A Scalable Module System

Florian Rabe[a], Michael Kohlhase[a]

[a] *Jacobs University Bremen, Computer Science, Germany*

Abstract

Symbolic and logic computation systems ranging from computer algebra systems to theorem provers are finding their way into science, technology, mathematics and engineering. But such systems rely on explicitly or implicitly represented mathematical knowledge that needs to be managed to use such systems effectively.

While mathematical knowledge management (MKM) "*in the small*" is well-studied, scaling up to large, highly interconnected corpora remains difficult. We hold that in order to realize MKM "*in the large*", we need representation languages and software architectures that are designed systematically with large-scale processing in mind.

Therefore, we have designed and implemented the Mmt language – a module system for mathematical theories. Mmt is designed as the simplest possible language that combines a module system, a foundationally uncommitted formal semantics, and web-scalable implementations. Due to a careful choice of representational primitives, Mmt allows us to integrate existing representation languages for formal mathematical knowledge in a simple, scalable formalism. In particular, Mmt abstracts from the underlying mathematical and logical foundations so that it can serve as a standardized representation format for a formal digital library. Moreover, Mmt systematically separates logic-dependent and logic-independent concerns so that it can serve as an interface layer between computation systems and MKM systems.

Email addresses: f.rabe@jacobs-university.de (Florian Rabe), m.kohlhase@jacobs-university.de (Michael Kohlhase)

URL: http://kwarc.info/frabe/ (Florian Rabe), http://kwarc.info/kohlhase/ (Michael Kohlhase)

Contents

1. Introduction

Mathematics is one of the oldest areas of human knowledge and provides science with modeling tools and a knowledge representation regime based on rigorous language. However, mathematical knowledge has become far too vast to be understood by one person – it has been estimated that the total amount of published mathematics doubles every ten to fifteen years [Odl95]. Indeed, for example, Zentralblatt Math [ZBM31] maintains a database of more than 3 million reviews for articles from 3500 journals from 1868 to 2012.

The currently practiced way to organize mathematical knowledge is to have humans build a cognitive representation of the contents in their minds and to communicate their results in natural – i.e., informal – language with interspersed formulas. This process is well-suited for doing mathematics "*in the small*" where human creativity is needed to create new mathematical insights. But the sheer volume of mathematical knowledge precludes this approach from organizing mathematics "*in the large*": Except for prestige projects such as the classification of finite simple groups [Sol95], collaboration in mathematics is largely small-scale.

This leads to increasing specialization and missed opportunities for knowledge transfer, and the question of supporting the management and dissemination of mathematical knowledge in the large remains difficult. This problem has been tackled in the field of *mathematical knowledge management* (MKM), which uses explicitly annotated content as the basis for mathematical software services such as semantics-based searching and navigation. MKM in the large has been pioneered in the field of *formal methods in software engineering*, where a sound logical foundation and the incorruptibility of computers are combined to verify computer systems. These computer-aided proofs rely on large amounts of formal knowledge about the programming language constructs and data structures, and the productivity of formal methods is restricted in practice by the effectivity of managing this knowledge.

We currently see five obstacles for large scale computerized MKM:

Informality As computer programs still lack any real understanding of mathematics, human mathematicians must make structures in mathematical knowledge sufficiently explicit. This usually means that the knowledge has to be formalized, i.e., represented in a formal, logical system. While it is generally assumed that all mathematical knowledge can in principle be formalized, this is so expensive that it is seldom even attempted.

Logical Heterogeneity One of the advantages of informal, but rigorous mathematics is that it does not force the choice of a formal system. There are many formal systems, each optimized for expressing and reasoning about different aspects of mathematical knowledge. All attempts to find the "mother of all logical systems" (and convince others to use it) have failed. Even though logics themselves can be made the objects of mathematical investigation and even of formalization (in logical frameworks), we do not have scalable methods for

efficiently dealing with heterogeneous, i.e., multi-logic, presentations of mathematical knowledge.

