Global, Regional, and Local Contexts

Florian Rabe[0000−0003−3040−3655]⋆

University Erlangen-Nuremberg

Abstract. We introduce UniFormal, a formal language for mathematical knowledge representation that continues the evolution of OpenMath and MMT. It is motivated by the lack of identifiers in these languages that exhibit the scoping behavior needed for, e.g., the fields in a record or the variables in a polynomial. 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 the identifiers of the current theory. A stack of regional contexts is used 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 a minimal language in this style, 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.

They all feature two levels of identifiers, which we will call *global* (i.e., symbols, OMS) and *local* (i.e., variables, OMV) here. This paper was motivated by the observation that neither adequately captures the scoping behavior of the identifiers in various standard constructions. We identify the issue and offer one potential solution.

For example, consider the identifiers x and y in the record $\{\mathbf{term}\, x : \mathtt{Int} = 0, \mathbf{term}\, y : \mathtt{Int} = 1\}$. They are not global (because the record may occur anonymously inside some expression), but they are not α-renamable variables either

⋆ The author was supported by the VoLL-KI project (see https://voll-ki.de) under German Research/Education Ministry (BMFTR) grant 16DHBKI089.

(because they may be referenced when selecting fields from the record). Essentially the same phenomenon occurs with anonymous theories (which behave very much like record types) and anonymous diagrams of theories as used in [CFS20].

A different example are the identifiers that are used as generators in various constructions. Consider the X in $R = \mathbb{R}[X]$. It cannot be a global identifier because X by itself is meaningless. But it cannot be an α-renamable local identifier either because we must be able to reference it from elsewhere, e.g., in the formula $X^2 + 1 \in R$. A closely related phenomenon occurs in generated groups like $\langle X, Y | X^2 = Y \rangle$, where X and Y are often referred to from outside of the definition of the group.

Proof assistants and computer algebra systems have encountered these issues and have developed ad hoc solutions. For example, Rocq [Roc25] and Isabelle [Pau94] do not use *anonymous* record types/theories and thus can use global identifiers for the fields. PVS [ORS92] uses anonymous records, and Mizar [TB85] uses structures that behave similarly; both use identifiers similar to the regional identifiers we will introduce below. Systems vary widely in how they represent the X in $\mathbb{R}[X]$, e.g., as the constructor of an inductive type, a bound variable, or as a string, or by using a least-fixed-point construction that avoids using any identifiers. SageMath [S+13] assumes that a universal ring of polynomials in all possible identifiers exists, of which $\mathbb{R}[X]$ is a subring; these identifiers are similar to our regional ones. While Mmt has proved remarkably resilient in light of retroactive extensions that changed fundamental design assumptions, e.g., [HKR12,WKR17], a foundation-independent solution for such identifiers has proved very difficult, to the point of essentially grinding Mmt development to a halt since 2023.

This paper presents a minimal standalone framework language, called UniFormal, that introduces an intermediate level, called the regional identifiers, and shows how it allows representing formal theories. 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 going back to at least [Dum12], 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 inferable and to serve as a starting point of a broader discussion of the new design ideas.

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 : `Bool`
	$\mid$ `Bool` $\mid$ `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
	$\mid \triangle^n$	model of n-th enclosing region (for $n \in \mathbb{N}$)
	$\mid \ldots$	other productions as needed
Contexts		
Γ	$::= D\ ([]\ T\ L)^*$	global+stack of (regional+local)
L	$::= D^*$	local context of bound variables

Fig. 1. UniFormal Syntax

2 UniFormal Syntax

The syntax of UniFormal is given in Fig. 1 (where [optional] and repeated* parts are marked as usual) and the meta-theoretical operations in 2.

There are several kinds of **declarations**. All but includes are **named**:

- Theory declarations **theory** $\tau = T$ give a name to a theory T.
- Symbol declarations introduce a named object such as a type or term.
- Include declarations **include** T create an inheritance relation between theories. Theory normalization eliminates them by merging all declarations of the included theory T into the including theory.

Each kind of named declaration corresponds to a kind of **expression**, with the identifiers referencing the former occurring as leaves of the latter. Expressions are anonymous transient syntax trees that are checked and interpreted against a context Γ.

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

A **theory** T is an expression that normalizes to a list $\{\vec{D}\}$ of declarations, essentially by expanding all theory definitions and include declarations. For $\Gamma \vdash T$ THY, we write $\overline{T}^{\Gamma}$ for this normal form of T. 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.

