Representing Structural Language Features in Formal Meta-Languages*

Dennis Müller[1,2][0000-0002-4482-4912]**, Florian Rabe[1][0000-0003-3040-3655], Colin Rothgang[3], and Michael Kohlhase[1][0000-0002-9859-6337]

[1] Computer Science, University Erlangen-Nuremberg
[2] Computational Logic, University of Innsbruck
[3] Mathematics, TU Berlin

Abstract. Structural language features are those that introduce new kinds of declarations as opposed to those that only add expressions. They pose a significant challenge when representing languages in meta-languages such as standard formats like OMDoc or logical frameworks like LF. It is desirable to use shallow representations where a structural language feature is represented by the analogous feature of the meta-language, but the richness of structural language features in practical languages makes this difficult. Therefore, the current state of the art is to encode unrepresentable structural language features in terms of more elementary ones, but that makes the representations difficult to reuse and verify. This challenge is exacerbated by the fact that many languages allow users to add new structural language features that are elaborated into a small trusted kernel, which allows for a large and growing set of features.

In this paper we extend the Mmt representation framework with a generic concept of structural features. This allows defining exactly the language features needed for elegant shallow embeddings of object languages. The key achievement here is to make this concept expressive enough to cover complex practical features while retaining the simplicity of existing meta-languages. We exemplify our framework with representations of various important structural features including datatype definitions and theory instantiations.

1 Introduction and Related Work

Motivation Language design is generally subject to the expressivity-simplicity trade-off. In particular, designing a representation language for mathematics involves adequately capturing the ways how mathematical knowledge is expressed in practice. On the other hand, the language must be as simple as possible to allow establishing meta-theoretical properties and obtaining (and maintaining!)

* The authors were supported by DFG grant RA-1872/3-1, KO 2428/13-1 OAF and EU grant Horizon 2020 ERI 676541 OpenDreamKit.
** The first author is supported by a postdoc fellowship of the German Academic Exchange Service (DAAD)

scalable implementations. This problem is exacerbated by the fact that the language features needed in the long run are often not apparent at the beginning of a project. Moreover, depending on the availability of resources and the interests of the user community, languages and systems may be used in applications not foreseen by the developers.

Because retroactive changes become increasingly costly once meta-theory or implementation have been developed, adding new language features is often not feasible. In a curse-of-success effect, language developers may find themselves overwhelmed by feature requests from users, which they cannot add easily or at all without breaking developments by other users. Therefore, it becomes important to design languages with extensibility in mind. This is particularly difficult for *structural* features, and especially challenging for meta-languages such as standardized representation formats like OMDoc [Koh06] or logical frameworks like LF [HHP93]: These languages partially derive their value from being simple and elegant and cannot afford constantly adding features. On the other hand, they cannot afford fixing the set of structural features either: That would require encoding all other features via complex, often non-compositional translations, which are difficult to verify and preclude interoperability.

A particularly successful model used in many proof assistants has been a two-component design: firstly, a small, fixed, and carefully-designed kernel is used as the ultimate arbiter of correctness; secondly, a higher-level and more flexible component reads user input and translates it into the kernel syntax, a process we call **elaboration**. For example, major proof assistants like Coq [Coq15] and Isabelle [Pau94] have over time arrived at this model, and attention is increasingly shifting towards the elaboration component. Pure LCF systems like HOL Light [Har96] can be seen as an extreme case with the host programming language as the (Turing-complete) higher-level language.

Elaboration is typically implemented *programmatically*, i.e., via arbitrary code in the tool's underlying programming language. In the simplest case, a new kind of declaration could be introduced as a type $N <: D$ where N holds the declarations and D is an abstract interface for arbitrary declarations, together with a function $N \to \mathrm{List}(D)$ that elaborates an instance of N into other declarations. If the logic is sufficiently strong, tools may use reflection to define programmatic elaboration inside the logic as done for Idris in [CB16] and Lean in [EUR+17]. Concrete examples of high-level language features with elaboration-based semantics include

- HOL-style subtype definitions [Gor88], elaborated into an axiomatic specification of a new type,
- Mizar's many different definition principles, elaborated into axiomatic specifications of new function symbols,
- Isabelle's so-called derived specification elements including inductive, record, and quotient types, elaborated subtype definitions of some appropriately large type, see e.g., [BHL+14],
- PVS's inductive types, elaborated into an axiomatic specification of the induction properties,

– Coq's record types, elaborated into inductive types with a single constructor,
– Coq's sections, elaborated into kernel declarations that abstract over all variables declared in the section.

As the examples already indicate, elaboration is recursive, e.g., the elaboration of an Isabelle inductive type definition may lead to a subtype definition, which is then elaborated further. It may also be nested, e.g., the elaboration of a nested Coq section first generates declarations in its parent section, which is then elaborated later.

