Global, Regional, and Local Contexts

Florian Rabe

University Erlangen-Nuremberg

Abstract. We introduce UniFormal, a formal language for mathematical knowledge representation that continues the evolution of OpenMath and MMT. Its core feature is a separation of identifiers and associated contexts into three levels: The global context maintains the usual toplevel declarations that can be imported across libraries and nested packages via qualified names. The local context maintains the usual α-renamable bound variables with narrow syntactic scope. The novel regional context sits in between these and maintains identifiers whose scoping behavior combines aspects of the other two levels, in particular allowing the representation of theories and term languages as regions. In particular, a stack of regional contexts is needed to allow for expressions that move between regions, e.g., when applying a theory morphism to transport an expression from one theory to another.
We present the a minimal language of its kind, focusing on the general intuitions underlying the design. We anticipate it to be extended flexibly towards specialized aspects of mathematical knowledge such as programming languages or theorem provers to enable interoperability through their shared core concepts. This approach has already proved successful in the design and implementation of a UniFormal programming language.

1 Introduction

OpenMath [BCC$^+$04] and MathML [ABC$^+$03] were introduced as universal representation formats for mathematical objects. OMDoc [Koh06] extended them to mathematical theories and documents. Mmt [RK13] specified the semantics of, implemented, and significantly evolved OMDoc.

All of them were designed to be foundation-independent, and the Mmt tool offers a flexible library for rapid prototyping and system interoperability. Mmt has proved remarkably resilient in light of retroactive extensions that changed fundamental design assumptions (e.g., [HKR12],[WKR17]). However, recently the author's attempt at improving the handling of identifiers in Mmt have been frustrated repeatedly, to the point of essentially grinding Mmt development to a halt since 2023.

The crux was twofold. Firstly, there was a conflict between what this paper will call *open* and *closed* theories. OpenMath content dictionaries are open, OMDoc tries to handle both without making the distinction clear, and Mmt theories are closed. UniFormal is able to systematically accommodate without complicating the language design. Secondly, both OpenMath and Mmt support *global* identifiers (symbols, OMS) and *local* ones (variables, OMV). But neither can handle certain identifiers that combine properties of both, such as the names fields of mathematical structures or the names of the generators in a freely generated structure. After agonizing about this problem for years, UniFormal solves it with a system of three levels of identifiers, calling the novel intermediate ones *regional* identifiers.

Concretely, we present UniFormal as a simple framework language for representing formal theories and models. We leave the base language open, allowing for individual instances of UniFormal to inject suitable type and proof systems and operational semantics. One such instance, a UniFormal Programming Language, has already been developed and has been used to type-check the examples in this paper.

All design choices are the results of a years-long trial and error process that hit numerous dead ends, and a key achievement is the simplicity of the resulting abstract syntax. While we have developed (and implemented) a rigorous semantics for UniFormal, this paper employs an intuition-focused presentation style with many side remarks, examples, and comparisons to other languages. Thus, this paper primarily explains and motivates the syntax itself, and it only presents enough of the formal details to make the intended semantics visible.

2 UniFormal Syntax

The core grammar and judgments of UniFormal are given in Fig. 1 (where [optional] and repeated* parts are marked as usual) and 2. **Theories** T are expressions that normalize to flat theories $\{\vec{D}\}$, i.e., list of declarations, written $\overline{T}^{\Gamma}$ where Γ is the context in which T occurs. **Declarations** are of three kinds:

- Theory declarations **theory** $\tau = T$ give a name to a nested theory T (which in practice is usually a flat theory).
- Symbol declarations introduce a named object such as a type, term, axiom, inference rule etc.
- Includes **include** T create an inheritance relation between theories. Normalization eliminates them by merging all declarations of the included theory T into the including theory.

The nesting of theories yields a tree of declarations with theories at the inner nodes and symbol declarations (or includes, if not normalized yet) at the leaves. Note that all declarations can have a corresponding expression as a definition (mandatory for theories) and all but includes are **named**.

Expressions are anonymous transient syntax trees that are checked and interpreted against a context Γ. The details of contexts are explained later—at this point it suffices to remember that Γ declares all identifiers that expressions can use. Formal systems typically have multiple kinds of named declarations and corresponding kinds of expressions, with the identifiers referencing the former occurring as leaves of the latter. Typical kinds are type symbols and types, function symbols and terms, predicate symbols and formulas, or axioms and proofs.

Declarations

$D ::= [\textbf{open}]\ \textbf{theory}\ \tau = T$	theory (open or closed)
$\mid\ [\textbf{total}]\ \textbf{include}\ T = [e]$	include
$\mid\ \textbf{type}\ a = [A]$	type symbol, with definiens
$\mid\ \textbf{term}\ c : A = [t]$	term symbol, with type, definiens

Theory Expressions

$T ::= \tau \mid \tau.\ldots.\tau.\tau$	regional/global identifier
$\mid\ e.T$	pushout of T along model e
$\mid\ \{D^*\}$	anonymous theory

Types

$A ::= x \mid a \mid \tau.\ldots.\tau.a$	local/regional/global identifier
$\mid\ e.A$	interpretation of A in model e
$\mid\ Mod(T)$	models of T
$\mid\ \blacktriangleright t$	proofs of $t : \texttt{Bool}$
$\mid\ \texttt{Bool} \mid \texttt{Int} \mid A^* \to A \mid \ldots$	example base types

Terms

$e, t ::= x \mid c \mid \tau.\ldots.\tau.c$	local/regional/global identifier
$\mid\ e.t$	interpretation of t in model e
$\mid\ mod(T)$	a concrete model
$\mid\ \uparrow^n$	model of n-th enclosing region (for $n \in \mathbb{N}$)
$\mid\ \ldots$	other productions as needed

Contexts

$\Gamma ::= D\ ([\vert]\ T\ L)^*$	global+stack of (regional+local)
$L ::= D^*$	local context of bound variables

Fig. 1. UniFormal Syntax

Except for fixing theories and theory expressions, UniFormal is agnostic in the choice of these kinds. For for the sake of example, we pick

- **Type** declarations $\textbf{type}\,a = [A]$ introduce a named type with an optional definition A. We also allow for type bounds and type operators but omit that here for simplicity.
- **Term** declarations $\textbf{term}\,c : A = [t]$ introduce a named term with type A and optional definition t. The type is mandatory in internal syntax, but can be omitted in surface syntax if it can be inferred.
- Axioms and theorems are special cases of terms with resp. without definition via the type $\blacktriangleright F$ of proofs of Boolean term F.

