An Exchange Format for Modular Knowledge

Florian Rabe, Michael Kohlhase
Jacobs University Bremen

Abstract

We present a knowledge exchange format that is designed to support the exchange of modularly structured information in mathematical and logical systems. This format allows to encode mathematical knowledge in a logic-neutral representation format that can represent the meta-theoretic foundations of the systems together with the knowledge itself and to interlink the foundations at the meta-logical level. This "logics-as-theories" approach, makes system behaviors as well as their represented knowledge interoperable and thus comparable. The advanced modularization features of our system allow communication between systems without losing structure.

We equip the proposed format with a web-scalable XML/URI-based concrete syntax so that it can be used as a universal interchange and archiving format for existing systems.

1 Introduction

Mathematical knowledge is at the core of science, engineering, and economics, and we are seeing a trend towards employing computational systems like semi-automated theorem provers, model checkers, computer algebra systems, constraint solvers, or concept classifiers to deal with it. It is a characteristic feature of these systems that they either have mathematical knowledge implicitly encoded in their critical algorithms or (increasingly) manipulate explicit representations of the relevant mathematical knowledge often in the form of logical formulae. Unfortunately, these systems have differing domains of applications, foundational assumptions, and input languages, which makes them non-interoperable and difficult to compare and evaluate in practice. Moreover, the quantity of mathematical knowledge is growing faster than our ability to formalize and organize it, aggravating the problem that mathematical software systems cannot share knowledge representations.

The work reported in this paper focuses on developing an exchange format between math-knowledge based systems. We concentrate on a foundationally unconstrained framework for knowledge representation that allows to represent the meta-theoretic foundations of the mathematical knowledge in the same format and to interlink the foundations at the meta-logical level. In particular, the logical foundations of domain representations for the mathematical knowledge can be represented as modules themselves and can be interlinked via meta-morphisms. This "logics-as-theories" approach, makes systems behavior as well as their represented knowledge interoperable and thus comparable at multiple levels. Note that the explicit representation of epistemic foundations also benefits systems whose mathematical knowledge is only implicitly embedded into the algorithms. Here, the explicit representation can serve as a documentation of the system as well as a basis for verification or testing attempts.

Of course communication by translation to the lowest common denominator logic — the current state of the art — is always possible. But such translations lose the very structural properties of the knowledge representation that drive computation and led to the choice of logical system in the first place. Therefore our format incorporates a module system geared to support flexible reuse of knowledge items via theory morphisms. This module system is the central part of the proposed format — emphasizing interoperability between theorem proving systems, and the exchange and reusability of mathematical facts across different systems. In contrast to the formula level, which needs to be ontologically unconstrained, the module system level must have a clear semantics (relative to the semantics of the formula level) and be expressive enough to encode current structuring practice so that systems can communicate without losing representational structure.

On a practical level, an exchange format should support interoperability by being easy to parse and standards-compatible. Today, where XML-parsers are available for almost all programming language and higher-level protocols for distributed computation are XML-based, an exchange format should be XML-based and in particular an extension of the MathML [ABC+03] and OpenMath [BCC+04] standards. Moreover, an exchange format must be *web-scalable* in the sense that it supports the distribution of resources (theories, conjectures, proofs, etc.) over the Internet, so that they can be managed collaboratively. In the current web architecture this means that all (relevant) resources must be addressable by a URI-based naming scheme [BLFM05]. Note that in the presence of complex modularity and reusability infrastructure this may mean that resources have to be addressable, even though they are only virtually induced by the inheritance structure.

2 Syntax

2.1 A Four-Level Model of Mathematical Knowledge

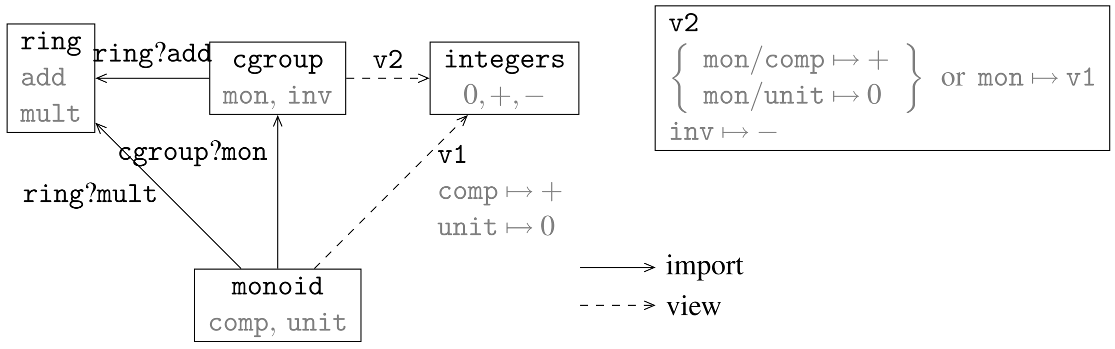

Figure 1: Example

Example 1. We begin the exposition of our language with a simple motivating example that already shows some of the crucial features of MMT and that we will use as a running example: Fig. 1 gives a theory graph for a portion of the algebraic hierarchy. The bottom node represents the theory of monoids, which declares operations for composition and unit. (We omit the axioms and the types here.) The theory `cgroup` for commutative groups arises by importing from `monoid` and adding an operation `inv` for the inverse element. This theory does not have to declare the operations for composition and unit again because they are imported from the theory of monoids. The import is named `mon`, and it can be referenced by the qualified name `cgroup?mon`; it induces a theory morphism from monoids to commutative groups.

Then the theory of rings can be formed by importing from `monoid` (via an import named `mult` that provides the multiplicative structure) and from `cgroup` (via an import named `add` that provides the additive structure). Because all imports are named, different import paths can be distinguished: By concatenating import and symbol names, the theory `ring` can access the symbols `add/mon/comp` (i.e., addition), `add/mon/unit` (i.e, zero), `add/inv` (i.e., additive inverse), `mult/comp` (i.e., multiplication), and `mult/unit` (i.e., one).

The node on the right side of the graph represents a theory for the integers declaring the operations 0, $+$, and $-$. The fact that the integers are a monoid is represented by the view `v1`. It is a theory morphism that is explicitly given by its interpretations of `comp` as $+$ and of `unit` as 0. (If we did not omit axioms, this view would also have to interpret all the axioms of `monoid` as — using Curry-Howard representation — proof terms.)