Beyond theories, UniFormal is flexible in what kinds of symbol declarations and expressions are used. Here we pick types and terms:

- **Type** declarations $\mathbf{type}\, a = [A]$ introduce a named type with an optional definition A. We could also allow type bounds and type operators, but we omit that here for simplicity.
- **Term** declarations $\mathbf{term}\, c : A = [t]$ 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.
- We represent axioms and theorems as special cases of terms with and without definition, respectively, via the type $\blacktriangleright F$ of proofs of Boolean term F.

Other languages may change these kinds as needed, e.g., to have kinds for universes, set, or proofs. Our results only require 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**, otherwise **abstract**. Similarly, we call a theory concrete if all declarations in its normal form are, otherwise abstract.

The details of **contexts** Γ are explained throughout the paper. Generally, the context must keep all information that an algorithm that traverses a formalization may need such as the type of every identifier that is in scope. Therefore, choosing the right data structure for the context can be as difficult as designing the entire formal system, and much of this paper is devoted to motivating and explaining our particular choice. We give the central definition below, but readers may simply think of Γ as a black box declaring all identifiers that expressions

can use, and then revisit the formal definition from time to time while reading the subsequent sections.

Definition 1 (Context). *A UniFormal context Γ is of the form $G\ \vec{F}$ where*

- *global context: G is a declaration of the form* **open theory** *root* $= \{\vec{\tau}\}$
  We think of τ as the set of all available toplevel declarations. The name root is irrelevant and signifies the root of the namespace hierarchy.
- *$\vec{F}$ is a stack in which each frame is of the form $|\, R\, L$ or $R\, L$ where*
  - *regional context: R is a theory of the form $\{\rho\}$.*
    We think of ρ as the set of all declarations that are available in the current *theory.*
  - *local context: a list L of variable declarations*
    We think of L as the set of bound variables in scope for the currently traversed subexpression.
  - *transparency: an optional modifier $|$ that indicates whether or not the identifiers of the previous frame remain in scope for the current frame.*

Contrary to the global context (which stays fixed during traversal) and the local context (which is transient in the sense that bound variables are declared, stay in scope for a while, and then go out of scope forever), the regional context can be changed back and forth. For example, if a traversal encounters $m : Mod(T)$ (which binds a variable m whose type makes it a model of theory T), it creates and pushes a fresh frame in which to process the anonymous theory T. Afterwards that frame is popped to remove the identifiers of T from the scope. But when m is used later on, say in an expression $m.c$ for some symbol c of T, the body of T must be brought back into scope to retrieve and process the type of c. Thus, we can jump to other theories by pushing onto the stack and return by popping in a way akin to how programming languages use a stack frame for each function call.

Switching to theory T by pushing a new frame for T will happen frequently, and we introduce a notation for it:

Definition 2 (Context Extension). *Given a context $\Gamma = G\ \vec{F}$ and a theory $\Gamma \vdash T$ THY, we define*

$$\Gamma + T := G\ (\vec{F},\ (\overline{T}^{\Gamma}\ \varnothing)) \qquad \Gamma + |T := G\ (\vec{F},\ (|\overline{T}^{\Gamma}\ \varnothing))$$

These extend $\vec{F}$ with a fresh frame initialized with the normal form of T as the regional and the empty list $\varnothing$ as the local context.

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

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

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 $=$.

The global context is the entire development. When traversal reaches the equality in the associativity axiom, the context stack contains a single frame: Its regional context contains $\mathbf{type}\, u$ and $\mathbf{term}\, op : (u, u) \to u$, which stem from the normalization of the preceding declarations of the current theory, i.e., $\mathbf{include}\, Magma$. Its local context declares the bound variables $x : u, y : u, z : u$.

3 Open vs. Closed Theories

3.1 Distinction

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. Analogues of Open and Closed Theories in Various Systems

UniFormal distinguishes two kinds of theories. We call a theory **open** if its declarations are freely accessible from the outside, otherwise **closed**. For example, in $\mathbf{open\,theory}\, \tau = \{\mathbf{term}\, c : A\}, \mathbf{term}\, d = \tau.c$, the symbol c of the open theory τ is referenced using a qualified identifier. If τ were closed, such a reference would be ill-formed. Closed theories require additional language features that regulate access to them.

