Understanding the Pragmatics of Module Systems for Mathematics

Mihnea Iancu, Michael Kohlhase, Florian Rabe

Computer Science, Jacobs University, Bremen, Germany
initial.last@jacobs-university.de

Abstract. Knowledge representation languages for mathematics must balance expressivity and minimality. The former increases coverage and ease of modeling whereas the latter simplifies implementing and meta-logical reasoning.
While extensibility at the object-level is well-understood, the richness of mathematical language has made it difficult to approach pragmatic module-level features systematically. Indeed, state-of-the-art languages provide a wide variety of advanced structuring constructs, often differing strongly across languages.
In this paper we propose a novel concept for the OMDoc/MMT format that allows naturally representing many important pragmatic features as well as their semantics. This includes such diverse features as generative functor applications, record types, (co)inductive types, sections, structured proofs, and realms. Most notably, these pragmatic features take the form of self-contained, maximally reusable modules that can be flexibly instantiated when modeling languages in MMT. We have implemented this new concept in the MMT system along with modules for important individual features.

1 Introduction

Designing a representation language for mathematics suited to mathematical knowledge management involves adequately capturing those aspects of mathematical knowledge that are relevant for machine-driven practical applications. For instance, proof checking requires explicit representations of the semantics of types and definitions but can largely ignore structure. Meanwhile, change management needs to introspect the structures used to organize the knowledge (e.g. namespaces, modules, records) and compute dependencies between them. In practice, we often observe a trade-off between a shallow encoding in a strong language and a deep encoding in a simple language.

For restricted domains, language, system and applications are often designed together for a specific goal (e.g. in a proof assistant), but this makes it hard to repurpose the system for new applications. For generic systems and languages on the other hand (like Mmt and OMDoc), it is desirable to provide a rich structure level in order to allow adequately representing the structuring mechanisms used in mathematics. However, generic approaches must also identify a minimal

set of structuring primitives to allow for simple meta-theoretical reasoning or complex algorithms such as parsing or type inference.

In this paper, we propose a language design that addresses this trade-off in generic representation languages. Our goal is to provide a scalable, extensible structure level. Our design allows complex structural language features to be defined from the ground up in terms of a basic set of primitive features. Structured declarations can be *strictified* by elaborating them to the core language when needed so that all algorithms can work on arbitrarily complex declarations. Therefore, the rich structure can be leveraged by some applications while being ignored by others.

The basic idea of this design is not new. Indeed, proof assistants like Coq [Coq15] or Isabelle [Pau94] are implemented in a way that allows programmers to add new kinds of declarations, which are elaborated into lower-level declarations. This is used for example in the datatype package of Isabelle [Bla+14]. Our main contribution is to capture the extension process at a very high-level of generality while still being able to fix the abstract syntax of the extended language once and for all. This is crucial to allow knowledge management algorithms (in particular, the many algorithms already part of MMT) to be implemented generically, i.e., independent of what structural features are added in the feature.

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. Finally, Section 5 concludes the paper and discusses future work.

2 MMT/OMDoc

OMDoc [Koh06] is a rich representation language for mathematical knowledge with a large set of primitives motivated by expressivity and user familiarity. The MMT [RK13b] language is a complete redesign of the formal core of OMDoc focusing on foundation-independence, scalability, modularity and minimality.

OMDoc and MMT exemplify the trade-off discussed above and this paper is part of an ongoing effort to extend the MMT core with a layer of structural extensions which recover the expressivity of OMDoc. This effort was started in [HKR12] which introduced an extension language for the statement level. It provided syntactic means for defining pragmatic language features and their semantics in terms of strict OMDoc. It this paper, we take a step further and give a much more expressive extension mechanism for both the statement and module levels. Then, the pragmatics from [HKR12] become a special case.

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** containing **modules** which are either **theories** or **views**. MMT theories are named sets of statements and are used to represent formal constructs such as logical frameworks, logics, and theories. At the **statement** 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 and include variable and symbol references as well as application and binding. Finally, Mmt views are morphisms between two theories that map constants in the source theory to expressions in the target theories. They are used to represent a variety of concepts including models of a theory, implementations of a specification and instances of a class. Finally, **global names** (or Mmt URIs) provide unique global identifiers for each Mmt declaration, which are essential for scalability. Concretely, a Mmt module with local name m in diagram γ has URI $\gamma?m$ and a statement with local name s inside m has URI $\gamma?m?s$. Additionally, we use the term **declarations** to encompass both Mmt modules and statements.

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 reasons of brevity. We only emphasize that Mmt is foundation-independent: The syntax does not assume any special constants (e.g., λ), and the semantic 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 [Rab14].

