Theory Expressions (A Survey)

Florian Rabe

Jacobs University Bremen

Abstract. Theories are an essential structuring concept in mathematics. Therefore, many formal languages for mathematical knowledge support some form of theory expressions, i.e., a set of built-in operators that return new theories. We survey the commonly-used theory-forming operators and the formal languages that feature them.

1 Introduction

Theories are one of the most fundamental concepts of mathematics. They allow the Bourbaki method [Bou74] of systematically stating every result (definitions, notations, theorems, algorithms, etc.) in the smallest possible context. Moreover, theory morphisms allow moving and thus reusing results between theories along complex translations [FGT92]. Finally, they allow systematically building large theories out of small reusable components, a technique crucial for scalability.

Consequently, virtually every formal language used for mathematics has some notion of theory. Moreover, almost all of these have developed at least limited (although sometimes somewhat ad hoc) support for theory expressions. Among the simplest examples is the union operator $S \cup T$, which returns the theory consisting of all declarations of the theories S and T.

However, even intuitively simple theory expressions tend to be very difficult to support in formal languages. Consequently, no state-of-the-art system supports all theory expressions needed to elegantly formalize even simple suites of mathematical examples. We posit that designing a satisfactory formal language of theory expressions is still an open problem.[1]

Therefore, this paper surveys the state of the art and serves as a starting point for systematically designing future languages. Our contribution is twofold. Firstly, we give a comprehensive collection of theory expressions that are used in informal and/or formal mathematics. Secondly, we survey current specification, deduction, and programming languages according to their theory expressions.

Sect. 2 introduces a general definition of basic theory. Sect. 3 describes three design choices, according to which languages with theory expressions can be classified. Even though these choices fundamentally affect the style of the language, they are mostly orthogonal to the question of which theory expressions are present. Sect. 5 compiles the set of operators that languages can provide to form complex theory expressions from basic theories. Accordingly, Sect. 4 introduces predicates that express properties of theory expressions. Finally Sect. 6

[1] In fact, this work is a by-product of the author's ongoing effort to do just that.

applies our terminology to concisely survey existing languages. We conclude in Sect. 7 with some design goals for future languages.

2 Basic Theories

Even though formal languages differ tremendously in their syntax and semantics (not to mention the tools implementing them), the concept of *theory* is used rather uniformly across all systems (although not always under that name; an overview can be found in [RK13]). A major feature of the MMT language [RK13] is that it provides an extremely general setting in which this observation can be made precise: It gives a concise definition of *theory* without making language-specific assumptions about syntax or semantics. Therefore, we use MMT to make our discussion precise. In the sequel we recap non-modular MMT theories and morphism.

Definition 1 (Theory). *A* ***basic theory*** T *is of the form*

T	$::=$	$D, \ldots, D$	*basic theory*
D	$::=$	$c[: t][= d]$	*symbol declaration with optional* ***type*** *and* ***definiens***
t, d	$::=$	$c \mid \ldots$	OPENMATH *objects built from symbols* c

such that each t *and* d *use no free variables and only use previously declared symbols. We omit the remaining productions for* OPENMATH *objects*

We write $\texttt{dom}(T)$ *for the set of symbol* ***identifiers*** c *declared in* T *and* $\texttt{Obj}(T)$ *for the set of closed objects using only symbols* $c \in \texttt{dom}(T)$.

These theories are similar to OpenMath content dictionaries [BCC+04] but simpler in some aspects and more precise in others.

Symbol declarations subsume virtually all basic declarations common in formal systems such as type/function/predicate symbols, axioms, theorems, inference rules, etc. In particular, axioms [theorems] can be represented via the propositions-as-types correspondence as declarations $c : F[= p]$, which assert the sentence F [via proof p]. Similarly, objects subsume virtually all objects common in formal systems such as terms, types, formulas, proofs.

Individual formal languages arise as fragments of MMT: They single out the well-formed objects and theories by defining the two MMT-**judgments** $\vdash_T o : o'$ (typing) and $\vdash_T o \equiv o'$ (equality) for every theory T and $o, o' \in \texttt{Obj}(T)$. The details can be found in [Rab14].

Definition 2 (Morphism). *For two theories* S *and* T*, a* ***basic morphism*** $M : S \to T$ *is a list of symbol assignments* $c := o$ *for* $o \in \texttt{Obj}(T)$ *such that there is exactly one assignment for each* $c \in \texttt{dom}(S)$.

M *induces a* ***homomorphic extension*** $M(-) : \texttt{Obj}(S) \to \texttt{Obj}(T)$*, which replaces every* S*-symbol in an* S*-object with the corresponding* T*-object provided by* M.

If T *is the current theory, we also call* M *a* ***ground morphism*** *of* S.