Foundational Assumptions Logical heterogeneity is not only a matter of optimization because different developments of mathematical knowledge make different foundational assumptions. The vast majority of classical mathematics has been formulated in axiomatic set theory following a platonist philosophy. And technical differences between set theories, like Zermelo-Fraenkel or Gödel-Bernays often do not matter. But there are other foundations such as those rejecting the axiom of choice, impredicative definitions, or the principle of excluded middle. Corresponding developments often take a more formalist stance and use different foundations such as higher-order logic or constructive type theory. These alternative foundations have become important in computer-supported formalized mathematics, where almost all proof assistants use foundations that are different from each other and different from classical set theory.

Modularity Modern developments of mathematical knowledge are highly modular. They take pains to identify minimal sets of assumptions so that results are applicable at the most general possible level. This modularity and the mathematical practice of "framing", i.e., of viewing objects of interest in terms of already understood structures, must be supported to even approach human capabilities of managing mathematical knowledge in computer systems.

Global Scale Mathematical research and applications are distributed globally, and mathematical knowledge is highly interlinked by explicit and implicit references. Therefore, a computer-supported management system for mathematical knowledge must support global interlinking and framing as well as management algorithms that scale up to very large (global) data sets.

In this paper we contribute to a uniform solution of four of the five challenges (all but the first[1]): We give a globally scalable module system for mathematical theories (MMT) that abstracts from and mediates between different logics and foundations. With this, we lay a conceptual and technical foundation for formal MKM in the large.

Because our solution draws intuitions from the fields of mathematics, formal methods, and knowledge management, we give a comprehensive overview over the relevant language features and introduce a terminology for them in Section 2. This gives us a solid footing to describe the central design choices underlying MMT in Section 3 and to compare the existing system to each other and to MMT in Section 4.

This is followed by the technical aspects of MMT. First we describe the formal syntax in Section 5. Then we define its semantics by giving an inference

[1] We have already solved the integration of formal and informal mathematical knowledge in the OMDoc format, whose formal part is a predecessor of the work presented in this paper. We plan to integrate this solution with the much stronger formal basis of MMT in the future.

system for well-formed expressions in Section 6 and discuss the meta-theoretical properties of MMT in Section 7. The syntax and semantics of MMT are parametric in what we call foundations, and we look at particular foundations in Section 8.

Finally, we discuss the web scalability of MMT in Section 9 and our implementations in Section 10 and conclude in Section 11.

2. Features of Knowledge Representation Languages

In this section, we develop a taxonomy for modular representation languages for mathematical knowledge. We will use this vocabulary in Section 3 to introduce the main features of MMT and in Section 4 to compare existing module systems to each other and to MMT.

The concepts we introduce are well-established in existing module systems. However, their names vary widely and individual names are heavily overloaded in the literature. Therefore, we give the names used for these concepts in various module systems in Figure 1, 2 and 3 in Section 2.1, 2.2, and 2.3, respectively. We will discuss these module system in detail in Section 4.

By a **module system**, we mean a formal language that provides constructs to express high-level design patterns such as namespaces, imports, parametricity, encapsulation, etc. Very often the modular features of a language can be separated from the non-modular ones. In that case, we call the fragment containing no modularity the **base language**. Typical base languages are logics, type theories, or programming languages. Almost all declarative base languages can be viewed as sequences of declarations (usually of concepts for describing the respective objects and of statements about them). Base language and module system can be designed together or independently, and in the latter case the module system may be designed before or after the base language. We will sometimes use the phrase **modular expression** to distinguish expressions of the module system from those of the base language.

2.1. Scoping Constructs

Module systems typically feature one or both of two main scoping constructs, for which we will use the names *package* and *module*. If both are present, the modules are declared within the packages, and a package can be seen as a group of modules. Figure 1 lists some synonyms commonly used in module systems.

Packages provide scopes for the grouping of related toplevel declarations into – possibly nested – components. The main purpose of packages is **namespace management**: Packages have names, and their named toplevel declarations are identified by a qualified name: a pair of a package name and a declaration name. This facilitates reuse and distribution of declarations over files and networks. Often the packaging structure is transparent to the semantics of the language; in that case the only semantics of packages is that they identify and locate the available toplevel declarations.