Declaration	D	::=	$M \mid S$
Module Level			
Diagram	γ	::=	M^*
Module	M	::=	$T \mid V$
Theory	T	::=	$l = \{S^*\}$
View	V	::=	$l : g \to g = \{S^*\}$
Statement Level			
Statement	S	::=	$C \mid I$
Constant	C	::=	$l[:E][=E]$
Include	I	::=	`include` g
Object Level			
Expression	E	::=	$l \mid g \mid E\ E^* \mid E\ \Gamma.E$
Context	Γ	::=	$\cdot \mid \Gamma, l[:E]$
Identifiers			
Global name	g	::=	$URI[?l[?l]]$
Local name	l	::=	`String`

Fig. 1: Mmt Grammar

3 Structural Features in MMT

Our extension of Mmt starts from the observation that statement and module-level declarations often have internal structure that differs from the structure of their semantics.

Definition 1. Let X be a set, then $\{\emptyset, X\}$ is a topology on X, it is called the **trivial topology** and $\langle X, \{\emptyset, X\}\rangle$ the **indiscrete space** over X.

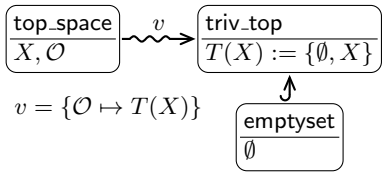

Fig. 2: Structural Dimorphism in Mathematics

3.1 Motivation

Consider the hybrid of a definition and a theorem on the left of Figure 2: This kind of "definition" is commonly used in informal mathematics and pragmatically combines two actions into one aggregated form: *i*) the assertion that $\{\emptyset, X\}$ is a topology on X and *ii*) defining the **trivial topology** as $\{\emptyset, X\}$ and the **indiscrete space** over X as $\langle X, \{\emptyset, X\}\rangle$. In MMT, this would correspond to the theory graph on the right in Figure 2. But note that the informal definition and the formal theory graph have corresponding sub-structures – here names and definienda. We call this phenomenon **structural dimorphism**[1]

To account for this we introduce a new kind of structural element which we call **pragmatic declaration** in MMT that has two distinct sets of declarations: one for the **internal** perspective of the declaration, namely the components from which it is formed and one for the **external** perspective, namely its semantics as visible from the containing module or theory graph. These two aspects are typically conflated in mathematical representation languages but making them explicit allows us to capture uniformly many commonly occurring phenomena from both formal languages and common mathematical language (CML).

In our example above, the full content of Figure 2 can be seen as the instance of a particular pragmatic declaration we need for understanding CML. But pragmatic declarations can do more: they allow us to uniformly model and adequately represent the internal/external aspects of structurally complex declaration containers such as records or structured proofs (with inline lemmas, sub-proofs, etc.). That is what we focus on in this paper, we model it as the case where the externals of a pragmatic declaration are practically inferable (or computable) from its internals in a process we call **elaboration**. We call such pragmatic declarations, **derived declarations**. Note that the practicality of elaboration is not a fixed point but is relative to the efficiency of algorithms and, in the informal case, the capability of natural language understanding. In particular, the definition on the left of Figure 2 cannot currently be automatically elaborated into the theory graph on the right.

3.2 Formal Definitions

Syntax We introduce two additional primitives to the MMT system and language. Firstly, **structural features** define the syntax and (via elaboration)

[1] cf. the usage in crystallography the phenomenon that some substances have two chemically identical but crystallographically distinct forms.

semantics of a new pragmatic declaration. Secondly, **derived declarations** are concrete instances of these structural features.

To allow full flexibility for language designers the former is not syntactically constrained: A structural feature is an arbitrary elaboration function. However, derived declarations are part of the MMT grammar, and the same general syntax is used for derived declarations of all features. Thus, we can introduce almost arbitrary new features and still have algorithms that treat them uniformly.

Definition 1 (Derived Declarations). *We extend the* MMT *grammar in Figure 1 with the following production:*

$$\textit{Derived declarations} \quad D_d \quad ::= \quad l : l\ E^* = \{S^*\}$$

*Given a derived declaration $l : \mathfrak{f}\ E_1, \ldots, E_m = \{S_1, \ldots, S_n\}$, we call l its **name**, $\mathfrak{f}$ its **feature**, the E_i its **arguments**, and the S_i its **internal declarations**.*

We differentiate between derived statements D_s that occur inside theories and derived modules D_m that occur inside a diagram. For simplicity, we do not cover the case of derived declarations in views. They are redundant because a view can instead provide a constant declaration for any constant in the elaboration of its domain. However, it is also possible (and often useful) for views to contain derived declarations, and this is supported by our implementation.