Instances may change these kinds as needed, but UniFormal requires that every symbol declaration may optionally carry an expression of the corresponding kind as its definition. If such a definition is present, we call the declaration **concrete**,

Judgment	Intuition
$\Gamma \vdash T$ THY	well-formed theory
$\Gamma \vdash S \hookrightarrow T$	inclusion between theories
$\Gamma \vdash S \equiv T$	equality (= inclusion in both directions)
$\Gamma \vdash A$ TYPE	well-formed type
$\Gamma \vdash A <: B$	subtyping
$\Gamma \vdash A \equiv B$	equality (= subtyping in both directions)
$\Gamma \vdash t : A$	well-formed typed terms
$\Gamma \vdash s \equiv t$	equality of terms

Function	Intuition
$\overline{T}^{\Gamma}$	normal form of theory T: a list $\vec{D}$ of declarations
$S \ll d$	merge declaration d into theory S
$\Gamma + T$	extend context Γ with a new frame for theory T

Fig. 2. UniFormal Judgments and Auxiliary Functions

otherwise **abstract**. Similarly, a theory is called *concrete* if all declarations in its normal form are, otherwise **abstract**

Example 1. The algebraic hierarchy (the Hello-World of mathematical module systems) starts with

$$\begin{array}{l}\textbf{theory}\, \mathit{Carrier} = \{\textbf{type}\, u\},\\ \textbf{theory}\, \mathit{Magma} = \{\textbf{include}\, \mathit{Carrier},\ \textbf{term}\, \mathit{op} : (u,u) \to u\}\\ \textbf{theory}\, \mathit{Semigroup} = \{\textbf{include}\, \mathit{Magma},\\ \quad \textbf{term}\, \mathit{assoc} :\blacktriangleright \forall x,y,z : u.\mathit{op}(x, \mathit{op}(y,z)) = \mathit{op}(\mathit{op}(x,y),z)\}\end{array}$$

Here and in subsequent examples, we assume an instance of UniFormal in which all the relevant constructors for types and terms are present such as $\to$, λ, $-(-)$ for functions, and the usual connectives and typed quantifiers like $\forall$ and $=$.

Local contexts L declare bound variables like $x : u, y : u, z : u$ above. The grammar uses the same syntax for them as for theories, but they may actually contain only term and type variables. Moreover, we always drop the keyword **term** for bound variables in examples.

3 Open vs. Closed Theories

3.1 Distinction

Formal systems often provide multiple kinds of theory-like features with different semantics. UniFormal distinguishes only two kinds, capturing what is arguably the most consequential difference: We call a theory **open** if its declarations are freely accessible from the outside, otherwise **closed**. For example, in **open theory** $\tau = \{\textbf{term}\, c : A\}, \textbf{term}\, d = \tau.c$, the symbol c of the open theory τ is referenced using a qualified identifier. If τ were closed, such a reference would

system	open	closed
Isabelle	theory	locale, type class
Rocq	package	module, type class, record
Mizar	article	structure
sTeX	module	mathstruct
OpenMath/MathML	content dictionary	—
MMT	—	theory
SageMath	module	category
Axiom	—	category
Java	package	class
C++	namespace	class, struct

Fig. 3. Open and Closed Theories in Various Systems

be ill-formed. When using closed theories C, languages must provide other features that regulate access to them. Many formal systems (see Fig. 3) use open and/or closed theories, sometimes in multiple variants and usually with a less clear-cut difference.

Our distinction of closed/open theories is analogous to the well-known **open/closed world assumption**. An open theory is effectively just a namespace. The set of inhabitants of the open world (in our case: the expressions over the open theory) is open to future changes: Symbols can be added, deleted, or moved between open theories at will, incurring “only” the cost of updating qualified references to them. Consequently and critically: Induction on the set of inhabitants is not allowed because it would be broken by extension.

A *closed* theory is like an open one except that it is “closed off” in the sense that induction on the set of inhabitants is allowed. The most important examples are the models of C (where induction is used to interpret all expression in the model), the language of C (where induction is used for induction on expression), or Prolog-style querying and negation-by-failure (where induction is used to exhaustively search all expressions). We focus on the models in the sequel.

The main purpose of open theories is to provide namespaces both inside files and across files as when structuring projects into packages and folders. In UniFormal, we assume the current project, with all its imports, to be provided as a single open theory G containing all available toplevel declarations. G is the root of a tree, whose inner nodes are nested open theory declarations (representing packages, folders, namespaces, etc.) and whose leafs are symbol and closed theory declarations.

Remark 1 (Imports). When packages are published in central repositories, languages usually provide **import** statements to tell the tool which open theories are going to be used. These may have some additional non-trivial semantics, e.g., making symbols available via unqualified names, renaming symbols, or selecting only some symbols of an open theory to import. But their main role is to provide extra-logical package management that builds the *open* theory that a project depends on. In particular, imports are very different from includes, which make *closed* theories available. UniFormal itself does not feature imports although particular instances typically will.

3.2 Models of a Theory

Consider a closed theory C that normalizes to a list of term/type declarations for names s_i. A **model** m of C is an object that provides an interpretation for every s_i. We can think of C and m as

- a record type and a record value,
- a logical theory and a model for it,
- a specification/interface and an implementation.

Note that this only works if C to be closed: changing the symbols of C affects whether a model is well-formed.

Remark 2 (Are Theories Types?). In OO-languages like Java, both the theory and its models can be represented as classes, usually called *abstract* and *concrete* classes. (Technically, we must distinguish between the concrete class and its instances. However, this distinction only becomes relevant if C contains mutable fields, which we do not consider here.) Moreover, the closed theories (i.e., the classes) double as the type of models. Alternatively, e.g., in SML, the two are strictly distinguished with models of a theory corresponding to structures implementing a signature. Structures and signatures are two additional kinds of symbols with a separate typing relation. In some languages, both styles of closed theories exist, e.g., Isabelle has locales on a separate level [KWP99] and records as types; Rocq similarly has modules and records [Coq15]. In [MRK18], we discuss this in detail and speak of *internalized theories* if there is a *type* of models of C.