The view `v2` is particularly interesting because there are two ways to represent the fact that the integers are a commutative group. Firstly, all operations of `cgroup` can be interpreted as terms over `integers`: This means to interpret `inv` as $-$ and the two imported operations `mon/comp` and `mon/unit` as $+$ and 0, respectively. Secondly, `v2` can be constructed along the modular structure of `cgroup` and use the existing view `v1` to interpret all operations imported by `mon`. In MMT, this can be expressed elegantly by the interpretation `mon` $\mapsto$ `v1`, which interprets a named import with a theory morphism. The intuition behind such an interpretation is that it makes the right triangle commute: `v2` is defined such that $\texttt{v2} \circ \mathit{cgroup?mon} = \texttt{v1}$. Clearly, both ways lead to the same theory morphism; the second one is conceptually more complex but eliminates redundancy. (This redundancy is especially harmful when axioms are considered, which must be interpreted as proofs.)

Ontology and Grammar We characterize mathematical theories on four levels: the **document**, **module**, **symbol**, and **object** level. On each level, there are several kinds of expressions. In Fig. 2, the relations between the MMT-concepts of the first three levels are defined in an ontology. The MMT knowledge items are grouped into six primitive concepts. **Documents** (*Doc*, e.g., the whole graph in Fig. 1) comprise the document level. **Theories** (*Thy*, e.g., `monoid` and `integers`) and **views** (*Viw*, e.g., `v1` and `v2`) comprise the module level. And finally **constants** (*Con*, e.g., `comp` and `inv`) and **structures** (*Str*, e.g., `mon` and `add`) as well as **assignments** to them (*ConAss*, e.g., `inv` $\mapsto -$, and *StrAss*, e.g., `mon` $\mapsto$ `v2`) comprise the symbol level.

In addition, we define four unions of concepts: First **modules** (*Mod*) unite theories and views, **symbols** (*Sym*) unite constants and structures, and **assignments** (*Ass*) unite the assignments to constants and structures. The most interesting union is that of **links** (*Lnk*): They unite the module level concept of views and the symbol level concept of structures. Structures being both symbols and links makes our approach balanced between the two different ways to understand them.

These higher three levels are called the structural levels because they represent the structure of mathematical knowledge: Documents are sets of modules, theories are sets of symbols, and links are sets of assignments. The actual mathematical objects are represented at the fourth level: They occur as arguments of symbols and assignments. MMT provides a formalization of the structural levels while being parametric in the specific choice of objects used on the fourth level.

$$\begin{tikzcd}
{} \arrow[r, dashed] & \text{subconcept of} & & Doc \\
{} \arrow[r] & \text{declared in} & & Mod \arrow[u] \\
& Thy \arrow[urr, dashed] & Lnk & & Viw \arrow[ul, dashed] \arrow[ll, dashed] \\
& Sym \arrow[u] & & & Ass \arrow[ull] \\
Con \arrow[ur, dashed] & & Str \arrow[ul, dashed] \arrow[uu, dashed] & ConAss \arrow[ur, dashed] & & StrAss \arrow[ul, dashed]
\end{tikzcd}$$

Figure 2: The MMT Ontology

The declarations of the four levels along with the meta-variables we will use to reference them are given in Fig. 3 and 4. The MMT knowledge items of the structural levels are declared with unqualified names (underlined Latin letter) in a certain scope and referenced by qualified names (Latin letter). Greek letters are used as meta-variables for composed expressions. Following practice in programming languages, we will use _ as an unnamed meta-variable for irrelevant values.

The grammar for MMT is given in Fig. 5 where *, $^+$, $|$, and $[-]$ denote repetition, non-empty repetition, alternative, and optional parts, respectively. The rules for the non-terminals URI and pchar are defined in RFC 3986. Thus, g produces a URI without a query or a fragment. (The query and fragment components of a URI are those starting with the special characters ? and #, respectively.) pchar, essentially, produces any Unicode character, possibly using percent-encoding for reserved characters. (Thus, percent-encoding is necessary for the characters ?/#[]% and all characters generally illegal in URIs.) In this section, we will describe the syntax of MMT and the intuition behind it in a bottom-up manner, i.e., from the object to the document level. Alternatively, the following subsections can be read in top-down order.

Level	Declared concept
Document	document g
Module	theory T, S, R, M
	view v
	structure or view m
Symbol	symbol s
	constant c
	structure h, i, j
	assignment to a constant
	assignment to a structure
Object	variable x

Figure 3: Meta-Variables for References to MMT Declarations

Variable	Type
Λ	library (list of document declarations)
γ	document body (list of module declarations)
ϑ	theory body (list of symbol declarations)
σ	view/structure body (list of assignments to symbols)
ω	term
μ	morphism

Figure 4: Meta-Variables for MMT Expressions

2.1.1 The Object Level

We distinguish two kinds of objects: **terms** and **morphisms**. Atomic terms are references to declared constants, and general terms are built up from the constants via operations such as application and binding as detailed below. Atomic morphisms are references to structures and views, and general morphisms are built up using structures, views, identity, and composition as detailed below.

The MMT objects are typed. For terms, we do not distinguish terms and types syntactically: Rather, typing is a binary relation on terms that is not further specified by MMT. Instead, it is left to be determined by the **foundation**, and MMT is parametric in the specific choice of typing relation. In particular, terms may be untyped or may have multiple types. For morphisms, the domain theory doubles as the type. Thus, every morphism has a unique type.

Well-formedness of objects is checked relative to a home theory. For a term ω, the home theory must declare or import all symbols that occur in ω. For a morphism, the home theory is the codomain. Objects with home theory T are also called **objects over** T. These relations are listed in Fig. 6.

Terms MMT-terms are a generalization of a fragment of OPENMATH objects ([BCC$^+$04]). They can be

- **constants** c declared or imported by the home theory,
- **variables** x declared by a binder,
- **applications** $@(\omega, \omega_1, \ldots, \omega_n)$ of ω to arguments ω_i,