Morphisms subsume a wide variety of inter-theory relations including inheritance, refinement, translations, and representation theorems. Ground morphisms of S allow formally representing implementations and models of S, in which case $M(-)$ is the interpretation function that maps objects to their denotation. Moreover, morphisms $M : S \to T$ subsume functors that map T-implementations/models to S-implementations/models.[2]

Crucially, morphisms **preserve judgments**, in particular they map theorems to theorems. Formally, $M : S \to T$ is *well-formed* if

- whenever c has type t and $c := o$ in M, then $\vdash_T o : M(t)$,
- whenever c has definiens d and $c := o$ in M, then $\vdash_T o \equiv M(d)$.

In that case, if $\vdash_S o : o'$ (or $\vdash_S o \equiv o'$), then $\vdash_T M(o) : M(o')$ (or $\vdash_T M(o) \equiv M(o')$). Again, the details can be found in [Rab14].

We write $\texttt{Thy}^\flat$ and $\texttt{Mor}^\flat(S,T)$ for the **sets** of basic theories and basic morphisms as defined above and $\texttt{Thy} \supseteq \texttt{Thy}^\flat$ and $\texttt{Mor}(S,T) \supseteq \texttt{Mor}^\flat(S,T)$ for the sets of complex theory and morphism expressions of specific languages.

3 Fundamental Aspects

3.1 Syntactic Status of Theories

We speak of **external theories** if a language allows theory declarations $T = \{\ldots, c_i[: t_i][= d_i], \ldots\}$ for a new identifier T but guarantees that T never occurs in objects. This introduces a 2-layered hierarchy of object- and theory-level such that the latter talks about the former but not vice versa. Therefore, external theories are conservative in the sense that they do not change the set of well-formed objects. External (ground) morphisms are defined accordingly.

External theories are used in virtually every practical system. The most well-known example are SML-style *module systems* [MTHM97]: Here external theories, ground morphisms of S, and morphisms $S \to T$ correspond to, respectively, signatures, structures typed by S, and functors $F : T \to S = \lambda x : T.\{\ldots, c_i := o_i, \ldots, \}$.

The sharp distinction between object- and theory-level is a disadvantage if we want to compute or reason with theories. Usually, that requires primitives that are already present in the object language such as equality or λ-abstraction, which have to duplicated at the theory level. Therefore, it is desirable to use **internal theories**: Here a language expresses theory declarations as a special case of symbol declarations, and theories become a special case of objects.

Often internal theories are more restricted than external ones: They may forbid certain declarations (e.g., imports, definitions, or notations), forbid certain t_i (e.g., type declarations or axioms), or forbid dependencies (e.g., c_1 occurring in t_2). The most widely used example are *records*. Here internal theories correspond

[2] Note that morphisms translate expressions one way but models the other way. We find this effect in all languages, and in MMT we can understand it in general: If we represent models as ground morphisms, then $M : S \to T$ maps the T-model $m : T \to U$ to the S-model $m \circ M : S \to U$.

to record types, which are usually restricted to the form $S = \{\ldots, c_i : t_i, \ldots\}$. Internal ground morphisms of S correspond to record terms $\{\ldots, c_i := o_i, \ldots\}$ of type S. And internal morphisms correspond to functions between record types.

Because external theories are usually present anyway, internal theories introduce some redundancy. This adds flexibility for ad hoc use, but formalizations and tool support may end up being duplicated when communities build large libraries of theories collaboratively. This motivates **internalized theories** where the external theories double as internal ones.

The most widely used example are *classes* in object-oriented programming languages. Theories correspond to abstract classes S, which are at the same time object-level types. Ground morphisms of S correspond to concrete classes M implementing S, which are (if we create a singleton instance of M) at the same time objects of type S. Morphisms $S \to T$ correspond to classes implementing S with a constructor argument of type T.

3.2 Formation of Complex Theories

We call the formation of complex theories **declaration-based** if the syntax of basic theories is extended to support import declarations. The simplest such declaration is inclusion as in[3]

$$\texttt{AbelianGroup} = \{\textbf{include}\,\texttt{Group}, \texttt{commutative} : \forall x, y. x \circ y \doteq y \circ x\}$$

Combined with anonymous theories (see Sect. 5.1), this allows defining many operators discussed in Sect. 5, including extension (as above), union (written as $\{\textbf{include}\,S, \textbf{include}\,T\}$), and pushout (via a more complex import declaration).

We call the formation **operator-based** if the syntax of objects is extended with operators that return theories. For example, the union of two theories would be $S \cup T$ for a symbol $\cup$ declared in an appropriate theory.

Declaration-based formation is supported by many languages (e.g., Mmt itself), often in addition to operator-based formation. Declaration-based formation was discussed and surveyed extensively in [RK13], and we will focus on operator-based formation in the sequel.

3.3 Semantic Status of Theories

Denotational semantics uses a class $\texttt{Mod}$ of models and functions (i) $[\![-]\!] : \texttt{Thy} \to \mathcal{P}(\texttt{Mod})$ assigning to each theory its class of models, and (ii) $[\![-]\!] : \texttt{Mor}(S,T) \to ([\![T]\!] \to [\![S]\!])$ assigning to each morphism its model translation functor. As a special case, we obtain the semantics of ground morphisms of S as elements of $[\![S]\!]$.

