists, we only need to construct $w_\Xi : \Omega \twoheadrightarrow \Xi$ and show that it has the required properties. We will show that the mediating morphism w from the `mixin` construction given $w_\Phi : \Omega \twoheadrightarrow \Phi$ and $w_\Delta \triangleq w_\Psi; w_0 : \Omega \twoheadrightarrow \Delta$ is the required morphism.

First we note that $w_\Delta; u_\Delta = w_\Phi; u_\Phi$ as required by the `mixin` construction for the mediating morphism since $w_\Delta; u_\Delta = w_\Psi; w_0; u_\Delta = w_\Xi; v_\Phi; u_\Phi = w_\Phi; u_\Phi$ by chasing around the diagram of the equality $h^{\models} \,;\, e^{\models} = f^{\models}$. Now taking $w_\Xi \triangleq w$ we need to show that $h^{\models}$ is a well defined morphism in $\mathbb{E}$ by showing it forms a commutative square. Suppose $x \in |\Delta|$. Then $x[v_\Delta][w_\Xi] = \pi_\Delta(x)[w_\Xi] = x[w_\Delta] = x[w_0][w_\Psi]$ as required. Next we need to show that equation (4) holds. It suffices to show that $w_\Xi; v_\Phi = w_\Phi$ since it is already required that $w_0; u_\Delta = w_\Gamma$. Suppose $y \in |\Phi|$. There are two possiblities, either $y = z[u_\Phi]$ for some $z \in |\Gamma|$, or $y \in |\Phi^+|$ where $\Phi = \Gamma[u_\Phi] \rtimes \Phi^+$. If $y \in |\Phi^+|$ then $y[v_\Phi][w_\Xi] = \pi_{\Phi^+}(y)[w_\Xi] = y[w_\Phi]$ as requried. In case $y = z[u_\Phi]$, then $y[v_\Phi][w_\Xi] = z[u_\Phi][v_\Phi][w_\Xi] = z[u_\Delta][v_\Delta][w_\Xi] = z[u_\Delta][w_0][w_\Psi] = z[w_\Gamma][w_\Psi] = z[u_\Phi][w_\Phi] = y[w_\Phi]$ as requiried.

Lastly we need to show that the mediating morphism $h^{\models}$ is the unique morphism satisfying equation (4). Let $j^{\models}$ be another morphism of $\mathbb{E}$, where $j^{\models}$ must have the same shape as $h^{\models}$ but with w_Ξ replaced with w'_Ξ. Suppose that

$$j^{\models} \,;\, f^{\models} = e^{\models}$$

We need to show that $w'_\Xi = w_\Xi$. Suppose $z \in |\Xi|$. There are two possiblities. Either $z = x[v_\Delta]$ for some $x \in |\Delta|$ or $z = y[v_\Phi]$ for some $y \in |\Phi^+|$. Suppose $z = x[v_\Delta]$. Then $z[w'_\Xi] = x[v_\Delta][w'_\Xi] = x[w_0][w_\Psi] = x[v_\Delta][w_\Xi] = z[w_\Xi]$ as required. On the other hand, suppose $z = y[v_\Phi]$. Then $z[w'_\Xi] = y[v_\Phi][w'_\Xi] = y[w_\Phi] = y[v_\Phi][w_\Xi] = z[w_\Xi]$ as required. So $w'_\Xi = w_\Xi$ and hence $j^{\models} = h^{\models}$, as required. □

The above proof illustrates that the `mixin` operation is characterized by the properties of a Cartesian lifting in the fibration of embeddings. Notice that a Cartesian lift is only characterised up to isomorphism. Thus there are potentially many isomorphic choices for a Cartesian lift and hence there are many possible choices for how to `mixin` an embedding into a view. This is the underlying reason why the `mixin` construction requires a pair of renaming functions. The renaming functions pick out a particular choice of mixin from the many possibilities. This ability to specify which `mixin` to construct is quite important as one cannot simply define a `mixin` to be "the" Cartesian lift, since "the" Cartesian lift is only defined up to isomorphism. It is important to remember that for *user syntax*, we cannot work up to isomorphism!

Next we will see that `combine` is a special case of `mixin`.

Theorem 2 *Given two embeddings $u_\Delta : \Gamma \hookrightarrow \Delta$ and $u_\Phi : \Gamma \hookrightarrow \Phi$ and renaming functions $\pi_\Delta : |\Delta| \to \mathbb{V}$ and $\pi_\Phi : |\Phi| \to \mathbb{V}$ sastifiying the requirement of the* `combine` *construction, then*

$$\mathfrak{M}(u_\Delta, u_\Phi, \pi_\Delta, \pi_{\Phi^+}) = \mathfrak{C}(u_\Delta, u_\Phi, \pi_\Delta, \pi_\Phi) \tag{5}$$

where $\Phi = \Gamma[u_\Phi] \rtimes \Phi^+$ and $\pi_{\Phi^+} = [x \mapsto \pi_\Phi]_{x \in |\Phi^+|}$, and equation 5 is interpreted componentwise.

Proof Suppose that

$$\mathfrak{C}\left(u_\Delta, u_\Phi, \pi_\Delta, \pi_\Phi\right) = \left\{\begin{array}{r} \texttt{pres} = \Xi_0 \rtimes \left(\Xi_\Delta \cup \Xi_\Phi\right) \\ \texttt{embed}_\Delta = v_\Delta : \Xi \to \Delta \\ \texttt{embed}_\Phi = v_\Phi : \Xi \to \Phi \\ \texttt{diag} = uv : \Xi \to \Gamma \\ \texttt{mediate} = \lambda\ w_\Delta\ w_\Phi\ .\ w_\Xi \end{array}\right\}$$

and

$$\mathfrak{M}\left(u_\Delta, u_\Phi, \pi_\Delta, \pi_{\Phi^+}\right) = \left\{\begin{array}{r} \texttt{pres} = \Xi' \\ \texttt{embed}_\Delta = v'_\Delta : \Xi' \to \Delta \\ \texttt{view}_\Phi = v'_\Phi : \Xi' \to \Phi \\ \texttt{diag} = uv' : \Xi' \to \Gamma \\ \texttt{mediate} = \lambda\ w_\Delta\ w_\Phi\ .\ w_{\Xi'} \end{array}\right\}$$

Recall that $\Xi = \Xi_0 \rtimes \left(\Xi_\Delta \cup \Xi_\Phi\right) = \Xi_0 \rtimes \left(\Xi_\Delta\right) \rtimes \Xi_\Phi$ where $\Xi_0 \triangleq \Gamma\left[z \mapsto \pi_\Delta\left(z\left[v_\Delta\right]\right)\right]_{z \in |\Gamma|}$, $\Xi_\Delta \triangleq \Delta^+\left[x \mapsto \pi_\Delta\left(x\right)\right]_{x \in |\Delta|}$ and $\Xi_\Phi \triangleq \Phi^+\left[y \mapsto \pi_\Phi\left(y\right)\right]_{y \in |\Phi|}$. In particular note that $\Xi_0 = \Gamma\left[v_\Delta\right]\left[z\left[v_\Delta\right] \mapsto \pi_\Delta\left(z\left[v_\Delta\right]\right)\right]_{z \in |\Gamma|}$. Since $\Delta = \Gamma\left[v_\Delta\right] \rtimes \Delta^+$, we have that

$$\begin{aligned} \Xi_0 \rtimes \Xi_\Delta &= \Gamma\left[v_\Delta\right]\left[z\left[v_\Delta\right] \mapsto \pi_\Delta\left(z\left[v_\Delta\right]\right)\right]_{z \in |\Gamma|} \rtimes \Delta^+\left[x \mapsto \pi_\Delta\left(x\right)\right]_{x \in |\Delta|} \\ &= \left(\Gamma\left[v_\Delta\right] \rtimes \Delta^+\right)\left[x \mapsto \pi_\Delta\left(x\right)\right]_{x \in |\Delta|} \\ &= \Delta\left[x \mapsto \pi_\Delta\left(x\right)\right]_{x \in |\Delta|} \end{aligned}$$

Recall also that $\Xi' = \Xi'_1 \rtimes \Xi'_2$ where $\Xi'_1 \triangleq \Delta\left[x \mapsto \pi_\Delta\left(x\right)\right]_{x \in |\Delta|}$ and $\Xi'_2 \triangleq \Phi^+\left[y \mapsto \pi'_{\Phi^+}\left(y\right)\right]_{y \in |\Phi|}$. So we see that $\Xi'_1 = \Xi_0 \rtimes \Xi_\Delta$.

