Work-in-progress: An MMT-Based User-Interface

Mihnea Iancu and Florian Rabe
Computer Science, Jacobs University, Bremen, Germany

Abstract

The MMT language constitutes a scalable representation and interchange format for formal mathematical knowledge. It is foundation-independent and permits natural representations of the syntax and semantics of virtually all declarative languages. This is leveraged in the MMT API, which provides a variety of generic logical and knowledge management services.

In this work-in-progress report, we present a recently started effort to add editing support for MMT (and thus for any language represented in MMT). To that end, we design a concrete text syntax for MMT and use it as an input language in two MMT user interfaces. Firstly, we connect the MMT API with the text editor jEdit in order to obtain IDE-like support. Secondly, we extend the MMT web server with interface components for wiki-like editing. In both cases, a tight integration of the MMT API and the user interface makes it easy to provide high-level services to the user.

1 Introduction

Deduction systems such as type checkers, proof assistants, or theorem provers have initially focused on soundness and efficiency. At this point the most natural design choice has usually been a read-eval-print loop with large input sequences stored in text files, e.g., [Pau94, TB85]. Over time systems reached maturity levels that called for more sophisticated user interfaces, but it has proved non-trivial to supplement these a posteriori.

The typical difficulty here is the integration of an interaction-oriented user interface with a sequential processing-oriented kernel. This problem is even more severe in the common situation where different programming languages are most suitable for the two respective components.

For example, for Isabelle, [Wen12] provides an IDE-like environment based on jEdit that is similar to ours. The key idea here is to expose much of the ML-based Isabelle kernel to a Scala layer [OSV07]. Since Scala is binary-compatible with Java (the language of jEdit), this permits a fine-granular integration of frontend and kernel, e.g., the editor can display the proof state at the cursor position. In some sense, the opposite approach is taken in [ABMU11], where a wiki-like frontend is provided for Coq and Mizar. Here the kernel is an abstract component that validates files and produces a dependency relation and HTML generation. This permits a scalable integration in a generic wiki framework.

The MMT project offers a somewhat different approach to this problem. It focuses neither on the deduction system nor on the user interface but instead on the content representation language: MMT, a prototypical declarative language. It is foundation-independent inspired by OPENMATH [BCC^{+}04] and OMDOC [Koh06], but it also has a rigorous semantics inspired by the Curry-Howard representation of logical frameworks like LF [HHP93]. Thus, MMT admits natural representations of the syntax and semantics of many formal systems.

Consequently, the implementation of MMT takes the form of an API centered around data structures modeling the MMT language. From this perspective, deduction systems and user interfaces become special cases of content production systems that interact with a central content store maintained by the MMT API. (In fact, we can see them as the extremely machine or human-oriented special case, respectively.)

This design choice is motivated by the assumption that making the representation language strong enough will ultimately benefit the design of both deduction systems and user interfaces as well as their integration.

In this work, we present a recently initiated work-in-progress aimed at validating the second half of that assumption: We design a user-interface based on the MMT API. More concretely, we investigate two different interfaces: an IDE-like one based on the text editor jEdit, which is suited for intensive use, and

Theory Graph	$\mathcal{G}$	$::=$	Mod^*
Module Declaration	Mod	$::=$	$\%\mathtt{sig}\,T =^{[T]} \{Sym^*\}$
Symbol Declaration	Sym	$::=$	$\%\mathtt{include}\ T \mid c[:\omega][=\omega]$
Term	ω	$::=$	$T?c \mid x \mid \omega\,\omega^+ \mid \omega X.\omega \mid \mathtt{String}$
Variable Context	X	$::=$	$\cdot \mid X,\ x[:\omega][=\omega]$
Theory Identifier	T	$::=$	MMT URI
Local Declaration Name	c	$::=$	MMT Name

Figure 1: Simplified MMT Grammar

a web browser-based one, which is suited for occasional use with a low entry barrier. Like MMT, our user interfaces are foundation-independent and applicable to any formal system represented in MMT.