But the elaboration-based approach has two major difficulties. Firstly, elaboration necessarily destroys the high-level structure. If only the kernel representation is effectively available to other applications (as we found is often the case, see [KR20]), it becomes harder to transfer and reuse developments. Secondly, programmatic elaboration offers a high degree of flexibility but also makes it harder to analyze or implement high-level declarations generically.

In a response to these issues, we introduced declaration patterns in [HKR12] and [Hor14]. They allowed describing elaboration *declaratively* inside the logic rather than programmatically. Declaration patterns were successful in many cases including typical logical declarations [HR15], Mizar's definition principles [IKRU13], and HOL type definitions [KR14]. But, being fully declarative, they expectedly could not cover many practically important examples. For example, to elaborate an inductive data type definition, one has to generate an inequality axiom for every pair of constructors — something that quickly becomes awkward to describe without a general purpose programming language.

Contribution We expand on the declarative approach of [HKR12,Hor14] by extending the MMT framework with a generic extension mechanism based on programmatic elaboration. Critically, despite being very general, the declarations introduced by structural features share the same simple syntactic shape, which allows for simple specifications and uniform implementations.

Because MMT allows implementing logical frameworks such as LF, this immediately yields corresponding extensions of these. Our design was strongly motivated by and evaluated in our work on exporting proof assistant libraries [KR20], where we had to model many high-level language features of proof assistants. For example, we have already used our design to represent PVS includes [KMOR17] or Coq-style sections [MRSC19] (see Subsection 4.2).

Overview This paper is organized as follows. In Section 2, we recap the parts of OMDOC/MMT, which we use as the underlying core language. Section 3 introduces the infrastructure for structural language extensions. We look at concrete instances and develop a varied array of MMT structural features in Section 4 and Section 5. Finally, Section 6 concludes the paper and discusses future work.

2 Preliminaries

OMDOC is a rich representation language for mathematical knowledge with a large set of primitives motivated by expressivity and user familiarity. The MMT

[RK13] language is a complete redesign of the formal core of OMDoc focusing on foundation-independence, scalability, modularity and minimality.

In Figure 1, we show a fragment of the Mmt grammar that we need in the remainder of this paper. Meta-symbols of the BNF format are given in color.

The central notion in Mmt is that of a **diagram** consisting of a list of modules. For our purposes, **theories** are the only modules we need. Mmt theories are named sets of statements and are used to represent formal constructs such as logical frameworks, logics, and theories. At the **declaration** level Mmt has **includes** and **constants**. Constants are meant to represent a variety of OMDoc declarations and are simply named symbols with an optional type and definition. The types and definitions are Mmt **expressions**, which are based on OpenMath. Expressions are references to bound variables x and constants c, bound variable declarations $x : E$, and complex expressions $c(E^*)$ (which include variable binding by using $x : E$ as a child).

Module Level			
Diagram	γ	::=	Mod*
Module	Mod	::=	Thy
Theory	Thy	::=	T = {Dec*}
Statement Level			
Declaration	Dec	::=	c[: E][:= E]
		|	Thy | `include` l
Object Level			
Expression	E	::=	x | c | x : E
		|	$c(E^*)$

x, c, T represent variable, constant, and theory names (strings) respectively

Fig. 1. Mmt Grammar

The semantics of Mmt provides an inference systems that includes in particular two judgments for typing and equality of expressions. Via Curry-Howard, the former includes provability, e.g., a theorem F is represented as a constant with type F, whose definiens is the proof. We have to omit the details here for brevity. We only emphasize that Mmt is foundation-independent: The syntax does not assume any special constants (e.g., λ), and the semantics does not assume any special typing rules (e.g., functional extensionality). Instead, any such foundation-specific aspects are supplied by special Mmt theories called **foundations**. For example, the foundation for the logical framework LF [HHP93] declares constants for `type`, λ, Π, and @ (for application) as well as the necessary typing rules. Thus, the Mmt module system governs, e.g., which typing rules are available in which theory. The details can be found in [Rab17].

3 Structural Features

Before we come to a formal definition, let us consider **record types** as an example for a structural feature.

A record type R is a collection of typed (in our case, optionally additionally defined) fields $x : T$. A record **term** r is a collection of assignments $x := d$ for each field, such that if R contains $x : T$, then d has type T. In **dependent** record types, T may additionally refer to previous fields.

For any such record term $r : R$, we then have a projection operator "." such that $r.x$ has type T and (if r is not primitive) $r.x = d$. As such, a record type

$R = [\![\ x_1\ :\ T_1 \ldots x_n\ :\ T_n\ ]\!]$ can be implemented as a high-level structure that elaborates into

- A single constructor $\mathrm{Con}_R : T_1 \to \ldots \to T_n \to R$,
- for each field $x : T$ in R, a projection function $\cdot.x$ of type $R \to T$, and
- Axioms asserting injectivity and surjectivity of the constructor, as well as appropriate equalities implying that constructor and projection functions commute appropriately.