We distinguish **open** and **closed** packaging. With open packages, all packages can refer to the toplevel declarations of all other packages via qualified

Synonym	Used in
for package	
cd base	OpenMath
document	OMDoc, Mmt
library	OBJ, CASL, PVS, Coq, Agda
theory	Isabelle[2]
package	Java
namespace	Twelf, XML
for primary modules	
content dictionary	OpenMath
theory	OMDoc, OBJ, IMPS, PVS, Mmt
specification	ASL, CASL
node	development graphs
locale	Isabelle
module type	Coq
module	Agda
signature	SML, Twelf
class	Java

Figure 1: Synonyms for Scoping Constructs

names. With closed packages, import declarations are necessary as only explicitly imported declarations are accessible. In both cases, import declarations are often used to make the imported declarations available without qualification.

When locating package declarations, we speak of **logical** package identifiers if package identifiers are independent of physical locations as given by file systems, databases, and networks; otherwise, we speak of **physical** identifiers. With logical identifiers, the location of resources requires a resolution algorithm that maps logical identifiers to physical locations. This resolution can be relegated to an extra-linguistic **catalog**. Catalogs provide an abstraction layer that makes the distribution of resources over physical locations transparent to the language and avoids conflicts due to naming conventions of operating systems and storage solutions. Using URI-based package identifiers, logical identifiers can be made globally unique to support global interlinking.

Like packages, **modules** are scoped groups of declarations. But contrary to packages, modules are opaque to the language semantics and are used to realize modular design patterns such as inheritance, instantiation, and hiding. For example, moving a declaration between packages has no semantic consequences except that references to the moved declaration must be updated. But a module has a meaning itself that will be affected if a declaration is removed or added.

[2]For Isabelle, the analogy is not as clear as for the other systems. We discuss this in Section 4.

For example, a mathematical theory should be represented as a module because moving axioms between theories changes the semantics; a mathematical paper should be represented as a package because some parts may be moved to other papers without affecting the semantics.

Module systems provide at least one kind of module declarations, which we call the **primary modules**: the ones that correspond to mathematical theories. They may also provide a number of secondary modules, in particular the ones we discuss in Section 2.3.

Languages provide a number of different types of declarations that may occur in the primary kind of module. The most typical declarations are sorts and types, constants and values, operations and functions, and predicates. These are usually named. Further examples of named or unnamed declarations are axioms, theorems, inference rules, abbreviations, or notations for parsing and printing. A named declaration within a module can often be identified as a triple of package name, module name, and declaration name.

As special cases, we obtain languages that feature only packages or only modules. In the former case, every package can be considered to contain a single unnamed module; this is the case in many XML-related languages where the packages are called namespaces such as in XQuery [W3C07]. In the latter case, all modules together can be considered as a single unnamed package; this is the case in SML where a configuration file is used to list the files over which the modules are distributed. We will call the latter **single-package** module systems.

2.2. Inheritance

Synonym	Used in
for named import	
parameter	OBJ
import	Isabelle
module	Coq
structure	SML, MMT, Twelf
aggregation	Java
for unnamed import	
import	OMDoc, OBJ, PVS, Isabelle
definitional link	development graphs
extension	CASL, IMPS, Java
inclusion	Coq, SML, Twelf

Figure 2: Synonyms for Inheritance between Primary Modules

In the simplest case, **inheritance** is a binary relation between modules, which is usually seen as an inheritance graph whose nodes are the modules and whose edges make up the inheritance relation. The individual edges are called **imports**: If T inherits from S, then T imports all knowledge items of S, which

then become available in T. An important distinction is whether the individual imports are **named** or **unnamed**. In the former case, the name of the import is available to refer to (i) the imported module as a whole, or (ii) the imported knowledge items via qualified names. Figure 2 lists some synonyms module systems commonly use for named and unnamed imports.