Our work is connected in a tight feedback loop with the design of both the MMT language and the API. Although at this point the various parts of the implementation have reached different levels of maturity, we find the design very promising and have adopted the present implementation in our everyday use.

We will briefly describe the main features of MMT in Sect. 2. Then we describe the data flow we use to convert user-written and thus presentation-oriented input into MMT content representations in Sect. 3. Finally, we describe our two user interfaces in Sect. 4 and 5.

2 The MMT Language

Syntax Fig. 1 describes a very simple fragment of the abstract syntax of the MMT language that is sufficient for the purposes of this paper. We refer to [RK11] for an extensive description.

The central notion is that of an MMT **theory** declaration $\%\mathtt{sig}\,T = \{Sym^*\}$, which introduces a theory with name T containing a list of symbol declarations. (See below for the optional meta-theory in a theory declaration.)

A **symbol** declaration $c : \omega = \omega'$ introduces a symbol named c with **type** ω and **definiens** ω'. Both type and definiens are optional. A symbol declaration of the form $\%\mathtt{include}\ T$ imports the theory T into the current theory.

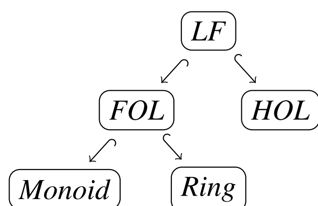
The MMT **terms** ω over a theory T are inspired by OPENMATH objects [BCC+04]. They are formed from constants $T?c$ declared in T, bound variables x, application $\omega\,\omega_1 \ldots \omega_n$ of a function ω to a sequence of arguments, and bindings $\omega X.\omega'$ using a binder ω, a bound variable context X, and a scope ω'. Moreover, MMT terms can mix content and presentation markup and thus include arbitrary presentation markup; for our purposes here, it is sufficient to permit any string as an MMT term.

Every MMT declaration is identified by a canonical, globally unique URI. In particular, the URIs of a constant with name c declared in theory T is given by $T?c$. URIs are **logical identifiers**. Therefore, implementations of MMT maintain a **catalog**, which translates URIs into physical identifiers, e.g., the URLs of files.

Due to the combination of such general symbol declarations with OPENMATH objects, MMT symbol declarations subsume most semantically relevant statements in declarative mathematical languages including function and predicate symbols, type and universe symbols, and — using the Curry-Howard correspondence — axioms, theorems, and inference rules.

Meta-Theories A distinguishing feature of MMT is the explicit formalization of **meta-theories**: A theory declaration $T =^M \{\vartheta\}$ carries an optional meta-theory M. In many situations, this is the same as

$\%\mathtt{sig}\, T = \{\%\mathtt{include}\, M,\ \vartheta\}$, i.e., the symbols of M are additionally available in T. But the meta-theory has a special role: MMT uses M to determine the valid syntax and semantics of T.



Meta-theories permit using theories as a single primitive to represent all formal systems involved in a development – in particular logical frameworks, logics or type theories, and logical theories. This is particularly powerful in conjunction with the MMT module system [RK11], which permits using a coherent module system at all meta-levels.

A typical example is given in the theory graph on the right. Here the theory *FOL* for first-order logic is used as the meta-theory for the logical theories *Monoid* and *Ring*. And the theory *LF* (representing the logical framework LF [HHP93]) is the meta-theory of *FOL* and *HOL* (representing higher-order logic).

Typically the upper-most meta-theory (*LF* in our example) declares only untyped symbols without definiens. This corresponds most closely to an OPENMATH content dictionary. For example, in the case of *LF* (a dependently-typed λ-calculus), these are the constants $\mathtt{type}$, λ, Π, and $\rightarrow$.

Then *FOL* uses these symbols to give types to all its symbols. The latter include the symbols $form : \mathtt{type}$ and $term : \mathtt{type}$ as well as connectives like $\wedge : form \rightarrow form \rightarrow form$. LF's higher-order abstract syntax is used to declare binders as typed symbols. And LF's dependent types are used to declare judgments and proof rules as symbols as well. Similarly, logical theories like *Monoid* use the symbols of *FOL* to give types to their symbols, e.g., $\circ : term \rightarrow term \rightarrow term$.