Denotational semantics allows **theories-as-types** in the sense that we treat every theory S as the type $[\![S]\!]$ of its models and morphisms as functions between

[3] Here and in all following examples, we will use common mathematical notation for objects and assume that all relevant symbols (e.g., $\forall$) have previously been declared or imported. Mmt [RK13] handles the latter formally using meta-theories.

these types. All example languages of Sect. 3.1 use theories-as-types. This is particularly appealing when using internal theories in languages that already use a type system. However, because $[\![S]\!]$ is usually a proper class, theories-as-types is not compatible with the standard interpretation of types as sets.

T is called **flattenable** if there is a basic theory $T^\flat$ such that $[\![T]\!] = [\![T^\flat]\!]$, and accordingly for (ground) morphisms. Because there are more model classes than basic theories, denotational semantics allows for unflattenable expressions.

Constructive semantics uses functions $-^\flat : \mathtt{Thy} \to \mathtt{Thy}^\flat$ and $-^\flat : \mathtt{Mor}(S,T) \to \mathtt{Mor}^\flat(S^\flat, T^\flat)$ (which must be the identity for basic expressions). This can be seen as the special case of denotational semantics where all expressions are flattenable. In this case, theory expressions do not introduce semantically new theories and only provide new ways of writing existing ones.

The most widely used example of unflattenable theories are theories with hidden declarations (see Sect. 5.1). To accommodate hiding, constructive semantics with **hidden declarations** was developed in [GR04]: It divides the declarations in $T^\flat$ into the visible and the hidden ones.

4 Relations between Theories

Several properties (incidentally all binary relations) about theories and morphisms are frequently used. Interestingly, they are mostly used with external theories even though they would also be useful at the object level as predicates on internal theories.

Two theories S and T are **equal**, written $S \equiv T$, iff $[\![S]\!] = [\![T]\!]$. Equality of (ground) morphisms is defined accordingly. Two theories S and T are **isomorphic**, written $S \cong T$, if there are morphisms $M : S \to T$ and $M' : T \to S$ such $[\![M]\!]$ and $[\![M']\!]$ are inverse to each other. Both are equivalence relations.

Equality and isomorphism may or may not be decidable, and languages may or may not provide an inference system for them. However, the equality of *flattenable* theories can be determined: $S \equiv T$ holds iff $\mathtt{dom}(S^\flat) = \mathtt{dom}(T^\flat)$ and for every $c \in \mathtt{dom}(S^\flat)$

- c has a type s in $S^\flat$ iff it has a type t in $T^\flat$, and if so $\vdash_{S^\flat} s \equiv t$,
- c has a definiens s in $S^\flat$ iff it has a definiens t in $T^\flat$, and if so $\vdash_{S^\flat} s \equiv t$.

Moreover, in that case, $\mathtt{Obj}(S) = \mathtt{Obj}(T)$. The corresponding criterion applies to the equality of morphisms. Thus, if flattening is computable, equality is decidable for theories and morphisms iff it is decidable for objects.

We say that T **includes** (or extends) S, written $S \hookrightarrow T$, if the morphism $i : S \to T$ mapping all S-identifiers to themselves is well-formed. Inclusion is an order relation (with respect to $\equiv$), which – under certain reasonable conditions – is equivalent to $[\![T]\!] \subseteq [\![S]\!]$. If $S \hookrightarrow T$, there are two definitions of when the inclusion is **conservative**:

- constructive semantics: if every MMT-judgment about S-objects is derivable in T iff it is derivable in S,
- denotational semantics: if every S-model can be extended[4] to a T-model.

[4] An extension of $m \in [\![S]\!]$ is an $m' \in [\![T]\!]$ such that $[\![i]\!](m') = m$.

Both definitions have a similar style, but they are not equivalent; for example, in sound and complete logics, denotational implies constructive conservativity but not vice versa.

5 Operators that Form Complex Theories

No formal language exists that supports all operators. Therefore, we state all our examples in a hypothetical extension of MMT.

5.1 Operators on Symbol Declarations

Anonymous Theory An anonymous theory is an expression of the form $\{\ldots, c[: t][= d], \ldots\}$. Intuitively, this is the simplest possible theory expression. However, it is non-trivial to define the scope of the identifiers c and to keep track of which identifiers are present in which theory. In particular, it is not possible to form qualified identifiers to disambiguate if the same c is used in multiple anonymous theories. Therefore, many languages restrict their use, e.g., only use them as the definiens of a theory identifier.

Anonymous (ground) morphisms are defined accordingly. These do not suffer from the same problem because morphisms do not introduce new identifiers.

Extension Theory extensions add declarations or declaration parts to an existing theory as in

$$\texttt{AbelianGroup} = \texttt{Group}\ \textbf{with}\{\texttt{commutative} : \forall x, y. x \circ y \doteq y \circ x\}$$

which extends the theory of groups with an axiom for commutativity. More subtly, we can add a definiens to an existing symbol without definiens as in

$$\texttt{Field} = \{\texttt{universe} : \texttt{Set}, \ldots\}$$

$$\texttt{VectorSpace} = \{F : \texttt{Field},\ V : \texttt{Set}, \ldots, \texttt{mult} : F.\texttt{universe} \times V \to V, \ldots\}$$

$$\texttt{RealVectorSpace} = \texttt{VectorSpace}\ \textbf{with}\ \{F := \mathbb{R}\}$$

In both cases, we obtain an inclusion morphism from the original to the extended theory. Both with constructive and denotational semantics, this inclusion is conservative if all added declarations have a definiens.