We will return to this example in more detail in Subsection 4.1.

There is a notable correspondance between record type declarations and **theories**, in that both consist (primarily) of declarations of the form $x : T$. It thus seems attractive to reuse the grammar of theories to allow for declaring record types as a high-level feature, motivating the following:

Definition 1. We extend the Mmt grammar by a new production rule:

$$\mathrm{Dec} ::= d : \mathfrak{f}\,(E^*)\ =\ \{\mathrm{Dec}^*\}$$

We call this a **derived declaration** of the structural feature $\mathfrak{f}$. d is the name of the derived declaration, the E are its **parameters** and the Decs its **internal declarations**.

Definition 2. A **structural feature** is a triple $(\mathfrak{f}, v, \epsilon)$, where:

1. $\mathfrak{f}$ is the **name** of the feature,
2. v is a **validity predicate** on derived declarations

$$D \coloneqq d : \mathfrak{f}\,(F_1 \ldots F_n)\ =\ \{S_1 \ldots S_m\}$$

   If $v(D)$ holds, we call D a (well-formed) **derived declaration** of $\mathfrak{f}$.
3. ϵ is called an **elaboration function**, mapping a derived declaration D of $\mathfrak{f}$ to its **elaboration**: a set of declarations, which we also call the **external declarations** of D.

Once a derived declaration is declared, we will (almost) never care about its internal declarations anymore. The corresponding structural feature checks whether a derived declaration D conforms to some specific pattern, checking its components and internal declarations separately (possibly generating errors), and **elaborates** D into a set of **external declarations** based on its constituents. Since checking often requires elaboration (and vice versa), the Mmt implementation unifies ϵ and v into a single method. The external declarations specify the intended semantics of the derived declaration.

The structural feature itself is written in Scala using the Mmt-API, which provides dedicated abstractions, and acts as a **rule** similarly to the typing rules mentioned in Section 2. Just like typing rules, structural features (or rather, their derived declarations) can thus be made available in precisely those theories where they are deemed valid.

For the rest of this paper, we will assume various extensions of LF as our foundations. If our external declarations contain **axioms**, we assume some fixed

logic declared in the foundation, providing a type `prop`, an operator $\vdash$ of type $\texttt{prop} \to \texttt{type}$, a typed equality operator $\doteq : \prod_{A:\texttt{type}} A \to A \to \texttt{prop}$ and the usual logical connectives.

However, it should be noted that the structural features presented herein can be easily adapted to any logic sufficiently strong to allow for defining (equivalents to) their external declarations.

4 Examples

We will now show the practical utility of these relatively abstract definitions in some paradigmatic cases at the declaration and module levels.

4.1 Datatypes

Inductive Types Structural features can provide a convenient syntax for declaring inductive types. Consider for example a (parametric) type of **lists** $\texttt{List}(A)$ of type A, which can be defined as the inductive type generated by the two constructors $\texttt{nil} : \texttt{List}(A)$ and $\texttt{cons} : A \to \texttt{List}(A) \to \texttt{List}(A)$. We devise two structural features with names `induct` and `indef`, allowing us to declare inductive types and inductive definitions respectively, as in Figure 2[4]. Naturally, parametric inductive types require a logic with (at least) shallow polymorphism.

```
induct Lists (A:type) | =
  List : type
  nil  : List
  cons : A → List → List | # 1 :: 2
```

```
indef Conc (Lists, B : type, ls : List B) | =
  conc : List B → List B | # 3 ++ 2
  Nil  = ls
  Cons = [b:B,l:List B] b :: (conc l)
```

Fig. 2. Lists as an Inductive Type and Concatenation as an Inductive Definition

If the underlying logic $\mathcal{L}$ provides primitive typing features that subsume inductive types, such as W-Types, `induct` and `indef` can elaborate into their (usually syntactically cumbersome) $\mathcal{L}$-correspondents. A structural feature elaborating into W-types is described in [Mül19].

In the absence of such a typing feature, we can instead elaborate into the corresponding constructors and axioms (expressed in some logic declared in our foundation) asserting their collective injectivity and surjectivity, in the manner which we will describe shortly. Importantly, we can use the same validity predicate and feature name for both variants, preserving the syntax of the structural features across logics. This ensures that whenever $\mathcal{L}'$ extends $\mathcal{L}$ by an inductive typing feature, any $\mathcal{L}$-theory using `induct` and `indef` remains a valid $\mathcal{L}'$-theory, but the elaboration in the stronger logic will consist of **defined** constants.

[4] These listings show our actual formalizations in concrete syntax and use a few semantically inessential features that go beyond the syntax introduced in Figure 1. Most notably, # introduces a notation and | separates declaration components.

