

Integrating Content and Narration-Oriented Document Formats

Deyan Ginev, Mihnea Iancu, and Florian Rabe

Jacobs University, Bremen, Germany

Abstract. Narrative, presentation-oriented assistant systems for mathematics on the one hand and formal, content-oriented ones on the other hand have so far been developed and used largely independently. The former – such as \LaTeX – excel at type-setting mathematical documents whereas the latter – such as proof assistants – excel at verifying the correctness of mathematical knowledge.

We present an integration between two such document formats and systems that combines their advantages: We use MMT system for the content-oriented representation and processing of mathematical objects embedded within \LaTeX documents. MMT dynamically processes the context-sensitive objects and replaces them with presentation markup that is fed back into the \LaTeX processor.

This permits existing \LaTeX -based document preparation workflows to be combined with formal content-based processing. We demonstrate this by using MMT to produce enriched \LaTeX that includes type reconstruction and enables semantic services in the resulting pdf and HTML.

1 Introduction and Related Work

The formalization of mathematics is a major trend in intelligent computer mathematics. A compelling vision for upcoming decades is to build a system that automatically verifies mathematical documents as they are written, thus providing absolute correctness assurances to mathematicians. The area of interactive proof assistants for mathematics has developed multiple systems in that direction, (e.g., the Isabelle framework, the HOL family of systems, Coq, Matita, and Mizar) and has achieved major successes, most recently the formal verification of the Feit-Thompson theorem in Coq.

However, we find in general that proof assistants are not tightly integrated with the workflows commonly used to produce (unverified) mathematical documents. Most of these, such as \LaTeX -based word processors, focus on the markup of metadata, narration, and presentation but avoid formalization entirely. Overcoming this barrier between proof assistants and document preparation workflows has been investigated in multiple ways.

Several proof assistants can export a \LaTeX version of their native input language (e.g., Isabelle and Mizar). Similarly, most can export documents and libraries as HTML, which provides added-value features like navigation and dynamic display. The exported documents do not fully utilize the narrative flexibility of \LaTeX or HTML and are not meant for further editing.

To support the editing work flow, Mizar’s document format has always been designed to closely resemble common narrative mathematical language. Recent work for Isabelle opens up the Isabelle kernel in order to permit better integration with other applications, in particular document-oriented editors [Wen11]. [ABMU11] develops a general Wiki infrastructure that can use Mizar and Coq as verification backends. In the Plato system [WAB07], TeXMacS is used as a document-oriented frontend whose documents are parsed and verified by the Omega proof assistant. All these approaches have in common that they focus on providing a more document-oriented interface to the proof assistant.

Dually, sTeX [Koh08] enhances the L^AT_EX document format with statements for providing content markup. Via L^AT_EXML [GSMK11], this permits generating OMDOC from the same source code that produces PDF documents. OMDOC itself [Koh06] is an XML format that integrates both content and narration markup. Both OMDOC and sTeX do not define a reference semantics that would permit verifying the documents. However, they are also sufficiently generic to eventually allow integration with MMT, if desired.

Somewhat in the middle, controlled natural language systems such as Naproche [CFK+09], MathLang [KMW04], and MathNat [HR10] have been used to parse and verify mathematical documents in formats that are more flexible than proof assistants’ but more restricted than free formats. Some of these permit L^AT_EX fragments as a part of their input language.

A major bottleneck has been that both proof assistants and controlled natural language systems are usually based on a fixed foundation (a logic in which all formalizations are carried out) and a fixed implementation that verifies documents written in a native input language. Therefore, the formalization of a mathematical document D in a proof assistant usually requires the – expensive – formalization of the background theory that connects D to the foundation. Moreover, the fixed implementation is limited to its input language, which in turn cannot be fully understood by any other system.

MMT [RK13] has been developed as a foundation-independent document format coupled with an application-independent implementation. Foundation-independence means that MMT makes no assumptions about the underlying logic and instead represents this logic explicitly as an MMT theory itself. This provides the flexibility to adapt the language to any formal system used to describe a mathematical domain. Application-independence means that the MMT system focuses on building a simple and transparent API along with reusable services. This permits integrating MMT easily into concrete applications and workflows.

In this paper, we leverage and evaluate MMT by integrating its content-oriented services with the L^AT_EX document format. We extend MMT with a concrete input language for mathematical formulas and use that to supplement L^AT_EX’s own formula language. MMT processes these formulas and transforms them into plain L^AT_EX. This has three advantages: *i*) formulas become easy to write and read (e.g., $1+2$ in \mathbb{Z}) while still providing the benefits of content markup (like in $\text{\isoinplus{x}{y}\mathbb{Z}}$), *ii*) MMT can apply type reconstruction services to infer implicit types and arguments and to signal errors for ill-typed

formulas, *iii*) MMT can produce enriched formulas that include hyper-references, tooltips, and interactive display.