Inheritance may lead to a **diamond situation** when the same module is imported in two different ways. The language may **identify** multiple imports of the same module or **distinguish** them. If all imports are named, multiply imported knowledge items can be referred to by different qualified names; this makes the distinguish-semantics natural. Similarly, if all imports are unnamed, multiply imported knowledge items have the same name so that the identify semantics is natural. In practice, one often wants to combine distinction and identification. If distinction is the default, **sharing** declarations are used to force the identification of some multiply imported knowledge items. If identification is the default, **renaming** declarations are used to force the distinction of some multiply imported knowledge items. Both paradigms are equally expressive, since identifiers can always be renamed.

A related problem is the **import name clash**, which arises when unnamed imports import from different modules which happen to contain knowledge items with the same local name (i.e. the name in the module). In large-scale developments, this is a very typical situation, which can be difficult to detect. Here module systems may signal an **error**, the knowledge item imported first can be **shadowed** by the one imported later, the name of the module can be used to form a unique **qualified** name, or **overload/identify**-semantics can be used. In the latter case, overloading resolution is used to disambiguate a reference to a knowledge item; and knowledge items that cannot be distinguished in this way (e.g., because they have the same types) are identified.

A more complex form of inheritance is **instantiation**. It means that when importing S into T, some names declared in S may be mapped to expressions of T. This set of mappings can be seen as the passing of argument values over T to parameters of S. If instantiations are possible, multiple imports of the same module with different instantiations should be distinguished. Therefore, the distinguish-semantics is more natural. But it is also possible to identify two imports iff they use the same instantiations.

Module systems differ as to what kind of mappings are allowed in instantiations. Some systems only allow the map of S-symbols to T-symbols. This has the advantage that it is easier to check whether a map is well-typed. Other systems allow mapping symbols to composed expressions. And systems with named imports and realizations can permit **realization maps** (see Section 2.3).

Another difference is which symbols or imports may be instantiated: We speak of a **free** instantiation if arbitrary symbols or imports can be instantiated. Free instantiations must **explicitly** associate some names of S with expressions of T. And we speak of **interfaced** instantiation if the declarations of S are divided into two blocks, and only the declarations in the first block — the interface — are available for instantiations. Interfaced instantiations are often **implicit**: The order of declarations in the interface of S must correspond to

the order of provided T-expressions. Furthermore, instantiations may be **total** or **partial**: Total instantiations provide expressions for all symbols or imports in (the interface of) S. Finally, some systems restrict inheritance to axioms; in such systems, imports must carry instantiations for all symbols; we speak of **axiom-inheritance**.

A further distinction regards the relation between the imports and the other declarations. We speak of **separated** imports if all imports must be given at the beginning of the module; otherwise, we call them **interspersed** imports. Separated imports are conceptually easier, but less expressive: At the beginning of a module, less syntactic material is available to form expressions that can be used in instantiations.

More general forms of imports permit **hiding** and **filtering** of declarations. Both are similar syntactically but not semantically. When importing from S to T, filtering a declaration of S means to exclude that declaration from the import. Hiding is more complicated – one way to think of it is that if a declaration is hidden, it is still imported but rendered inaccessible. In both cases, it is necessary to maintain a dependency relation between declarations: If a declarations is hidden or filtered, so must be all declarations that depend on it.

Hiding can be quite difficult to formalize but has an elegant interpretation in the context of algebraic specification. There, it is used to represent the hiding of implementation details or auxiliary constants. For example, implementations of a specification S must also implement the hidden functions of S, but are considered equal if they differ only in the implementation of hidden functions. We speak of **simple** hiding in this case.

Complex hiding arises if not only declarations, i.e., atomic expressions, can be hidden but composed expressions as well. Syntactically, a complex hiding from S to T can be seen as a morphism from T to S in a category of specifications. Then simple hiding is the special case where S is a fragment of T and the morphism maps every S-symbol to itself. Complex hiding has the appeal that instantiation and hiding become dual to each other.

2.3. Realizations

Many module systems use a concept that we will call **realization**. Its treatment varies substantially between systems, which makes it more difficult to describe abstractly. The common intuition is that we can often think of a primary module as a specification, an interface, or a behavioral description. Then the realizations are the objects that conform to such a specification. Very often it is fruitful to consider the primary modules as types and apply the intuitions of type theory to them. Then a realization is a value that is typed by a primary module.