Elaborating ***induct*** A derived declaration

$$D_{ind} : \texttt{induct}\,(t_1 : T_1 \ldots t_n : T_n) \;=\; \{S_1 \ldots S_m\}$$

is elaborated as follows:

1. Any type-level declaration $S_i : \texttt{type}$ is elaborated into

$$D_{ind}/S_i \;:\; \prod_{t_1:T_1,\ldots,t_n:T_n} \texttt{type}.$$

   This allows for mutually inductive and parametric types.
   Let $I_1 \ldots I_k$ be the type-level declarations of D_{ind}. The remaining declarations need to either *i*) have type I_i, or *ii*) have type $T'_1 \to \ldots \to T'_k \to I_i$ for some types $T'_1 \ldots T'_k$, and are thus assumed to be constructors.
2. For each remaining $S_i : T \to I_i$[5], we extend the elaboration by the constructor

$$D_{ind}/S_i \;:\; \prod_{t_1:T_1,\ldots,t_n:T_n} T \to I_i$$

3. **(No-confusion)** For each constructor S_i, we add
   – an axiom that S_i is injective in each argument, and
   – an axiom, that $S_j(a) \neq S_i(b)$ for any other constructor $S_i \neq S_j$ and sequences or arguments (a) (b) of adequate arity and types.
4. **(No-junk)** Several axioms that guarantee that the inductively defined types in the elaboration are **initial models** of their respective model category. This is the trickiest part of the construction and treated in detail in [Rot20].

Elaborating ***indef*** Having an inductive type T_I elaborated from D_{ind}, we can design `indef` to allow for conveniently specifying inductive definitions and consequently (using *judgments-as-types*) proofs by induction.[6]

A derived declaration $D_{def} : \texttt{indef}\,(D_{ind}, t_1 : T_1 \ldots t_n : T_n) \;=\; \{S_1 \ldots S_m\}$ has to satisfy the following properties (which collectively constitute the validity predicate) in order to be considered well-formed:

1. The first declaration S_1 has to have function type $T_I \to A$ for some type A.
2. For every constructor $\texttt{con} : T'_1 \to \ldots \to T'_k \to T_I$, there has to be an S_i with the same name, being a defined constant

$$\texttt{con} \;:\; T'_1 \to \ldots \to T'_k \to A \;:=\; \lambda a_1 : T'_1, \ldots, a_k : T'_k.\; t$$

The elaboration then consists of the following external declarations:

1. A constant $D_{def}/S_1 \;:\; \prod_{t_1:T_1,\ldots,t_n:T_n} T_I \to A$,
2. For every constructor $\texttt{con} : T'_1 \to \ldots \to T'_k \to T_I$ and corresponding internal declaration $\texttt{con} \;:\; T'_1 \to \ldots \to T'_k \to A \;:=\; \lambda a_1 : T'_1, \ldots, a_k : T'_k.\; t$, an axiom

$$D_{def}/\texttt{con} \;:\; \prod_{t_1:T_1,\ldots,t_n:T_n} \prod_{a_1:T'_1,\ldots,a_k:T'_k} \vdash\; D_{def}/S_1(t_1, \ldots, t_n, \texttt{con}(a_1, \ldots, a_k)) \doteq t$$

[5] For notational simplicity, we only consider the case of unary constructors; the generalization to n-ary constructors is clear

[6] For simplicity, we restrict ourselves to the case where D_{ind} elaborates into a single inductive type (ignoring mutual induction).

Records In [MRK18,Mül19], we describe an operator `Mod`, which takes as argument a (reference to a) theory $\mathcal{T}$ and returns a (dependent) record type with manifest fields whose fields correspond to the declarations in $\mathcal{T}$ – effectively yielding a type of models of $\mathcal{T}$. This assumes a background logic $\mathcal{L}$ with record types as a typing feature.

For the common case that we want to have the `Mod`-type be *i*) a named record type and *ii*) only need $\mathcal{T}$ in order to define `Mod` $\mathcal{T}$, we can introduce a structural feature `rectp` with the same functionality as `Mod`. In this case, the validity predicate accepts any theory and the elaboration simply consists of a named record with the inner declarations as fields. If a derived declaration of `rectp` has additional parameters $t_i : T_i$, these are λ-bound to the corresponding external declaration; i.e. a derived declaration

$$D_{rectp} : \texttt{rectp}\,(R,\ t_1 : T_1, \ldots t_n : T_n) \ = \ \{s_1 \ : \ T_1'[\,\coloneqq\ d_1] \ldots s_m \ : \ T_m'[\,\coloneqq\ d_m]\}$$

elaborates to

$$R \ : \prod_{t_1:T_1,\ldots,t_n:T_n} \texttt{type} \ \coloneqq \ \lambda t_1 : T_1, \ldots, t_n : T_n.\ [\![\ s_1 \ : \ T_1'[\,\coloneqq\ d_1] \ldots s_m \ : \ T_m'[\,\coloneqq\ d_m]\ ]\!]\,.$$