We overcome two major challenges to integrate this processing with existing \LaTeX workflows. Firstly, on the technical side, the formulas have to be recognized in the \LaTeX document, processed by MMT, and replaced with generated \LaTeX fragments that are inserted into the document. Secondly, on the theoretical side, we have to account for the fact that formulas are usually not self-contained and depend on a context defined both by the containing document and other documents, e.g., cited \LaTeX documents or formalizations of background theories in proof assistants. Therefore, we have to make these dependencies explicit in the \LaTeX source and make MMT aware of them.

We introduce the input language for MMT in Sect. 3 and solve the above theoretical challenge in Sect. 4. Then we provide two solutions to the technical challenge by integrating MMT with the `pdflatex` and the \LaTeX XML [GSMK11] processors in Sect. 5 and 6, respectively.

We exemplify our development using a running example comprising MMT theories for the logical framework LF [HHP93], sorted-first order logic (SFOL) defined in LF, and monoids and monoid actions defined in SFOL. The sources and the generated documents for all examples are available at [GIR13]. The present paper is already written in MMT- \LaTeX . Therefore, it contains the running example not only as \LaTeX listings but also as processed \LaTeX , which permits experiencing the added-value functionality directly.

2 Preliminaries

```

Theory ::= theory  $T : M = (s [ : \text{Type} ] [\text{Notation} ] | \text{include } T)^*$ 
Type   ::=  $c | x | \text{number} | A(\text{Type}^*) | B(\text{Type}; (x\{k \mapsto \text{Type}\})^*; \text{Type})$ 

```

Fig. 1. A Fragment of the MMT Grammar

We introduce a very simple fragment of MMT’s abstract syntax in Fig. 1 (where BNF meta-symbols are highlighted) that covers our running example. An MMT **theory** $\text{theory } T : M = \Sigma$ defines a theory T with meta-theory M consisting of the declarations in Σ . The **meta-theory** relation between theories is crucial to obtain foundation-independence: The meta-theory gives the language, in which the theory is written. In our running example, the meta-theory of the theory of monoids will be sorted first-order logic, whose meta-theory in turn will be the logical framework LF. All three languages are represented uniformly as MMT theories. Theories are subject to the MMT module system, and we will restrict attention to the simplest case: If a theory T contains a declaration **include** S , then all declarations of S are **included** into T . In our running example, we will use that to extend the theory of monoids to the theory of monoid actions.

Within theories, we declare symbols s with optional types formed from the symbols of the containing theory, its meta-theory, and its included theories. Because the type system is not fixed and the meta-theory may declare arbitrary symbols to introduce new type systems, this is general enough to uniformly represent diverse statements of formal languages such as function symbols, examples, axioms, and inference rules.

The types are essentially the OPENMATH **objects** [BCC⁺04] formed from the available symbols. Here, it suffices to consider the fragment of objects formed from the symbols s , variables x , number literals, applications $A(f, t_1, \dots, t_n)$ of f to the t_i , and bindings $B(b; x_1\{k \mapsto A_1\}, \dots, x_n\{k \mapsto A_n\}; t)$ where a binder b binds the variables x_i with attribution key k and value A_i in the scope t . For simplicity, we assume exactly one attribution per variable although we support the general case of course.

Finally, the grammar already indicates the notations that we introduce later in this paper.

3 Concrete and Pragmatic Syntax for MMT

While MMT theories and meta-theories permit customizing the abstract syntax and the semantics of OPENMATH objects, MMT has so far not permitted customizing the concrete syntax that is visible to the user. Therefore, firstly, we introduce a *notation language* for MMT expressions that governs the transformation between concrete and abstract syntax. Secondly, we introduce *pragmatic and strict syntax* as a new concept that permits abstract syntax transformations that move between encoding levels. By varying the strictness, we can view objects either as first-order logic formulas or as LF expressions.

3.1 A Notation Language for MMT

Notations act as the rules of compositional, bidirectional transformations between abstract and text-based concrete syntax.

Components To write notations conveniently, we represent complex terms as a pair of a symbol s (the **head**) and a list of consecutively numbered **components** as described in Fig. 2. Components can be arguments \mathcal{A}_n , bound variables \mathcal{V}_n , and scopes \mathcal{S}_n . Each notation is attached to an MMT symbol s and governs the transformation of all objects with head s . The transformations are compositional in the sense that they do not inspect the components.

A notation for a complex term is given by mixing delimiters with component identifiers. This corresponds, e.g., to the mixfix notations of Isabelle [Pau94]. For example, a notation for infix conjunction is given as $\mathcal{A}_1 \wedge \mathcal{A}_2$, and the usual notation for the universal quantifier is given as $\forall \mathcal{V}_1. \mathcal{S}_2$. We will introduce some generalizations in the remainder of this section.