Most formal systems (see Fig. 3) use a similar combination of open and closed theories, albeit sometimes in multiple variants and/or with a less clear-cut difference. Notably, MathML and MMT do not make the distinction, and the author now sees that as a deficiency.

The open/closed theory distinction is analogous to the **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 downstream 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. Common language features to access closed theories and to leverage this induction are (i) the models of C (where induction is used to interpret all expressions in the model), (ii) the term language of C (where induction is used for induction on expressions), or (iii) Prolog-style querying and negation-by-failure (where induction is used to exhaustively search all expressions). This paper formally defines only (i), but UniFormal's closed theories can be used for the others as well.

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 dependencies, to be provided as a single open theory G containing all available toplevel declarations—the global context. G is the root of a tree, whose inner nodes are nested open theories (representing packages, folders, namespaces, etc.) and whose leaves are symbol and closed theory declarations.

Remark 1 (Imports). When packages are published in central repositories, many languages 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 consisting of a list of term/type declarations for names s_i. Intuitively, 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.

This only makes sense if C is closed: changing the symbols of C changes what the models are.

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. Many computer algebra systems like Axiom [JS92], SageMath [S+13], or FoCaLiZe [H+12] do the same. 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 $mod(\mathbf{include}\, Magma, \mathbf{type}\, u = \mathtt{Int}, \mathbf{term}\, op = \lambda x, y.x + y)$ 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):

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

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:

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

$$\frac{\Gamma \vdash m : Mod(C) \quad \Gamma +\!| C \vdash S\ \mathrm{THY}}{\Gamma \vdash m.S\ \mathrm{THY}}$$

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:

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

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 $\Gamma \vdash m.\mathtt{Int} \equiv \mathtt{Int}$. The grayed out operation replaces occurrences of $\triangle^0$ (which refers to the current region) with $mod(T)$. It is defined in Sect. 5.2.

Remark 3 (The Type of Types). The combination of abstract type fields and a type of models allows defining the *type of types* as $Mod(\{\mathbf{type}\, univ\})$. 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 sophisticated solutions. In programming languages, having the type of types can be harmless; sometimes, like in Scala, the type of types simply

exists, and one has to dive deep into the language semantics to see where and how any issues are handled.

A rigorous semantics to UniFormal must carefully work out the details in a way that retains consistency. While this is very difficult, we ignore the issue here because it is orthogonal to our results. In fact, we are not even sure if a universal representation language should address the issue at all or if it should (at the risk of inconsistency and akin to informal mathematics) leave the details unspecified to avoid over-committing to any one solution.

Remark 4 (Dependent Types). The combination of nested theories and the type of models allows creating dependent types (in the sense of type expressions with terms as subexpressions). For example, $\mathbf{theory}\,\tau = \{\mathbf{term}\,c : \texttt{Int},\ \mathbf{theory}\,\nu = \{\ldots\}\}$ yields, for any value n, a different type $Mod(mod(\mathbf{include}\,\tau, \mathbf{term}\,c = n).\nu)$. Abstract type fields in theories have a similar effect. Therefore, it is difficult to design a language for theories and models without allowing some form of dependent types, and UniFormal does so.

3.3 The Term Language of a Theory

Term language types can be seen as dual to the type of models of a theory. For example, consider $\mathbf{theory}\,N = \{\mathbf{type}\,n, \mathbf{term}\,z : n, \mathbf{term}\,s : n \to n\}$ (assuming we have function types $\to$). Its term language is that of the natural numbers. More generally, we can represent families of mutually recursive inductive types or context-free grammars (using one type symbol per non-terminal) like this.

Constructing objects through term languages is commonly done, albeit usually under a different name: it subsumes 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.

While we do not consider it in this paper, our implementation already supports an extension of UniFormal with term language types: It allows forming the type $T[A]$, which holds the normal terms over theory T of type A. Induction on the type $T[A]$ is possible but only from outside of T. For example, in $\mathbf{theory}\,N = \{\ldots\}, \mathbf{term}\,even : N[n] \to \texttt{Bool} = \ldots$, induction on the term language of N is allowed in the definition of *even* but not inside N.