Library	Λ	::=	Doc^*
Document	Doc	::=	$g := \{\gamma\}$
Document body	γ	::=	Mod^*
Module	Mod	::=	$Thy \mid Viw$
Theory	Thy	::=	$\underline{T} \overset{[T]}{:=} \{\vartheta\}$
View	Viw	::=	$\underline{v} : T \to T \overset{[\mu]}{:=} \{\sigma\} \mid \underline{v} : T \to T := \mu$
Theory body	ϑ	::=	Sym^*
Symbol	Sym	::=	$Con \mid Str$
Link body	σ	::=	Ass^*
Assignment	Ass	::=	$ConAss \mid StrAss$
Constant	Con	::=	$\underline{c} : \omega := \omega \mid \underline{c} : \omega \mid \underline{c} := \omega \mid \underline{c}$
Structure	Str	::=	$\underline{i} : T \overset{[\mu]}{:=} \{\sigma\} \mid \underline{i} : T := \mu$
Ass. to constant	$ConAss$	::=	$\underline{c} \mapsto \omega$
Ass. to structure	$StrAss$	::=	$\underline{i} \mapsto \mu$
Term	ω	::=	$\top \mid c \mid x \mid \omega^{\mu} \mid @(\omega, \omega^+)$ $\mid \beta(\omega, \Upsilon, \omega) \mid \alpha(\omega, \omega \mapsto \omega)$
Variable context	Υ	::=	$\cdot \mid \Upsilon, \omega$ if $str(\omega)$ of the form x
Morphism	μ	::=	$id_T \mid i \mid v \mid \mu \bullet \mu$
Document reference	g	::=	URI, no query, no fragment
Module reference	T, v	::=	$g?\underline{T} \mid g?\underline{v}$
Symbol reference	c, i	::=	$T?\underline{c} \mid T?\underline{i}$
Assignment reference	a	::=	$v?\underline{c} \mid v?\underline{i}$
Local name	$\underline{T}, \underline{v}, \underline{c}, \underline{i}$	::=	$C^+[/C^+]^*$
Variable name	x	::=	C^+
Character	C	::=	pchar
	URI, pchar		see RFC 3986 [BLFM05]

Figure 5: The Grammar for Raw MMT Expressions

	Atomic object	Composed object	Type	Checked relative to
Terms	constant	term	term	home theory
Morphisms	structure/view	morphism	domain	codomain

Figure 6: The Object Level

- **bindings** $\beta(\omega_1, \Upsilon, \omega_2)$ by a binder ω_1 of a list of variables Υ with scope ω_2,
- **attributions** $\alpha(\omega_1, \omega_2 \mapsto \omega_3)$ to a term ω_1 with key ω_2 and value ω_3,
- **morphism applications** ω^{μ} of μ to ω,
- **special term** $\top$.

A term over T, may use the constant $T?\underline{c}$ to refer to a previously declared symbol $\underline{c}$. And if $\underline{i}$ is a previously declared structure instantiating S, and $\underline{c}$ is a constant declared in S, then T may use $T?\underline{i}/\underline{c}$ to refer to the imported constant. By concatenating structure names, any indirectly imported constant has a unique qualified name.

The attributions of OpenMath mean that every term can carry a list of key-value pairs that are themselves OpenMath objects. In particular, attributions are used to attach types to bound variables. In OpenMath, the keys must be symbols, which we relax to obtain a more uniform syntax. Because OpenMath specifies that nested attributions are equivalent to a single attribution, we can introduce attributions of multiple key-value pairs as abbreviations:

$$\alpha(\omega, \omega_1 \mapsto \omega'_1, \ldots, \omega_n \mapsto \omega'_n) := \alpha(\ldots(\alpha(\omega, \omega_1 \mapsto \omega'_1), \ldots), \omega_n \mapsto \omega'_n)$$

We use the auxiliary function $str(\cdot)$ to strip toplevel attributions from terms, i.e.,

$$str(\alpha(\omega, \_)) = str(\omega) \qquad str(\omega) = \omega \text{ otherwise.}$$

This is used in the grammar to make sure that only attributed variables and not arbitrary terms may occurs in the context bound in a binding.

The special term $\top$ is used for terms that are inaccessible because they refer to constants that have been hidden by a structure or view. $\top$ is also used to subsume hidings under assignments (see below).

Example 2 (Continued). The running example only contains the atomic terms given by symbols. Composed terms arise when types and axioms are covered. For example, the type of the inverse in a commutative group is $@(\rightarrow, \iota, \iota)$. Here $\rightarrow$ represents the function type constructor and ι the carrier set. These two constants are not declared in the example. Instead, we will add them later by giving `cgroup` a meta-theory, in which these symbols are declared. A more complicated term is the axiom for left-neutrality of the unit:

$$\omega_e := \beta(\forall, \alpha(x, \texttt{oftype} \mapsto \iota), @(=, @(e?\texttt{monoid?comp}, e?\texttt{monoid?unit}, x), x)).$$

Here $\forall$ and $=$ are further constants that must be declared in the so far omitted meta-theory. The same applies to `oftype`, which is used to attribute the type ι to the bound variable x. We assume that the example is located in a document with URI e. Thus, for example, $e?$`monoid?comp` is used to refer to the constant `comp` in the theory `monoid`.

Morphisms Morphisms are built up by compositions $\mu_1 \bullet \mu_2$ of structures i, views m, and the identity id_T of T. Here μ_1 is applied before μ_2, i.e., $\bullet$ is composition in diagram order. Morphisms are not used in OpenMath, which only has an informal notion of theories, namely the content dictionaries, and thus no need to talk about theory morphisms. A morphism application ω^{μ} takes a term ω over S and a morphism μ from S to T, and returns a term over T. Similarly, a morphism over S, e.g., a morphism μ' from R to S becomes a morphism over T by taking the composition $\mu' \bullet \mu$. The normalization given below will permit to eliminate all morphism applications.

Example 3 (Continued). In the running example, an example morphism is

$$\mu_e := e?\texttt{cgroup?mon} \bullet e?\texttt{v2}.$$

It has domain $e?$`monoid` and codomain $e?$`integers`. The intended semantics of the term ${\omega_e}^{\mu_e}$ is that it yields the result of applying μ_e to ω_e, i.e.,

$$\beta(\forall, \alpha(x, \texttt{oftype} \mapsto \iota), @(=, @(+, 0, x), x)).$$

Here, we assume μ_e has no effect on those constants that are inherited from the meta-theory. We will make that more precise below.

2.1.2 The Symbol Level

We distinguish symbol declarations and assignments to symbols. **Declarations** are the constituents of theories: They introduce named objects, i.e., the **constants** and **structures**. Similarly, **assignments** are the constituents of links: A link from S to T can be defined by a sequence of assignments that instantiate constants or structures declared in S with terms or morphisms, respectively, over T. This yields four kinds of knowledge items which are listed in Fig. 7. Both constant and structure declarations are further subdivided as explained below.

	Declaration	Assignment
Terms	of a constant Con	to a constant $\underline{c} \mapsto \omega$
Morphisms	of a structure Str	to a structure $\underline{i} \mapsto \mu$

Figure 7: The Statement Level

Declarations There are two kinds of symbols:

- **Constant declarations** $\underline{c} : \tau := \delta$ declare a constant $\underline{c}$ of type τ with definition δ. Both the type and the definition are optional yielding four kinds of constant declarations. If both are given, then δ must have type τ. In order to unify these four kinds, we will sometimes write $\perp$ for an omitted type or definition.