Our notation language has the added benefit that it induces an arity system, i.e., by attaching a notation to a symbol s , we also declare what kind of complex

Complex term	Components
$A(s, \mathcal{A}_1, \dots, \mathcal{A}_n)$	$\mathcal{A}_1, \dots, \mathcal{A}_n$
$B(s; \mathcal{V}_1, \dots, \mathcal{V}_n; \mathcal{S}_{n+1})$	$\mathcal{V}_1, \dots, \mathcal{V}_n, \mathcal{S}_{n+1}$
$B(A(s, \mathcal{A}_1, \dots, \mathcal{A}_n); \mathcal{V}_{m+1}, \dots, \mathcal{V}_{m+n}; \mathcal{S}_{m+n+1})$	$\mathcal{A}_1, \dots, \mathcal{A}_n, \mathcal{V}_{m+1}, \dots, \mathcal{V}_{m+n}, \mathcal{S}_{m+n+1}$
$x\{s \mapsto \mathcal{A}_1\}$	\mathcal{A}_1

Fig. 2. Components of Complex Objects whose Head is the Symbol s

terms can be formed from the head s . For example, the notation for conjunction above implies that conjunction is binary and constructs application object. The notation for the universal quantifier implies that \forall constructs a binding object with one bound variable.

Argument and Variable Sequences The arity system induced by the notations given above is not flexible enough because it would give all symbols a fixed arity. This would preclude applying conjunction to any number of arguments or to bind any number of variables in a universal quantifier. Therefore, we expand the notation language to permit *flexary* applications and binders. We achieve this by adding the following modified component identifiers:

Modified Identifier	Meaning
$\mathcal{A}_n D \dots$	argument sequence in position n separated by delimiter D
$\mathcal{V}_n D \dots$	variable sequence in position n separated by delimiter D

Thus, we can give a notation for the flexary conjunction as $\mathcal{A}_1 \wedge \dots$ and a notation for the flexary universal quantifier as $\forall \mathcal{V}_1, \dots \mathcal{S}_2$. Note that we retain the consecutive numbering even though the component identifier \mathcal{V}_1 represents an unknown number of variable declarations.

Implicit Arguments and Types Another important aspect is that not all components always have to be given explicitly in concrete syntax. For example, it is typical to omit the type of a bound variable if it can be inferred from the context. Similarly, arguments can be omitted if their value can be inferred from the remaining arguments. We speak of *implicit* types and arguments.

Therefore, we introduce a simple convention: If a component number n is absent in a notation but a higher number is present, then the missing component is assumed to be an implicit argument \mathcal{A}_n . Similarly, if a binding object binds a variable without an attribution, we assume that the type is implicit.

In both cases, our new MMT parser inserts fresh meta-variables representing the implicit objects. MMT type reconstruction later (attempts to) infer these object from the context.

Notations We can now collect the above concepts and define our notations precisely. We use the following grammar, where a delimiter can essentially be any whitespace-free Unicode string:

Notation	Notation	$::=$	$(\mathbf{C} \mid \mathbf{S} \mid \mathbf{D})^*$
Component	\mathbf{C}	$::=$	$\mathcal{A}_n \mid \mathcal{V}_n \mid \mathcal{S}_n$
Component Sequence	\mathbf{S}	$::=$	$\mathbf{C} \mathbf{D} \dots$
Delimiter	\mathbf{D}		

Some interesting non-trivial notations that MMT now supports are given in Fig. 3. We refer to the appendix for details regarding the parsing algorithm and only point out that a notation technically also includes an integer precedence, which is used to resolve ambiguities when multiple notations are applicable.

Symbol	Concrete syntax	Abstract syntax	Notation
Set	$\{x_1, \dots, x_n\}$	$\mathbf{A}(\text{set}, x_1, \dots, x_n)$	$\{\mathcal{A}_1, \dots\}$
Set attribution	$x \in A$	$x\{\in \mapsto A\}$	$\in \mathcal{A}_1$
Cardinality	$ S $	$\mathbf{A}(\text{card}, S)$	$ \mathcal{A}_1 $
Comprehension	$\{x \in A \mid p(x)\}$	$\mathbf{B}(\text{comp}; x\{\in \mapsto A\}; p(x))$	$\{\mathcal{V}_1 \mid \mathcal{S}_2\}$
Replacement	$\{f(x) : x \in A\}$	$\mathbf{B}(\text{repl}; x\{\in \mapsto A\}; f(x))$	$\{\mathcal{S}_2 : \mathcal{V}_1\}$
Sum	$\sum x = a \text{ to } b.f(x)$	$\mathbf{B}(\mathbf{A}(\text{sum}, a, b); x\{\in \mathbb{Z} \mapsto; \}f(x))$	$\sum \mathcal{V}_3 = \mathcal{A}_1 \text{ to } \mathcal{A}_2 . \mathcal{S}_4$

Fig. 3. MMT Notation Examples

3.2 Pragmatic Abstract Syntax

It is generally difficult to combine content representations that are intuitive to humans with generic type-checking algorithms. If we use a logical framework – in our case LF [HHP93] – and define logics and type theories as MMT theories T with meta-theory LF, we obtain the latter: Type-checking is only implemented once for LF and induces type-checking for T .