Definition 2 (Structural Features). *A **structural feature** is a tuple $\langle \mathfrak{f}, n, \mathcal{E} \rangle$ for a name $\mathfrak{f}$ and a number $n \in \mathbb{N}$.*

*Here the elaboration function $\mathcal{E}$ must map any derived declaration D_d of the form $l : \mathfrak{f}\ E_1, \ldots, E_n = \{\Sigma_S\}$ to a list of declarations Σ_D which we call the **external declarations** of D_d.*

Structural features are extra-linguistic and are defined in papers (as in Section 4) or implemented in MMT plugins, but not necessarily in MMT syntax. However, not every MMT theory may use any structural feature — we piggyback on MMT's modular foundation management: We define that a derived declaration can only be well-formed if its feature is provided by or included into the current foundation.

Inference System The judgments that we need to define the semantics of derived declarations are given in Figure 3. Most of these are essentially the same as used for MMT in [RK13a].

The key novelty are the judgments for elaboration. $\gamma \vdash_m S \rightsquigarrow \Sigma$ defines result of elaborating a statement S, and $\gamma \vdash M \triangleright \Sigma$ describes the result of elaborating all statements in a module. Similarly, $\gamma \vdash M \rightsquigarrow \Sigma_M$ defines result of elaborating a module M, and $\gamma \vdash M \triangleright \Sigma_M$ describes the result of elaborating all modules in a diagram.

To simplify the inference system at this point, we assume that elaboration is fully recursive, i.e., its result is always a list of strict MMT declarations.

To define the elaboration semantics, we crucially add rules for the lookup judgment: $\vdash \gamma(g) \Rightarrow S$ describes that the identifier g resolves to the statement

Judgment	Intuition	new
$\vdash \gamma$	γ is a well-formed theory graph γ	
$\gamma \vdash M$	adding M at the end of γ preserve well-formedness of γ	
$\gamma \vdash_m D$	if m is the URI of a module in γ, then adding D at the end of the body of m preserves well-formedness of γ	
$\gamma, \Gamma \vdash_T E$ `wff`	expression E is well-formed over theory T in context Γ	
$\gamma \vdash_m S \rightsquigarrow \Sigma$	statement S in module m elaborates to Σ	✓
$\gamma \vdash M \triangleright \Sigma$	the elaborated body of the module M is Σ	✓
$\gamma \vdash M \rightsquigarrow \Sigma_m$	module M in diagram γ elaborates to Σ_M	✓
$\vdash \gamma \triangleright \Sigma_M$	the elaboration of a diagram γ is Σ_M	✓
$\vdash \Sigma(l) \Rightarrow D$	Σ contains declaration D with name l	
$\vdash \gamma(g) \Rightarrow S$	looking up URI g in γ yields (module or declaration) S	

Fig. 3: Judgments

S. Thus, by defining appropriate identifiers and their lookup, we can control how users can refer to the declarations obtained by elaboration.

$$\frac{}{\vdash \cdot} \qquad \frac{\vdash \gamma(T) \Rightarrow \text{undefined}}{\gamma \vdash T = \{\cdot\}} \qquad \frac{\vdash \gamma(v) \Rightarrow \text{undefined}}{\gamma \vdash v : S \to T = \{\cdot\}}$$

$$\frac{\vdash \gamma(m?c) \Rightarrow \text{undefined} \quad [\gamma, \cdot \vdash_m E \texttt{ wff}] \quad [\gamma, \cdot \vdash_m E' \texttt{ wff}]}{\gamma \vdash_m c[:E][=E']}$$

$$\frac{\vdash \gamma(m') \Rightarrow M \quad m, m' \text{ compatible}}{\vdash_m \texttt{include } m'}$$

Fig. 4: Basic Rules for Building Diagrams

Our rules subtly change the existing rules for Mmt. Therefore, we repeat the most important rules needed to understand the effect of our new rules in Figure 4 and 5.

The rules in Figure 4 govern the building of diagrams by adding statements one by one. For reasons of brevity, all rules omit the hypotheses necessary for well-*typed*ness. For example, $\gamma \vdash_m c : E = E'$ requires that E' has type E. Similarly, we gloss over the details of include declarations. By saying that m and m' must be compatible we mean that modules m may include module m' only if that does not conflict with the declarations already in m. Most importantly, each declaration in a theory must have a unique name, and a view may provide at most one declaration for each constant in its domain.