- **Structure declarations** $\underline{i} : S \overset{[\mu]}{:=} \{\sigma\}$ declare a structure $\underline{i}$ from the theory S defined by assignments σ. Such structures can have an optional meta-morphism μ (see below). Alternatively, structures may be introduced using an existing morphism: $\underline{i} : S := \mu$, which simply means that $\underline{i}$ serves as an abbreviation for μ; we call these structures **defined structures**. While the domain of a structure is always given explicitly (in the style of a type), the codomain is always the theory in which the structure is declared. Consequently, if $\underline{i} : S := \mu$ is declared in T, μ must be a morphism from S to T.

In well-formed theory bodies, the declared or imported names must be unique.

Assignments Parallel to the declarations, there are two kinds of assignments that can be used to define a link m:

- **Assignments to constants** of the form $\underline{c} \mapsto \omega$ express that m maps the constant $\underline{c}$ of S to the term ω over T. Assignments of the form $\underline{c} \mapsto \top$ express that the constant $\underline{c}$ is **hidden**, i.e., m is undefined for $\underline{c}$.

- **Assignments to structures** of the form $\underline{i} \mapsto \mu$ for a structure $\underline{i}$ declared in S and a morphism μ over (i.e., into) T express that m maps the structure $\underline{i}$ of S to μ. This results in the commuting triangle $S?\underline{i} \bullet m = \mu$.

Both kinds of assignments must type-check. For a link m with domain S and codomain T defined by among others an assignment $\underline{c} \mapsto \omega$, the term ω must type-check against τ^m where τ is the type of $\underline{c}$ declared in S. This ensures that typing is preserved along links. For an assignment $\underline{i} \mapsto \mu$ where $\underline{i}$ is a structure over S of type R, type-checking means that μ must be a morphism from R to T.

Virtual Symbols Intuitively, the semantics of a structure $\underline{i}$ with domain S declared in T is that all symbols of S are imported into T. For example, if S contains a symbol $\underline{s}$, then $\underline{i}/\underline{s}$ is available as a symbol in T. In other words, the slash is used as the operator that dereferences structures. Another way to say it is that structures create virtual or induced symbols. This is significant because these virtual symbols and their properties must be inferred, and this is non-trivial because it is subject to the translations given by the structure. While these induced symbols are easily known to systems built for a particular formalism, they present great difficulties for generic knowledge management services.

Similarly, every assignment to a structure induces virtual assignments to constants. Continuing the above example, if a link with domain T contains an assignment to $\underline{i}$, this induces assignments to the imported symbols $\underline{i}/\underline{s}$. Furthermore, assignments may be **deep** in the following sense: If $\underline{c}$ is a constant of S, a link with domain T may also contain assignments to the virtual constant $\underline{i}/\underline{c}$. Of course, this could lead to clashes if a link contains assignments for both $\underline{i}$ and $\underline{i}/\underline{c}$; links with such clashes are not well-formed.

Example 4 (Continued). The symbol declarations in the theory `cgroup` are written formally like this:

$$\texttt{inv} : @(\rightarrow, \iota, \iota) \quad \text{and} \quad \texttt{mon} : e?\texttt{monoid} := \{\}.$$

The latter induces the virtual symbols $e?\texttt{cgroup}?\texttt{mon}/\texttt{comp}$ and $e?\texttt{cgroup}?\texttt{mon}/\texttt{unit}$.

Using an assignment to a structure, the assignments of the view `v2` look like this:

$$\texttt{inv} \mapsto e?\texttt{integers}?- \quad \text{and} \quad \texttt{mon} \mapsto e?\texttt{v1}.$$

The latter induces virtual assignments for the symbols $e?\texttt{cgroup}?\texttt{mon}/\texttt{comp}$ and $e?\texttt{cgroup}?\texttt{mon}/\texttt{unit}$. For example, $e?\texttt{cgroup}?\texttt{mon}/\texttt{comp}$ is mapped to $e?\texttt{monoid}?\texttt{comp}^{e?\texttt{v1}}$.

The alternative formulation of the view `v2` arises if two deep assignments to the virtual constants are used instead of the assignment to the structure `mon`:

$$\texttt{mon}/\texttt{comp} \mapsto e?\texttt{integers}?+ \quad \text{and} \quad \texttt{mon}/\texttt{unit} \mapsto e?\texttt{integers}?0$$

2.1.3 The Module Level

The module level consists of two kinds of declarations: theory and view declarations.

- **Theory declarations** $\underline{T} \overset{[M]}{:=} \{\vartheta\}$ declare a theory $\underline{T}$ defined by a list of symbol declarations ϑ, which we call the body of $\underline{T}$. Theories have an optional meta-theory M.
- **View declarations** $\underline{v} : S \rightarrow T \overset{[\mu]}{:=} \{\sigma\}$ declare a link $\underline{v}$ from S to T defined by a list of assignments σ. If S has a meta-theory M, a meta-morphism μ from M to T must be provided. Just like structures, views may also be defined by an existing morphism: $\underline{v} : S \rightarrow T := \mu$.

Meta-Theories Above, we have already mentioned that theories may have meta-theories and that links may have meta-morphisms. Meta-theories provide a second dimension in the graph induced by theories and links. If M is the meta-theory of T, then there is a special structure instantiating M in T, which we denote by $T?..$. M provides the syntactic material that T can use to define the semantics of its symbols: T can refer to a symbol $\underline{s}$ of M by $T?../\underline{s}$. While meta-theories could in principle be replaced with structures altogether, it is advantageous to make them explicit because the conceptual distinction pervades mathematical discourse. For example, systems can use the meta-theory to determine whether they understand a specific theory they are provided as input.

Because theories S with meta-theory M implicitly import all symbols of M, a link from S to T must provide assignments for these symbols as well. This is the role of the meta-morphism: Every link from S to T must provide a meta-morphism, which must be a morphism from M to T. Defined structures or views with definition μ do not need a meta-morphism because a meta-morphism is already implied by the meta-morphisms of the links occurring in μ.