Removal This is a relatively rare dual of extension that removes declarations from a theory. It is very useful in the typical case where an existing theory library is not maximally modular, e.g., we might find

$$\texttt{SkewField} = \texttt{Field}\ \textbf{without}\ \texttt{commutativeMult}$$

Note that this operator is more complex than extension because it must also remove all declarations that depend on the removed ones such as theorems whose proofs use commutativity.

Removal induces an inclusion morphism from the shrunk to the original theory. In [RKS11], a form of partial morphisms (called *filtering*) was introduced so that removal induces a filtering morphism in the opposite direction.

Hiding This operator is syntactically similar to removal but behaves very differently: It makes an identifier inaccessible but retains the object it identifies. A typical application is to hide auxiliary declarations that should not be exported.

The denotational semantics of a theory with hiding is the class of all models of the public interface that can be extended to models of the hidden declarations, i.e., $[\![T\,\mathbf{hide}\,c]\!] = \mathtt{image}([\![i]\!])$ where i is the inclusion morphism $T\,\mathbf{without}\,c \hookrightarrow T$. Theories with hiding cannot be flattened in general. Using constructive semantics with hidden declarations, we can define $(T\,\mathbf{hide}\,c)^\flat$ as $T^\flat$ with c and all declarations depending on it marked hidden. [GR04] formalizes this by representing basic theories as a pair $I \hookrightarrow T$ such that T declares the internally available identifiers and I is the publicly visible interface.

We call the above *hiding along inclusions*. We obtain *general hiding* by allowing any morphism $M : I \to T$ and putting $[\![T\,\mathbf{hide}\,M]\!] = \mathtt{image}([\![M]\!])$.

Renaming The operator $T[c \rightsquigarrow c']$ leaves T unchanged except for renaming the identifier c to c'. Renaming is only occasionally interesting in itself. But it can be crucial to avoid name clashes when combining theories (e.g., as in Sect. 5.3).

5.2 Parametrization and Instantiation

The vector space example from Sect. 5.1 can alternatively be written as a parametric theory:

$$\mathtt{VectorSpace}(F : \mathtt{Field}) = \{V : \mathtt{Set}, \ldots, \mathtt{mult} : F.\,\mathtt{universe} \times V \to V, \ldots\}$$

$$\mathtt{RealVectorSpace} = \mathtt{VectorSpace}(\mathbb{R})$$

The parameters can be normal values (i.e., typed by a object type) or ground morphisms (i.e., typed by a theory). The above example uses the latter, i.e., the parameter F represents a ground morphism (intuitively: a model) of the theory `Field`. When using theories-as-types, the latter is a special case of the former. Otherwise, the two are orthogonal features, and we speak of *object-parameters* and *theory-parameters*.

A common example of parametric theories are classes with type parameters such as Java generics. Using internal theories and type-valued functions, parametric theories are a special case of λ-abstractions that return a record type, and instantiation becomes a special case of function application. However, as these are rather strong requirements (satisfied, e.g., by the dependent type theory underlying Coq [Coq14]), parametric theories are often a separate feature.

Parametrization and instantiation of (ground) morphisms can be defined accordingly. For example,

$$\mathtt{FiniteField}(p : \mathtt{Prime}) : \mathtt{Field} = \{\mathtt{universe} := \mathbb{Z}_p, \ldots\}$$

is a parametric ground morphism of the theory of fields.

5.3 Colimit Operators

Colimits form larger theories out of smaller ones. The resulting theories are flattenable and retain the identifiers of their constituents.

Empty The empty theory $\{\}$ and the empty morphism $\{\} : \{\} \to T$ are trivial nullary operators that are special cases of anonymous expressions. Their main value is that $\{\}$ is an initial object in the category of theories.

Union Union is a very common binary operator on theories. For example,

$$\texttt{AbelianGroup} = \texttt{Group} \cup \texttt{CommutativeMagma}$$

If we think of basic theories as sets of declarations, the flattening of the union theory can be defined by $(S \cup T)^\flat = S^\flat \cup T^\flat$. Thus, we have $S \hookrightarrow S \cup T$ and $T \hookrightarrow S \cup T$. However, the union operator is partial because $S^\flat \cup T^\flat$ is only well-formed if $S^\flat$ and $T^\flat$ assign the same type and definiens to every identifier in $\texttt{dom}(S^\flat) \cap \texttt{dom}(T^\flat)$ $(*)$. This can be non-trivial to check, especially if equality of objects is undecidable. Above `AbelianGroup` is well-formed assuming that $\texttt{Magma} \hookrightarrow \texttt{Group}$ and $\texttt{Magma} \hookrightarrow \texttt{CommutativeMagma}$.