The rules in Figure 5 define the well-formed expressions relative to a theory T. We omit the straightforward but tedious rules for binders as well as the ones for views.

$$\frac{\vdash \gamma(T?c) \Rightarrow c[:E][=E']}{\gamma, \cdot \vdash_T c\ \texttt{wff}} \qquad \frac{\vdash \Gamma(x) \Rightarrow x[:E]}{\gamma, \Gamma \vdash_T x\ \texttt{wff}}$$

$$\frac{\gamma, \Gamma \vdash_T E_i\ \texttt{wff} \quad i = 0, \ldots, n}{\gamma, \Gamma \vdash_T E_0\, E_1 \ldots E_n\ \texttt{wff}} \qquad \frac{\text{accordingly}}{\gamma, \Gamma \vdash_T E_0\, \Gamma_0.E_1\ \texttt{wff}}$$

Fig. 5: Rules for Building Expressions

$$\frac{}{\gamma \vdash T = \{\cdot\} \triangleright \cdot} \qquad \frac{}{\vdash \cdot \triangleright \cdot}$$

$$\frac{\gamma \vdash T = \{\Theta\} \triangleright \Theta' \quad \gamma \vdash_T D \rightsquigarrow \Sigma}{\gamma \vdash T = \{\Theta, D\} \triangleright \Theta', \Sigma} \qquad \frac{\vdash \gamma \triangleright \gamma' \quad \gamma \vdash M \rightsquigarrow \Sigma_M}{\vdash \gamma, M \triangleright \gamma', \Sigma_M}$$

$$\frac{}{\gamma \vdash_m \texttt{include}\, m \rightsquigarrow \texttt{include}\, m} \qquad \frac{}{\gamma \vdash_m c[:E][=E'] \rightsquigarrow c[:E][=E']}$$

$$\frac{\gamma \vdash T = \{\Sigma\} \triangleright \Sigma'}{\gamma \vdash T = \{\Sigma\} \rightsquigarrow T = \{\Sigma'\}} \qquad \frac{}{\gamma \vdash v : S \to T = \{\Sigma\} \rightsquigarrow v : S \to T = \{\Sigma\}}$$

Fig. 6: Rules for Elaboration

Then we are ready to state our new rules. The rules in Figure 6 govern elaboration. Modules are elaborated declaration-wise, and include and constant declarations elaborate to themselves. Similarly, diagrams are elaborated module-wise, theories elaborate recursively and views elaborate to themselves. The rules for elaborating a derived declaration must be provided by the respective structural feature.

While structural feature may define the elaboration arbitrarily, the basic syntax of derived declarations is fixed. This is captured by the rules in Figure 7. They describe the syntactic properties of a derived declaration D of the form $d : \mathfrak{f}\ E_1, \ldots, E_n = \{\Sigma\}$ relative to its container C which is a theory for a derived statement and a diagram for a derived module. We may append D to C if : (i) d is a fresh name in C; (ii) all arguments E_i are well-formed expressions over C; (iii) the declarations in Σ are well-formed relative to C. If C is a diagram this implies the only well-formed arguments are references to modules and the only valid features are MMT-level features, since nothing else is visible outside of a theory.

$$\frac{\vdash \gamma(T?d_s) \Rightarrow \text{undefined} \quad \gamma, \cdot \vdash_T E_i \ \texttt{wff} \quad i = 1, \ldots, n \quad \vdash \gamma, T/d_s = \{\texttt{include}\ T,\ \Sigma\}}{\gamma \vdash_T d_s : \mathfrak{f}\ E_1, \ldots, E_n = \{\Sigma\}}$$

$$\frac{\vdash \gamma(d_m) \Rightarrow \text{undefined} \quad \gamma, \cdot \vdash E_i \ \texttt{wff} \quad i = 1, \ldots, n \quad \vdash \gamma, d_m = \{\Sigma\}}{\gamma \vdash d_m : \mathfrak{f}\ E_1, \ldots, E_n = \{\Sigma\}}$$

Fig. 7: Foundation-Independent Rules for Derived Declarations

3.3 Referencing Derived Declarations

In addition to the elaboration rules above, we also extend the rules for URIs in order to preserve the crucial MMT invariant that every declaration has an unique URI. For the scope of this paper, we adopt the perspective that elaboration is fully recursive (rather than incremental). Therefore we do not need URIs for partially elaborated forms in the case of derived declarations nested into each other. However, we do note that incremental elaboration is useful in practice (e.g. for efficiency or for algorithms implementing special behavior for particular features) and we do cover it in the implementation.

The judgments are given in Figure 8 and formalize URI-based lookup in MMT diagrams with respect to derived declarations. Their behavior follow the intuitions described above and allow acess to both the internal and external declarations of each derived declaration. Lookup for non-derived declarations is unaffected.

$$\frac{\vdash \gamma(m) \Rightarrow M \quad \gamma \vdash M \ \triangleright\ \Sigma \quad \vdash \Sigma(s) \Rightarrow S}{\vdash \gamma(m?s) \Rightarrow S}$$

$$\frac{\vdash \gamma(m?d) \Rightarrow d : \mathfrak{f}\ E = \{\Sigma\} \quad \vdash \Sigma(s) \Rightarrow S}{\vdash \gamma(m/d?c) \Rightarrow C}$$

$$\frac{\vdash \gamma \ \triangleright\ \gamma' \quad \vdash \gamma'(m) \Rightarrow M}{\vdash \gamma(m) \Rightarrow M} \qquad \frac{\vdash \gamma(m) \Rightarrow m : \mathfrak{f}\ E = \{\Sigma\} \quad \vdash \Sigma(s) \Rightarrow S}{\vdash \gamma(m?c) \Rightarrow C}$$

Fig. 8: Rules for Lookup of URIs

4 A Library of Structural Features

We apply our language to obtain a library of reusable structural features. Thus, when representing languages in MMT, users can import the precise structural features needed for their target language (or define their own features if not yet in the library). Our objective is to explore the breadth of applications; therefore, we have opted to give many features even if that means that not every feature can be defined in all detail.