4 Inclusion of Theories

4.1 Definition

Intuitively, normalization of $\mathbf{include}\,S$ occurring in T means to copy all declarations of S into T. Only closed theories can be included—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, an inclusion morphism exists, written $\Gamma \vdash S \hookrightarrow T$, 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 $\mathbf{include}\, S$, then $\Gamma \vdash S \hookrightarrow T$.

4.2 Relation to Subtyping

Inclusion induces **subtyping** between the types of models: if $\Gamma \vdash S \hookrightarrow T$, then $\Gamma \vdash Mod(T) <: Mod(S)$. Informally, if T is bigger than S, than defining all of T implies defining all of S. This inclusion=supertype relationship is common in module systems, and languages often say that T **inherits** 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].

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 forms a subtype of the term language of T, i.e., term language types and model types behave dually.

Remark 5 (Decidability). Note that equality of types and terms is needed to determine theory inclusion, and theory inclusion is needed for subtype checking and thus for term/type equality checking. In particular, the equality of types and terms must be decidable for theory inclusion to be decidable. Whether or not that is the case, depends on what other types are added in a particular UniFormal instance.

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 $S + T := \{\mathbf{include}\, S, \mathbf{include}\, T\}$.

More generally, we need a **dependent union** $S \ll \vec{D} := \{\mathbf{include}\, S, \vec{D}\}$ to extend S with declarations $\vec{D}$ that may already use the symbols of S. For example, we have $\{\mathbf{type}\, a\} \ll \mathbf{term}\, c : a \;\equiv\; \{\mathbf{type}\, a,\ \mathbf{term}\, c : a\}$. That raises the question of **merging declarations**, e.g., do we have that

$$\{\mathbf{term}\, c : \mathtt{Int}\} \ll \mathbf{term}\, c = 1 \quad \equiv \quad \{\mathbf{term}\, c : \mathtt{Int} = 1\}$$

Dependent unions with such merging behavior are central to many systems including the inheritance systems of OO-languages and mixin or trait-based module systems. We adopt that idea in UniFormal and define theory normalization in such a way that multiple declarations for the same name are merged.

Theories are **normalized** by merging in every declaration individually as in $S \ll D_1 \ll \ldots \ll D_n$. In particular, we normalize $\mathbf{include}\, T$ by normalizing T into a list of declarations and then merging those in one at a time. A theory is well-formed if its normalization succeeds. It is difficult to define it in any other way because the normal form of a theory is needed to check references to its identifiers. However, this does not require normalizing each expression in a theory, only to expand all includes and merge all declarations of the same name.

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:

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

Note that merging declarations 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

$$\{\mathbf{term}\, odd : \texttt{Int} \to \texttt{Bool},\ \mathbf{term}\, even : \texttt{Int} \to \texttt{Bool} = \ldots,\ \mathbf{term}\, odd = \ldots\}$$

Not all merges are well-formed, e.g., UniFormal rejects merges if two definitions for the same name are not identical as in $\{\mathbf{term}\, c : \texttt{Int} = 1\} \ll \mathbf{term}\, c : \texttt{Int} = 0$. If multiple types are merged, the least upper bound is computed. This is non-trivial in particular for model types, where it is obtained by taking the union of theories as in $\{\mathbf{term}\, c : Mod(S)\} \ll \mathbf{term}\, c : Mod(T) \equiv \{\mathbf{term}\, c : Mod(S+T)\}$. That corresponds to the typical behavior in languages with inheritance. To protect the user from inadvertently introducing an inconsistent theory, UniFormal flags an error if the bound ends up being the empty type as in $\{\mathbf{term}\, c : \texttt{Int}\} \ll \mathbf{term}\, c : \texttt{Bool} \equiv \{\mathbf{term}\, c : \texttt{Void}\}$.

Remark 6 (Inconsistent Merges). If we see a symbol's type and definition simply as special cases of axioms, there is no need to ever reject a merge. We could merge, e.g., $\mathbf{term}\, c : \texttt{Int}$ and $\mathbf{term}\, c : \texttt{Bool}$ into $\mathbf{term}\, c : \texttt{Void}$, or $\mathbf{term}\, c = 1$ and $\mathbf{term}\, c = 0$ into a theory in which $0 \equiv 1$ holds. Typically, the resulting theories will be inconsistent, but that is not problematic per se. An alternative design could accept all merges and defer such inconsistency analysis to linters.