For example, in SML, the structures are the realizations of the signatures. In Java, the instances of a classes are its realizations. In logic, the models are the realizations of the theories. In formal specification, the implementations are the realizations of the specifications.

Synonym	Used in
for grounded realization	
interpretation	Isabelle
(not parametrized) module	Coq
(toplevel) structure	SML
instance	Java
for view	
theory-inclusion	OMDoc
view	OBJ, CASL, Mmt, Twelf
morphism	ASL
postulated link	development graphs
interpretation	IMPS
for functor	
sublocale	Isabelle
(parametrized) module	Coq
functor	SML

Figure 3: Synonyms for Secondary Modules

In practice, realizations of a primary module S are often given in terms of some context C. Then the realization must provide values over C for all symbols declared in S. Three special cases are of particular importance and frequently occur as **secondary modules**, i.e., other kinds of declarations that can occur in packages besides the primary modules from Figure 1.

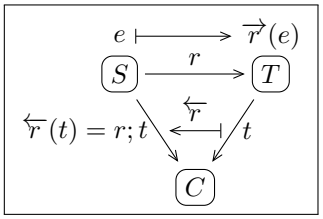
Firstly, if C is the empty context – or more precisely: the global environment implicitly determined by the base language – we speak of **grounded** realizations. For example, in formal specification, the grounded realizations of S are the programs implementing S; the implicit global environment is given by the built-in datatypes and values of the programming language. In logic, the grounded realizations are the models of S; the implicit global environment is given by the foundation of mathematics, e.g., set theory. If we think of the primary modules as types, then we can think of the grounded realizations as values of these types.

Secondly, if C is another module T, we speak of **views** from S to T. Views are closely associated with the intuitions of category theory: Many declarative languages can be naturally formulated as categories with the primary modules as objects and views as morphisms. Views permit framing in the following sense: A view from S to T interprets all declarations of S in terms of T and thus frames T from the perspective of S. Views yield a **homomorphic extension** that maps expressions over S to expressions over T: All symbols in an S-expression are replaced with their T-definition provided by the view. In logic, such views were introduced as *relative interpretations* in [End72].

Thirdly, if C is a formal context binding a list of named declarations, we speak of **functors** with parameter list C and return type S. Functors are closely associated with the intuitions of type theory: If modules are seen as

types and realizations as values, then functors are the module-level analogue of functions. Thus, functors can be used to compute with realizations. For example, if $r(x)$ is a realization of T that is given in terms of a realization x of S, then λ-abstraction yields a functor $\lambda x : S.r(x)$. Then **functor application** maps a realization of S to a realization of T by β-reduction. A module system is **higher-order** if functors may take other functors as arguments.

Both views and functors subsume grounded realizations as a special case: in the former case as views into the empty module, in the latter case as nullary functors. Figure 3 lists some secondary modules used in common module systems and their synonyms.



Views from S to T and unary functors with parameter list $x : T$ and return type S are connected by an adjunction in the sense of category theory [Law63]. Intuitively, a view from S to T corresponds to a functor from T to S and vice versa. The duality between views and functors can be stated as an adjunction between syntax and semantics leading to two important translation functions. Consider a realization r of S in terms of T as in the diagram on the right. The **syntactic translation** $\overrightarrow{r}$ maps S-expressions e to T-expressions using the homomorphic extension of r. The **semantic translation** $\overleftarrow{r}$ maps realizations t of T (in terms of some context C) to realizations of S (in the same context) by composition/application. If a language provides concrete syntax for the semantic translation, it is possible to form **realization expressions**.

For example, let S be the theory of monoids and T the theory of groups, and let r realize every symbol of the language of monoids by its analogue in the language of groups. Then the syntactic translation maps an expression in the language of monoids to the corresponding expression in the language of groups. And the semantic translation maps every group to itself seen as a monoid.

Despite this duality, these two notions are often associated with very different intuitions, and languages often use only one of the two and do not explicate the duality. Specifically, module systems using views usually focus on the syntactic translation, and module systems using functors usually focus on the semantic translation.