Before describing individual features, we introduce some auxiliary definitions. In the MMT implementation, these are provided as utility functions.

Definition 3 (Renaming). For a partial function r from names to names, we write $[r]\Sigma$ for the result of replacing in Σ for every c with $r(c) = d$
- every declaration named c with the corresponding declaration named d,
- every expression E with the expression obtained by replacing every occurrence of a name c or c/l in E with d or d/l, respectively.

We write $[c/\_]\Sigma$ for $[\ldots, n \mapsto c/n, \ldots]\Sigma$ where the ellipsis runs over the names of all declarations in Σ.

Definition 4 (Translation). For a partial function t from names to expressions, we write $\overline{t}(E)$ for the result of replacing in E every occurrence of a name c for which $t(c) = E'$ with E'.

4.1 Derived Statements

Local Scopes Structuring mathematical developments by grouping similar or related declarations together is common in both informal and formal mathematics. For example, LaTeX uses sectioning and environments, OMDoc uses `<tgroup>` elements, and many proof assistants use modules.

The structural feature $\mathtt{scope}$ of arity 0 yields the simplest special case. It is defined by $\gamma \vdash_T s : \mathtt{scope} = \{\Sigma\} \rightsquigarrow [s/\_]\Sigma$. Here the external declarations are the same as the internal ones except for qualifying them with the name of the scope.

Sections Sections are similar to local scopes except they may contain local variables, i.e., constants which are in scope only inside the section but not visible from the outside. This is well-known from informal mathematics and also part of some proof assistants like Coq (which allows declaring `Variable`s in a `Section`). The idea is that some constants (the local variables) (i) behave like normal constants from the perspective of the internal declarations, (ii) but do not appear among the external declarations because all externals abstract over them.

We use a structural feature $\mathtt{section}$ with arity 0 and define $\gamma \vdash_T s : \mathtt{section} = \{\Sigma_0\} \rightsquigarrow \Sigma_2$ as follows. First, let Σ_1 be the recursive elaboration of Σ_0. We define Σ_2 by induction on the declarations in Σ_1:
- For every constant without definiens, nothing. These are the local variables.

– For every constant declaration C of the form $c[\,:E] = E'$, let Γ be the context containing all local variable declarations that precede C. Then Σ_2 contains $c[\,:\Pi\Gamma.\bar{t}(E)] = \lambda\Gamma.\bar{t}(E')$ where t is defined below.

Abstracting away local variables is not foundation-independent: It requires a foundation with a Π and λ binder such as LF. This is necessary to bind the local variables as seen above. Thus, the feature `section` must be defined in a foundation that imports a theory for a dependently-typed λ-calculus.

Moreover, because every constant c elaborates to a function that takes the local variables as arguments, every reference to it must be applied to these local variables. This is the purpose of the translation t: It translates every reference a Σ_1-constant c that is not a local variable to $@\,c\,c_1\,\ldots\,c_n$, where the c_i are the names of the constants in Γ. Here $@$ is the constant for function application that goes with Π and λ.

There are multiple variants of this feature. Firstly, Σ_2 does not use the name of the section as a qualifier for the external declarations. Alternatively, we could qualify the declarations as with `scope`. Secondly, Σ_2 abstracts over all local variables that precede a constant declaration C. Alternatively, we could abstract only over those that actually occur in C. Thirdly, we could also use local definitions, i.e., specially marked defined constants that are eliminated during elaboration, namely by expanding their definitions.

Generative Pushout Generative pushouts are already a primitive part of the MMT language [RK13a], where they are called structures due to their inspiration by the SML module system. We can now recover them as a special unary structural features `structure`.

The intuition behind $\gamma \vdash_T s : \texttt{structure}\ E = \{\Phi\} \rightsquigarrow \Sigma'$ is that
– E is an expression evaluating to some theory, say with body Σ,
– Φ is interpreted as a partial morphism from Σ to T, say with domain Σ_0,
– adding Σ' to T yields the pushout of Φ and the inclusion $\Sigma_0 \hookrightarrow \Sigma$.