Example 5 (Continued). An MMT theory for the logical framework LF could be declared like this

$$\texttt{lf} := \{\texttt{type}, \texttt{funtype}, \ldots\}$$

where we only list the constants that are relevant for our running example. If this theory is located in a document with URI m, we can declare a theory for first-order logic in a document with URI f like this:

$$\texttt{fol} \overset{m?\texttt{lf}}{:=} \{\texttt{i} : ??../\texttt{type}, \texttt{o} : ??../\texttt{type}, \texttt{equal} : @(??../\texttt{funtype}, ??\texttt{i}, ??\texttt{i}, ??\texttt{o}), \ldots\}$$

Here we already use relative names (see Sect. 2.3.2) in order to keep the notation readable: Every name of the form $??\underline{s}$ is relative to the enclosing theory: For example, $??\texttt{i}$ resolves to $f?\texttt{fol}?\texttt{i}$. Furthermore, we use the special structure name .. to refer to those constants inherited from the meta-theory. Again we restrict ourselves to a few constant declarations: types `i` and `o` for terms and formulas and the equality operation that takes two terms and returns a formula.

Then the theories `monoid`, `cgroup`, and `ring` can be declared using $f?\texttt{fol}$ as their meta-theory. For example, the declaration of the theory `cgroup` finally looks like this:

$$\texttt{cgroup} \overset{f?\texttt{fol}}{:=} \left\{\texttt{inv} : @(??../../\texttt{funtype}, ??\texttt{i}, ??\texttt{i}),\ \texttt{inv} : e?\texttt{monoid} \overset{e?\text{cgroup}?..}{:=} \{\}\right\}$$

Here $../../\texttt{funtype}$ refers to the function type constant declared in the meta-theory of the meta-theory. And the structure `mon` must have a meta-morphism with domain $f?\texttt{fol}$ and codomain $e?\texttt{cgroup}$. This is trivial because the meta-theory of $e?\texttt{cgroup}$ is also $f?\texttt{fol}$: The meta-morphism is simply the implicit structure $e?\texttt{cgroup}?..$ via which $e?\texttt{cgroup}$ inherits from $f?\texttt{fol}$.

A more complicated meta-morphism must be given in the view `v1` if we assume that the meta-theory of `integers` is some other theory, i.e., a representation of set theory.

Structures and Views Both structures and views from S to T are defined by a list of assignments σ that assigns T-objects to the symbols declared in S. And both induce theory morphisms from S to T that permit to map all objects over S to objects over T. The major difference between structures and views is that a view only relates two fixed theories without changing either one. On the other hand, structures from S to T occur within T because they change the theory T. Structures have **definitional flavor**, i.e., the symbols of S are imported into T. In particular, if σ contains no assignment for a constant $\underline{c}$, this is equivalent to copying (and translating) the declaration of $\underline{c}$ from S to T. If σ does provide an assignment $\underline{c} \mapsto \omega$, the declaration is also copied, but in addition the imported constant receives ω as its definiens.

Views, on the other hand, have **theorem flavor**: σ *must* provide assignments for the symbols of S. If a constant $\underline{c} : \tau$ represents an axiom stating τ, giving an assignment $\underline{c} \mapsto \pi$ means that π is a proof of the translation of τ.

Therefore, the assignments defining a structure may be (and typically are) partial whereas a view should be total. This leads to a crucial technical difficulty in the treatment of structures: Contrary to views from S to T, the assignments by themselves in a structure from S to T do not induce a theory morphism from S to T — only by declaring the structure do the virtual symbols become available in T that serve as the images of (some of) the symbols of S. This is unsatisfactory because it makes it harder to unify the concepts of structures and views.

Therefore, we admit **partial views** as well. As it turns out, this is not only possible, but indeed desirable. A typical scenario when working with views is that some of the specific assignments making up the view constitute proof obligations and must be found by costly procedures. Therefore, it is reasonable to represent partial views, namely views where some proof obligations have already been discharged whereas others remain open. Thus, we use hiding to obtain a semantics for partial views: All constants for which a view does not provide an assignment are implicitly hidden, i.e., $\top$ is assigned to them.

If a link m from S to T is applied to an S-constant that is hidden, there are two cases: If the hidden symbol has a definition in S, it is replaced by this definition before applying the link. If it does not have a definition, it is mapped to $\top$. Hiding is strict: If a subterm is mapped to $\top$, then so is the whole term. In that case, we speak of hidden terms.

2.1.4 The Document Level

Document declarations are of the form $g := \{\gamma\}$ where γ is a document body and g is a URI identifying the document. The meaning of a document declaration is that γ is accessible via the name g. Since g is a URI, it is not necessarily only the name, but can also be the primary location of γ. By forming lists of documents, we obtain **libraries**, which represent mathematical knowledge bases. Special cases of libraries are single self-contained documents and the internet seen as a library of MMT documents.

Documents provide the border between formal and informal treatment: How documents are stored, copied, cached, mirrored, transferred, and combined is subject to knowledge management services that may or may not take the mathematical semantics of the document bodies in the processed documents into account. For example, libraries may be implemented as web servers, file systems, databases, or any combination of these. The only important thing is that they provide the query interface described below.

Theory graphs are the central notion of MMT. The theory graph is a directed acyclic multigraph. The nodes are all theories of all documents in the library. And similarly, the edges are the structures and views of all documents. Then theory morphisms can be recovered as the paths in the theory graph.

2.2 Querying a Library

MMT is designed to scale to a mathematical web. This means that we define access functions to MMT libraries that have the form of HTTP requests to a RESTful web server [Fie00]. Specifically, there is a lookup function that takes a library Λ and a URI U as arguments and returns an MMT fragment $\Lambda(U)$. This specifies the behavior of a web server hosting an MMT library in response to GET requests. Furthermore, all possible changes to an MMT library can be formulated as POST, PUT, and DELETE requests that add, change, and delete knowledge items, respectively.

It is non-trivial to add such a RESTful interface to formal systems a posteriori. It requires the rigorous use of URIs as identifiers for all knowledge items that can be the object of a change. And it requires to degrade gracefully if the documents in a library are not in the same main memory or on the same machine. In large applications, it is even desirable to load only the relevant parts of a document into the main memory and request further fragments on demand from a low-level database service. In MMT, web-scalability is built into the core: All operations on a library Λ including the definition of the semantics only depend on the lookup function $\Lambda(-)$ and not on Λ itself. In particular, two libraries that respond in the same way to lookup requests are indistinguishable by design. Therefore, MMT scales well to web-based scenarios.

Here we will only give a brief overview over the lookup function $\Lambda(-)$. It is a partial function from URIs to MMT fragments. For example, assume Λ to contain a theory S containing the symbol declarations $\underline{c} : \tau$, $\underline{c'} : \tau'$, and $\underline{h} : R := \{\}$; furthermore, assume a theory $T = g?\underline{T}$ declaring a structure $\underline{i} : S := \{\underline{c} \mapsto \delta, \underline{h} \mapsto \mu\}$.