Next we show that $\pi'_\Phi = \pi_\Phi$. If $y \in |\Phi|$ then either $y \in \left|\Phi^+\right|$ or there is some $z \in \Gamma$ such that $y = z\left[v_\Phi\right]$. If $y \in \left|\Phi^+\right|$ then $\pi'_\Phi\left(y\right) = \pi_\Phi\left(y\right) = \pi_\Phi\left(y\right)$. If $y = z\left[v_\Phi\right]$, then $\pi'_\Phi\left(y\right) = z\left[u_\Delta\right]\left[x \mapsto \pi_\Delta\left(x\right)\right]_{x \in |\Delta|} = \pi_\Delta\left(z\left[u_\Delta\right]\right) = \pi_\Phi\left(z\left[u_\Phi\right]\right) = \pi_\Phi\left(y\right)$. Therefore $\Xi'_2 = \Xi_\Phi$ and hence $\Xi' = \Xi$.

Next we need to show that $v'_\Delta = v_\Delta$ and $v'_\Phi = v_\Phi$. First we see that v'_Δ and v_Δ are both defined to be $\left[x \mapsto \pi_\Delta\left(x\right)\right]_{x \in |\Delta|}$, so clearly they are equal. Next we see that $v_\Phi \triangleq \left[y \mapsto \pi_\Phi\left(y\right)\right]_{y \in |\Phi|}$ and $v'_\Phi \triangleq \left[y \mapsto \pi'_\Phi\left(y\right)\right]_{y \in |\Phi|}$ are equal because $\pi'_\Phi = \pi_\Phi$. This also gives that $uv = uv'$.

Lastly we show that the mediating morphism of the `combine` is the same as the mediating morphism of the `mixin`. Suppose we are given $w_\Delta : \Delta \to \Omega$ and $w_\Phi : \Phi \to \Omega$ such that $u_\Delta; w_\Delta = u_\Phi; w_\Phi : \Gamma \to \Omega$. To show that the mediating morphism produced by `combine`, $w_\Xi : \Xi \to \Omega$ is the same as the medating morphism produced by the `mixin`, it suffices to prove that the mediating morphism satifies the universal property of the Cartesian lift, since such a morphism is unique. Thus it suffices to show that $v_\Phi; w_\Xi = w_\Phi : \Phi \to \Omega$ and $v_\Delta; w_\Xi = w_\Delta : \Delta \to \Omega$. Let $y \in |\Phi|$. Then $y\left[v_\Phi\right]\left[w_\Xi\right] = \pi_\Phi\left(y\right)\left[w_\Xi\right] = y\left[w_\Phi\right]$. Let $x \in |\Delta|$. Then $x\left[v_\Delta\right]\left[w_\Xi\right] = \pi_\Delta\left(x\right)\left[w_\Xi\right] = x\left[w_\Delta\right]$ as required.

Combine is rather well-behaved. In particular,

Proposition 1 $\mathfrak{C}\left(u_\Delta, u_\Phi, \pi_\Delta, \pi_\Phi\right) = \mathfrak{C}\left(u_\Phi, u_\Delta, \pi_\Phi, \pi_\Delta\right)$*, i.e.* `combine` *is commutative.*

It turns out that `combine` also satisfies an appropriate notion of associativity. In other words, we can compute limits of cones of embeddings.

4.2 No Lifting Views over Views

Why do we restrict ourselves to the fibration of embeddings? Why not allow mixins of arbitrary views over arbitrary views? If such `mixin` s were allowed, then the notion of a Cartesian lifting reduces to that of pullback. But to demand that the category of contexts and views be closed under *all* pullbacks would require too much from our type theory: we would need to have all equalizers (as we already have all products). In particular, at the type level, this would force us to have subset types, which is something we are not willing to impose. Thus a restriction is needed and our proposed restriction of only mixing in embeddings into views appears to be practical. Taylor [75] is a good source of further reasons for the naturality of restricting to this case.

5 Presentation Combinators

We can now give a set of combinators that are well-suited to the semantics described above.

5.1 Grammar

In the definition of the grammar, we use A, B to denote theories and/or views, x and y to denote symbols, t for terms of the underlying type theory and l for (raw) contexts from the underlying type theory. Recall that we are parametric in the type theory, and so we do not give a grammar for types and terms, i.e. the exact syntax of l and t below.

$$
\begin{array}{rl}
\mathsf{tpc} ::= & \mathsf{Theory}\ \{l\} \\
\mid & \mathsf{extend}\ A\ \{l\} \\
\mid & \mathsf{rename}\ A\ \mathrm{r} \\
\mid & \mathsf{combine}\ A\ r_1\ B\ r_2 \\
\mid & \mathsf{mixin}\ A\ r_1\ B\ r_2 \\
\mid & \mathsf{view}\ A\ \mathsf{as}\ B\ \mathsf{via}\ \mathrm{v} \\
\mid & A\ ;\ B
\end{array}
\qquad
\begin{array}{rl}
\mathrm{r} ::= & \{\mathrm{ren}\} \\
\mathrm{v} ::= & \{\mathrm{assign}\} \\
\mathrm{ren} ::= & x\ \mathsf{to}\ y \\
\mid & \mathrm{ren},\ x\ \mathsf{to}\ y \\
\mathrm{assign} ::= & x\ \mathsf{to}\ t \\
\mid & \mathrm{assign},\ x\ \mathsf{to}\ t
\end{array}
$$

Informally, these forms correspond to an explicit theory, a theory extension, a renaming, combining two extensions, mixing in a view and an extension, explicit view, and sequencing views. The empty theory is $\mathsf{Theory}\ \{\}$.

What might be surprising is that we do not have a separate language for presentations and views. This is because our language does not have a single semantics in terms of presentations, embeddings or views but rather has *several* compatible semantics. In other words, our syntax will yield objects of $\mathbb{C}$, objects of $\mathbb{E}$ (i.e. embeddings) and morphisms of $\mathbb{C}$ (views).

The semantics is given by defining three partial maps, $[\![-]\!]_{\mathbb{B}} : \mathsf{tpc} \rightharpoonup |\mathbb{C}|$, $[\![-]\!]_{\mathbb{E}} : \mathsf{tpc} \rightharpoonup |\mathbb{E}|$, $[\![-]\!]_{\mathbb{B}^{\rightarrow}} : \mathsf{tpc} \rightharpoonup \mathrm{Hom}_{\mathbb{C}}$. We also use $[\![-]\!]_{\pi}$ for the straightforward semantics in $\mathbb{V} \to \mathbb{V}$ of a renaming. As we define $[\![-]\!]_{\mathbb{B}}$ and $[\![-]\!]_{\mathbb{E}}$ in terms of $[\![-]\!]_{\mathbb{B}^{\rightarrow}}$, we define the latter first.

Morphisms of $\mathbb{C}$ are views:

$$
\begin{aligned}
[\![-]\!]_{\mathbb{B}^\to} &: \mathsf{tpc} \rightharpoonup \mathrm{Hom}_{\mathbb{C}} \\
[\![\mathsf{Theory}\ \{l\}]\!]_{\mathbb{B}^\to} &= !_l : [\,] \to [\![l]\!]_{\mathbb{B}} \\
[\![\mathsf{extend}\ A\ \mathsf{by}\ \{l\}]\!]_{\mathbb{B}^\to} &= \mathfrak{E}\,([\![A]\!]_{\mathbb{B}}, l)\,\texttt{.embed} \\
[\![\mathsf{rename}\ A\ r]\!]_{\mathbb{B}^\to} &= \mathfrak{R}\,([\![A]\!]_{\mathbb{B}}, [\![r]\!]_{\pi})\,\texttt{.embed} \\
[\![\mathsf{combine}\ A_1\ r_1\ A_2\ r_2]\!]_{\mathbb{B}^\to} &= \mathfrak{C}\,([\![A_1]\!]_{\mathbb{E}}, [\![A_2]\!]_{\mathbb{E}}, [\![r_1]\!]_{\pi}, [\![r_2]\!]_{\pi})\,\texttt{.diag} \\
[\![\mathsf{mixin}\ A_1 r_1, A_2 r_2]\!]_{\mathbb{B}^\to} &= \mathfrak{M}\,([\![A_1]\!]_{\mathbb{B}^\to}, [\![A_2]\!]_{\mathbb{E}}, [\![r_1]\!]_{\pi}, [\![r_2]\!]_{\pi})\,\texttt{.diag} \\
[\![\mathsf{view}\ A\ \mathsf{as}\ B\ \mathsf{via}\ \mathrm{v}]\!]_{\mathbb{B}^\to} &= [v] : [\![A]\!]_{\mathbb{B}} \to [\![B]\!]_{\mathbb{B}} \\
[\![A; B]\!]_{\mathbb{B}^\to} &= [\![A]\!]_{\mathbb{B}^\to}; [\![B]\!]_{\mathbb{B}^\to}
\end{aligned}
$$