But this also restricts the set of symbols that may be used as the head of a complex term. For example, without a logical framework, the conjunction $F \wedge G$ is represented in MMT as $\mathbf{A}(\wedge, F, G)$. But if we define sorted first-order logic (SFOL) in LF, the representation becomes $\mathbf{A}(@, \wedge, F, G)$ where $@$ is the application symbol of LF. A similar, more well-known effect exists for binders where it is usually called higher-order abstract syntax: The binding $\forall x.P$ is represented in pure SFOL as $\mathbf{B}(\forall; x; P)$ but in LF as $\mathbf{A}(@, \forall, \mathbf{B}(\lambda; x; P))$, where λ is the abstraction symbol of LF. Finally, the type attribution $x \in A$ is represented in pure SFOL as $x\{\in \mapsto A\}$ but in LF as $x\{:\mapsto \mathbf{A}(@, \in, A)\}$ where $:$ is the type attribution symbol of LF.

Intuitively, LF squeezes in its own symbols as the heads of all complex terms. This permits LF to “claim ownership” of all complex terms so that their semantics can be induced by that of LF. For example, MMT’s type checking algorithm uses the head of a symbol to ascertain which type checking rule to apply. But this is awkward in practice because SFOL-users perceive the additional symbols as artifacts of an encoding that should remain transparent.

Therefore, we introduce a new principle into MMT’s abstract syntax:

Definition 1. An MMT theory M may distinguish 3 symbols A , B , and K called the **strict** application, binding, and type attribution symbol, respectively.

Let $s \in M$ denote that s is a symbol and is declared in or imported into M . An object t over an M -theory T is called **M -strict** if $s \in M$ for the heads s of all complex subobjects of t .

We simply speak of a strict term if it is M -strict for the highest meta-theory, i.e., in our running example if it is LF-strict. We speak of a *pragmatic* term if it is not strict.

Example 1. The theory LF for a dependently-typed λ -calculus [HHP93] declares the symbols Π , \rightarrow , λ , $\@$, and \cdot .

We distinguish the strict symbols $A = \@$, $B = \lambda$, and $K = \cdot$.

We use the notations $\mathcal{A}_1 \rightarrow \dots$ for \rightarrow and $\text{Pi}\mathcal{V}_1.\mathcal{S}_2$ for Π .

We can now define strictification, the operation that inserts A , B , and K everywhere in order to turn a pragmatic T -term into a strict one:

Definition 2. Assume that M distinguishes A , B , and K as above. The M -strictification \bar{t} of a pragmatic term t is defined as follows:

$$\begin{aligned} \bar{t} &= t \quad \text{if } t \text{ symbol, variable, or number} \\ \overline{A(f, t_1, \dots, t_n)} &= \begin{cases} A(f, \bar{t}_1, \dots, \bar{t}_n) & \text{if } f \in M \\ A(A, \bar{f}, \bar{t}_1, \dots, \bar{t}_n) & \text{otherwise} \end{cases} \\ \overline{B(b; X_1, \dots, X_n; t)} &= \begin{cases} B(b; \bar{X}_1, \dots, \bar{X}_n; \bar{t}) & \text{if } b \in M \\ A(A, \bar{b}, B(B; \bar{X}_1, \dots, \bar{X}_n; \bar{t})) & \text{otherwise} \end{cases} \\ \overline{x\{k \mapsto t\}} &= \begin{cases} x\{k \mapsto \bar{t}\} & \text{if } k \in M \\ x\{K \mapsto A(k, \bar{t})\} & \text{otherwise} \end{cases} \end{aligned}$$

An extended example is presented in the next section.

3.3 Pragmatic Notations for First-Order Logic

We combine notations and pragmatics to obtain a few MMT theories that will serve as running examples. SFOL is a theory with meta-theory LF that represents a fragment of sorted first-order logic, and Monoid is a theory with meta-theory SFOL. Fig. 4 and 5 list the symbols declared in SFOL and Monoid, respectively, with their intuitive meanings, types, and notations.

`sort`, `elem`, and `prop` declare the LF types holding the SFOL-objects. For example, the sort for the base set of a monoid is declared as a constant with type `sort`, and then `elem univ` is the LF-type of SFOL-terms of that sort. Thus, the constant unit has sort `univ`, and the constant `comp` is a binary function symbol on the sort `univ`.

The types of equality and universal quantifier are dependent function types: They take a sort $A : \text{sort}$ as a first argument. Note that the notations make this argument implicit by not mentioning the component \mathcal{A}_1 .