In UniFormal, $Mod(C)$ is the type of models of C, and $mod(T)$ for a concrete theory T is a term of such a type. For example, $Mod(Magma)$ is the type of magmas, and $mod(\textbf{include}\, Magma,\ \textbf{type}\, u = \texttt{Int},\ \textbf{term}\, op = \lambda x, y.x + y)$ is a model of it. Thus, closed and concrete theories give rise to type and terms. To type-check $mod(T)$, we need to check that T indeed defines every field of C and does not conflict with any definitions already present in C. That is one purpose of the inclusion judgment (which we define in Sect. 4):

$$\frac{\Gamma \vdash C\ \mathrm{THY}}{\Gamma \vdash Mod(C)\ \mathrm{TYPE}} \qquad \frac{\Gamma \vdash C \hookrightarrow T \qquad T \text{ is concrete}}{\Gamma \vdash mod(T) : Mod(C)}$$

The elimination forms take a model $m : Mod(C)$ and project out C-symbols s, commonly written as $m.s$. We generalize this syntax to allow applying m to an arbitrary C-*expression*. This corresponds to the application of the interpretation function that maps C-expressions to their semantics in m:

$$\frac{\Gamma \vdash m : Mod(C) \qquad \Gamma + |C \vdash A\ \mathrm{TYPE}}{\Gamma \vdash m.A\ \mathrm{TYPE}} \qquad \frac{\Gamma \vdash m : Mod(C) \qquad \Gamma + |C \vdash t : A}{\Gamma \vdash m.t : m.A}$$

$$\frac{\Gamma \vdash m : \mathit{Mod}(C) \qquad \Gamma + |C \vdash S \text{ THY}}{\Gamma \vdash m.S \text{ THY}}$$

Here $\Gamma + |T$ extends the context with the declarations of C in order to build the C-expressions. The details are defined in Sect. 5.2.

The computational behavior of $m.-$ is that $m.s$ (for a symbol s) projects out the definition of the field of s in m:

$$\frac{\Gamma \vdash \mathit{mod}(T) : \mathit{Mod}(C) \qquad \textbf{type}\, a = A \text{ in } \overline{C}^{\Gamma}}{\Gamma \vdash \mathit{mod}(T).a \equiv \mathit{subs}^{\mathit{mod}(T)}(A)}$$

and accordingly for term and theory symbols For composed expressions, $m.-$ behaves like a homomorphic extension, e.g., $\Gamma \vdash m.(A \to B) \equiv m.A \to m.B$ and $\Gamma \vdash m.\texttt{Int} \equiv \texttt{Int}$. The grayed out operations replaces any occurrences of $\uparrow^0$ (which refers to the current region) with $\mathit{mod}(T)$. It is defined in the appendix.

Remark 3 (The Type of Types). The combination of abstract type fields and a type of models allows defining the *type of types* as $\mathit{Mod}(\{\textbf{type}\, \mathit{univ}\})$. This raises the question of consistency when giving a semantics because higher universes may be needed to hold this type. Both set theoretical systems like Mizar [TB85] and type theoretical systems based on Martin-Löf type theory [ML84] have developed solutions. In programming languages, having the type of types can be harmless. In several OO-languages like Scala it simply exists, and one has to dive deep into the language semantics to see where and how any issues are handled.

We ignore the resulting issues here. In fact, we are not even sure if a universal representation language should address the issue at all or if it should leave it open to avoid over-committing to any one solution.

Remark 4 (Dependent Types). The combination of abstract type fields and the type of models also allows creating dependent types (in the sense of type expressions with terms as subexpressions) as in

$$\textbf{theory}\, \mathit{Vector} = \{\textbf{term}\, \mathit{length} : \texttt{Int},\ \textbf{type}\, u\}$$

which allows using $\mathit{mod}(\textbf{include}\, \mathit{Vector}, \textbf{term}\, \mathit{length} = n).u$ as the type of vectors of length n. More generally, $\mathit{Mod}(T)$ is a dependent type whenever T contains a term definition, and so is $m.A$ for a type A. Therefore, UniFormal instances must support dependent types.

3.3 The Term Language of a Theory

Models are not the only way to utilize closed theories. We sketch term language types because they can be seen as a dual to model types. For example, consider the theory N given by $\{\textbf{type}\, n, \textbf{term}\, z : n, \textbf{term}\, s : n \to n\}$ (assuming we have function types $\to$). Its term language is that of the natural numbers. More generally, a family of mutually recursive inductive types or a context-free grammar (using one type symbol per non-terminal) can be represented elegantly like this.

Constructing objects through term languages is commonly done, albeit usually under a different name: It includes inductive types, Herbrand models, free objects over some generators such as rings of polynomials, the set of syntax trees over a grammar, or the set of derivations over an inference system.

We leave the formal details to define term language types to future work. But our definitions anticipate and our implementation already supports them: the type $T[A]$ holds the normal terms over theory T of type A. Induction on the type $T[A]$ is possible but only from outside of T: e.g., in $\{Nat\}N, \textbf{term}\, even : Nat[n] \to \texttt{Bool} = \dots$ the definition of *even* may perform induction on the term language of N, but no such induction is allowed inside N.

4 Inclusion of Theories

4.1 Definition

Intuitively, every **include** S occurring in T copies all declarations of S into T. S must be closed because: including an open theory would be redundant because we can already refer to their symbols anyway. In terms of theory morphisms, the include declaration induces an inclusion morphism from S to T.

Precisely, for theory expressions S, T, we define: $\Gamma \vdash S \hookrightarrow T$ holds iff every declaration in $\overline{S}^{\Gamma}$ is also part of $\overline{T}^{\Gamma}$ in the following sense:

- If the former has a term symbol c of type A [with definition t], then the latter has a term symbol c of type A' such that $\Gamma + T \vdash A <: A'$ [and with definition t' such that $\Gamma + T \vdash t \equiv t'$].
- If the former has a type symbol a [with definition A], then the latter has a type symbol a [with definition A' such that $\Gamma + T \vdash A \equiv A'$].
- If the former has a theory symbol σ with body Σ, then the latter has one with the same body.

Clearly, this relation is reflexive and transitive, which is critical for efficient implementations. Moreover, we define **equality of theories** as inclusion in both directions, thus making inclusion an order.

The normalization of theories (see Sect. 4.3) is defined in such a way that include declarations do indeed induce a morphism: We can prove that if $\Gamma \vdash T$ THY and the body of T contains **include** S, then $\Gamma \vdash S \hookrightarrow T$.