Concretely, Σ' arises from $[s/\_]\Sigma$ by merging in all declarations in Φ. For example, if Σ contains $c : A$ and Φ contains $c = a$, then Σ' contains $s/c : [s/\_]A = a$. We refer to [RK13a] for the details.

Structured Proofs Both formal and informal proofs are usually highly structured, using local definitions and assumptions for intermediate proof steps. Core MMT, on the other hand, represents proofs as expressions that correspond to derivations in an appropriate inference system.

We introduce a unary structural feature `proof` that elaborates into a single declarations. We have $\gamma \vdash_T s : \texttt{proof}\ E = \{\Sigma\} \rightsquigarrow s : E = p$, i.e., E is the assertion to prove and the structured proof Σ is flattened into a single proof object p. For a given Σ, we define p by induction on Σ using a judgment $\Sigma \gg p$.

There are various declarations that are useful to allow in Σ. The simplest case is a defined constant, which can represent both a local definition and a local lemma:

$$\frac{\Sigma \gg p}{c[\,:E] = E',\ \Sigma \gg \overline{c \mapsto E'}(p)}$$

The second most important case are undefined constant declarations. In a foundation with universal quantifier $\forall$ and implication $\Rightarrow$ and corresponding introduction rules $\forall_I$ and $\Rightarrow_I$, they can represent a local parameters or a local assumption, respectively. These can be elaborated into the application of the corresponding introduction rule: Depending on whether E is a proposition (left) or a type (right), we put

$$\frac{\Sigma \gg p}{c : E,\ \Sigma \gg \forall_I c : E.p} \qquad \frac{\Sigma \gg p}{c : E,\ \Sigma \gg \Rightarrow_I c : E.p}$$

Recall that derived declarations can be nested. Nested structured proofs can be used to elegantly represent case distinctions and similar proof principles, which have to branch into one structured proof for each case. Because every structured proof elaborates into a single defined constant, the case for defined constants above is already enough to handle the elaboration of nested structures proofs.

The cases above are already expressive enough to represent basic OMDoc proofs [Koh06]. There the proof steps are *symbol declarations* (which correspond to parameters above), *definitions* (local definitions), *hypotheses* (local assumptions) and *derivations* which can be direct (lemmas) or OMDoc subproofs (nested derived proofs). But, we can also define auxiliary structural features that are only allowed inside a $\mathfrak{proof}$ to capture more advanced proof principles.

(Co-)Inductive Data Types Most proof assistants support inductive data types, either as a primitive feature or as an elaborated feature. They can be very naturally represented using a structural feature $\mathfrak{inductive}$ of arity 0. The intended syntax and semantics are best explained by example:

$$Nat : \mathfrak{inductive} = \{n : \mathtt{type},\ z : n,\ s : n \to n\}$$

i.e., the internal declarations declare the constructed type and its constructors. For the elaboration, we have $\gamma \vdash_T i : \mathfrak{inductive} = \{\Sigma\} \rightsquigarrow [i/\_]\Sigma,\ G$, where G is the list of generated induction axioms and recursion schemata. Coinductive data types can be represented accordingly using a feature $\mathfrak{coinductive}$.

The details of the allowed internal declarations and the generated external declarations in G may vary widely depending on the used foundation and the choice of (co)induction principles. For example, we need at least some kind of simple type theory to state the internal declarations. However, we cannot allow any internal declaration: It is usually required that the return type of each constructor is declared inside Σ, and negative occurrences of the constructed type in the constructors may be restricted.

Note that our representation requires explicitly declaring the constructed type. This is advantageous because it allows representing many different variants uniformly. For example, if we allow more than one internal type declaration, we obtain groups of mutually inductive types. If we allow parametric types, e.g., $\mathtt{list} : \mathtt{type} \to \mathtt{type}$, we obtain families of inductive types.

Similarly, we can generalize what constructors are allowed. If we allow identity types as return types, we can represent the higher inductive types recently studied in homotopy type theory [The15]. If we allow declaring rewrite rules, we can define inductive functions for selectors and testers as in PVS [ORS92] or for induction-recursion style definitions as in Agda [Nor05].

Record Types Record types can be treated dually to inductive types. Indeed, we can think of them as labeled product types (i.e., with named rather than numbered projections) and of the latter as (recursive) labeled coproduct types (with named injections). Thus, the internal declarations could contain a type $r : \texttt{type}$ and some projections of the form $p_i : r \to E_i$.

However, we can also use a slightly simpler structural feature that is often more elegant. Here we omit the internal declaration of r and only declare the fields as $p_i : E_i$. Then we have, for example,

$$\gamma \vdash_T r : \mathfrak{record} \ = \{\dots, f_i : E_i, \dots, \} \ \rightsquigarrow$$

$$r/type : \texttt{type},\ r/make : E_1 \to \dots \to E_n \to E' \to r/type,\ \dots, r/f_i : r/type \to E_i,\ \dots$$

The case of dependent record types is handled accordingly.