Then the lookups of $S?\underline{c}$ and $S?\underline{h}$ yield $\underline{c} : \tau$ and $\underline{h} : R := \{\}$, respectively. These are lookups of explicit symbol declarations — an important feature of MMT is that all induced, symbols can be looked up in the same way. For example, $\Lambda(T?\underline{i}/\underline{c})$ yields $\underline{i}/\underline{c} : \tau^{T?\underline{i}} := \delta$. Here $\underline{i}/\underline{c}$ is the unique name of the constant $\underline{c}$ induced by the structure $\underline{i}$; $\tau^{T?\underline{i}}$ is the translation of the type of $S?\underline{c}$ along the structure $T?\underline{i}$; and the assignment $\underline{c} \mapsto \delta$ causes δ to occur as the definiens of the constant $\underline{i}/\underline{c}$. Similarly, the lookup of the induced structure $T?\underline{i}/\underline{h}$ yields $\underline{i}/\underline{h} : R := \mu$. Here the assignment $\underline{i} \mapsto \mu$ causes $\underline{i}/\underline{h}$ to be defined by a morphism.

To query and edit documents, assignments are also accessible by URIs even though they are not referenced by any rule of the syntax. An assignment $\underline{c} \mapsto \omega$ in a view v is referenced by the URI $v?\underline{c}$ and similarly for assignments to structures. To access assignments in structures, we distinguish the URIs $g?\underline{T}?\underline{i}/\underline{c}$ and $g?\underline{T}/\underline{i}?\underline{c}$: The former identifies the constant $\underline{c}$ that is induced by $\underline{i}$ as defined above; the latter identifies the assignment $\underline{c} \mapsto \delta$ to the symbol $\underline{c}$ in the structure $T?\underline{i}$. In particular, the lookup of the former always is always defined, while the lookup of the latter is undefined if no assignment is present.

Just like symbols induced by structures have URIs, so have assignments induced by assignments to structures. For example, if additionally R declares a constant $\underline{d} : \rho := \perp$, then the lookup of $g?\underline{T}/\underline{i}?\underline{h}/\underline{d}$ yields $\underline{h}/\underline{d} \mapsto (R?\underline{d})^{\mu}$. This means that the assignment $\underline{h} \mapsto \mu$ in $T?\underline{i}$ induces an assignment to $\underline{h}/\underline{d}$. The lookup of assignments to induced structures is defined accordingly.

The above lookup functions are actually modified to accommodate for hiding. If the lookup of a constant would yield $\underline{c} : \tau := \delta$ according to the above definition, but δ is (or simplifies to) $\top$, then the lookup is actually undefined.

Example 6 (Continued). If Λ is a library containing the three documents with URIs m, f, and e from the running example, we obtain the following lookup results:

- $\Lambda(e?\texttt{monoid}) = (f?\texttt{fol},\ \vartheta)$ where ϑ contains the declarations for `comp` and `unit`,
- $\Lambda(e?\texttt{cgroup/mon}) = (e?\texttt{monoid},\ e?\texttt{cgroup},\ e?\texttt{cgroup}?..,\ \cdot)$,
  i.e., $e?\texttt{cgroup/mon}$ is a morphism from $e?\texttt{monoid}$ to $e?\texttt{cgroup}$ with meta-morphism $e?\texttt{cgroup}?..$, and without any assignments,
- $\Lambda^{e?\texttt{monoid}}(\texttt{unit}) = (e?\texttt{monoid}?../\texttt{i},\ \perp)$,
  i.e., the theory `monoid` has a constant `unit` with type $e?\texttt{monoid}?../\texttt{i}$ and no definition,
- $\Lambda^{e?\texttt{cgroup}}(\texttt{mon/unit}) = (e?\texttt{monoid}?../\texttt{i}^{e?\texttt{cgroup}?\texttt{mon}},\ \perp)$,
  i.e., the type of the virtual constant `mon/unit` arises by translating the type from the source theory along the importing structure,
- $\Lambda^{e?\texttt{cgroup/mon}}(\texttt{unit}) = \perp$,
  i.e., the lookup is undefined because the structure $e?\texttt{cgroup/mon}$ does not have an assignment for `unit`,
- the lookup $\Lambda^{e?\texttt{v2}}(\texttt{mon/unit})$ yields $e?\texttt{integers}?0$ if the variant with the deep assignment $\texttt{mon/unit} \mapsto e?\texttt{integers}?0$ is used to define `v2`, and $e?\texttt{monoid}?\texttt{unit}^{e?\texttt{v1}}$ if the variant with the structure assignment $\texttt{mon} \mapsto e?\texttt{v1}$ is used.

2.3 Concrete Syntax

2.3.1 XML-Encoding

The XML grammar mostly follows the abstract grammar. Documents are `omdoc` elements with a theory graph, i.e., `theory` and `view` elements as children. The children of theories are `constant` and `structure`. And the children of views and structures are `maps` (assignment to a constant or structure).

Both terms and morphisms are represented by OPENMATH elements. And all names of theories are URIs. The grammar does not account for the well-formedness of objects and names. In particular, well-formedness is not checked at this level.

Formally, we define an encoding function $E(-)$ that maps MMT-expressions to sequences of XML elements. The precise definition of $E(-)$ is given in Fig. 9 and 10 where we assume that the following namespace bindings are in effect:

```
xmlns="http: // www.omdoc.org/ns/omdoc"
xmlns:om="http: // www.openmath.org/OpenMath"
```

And we assume the following content dictionary with cdbase `http://cds.omdoc.org/omdoc/mmt.omdoc`, which is itself given as an OMDOC theory:

```
<omdoc>
  <theory name="mmt">
    <constant name="hidden"/>
    <constant name="identity"/>
    <constant name="composition"/>
  </theory>
</omdoc>
```

We abbreviate the `OMS` elements referring to these symbols by *OMDoc*(`identity`) etc.

The encoding of the structural levels is straightforward. The encoding of objects is a generalization of the XML encoding of OPENMATH objects. We use the `OMS` element of OPENMATH to refer to symbols, theories, views, and structures. The symbol with name *OMDoc*(`hidden`) is used to encode the special term $\top$. To encode morphisms, *OMDoc*(`identity`) and *OMDoc*(`composition`) are used. *OMDoc*(`identity`) takes a theory as an argument and returns a morphism. *OMDoc*(`composition`) takes a list of structures and views as arguments and returns their (left-associative) diagram order composition. Morphism application is encoded by reusing the `OMA` element from OPENMATH.