4.2 Relation to Subtyping

Inclusion induces **subtyping** between the types of models:

$$\frac{\Gamma \vdash S \hookrightarrow T}{\Gamma \vdash Mod(T) <: Mod(S)}$$

Informally, if T is bigger than S, than defining all of T implies defining all of S. This inclusion=supertype relationship is common in OO-languages where T is said to **inherit** from S. Alternatively, languages may require applying an explicit forgetful functor to turn a model of T into a model of S, e.g., with

Rocq or Isabelle records, or infer the forgetful functor like Mizar [TB85]. Here we adopt the former view and consider it a subtype.

Even though we do not formally define term language types here, we mention the anticipated subtyping rule because of its symmetry: If $\Gamma \vdash S \hookrightarrow T$, then the term language of S form a subtype of the term language of T, i.e., term language types and model types behave dually.

Remark 5 (Decidability). Whether or not typing is decidable, depends on what other types are added in a particular UniFormal instance. Because equality of types and terms is critical to determine inclusion, and inclusion is critical to determine subtyping, equality of types and terms must be decidable for a particular UniFormal instance to be decidable.

4.3 Union and Mixin of Theories

We do not need primitive syntax for the **union** of theories because we can simply define it as $S + T := \{\textbf{include}\, S,\ \textbf{include}\, T\}$.

More generally, we need a **dependent union** $S \ll \vec{D} := \{\textbf{include}\, S,\ \vec{D}\}$ to extend S with declarations $\vec{D}$ that may already use the symbols of S. Its normal form is obtained by iteratively adding one declaration in $S \ll D_1 \ll \ldots \ll D_n$. For example, we have $\{\textbf{type}\, a\} \ll \textbf{term}\, c : a \ \equiv\ \{\textbf{type}\, a,\ \textbf{term}\, c : a\}$.

That raises the question of **merging declarations**, e.g., do we have that

$$\{\textbf{term}\, c : \texttt{Int}\} \ll \textbf{term}\, c = 1 \qquad \equiv \qquad \{\textbf{term}\, c : \texttt{Int} = 1\}$$

We define normalization in UniFormal with this intuition of merging declarations. This kind of dependent union with merging behavior is central to many module systems including the inheritance systems of OO-languages and mixin or trait-based module systems.

Theories are **normalized** by merging in every declaration individually. In particular, the semantics of **include** S is to compute the normal form of S and merge in all declarations.

Example 2. The theory *Group* is usually defined by including *Monoid*. But actually *Monoid* needs both neutrality axioms whereas *Group* needs only one. UniFormal can capture this by defining some fields of the included theory:

$$\begin{array}{l}
\textbf{theory}\, \mathit{Monoid} = \{\textbf{include}\, \mathit{Semigroup},\ \textbf{term}\, e : u \\
\quad \textbf{term}\, \mathit{neutL} :\blacktriangleright \forall x : u.\mathit{op}(e,x) = x,\ \textbf{term}\, \mathit{neutR} :\blacktriangleright \forall x : u.\mathit{op}(e,x) = x\} \\
\textbf{theory}\, \mathit{Group} = \{\textbf{include}\, \mathit{Monoid},\ \text{(axioms for inverse omitted)}, \\
\quad \textbf{term}\, \mathit{neutR} = \text{(now provable, proof omitted)}\}
\end{array}$$

Note that this can result in a normal form with two mutually recursive declarations, i.e., normal theories are *sets* rather than lists of declarations. A neat side effect of this is an easy handling of recursive functions:

Example 3 (Recursion). We define the mutually recursive functions *odd* and *even* (omitting the usual definitions) as the theory below, which normalizes to 2 mutually recursive declarations

$$\textbf{theory}\,\tau = \{\textbf{term}\,odd : \texttt{Int} \to \texttt{Bool},\ \textbf{term}\,even : \texttt{Int} \to \texttt{Bool} = \ldots,\ \textbf{term}\,odd = \ldots\}$$

Not all merges are well-formed, e.g., UniFormal rejects merges if two definitions for the same name are not equal as in $\{\textbf{term}\,c : \texttt{Int} = 1\} \ll \textbf{term}\,c : \texttt{Int} = 0$. If multiple types are merged, the least upper bound is computed, e.g., $\{\textbf{term}\,c : \texttt{Int}\} \ll \textbf{term}\,c : \texttt{Bool} \equiv \{\textbf{term}\,c : \texttt{Void}\}$, which triggers an error because c cannot have the empty type `Void`. For model types, the least upper bound is computed by taking the union of theories as $\{\textbf{term}\,c : Mod(S)\} \ll \textbf{term}\,c : Mod(T) \equiv \{\textbf{term}\,c : Mod(S+T)\}$. That corresponds to typical behavior in OO-languages.

Note that theory normalization is a necessary step of type-checking: a theory is well-formed if its normalization succeeds. It is undecidable if a theory is well-formed, but it if is, its normal form is statically known.

Remark 6 (Overloading). Some languages with inheritance, in particular OO-languages, allow for overloading: Then merging $\textbf{term}\,c : \texttt{Int}$ and $\textbf{term}\,c : \texttt{Bool}$ is not an error. Instead, it results in two different declarations that share the symbol name. This issue can be resolved if we assume that the official internal identifiers is a pair of c and (some normal form of) its type. This is, e.g., how official Java identifiers are formed to disambiguated overloaded names. Our language will not support overloading but could be extended to do so.

Remark 7 (Overriding). Overriding arises if two conflicting definitions are present, e.g., if it is not an error for T to declare $\textbf{term}\,\texttt{Int} : 1 =$ and $\textbf{term}\,c : \texttt{Int} = 0$. Many OO-languages allow this, typically when the former is inherited, say from S, and the latter is local in T. Then the local definition overrides the inherited one silently (as in Java) or only if an "override" keyword is present (as in Scala). Either way, this is problematic because the inclusion from S to T is now broken in the sense that the original definition from S is no longer valid in T. This throws every proof carried out in S in question because it might have relied on the old definition. In terms of theory morphisms, in the presence of overriding, an include declaration does not induce an inclusion morphism anymore. Therefore, we reject overriding in UniFormal.

Remark 8 (Redefinition). A special case of overriding is can be allowed safely: if the overriding definition is extensionally equivalent to the overridden one. For example, if S is the theory of groups is included to form the theory T of commutative groups, a computer algebra system is well-advised to override some algorithms of S with more efficient versions for commutative groups. Similarly, Mizar uses redefinitions (albeit in open theories, rather than in closed ones) to switch the definition of an identifier to an equal one that is more helpful in the current context. Redefinitions do not suffer from the drawbacks of general overriding and could be added to our language, but we leave them to future work.