Remark 7 (Overloading). Some languages with inheritance, in particular OO-languages, allow for overloading: Then merging **term** c : `Int` and **term** c : `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 8 (Overriding). Overriding arises if there are two conflicting definitions, e.g., if it is not an error for T to declare **term** c : `Int` $= 1$ and **term** c : `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 9 (Redefinition). A special case of overriding can be allowed safely: if the overriding definition is extensionally equivalent to the overridden one. For example, if the theory S 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 [TB85] 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 are thus compatible with UniFormal. But they require proofs about the equality of terms and are therefore very difficult.

4.4 Total Includes

We define an inclusion $\Gamma \vdash S \hookrightarrow T$ to be **total** if every abstract symbol declaration in $\overline{S}^{\Gamma}$ is concrete in $\overline{T}^{\Gamma}$. Thus, a total include is conservative in the sense that it does not change the semantics of T. It can be seen as T implementing the interface S, or 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

$$\begin{array}{l}\textbf{theory}\ \mathit{Preorder} = \{\textbf{term}\ r : (u,u) \to \texttt{Bool},\ (\text{axioms omitted})\},\\ \textbf{theory}\ \mathit{Monoid} = \{(\text{as above}),\ \textbf{total include}\ \mathit{Preorder},\\ \quad \textbf{term}\ r = \lambda x, y.\exists d.\mathit{op}(x,d) = y\}\end{array}$$

the keyword **total** asserts that $Monoid$ not just inherits from but *implements* the theory $Preorder$. Note that all types in the definition of r can be inferred.

Here the totality check at the end of $Monoid$ would fail because the proofs of the axioms (reflexivity and transitivity) are still missing.

Secondly, an include can have a definition. $\mathbf{include}\, C = e$ 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. $\mathbf{include}\, C = e$ is equivalent to declaring $\mathbf{total\,include}\, C$ followed by $\mathbf{term}\, s = e.s$ resp. $\mathbf{type}\, s = e.s$ 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 $\mathbf{open\,theory}\, \tau = \{\mathbf{include}\, C, \ldots\}$ includes closed theory C, we call τ **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 well-formed only in contexts that also include all guards of C. This allows writing conservative extensions of C as open theories guarded by C.

Writing a new closed theory D that extends C for this purpose would be 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 for the extensions as in

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

Now assume we have $m : Mod(Monoid)$ and want to access the global identifier $g := Doubling.double$ 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 $\Gamma | + Monoid$ where the guard $Monoid$ is available so that checking g succeeds.

5 Global vs. Regional vs. Local Level

Finally, we can understand UniFormal's level of identifiers. The table below gives an overview. Note that these levels are orthogonal to the symbol *kinds*, 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 $(t_1, \ldots, t_n)$, where each t_i may introduce variables whose scope extends to the end of the block, as in $(\textbf{term}\, x : \texttt{Int} = 1, \textbf{term}\, y : \texttt{Int} = x + 1, f(y))$. 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 $((\exists x : \texttt{Int}.P) \Rightarrow B) \wedge C$ where x is visible in B but not in C [KK97]; and tactics in imperative proof languages such as *intros* in Rocq that dynamically introduce identifiers.

Remark 10 (Theory Variables). We allow for term and type declarations in local contexts but not for 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 of the form $\{\textbf{term}\, c : A, \ldots\}$. 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; or if we add term language types, then c must occur in terms over 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 an entirely different name clash behavior than both local and global identifiers: a second regional declaration for the same name is merged into the first one.

Remark 11 (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 $Record\, R := \{a : A, b : B\}$. 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 use a three-partite context holding the global, regional, and local identifiers. For example, when considering the term t in

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

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

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

Type-checking $Point$ nested inside $2Pointed$ requires opening a new region for the symbol c, while keeping the region of $2Pointed$ with the symbol a accessible. Nested theories also occur every time an *anonymous* theory C is used in $Mod(C)$ or $mod(C)$. Therefore, we use the **stack of regions** in Def. 1.

Reconsider the rule for checking the model type $Mod(C)$ from Sect. 3.2. To check $\Gamma \vdash Mod(\{\vec{D}\})$ 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 $\Gamma \vdash m : Mod(C)$ is the same as a theory morphism from C to the innermost region of Γ.