We then define

$$
\begin{aligned}
[\![-]\!]_{\mathbb{B}} &: \mathsf{tpc} \rightharpoonup |\mathbb{C}| \\
[\![\mathsf{view}\ A\ \mathsf{as}\ B\ \mathsf{via}\ \mathrm{v}]\!]_{\mathbb{B}} &= \bot \\
[\![X]\!]_{\mathbb{B}} &= \mathsf{cod}\ [\![X]\!]_{\mathbb{B}^\to}
\end{aligned}
$$

$$
\begin{aligned}
[\![-]\!]_{\mathbb{E}} &: \mathsf{tpc} \rightharpoonup |\mathbb{E}| \\
[\![\mathsf{mixin}\ A_1 r_1, A_2 r_2]\!]_{\mathbb{E}} &= \bot \\
[\![\mathsf{view}\ A\ \mathsf{as}\ B\ \mathsf{via}\ \mathrm{v}]\!]_{\mathbb{E}} &= \bot \\
[\![X]\!]_{\mathbb{E}} &= [\![X]\!]_{\mathbb{B}^\to}
\end{aligned}
$$

We thus get 3 *elaborators*, as the members of $|\mathbb{C}|$, $|\mathbb{E}|$ and $|\mathrm{Hom}_{\mathbb{C}}|$ can all be represented syntactically in the underlying type theory.

Note that we could have interpreted $[\![\mathsf{view}\ A\ \mathsf{as}\ B\ \mathsf{via}\ \mathrm{v}]\!]_{\mathbb{B}}$ as $\mathsf{cod}[\![\mathsf{view}\ A\ \mathsf{as}\ B\ \mathsf{via}\ \mathrm{v}]\!]_{\mathbb{B}^\to}$, rather than as $\bot$ but this is not actually helpful, since this is just $[\![B]\!]_{\mathbb{B}}$, which is not actually what we want. What we would really want is the result of doing the substitution v into A but the resulting presentation may no longer be well-formed. So we chose to interpret the attempt to take the object component of a view as a specification error. Similarly, even though we can give an interpretation as an embedding for `mixin` when A_1 turns out to be an embedding and also for an embedding r in a view context (i.e. $\mathsf{view}\ A\ \mathsf{as}\ B\ \mathsf{via}\ \mathrm{r}$), we also choose to make these specification errors as well.

We should also note here that in our implementation, we allow raw renamings ($\{\mathrm{ren}\}$) and assignments ($\{\mathrm{assign}\}$) to be named, for easier reuse. While renamings can be given a simple categorical semantics (they induce a natural transformation on $\mathbb{C}$), assignments really need to be interpreted contextually since this requires checking that terms t are well-typed.

5.2 Type System

We build a type system, whose purpose is to ensure that well-typed expressions are denoting. Note that although we require that the underlying system have kinds, to enable the declaration of new types, we omit this below for clarity. Adding this is straightforward.

$$\frac{\varnothing \subseteq T}{[\,] : \text{Perm T}} \qquad \frac{P : \text{Perm S} \qquad S \cap \{x,y\} = \varnothing \qquad S \cup \{x,y\} \subset T}{P,[x \leftrightarrow y] : \text{Perm T}}$$

$$\frac{A : \texttt{Th} \qquad B : \texttt{Th} \qquad \texttt{B}_{\ \texttt{ctx}} \vdash x_i : \sigma_i[x_1,\cdots,x_{i-1}] \qquad \texttt{A}_{\ \texttt{ctx}} \vdash y_i : \sigma_i[y_1/x_1,\cdots,y_{i-1}/x_{i-1}]}{[x_i \mapsto y_i]_{i \le n} : \texttt{Assign A B}}$$

Fig. 6 Types for permutations and assignments

$$\frac{\vdash \ell\ \texttt{ctx}}{\texttt{Theory}\ \{\ell\} : \texttt{Emb}\ \varnothing\ \ell} \qquad \frac{A : \texttt{Th} \qquad r : \texttt{Perm}\ |\texttt{A}|}{\texttt{rename}\ A\ r : \texttt{Emb A (A[r])}}$$

$$\frac{A : \texttt{Th}}{\texttt{extend A by}\ \{\} : \texttt{Emb A A}} \qquad \frac{B = \texttt{extend A by}\ \{\ell\} \qquad B : \texttt{Th} \qquad x \notin |B| \qquad \texttt{B}_{\ \texttt{ctx}} \vdash \texttt{t:type}}{\texttt{extend A by}\ \{\ell,\ \texttt{x : t}\} : \texttt{Emb A (B,x:t)}}$$

$$\frac{\begin{array}{c} C,A',B' : \texttt{Th} \qquad A' = C[A] \rtimes A^+ \qquad B' = C[B] \rtimes B^+ \\ A : \texttt{Emb C A}' \qquad B : \texttt{Emb C B}' \qquad r_1, r_2 : \texttt{Perm}\ \varnothing \qquad \forall x \in |A'|, y \in |B'|. r_1(x) = r_2(y) \Leftrightarrow \exists z \in |C|\,.\, x = z[A] \wedge y = z[B] \end{array}}{\texttt{combine}\ A\ r_1\ B\ r_2 : \texttt{Emb C (C[A]} \rtimes \texttt{r}_1 \cdot \texttt{A}^+ \rtimes \texttt{r}_2 \cdot \texttt{B}^+\texttt{)}}$$

Fig. 7 Types for core combinators

First, we need a couple of preliminary types, shown in Figure 6. We use S and T to denote finite sets of variable from $\mathbb{V}$ and $\texttt{A}_{\ \texttt{ctx}}$ means the well-formed context that theory A elaborates to. The rule for assignments is otherwise the standard one for morphisms of the category of contexts (p.602 of [48]).

We need to define 3 sets of typing rules, one for each semantic. The rules are extremely similar to each other for most of the combinators and thus we give a lighter presentation by grouping the similar ones together. More specifically, we introduce judgement for 3 new types: `Th` for theory presentations, `Emb A B` for embeddings from presentation A to presentation B and `View A B` for views from presentation A to presentation B. Also, recall that embeddings `Emb A B` can be factored as a renaming (bijection) of the symbols of A to a subset of those of B, followed by the addition of new symbols. We denote this factoring of `View A B` as $A[B] \rtimes B^+$. The renaming thus induced on symbols z of A will be denoted $z[B]$, i.e. z as viewed inside B.

Figure 7 shows the judgements for `Emb A B` for all combinators except for `mixin`, `view` and sequential composition ;. The judgements for `Th` are obtained from these by replacing the final `: Emb A B` with `: Th`. The judgements are defined by mutual recursion: a `combine`, even elaborated as a theory, does take two views as arguments. The judgements for `Emb A B` are obtained by replacing the final `: Emb A B` with `: View A B` (recall that all embeddings are views).

$$\frac{A,B,C : \texttt{Th} \qquad X : \texttt{View A B} \qquad Y : \texttt{View B C}}{X;Y : \texttt{Th}} \qquad \frac{A,B,C : \texttt{Th} \qquad X : \texttt{Emb A B} \qquad Y : \texttt{Emb B C}}{X;Y : \texttt{Emb A C}}$$

$$\frac{A,B,C : \texttt{Th} \qquad X : \texttt{View A B} \qquad Y : \texttt{View B C}}{X;Y : \texttt{View A C}}$$

Fig. 8 Types for composition (;)

Figure 8 shows the 3 judgements for ;. These are also mutually recursive but in a non-uniform pattern. Also, in light of Proposition 4 below, the first and third rules really stand for 4 rules each.

$$\frac{A : \texttt{View C A}' \qquad B : \texttt{Emb C B}' \qquad \begin{array}{c} C, A', B' : \texttt{Th} \qquad A' = C[A] \rtimes A^+ \qquad B' = C[B] \rtimes B^+ \\ r_1, r_2 : \texttt{Perm}\ \varnothing \qquad \forall x \in |A'|, y \in |B'|. r_1(x) = r_2(y) \Leftrightarrow \exists z \in |C|\,.\, x = z[A] \wedge y = z[B] \end{array}}{\texttt{mixin}\ A\ r_1\ B\ r_2 : \texttt{Th}}$$

$$\frac{A : \texttt{View C A}' \qquad B : \texttt{Emb C B}' \qquad \begin{array}{c} C, A', B' : \texttt{Th} \qquad A' = C[A] \rtimes A^+ \qquad B' = C[B] \rtimes B^+ \\ r_1, r_2 : \texttt{Perm}\ \varnothing \qquad \forall x \in |A'|, y \in |B'|. r_1(x) = r_2(y) \Leftrightarrow \exists z \in |C|\,.\, x = z[A] \wedge y = z[B] \end{array}}{\texttt{mixin}\ A\ r_1\ B\ r_2 : \texttt{View C}\ (\texttt{C[A]} \rtimes \texttt{r}_1 \cdot \texttt{A}^+ \rtimes \texttt{r}_2 \cdot \texttt{B}^+)}$$

$$\frac{A : \texttt{Th} \qquad B : \texttt{Th} \qquad v : \texttt{Assign A B}}{\texttt{view}\ A\ \texttt{as}\ B\ \texttt{as}\ v : \texttt{View A B}}$$

Fig. 9 Types for mixin and view

The rules for `mixin` (in Figure 9) are very similar to those for `combine`, except the first argument A is now a view and the result is either a presentation or a view. A `view` of course just elaborates to a view but from an assignment.