4.4 Total Includes

We define an inclusion $\Gamma \vdash S \hookrightarrow T$ to be **total** if every symbol declaration in $\overline{S}^{\Gamma}$ has a definition in T. Thus, a total include is conservative in the sense that S does not change the primitive symbols of T. It can also be seen as a special case of a theory morphism from S to T that maps every symbol of S to the corresponding definition in T. UniFormal provides two ways to make an include total.

Firstly, the keyword **total** triggers a totality check at the end of the containing theory. For example, in

$$\begin{aligned}&\textbf{theory}\, Monoid = \{\text{(as above)}, \textbf{total}\,\textbf{include}\, Preorder,\\ &\quad \textbf{term}\, r = \lambda x, y.\exists d.op(x, d) = y\}\end{aligned}$$

The keyword **total** asserts that $Monoid$ not just inherits from but *implements* the theory $Preorder$, which declares a relation $r : (u, u) \to \texttt{Bool}$. Note that all types in the definition of r can be inferred. The totality check at the end of $Monoid$ would fail because the proofs of the axioms for reflexivity and transitivity are still missing.

Secondly, an include can have a definition. $\textbf{include}\, C = e$ is a legal declaration in T iff $\Gamma + T \vdash e : Mod(C)$, i.e., the definition of an include must be a model of the included theory. $\textbf{include}\, C = e$ is equivalent to declaring $\textbf{total}\,\textbf{include}\, C$ followed by $\textbf{term}\, s = e.s$ resp. $\textbf{type}\, s = e.s$ for every term/type symbol s of C. In programming language terms, defined includes are **delegation**: If T already has a term e that implements the interface C, it can use a defined include to delegate all C-calls to e.

4.5 Guarded Theories and Conservative Extensions

Only closed theories make sense to be included. Accordingly, we might say that only closed theories may include others. But here it turns out, a small generalization yields a critical expressivity gain.

If $\textbf{open}\,\textbf{theory}\, \tau = \{\textbf{include}\, C, \ldots\}$ includes closed theory C, we call T **guarded** by C. Like for all open theories, access to symbols s of T is by qualified identifiers $\tau.s$ But now we define $\tau.s$ to be only well-formed only in contexts that also include all guards of C.

This provides a very easy way to maintain conservative extensions that add definitions and theorems to a closed theory C. Writing a new theory D that extends C for this purpose is problematic for three reasons: Firstly, any model $m : Mod(C)$ must be cast into a model of D to utilize the definition/theorem. Secondly, the normal forms (which must be computed frequently) of the closed theories become bigger, presenting serious scalability issues. Therefore, both IMPS [FGT93] and Isabelle locales [KWP99] allow adding defined symbols to C from the outside. But this is has a rather imperative flavor, and does not address the third issue: having many such extension by different authors increases the danger of name clashes when multiple extensions must be included at the same time.

UniFormal solves all issues by combining small closed theories that prioritize the abstract symbols with large guarded open theories that contain the extensions as in

$$\textbf{open theory}\, \mathit{Doubling} = \{\textbf{include}\, \mathit{Monoid},\ \textbf{term}\, \mathit{double} = \lambda x.\mathit{op}(x,x)\}$$

Now assume we have $m : \mathit{Mod}(\mathit{Monoid})$ and want to access the global identifier $g := \mathit{Doubling.double}$ on m. We can do this by simply writing $m.g$ requiring no further extensions of the syntax: the typing rule (as shown above) for $m.g$ checks g in context $\Gamma|+\mathit{Monoid}$ where the guard Monoid is available so that checking g succeeds.

5 Global vs. Regional vs. Local Level

Choosing the right data structure for the context can be as difficult as designing the entire formal system, which is why our presentation ends with treatment of contexts and identifiers. The table below gives an overview of the three levels of identifiers. These are orthogonal to the symbol *kinds* discussed above, i.e., we have global, regional, and local identifiers for terms, types, or theories.

level	global	regional	local
paradigm	toplevel declaration	field in class/record	bound variable
identifiers	qualified identifiers	names	variables
declared in	open theory	closed theory	block, binder
scoping	global	global but guarded	lexical, narrow
name clash handling	qualification	merging	shadowing
semantics	fixed	context-specific	place-holder

Fig. 4. The Levels of UniFormal Identifiers

5.1 Identifiers

The **global identifiers** are the ones declared in open theories. They have a qualified identifier rooted at the toplevel, via which they can be accessed from every context. Consequently, they cannot be α-renamed without breaking downstream content. Name clashes are disambiguated by putting symbols of the same name into different open theories resulting in different qualified names.

The semantics of a global identifier is fixed: Most global declarations include a definiens that explicitly states the meaning of the global identifier. If no definiens is present, the meaning is still fixed but unknown, e.g., a stub declaring only the type of a foreign function that is provided by the runtime environment.

The **local identifiers** are the bound variables. Their scopes are lexical and always contained within an expression. They are easily α-renamable, and name clashes are resolved by shadowing, i.e., each local identifier is resolved to the innermost binding of that name.

Local identifiers are usually introduced **declaratively** by a binder as in $\forall x : A.t$, in which case the scope of x is t. But UniFormal also allows **imperative** bindings: This uses block terms $(t_1, \dots, t_n)$, where each t_i may introduce variables whose scope is to the end of the block, as in $(\mathbf{term}\, x : \mathtt{Int} = 1,\ \mathbf{term}\, y : \mathtt{Int} = x + 1,\ f(y))$. More complex examples of imperatively binding terms are pattern-matching definitions like $\mathit{head} :: \mathit{tail} = l$ for some list l; dynamic binding as in discourse representation terms like $((\exists x : \mathtt{Int}.P) \Rightarrow B) \wedge C$ where x is visible in B but not in C [KK97]; and tactics in imperative proof languages (which inspired our naming) such as *intros* in Rocq that dynamically introduce identifiers.

Remark 9 (Theory Variables). We allow for term and type declarations in local contexts but not theory declarations. It is intriguing to have theory variables to formalize computations on theories like the ones in [SR19,CO12]. But it is an open problem how to do that best.

The global and local identifiers as described above are standard. The idea of **regional identifiers** is new: They are the symbols declared in closed theories, and they combine properties of global and local identifiers.