Symbol	Intuition	Type	Notation
sort	type of		
elem	sorts	type	sort
prop	elements of a sort	sort \rightarrow type	$\in 1$
	propositions	type	prop
equal	equality	$\Pi A : \text{sort}.\text{elem } A \rightarrow \text{elem } A \rightarrow \text{prop}$	$\mathcal{A}_2 = \mathcal{A}_3$
forall	universal	$\Pi A : \text{sort} . (\text{elem } A \rightarrow \text{prop}) \rightarrow \text{prop}$	$\forall \mathcal{V}_2 . \mathcal{S}_3$

Fig. 4. SFOL defined with meta-theory LF

Symbol	Intuition	Type	Notation
univ	base set	sort	univ
unit	neutral element	elem univ	e
comp	composition	elem univ \rightarrow elem univ \rightarrow elem univ	$\mathcal{A}_1 \circ \mathcal{A}_2$

Fig. 5. Monoid defined with meta-theory SFOL

Equality then takes two further arguments – terms of sort A – and returns a proposition. The type of the universal quantifier uses higher-order abstract syntax: It takes a function $\text{elem } A \rightarrow \text{prop}$ as a second argument, which is the LF encoding of propositions with a free variable of sort A . It then returns a proposition.

The universal quantifier exemplifies the difference between pragmatic and strict objects. Pragmatically, it is a binder as embodied by its notation: It takes an implicit argument and then binds one variable in a scope. Strictly, it is some LF-symbol with a higher-order function type.

The following example uses the axiom for the neutral element to demonstrate the interplay between parsing, strictification, and type reconstruction. Here we assume precedences are used that avoid brackets, and $?$ represents a meta-variable for the implicit argument.

Concrete syntax $\forall x \in \text{univ} . x \circ e = x$
Abstract syntax pragmatic, i.e., SFOL-strict $B(A(\text{forall}, ?); x\{\text{elem} \mapsto \text{univ}\}; A(\text{equal}, A(\text{comp}, x, \text{unit}), x))$
LF-strict $A(@, A(\text{forall}, ?), B(\lambda; x\{:\mapsto A(\text{elem}, ?)\}; A(\text{equal}, A(\text{comp}, x, \text{unit}), x)))$
LF-strict, after type reconstruction $A(@, A(\text{forall}, \text{univ}), B(\lambda; x\{:\mapsto A(\text{elem}, \text{univ})\}; A(\text{equal}, A(\text{comp}, x, \text{unit}), x)))$

Note that the type attribution in $x\{\in \mapsto \text{univ}\}$ is redundant as well. It can indeed be inferred by MMT and is given here to exemplify the use of a pragmatic type attribution.

4 A Document Format Unifying LaTeX and MMT

In this section, we design a unified format for documents that are at the same time a \LaTeX and an MMT document. The basic idea is to add \LaTeX commands that are orthogonal to \LaTeX processing, produce no visible output, and capture the structure of MMT theories.

Following the grammar, we use three commands for the basic structure of **theories and symbols**. The environment `mmtheory` declares an MMT theory, taking as arguments the name and the optional meta-theory. The command `mmtinclude` with one argument declares an inclusion between two theories. The command `mmtconstant` declares an MMT constant, taking as arguments the name, type, and notation. This command carries one further optional argument: a second notation to be used when rendering objects formed from this symbol. The second notation is useful to use presentation-specific features such as fonts or subscripts that do not affect parsing.

Example 2. We provide an MMT- \LaTeX document for sorted first-order logic, which has the flavor of a literate MMT document, i.e., we define SFOL with meta-theory LF and use \LaTeX for documentation. We only provide an abridged version of the MMT- \LaTeX source; like all our examples, the full version is available at [GIR13]. The rendering is shown in Fig. 6.

```
\begin{theory}[LF]{SFOL}
We define sorted first-order logic in terms of LF.
...
If  $X$  is a sort,  $\text{in } X$  is the type of its terms.
\mmtconstant[\text{in } 1]{elem}{sort -> type}{in 1}
...
 $x =_A y$  is the equality of  $x$  and  $y$  at sort  $A$ .
\mmtconstant[ $=_{1}3$ ]{equal}
  {\Pi A : sort . in A -> in A -> form}{ $= = 3$ }
...
\end{theory}
```

Fig. 6. The Rendering of the Theory SFOL

For mathematical **objects**, in analogy to \LaTeX 's `OBJ`, we define `!OBJ!` for objects given in MMT syntax.

It is typical in mathematical documents that certain toplevel variables are bound within the text and not within a formula. Therefore, we define two additional commands for textual variable declarations:

- `mmtvar` declares a variable taking the name and the type as arguments.
- The environment `mmtscope` declares a scope for textual variables. Variables declared inside a scope are only visible to objects within that scope. Of

course, scopes can be nested arbitrarily. Often this environment is used implicitly by redefining, e.g., `theorem` and `definition` environment to automatically create a scope.

Moreover, because MMT can escape into foreign formats and parse and type-check objects partially, it is possible to escape back and forth between MMT and \LaTeX syntax.