Of course, we then have a basic soundness result:

Theorem 3 *The following hold:*

- *If* $C : \texttt{Th}$ *then* $[\![C]\!]_{\mathbb{B}}$ *is defined.*
- *If* $C : \texttt{Emb A B}$ *then* $[\![C]\!]_{\mathbb{E}}$ *is defined.*
- *If* $C : \texttt{View A B}$ *then* $[\![C]\!]_{\mathbb{B}^{\rightarrow}}$ *is defined.*

The proofs all proceed by induction on the derivations. As they completely mirror the proofs from the previous section and no new ideas are needed, so these are omitted.

As can be seen both in the algorithms and in the rules above, the interpretation as a presentation of an embedding or a view are the target theories themselves:

Proposition 2 *If* $C : \texttt{Emb A B}$ *then* $[\![C]\!]_{\mathbb{B}} = [\![B]\!]_{\mathbb{B}}$.

Proposition 3 *If* $C : \texttt{View A B}$ *then* $[\![C]\!]_{\mathbb{B}} = [\![B]\!]_{\mathbb{B}}$.

Proposition 4 *If* $C : \texttt{Emb A B}$ *then* $C : \texttt{View A B}$.

Again, these proceed by induction on the derivations. The type system was created by reverse-engineering these properties and so the results follow the reasoning embedded in the proofs in the previous section.

6 Examples

We show some progressively more complex examples, drawn from our library [19]. These are chosen to illustrate the power of the combinators and how they solve the various problems we highlighted in §2.

The simplest use of combine comes very quickly in a hierarchy built using *tiny theories*, namely when we construct a pointed magma from a magma and (the theory of) a point.

```
Empty := Theory { }
Carrier := extend Empty {U : Set}
Magma   := extend Carrier {∗ : U → U → U}
Pointed := extend Carrier {e : U}
PointedMagma := combine Magma {} Pointed {}
```

The definition of PointedMagma can alternatively be written as (Magma || Pointed), where || is used as sugar for combine.
If we want a theory of *two* points, we need to rename one of them:

```
TwoPointed := combine Pointed {} Pointed {e ↦ e′}
```

We can also extend by properties:

```
LeftUnital := extend PointedMagma
                {leftIdentity : {x : U} → e ∗ x = x}
```

This illustrates a design principle: properties should be defined as extensions of their minimal theory. Such minimal theories are most often *signatures*, in other words property-free theories. By the results of the previous section, this maximizes reusability. Even though signatures have no specific status in our framework, they arise very naturally as "universal base points" for theory development.

`LeftUnital` has a natural dual, `RightUnital`. While `RightUnital` is straightforward to define explicitly, this should nevertheless give pause, as this is really duplicating information which already exists. We can use the following self-view to capture that information:

```
Flip := view Magma as Magma via {∗ ↦ λ x y · y ∗ x}
```

Note that there is no interpretation for $[\![\texttt{Flip}]\!]_{\mathbb{B}}$ as a theory or as an embedding; if we were to perform the substitution directly, we would obtain

```
Theory { U : type; fun (x,y). y ∗ x : (U,U) → U }
```

which is ill-defined since it has a non-symbol on the left-hand-side and it contains the undefined symbol $*$.

One could be tempted to write

```
RightUnital := mixin Flip {} LeftUnital {}
```

but this is also incorrect since `LeftUnital` is an extension from `PointedMagma`, not `Magma`. The solution is to write

```
RightUnital := mixin Flip {}, (PointedMagma ; LeftUnital) {}
```

which gives a correct result but with an axiom still called `leftIdentity`; the better solution is to write

```
RightUnital := mixin Flip {}
 (rename (PointedMagma ; LeftUnital)
         {leftIdentity to rightIdentity})
```

which is the `RightUnital` we want. The construction also makes available an embedding from `Magma` (as if we had done the construction manually) as well as views from `LeftUnital` and `Magma`.

The syntax used above is sub-optimal: the path `PointedMagma;LeftUnital` may well be needed again and should be named. In other words,

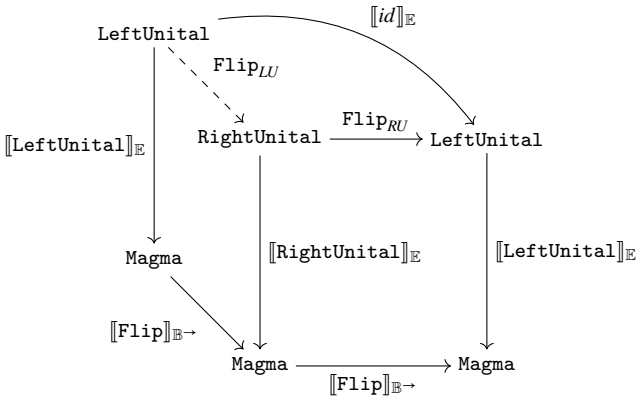

Fig. 10 Construction of `LeftUnital` and `RightUnital`. See the text for the interpretation.

```
LeftUnit := PointedMagma ; LeftUnital
```

is a useful intermediate definition.

The previous examples reinforce the importance of signatures and of morphisms from signatures to "interesting" theories as important, separate entities. For example, `Monoid` as an *embedding* is most usefully seen as a morphism from `PointedMagma`.

Our machinery also allows one to construct the inverse view, from `LeftUnital` to `RightUnital`. Consider the view `Flip;LeftUnital` and the identity view from `LeftUnital` to itself. These are exactly the inputs for `mediate`, which returns a (unique) view from `LeftUnital` to `RightUnital`. Furthermore, we obtain (from the construction of the mediating view) that this view composes with that from `RightUnital` to `LeftUnital` to give the identity. This is illustrated in Figure 6 where the $[\![-]\!]_{\mathbb{B}\to}$ annotations on nodes are omitted; note that the morphisms are in $\mathbb{C}$, not $\mathbb{P}$. Let

$$RU = \mathfrak{C}\left([\![\texttt{Flip}]\!]_{\mathbb{B}\to}, [\![\texttt{LeftUnital}]\!]_{\mathbb{E}}, [\![id]\!]_{\pi}, [\![[\![\texttt{leftIdentity} \mapsto \texttt{rightIdentity}]\!]]\!]_{\pi}\right)$$

then $\texttt{Flip}_{RU} = RU.\texttt{view}_{\texttt{LeftUnital}}$ and

$$\texttt{Flip}_{LU} = RU.\texttt{mediate}_{\texttt{LeftUnital}}\left([\![\texttt{LeftUnital}]\!]_{\mathbb{E}}; [\![\texttt{Flip}]\!]_{\mathbb{B}\to}, [\![id]\!]_{\mathbb{E}}\right)$$

The constrution of `mediate` insures that $\texttt{Flip}_{LU};\texttt{Flip}_{RU} = [\![id]\!]_{\mathbb{E}}$, *provided* that we know that

$$[\![\texttt{Flip}]\!]_{\mathbb{B}\to};[\![\texttt{Flip}]\!]_{\mathbb{B}\to} = [\![id]\!]_{\mathbb{B}\to} : [\![\texttt{Magma}]\!]_{\mathbb{B}} \to [\![\texttt{Magma}]\!]_{\mathbb{B}}.$$

The above identity is not, however, structural, it properly belongs to the underlying type theory: it boils down to asking if

$$\forall x : U.\texttt{flip}\,(\texttt{flip}\ \texttt{x}) =_{\beta\eta\delta} x$$

or, to use the notation of §3.1,

$$[U : \texttt{Type}, x : U] \vdash \texttt{flip}\,(\texttt{flip}\ \texttt{x}) \equiv x : U.$$

7 Discussion

We comment on the applicability of our work, the use of definitions in theories, an implementation and drawbacks of our choices.