Consider a closed theory C given by $\{\mathbf{term}\, c : A, \dots\}$. Like local and contrary to global identifiers, there is no way to assign a qualified identifier to c (because C might appear anonymously), the symbol c acts as a placeholder for values that are to provided later, and α-renaming feels natural. But like global and contrary to local identifiers, c can be referenced from the outside, e.g., if we have a model $m : Mod(C)$, we can call $m.c$ to access its definition of c. Thus, c as a name has global scope, but its semantics varies with the theory in which it is declared. Finally, regional identifiers have a different name clash behavior from both local and global identifiers: a second regional declaration for the same name is merged into the first one.

Regional identifiers are not only used for types of models. For example, we anticipate their use for the constructors of inductive types or the generators of free data structures.

Remark 10. Named Record Types Some formal systems nudge or even force users to introduce record types only if they are tied to a global identifier. For example, Rocq features named record *declarations* of the form $\mathit{Record}\, R := \{a : A, b : B\}$. In this case, R and thus a and b can be taken as global identifiers. This avoids the need for regional identifiers, greatly simplifying the language, but preventing the construction of anonymous record types. This subtle design decision of named vs. anonymous record types is often barely noticeable for users, but has massive consequences for system development.

5.2 Contexts

We need a three-partite context declaring the global, regional, and local identifiers. For example, when considering the term t in

$$\mathbf{open\,theory}\,\tau_1 = \{\vec{D},\ \mathbf{open\,theory}\,\tau_2 = \{\vec{E},\ \mathbf{theory}\,\tau = \{\vec{F},\ \mathbf{term}\,c = \lambda x : A.t\}\}\}$$

the global context holds $\vec{D}$ and $\vec{E}$, the regional context $\vec{F}$, and the local context $\mathbf{term}\,x : A$. t can reference global declarations by, e.g., $\tau_1.\tau_2.s$ for a symbol s from $\vec{E}$, and references regional and local identifiers by name.

But we have to go one step further. Consider nested closed theories as in

$$\begin{array}{l}\mathbf{theory}\,2\mathit{Pointed} = \{\mathbf{type}\,a,\ \mathbf{theory}\,\mathit{Point} = \{\mathbf{term}\,c : a\},\\ \quad \mathbf{term}\,p : \mathit{Mod}(\mathit{Point}),\ \mathbf{term}\,q : \mathit{Mod}(\mathit{Point})\}\end{array}$$

Type-checking *Point* nested inside 2*Pointed* requires opening a new region for the symbol c, while keeping the region of 2*Pointed* with the symbol a accessible. Nested theories also occur every time an *anonymous* theory C is used in $\mathit{Mod}(C)$ or $\mathit{mod}(C)$. Therefore, we use a **stack of regions** in our final definition of context:

Definition 1 (Context). *A UniFormal context Γ is of the form $G;\vec{F}$ where*
- *global context: G is an open theory holding the available toplevel declarations*
- *$\vec{F}$ is a stack in which each frame $[|]\,R\,L$ consists of*
  - *transparency: an optional modifier $|$ that indicates whether a region can see identifiers from the frame before it,*
  - *regional context: a closed theory R (in normal form) holding the currently available regional identifiers,*
  - *local context: a list L of term/type declarations.*

Given such a Γ and a theory $\Gamma \vdash T$ THY, we define ***context extension*** *by*

$$\Gamma + T := G;\vec{F},\ T\,\{\} \qquad \Gamma + |T := G;\vec{F},\ |T\,\{\}$$

When traversing an expression, we may jump to a different region by pushing it onto the stack and may return by popping from the stack in a way akin to how programming languages use a stack frame for each function call. Reconsider the rule for checking the model type $\mathit{Mod}(C)$ from Sect. 3.2. To check $\Gamma \vdash \mathit{Mod}(\{\vec{D}\})$ TYPE, we check $\Gamma \vdash \{\vec{D}\}$ THY. To check the latter, we check each declaration D_i in the context $\Gamma + (D_1 \ldots, D_{i-1})$. Not using the $|$ modifier means that each D_i can see the outer regions.

In contrast, in the elimination rule for model types, which concludes with $\Gamma \vdash m.A$ TYPE, the hypothesis is $\Gamma + |C \vdash A$ TYPE. Here the $|$ modifier ensures that A can only see C-identifiers no matter in which context $m.A$ is used. That corresponds to $m.-$ representing the application of morphism m to a C-expression: a term $\Gamma \vdash m : \mathit{Mod}(C)$ is the same as a theory morphism from C to the innermost region of Γ.

Technically, we must actually write $\mathbf{theory}\,\mathit{Point} = \{\mathbf{term}\,c :\uparrow 1a\}$ above, where $\uparrow^1$ refers to the first enclosing region. That is necessary to differentiate if two nested regions define the same identifier.

Example 4. Consider the theory of bimagmas where two binary operations on the same carrier are present (like in a ring):

$$\begin{aligned}&\textbf{theory}\, BiMagma = \{\textbf{include}\, Carrier,\\ &\quad \textbf{term}\, add : Mod(\textbf{include}\, Magma,\ \textbf{type}\, u = \uparrow^1 .u)\\ &\quad \textbf{term}\, mult : Mod(\textbf{include}\, Magma,\ \textbf{type}\, u = \uparrow^1 .u)\}\end{aligned}$$

Here both $BiMagma$ and the nested regions in the argument of $Mod(-)$ declare unrelated regional identifiers u. In this particular example, we happen to want to identify them and therefore extend the additive and the multiplicative magma with **type** $u = \uparrow^1 .u$.

Finally, we can define the semantics of contexts and identifiers:

Definition 2 (Lookup in a Context). *Consider a context $\Gamma = G; F_l, \dots, F_1$ where each F_i is of the form $R_i\, L_i$ or $|\, R_i\, L_i$. Let k be the smallest number such that F_k uses the $|$ modifier. Then:*

- *A global identifier $\tau_1 . \dots . \tau_n . s$ resolves to the declaration for s that occurs in G in a sequence of nested open theories τ_i.*
- *A regional identifier s resolves to the declaration for s in R_1.*
- *The special identifier $\uparrow^j$ is well-formed if $j \leq k$, and then its type is $shift^j(Mod(R_j))$. In particular, $\uparrow^j .s$ resolves to the declarations for s in region R_j.*
- *A local identifier x resolves to $shift^j(d)$ where d is the last declaration for x in $L_k, \dots, L_1$ and d is found L_j.*

$shift^j(-)$ can be treated as the identity. It is defined in the appendix.