Analogously, we can introduce a structural feature `rectm` with derived declarations $D_{rectm} : \texttt{rectm}\,(r,\ D_{rectp}\ F_1 \ldots F_n) \ = \ \{S_1 \ldots S_m\}$, where each S_i is a defined constant whose name corresponds to an (undefined) field of R. Additional parameters $t_i : T_i$ are again λ-bound; i.e. a derived declaration

$$D_{rectm} : \texttt{rectm}\,(D_{rectp},\ t_1 : T_1 \ldots t_n : T_n\ R) \ = \ \{s_1 \ \coloneqq\ d_1 \ldots s_m \ \coloneqq\ d_m\}$$

elaborates to the named record term

$$r \ : \prod_{t_1:T_1,\ldots,t_n:T_n} R \ \coloneqq \ \lambda t_1 : T_1, \ldots, t_n : T_n.\ \langle\!|\ s_1 \ \coloneqq\ d_1 \ldots s_m \ \coloneqq\ d_m\ |\!\rangle\,.$$

One big advantage of this approach in MMT surface syntax is that each field in a `rectm`-declaration can be checked separately against the corresponding field in the record type, whereas the elaborated expression $\langle\!|\ s_1 \ \coloneqq\ d_1 \ldots s_m \ \coloneqq\ d_m\ |\!\rangle$ is treated as a single term and checked as a whole. While this does not make a difference semantically, it allows for much better localization of errors and more helpful error messages.[1]

EdN:1

In a logic $\mathcal{L}$ without a notion of record types, the structural feature `rectp` can instead elaborate a derived declaration in the manner described in Section 3.

4.2 Module System

MMT Structures are an MMT primitive kind of theory morphisms, that essentially behave like named includes with modification: A structure $S : \mathcal{T}_1 \to \mathcal{T}_2$ makes all declarations in $\mathcal{T}_1$ accessible in $\mathcal{T}_2$, but allows for

[1] EdNote: The same applies to the indef feature.

– supplying additional names (**aliases**) to constants via `@`-annotations,
– changing notations of constants and
– supplying definientia for previously undefined constants.

In particular, unlike includes, multiple structures with the same domain are not idempotent. A typical example for structures is given in Figure 3: A theory of rings is constructed using two structures for addition (from `AbelianGroup`) and multiplication (from `Monoid`) whose universes are defined to be the same type R and whose operations and units are renamed and provided with adequate notations. The axioms in the domain theories are thus automatically imported into `Ring`.

```
theory Monoid =
    U : type
    op : U → U → U |# 1 ∘2
    unit : U | # e
    axiom1 : ⊢ ∀ [x] x ∘ e ≐ x
    ...
```

```
theory AbelianGroup =
    include ?Monoid
    inv : U → U |# 1 ^−1
    axiom1 : ⊢ ∀ [x] x ∘ (x ^−1) ≐ e
    ...
```

```
theory Ring =
    R : type
    structure addition : ?AbelianGroup =
        U = R |
        op @ plus | # 1 + 2
        unit @ zero | # 0
        inv @ minus | # − 1
    structure multiplication : ?Monoid =
        U = R |
        op @ times | # 1 · 2
        unit @ one | # 1
    ...
```

Fig. 3. Theories for Monoids, Abelian Groups and Rings using Structures

As mentioned, structures are MMT primitives. However, they can be easily defined using structural features: A derived declaration $S : \texttt{structure}\,(\mathcal{T}_1) = \{S_1 \dots S_m\}$ satisfies the validity predicate iff:

1. $\mathcal{T}_1$ is a valid theory,
2. any constant S_i shares a name with a constant in $\mathcal{T}_1$ and
3. for any defined constant $S_i = (c := d)$ with $c : T$ in $\mathcal{T}_1$, we demand that d type checks against T', where T' is T with constant references from $\mathcal{T}_1$ appropriately substituted by their S-counterparts.

The elaboration then consists simply of the appropriately modified copies of the declarations in $\mathcal{T}_1$ with their names prefixed by $S/$.

Coq-Style Sections In [MRSC19], we present an import of the Coq Library into the MMT system. In order to preserve the original syntax of the library as closely as possible, this necessitated mirroring Coq's module system (Sections, Modules, Module Types) within MMT, which was done using structural features. Exemplary, we will look at Coq Sections.

A Coq Section is a named theory, in which it is allowed to introduce **variables** via declarations. After a section ends, its contents are accessible with all variables becoming Π-bound to all subsequent declarations.