4.2 Derived Modules

Diagrammatic Pushouts A convenient way of defining theories, particularly in highly modular domains such as algebraic structures is to combine two theories to produce a larger one. A particularly common case is that of pushouts – shown in Figure 10 – where two theories B and B' that share an A via two morphisms m_B and m'_B, are joined to produce a new theory C. C and the induced morphisms i_B and $i_{B'}$ cannot be constructed generically as it is non-trivial how to enforce that i_B and i'_B agree on the assignments for A-symbols. But, in the presence of a logic with strong enough equality, this can be enforced by adding the corresponding equality judgments at the logic level.

But, another issue which is avoiding name clashes (by producing unique names) for the statements constructed in C. The standard solution for Mmt is to use different local scopes (as described above in Section 4.1) for statements produced via B and B'. But this can quickly result in unwieldy theories with unintuitive statement names. An alternative, described in [CO12] is to give renamings as additional arguments to the combine operation which are used to produce the names in C.

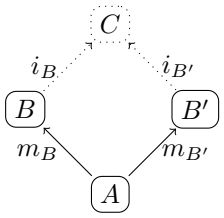

Fig. 9: Elaboration of diagrammatic pushouts

Below, we introduce diagrammatic pushouts as a binary structural feature $\mathfrak{combine}$ with a semantics inspired from [CO12]. The arguments are the two morphisms m_B and $m_{B'}$ therefore inducing their shared domain (A), and their codomains (B and B'). The default generated names in the elaboration are added a local scope. Then, the internals can be used to add unscoped names

for all or some statements by giving aliases as MMT constants. We define the elaboration of a derived module $C = c : \mathfrak{combine}\ m_B, m_{B'} = \{\Sigma\}$ as containing the following:

- the import from B as the generative pushout $L = \mathit{left} : \mathfrak{structure}\ B = \{\cdot\}$. We denote by i_L the morphism from B to C induced by L.
- the import from B' as the generative pushout $R = \mathit{right} : \mathfrak{structure}\ B' = \{\cdot\}$ with i_R being the induced morphism.
- an equality judgement $ax/c_a = \vdash i_L(m_B(c_a)) \doteq i_R(m_{B'}(c_a))$ for each statement $c_a \in A$, where $\doteq$ is the equality of the current logic.
- the aliases in Σ, for instance $c = left/c_b$ for adding c as an alias for the constant c_b from B.

Theory Isomorphisms Different definitions or developments yielding equivalent concepts is a regular occurrence in both formal and informal mathematics. In module systems like MMT, this typically manifests as theories that are *isomorphic*. We introduce theory isomorphism as a binary structural feature $\mathfrak{isomorphism}$ where the two arguments refer to the theories in question, and the internals are the pairs of equivalent symbols that define the isomorphism represented as MMT constants.

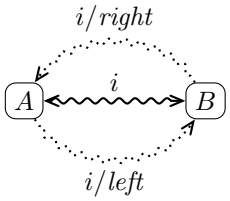

Fig. 10: Elaboration of Theory Isomorphisms

We define the elaboration of a derived declaration $i : \mathfrak{isomorphism}\ A, B = \{\Sigma\}$ to be the two views $i/\mathit{left} : A \to B = \{\Sigma\}$ and $i/\mathit{right} : B \to A = \{\Sigma_r\}$ where Σ_r is constructed inductively as follows:

- Σ_r starts out as the empty body $\{\cdot\}$
- for every constant $c_a = c_b$ in Σ (representing a symbol pair) we add $c_b = c_a$ to Σ_r

Effectively, Σ_r is the reverse of Σ and we obtain the diagram in Figure 10 as the result of elaboration.

Realms A realm [CFK14] is a theory graph construct formed from a set of equivalent formal developments called *pillars* together with an *interface theory* that aggregates the symbols from each pillar. The pillars are sets of theories where each of them is a conservative extension of one distinguished theory called the *base*. Realms are useful because they abstract from an underlying formal development and provide practitioners with just the useful symbols and theorems via the interface theory. Common examples are the different ways to define natural or real numbers, groups or topologies.

For a formal description of realms we refer to [CFK14] and also to [IK15] where an algorithmic procedure for producing the interface theory from the pillars was developed. Here, we simply reframe realms as a structural extension and omit the details.

We first define pillars as an auxiliary structural feature $\mathfrak{pillar}$ with flexible arity. The arguments represent (references to) the theories that form the pillar with the first argument being the base theory.