6 Conclusion

We have introduced the UniFormal language for an OO-style definition of mathematical theories and models. The key to obtaining a simple language that admits an elegant formal semantics was a novel notion of contexts that distinguishes global, regional, and local identifiers with a stack of regional contexts representing the nesting of regions.

Concrete instances of UniFormal can extend it towards specialized languages like logics, programming languages, or flexiformal languages, while enabling interoperability through a shared theory layer. .Most commonly used features have rules that do not interact with the context, except possible extending the context with local identifiers. Examples are types for dependent function and product types, and collection type like lists, sets, as well as the type $\vdash F$ of proofs of F, to represent proofs-as-terms.

We have already built one such instance: a UniFormal programing language[1] that features dependent function types, collection types, as well as mutable regional and local identifiers and the usual control flow operators.

[1] Source code and examples available at `https://github.com/UniFormal/UPL/`

References

ABC$^+$03. R. Ausbrooks, S. Buswell, D. Carlisle, S. Dalmas, S. Devitt, A. Diaz, M. Froumentin, R. Hunter, P. Ion, M. Kohlhase, R. Miner, N. Poppelier, B. Smith, N. Soiffer, R. Sutor, and S. Watt. Mathematical Markup Language (MathML) Version 2.0 (second edition), 2003. See http://www.w3.org/TR/MathML2.

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.

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.

Coq15. Coq Development Team. The Coq Proof Assistant: Reference Manual. Technical report, INRIA, 2015.

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

HKR12. F. Horozal, M. Kohlhase, and F. Rabe. Extending MKM Formats at the Statement Level. In J. Campbell, J. Carette, G. Dos Reis, J. Jeuring, P. Sojka, V. Sorge, and M. Wenzel, editors, *Intelligent Computer Mathematics*, pages 64–79. Springer, 2012.

KK97. M. Kohlhase and S. Kuschert. Dynamic lambda calculus. In *Meeting on Mathematics of Language*, pages 85–92, 1997. https://kwarc.info/people/mkohlhase/papers/dlc00.pdf.

Koh06. M. Kohlhase. *OMDoc: An Open Markup Format for Mathematical Documents (Version 1.2)*. Number 4180 in Lecture Notes in Artificial Intelligence. Springer, 2006.

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

ML84. P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

MRK18. D. Müller, F. Rabe, and M. Kohlhase. Theories as Types. In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *Automated Reasoning*, pages 575–590. Springer, 2018.

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

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

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.

WKR17. T. Wiesing, M. Kohlhase, and F. Rabe. Virtual Theories – A Uniform Interface to Mathematical Knowledge Bases. In J. Blömer, I. Kotsireas, T. Kutsia, and D. Simos, editors, *Mathematical Aspects of Computer and Information Sciences*, pages 243–257. Springer, 2017.

A Shifting and Substituting for the this-Operator

The expression $\uparrow^0$ corresponds to the "this" or "self" keyword present in many OO-language. It refers to the instance of the current theory. $\uparrow^n$ for $n > 1$ generalizes this principle, e.g., $\uparrow^1$ refers to the instance of the theory enclosing the current theory.

We can think of each $\uparrow^n$ as a variable implicitly declared at the beginning of each region and representing the region itself. To refer to that variable, $\uparrow^n$ uses its de-Bruijn index n. In particular, $\uparrow^0$ represents the current region, and when computing expressions like $m.c$, all occurrences of $\uparrow^0$ in the declaration of c must be substituted with m.

Akin to the de-Bruijn representation of variable binding, we must define shifting and substitution operations to move expressions between regions. Firstly, to move expressions originating in the j-th *enclosing* region into the current region, we apply $shift^j(\ -\ )$: it shifts all occurrences of $\uparrow^i$ to $\uparrow^{j+i}$. Actually, we must define an auxiliary function $shift^{j,k}(E)$ and then define $shift^j(E) := shift^{j,0}(E)$. The latter is defined by induction on E with k tracking the number of theories we have already traversed into, i.e.,

$$shift^{j,k}(\{\Sigma\}) := \{shift^{j,k+1}(\Sigma)\} \qquad shift^{j,k}(mod(T)) := mod(shift^{j,k+1}(T))$$

$$shift^{j,k}(Mod(T)) := mod(shift^{j,k+1}(Mod(T)))$$

$$shift^{j,k}(\ \uparrow^i\ ) := \uparrow^{i+j} \quad \text{if} \quad i \geq k \qquad shift^{j,k}(\ \uparrow^i\ ) := \uparrow^i \quad \text{if} \quad i < k$$

Example 5. Consider Ex. 4 and assume that $Carrier$ additionally defines $\textbf{type}\, v = u$. Now assume we use the expression $\uparrow^1 .v$ inside the type of add. The unshifted type of $\uparrow^1$ is $Mod(BiMagma)$, which at this point is $Mod(\{\textbf{type}\, u,\ \textbf{type}\, v = u\})$. After shifting, we obtain $Mod(\{\textbf{type}\, u,\ \textbf{type}\, v = \uparrow^1 .u\})$ and $\uparrow^1 .v$ definition-expands to $\uparrow^1 .u$.

Dually we define the function $subs^m(E)$ to move an expression E from an *enclosed* region to the current one—an operation that can only work if we have a model m of the enclosed region. $subs^t(E)$ is defined like $shift^{-1,0}(E)$ with the exception that $subs^m(\ \uparrow^0\ ) := t$. In particular, we have $subs^m(shift^1(E)) = E$.

B Normalizing Theories

We have already sketched the algorithm for normalizing theories, and we fill in the details now and define the normal form $\overline{T}^{\Gamma}$ of every $\Gamma \vdash T$ THY. While it is relatively straightforward in principle, it must be interwoven with type-checking. For example, there is no way to determine if $\{\textbf{include}\, S,\ \textbf{term}\, c : A\}$ is well-formed without normalizing S first: The normal form of S might already contain a declaration of c, and well-formedness must check if the two declarations of c can be merged.