7.1 Applicability

We have tried to be parametric in the underlying dependently typed theory (DTT). From a categorical point of view, contextual categories [21] as a model of DTT informs us that this is feasible. A numbers of features can be added to the type theory, at no harm to the combinators themselves – see Jacobs [48] and Taylor [75] for many such features. We do make full use of the fact that we target dependent type theories and so are not sure if our approach can be adapted, or even well suited, to weaker type theories.

7.2 Definitional extensions

One of the features that we initially built in was to allow *definitions* in our theory presentations. This restriction is only for presentations, general morphisms are also definitions. Definitions in theory presentations are useful when transporting theorems from one setting to another, as is done when using the "Little Theories" method [30]. It is beneficial to first build up *towers of conservative extensions* above each of the theories, so as to build up a more convenient common vocabulary, which then makes interpretations easier to build (and use) [17]. However, this complicates the meta-theory, and that feature has been removed from the current version. We hope to use ideas from [58] towards this goal. A referee helpfully pointed out that we can likely regain this feature by restricting embeddings so that they necessarily map undefined symbols to undefined symbols but we have not been had the time to verify this.

7.3 Implementation

We have implemented a "flattener" for our semantics, which turns a presentation A given in our language into a flat presentation $\mathsf{Theory}\{\ l\}$ by computing $\mathsf{cod}(\llbracket A \rrbracket_{\mathbb{E}})$. We have been very careful to ensure that all our constructions leave no trace of the construction method in the resulting flattened theory. We strongly believe that users of theories do not wish to be burdened by such details. Tom Hales [39, point # 7] makes this point convincingly.

Furthermore, we want developers to have maximal freedom in designing a modular, reusable and maintainable hierarchy without worrying about backwards compatibility of the *hierarchy*, only the end results: the theory presentations.

Three prototypes have been created : a stand-alone version [20, 19], one as emacs macros aimed at Agda [3], another [67] in MMT [65]. Unfortunately, although useful, none of these are entirely faithful to the semantics presented here. We have a full on atop tog [54], an implementation of Martin-Löf Type Theory. Here we focus on the tog implementation.

We keep an explicit theory graph, as well as a list of named renamings (so that they are easier to reuse). For example

```
Map plus-zero = {op to + ; e to 0}
```

is then used in the definition of `AdditiveMonoid`

```
AdditiveMonoid =
  combine AdditivePointedSemigroup {} Monoid plus-zero
    over PointedSemigroup
```

In the syntax above, we use an explicit `over` clause, which is not in §5.1. This is redundant information but it seems to make the overall expression much easier to understand, as it documents the common source of the morphisms. We use this pervasively in our `combine` expressions.

Every theory expression adds both theories and morphisms to the graph. Currently, theories are given names by the user and morphisms get system-derived names (prefixed by "To" for views, prefixed by "To" and suffixed by "1", "2" and "D" for the left, right and diagonal morphism for mixins). We are looking at improving the usability of this aspect. Generally we have found that we can refer to most morphisms implicitly.

Our rebuilt library consists of 230 theories, up to `Ring` and `BoundedDistributiveLattice`, using the tiny theories approach. Figure 11 shows how `AdditiveMonoid` is described in the library, capturing most of the structure described in Figure 2.

The flattener gives the following presentation for `AdditiveMonoid`, a dependent record parameterized over the carrier, which is type checked by Tog:

```
record AdditiveMonoid (A : Set) : Set where
  constructor AdditiveMonoidC
  field
    + : A -> A -> A
    associative_+ :
      (x : A) (y : A) (z : A) -> + (+ x y) z == + x (+ y z)
    0 : A
    lunit_0 : (x : A) -> + 0 x == x
    runit_0 : (x : A) -> + x 0 == x
```

Because of technical issues with how Tog works[4], it turns out that having the carrier automatically extracted into a parameter scales better. The two definitions are isomorphic and can be proven so in Agda (and Coq and Idris).

7.4 Drawbacks

While our work shows that these combinators work very well for building a substantial library of theories, it is not particularly beginner friendly. To use it well requires understanding the full graph structure of theories.

If one were to draw the graph of derived morphisms for `AdditiveMonoid`, the resulting picture is not quite the one introduced in Figure 2. Consider for example the following three morphisms.

- `AdditiveUnital` ⟶ `AdditiveMonoid`
- `AdditiveSemigroup` ⟶ `AdditiveMonoid`
- `Monoid` ⟶ `AdditiveMonoid`

[4] i.e. because it lacks universes

```
Map plus = {op to +}
Map zero = {e to 0}
Map plus-zero = {op to + ; e to 0}

Theory Empty = {}
Carrier = extend Empty {A : Set}
Pointed = extend Carrier {e : A}
PointedZero = rename Pointed zero
Magma = extend Carrier {op : A → A → A}
AdditiveMagma = rename Magma plus

PointedMagma  = combine Pointed {} Magma {} over Carrier
Pointed0Magma = combine PointedZero {} PointedMagma zero
                  over Pointed
PointedPlusMagma =
  combine AdditiveMagma {} PointedMagma plus over Magma
AdditivePointedMagma =
  combine Pointed0Magma plus PointedPlusMagma zero
    over PointedMagma
Semigroup =
  extend Magma
       {associative_op : {x y z : A} → op (op x y) z == op x (op y z)}
AdditiveSemigroup =
  combine AdditiveMagma {} Semigroup plus over Magma
PointedSemigroup = combine Semigroup {} PointedMagma {} over Magma
AdditivePointedSemigroup =
  combine AdditivePointedMagma {} PointedSemigroup plus-zero
    over PointedMagma
LeftUnital  = extend PointedMagma {lunit_e : {x : A} → op e x == x}
RightUnital = extend PointedMagma {runit_e : {x : A} → op x e == x}
Unital = combine LeftUnital {} RightUnital {} over PointedMagma
AdditiveUnital =
  combine AdditivePointedMagma {} Unital plus-zero
    over PointedMagma
Monoid = combine Unital {} PointedSemigroup {} over PointedMagma
AdditiveMonoid =
  combine AdditivePointedSemigroup {} Monoid plus-zero
    over PointedSemigroup
```

Fig. 11 Definitions of a theory graph up to AdditiveMonoid.

Only the last two of these can be naturally inferred from the resulting graph. Of course, using explicit calls to `view` and `diag` lets one see the necessary components. Nevertheless, getting at them is not easy.

Unfortunately, computing general colimits, or using diagram-level combinators [67] is not necessarily a solution, as these make the naming problem considerably worse.

We believe that, because pushouts are associative and commutative, providing syntax for chaining them would make getting at all induced morphisms easy. Unfortunately, this is not the case.

8 Related Work

Building large formal libraries while leveraging the structure of the domain is not new and has been tackled by theoreticians and system builders alike. We have been particularly in-

spired by the early work of Goguen and Brustall on Clear [11,12] and OBJ [35]. The semantics of their combinators is given in the category of specifications and specification morphisms. ASL [82] and Tecton [52,51] are also systems that embraced the idea of theory expressions early on. The Specware [70,71] system focuses on using the structure of theory and theory morphisms to compose specifications via applying refinement combinators and in the end to generate software that is correct-by-construction. The idea of defining combinators over theories has also been used in Maude[28] and Larch [38]. These systems gave us basic operational ideas and some of the semantic tools we needed.

However, note that the systems above are largely specification systems, whereas our interests lie largely in interactive theorem proving (ITP) systems. This explains some differences in requirements. In ITP, instantiating theories, in the same system, is a crucial operation and so we need to be able to "see" the theories, i.e. they need to have a syntactic representation.

Institutions might appear to be an ideal setting for our work. The relation to categorical logic has been worked out [36]. However, the issues with names remains. Also, it is unclear if the fibrations, crucial for mixin, are present in this setting.

A successor of ASL, CASL [24] and its current implementation Hets [56] offers many more combinators than we do for structuring specifications, as well as a documented categorical semantics. To compare with our `combine`, CASL has a `sum` operation (on a "same name, same thing" basis [6, p. 17]) that builds a colimit, similar to what is also done in Specware. However the use of "same name, same thing" is problematic when combining theories developed independently that might (accidentally) use the same name for different purposes, as we show below in the example that combines specifications `S1` and `S2` over `Carrier`. CASL structured specifications offer additional features: they allow a user to specify that a model of a specification should be *free*, or to derive a specification from a previous one via *hiding* certain fields. Unfortunately, as the CASL manual also points out [24, p. 36] "The interpretation is essentially based on model classes – a 'flattening' reduction to sets of sentences is not possible, in general (due to the presence of constructs such as hiding and freeness)." In other words, one cannot provide a constructive instantiation of these, which is incompatible with the ITPs we target.