The requirement $(*)$ can be relaxed. If both $S^\flat$ and $T^\flat$ declare c with the same type but only one of them assigns a definiens, then that definiens can be used in $(S \cup T)^\flat$. (The according provision could be used for types, but that is rare.) With this relaxation, the union operator is also called *mixin composition*.

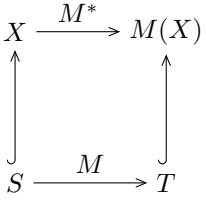
Even more relaxed, we can define a variant where the definiens in $T^\flat$ *overrides* a definiens in $S^\flat$ (or vice versa). Some programming languages use the `override` keyword for that purpose. This is often considered harmful because we no longer have $S \hookrightarrow S \cup T$.

An orthogonal generalization is *dependent union*, which can also be seen as a variant of extension. If $T(M)$ is a theory with a theory-parameter M of S, then $S \cup T(id_S)$ extends S with the declarations of T where M is instantiated with the identity morphism of S (see Sect. 5.4). Dependent unions behave similarly to Σ-types: Models of $S \cup T$ are certain pairs $(M, M') \in [\![S]\!] \times [\![T]\!]$, and models of $S \cup T(id_S)$ are dependent pairs $(M, M') \in \Sigma_{M \in [\![S]\!]} [\![T]\!](M)$.

Pushout The *general pushout* $\mathbf{pushout}\, M, M'$ takes two morphisms $M : S \to T$ and $M' : S \to T'$ as arguments. Intuitively, it combines T and T' in such a way that they share exactly S. We obtain $(\mathbf{pushout}\, M, M')^\flat$ as the union of $T^\flat$ and $T'^\flat$ extended with axioms $c : M(c) \doteq M'(c)$ for all $c \in \texttt{dom}(S^\flat)$.[5] Thus, all pushouts exist if we use a logic that can express the equality of anything that can be declared. Example are untyped first-order logic or type theories with sentences for type equality.

[5] Actually, this yields a pushout only if $\texttt{dom}(S^\flat)$, $\texttt{dom}(T^\flat)$, and $\texttt{dom}(T'^\flat)$ are pairwise disjoint. Otherwise, declarations have to be renamed accordingly.

Of particular importance is the special case of *pushout of inclusions*. Here $M' : S \hookrightarrow X$ is an inclusion, and we write the pushout as **pushout** X **via** M or simply $M(X)$. It can be desirable to treat this case as a separate operator because it allows a much simpler semantics: We do not have to add axioms when flattening (and thus do not have to generate proof obligations when giving models later on). The key idea of constructing the pushout of an inclusion is that $M(X)$ extends T with the translation via M of those declarations that X extends S with, e.g.,



$$M(S, \ldots, c : t = d, \ldots) \;=\; T, \ldots, c : M^*(t) = M^*(d), \ldots$$

where M^* is like M but additionally maps $c := c$ for $c \in \mathtt{dom}(X) \setminus \mathtt{dom}(S)$. Thus, we can think of $M(X)$ as the homomorphic extension of M to X, which justifies our notation.[6]

For example, given an extension $\mathtt{FOL} \hookrightarrow \mathtt{Group}$ (i.e., the theory of groups defined by extending first-order logic with non-logical symbols) and a morphism $\mathtt{FOL2HOL} : \mathtt{FOL} \to \mathtt{HOL}$, which represents a logic translation, we can write

$$\mathtt{HOLGroup} = \textbf{pushout}\ \mathtt{Group}\ \textbf{via}\ \mathtt{FOL2HOL}$$

for the HOL-theory arising from Group by translating every declaration of FOL according to FOL2HOL.

Together with anonymous morphisms, this allows extending a theory with a definiens: The vector space example from Sect. 5.1 can be written as

$$\mathtt{RealVectorSpace} = \textbf{pushout}\ \mathtt{VectorSpace}\ \textbf{via}\ \{F := \mathbb{R}\}$$

Another important special case arises when M and M' are inclusions and $S \hookrightarrow I$, where I is the largest theory included into both T and T'. We can write it as **pushout** T, T' **over** S. It can be seen as a variant of union with limited sharing: It combines T and T' sharing exactly the declarations in S and duplicating the ones for $\mathtt{dom}(I) \setminus \mathtt{dom}(S)$.[7] In particular, if $S = \{\}$, we obtain the disjoint union of T and T', and if $S = I$, we obtain $T \cup T'$.

For example to define the theory Ring we would use

$$\textbf{pushout}\ \mathtt{AbelianGroup}, \mathtt{Monoid}\ \textbf{over}\ \mathtt{CarrierSet} \qquad (1)$$

where $I = \mathtt{Monoid}$. This yields the combination of AbelianGroup and Monoid that shares the logical symbols and the carrier set but contains two copies of the binary operation.

[6] Actually, this yields a pushout only if $\mathtt{dom}(X^\flat) \setminus \mathtt{dom}(S^\flat)$ and $\mathtt{dom}(T^\flat)$ are disjoint.