Semantics The semantics of MMT consists of two parts. Firstly, a generic structural semantics handles all foundation-independent aspects. For example, a term ω is structurally well-formed over the theory T if it only uses constants declared or included into T.

Secondly, MMT treats all expressions (such as terms, types, kinds, etc.) uniformly as MMT terms and does not fix a specific type system. Instead, type systems are provided by **foundations**, which equip the upper-most meta-theories with a typing relation. In the example above, a foundation for *LF* defines the relation $X \vdash_T \omega : \omega$ for any theory T with (possibly indirect) meta-theory *LF*. Thus, MMT is parametric in the type system, and the appropriate foundation is chosen according to the context.

Implementation The MMT API is centered around a model of MMT theory graphs implemented in the Scala programming language [OSV07]. Based on this model, it provides various generic MKM services, in particular presentation and interactive browsing [GLR09], a versioned XML database backend [KRZ10], project management [HIJ+11], change management [IR12], and querying [Rab12].

Implementations of particular foundations are added to MMT as plugins. In the simplest case, these are realized as wrappers around existing tools. Such wrappers currently exist for Mizar [TB85], Twelf [PS99], TPTP [SS98], and OWL [W3C09]. Alternatively, a foundation can be implemented natively. This is more difficult but permits optimal integration with the API. Such a plugin exists for LF, and this induces a native foundation for every formal system represented in LF (e.g., those represented in the LATIN atlas [CHK+11]).

3 From Presentation to Content

We use **interpretation** to refer to the process that transforms presentation-oriented representations into content-oriented ones. In particular, in the case of text input, this corresponds to the typical pipeline of lexing, parsing, and type checking. Our interpretation algorithm is divided into four independent

components that are explained in detail below: *(i)* declaration structuring *(ii)* elaboration of declaration level extensions *(iii)* term structuring *(iv)* term validation

The whole interpretation algorithm tracks the **provenance** of terms. In particular, declaration structuring annotates each declaration and each object in it with the begin-end line-column position. Similarly, term structuring annotates every subobject with its coordinates relative to the beginning of the object. Moreover, all terms produced by a validation procedure (e.g., inferred types or error messages) are annotated with information about the producing procedure. Thus, we can offer content-based services in the presentation-oriented representation such as error hyperlinking, crosslinking between declarations, dynamically displaying inferred content, or highlighting impacted declarations.

3.1 Interpreting Declarations

Structuring The first phase parses MMT declarations such as theory, include, and constant declarations. We omit the details of the concrete text syntax here, which follows the abstract syntax from Fig. 1 very closely. Most notably, this phase leaves all MMT terms as they are and represents them internally as strings.

This phase has two important properties: Firstly, it is **local** in the sense that it can process any MMT theory without retrieving any of its dependencies (such as included theories). Secondly, because we permits mixed content/presentation representation of terms, the result is already **well-formed** MMT.

As errors in the declaration structure are rare and easy to fix, we can assume that this phase always succeeds. Thus, the declaration structure is always available and can be regenerated permanently while the user is typing.

Subsequent phases focus on the interpretation of terms, and these can be executed asynchronously and without any particular order.

Declaration Level Extensions It is straightforward to add to the grammar from Fig. 1 the two primitives of **extension** and **pragmatic** declarations recently presented in [HKR12]. Extensions declarations introduce alternatives to constant declarations, whose semantics is given by elaboration into a list of constant declarations. Pragmatic declarations make use of these extensions.

For example, we can declare extensions for axiom declarations, implicit definitions, case-based function definitions, etc. We only consider a trivial example here and refer to [HKR12] for details. The following extension declaration introduces an extension for axioms:

$$\texttt{\%extension}\ axiom = \lambda_{F:formula}\{assertion : proof\ F\}$$

An axiom declaration takes one formula F as an argument and elaborates into a single constant declaration ax of type $proof\ F$ – the type of proofs of F.