Figure 4 shows an example of a Coq section. A and f are declared as variables and used like constants in the remainder of the section. The defined constant $ltof$ within the section hence takes two arguments a, b. After the section is closed however, $ltof$ is used as a quaternary function, with its arguments being the type A, the function f and the two arguments a, b.

```
Section Well_founded_Nat.
  Variable A : Type.
  Variable f : A -> nat.
  Definition ltof (a b:A)
    := f a < f b.
  Definition gtof (a b:A)
    := f b > f a.
End Well_founded_Nat.
```

```
Section Well_founded_Nat =
  A : type      | role Variable
  f : A → nat   | role Variable
  ltof : A → A → prop |
    = [a,b] f a < f b
  gtof : A → A → prop |
    = [a,b] f a > f b
```

Fig. 4. A Coq Section and its MMT Counterpart

The right side of Figure 4 shows the same Section, but expressed in MMT syntax using a new structural features `Section`. Variables are marked with the `role Variable` flag. The validity predicate accepts any theory. A derived declaration $D = Sec : \texttt{Section}\,() = \{S_1 \dots S_n\}$ is elaborated as follows:

1. Any constant with the `role Variable` flag is not elaborated,
2. for any other constant $S_i = c : T\ [:= d]$, let $v_1 : T_1 \dots v_n : T_n$ be all variables declared in D prior to S_i. Then extend the elaboration of D by

$$Sec/c : \prod_{v_1:T_1,\dots,v_n:T_n} T\ [:= \lambda v_1 : T_1, \dots, v_n : T_n.\ d]$$

PVS-Style Includes In [KMOR17,Mül19], we present an import of the PVS Prelude and NASA Libraries into the MMT system. One of the peculiarities of the PVS language is their prevalent use of parametric theories. These are commonly used whenever results involving multiple models of the same theory are needed; e.g. a theory of groups in PVS would be parametric in the signature of groups (i.e. $\texttt{Group}(U, \circ, e, {}^{-1})$), such that whenever a result relating two groups would be needed, the containing theory would simply import two instances of the theory of groups with different parameters. Additionally, parametric theories can be included "as is", in which case the parameters can be provided individually each time a symbol from the included theory is used (e.g.

`Group`?`associativity`$[\mathbb{Z}, +, 0, -]$). Effectively this makes each constant in the included theory take additional arguments for the theory parameters.

While MMT supports parametric theories, they are treated quite differently than in PVS. Theory parameters need to be supplied whenever a parametric theory is included, and by the definitional property of **implicit morphisms** (which includes are, see [RM18]), at most one theory morphism between two theories may be implicit. This means that a parametric theory can only be included in another theory once with one fixed set of parameters supplied.

As with Coq sections, we hence opted for using a structural feature `ParInclude` whose derived declarations take a single parameter $\mathcal{T}$ for the included theory and no inner declarations. The elaboration of such a derived declaration then consists of the constants in $\mathcal{T}$ with the theory parameters of $\mathcal{T}$ being Π-bound analogously to our treatment of Variables in Coq sections above. This treatment subsumes all possible use cases of includes in PVS.

Notably, this comes at the cost of blowing up theories massively, since any use of the feature copies all declarations of the included theory in its elaboration, slowing down various MMT services noticeably. However, since the declarations can be elaborated individually, this allows for future improvements by potentially treating the elaborations lazily.

While the grammar presented in this paper requires derived declarations to be named – as the actual abstract syntax of MMT does – the actual concrete syntax allows specific features (such as `ParInclude`) to omit names. This way, include-like features do not have to be named by a user, and internal names are generated instead.

4.3 Declaration Patterns

Now we recover the declarative special case introduced in [HKR12,Hor14] as a special case of our structural features.

Specification A declaration pattern is a structural features whose elaboration is so simple that it can be specified declaratively in the meta-language. We recap the definition of [HKR12].

A **declaration pattern** is a declaration of the form `pattern` $P(\Gamma) = \{\Delta\}$ where P is the name of the pattern, $\Gamma = x_1 : E_1, \ldots, x_m : E_m$ is a context declaring parameters that are bound in $\Delta = D_1, \ldots, D_n$, and the D_i are declarations. The D_i can be arbitrary declarations, and we assume they can be elaborated into constant declarations $c_1 : F_1, \ldots, c_n : F_n$.

Thus, a declaration pattern is essentially the same as a theory with some parameters. It is also similar to a parametric record type — except that it does not introduce a type, i.e., $P(e_1, \ldots, e_m)$ cannot be used as a type. The latter ensures that any declaration (including type declarations) that can be used in theories can also be used in patterns.