[7] Then this yields a pushout only if $\mathtt{dom}(T^\flat) \setminus \mathtt{dom}(I^\flat)$ and $\mathtt{dom}(T'^\flat) \setminus \mathtt{dom}(I^\flat)$ are disjoint. Moreover, when constructing the pushout, we have to duplicate the non-shared declarations of I, i.e., the ones for the identifiers in $\mathtt{dom}(I^\flat) \setminus \mathtt{dom}(S^\flat)$; this requires generating a fresh set of identifiers for them.

General Colimits Empty theory, union, and pushout are special cases of colimits in the category of theories. For many languages (e.g., first-order logic), this category is cocomplete, i.e., all colimits exist.[8] Therefore, it makes sense to introduce a single operator for colimits (and one for the universal morphisms out of them). However, there are two difficulties.

Firstly, colimits take a diagram as their argument, i.e., a multi-graph of theories and morphisms. Even though straightforward conceptually, it is difficult to design a human- and machine-friendly syntax for diagrams.

Secondly, colimits are only unique up to isomorphism, and many characteristic properties are stated as isomorphisms, not as equalities between theories. But the constructive semantics must make a concrete choice for the identifiers in the colimit (especially if we want to implement flattening). For example, in (1), `AbelianGroup` and `Monoid` share the binary operation $\circ$, but we need two different identifiers in the pushout. Thus, even if we know that *a* colimit exists, there may not be a canonical choice to call *the* colimit. Therefore, systems must either generate fresh identifiers, or require/allow users to supply identifiers as in

$$\textbf{pushout}\ \texttt{AbelianGroup}[\circ \rightsquigarrow +], \texttt{Monoid}[\circ \rightsquigarrow \cdot]\ \textbf{over}\ \texttt{CarrierSet}$$

This can be very tedious if many renamings are necessary so that qualified identifiers are useful as in

$$\textbf{pushout}\ \texttt{AbelianGroup}\ \textbf{as}\ \texttt{add}, \texttt{Monoid}\ \textbf{as}\ \texttt{mult}\ \textbf{over}\ \texttt{CarrierSet}$$

Either solution destroys coherence, i.e., different constructions of the same colimit may result in isomorphic but unequal theories. This can be extremely harmful because tools must check the relations $S \equiv T$ and $S \hookrightarrow T$ very often, e.g., to remove duplicates from the list of imported theories. Thus, we have to make the colimit operator partial in a way that guarantees coherence (as done in [Rab14] for pushouts in Mmt) or use a complex calculus that keeps track of isomorphisms.

5.4 Operators Specific to Morphisms

As seen above, most theory operators come with characteristic morphisms into or out of the constructed theories. This is not surprising: If we use theories-as-types, the morphisms correspond to the introduction and elimination forms of the respective theory operator.

However, two operators are unique to morphisms and provide the basic structure of the category of theories. $id_T : T \to T$ is the *identity* morphism of T, and $id_T^\flat = \{c := c \mid c \in \texttt{dom}(T^\flat)\}$. And if $M : R \to S$ and $N : S \to T$, then $N \circ M : R \to T$ is the *composition* morphism, and $(N \circ M)^\flat = \{c := N(M(c)) \mid c \in \texttt{dom}(R^\flat)\}$.

[8] For the practically relevant case of colimits of *finite* diagrams, this already follows from the existence of empty theory and pushout.

5.5 Theory Polymorphism

Many mathematical statements quantify over all theories or all theories of a certain shape. Most importantly, universal algebra permanently works with an arbitrary algebraic theory. We speak of *theory polymorphism* if variables can range over all theories, and of *bounded theory polymorphism* if they can range over all theories that extend a given theory. For example,

$$\texttt{Abelian}(T \hookleftarrow \texttt{Magma}) = T \,\textbf{with}\{\texttt{commutative} : \forall x, y. x \circ y = y \circ x\}$$

adds the commutativity axiom to any theory T that extends `Magma` (and thus has a binary operator $\circ$).

Languages that use theories-as-types can offer (bounded) theory polymorphism as a special case of (bounded) type polymorphism: $\lambda T <: B.K$ takes a type parameter T that is required to be a subtype of the bound B. Then we can use $T <: \texttt{Magma}$ because subtyping of theories corresponds to inclusion.

A major difficulty is that many theory operators are partial (e.g., union or pushout), and well-formedness depends on the domains of the argument theories. For example, $\texttt{Abelian}(T)$ is only well-formed if $\texttt{commutative} \notin \texttt{dom}(T)$. But we do not know $\texttt{dom}(T)$ if T is a variable, which makes it difficult to statically check expressions with theory polymorphism.