The following pragmatic declaration makes use of this extension to declare an axiom called *myax* that asserts that $0 \neq 1$:

$$\texttt{\%axiom}\ myax\ 0 \neq 1$$

The meta-theory determines whether an extension may be used. For example, in *FOL*, we declare extensions for n-ary function and predicate symbols in addition to the above extension for axioms. Then all theories with meta-theory *FOL* can be formalized using only those three extensions.

These declarations and their semantics are already part of the MMT API but have not been integrated into the interpretation algorithm yet. Our plan is that declaration structuring interprets every unknown keyword (e.g., `%axiom`) as the name of an extension and produces the corresponding pragmatic declaration.

Symbol Declaration	*Sym*	::=	$c[:\omega][=\omega][\#\nu,p]$
Notation	ν	::=	(A | .A | D)*
Argument	A	::=	Integer | Integer D ...
Delimiter	D	::=	String
Precedence	p	::=	Integer

Figure 2: MMT Notation Grammar

Structuring will not yet include an analysis which language extensions are available (which is a non-local property). Instead, an elaboration step retrieves the extensions declared in the respective meta-theory, checks pragmatic declarations against them, and adds the elaboration to the MMT content store.

In our example, elaboration produces the declaration *myax.assertion* : *proof F* with F substituted by $0 \neq 1$. Elaboration is orthogonal to the interpretation of terms. Therefore, the substitution has to be delayed until the term *proof F* has been interpreted.

3.2 Interpreting Terms

MMT's abstract typing relation between terms makes it difficult to take typing information into account while interpreting terms. Therefore, we opt for a two-phased approach. First, *term structuring* uses notations to parse strings into (possibly ill-typed) content markup in a type system-independent way. Then *validation* transforms these terms into well-typed ones; this phase includes foundation-specific type-checking (relegated to plugins) and may infer missing information. We take a broad view on inference here: Validation may, for example, add omitted types or implicit arguments, or discharge proof obligations specified by the user or generated during type checking.

This two-phased approach makes sense in our context but forgoes more sophisticated solutions (e.g., [CZ07]) that use typing information to resolve ambiguities that arise during term structuring.

Structuring The term structuring phase uses a simple generic notation language. In addition to type and definiens, every constant declaration carries an optional notation ν together with a precedence p. Basic notations are inspired by the mixfix form known, e.g., from Isabelle [Pau94]; however, we avoid using flexible precedences for simplicity. In addition, we support flexary operators: constants may take sequences as arguments, which are parsed and rendered by using a separator symbol.

The resulting modifications of the MMT grammar are given in Fig. 2. A notation ν is a list of argument references A or $.A$ and delimiters D. Here $.A$ is a variant of Andrews' dot notation: The argument extends as far to the right as consistent with the placement of brackets. If the n-th argument is a normal argument, the argument reference is simply n. If it is a sequence arguments, the argument reference is of the form $n\,D\ldots$ where D is the delimiter used to separate the elements of the sequence.

For example, we have two ways to declare the conjunction connective: $\wedge\ \#\ 1 \wedge 2$ is used for the usual binary operator. $\wedge\ \#\ 1 \wedge \ldots$ makes it a flexary operator that takes a $\wedge$-separated sequence of arguments.

The implementation is straightforward. It is "almost local" in the sense that all notations from all included theories must be available, but nothing else. Therefore, we first scan all theories and parse their notations (which is easy and quick). Then terms can be structured in any order.

Currently, our notation language does not provide special treatment for binders. Instead, a bound variable is seen as a normal argument (which is always presented as $x:\omega=\omega'$). For example, a flexary λ-binder takes a sequence of bound variables and one scope; a notation is given by $\lambda\ \#\ [1,\ldots]\,.2$

Validation Validation is carried out by foundation-dependent plugins. We do not cover individual plugins here and only discuss how our architecture supports the development of such plugins by handling all bookkeeping tasks. The goal is that plugin implementers have to provide only the core validation algorithms.