Then, a realm is a nullary structural feature $\mathfrak{realm}$ whose internal declarations are either pillars or isomorphisms. The elaboration function for a realm $r : \mathfrak{realm} = \{\Sigma\}$ with n pillars is a theory $r/face$ (the interface theory) together with n views r/v_i that justify the interface theory as an abstraction of each pillar. The theory and views are constructed following the *face-generation* algorithm described in [IK15] which basically traverses the pillars to produce the face by aggregating statements that are unique modulo the given isomorphisms. Figure 11 shows the diagram obtained as the result of elaboration.

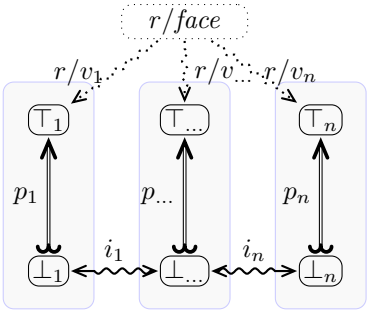

Fig. 11: Elaboration of a Realms

5 Conclusion and Future Work

In this paper we returned to the ever-present design question for mathematical representation languages: *how can we balance expressivity pragmatic adequacy with scalability and implementability issues*. We identified structural dimorphism between the external and internal views as a central property of many pragmatic language features and used it as the main component of a new language feature in the OMDoc/Mmt format: derived declarations where the internal declarations can be elaborated to external ones. We have worked out the details of this extension and have shown that the new feature can capture many extensions of Mmt that were present in OMDoc and even re-interpret some of the features of OMDoc itself. The extension and re-interpretation have been implemented in the Mmt system and have led to a simplification and further regularization of the code base.

The primary goal for future research is to extend the notion of derived declarations to full "pragmatic declarations", where current technology does not allow elaboration, and we need to resort to parallel markup at the structure level instead. We conjecture that most of the mechanisms presented in this paper still hold.

Acknowledgements We acknowledge financial support from the OpenDreamKit Horizon 2020 European Research Infrastructures project (#676541).

References

[Bla+14] J. Blanchette et al. "Truly Modular (Co)datatypes for Isabelle/HOL". In: *Interactive Theorem Proving*. Ed. by G. Klein and R. Gamboa. Springer, 2014, pp. 93–110.

[CFK14] Jacques Carette, William Farmer, and Michael Kohlhase. "Realms: A Structure for Consolidating Knowledge about Mathematical Theories". In: *Intelligent Computer Mathematics 2014*. Conferences on Intelligent Computer Mathematics. (Coimbra, Portugal, July 7–11, 2014). Ed. by Stephan Watt et al. LNCS 8543. MKM Best-Paper-Award. Springer, 2014, pp. 252–266. URL: http://kwarc.info/kohlhase/submit/cicm14-realms.pdf.

[CO12] J. Carette and R. O'Connor. "Theory Presentation Combinators". In: *Intelligent Computer Mathematics*. Ed. by J. Jeuring et al. Vol. 7362. Springer, 2012, pp. 202–215.

[HHP93] R. Harper, F. Honsell, and G. Plotkin. "A framework for defining logics". In: *Journal of the Association for Computing Machinery* 40.1 (1993), pp. 143–184.

[HKR12] Fulya Horozal, Michael Kohlhase, and Florian Rabe. "Extending MKM Formats at the Statement Level". In: *Intelligent Computer Mathematics*. Conferences on Intelligent Computer Mathematics (CICM). (Bremen, Germany, July 9–14, 2012). Ed. by Johan Jeuring et al. LNAI 7362. Berlin and Heidelberg: Springer Verlag, 2012, pp. 65–80. URL: http://kwarc.info/kohlhase/papers/mkm12-p2s.pdf.

[IK15] Mihnea Iancu and Michael Kohlhase. "Math Literate Knowledge Management via Induced Material". In: *Intelligent Computer Mathematics 2015*. LNCS. submitted. Springer, 2015, pp. 187–202. URL: http://kwarc.info/kohlhase/papers/cicm15-induced.pdf.

[Koh06] Michael Kohlhase. OMDoc – *An open markup format for mathematical documents [Version 1.2]*. LNAI 4180. Springer Verlag, Aug. 2006. URL: http://omdoc.org/pubs/omdoc1.2.pdf.

[Nor05] U. Norell. *The Agda WiKi*. http://wiki.portal.chalmers.se/agda. 2005.

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

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

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

[RK13a] F. Rabe and M. Kohlhase. "A Scalable Module System". In: *Information and Computation* 230.1 (2013), pp. 1–54.

[RK13b] Florian Rabe and Michael Kohlhase. "A Scalable Module System". In: *Information & Computation* 0.230 (2013), pp. 1–54. URL: http://kwarc.info/frabe/Research/mmt.pdf.

[Coq15] Coq Development Team. *The Coq Proof Assistant: Reference Manual*. Tech. rep. INRIA, 2015.

[The15] The Univalent Foundations Project. *Homotopy Type Theory*. Univalent Foundations Project, 2015.