Technically, we must actually write $\mathbf{theory}\, Point = \{\mathbf{term}\, c : \triangle^1.a\}$ above, where $\triangle^1$ 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 (as defined in Ex. 1) are present, like in a ring:

$$\begin{array}{l}\mathbf{theory}\, BiMagma = \{\mathbf{include}\, Carrier,\\ \quad \mathbf{term}\, add : Mod(\mathbf{include}\, Magma,\, \mathbf{type}\, u = \triangle^1.u)\\ \quad \mathbf{term}\, mult : Mod(\mathbf{include}\, Magma,\, \mathbf{type}\, u = \triangle^1.u)\}\end{array}$$

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 $\mathbf{type}\, u = \triangle^1.u$.

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

Definition 3 (Lookup in a Context). *Consider a context $\Gamma = G\ F_l, \ldots, F_1$ 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 $\tau_1.\ldots.\tau_n.s$ 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 $\triangle^j$ is well-formed if $j \leq k$; its type is $shift^j(Mod(R_j))$. In particular, $\triangle^j.s$ 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 $L_k, \ldots, L_1$ and d is found L_j.*

Here, the expression $\triangle^0$ corresponds to the "this" or "self" keyword present in many OO-language. It refers to the instance of the current theory. $\triangle^n$ for $n > 1$ generalizes this principle, e.g., $\triangle^1$ refers to the instance of the theory enclosing the current theory. We can think of each $\triangle^n$ as a variable implicitly declared at the beginning of each region and representing the region itself. To refer to that variable, $\triangle^n$ uses its de-Bruijn index n. In particular, $\triangle^0$ represents the current region, and when computing expressions like $m.c$, all occurrences of $\triangle^0$ 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 $\triangle^i$ to $\triangle^{j+i}$.

Example 5. Consider Ex. 4 and assume that $Carrier$ additionally contains the declaration $\mathbf{type}\, v = u$. Now assume we use the expression $\triangle^1.v$ inside the type of *add*. The unshifted type of $\triangle^1$ is $Mod(BiMagma)$, which at this point is $Mod(\{\mathbf{type}\, u,\, \mathbf{type}\, v = u\})$. This becomes $Mod(\{\mathbf{type}\, u,\, \mathbf{type}\, v = \triangle^1.u\})$ after shifting, and $\triangle^1.v$ definition-expands to $\triangle^1.u$.

Dually we define the function $subs^m(E)$ to move an expression E from an *enclosed* region to the current one—an operation that requires a model m of the enclosed region. $subs^m(E)$ behaves similarly to $shift^{-1}(E)$ with the exception that $subs^m(\ \triangle^0\ ) := m$. In particular, we have $subs^m(shift^1(E)) = E$.

6 Conclusion

We have introduced the UniFormal language for an OO-style definition of mathematical theories and models. The key to obtaining a simple syntax that admits a rigorous semantics was a novel notion of contexts that distinguishes global, regional, and local identifiers, with a stack of regional contexts to allow jumping back and forth between regions. While global and local identifiers are well-understood, the regional identifiers offer a novel systematic representation of various common constructions that can be difficult to represent otherwise. Examples include record fields or generators like the variables in a polynomial ring.

Concrete instances of UniFormal can extend it towards specialized languages like logics, programming languages, or or flexiformal languages like sTeX [Koh08], while enabling interoperability through a shared theory layer. Indeed, such extensions can be designed by adding features that do not interact substantially with the context.

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.

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.

CFS20. J. Carette, W. Farmer, and Y. Sharoda. Leveraging the information contained in theory presentations. In C. Benzmüller and B. Miller, editors, *Intelligent Computer Mathematics*, volume 12236, pages 55–70. Springer, 2020.

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.

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

Dum12. S. Dumbrava. A Type Theory based on Reflection. Master's thesis, Jacobs University Bremen, 2012.

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

H+12. T. Hardin et al. The focalize essential, 2012. http://focalize.inria.fr/.

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.

JS92. R. Jenks and R. Sutor. *AXIOM: The Scientific Computation System*. Springer, 1992.

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.

Koh08. M. Kohlhase. Using LaTeX as a Semantic Markup Format. *Mathematics in Computer Science*, 2(2):279–304, 2008.

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.

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.

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

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

Roc25. Rocq Consortium. The Rocq Prover: Reference Manual. Technical report, INRIA, 2025.

S+13. W. Stein et al. *Sage Mathematics Software*. The Sage Development Team, 2013. http://www.sagemath.org.

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.