Example 3. Building on SFOL, we declare the theory Monoid using MMT variables and objects to write the axioms. The result is shown in Fig. 7. The \LaTeX code producing the associativity axiom is shown below:

```
\begin{mmtscope}
  For all \mmtvar{x}{in M},\mmtvar{y}{in M},\mmtvar{z}{in M}
  it holds that !(x * y) * z = x * (y * z)!
\end{mmtscope}
```

Fig. 7. The Rendering of the Theory Monoid

This method extends easily to more complex logics. For example, we can describe higher-order logic or ZFC set theory in LF and then use those as meta-theories for mathematical articles. We can also formalize richer type theories, e.g., including as inductive types, record types, or number literals.

\LaTeX source	MMT	\LaTeX style file	defined theory
	lf.mmt	lf.sty	LF
sfol.tex	*sfol.mmt	*sfol.sty	SFOL
algebra1.tex	*algebra1.mmt	*algebra1.sty	Monoid
algebra2.tex	*algebra2.mmt	*algebra2.sty	Action

Fig. 8. Files involved in the running example (* marks generated files)

While **imports between documents** are implicit in common mathematical articles, at most marked up using citations, MMT- \LaTeX document processors must be able to retrieve the relevant portions of imported documents. Therefore, we use the command `mmtimport` with one argument to mark these imports.

When processing an MMT- \LaTeX document D , MMT produces two additional files: an MMT file containing only the MMT symbol declarations made in that document and a \LaTeX package containing commands for rendering these symbols. When encountering the command `\mmtimport{D}`, MMT and \LaTeX load the respective generated file.

Our running example given at [GIR13] is split into multiple files as listed in Fig. 8. For the sake of example, we assume that LF is not defined in a \LaTeX

document but in some other MMT knowledge base. Consequently, we need to provide a corresponding \LaTeX package manually.

Example 4. The file `algebra2` imports `algebra1` (which prompts \LaTeX to load `algebra1.sty`, `sfol.sty`, and `lf.sty`). Then we define the theory of a monoid action by extending the theory of monoids as follows:

```
\mmtimport{algebra1}
\begin{theory}[?SFOL]{Action}
  Consider a monoid  $(M, \circ, e)$  \mmtinclude{?Monoid}
  and a sort called  $set$ . \mmtconstant{set}{sort}{set}

  An \emph{action} of the monoid on  $set$  maps elements of  $M$ 
  to endomappings of  $set$ .
  \mmtconstant[2^1]{act}{in M -> in set -> in set}{2 ^ 1}
  ... such that
  \begin{itemize}
    \item !forall x . (x ^ e) = x!
    \item !forall m. forall n. forall a. (a^m)^n=a^(m*n)!
  \end{itemize}
\end{theory}
```

For the sake of example, we write the axioms with universal quantifiers this time and omit the types of the bound variables. The resulting \LaTeX is given in Fig. 9. Note how the types of the bound variables are displayed as tooltip strings and how the implicit argument of the equality is inferred and added automatically. All symbols carry automatically generated hyperlinks to their point of declaration.

Fig. 9. The Rendering of the Theory Action

5 Integrating MMT into pdf Preparation

The additional \LaTeX commands of our unified document format are processed normally by `pdflatex`. Their expansion eventually produces one of a few new low-level commands that communicate with an MMT server running in the background via HTTP. These commands send the MMT-specific information to the server, and wait for MMT's response. Depending on the command, MMT may respond with generated \LaTeX snippets, which are then further processed by \LaTeX .

This architecture introduces relatively minor communication overhead and permits retaining the full flexibility of \LaTeX . However, in order for \LaTeX to call other programs, it must be run with the `write18 enabled` parameter – a command line switch supported by all major \LaTeX distributions.

```
\lstineline|\mmtconstant[2=_13]{equal}
  {Pi A : sort . in A -> in A -> form}{2 = 3}|
```

For instance, the command given above first registers the declaration of the corresponding symbol in the MMT server. Then MMT generates and returns a \LaTeX command definition for this symbol: `\newcommand\mmtSF0Lequal[3]{#2=_{\#1}\#3}`. The generated commands like `mmtSF0Lequal` remain invisible to the user and only occur in \LaTeX snippets that MMT generates later. Namely, if we later use the command `!x=y!`, this formula is sent to MMT, and – if MMT can infer that x and y have sort A – MMT responds with `\mmtSF0Lequal{A}{x}{y}`.

Actually, the above example is simplified – the snippets returned by MMT are much more complex in order to produce semantically enriched output. For the pdf workflow, we have realized three example features. Recall that the present paper is an MMT- \LaTeX document so that these features can be tried out directly.

Firstly, every generated command definition carries a label, and every later use of that command carries a link to the point of declaration. This permits easy type and definition lookup by clicking on symbols. For example, clicking on the equality symbol in Ex. 4 navigates to the definition of equality in Ex. 2. This links also work across documents.