For concreteness, we conducted some experiments using the CASL online tool [76] based on HETS [56]. Note that the definition of `Ring` below is not the one in the CASL library, which is much more disciplined than this.

1. Suppose we define `Ring` by combining `Monoid` and `AbelianGroup`, where each is defined independently, without extending a common theory.

```
spec Monoid =
  sort Elem
  ops  e: Elem;
       __ * __: Elem * Elem -> Elem, assoc, unit e
end

spec AbelianGroup =
  sort Elem
  ops  e: Elem;
       __ * __: Elem * Elem -> Elem, assoc, unit e, comm
  forall x: Elem
  . exists x': Elem . x' * x = e   %(inv_Group)%
```

```
spec Ring =
  AbelianGroup with sort Elem,
                    ops __ * __ |-> __ + __,
                          e     |-> 0
  and
    Monoid with ops e, __*__
    then
      forall x,y,z:Elem
      . (x + y) * z = (x * z) + (y * z)          %(distr1_Ring)%
      . z * ( x + y ) = (z * x) + (z * y)        %(distr2_Ring)%
end
```

This definition is accepted and the correct definition of `Ring` is computed. Understanding exactly what pushout this is supposed to denote is not obvious. Note that the CASL library includes the following definition of a view from `AbelianGroup` to `Ring`

```
view AbelianGroup_in_Ring_add :
   AbelianGroup to Ring =
      ops    e    |-> 0,
          __ * __ |-> __ + __
end
```

and uses a verbatim copy of it in the definition of Ring. Our approach eliminates this redundancy.

2. Defining the following sum operation

```
spec Carrier =
  sort Elem

spec S1 =
  Carrier
  then
    ops e : Elem
       __*__ : Elem * Elem -> Elem, unit e

spec S2 =
  Carrier
  then
    ops e : Elem
       __+__ : Elem * Elem -> Elem, unit e

spec S3 = S1 and S2
```

the computed `S3` has a carrier, two operations, and one point that acts as the unit of both operations. A proper pushout over `Carrier` should contain two points, each being the unit for one of the operations.

We also ran the same examples in Isabelle/HOL, with similar results. We have not been able to find a clear explanation of how Specware or Maude handle name clash problems. Recently, [23] gives alternatives for choosing names when a clash happen. Our choice to ask library developers to resolve the issue does put an extra burden on them, but also gives them the perfect opportunity to choose tradional notations.

Limitations of "same name, same thing" were already recognized in [62], which focuses on composing specifications and checking for isomorphism. There is no clear exposition on how name clashes are dealt with in this approach. [62] shares with us the desire for extreme modularity, taking a different route to get there. As far as we know, no implementation of this work survives.

More abstractly, Universal algebra [79, 10], specifically its constructions, has introduced the idea of manipulating theory presentations as algebraic objects. The generalization from the single-sorted equational approach to the dependently typed setting is discussed by Cartmell [21] and Taylor [75]. As we eschew all matters dealing with models, the syntactic manipulation aspects of universal algebra generalize quite readily. The syntactic concerns are also why *Lawvere theories* [53] are not as important to us. Sketches [5] certainly could have been used but would have led us too far away from directly using structures already present in the λ-calculus (namely contexts).

The Harper-Mitchell-Moggi work on Higher-order Modules [41] covers some of the same themes we do: a set of constructions (at the semantic level) similar to ours is developed for ML-style modules. However, it does not provide external syntax and does not address the application to structuring a large library of theories (or modules). Also, they do not use fibrations, since they avoided such issues "by construction". Moggi returned to this topic [55] and did make use of fibrations as well as *categories with attributes*, a categorical version of contexts. Post-facto, it is possible to recognize some of our ideas as being present in Section 6 of that work; the emphasis is however completely different. In that same vein *Structured theory presentations and logic representations* [42] does have a set of combinators. However, some parts of the semantics are unclear: Definition 3.3 of the signature of a presentation requires that both parts of a union must have the same signature (to be well formed) and yet their Example 3.6 on the next page is not well formed. That being said, many parts of the theory-level semantics is the same.

A 1997 Ph.D. thesis by Sherri Shulman [69] presents combinators to combine, generalize, extend, and instantiate theory presentations. It shares some of our ideas on extreme modularity. However, the semantics are unclear at some points, especially in cases where theories have parts in common; there are heavy restrictions on naming and no renaming, which makes the building of large hierarchies fragile.

Taylor's magnificent "Practical Foundations of Mathematics" [75] does concern itself with syntax. Although the semantic component is there, there are no algorithms and no notion of building up a library. The categorical tools are presented too but not in a way to make the connection sufficiently clear so as to lead to an implementable design. That work did lead us to investigate aspects of Categorical Logic and Type Theory, as exposed by Jacobs [48]. This work and the vast literature on categorical approaches to dependent type theory [61, 33, 47, 45, 44, 43, 29, 2, 22, 74, 32, 31] reveal that the needed structure really is already present and just needs to be reflected back into syntax.

MMT [65] shares the same motivation of building theories modularly with morphisms connecting them. It is foundation independent and possesses some rather nice web-based tools for pretty display. Its prototype theory expressions are built based on our work, which we shared with the authors. However, apart from inclusions, all morphisms need to be given manually — [27] shows some examples. Many of the scaling problems that we have identified are still present. Also, the extend operation (named `include`) is theory-internal and its semantics is not given through flattening. The result is that their theory hierarchies explicitly suffer from the "bundling" problem, as lucidly explained in [73], who introduce type classes in Coq to help alleviate this problem.

Coq has both *Canonical Structures* and *type classes* [73] but no combinators to make new ones out of old. Similarly, Lean [57] has some (still evolving) structuring mechanisms but not combinators to form new theories from old.

Isabelle's *locales* support *locale expressions*[4], which are also reminiscent of ours. However, we are unaware of a denotational semantics for them; furthermore, neither combine nor mixin are supported; their merge operation uses a same-name-same-thing semantics. Axiom [49] does support theory formation operations but these are quite restricted, as well as defined purely operationally. They were meant to mimic what mathematicians do informally when operating on theories. To the best of our knowledge, no semantics for them has ever been published.

9 Conclusion

There has been a lot of work done in mathematics to give structure to mathematical theories, first via universal algebra, then via category theory. But even though a lot of this work started out being syntactic, very quickly it became mostly semantic within a non-constructive metatheory and thus largely useless for the purposes of concrete implementations and full automation.

Here we make the observation that, for dependent type theories in common use, the category of theory presentations coincides with the opposite of the category of contexts. This allows us to draw freely from developments in categorical logic, as well as to continue to be inspired by algebraic specifications. Interestingly, key here is to make the opposite choice as Goguen's (as the main inspiration for the family of OBJ languages) in two ways: our base language is firmly higher-order, as well as dependently typed, while our "module" language is first-order and we work in the opposite category.

We provide a variant of "theory expression combinators" inspired by this semantics from categorical logic. We carefully outline the target audience for these combinators (system builders) and the requirements we feel they must fulfill. Our implementation shows that we seem to have succeeded in doing so. We can capture mathematical structure quite efficiently, while still allowing full flattening.

The design was firmly driven by its main application: to build a large library of mathematical theories, while capturing the inherent structure known to be present in such a development. To reflect mathematical practice, it is crucial to *take names seriously*. Thus renamings are fundamental, as they are in specification languages, but rather unlike today's interactive theorem provers. Categorical semantics and the desire to capture structure inexorably lead us towards considering *theory morphisms* as the primary notion of study — even though our original goal was grounded in the theories themselves. The drawback is that the resulting system requires more knowledge of theory graphs than most users really want to learn. This is one reason we aim this work at system builders.

Paying close attention to the "conventional wisdom" of category logic and categorical treatments of dependent type theory led to taking both cartesian liftings and mediating morphisms as important concepts. Doing so immediately improved the compositionality of our combinators. Noticing that this puts the focus on fibrations was also helpful. Unfortunately, taking names seriously means that the fibrations are not cloven; we turn this into an opportunity for users to retain control of their names, rather than to force some kind of "naming policy".

A careful reader will have noticed that our combinators are "external", in the sense that they take and produce theories (or morphisms or ...). Current programming languages tend

to provide "internal" combinators, such as `include`, potentially with post facto qualifiers (such as `ocaml`'s `with` for signatures) to "glue" together items that would have been identified in a setting where morphisms, rather than theories, are primary. Furthermore, we are unaware of any system that guarantees that their equivalent to our `combine` is *commutative* (Proposition 1). Lastly, this enables future features, such as limits of diagrams, rather then just binary combinations/mixins.