Most importantly, validation is decoupled from the user interface. Thus, plugins can asynchronously scan the present terms and apply validation procedures to them. If successful, the result is added to the content store. This may include partial results: for example, if type checking fails at a subterm only, that subterm can be wrapped in an error term giving the expected and the found type.

Moreover, our architecture provides a basis for change management. If the validation of a particular term produces a list of dependencies, MMT can maintain these and revalidate terms whenever their dependents are changed. However, this was added to MMT only recently in [IR12], and is not utilized for the user interface yet.

4 Towards an MMT-based IDE

jEdit is a widely-used Java-based text editor [jEd]. It is particularly interesting as a frontend for formal systems due to its strong plugin infrastructure that can be used to provide IDE-like functionality. Thus, it provides a lightweight alternative to full IDE frameworks like Eclipse [Ecl]. jEdit has also been employed successfully in [Wen12] as a frontend for Isabelle despite the cost of bridging the programming language barrier between ML and Scala/Java.

For our purposes, the situation is even easier as MMT is written in Scala already. Therefore, it is straightforward to integrate MMT's functionality with jEdit, and we have done that in an MMT plugin for jEdit (available at [Rab08]).

This plugin maintains an instance of the MMT API's content store. Our parser is called whenever a file is edited and fills the API's data structures with the result of declaration structuring. (Recall that this is fast and local but already produces well-formed MMT data structures.) In conjunction with the Sidekick plugin [Sid], this is enough to provide an *outline* view (see Fig.3) that provides joint focus between outline and text area.

Moreover, it is already sufficient to offer *auto-completion*: In Fig. 4, we see how the list of all identifiers that can complete a given string is displayed. This list is context-sensitive, i.e., only identifiers that are in scope are listed.

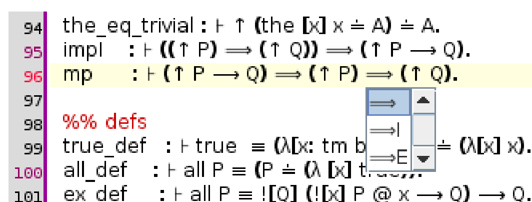


Figure 4: jEdit Autocomplete

Together with provenance tracking, this permits *hyperlinking*: Using the framework of the Hyperlinks plugin [Hyp], hovering over an occurrence of an identifier displays its MMT URI, and clicking on it jumps to its declaration (see Fig. 5).

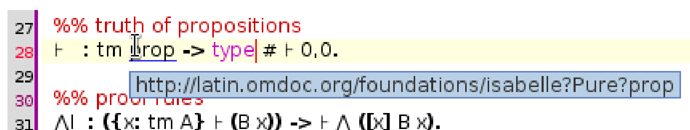


Figure 5: jEdit Hyperlinking

Plugins for foundation-specific validation report errors to the MMT API, and these are displayed using the ErrorList plugin [Err], which shows all known errors in a hyperlinked way. This is shown in Fig. 3 using an error reported by the Twelf system [PS99].

Finally, using the Console plugin [Con], users can directly interact with the MMT shell, e.g., to inspect values or to call MKM services.