An **instance** of the pattern P (assumed to be declared as above) is a declaration of the form `instance` $p : P(e_1, \ldots, e_m)$ where p is the name of the instance

and the e_i satisfy $e_i : E'_i$ where E' arises from E by substituting each preceding x_i with e_i. The semantics of such an instance declaration is that it induces the declarations $p/c_1 : F'_1, \ldots, p/c_n : F'_n$, where the p/c_i are qualified names and the F'_i arise as above. Thus, Δ can be seen as the declarative definition of the elaboration of the instance p.

Implementation Declaration patterns introduce two new kinds of declarations (patterns and instances), and we capture them with two structural features.

Firstly, the structural feature for patterns uses the name $\mathfrak{f} = \texttt{pattern}$. A derived declaration $P : \texttt{pattern}(A_1, \ldots, A_m) = \{\Delta\}$ is valid iff each A_i is of the form $x_i : E_i$ and concatenating those yields a valid context, and if the declarations in Δ are valid relative to that context. It elaborates to nothing.

Secondly, the structural feature for instances uses the name $\mathfrak{f} = \texttt{instance}$. A derived declaration $p : \texttt{instance}(A) = \{\}$ is valid iff A is of the form $P(e_1, \ldots, e_m)$ for a pattern P that was declared in the current scope and the e_i satisfy the respective type constraints. Defining the elaboration of such a derived declaration is straightforward and proceeds exactly as specified above.

We do not touch on the issues of concrete syntax in this paper, it is straightforward to see that only simple notational rules are needed to make these derived declarations mimic the concrete syntax of [HKR12] entirely. This is already supported by our implementation.

Notably, the resulting implementation of declaration patterns is significantly simpler and easier to read, understand, and verify than the existing prototype implementation that had been built as a part of [Hor14]. This is because all the bureaucracy of elaboration is now covered uniformly by the framework so that the code can focus on the semantically relevant details. But more importantly, the prototype implementation was developed as an extension of Mmt in a PhD thesis and was never well-integrated with the rest of the code. Such a deep integration would have gone beyond the resources and purpose of that PhD thesis. Because structural features are now deeply integrated with Mmt out of the box, our new implementation is not only simpler but also better than the old prototype.

5 Module-Level Features

Definition To simplify the presentation, we have so far only considered structural features that extend the syntax *inside* theories. But it is natural to also consider extending the module-level syntax. We specify and implement this in essentially the same way. The key step is to allow derived declarations as modules, i.e., we add a production to the Mmt grammar and speak of **derived modules**:

$$\text{Mod} ::= m : \mathfrak{f}(E^*) = \{\text{Dec}^*\}$$

Module-level structural features are defined and used in the same way as above except for two subtleties. Firstly, the elaboration of a derived module

may only produce other modules. This makes sense as toplevel declaration must elaborate to other toplevel declarations.

Secondly, it is difficult how to specify where a module-level structural feature may be used. For derived declaration, which occur inside a theory, this is easy: the declaration may be used if the respective structural feature rule is visible to the containing theory. But derived modules, which may occur on toplevel, do not have a containing theory. It is not desirable to introduce a global scope that would define which module-level features are in scope as that would preclude restricting a module-level feature to specific object-languages. We are still experimenting with different designs for this issue. For now we use the containing file as the scope.

Diagram Definitions In [SR19], we added diagram expressions and diagram definitions in MMT. The former are expressions that use special constants to capture the syntax of MMT diagrams (i.e., the non-terminal γ). The latter are modules of the form `diagram` $d = E$. Their semantics is that *i*) E is evaluated into a diagram expression, say declaring theories $T_i = \{\Delta_i\}$ and *ii*) new modules $d/T_i = \{\Delta_i\}$ are created.

It is straightforward to realize this as a module-level structural feature. In fact, the implementation of structural features reported in this paper predates the work in [SR19], which already used them to implement diagram definitions.

Theory Morphisms and Logical Relations In Section 2, we mentioned that MMT supports other modules than theories. Two such features have been realized so far.

Firstly, **views** are modules of the form $v : S \to T = \{\text{Dec}^*\}$. These have been a primitive feature of MMT from the beginning. We can easily realize the **syntax** of views as derived modules. This is tempting because it would allow significantly simplifying the language. However, a currently unsolved problem is that the **semantics** cannot be reduced to elaboration: A view cannot in general be elaborated into anything simpler.

Secondly, [RS13] introduced logical relations as a module-level declaration. Rabe never implemented them in MMT because they, like views, are syntactically a special case of derived modules so that it made sense to defer their implementation until a general solution for derived modules is available. We intend to revisit them in future work.

6 Conclusion

We have presented a meta-language-based infrastructure of structural features in the MMT system and some paradigmatic examples that show its power. Structural features allow flexibly extending formal mathematical languages with new kinds of declarations without having to enlarge the trusted core of the system. In a meta-logical system, structural features are especially interesting because we need them to represent object languages and because the module system itself can restrict their availability to particular object logics.