But even more complex forms of theory polymorphism are necessary to elegantly formalize even elementary examples. In particular, we often need to iterate over the list of declarations in a theory and define the cases of the iteration by pattern matching on the shape of the declaration. The following example defines the theory of homomorphisms φ between models m, m' of an arbitrary first-order theory T:

$$\begin{array}{l}
\texttt{Hom}(T \hookleftarrow \texttt{FOL}, m : T, m' : T) = \{ \\
\quad \varphi : m.\,\texttt{universe} \to m'.\,\texttt{universe}, \\
\quad \textbf{iterate}\, T \\
\qquad \textbf{case}\, f : \texttt{FunSymb}(n)\, \textbf{put}\, f : \forall x : m.\,\texttt{universe}^n \,.\, \varphi(m.f(x)) \doteq m'.f(x\,\texttt{map}\,\varphi) \\
\qquad \textbf{case}\, p : \texttt{PredSymb}(n)\, \textbf{put}\, p : \forall x : m.\,\texttt{universe}^n \,.\, m.p(x) \Rightarrow m'.p(x\,\texttt{map}\,\varphi) \\
\}
\end{array}$$

This assumes `FOL` has been defined using the formalism of [Hor14] with declaration patterns `FunSymb` and `PredSymb` for n-ary function and predicate symbols.

6 Selected Formal Systems with Theory Expressions

6.1 Logic-Independent Languages

Generic representation languages such as OpenMath [BCC+04] and Mmt [RK13] use external theories with declaration-based formation. The current Mmt implementation also supports parametric theories and morphisms, identity, composition, empty, union, pushout along inclusion, and anonymous morphisms.

6.2 Specification Languages

Structured specification languages [SW83,SST92] were systematically designed to provide a set of logic-independent primitives for constructing external theories with denotational semantics. Logic-independence is achieved by using an arbitrary institution [GB92] to provide the basic theories and models. Because institutions abstract from the inner structure of theories and objects, it is not possible to define all operators from Sect. 5 at this level. The central operators are union, translation (a special case of pushout of inclusion), and general hiding.

OBJ3 [GWM+93] uses external theories and morphisms with constructive semantics. It supports theories with theory-parameters, parametric morphisms (not implemented though), renaming, and union (called *sum*).

CASL [CoF04] provides the operators of ASL as well as anonymous theories, extension, and theories and morphisms with theory-parameters. Hiding is restricted to inclusions, which allows implementations (e.g., Hets [MML07]) to use constructive semantics with hidden declarations. Basic morphisms are restricted to assignments of the form $c := c'$.

Specware [SJ95] uses external theories (called *specifications*) and morphisms with constructive semantics. It supports anonymous theories and morphisms, renaming (called *translation*), pushout of inclusion (called *substitution*), and general colimits. The latter take diagram expressions as arguments, which are given as a list of nodes and edges labeled with theory and morphism expressions, respectively. Specware also uses internal theories via record types.

MathScheme [CFO10] is not intended as a specification language, but its results regarding theory expressions [CO12] are similar to the above languages. It uses external theories with constructive semantics and supports anonymous theories, empty theory, renaming, extension, pushout (called *combination*), and composition. Except for renaming in the arguments of a pushout, theory expressions cannot be nested.

6.3 Proof Assistants

All proof assistants use constructive semantics.

Isabelle [Pau94] uses three kinds of external theories (called *theory*, *type class*, and *locale*), and external morphisms between some of them (called *interpretation*, *sublocale*, or *subclass*). Theories are mostly used for namespace management, and type classes can be seen as special cases of locales; thus, locales are the best analogue to our theories. Locale expressions are extensions of unions of locale instances, where locale instances extend basic locales with definitions.

PVS [ORS92] uses external theories with theory-parameters. It supports pushout of inclusion via anonymous morphisms (called *interpretation*).

Coq [Coq14] uses two independent features that support theories. Firstly, it uses external theories (called *modules*) evolved from the ML module system. Secondly, it uses a variant of record types that (although not systematically intended as such) has the flavor of internalized theories. Record types are declared (only) by a toplevel macro, which is expanded into an inductive type with a

single constructor. Moreover, record types may be parametric, and a declaration in a record type may carry definiens and notation.

This makes Coq one of the few languages that provide strong support for both external and internal theories. The experience of Gonthier's formalizations of algebra [GGMR09] confirm that internal theories are more flexible.

Moreover, dependent type theories with universes (like the one underlying Coq) are rich enough for an even more expressive variant of internalized theories. Here theories are represented as a type I of identifiers and a function $I \to \texttt{type}$ mapping every identifier to its type. This yields a type of theories and thus allows theory polymorphism. This was applied to universal algebra in [Cap99]. But such theories are even weaker as an abstraction mechanism than record types.

Mizar [TB85] uses an abbreviation for introducing toplevel record types (called *structs*) that is similar to the one in Coq. It uses a special set of identifiers and expands records into functions out of that set. Moreover, Mizar allows representing axioms as unary predicates (called *attributes*) on these record types, and the extension of a theory with an axiom (called *mode*) as a predicate subtype. A sophisticated concrete syntax hides the expansion and makes the internal theories appear similar to external ones.

HOL Light [Har96] inherits external theories from the OCaml toplevel within which it is implemented. As such it inherits the theory expressions of OCaml.

6.4 Programming Languages

These mostly use constructive semantics with hidden declarations.