The encoding of names is giving separately in Fig. 8 on the right. There are two different ways to encode names. In `OMS` elements, triples $(g, \underline{m}, \underline{s})$ of document, module, and symbol name are used, albeit with more fitting attribute names. These triples correspond to the $(cdbase, cd, name)$ triples of the OPENMATH standard ([BCC+04]). We also use them to refer to module level names by omitting the *name* attribute. When names occur in attribute values, their URIs are used.

`OMS`-triple	$E(g?\underline{m}?\underline{s})$	base="g" module="$\underline{m}$" name="$\underline{s}$"
	$E(g?\underline{m})$	base="g" module="$\underline{m}$"
URI	$E(g?\underline{m}?\underline{s})$	$g?\underline{m}?\underline{s}$
	$E(g?\underline{m})$	$g?\underline{m}$

Figure 8: XML Encoding of Names

2.3.2 Relative Names

In practice it is very inconvenient to always give qualified names. Therefore, we define relative references as a relaxation of the syntax that is elaborated into the official syntax.

A **relative reference** consists of three optional components: a document reference g, a module reference m and a symbol reference s. We write relative references as triples (g, m, s) where we write $\perp$ if a component is omitted. g must be a URI reference as defined in RFC 3986 ([BLFM05]) but without query or fragment. m and s must be unqualified names, i.e., slash-separated non-empty sequences of names. Furthermore, m ans s may optionally start with a slash, which is used to distinguish absolute module and symbol references from relative ones.

An **absolute reference**, which serves as the base of the resolution, is an MMT-name G, $G?\underline{M}$, or $G?\underline{M}?\underline{S}$. Then the resolution of relative references is a partial function that takes a relative reference

Library	$E(Doc_1,\ldots,Doc_n)$	$E(Doc_1)\ \ldots\ E(Doc_n)$
Document	$E(g := \{Mod_1,\ldots,Mod_n\})$	<**omdoc**>$E(Mod_1)\ldots E(Mod_n)$</**omdoc**>
Theory	$E(\underline{T} \overset{[M]}{:=} \{Sym_1,\ldots,Sym_n\})$	<**theory** name="$\underline{T}$" [metatheory="$E(M)$"]> $E(Sym_1)\ldots E(Sym_n)$ </**theory**>
View	$E(\underline{v} : S \to T \overset{[\mu]}{:=} \{\sigma\})$	<**view** name="$\underline{v}$" from="$E(S)$" to="$E(T)$"> [<**metamorphism**>$E(\mu)$</**metamorphism**>] $E(\sigma)$ </**view**>
	$E(\underline{i} : S \to T := \mu)$	<**view** name="$\underline{i}$" from="$E(S)$" to="$E(T)$"> <**definition**> <**OMOBJ**>$E(\mu)$</**OMOBJ**> </**definition**> </**view**>

Figure 9: XML Encoding of Document and Module Level

$R = (g,m,s)$ and an absolute reference B as input and returns an MMT-name $resolve(B,R)$ as output. It is defined as follows:

- If $g \neq \perp$, then possible starting slashes of m and s are ignored and
  - if $R = (g,m,s)$: $resolve(B,R) = (G+g)?m?s$,
  - if $R = (g,m,\perp)$: $resolve(B,R) = (G+g)?m$,
  - if $R = (g,\perp,\perp)$: $resolve(B,R) = G+g$,

  where $G+g$ denotes the resolution of the URI reference g relative to the URI G as defined in RFC 3986 ([BLFM05]).

- If $g = \perp$ and $m \neq \perp$, then a possible starting slash of s is ignored and
  - if $R = (\perp,m,s)$: $resolve(B,R) = G?M+m?s$,
  - if $R = (\perp,m,\perp)$: $resolve(B,R) = G?M+m$,

  where $M+m$ resolves m relative to M: If M is not defined or if m starts with a slash, $M+m$ is m with a possible starting slash removed; otherwise, it is M/m.

- If $g = m = \perp$ and M is defined, then $resolve(B,R) = G?M?S+s$, where $S+s$ is defined like $M+m$ above.

- $resolve(B,R)$ is undefined otherwise.

Relative references can also be encoded as URIs: The triple (g,m,s) is encoded as $g?m?s$. If components are omitted, they are encoded as the empty string. Trailing (but not leading) ? characters can be dropped. For example,

- $(g,m,\perp)$ is encoded as $g?m$,
- $(\perp,/m,s)$ is encoded as $?/m?s$,
- $(\perp,\perp,s)$ is encoded as $??s$,

This encoding can be parsed back uniquely by splitting a URI into up to three components around the separator ?.

Constant	$E(\underline{c} : [\tau] := [\delta])$	<**constant** name="$\underline{c}$"> [<**type**> <**OMOBJ**>$E(\tau)$</**OMOBJ**> </**type**>] [<**definition**> <**OMOBJ**>$E(\delta)$</**OMOBJ**> </ **definition** >] </**constant**>
Structure	$E(\underline{i} : S \overset{[\mu]}{:=} \{\sigma\})$	<**structure** name="$\underline{i}$" from="$E(S)$"> [<**metamorphism**>$E(\mu)$</**metamorphism**>] $E(\sigma)$ </**structure**>
	$E(\underline{i} : S := \mu)$	<**structure** name="$\underline{i}$" from="$E(S)$"> <**definition**> <**OMOBJ**>$E(\mu)$</**OMOBJ**> </ **definition** > </**structure**>
Assignments	$E(Ass_1, \dots, Ass_n)$	$E(Ass_1) \dots E(Ass_n)$
	$E(\underline{c} \mapsto \omega)$	<**conass** name="$E(\underline{c})$"> <**OMOBJ**>$E(\omega)$</**OMOBJ**> </**conass**>
	$E(\underline{i} \mapsto \mu)$	<**strass** name="$E(\underline{i})$"> <**OMOBJ**>$E(\mu)$</**OMOBJ**> </**strass**>
Term	$E(c)$	<**om:OMS** $E(c)$/>
	$E(x)$	<**om:OMV** name="x"/>
	$E(\top)$	*OMDoc*(`hidden`)
	$E(\omega^{\mu})$	<**om:OMA**>$E(\mu)$ $E(\omega)$</**om:OMA**>
	$E(@(\omega_1, \dots, \omega_n))$	<**om:OMA**>$E(\omega_1)$ … $E(\omega_n)$</**om:OMA**>
	$E(\beta(\omega_1, x_1, \dots, x_n, \omega_2))$	<**om:OMBIND**> $E(\omega_1)$ <**om:OMBVAR**> $E(x_1)$ … $E(x_n)$ </**om:OMBVAR**> $E(\omega_2)$ </**om:OMBIND**>
	$E(\alpha(\omega_1, \omega_2 \mapsto \omega_3))$	<**om:OMATTR**> <**om:OMATP**>$E(\omega_2)$ $E(\omega_3)$</**om:OMATP**> $E(\omega_1)$ </**om:OMATTR**>
Morphism	$E(id_T)$	<**om:OMA**> *OMDoc*(`identity`) <**om:OMS** $E(T)$/> </**om:OMA**>
	$E(\mu_1 \bullet \dots \bullet \mu_n)$	<**om:OMA**> *OMDoc*(`composition`) $E(\mu_1) \dots E(\mu_n)$ </**om:OMA**>
	$E(i)$	<**om:OMS** $E(i)$/>
	$E(v)$	<**om:OMS** $E(v)$/>

Figure 10: XML Encoding of Symbol and Object Level

3 Advanced Concepts

In this section, we give an overview over the most important advanced concepts related to MMT. Full definitions are given in [Rab08].