Secondly, every inferred type of a bound variable is attached to the bound variable as a tooltip. For example, hovering over the bound variable x in Ex. 4 shows it to be an element of the sort *set*. Here we meet the limits of the pdf API: The tooltip can only be a string. Therefore, we use MMT concrete syntax in the tooltip, which is still helpful to authors but less so to readers. It would be straightforward to generate the rendered type if pdf supported it.

An alternative solution is to put the inferred type into the margin or next to the formula. Then we can make it visible upon click using pdf JavaScript. As this is poorly supported by pdf viewers though, we propose another solution: Similar to a list of figures, we produce a list of formulas, in which every numbered formula occurs with all inferred parts. This has the additional benefit that the added value is preserved even in the printed version.

Thirdly, if MMT encounters a type reconstruction error, it returns a \LaTeX snippet consisting of the symbol ζ whose tooltip is the error message. In addition, an undefined command is called so that users can find errors both by browsing the pdf and by scanning the \LaTeX error messages.

Finally, independent of the above, we must provide a way to turn an MMT- \LaTeX document into a pure \LaTeX document in order to submit it to publishers. Therefore, we provide a switch that records all results returned by MMT in an auxiliary file. Then, MMT communication can be disabled and \LaTeX snippets pulled from this file instead.

6 Integrating MMT into HTML Preparation

\LaTeX XML [GSMK11] has been developed to transform \LaTeX documents into a variety of XML formats, notably XHTML and HTML5. It is based on a reimplementation of \TeX 's parser and provides a rich, user-accessible customization layer. Therefore, we can reuse the pdf \LaTeX workflow from Sect. 5 but selectively add enhancements.

Concretely, the \LaTeX snippets generated by MMT contain a number of additional commands, which produce no output when running with `pdflatex`. We provide \LaTeX XML bindings that capture these commands and carry out additional computations to produce `MATHML` and JavaScript that yield further interactive services in the generated document.

This includes in particular showing the inferred types and arguments dynamically. We can also link occurrences of a bound variable to its declaration and dynamically fold subobjects. In fact, any existing interactive services based on `HTML+MATHML` can be enabled easily by including appropriate JavaScript.

A more advanced feature arises if we make use of \LaTeX XML's named environments. For example, in the listing below, the optional `item` parameter for the `mmtvar` declaration of `B` confines the scope of `B` to the current item. Similarly, the variable `A` is in scope for the whole section.

\LaTeX XML keeps track of the currently open narrative structures (chapters, sections, subsections, etc.) and named \LaTeX environments (theories, axioms, theorems, etc.) and binds declared variables to the exact narrative structure desired by the author. When the scope parameter is omitted, the declarations are scoped locally, to the nearest open \LaTeX environment or narrative structure. The default behavior can itself be customized via a `mmtdefaultscope` macro.

```
\section{A Section with a Variable}
\mmtvar[section]{A}{form}
\begin{itemize}
\item Consider a sort \mmtvar[item]{B}{form}. Then ...
\end{itemize}
```

This behavior is realized by additional communication between \LaTeX XML and MMT— when a \LaTeX environment or sectioning block is opened, \LaTeX XML starts a named `mmtscope` and inversely closes it when the \LaTeX scope expires. For example, Fig. 10 shows an MMT error message where the scope of a variable was restricted to an `item` and the variable used elsewhere.

- If `X` is a `sort` then `∈ X` assigns to `X` its type of terms.
- Note that `x†` is out of scope beyond its item.

• `form` is the type of formulas. **MMT Error: parser:sourceerrorat1.1:unboundtoken:X**

Generated on Tue Mar 12 21:53:40 2013 by [\$\text{\LaTeX}\$ XML](#)

Fig. 10. MMT Error Report in an Active Document

7 Conclusion

There is an unrealized potential in formalized mathematics, based on an instance of the Pareto principle: Most of the practical benefits of formalization can be obtained with only a small part of the formalization effort. If we focus on formalizing those steps in a mathematical article that the author explicitly wrote, we can already produce semantically enriched and partially verified documents.

We have developed an integration of MMT and \LaTeX that attempts to exploit this potential. Our approach separates concerns optimally giving MMT authority over content structure and validation and \LaTeX authority over narrative structure and document preparation. The resulting system retains existing \LaTeX workflows and type-setting algorithms and combines them with reconstruction and validation algorithms from formalized mathematics.

Of course, we are not able to verify a document completely this way. But our approach still helps in two ways. Firstly, it can act as a bridge technology that exposes a wider audience of authors of mathematical texts to formalization tools while retaining their existing workflows. Secondly, it enables exporting the high-level structure of a document and its remaining proof obligations for further refinement in a proof assistant. This helps synchronize and establish cross-references between the mathematical document and its detailed formalization.

Our approach permits major generalizations in two directions. The MMT services are not limited to mathematical formulas and can similarly be applied to the theory and proof structure. Dually, we can apply this infrastructure to other document formats than \LaTeX that maintains mathematical content. Any system that communicates context information to MMT in a similar way as we do here can make use of MMT's capabilities to transform user-written strings into semantically enriched formula objects.