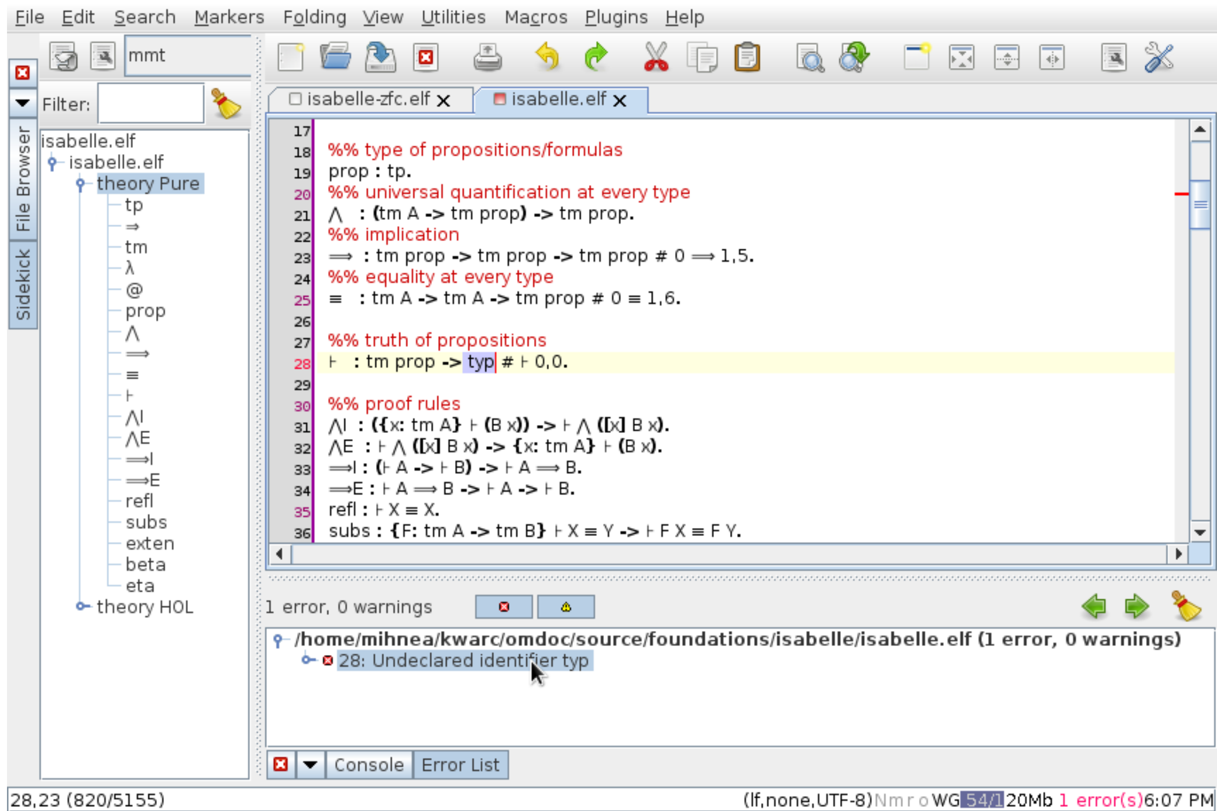


Figure 3: jEdit-base MMT IDE

5 Towards an MMT-based Wiki

The web interface of MMT provides support for browsing MMT repositories. This includes interactive features such as type inference, sub-term folding, and hiding/viewing reconstructed types, implicit arguments, implicit binders or redundant brackets. An instance of the MMT browser is available at [KMR09], where it serves the logic atlas of the LATIN project.

For example, Fig. 6 shows a fragment of the browser windows displaying a theory IMPExt with meta-theory LF. It imports the theory IMP of the implication connective imp and derives two additional proof rules. The declaration imp2I derives the rule $\frac{A,B\vdash C}{\vdash A\,imp\,(B\,imp\,C)}$. The user selected an expression (by clicking on its toplevel node) and chose type inference via the context menu.

The implementation of this feature uses server-side query evaluation. The MMT rendering engine adds parallel markup annotations that identify each rendered subobject. These are used by the JavaScript interface to build a query in the QMT query language for MMT [Rab12]. That query is evaluated on the server and results in the rendered inferred type, which is sent back to the client and displayed.

In a similar way, we obtain web-based editing support. Here, the client sends a query to update a declaration with a new value that is provided as text.