We used our first prototype [19] to capture the knowledge for most of the theories on Jipsen's list [50] as well as many others from Wikipedia, most of the modal logics on Halleck's list [40], as well as two formalizations of basic category theory, once dependently-typed and another following Lawvere's ETCS approach as presented on the nLab [59]. Totally slightly over 1000 theories in slightly over 2000 lines of code, this demonstrates that our combinators, coupled with the *tiny theories* approach, does seem to work. Our current implementation [16] atop Tog is rebuilding this library and seems to get there with even fewer lines of code.

Even more promising, our use of standard categorical constructions points the way to simple generalizations which should allow us to capture even more structure, without having to rewrite our library. Furthermore, as we are largely independent of the details of the type theory, this structure seems very robust and our combinators should thus port easily to other systems. We are currently actively investigating this for Agda and Lean.

Acknowledgements We warmly thank one referee for their very detailed comments and who clearly went above and beyond to help us. We also wish to thank the editor, who was both patient with us and agreed to our unusual request of relaying a few extra question to the referee for us.

Thanks also to Michael Kohlhase, Florian Rabe and William M. Farmer for many fruitful conversations on this topic. thankful to them. Also, thanks to the participants of the Dagstuhl Seminar 18341 "Formalization of Mathematics in Type Theory" whose interest in this topic helped insure that this paper got finished.

References

1. Agda Standard Library. `https://github.com/agda/agda-stdlib`
2. Ahrens, B., Lumsdaine, P.L.: Displayed categories. arXiv preprint arXiv:1705.04296 (2017)
3. Al-hassy, M.: next-700-module-systems. `https://github.com/alhassy/next-700-module-systems`. Accessed: 2019-11-20
4. Ballarin, C.: Locales and locale expressions in isabelle/isar. In: International Workshop on Types for Proofs and Programs, pp. 34–50. Springer (2003)
5. Barr, M., Wells, C.: Category theory for computing science, vol. 49. Prentice Hall New York (1990)
6. Bidoit, M., Mosses, P.D.: CASL User Manual: Introduction to Using the Common Algebraic Specification Language, vol. 2900. Springer (2003)
7. Bracha, G.: The programming language jigsaw: Mixins, modularity and multiple inheritance. Ph.D. thesis, The University of Utah (March 1992)
8. Brady, E.C.: IDRIS — systems programming meets full dependent types. In: Proceedings of the 5th ACM workshop on Programming languages meets program verification, PLPV '11, pp. 43–54. ACM, New York, NY, USA (2011). DOI http://doi.acm.org/10.1145/1929529.1929536. URL `http://doi.acm.org/10.1145/1929529.1929536`
9. Brooks, F.P.: The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (2nd Edition), anniversary edn. Addison-Wesley Professional (1995). URL `http://www.worldcat.org/isbn/0201835959`
10. Burris, S., Sankappanavar, H.: A course in universal algebra. Graduate texts in mathematics. Springer-Verlag (1981). Available free at `http://www.math.uwaterloo.ca/~snburris/htdocs/ualg.html`
11. Burstall, R.M., Goguen, J.A.: Putting theories together to make specifications. In: IJCAI, pp. 1045–1058 (1977)

12. Burstall, R.M., Goguen, J.A.: The semantics of clear, a specification language. In: D. Bjørner (ed.) Abstract Software Specifications, *Lecture Notes in Computer Science*, vol. 86, pp. 292–332. Springer (1979)
13. Carette, J., Farmer, W.M.: High-level theories. In: A.e.a. Autexier (ed.) Intelligent Computer Mathematics, *Lecture Notes in Computer Science*, vol. 5144, pp. 232–245. Springer-Verlag (2008)
14. Carette, J., Farmer, W.M., Jeremic, F., Maccio, V., O'Connor, R., Tran, Q.: The mathscheme library: Some preliminary experiments. Tech. rep., University of Bologna, Italy (2011). UBLCS-2011-04
15. Carette, J., Farmer, W.M., Sharoda, Y.: Biform theories: Project description. In: International Conference on Intelligent Computer Mathematics, pp. 76–86. Springer (2018)
16. Carette, J., Farmer, W.M., Sharoda, Y.: Leveraging the information contained in theory presentations. In: C. Benzmüller, B. Miller (eds.) Intelligent Computer Mathematics, pp. 55–70. Springer International Publishing, Cham (2020)
17. Carette, J., Farmer, W.M., Wajs, J.: Trustable communication between mathematical systems. In: T. Hardin, R. Rioboo (eds.) Proceedings of Calculemus 2003, pp. 58–68. Aracne, Rome (2003)
18. Carette, J., Kiselyov, O.: Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. Sci. Comput. Program. **76**(5), 349–375 (2011)
19. Carette, J., O'Connor, R.: Prototype of Mathscheme Combinators. `https://github.com/JacquesCarette/MathScheme/tree/master/prototype`
20. Carette, J., O'Connor, R.: Theory presentation combinators. In: J. Jeuring, J.a. Campbell, J. Carette, G. Reis, P. Sojka, M. Wenzel, V. Sorge (eds.) Intelligent Computer Mathematics, *Lecture Notes in Computer Science*, vol. 7362, pp. 202–215. Springer Berlin Heidelberg (2012). DOI 10.1007/978-3-642-31374-5_14. URL `http://dx.doi.org/10.1007/978-3-642-31374-5_14`
21. Cartmell, J.: Generalised algebraic theories and contextual categories. Annals of Pure and Applied Logic **32**, 209 – 243 (1986). DOI 10.1016/0168-0072(86)90053-9. URL `http://www.sciencedirect.com/science/article/pii/0168007286900539`
22. Clairambault, P., Dybjer, P.: The biequivalence of locally cartesian closed categories and martin-löf type theories. Mathematical Structures in Computer Science **24**(6) (2014)
23. Codescu, M., Mossakowski, T., Rabe, F.: Canonical Selection of Colimits. In: P. James, M. Roggenbach (eds.) Recent Trends in Algebraic Development Techniques, pp. 170–188. Springer (2017)
24. CoFI (The Common Framework Initiative): CASL Reference Manual. LNCS Vol. 2960 (IFIP Series). Springer-Verlag (2004)
25. mathlib Community, T.: The lean mathematical library (2019)
26. Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A.P.: Traits: A mechanism for fine-grained reuse. ACM Trans. Program. Lang. Syst. **28**(2), 331–388 (2006). DOI 10.1145/1119479.1119483. URL `http://doi.acm.org/10.1145/1119479.1119483`
27. Dumbrava, S., Horozal, F., Sojakova, K.: A case study on formalizing algebra in a module system. In: Proceedings of the 1st Workshop on Modules and Libraries for Proof Assistants, MLPA '09, pp. 11–18. ACM, New York, NY, USA (2009). DOI 10.1145/1735813.1735816. URL `http://doi.acm.org/10.1145/1735813.1735816`
28. Durán, F., Meseguer, J.: Maude's module algebra. Science of Computer Programming **66**(2), 125–153 (2007)
29. Dybjer, P.: Internal type theory. In: International Workshop on Types for Proofs and Programs, pp. 120–134. Springer (1995)
30. Farmer, W.M., Guttman, J.D., Thayer, F.J.: Little theories. In: CADE-11: Proceedings of the 11th International Conference on Automated Deduction, pp. 567–581. Springer-Verlag, London, UK (1992)
31. Fiore, M.P.: Mathematical models of computational and combinatorial structures. In: International Conference on Foundations of Software Science and Computation Structures, pp. 25–46. Springer (2005)
32. Firore, M., Turi, D.: Semantics of name and value passing. In: Proceedings 16th Annual IEEE Symposium on Logic in Computer Science, pp. 93–104. IEEE (2001)
33. Gambino, N., Larrea, M.F.: Models of martin-löf type theory from algebraic weak factorisation systems (2019)
34. Geuvers, H., Pollack, R., Wiedijk, F., Zwanenburg, J.: A constructive algebraic hierarchy in coq. Journal of Symbolic Computation **34**(4), 271 – 286 (2002). DOI https://doi.org/10.1006/jsco.2002.0552. URL `http://www.sciencedirect.com/science/article/pii/S0747717102905523`
35. Goguen, J., Kirchner, C., Kirchner, H., Mégrelis, A., Meseguer, J., Winkler, T.: An introduction to obj 3. In: International Workshop on Conditional Term Rewriting Systems, pp. 258–263. Springer (1987)
36. Goguen, J.A., Mossakowski, T., de Paiva, V., Rabe, F., Schröder, L.: An institutional view on categorical logic. Int. J. Software and Informatics **1**(1), 129–152 (2007)
37. Grabowski, A., Schwarzweller, C.: On duplication in mathematical repositories. In: S. Autexier, J. Calmet, D. Delahaye, P. Ion, L. Rideau, R. Rioboo, A. Sexton (eds.) Intelligent Computer Mathematics, *Lecture Notes in Computer Science*, vol. 6167, pp. 300–314. Springer Berlin / Heidelberg (2010). DOI 10.1007/978-3-642-14128-7_26