First we simplify T to a list of declarations:

- If T is a global resp. regional identifier that resolves to $\textbf{theory}\,\tau = \{S\}$, then we continue normalizing S.
- If $T = \{\vec{D}\}$, then we return $\vec{D}$.
- For $T = m.S$, the typing rule given in Sect. 2 requires $\Gamma + |C \vdash S$ THY. To normalize $m.S$, we normalize S in $\Gamma + |C$ and then replace every expression e in the normal form with $m.e$.

Then we iteratively merge the obtained list $D_1, \ldots, D_n$ of declarations into $\{\} \ll D_1 \ll \ldots \ll D_n$.

$\{\vec{D}\} \ll d$ is defined as follows. If d is named and $\vec{D}$ already contains a declaration for the same name but for a different kind (term, type, or theory), the theory is ill-formed. Otherwise, we proceed by case distinction on d:

- Include declarations $d = \textbf{include}\,S$: $\{\vec{D}\} \ll d = \{\vec{D}\} \ll d_1 \ll \ldots \ll d_n$ where $\overline{S}^{\Gamma} = d_1, \ldots, d_n$, i.e., we merge in all declarations of the normalized included theory iteratively.
- Theory declarations $d = \textbf{theory}\,\sigma = S$: If there is no theory declaration for σ in $\vec{D}$, we simply have $\vec{D} \ll d = \vec{D}, \textbf{theory}\,\sigma = \overline{S}^{\Gamma}$, i.e., we normalize the body of the new theory.
  Otherwise, let $\textbf{theory}\,\sigma = \{\Sigma\}$ be in $\vec{D}$. It would be easy to forbid this case entirely. But we can do slightly better as Ex. 6 shows. We can put $\{\vec{D}\} \ll d = \vec{D}^{-\sigma}, \textbf{theory}\,\sigma = \Sigma' \ll \textbf{include}\,S$ if, compared to Σ, this only adds concrete fields to Σ or to any theory declared in Σ.
- Term declarations $\textbf{term}\,c : A = t$ (where t may be present of absent): Let $\textbf{term}\,c : A' = t'$ be the declaration already present in $\vec{D}$ (where t' may be present or absent); if no such declaration exists, we assume it is given by $\textbf{term}\,c : \texttt{Any}$. We check $\Gamma \vdash A$ TYPE. If t is present, we check $\Gamma \vdash t : A \sqcap A'$ and put $k := t$. If only t' is present, we proceed accordingly. If both t and t' are present, we also check $\Gamma \vdash t \equiv t'$. Then $\{\vec{D}\} \ll d = \vec{D}^{-c}, \textbf{term}\,c : A \sqcap A' = k$ (where k is absent if neither t nor t' are present).
- Type declarations $\textbf{type}\,a = A$ (where A may be present of absent): Let A' be such that $\textbf{type}\,a = A'$ is the declaration already present in $\vec{D}$ (where A' may be present or absent); if no such declaration exists, we assume it is given by $\textbf{type}\,a$. If A is present, we check $\Gamma \vdash A :$ and put $k := A$. If only A' is present, we put $k := A'$. If both A and A' are present, we also check $\Gamma \vdash A \equiv A'$. Then $\{\vec{D}\} \ll d = \vec{D}^{-a}, \textbf{type}\,a = k$ (where k is absent if neither A nor A' are present).

Here $\vec{D}^{-s}$ arises from $\vec{D}$ by dropping any existing declaration for the symbol s.

Example 6 (Extending Theories). Consider the following variant of the algebraic hierarchy in which every theory has a nested theory *Finite*:

$$\textbf{theory}\,\mathit{Carrier} = \{\textbf{type}\,u,\ \textbf{theory}\,\mathit{Finite} = \{\textbf{term}\,\mathit{finite} :\blacktriangleright \ldots\}\}$$
$$\textbf{theory}\,\mathit{Magma} = \{\textbf{include}\,\mathit{Carrier},\ \vec{D},\ \textbf{theory}\,\mathit{Finite} = \{\vec{E}\}\}$$

where the symbol $finite$ expresses the finiteness of the carrier (e.g., by enumerating the elements). Normalization turns $Magma$ into

$$\textbf{theory}\, Magma = \{\textbf{type}\, u,\ \vec{D},\ \textbf{theory}\, Finite = \{\textbf{term}\, finite :\blacktriangleright \ldots,\ \vec{E}\}\}$$

And the type $Mod(\textbf{include}\, Magma,\ \textbf{include}\, Finite)$ yields the type of finite magmas. (Note that the include of $Finite$ is only well-formed because the theory $Finite$ was brought into the current region by the include of $Magma$.)

This has the effect that $\vec{D}$ can contain all properties that hold for all magmas and $\vec{E}$ can contain those that only hold for finite carriers. This allows keeping related properties near each other and avoids introducing theory names like $FiniteX$ for every theory X, which gets very cumbersome when many such orthogonal features are used at the same time.

However, we must employ the restriction that $\vec{E}$ does not change the abstract fields of $Finite$: it must not add, change (e.g., by restricting the type), or remove (by defining) abstract fields of $Finite$. Otherwise, terms of type $Mod(Finite)$ over theory $Carrier$ would no longer be well-formed terms over $Magma$, which would violate the invariant of includes. That restriction is acceptable in practice because the most important use case for this pattern is adding defined declarations, e.g., to state additional algorithms or theorems for the finite case.

Intersection and Union of Types Finally, we need to define least upper bound $A \sqcap B$ for any two types. Note that this is not an operation of the syntax that is accessible to the user in some kind of intersection type system. Instead, it is an meta-level definition that computes the greatest lower bound of two types.

Its definition depends on which other types a UniFormal instance adds. For the types mentioned here, we have $\texttt{Int}\sqcap\texttt{Bool} := \texttt{Void}$ and $Mod(C)\sqcap Mod(D) := Mod(C + D)$, i.e., intersection of model types corresponds to union of theories.

If we also add types with contravariant subtyping rules, such as function types $A \to B$, we also have to define the greatest lower bound of types, e.g., to compute $(A \to B) \sqcap (A' \to B') := (A \sqcup A') \to (B \sqcap B')$.

Consequently, we also need the union of model types $Mod(C) \sqcup Mod(D) := Mod(C \cap D)$ where $\cap$ computes the intersection of theories. $\cap$ is much trickier to define than union of theories. One option is to normalize C and D and then choose those declarations that occur in both. That is what our implementation does. But that often produces large expressions that the user does not want to see. An alternative approach is to approximate $C \cap D$ as the union of all theory *names* that are explicitly included into both C and D. While not technically the intersection, it is often more practical.