For example, Figure 7(a) shows a simple MMT theory with meta-theory LF while in Figure 7(b) the same theory is rendered as text (using the notations declared for LF primitives) and made available for editing. The theory contains three declarations, two types – o for propositions and i for sets – and one operator. The operator – *and* – is declared with the usual type and with an infix notation which preserves argument order (left argument will be applied at position 0 and the right argument is applied at position

document derived.omdoc

remote module FalsityExt

remote module NEGExt

theory IMPExt meta lf

include IMP

imp2I : ((ded A → ded B → ded C) → ded A imp (B imp C))

= [f:ded A → ded B → ded C]impI ([p:ded A]impI ([q:ded B]f p q))

imp2E : (ded A imp (B imp C) → ded A → ded B → ded C)

= [p:ded A imp (B imp C)][q:ded A][r:ded B]impE (impE p q) r

remote module CONJExt

remote module DISJExt

remote module Equiv

type

ded A imp (B imp C)

infer type

reconstructed types

implicit arguments

implicit binders

redundant brackets

Fold

Figure 6: Dynamic Type Inference

document test.omdoc

theory test meta lf

o : type

i : type

and : o → o → o

(a) MATHML rendering

document test.omdoc

```
%sig test = {
  o : type.
  i : type.
  and : o -> o -> o # 0 and 1,5.
}.
```

Compile

(b) Notation based presentation

Figure 7: A simple MMT theory with meta-theory LF

1). Also, it has precedence 5 and we omit the associativity here for simplicity.

Figure 8 shows the same theory updated with two new constant declarations. The first constant – *not* – is declared with ! as notation and precedence 10 (higher than *and*). While using ! instead of *not* as a notation is arguably unnecessary for such a simple theory, it does highlight the flexibility of our notation language in the sense that the delimiter values are free and may be unrelated to the original constant name. The second constant – *or* – is defined using *not* and *and* in the usual way.

Since structuring is successful, the resulting MMT theory is rendered using MATHML. Otherwise the raised error would have been returned together with a source reference. Note that the higher precedence of the constant *not* (compared to *and*) makes it bind stronger to the variables and serves to disambiguate the input without needing

document test.omdoc

```
%sig test = {
  o : type.
  i : type.
  and : o -> o -> o # 0 and 1,5.
  not : o -> o # ! 0,10.
  or : o -> o -> o = [a : o][b : o] !(!a and !b).
}.
```

Compile

theory test

o : type

i : type

and : o → o → o

not : o → o

or : o → o → o

= [a:o][b:o]not (and (not a) (not b))

Figure 8: The updated theory

extra parentheses.

6 Conclusion

We have presented the first steps of our ongoing efforts to supplement the foundation-independent content representation language MMT with IDE and wiki-like editing frontends. Future work will smoothen the integration of the frontends with the MMT kernel and expose more MMT MKM services to the user.

Despite the relatively early stage of development, it is apparent that MMT provides a very promising basis for a foundation-independent editing framework for formal systems. We expect that our work will evolve into a powerful generic user interface that can be instantiated with arbitrary formal systems. This is particularly attractive for young formal systems with little existing tool support but may also offer interesting alternatives for established systems.

Finally, in future work, we plan to complement our human-oriented interface components for content production with machine-oriented ones. Firstly, this includes the integration of sophisticated external theorem provers. Secondly, it includes the design of an MMT/Scala-based framework in which validation algorithms for different foundations can be composed modularly.

Acknowledgments The parser used for declaration structuring was implemented by Alin Iacob.

References

[ABMU11] Jesse Alama, Kasper Brink, Lionel Mamane, and Josef Urban. Large formal wikis: Issues and solutions. In James Davenport, William Farmer, Florian Rabe, and Josef Urban, editors, *Intelligent Computer Mathematics*, number 6824 in LNAI, pages 133–148. Springer Verlag, 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.

[CHK+11] M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, and F. Rabe. Project Abstract: Logic Atlas and Integrator (LATIN). In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, volume 6824 of *Lecture Notes in Computer Science*, pages 287–289. Springer, 2011.

[Con] jEdit Console Plugin. http://plugins.jedit.org/plugins/?Console. seen May 2012.