Well-formed Expressions The well-formedness of expressions is defined by an inference system about MMT fragments. This system guarantees in particular the uniqueness of all names. The most difficult of the well-formedness definition is to deal with assignments to structures: In general, an assignment $\underline{i} \mapsto \mu$ to a structure with domain S may cause inconsistencies if a constant $S?\underline{c}$ has a definition in S that differs from $(S?\underline{c})^{\mu}$. The definition of well-formedness makes sure that such inconsistencies are prevented. And it does so in a way that can be checked efficiently because the modular structure is exploited throughout the well-formedness checking.

MMT does not provide a definition of well-*typed* terms or of logical consequence. Rather, the MMT inference system is parametric in the judgments for equality and typing of terms. The definitions of these judgments must be provided externally. In our MMT implementation, these judgments are defined by plugins where the meta-theory of the home theory of a term ω determines which plugin is used to check the well-typedness of ω.

Semantics The representation of theory graphs is geared towards expressing mathematical knowledge with the least redundancy by using inheritance between theories. This style of writing mathematics has been cultivated by the Bourbaki group ([Bou74]) and lends itself well to a systematic development of theories. However, it is often necessary to eliminate the modular structure, for example when interfacing with a system that cannot understand it or to elaborate modular theories into a non-modular trusted core.

Therefore, we define a so-called **flattening** operation that eliminates all structures, meta-theories, and morphisms, and reduces theories to collections of constants, possibly with types and definitions, which conforms to the non-modular logical view of theories. For a given MMT-library Λ, we can view the flattening of Λ as its semantics, since flattening eliminates all specific MMT-representation infrastructure. Essentially, the flattening replaces all structures with the induced symbols and all assignments to structures with the induced assignments as described in Sect. 2.2. The crucial invariant of the flattening is that if a library is well-formed, then so is its flattening; and furthermore, a library and its flattening are indistinguishable by the lookup function $\Lambda(-)$. This guarantees that flattening can be performed transparently. In particular, the above-mentioned external definitions of typing and equality are only needed for the non-modular case.

Applications There are two main use cases for MMT: the use of MMT as a simple interface language between (i) two logical systems as well as between (ii) a logical system and a knowledge management service.

The first use case uses MMT as an interlingua for system interoperability. This works well because the choice of primitives in MMT is the result of a careful trade-off between simplicity and expressivity. Thus, MMT provides a simple abstraction over the most important language constructs that are common to systems maintaining logical libraries, while deliberately avoiding other features that would have increased the burden on systems reading MMT. Here the logics-as-theories approach provides an extremely helpful way to document the semantics of the logic-specific concepts and their counterparts in other logics; furthermore, systems can use the meta-theory relation to detect which theories they can interpret and how to interpret them.

By knowledge management services, we mean services that can be applied to logical knowledge but are not primarily logical in nature. Such services include user interaction, concurrent versioning, management of change, and search. Implementing such services often requires intricate details about the underlying system, thus severely limiting the set of potential developers. But in fact these services can often be developed independently of the logical systems and with relatively little logical expertise. Here MMT comes in as an interface language providing knowledge management services with exactly the information they need while hiding all logic- and system-specific details. Therefore, for example, the concrete syntax of MMT fully marks up the term structure, while the well-formedness definition of MMT does not take type-checking into account.

4 Conclusion

We have presented an exchange format for mathematical knowledge that supports system interoperability by providing an infrastructure for efficiently re-using representations of mathematical knowledge and for formalizing foundational assumptions and structures of the underlying logical systems themselves in a joint web-scalable representation infrastructure.

We consider it a major achievement of the work reported here that the MMT format integrates theoretical aspects like the syntax/semantics interface of the module systems or meta-logical aspects with practical considerations like achieving web-scalability via a URI-based referencing scheme that is well-integrated with the module system.

The proposed MMT format takes great care to find a minimal set of primitives to keep the representational core of the language small and therefore manageable but expressive enough to cover the semantic essentials of current (semi)-automated theorem provers, model checkers, computer algebra systems, constraint solvers, concept classifiers and mathematical knowledge management systems.

We are currently evaluating the MMT format and in particular are developing translators from the Isabelle [Pau94] and CASL [CoF04] formats into MMT. Both representation formats have strong module systems and well-specified underlying semantics, which make them prime evaluation targets. We are also integrating the MMT infrastructure into the upcoming version 2 of the OMDOC format [Koh06].

References

[ABC+03] Ron Ausbrooks, Stephen Buswell, David Carlisle, Stéphane Dalmas, Stan Devitt, Angel Diaz, Max Froumentin, Roger Hunter, Patrick Ion, Michael Kohlhase, Robert Miner, Nico Poppelier, Bruce Smith, Neil Soiffer, Robert Sutor, and Stephen Watt. Mathematical Markup Language (MathML) version 2.0 (second edition). W3C recommendation, World Wide Web Consortium, 2003.

[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`.

[BLFM05] Tim Berners-Lee, Roy. Fielding, and L. Masinter. Uniform resource identifier (URI): Generic syntax. RFC 3986, Internt Engineering Task Force, 2005.

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

[CoF04] CoFI (The Common Framework Initiative). CASL *Reference Manual*. LNCS 2960 (IFIP Series). Springer, 2004.

[Fie00] R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[Koh06] Michael Kohlhase. OMDOC – *An open markup format for mathematical documents [Version 1.2]*. Number 4180 in LNAI. Springer Verlag, 2006.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. LNCS. Springer Verlag, 1994.

[Rab08] F. Rabe. *Representing Logics and Logic Translations*. PhD thesis, Jacobs University Bremen, 2008.