38. Guttag, J.V., Horning, J.J.: Larch: languages and tools for formal specification. Springer Science & Business Media (2012)
39. Hales, T.: A review of the lean theorem prover (2018). URL https://jiggerwit.wordpress.com/2018/09/18/a-review-of-the-lean-theorem-prover/
40. Halleck, J.: Logic system interrelationships. http://www.horizons-2000.org/2.%20Ideas%20and%20Meaning/John%20Halleck%27s%20Logic%20System%20Interrelationships.html. Accessed: December 14, 2018
41. Harper, R., Mitchell, J.C., Moggi, E.: Higher-order modules and the phase distinction. In: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 341–354. ACM (1989)
42. Harper, R., Sannella, D., Tarlecki, A.: Structured theory presentations and logic representations. Annals of Pure and Applied Logic **67**(1), 113 – 160 (1994). DOI https://doi.org/10.1016/0168-0072(94)90009-4. URL http://www.sciencedirect.com/science/article/pii/0168007294900094
43. Hofmann, M.: On the interpretation of type theory in locally cartesian closed categories. In: International Workshop on Computer Science Logic, pp. 427–441. Springer (1994)
44. Hofmann, M.: Extensional concepts in intensional type theory. Ph.D. thesis, University of Edinburgh (1995)
45. Hofmann, M.: Syntax and semantics of dependent types. In: Extensional Constructs in Intensional Type Theory, pp. 13–54. Springer (1997)
46. Huet, G.P., Saïbi, A.: Constructive category theory. In: G.D. Plotkin, C. Stirling, M. Tofte (eds.) Proof, Language, and Interaction, Essays in Honour of Robin Milner, pp. 239–276. The MIT Press (2000)
47. Jacobs, B.: Comprehension categories and the semantics of type dependency. Theoretical Computer Science **107**(2), 169 – 207 (1993). DOI https://doi.org/10.1016/0304-3975(93)90169-T. URL http://www.sciencedirect.com/science/article/pii/030439759390169T
48. Jacobs, B.: Categorical Logic and Type Theory. No. 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Amsterdam (1999)
49. Jenks, R.D., Sutor, R.S.: Axiom: the scientific computation system. Springer (1992)
50. Jipsen, P.: List of mathematical structures. http://math.chapman.edu/~jipsen/structures/doku.php. Accessed: December 14, 2018
51. Kapur, D., Musser, D.: Tecton: a framework for specifying and verifying generic system components. Tech. rep., Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York (1992). RPI-92-20
52. Kapur, D., Musser, D.R., Stepanov, A.A.: Tecton: A language for manipulating generic objects. In: Program Specification, Proceedings of a Workshop, pp. 402–414. Springer-Verlag, London, UK, UK (1982). URL http://dl.acm.org/citation.cfm?id=647531.729043
53. Lawvere, W.F.: Functorial Semantics of Algebraic Theories. Reprints in Theory and Applications of Categories **4**, 1–121 (2004). URL http://tac.mta.ca/tac/reprints/articles/5/tr5.pdf
54. Mazzoli, F., Danielsson, N.A., Norell, U., Vezzosi, A., Abel, A.: Tog, a prototypical implementation of dependent types. https://github.com/bitonic/tog
55. Moggi, E.: A category-theoretic account of program modules. In: Category Theory and Computer Science, pp. 101–117. Springer-Verlag, London, UK, UK (1989). URL http://dl.acm.org/citation.cfm?id=648332.755571
56. Mossakowski, T., Maeder, C., Lüttich, K.: The heterogeneous tool set, hets. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 519–522. Springer (2007)
57. de Moura, L., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The lean theorem prover (system description). In: A.P. Felty, A. Middeldorp (eds.) Automated Deduction - CADE-25, pp. 378–388. Springer International Publishing, Cham (2015)
58. Nederpelt, R., Geuvers, H.: Type Theory and Formal Proof: An Introduction, 1st edn. Cambridge University Press, New York, NY, USA (2014)
59. nLab authors: fully formal ETCS. http://ncatlab.org/nlab/show/fully Revision 31
60. Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Dept. Comp. Sci. and Eng., Chalmers Univ. of Technology (2007)
61. North, P.R.: Type-theoretic weak factorization systems (2019)
62. Oriat, C.: Detecting equivalence of modular specifications with categorical diagrams. Theor. Comput. Sci. **247**(1-2), 141–190 (2000)
63. Pitts, A.M.: Categorical logic. In: S. Abramsky, D.M. Gabbay, T.S.E. Maibaum (eds.) Handbook of Logic in Computer Science, Volume 5. Algebraic and Logical Structures, pp. 39–128. Oxford University Press (2000). URL http://www.oup.co.uk/isbn/0-19-853781-6
64. Pitts, A.M.: Nominal sets: Names and symmetry in computer science, vol. 57. Cambridge University Press (2013)

65. Rabe, F., Kohlhase, M.: A scalable module system. Inf. Comput. **230**, 1–54 (2013). DOI 10.1016/j.ic.2013.06.001. URL http://dx.doi.org/10.1016/j.ic.2013.06.001
66. Rabe, F., Müller, D.: Structuring theories with implicit morphisms. In: J.L. Fiadeiro, I. Ţuţu (eds.) Recent Trends in Algebraic Development Techniques, pp. 154–173. Springer International Publishing, Cham (2019)
67. Rabe, F., Sharoda, Y.: Diagram combinators in mmt. In: C. Kaliszyk, E. Brady, A. Kohlhase, C. Sacerdoti Coen (eds.) Intelligent Computer Mathematics, pp. 211–226. Springer International Publishing, Cham (2019)
68. Sakkinen, M.: Disciplined inheritance. In: ECOOP, vol. 89, pp. 39–56 (1989)
69. Shulman, S.J.: A meta-theory for structured presentations in the coc. Ph.D. thesis, Oregon Graduate Institute of Science & Technology, Beaverton, OR, USA (1997). UMI Order No. GAX97-24794
70. Smith, D.R.: Constructing specification morphisms. Journal of Symbolic Computation **15**, 5–6 (1993)
71. Smith, D.R.: Mechanizing the development of software. In: M. Broy, R. Steinbrueggen (eds.) Calculational System Design, Proceedings of the NATO Advanced Study Institute, pp. 251–292. IOS Press, Amsterdam (1999)
72. Spitters, B., van der Weegen, E.: Type classes for mathematics in type theory. CoRR **abs/1102.1323** (2011). URL http://arxiv.org/abs/1102.1323
73. Spitters, B., van der Weegen, E.: Type classes for mathematics in type theory. Mathematical Structures in Computer Science **21**(4), 795–825 (2011)
74. Streicher, T.: Semantics of type theory: correctness, completeness and independence results. Springer Science & Business Media (2012)
75. Taylor, P.: Practical Foundations of Mathematics. Cambridge Studies in Advanced Mathematics. Cambridge University Press (1999). URL http://books.google.ca/books?id=iSCqyNgzamcC
76. Team, H.D.: Hets casl web interface. URL http://rest.hets.eu/
77. Team, T.C.D.: The coq proof assistant, version 8.12.0 (2020). DOI 10.5281/zenodo.4021912. URL https://doi.org/10.5281/zenodo.4021912
78. Univalent Foundations Program, T.: Homotopy Type Theory: Univalent Foundations of Mathematics. http://homotopytypetheory.org/book, Institute for Advanced Study (2013)
79. Whitehead, A.: A treatise on universal algebra: with applications. No. v. 1 in Cornell University Library historical math monographs. The University Press (1898). URL http://books.google.ca/books?id=XUwNAAAAYAAJ
80. Wiki, H.: Functor-applicative-monad proposal (2015). URL https://wiki.haskell.org/Functor-Applicative-Monad_Proposal. [Online; accessed 14-November-2019]
81. Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W., Kolovos, D., Paige, R., Lauder, M., Schürr, A., Wagelaar, D.: A comparison of rule inheritance in model-to-model transformation languages. In: J. Cabot, E. Visser (eds.) Theory and Practice of Model Transformations, pp. 31–46. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
82. Wirsing, M.: Structured algebraic specifications: A kernel language. Theoretical Computer Science **42**, 123–249 (1986)