References

- ABMU11. J. Alama, K. Brink, L. Mamane, and J. Urban. Large formal wikis: Issues and solutions. In J. Davenport, W. Farmer, J. Urban, and F. Rabe, editors, *Intelligent Computer Mathematics*, pages 133–148. Springer, 2011.
- 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>.
- CFK⁺09. Marcos Cramer, Bernhard Fisseni, Peter Koepke, Daniel Kühlwein, Bernhard Schröder, and Jip Veldman. The naproche project controlled natural language proof checking of mathematical texts. In Norbert E. Fuchs, editor, *CNL*, number 5972 in *Lecture Notes in Computer Science*, pages 170–186. Springer, 2009.
- GIR13. D. Ginev, M. Iancu, and F. Rabe. MMT-LaTeX Example Documents, 2013. see <https://svn.kwarc.info/repos/frabe/Papers/mmt-latex/examples>.
- GSMK11. D. Ginev, H. Stamerjohanns, B. Miller, and M. Kohlhase. The LaTeXML Daemon: Editable Math on the Collaborative Web. In J. Davenport, W. Farmer, J. Urban, and F. Rabe, editors, *Intelligent Computer Mathematics*, pages 292–294. Springer, 2011.

- HHP93. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.
- HR10. Muhammad Humayoun and Christophe Raffalli. Mathnat - mathematical text in a controlled natural language. In *Special issue: Natural Language Processing and its Applications*, volume 46 of *Journal on Research in Computing Science*. National Polytechnic Institute, Mexico, 2010.
- KMW04. Fairouz Kamareddine, Manuel Maarek, and Joe B. Wells. MathLang: An experience driven language of mathematics. In *Electronic Notes in Theoretical Computer Science 93C*, pages 138–160. Elsevier, 2004.
- Koh06. M. Kohlhase. *OMDoc: An Open Markup Format for Mathematical Documents (Version 1.2)*. Number 4180 in *Lecture Notes in Artificial Intelligence*. Springer, 2006.
- Koh08. M. Kohlhase. Using L^AT_EX as a Semantic Markup Format. *Mathematics in Computer Science*, 2(2):279–304, 2008.
- Pau94. L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.
- RK13. F. Rabe and M. Kohlhase. A Scalable Module System. *Information and Computation*, 2013. conditionally accepted; see <http://arxiv.org/abs/1105.0548>.
- WAB07. M. Wagner, S. Autexier, and C. Benzmüller. PLATO: A Mediator between Text-Editors and Proof Assistance Systems. In S. Autexier and C. Benzmüller, editors, *7th Workshop on User Interfaces for Theorem Provers (UITP'06)*, pages 87–107. Elsevier, 2007.
- Wen11. M. Wenzel. Isabelle as a document-oriented proof assistant. In J. Davenport, W. Farmer, J. Urban, and F. Rabe, editors, *Intelligent Computer Mathematics*, pages 244–259. Springer, 2011.

A Details of the Parsing Algorithm

Definition 3. A notation is valid if

- it contains at least one delimiter,
- it contains a scope component iff it contains a variable component, and then it contains exactly one scope component,
- argument components have lower numbers than variable components, which in turn have lower numbers than scope components,
- all components use consecutive number without duplicates except that argument component numbers may be skipped.

Definition 4. A notation set consists of a set of pairs each consisting of a notation and an integer. The integers are called the precedences of the respective notations.

2 notations are ambiguous if they use the same delimiters in the same order.

A notation set is called valid if all notations are valid and if no two ambiguous notations have the same precedence.

Due to the presence of bound variables, precedences, sequence arguments, and sequence variables, no out-of-the-box parsing tool is appropriate for our notation language. Therefore, we have implemented a novel parser. We omit the details of the parsing algorithm here and only sketch the main idea.

1. input: a valid notation set and a list l of tokens
2. state: a tree P in which some inner nodes are labeled by notations and whose leaves are labeled by the tokens in l .
 P is initialized as an unlabeled node whose children are the tokens in l
3. We group the notations by precedence and sort the groups by decreasing order of precedence.
4. We apply each group G of notations exhaustively to P . In each case, for every unlabeled node N
 - (a) We process the children of N in shift-reduce style: We scan from left to right until all delimiters of some notation $n \in G$ have been found.
 - (b) We reduce according to n and continue.
5. We traverse P , collecting for each node the set B of bound variable declarations governing it.

While traversing, we compute an MMT term for each node as follows:

- For each inner node labeled by a notation, we construct the corresponding complex MMT term.
For each implicit argument and type, a fresh meta-variable is inserted.
 - For each unlabeled inner node with children c_0, \dots, c_n , we construct the MMT term $A(c_0, c_1, \dots, c_n)$.
 - For each leaf node l , if $l \in B$, we construct a variable reference, otherwise a symbol reference.
6. output: the term constructed for the root node
or an error if any of the above steps caused ambiguities