The work presented here was to a large extent motivated by and developed for building exports of theorem prover libraries. In these, structural features allowed defining derived language features of theorem prover languages so that exports could stay shallow, i.e., structure-preserving, while also capturing the deep elaboration into kernel features that is needed for verification. Without the infrastructure presented in this paper, only deep implementations would have been possible and we would have been restricted to much less structured — and thus less searchable and reusable — exports. Moreover, it will prove critical for interoperability and library translations between theorem provers: even if target and source system have the exact same structural feature, a translation is practically very difficult if the intermediate representation is based on only the elaborated declarations.

In future work, we plan to represent more advanced features of theorem prover languages, starting with Isabelle and Coq. An open theoretical question is how to translate derived declarations along views in such a way that translation commutes with elaboration — this does not hold for every structural feature, and establishing sufficient criteria would be very valuable for modular reasoning in large libraries. Finally, we will improve MMT's abilities to represent the concrete syntax of derived declarations in order to mimic even more closely arbitrary object language syntax; this will allow for prototyping domain-specific languages in a way that entirely hides the logical framework from the user.

References

BHL+14. J. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, and D. Traytel. Truly modular (co)datatypes for Isabelle/HOL. In G. Klein and R. Gamboa, editors, *Interactive Theorem Proving*, pages 93–110. Springer, 2014.

CB16. D. Christiansen and E. Brady. Elaborator reflection: extending idris in idris. In J. Garrigue, G. Keller, and E. Sumii, editors, *International Conference on Functional Programming*, pages 284–297. ACM, 2016.

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

EUR+17. G. Ebner, S. Ullrich, J. Roesch, J. Avigad, and L. de Moura. A Metaprogramming Framework for Formal Verification. *Proceedings of the ACM on Programming Languages*, 1(ICFP):34:1–34:29, 2017.

Gor88. M. Gordon. HOL: A Proof Generating System for Higher-Order Logic. In G. Birtwistle and P. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer-Academic Publishers, 1988.

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

HHP93. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

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.

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

HR15. F. Horozal and Florian Rabe. Formal Logic Definitions for Interchange Languages. In M. Kerber, J. Carette, C. Kaliszyk, Florian Rabe, and V. Sorge, editors, *Intelligent Computer Mathematics*, pages 171–186. Springer, 2015.

IKRU13. M. Iancu, M. Kohlhase, F. Rabe, and J. Urban. The Mizar Mathematical Library in OMDoc: Translation and Applications. *Journal of Automated Reasoning*, 50(2):191–202, 2013.

KMOR17. M. Kohlhase, D. Müller, S. Owre, and F. Rabe. Making PVS Accessible to Generic Services by Interpretation in a Universal Format. In M. Ayala-Rincon and C. Munoz, editors, *Interactive Theorem Proving*, pages 319–335. Springer, 2017.

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

KR14. C. Kaliszyk and F. Rabe. Towards Knowledge Management for HOL Light. In S. Watt, J. Davenport, A. Sexton, P. Sojka, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 357–372. Springer, 2014.

KR20. Michael Kohlhase and Florian Rabe. Experiences from exporting major proof assistant libraries. submitted, 2020. URL: https://kwarc.info/people/frabe/Research/KR_oafexp_20.pdf.

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.

MRSC19. Dennis Müller, Florian Rabe, and Claudio Sacerdoti Coen. The Coq Library as a Theory Graph. In Cezary Kaliszyck, Edwin Brady, Andrea Kohlhase, and Claudio Sacerdoti Coen, editors, *Intelligent Computer Mathematics (CICM) 2019*, number 11617 in LNAI. Springer, 2019. doi:10.1007/978-3-030-23250-4.

Mül19. Dennis Müller. *Mathematical Knowledge Management Across Formal Libraries*. PhD thesis, Informatics, FAU Erlangen-Nürnberg, 10 2019. URL: https://kwarc.info/people/dmueller/pubs/thesis.pdf.

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

Rab17. Florian Rabe. How to Identify, Translate, and Combine Logics? *Journal of Logic and Computation*, 27(6):1753–1798, 2017.

RK13. Florian Rabe and Michael Kohlhase. A scalable module system. *Information & Computation*, 0(230):1–54, 2013. URL: http://kwarc.info/frabe/Research/mmt.pdf.

RM18. Florian Rabe and Dennis Müller. Structuring theories with implicit morphisms. 2018. URL: http://wadt18.cs.rhul.ac.uk/submissions/WADT18A43.pdf.

Rot20. Colin Rothgang. Theories as inductive types, 05 2020. B.Sc. Thesis, expected May 2020.

RS13. Florian Rabe and Kristina Sojakova. Logical Relations for a Logical Framework. *ACM Transactions on Computational Logic*, 2013. URL: http://kwarc.info/frabe/Research/RS_logrels_12.pdf.

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.