We can also recover imports as special cases of realizations. **Imports** are similar to views in that every import from S to T yields a realization of S in terms of T. Just like views, their semantics can be given as a morphism from S to T. Using the intuitions of type theory, declaring a named import from S can be seen as the declaration of a symbol of type S.

If T has a named import from S, then instantiations of T may instantiate that import with a realization expression that realizes S. If a language supports that, we speak of **realization maps**.

Finally, we have the notion of **subtyping** between modules. If every expression over S is also an expression over T, then S is a **syntactic subtype** of T.

Dually, if every realization of S is also a realization of T, then S is a **semantic subtype** of T. If both subtyping relations are present in a language, then they are usually opposites of each other.

The subtype realization can be demonstrated by giving a realization r of S in terms of T whose syntactic and semantic translations are inclusions. Then we can speak of **nominal subtyping** if r is an import, and of **structural subtyping** if r is a view.

2.4. Semantics

There are two ways to give a formal semantics of modular expressions. We speak of a **model theoretical** semantics if models are used to interpret modules. This is typical in the algebraic specification community. We speak of a **proof theoretical** semantics if the semantics is given by typing judgments and inference rules.

Often for some or all modular expressions, there is an expression of the base language with the same semantics. In the type and proof theory community, this is often built-in: The semantics of a modular expression is defined by transforming it into a non-modular one; this is called **elaboration**. In contrast, in languages with a model theoretical semantics, it is often a theorem about the semantics and often called **flattening**.

We say that a module system is **conservative** if every modular expression can be flattened or elaborated into an expression of the base language. Language features that typically prevent conservativity are higher-order functors and hiding.

Similar to conservativity is the **internalization** of a module system. Certain languages are able to express module level judgments in terms of the typing judgment of the base language. This is possible for example, if any concept that can be declared in a primary module can also be declared as a field in a record type. Then primary modules can be internalized as record types, realizations as record terms, grounded realizations as ground terms, functors as functions, and views as unary functions. However, often such expressive record types are not present and can only be added at great cost, e.g., an internalized module system for simple type theory requires type polymorphism. Moreover, in languages where declarations build on each other, dependent record types are needed.

2.5. Genericity

A **logical framework** is a formal representation system that provides an uncommitted set of primitives. Such a framework can be used as a meta-language to define other languages. We call a module system **generic** if it is not specific to a certain base language, but defined within a logical framework. A generic module system is parametrized by an arbitrary base language defined within the logical framework.

We distinguish further whether the logical framework is based on **set theory** or **type theory**. The former typically has a model theoretical, the latter a proof

theoretical semantics. The choice of framework often implies a foundational commitment because the framework must make some assumptions about the base language.

For example, set/model theoretical module systems may assume the semantics of the base language as an institution. An example is ASL based on the framework of institutions [GB92, SW83]. This implies a commitment to a certain axiomatic set theory in which models and institutions are given: For example, foundation with axioms for choice or large cardinals yield different models for a certain module than foundations without them.

Similarly, a type/proof theoretical module system may assume the semantics of the base language as a system of judgments and inference rules. An example is the locale module system based on the logical framework Isabelle [Pau94, KWP99]. This implies a commitment to a formal language in which judgments and inference rules are described. But different logical frameworks permit the representation of different object logics.

We use the term **foundation** to refer to the mathematical theory that formalizes this implicit commitment: the axiomatic set theory in the former, and the logical framework in the latter case. We call a module system **foundation-independent** if it avoids such a commitment. This can for instance be achieved by explicitly representing the foundation itself as a module. Foundation-independent module systems are not only parametric in the base language but also in the foundation used to express the semantics of the base language.

2.6. Degree of Formality

Mathematics has traditionally been written in natural language with interspersed formulas. This is different from the fully formal style that is often used in computer-supported mathematics. Even though the focus of MMT is on formal languages, it is worthwhile to discuss informal languages as well because many aspects of module systems are independent of the degree of formality.

Formal languages are based on a formal syntax with a precisely defined semantics. The syntax is based on a formal grammar that can be implemented so that computers can parse and understand it. A typical service that a computer can offer for a formal language is the **validation** of knowledge to guarantee correctness. Computers can also automatically generate knowledge, such as in automated theorem proving where the generated knowledge item is a proof. This category also includes controlled grammars of natural language that are used to give formal representations a more human-friendly appearance.