SML [MTHM97] uses external theories and (ground) morphisms as described in Sect. 3.1. It supports anonymous theories, extending with a definiens (the **where** keyword), and hiding declarations (via *structural matching*). Only type declarations in signatures may carry a definiens. Other ML variants such as OCaml use similar module systems.

MixML [RD13] redesigns the ML module system. It uses anonymous theories (called *modules*), parametric theories (called *suspension*), and dependent mixin composition (called *linking*) as primitives.

Java [GJJ96] uses internalized theories as described in Sect. 3.1. It supports extension and union with overriding. However, only the first theory in a union may contain definitions (the others are called *interfaces*). Theories and (ground) morphisms may use bounded polymorphism (called *generics*).

Scala [OSV07] gently redesigns the Java class system. It additionally supports anonymous theories and mixin composition (using the keyword **with**). Because it supports higher-order functions, we can form the type $T \Rightarrow S$ of morphisms from S to T; therefore, parametric (ground) morphisms can be plain functions.

7 Conclusion

We showed a large variety of theory operators along with examples of mathematical theories, whose elegant formalization depends on these operators. Moreover, we summarized which operators are supported by state-of-the-art formal

languages. We observe that none of these offer a satisfactory combination of expressive power and reasoning support.

Based on our survey, we consider the following requirements crucial in the design of future theory expression languages.[9]

Theories should be language-independent. A major problem in formalized mathematics is the incompatibility between languages. But languages are highly compatible regarding (basic) theories. Thus, if we develop theory expressions language-independently, we can use them as interfaces between different formalization systems as well as between formalization and MKM systems.

Theories should be distinguished ontologically. It is tempting to identify theories and objects, e.g., by using internal theories. While that makes many developments simpler, it deemphasizes some the key properties that motivated using theories in the first place: Abstraction, inheritance, and reuse along morphisms should be special operations.

The theory language should be extensible. It appears hopeless to fix a small, expressive subset of theory operators once and for all. Indeed, even if many other operators are definable, these definitions themselves can usually not be stated in the formal language. For a language with user-definable theory operators, we need (at least) support for functions taking and returning theories.

Theories should be internalized. This is the most promising approach to combine the rich expressivity of the object level with the ontological simplicity of the theory level. There is currently no logical language that systematically supports it.

References

BCC+04. S. Buswell, O. Caprotti, D. Carlisle, M. Dewar, M. Gaetano, and M. Kohlhase. The Open Math Standard, Version 2.0. Technical report, The Open Math Society, 2004. See http://www.openmath.org/standard/om20.

Bou74. N. Bourbaki. *Algebra I*. Elements of Mathematics. Springer, 1974.

Cap99. V. Capretta. Universal Algebra in Type Theory. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics*, pages 131–148. Springer, 1999.

CFO10. J. Carette, W. Farmer, and R. O'Connor. The mathscheme project, 2010. http://www.cas.mcmaster.ca/research/mathscheme.

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

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

Coq14. Coq Development Team. The Coq Proof Assistant: Reference Manual. Technical report, INRIA, 2014.

FGT92. W. Farmer, J. Guttman, and F. Thayer. Little Theories. In D. Kapur, editor, *Conference on Automated Deduction*, pages 467–581, 1992.

[9] These requirements may be a little subjective. But we hold that none of them should be dismissed without good reason.

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

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

GJJ96. J. Gosling, W. Joy, and G. Steele Jr. *The Java Language Specification*. Addison-Wesley, 1996.

GR04. J. Goguen and G. Rosu. Composing Hidden Information Modules over Inclusive Institutions. In O. Owe, S. Krogdahl, and T. Lyche, editors, *From Object-Orientation to Formal Methods, Essays in Memory of Ole-Johan Dahl*, pages 96–123. Springer, 2004.

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

Har96. J. Harrison. HOL Light: A Tutorial Introduction. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 265–269. Springer, 1996.

Hor14. F. Horozal. *A Framework for Defining Declarative Languages*. PhD thesis, Jacobs University Bremen, 2014.

MML07. T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editor, *Tools and Algorithms for the Construction and Analysis of Systems 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522, 2007.

MTHM97. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML,* Revised edition. MIT Press, 1997.

ORS92. S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, pages 748–752. Springer, 1992.

OSV07. M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. artima, 2007.

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

Rab14. F. Rabe. How to Identify, Translate, and Combine Logics? *Journal of Logic and Computation*, 2014. doi:10.1093/logcom/exu079.

RD13. A. Rossberg and D. Dreyer. Mixin' Up the ML Module System. *ACM Transactions on Programming Languages and Systems*, 35(1), 2013.

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

RKS11. F. Rabe, M. Kohlhase, and C. Sacerdoti Coen. A Foundational View on Integration Problems. In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 107–122. Springer, 2011.

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

SST92. D. Sannella, S. Sokolowski, and A. Tarlecki. Toward Formal Development of Programs from Algebraic Specifications: Parameterisation Revisited. *Acta Informatica*, 29(8):689–736, 1992.

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

TB85. A. Trybulec and H. Blair. Computer Assisted Reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28. Morgan Kaufmann, 1985.