[CZ07] Claudio Sacerdoti Coen and Stefano Zacchiroli. Spurious disambiguation error detection. In Manuel Kauers, Manfred Kerber, Robert Miner, and Wolfgang Windsteiger, editors, *Calculemus/MKM*, volume 4573 of *Lecture Notes in Computer Science*, pages 381–392. Springer, 2007.

[Ecl] Eclipse IDE. http://www.eclipse.org/. seen May 2012.

[Err] jEdit ErrorList Plugin. http://plugins.jedit.org/plugins/?Errorlist. seen May 2012.

[GLR09] J. Gičeva, C. Lange, and F. Rabe. Integrating Web Services into Active Mathematical Documents. In J. Carette, L. Dixon, C. Sacerdoti Coen, and S. Watt, editors, *Intelligent Computer Mathematics*, volume 5625 of *Lecture Notes in Computer Science*, pages 279–293. Springer, 2009.

[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.

[HIJ+11] F. Horozal, A. Iacob, C. Jucovschi, M. Kohlhase, and F. Rabe. Combining Source, Content, Presentation, Narration, and Relational Representation. In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, volume 6824 of *Lecture Notes in Computer Science*, pages 211–226. Springer, 2011.

[HKR12] F. Horozal, M. Kohlhase, and F. Rabe. Extending MKM Formats at the Statement Level. In J. Campbell, J. Carette, G. Dos Reis, J. Jeuring, P. Sojka, V. Sorge, and M. Wenzel, editors, *Intelligent Computer Mathematics*, Lecture Notes in Computer Science. Springer, 2012.

[Hyp] jEdit Hyperlinks Plugin. http://plugins.jedit.org/plugins/?Hyperlinks. seen May 2012.

[Int] IntelliJ IDEA. http://www.jetbrains.com/idea/. seen May 2012.

[IR12] M. Iancu and F. Rabe. Management of Change in Declarative Languages. In J. Campbell, J. Carette, G. Dos Reis, J. Jeuring, P. Sojka, V. Sorge, and M. Wenzel, editors, *Intelligent Computer Mathematics*, Lecture Notes in Computer Science. Springer, 2012.

[jEd] jEdit: Programmer's Text Editor. http://www.jedit.org/. seen May 2012.

[KMR09] M. Kohlhase, T. Mossakowski, and F. Rabe. The LATIN Project, 2009. see https://trac.omdoc.org/LATIN/.

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

[KRZ10] M. Kohlhase, F. Rabe, and V. Zholudev. Towards MKM in the Large: Modular Representation and Scalable Software Architecture. In S. Autexier, J. Calmet, D. Delahaye, P. Ion, L. Rideau, R. Rioboo, and A. Sexton, editors, *Intelligent Computer Mathematics*, volume 6167 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2010.

[Net] NetBeans IDE. http://netbeans.org/. seen May 2012.

[OSV07] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. artima, 2007.

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

[PS99] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, 1632:202–206, 1999.

[Rab08] F. Rabe. The MMT System, 2008. see https://trac.kwarc.info/MMT/.

[Rab12] F. Rabe. A Query Language for Formal Mathematical Libraries. In J. Campbell, J. Carette, G. Dos Reis, J. Jeuring, P. Sojka, V. Sorge, and M. Wenzel, editors, *Intelligent Computer Mathematics*, Lecture Notes in Computer Science. Springer, 2012.

[RK11] F. Rabe and M. Kohlhase. A Scalable Module System. see http://arxiv.org/abs/1105.0548, 2011.

[Sid] jEdit SideKick Plugin. http://plugins.jedit.org/plugins/?Sidekick. seen May 2012.

[SS98] G. Sutcliffe and C. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[TB85] A. Trybulec and H. Blair. Computer Assisted Reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28, 1985.

[W3C09] W3C. OWL 2 Web Ontology Language, 2009. http://www.w3.org/TR/owl-overview/.

[Wen12] Makarius Wenzel. Isabelle/jEdit — a Prover IDE within the PIDE framework. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors, *Intelligent Computer Mathematics*, number 7362 in LNAI. Springer Verlag, 2012.