Informal languages do not have a formal syntax and are based on unrestricted natural language. While mathematicians use informal language rigorously to obtain an unambiguous semantics, this semantics can only be understood by humans but not by machines. Therefore, only shallow machine-processing services are available such as authoring, storing, and distributing papers and books.

But mathematicians frequently use formal objects within natural language. This has motivated the design of **semi-formal** representation languages that

combine formal and informal representations and degrade gracefully when the latter is used. The automated type-setting provided by LaTeX is a simple example; here the formal representation aspects include the structuring of text into, e.g., definitions, theorems, and formulas.

Note that in the example of LaTeX, the formulas themselves are not formal in our sense: While formal symbols are used, the representation is still human-oriented, and machines can usually not determine the syntax tree of a formula from its LaTeX representation. Such representations are called **presentation**-based and distinguished from **content**-based representations that make the syntax tree accessible to machines.

2.7. Scalability

For machine-processable representation languages, performance and language design are not always orthogonal. We are specifically interested in language aspects that affect scalability.

We call a module system **web standard-compliant** if it provides a concrete syntax that uses XML [W3C98] for all language expressions and URIs [BLFM05] for all identifiers. Such languages can easily support IRIs [DS05] as identifiers, too, which add support for international characters. XML enables standardized document fragment access by technologies like XPath [W3C99] and document fragment aggregation by XQuery [W3C07]. Deployment on web servers allows distributed storage and flexible access methods. URIs provide a standardized and flexible language for logical identifiers. They support the unambiguous identification of all meaningful components of modular theories and provide an abstraction layer over physical locations. An **XML catalog** can translate URIs into their physical locations represented as URLs.

A common feature in implementations of formal languages is a distinction between **external** and **internal** syntax. The former is more relaxed in order to ease reading and writing for humans, whereas the latter is stricter and fully disambiguated to ease machine-processing. An *interpretation* algorithm is used to obtain the internal representation from the external one. Typical steps of the interpretation algorithm are parsing of infix operators using precedences, disambiguation of overloaded symbol names, inference of omitted types, and automated proof search to discharge incurred proof obligations. Moreover, often the internal syntax is non-modular, and the interpretation includes the elaboration or flattening. If interpretation is succeeded by a phase that exports a serialized representation of the internal syntax, possibly optimized for further processing, we speak of *compilation*; this is typical for programming languages.

If different systems are to communicate mathematical knowledge, a complex reconstruction algorithm can be problematic. If internal syntax is communicated, human-oriented information is lost; when external syntax is communicated, the receiving system must implement the costly reconstruction. Therefore, we speak of **authoring-oriented** languages if the reconstruction algorithm is complex and of **interchange-oriented** languages if it is simple (or even the identity).

We speak of **incremental** processing if modular expressions can be processed step-wise. We say that a language is **decomposable** if there is an algorithm that decomposes a modular declaration into a sequence of **atomic** declarations with an acyclic dependency relation. We say that a language is **order-invariant** if the semantics is independent of the order of declarations as as long as the order respects the dependency relation. A decomposable, order-invariant language permits representing documents as sets rather than lists of declarations, which greatly simplifies distribution and storage in databases.

The flattening (elaboration) operation is usually defined by induction on expressions and leads to an exponential increase in size. We speak of **eager** flattening if every induction step requires the recursive flattening of all subexpressions. If we regard flattening as the evaluation of a modular expression, this corresponds to call-by-value evaluation. We speak of **lazy** flattening if a corresponding call-by-reference evaluation is possible. In the latter case, the exponential blow-up may be avoided.

3. Central Features of MMT

We will now discuss the central design goals that have guided the development of MMT in terms of the concepts introduced above. For other systems with different applications and design choices see Section 4.

In this section, we will also introduce a running example that we will use throughout the paper. It is taken from the LATIN atlas [CHK+11], which was built using MMT. The LATIN project developed a library of formalizations of logics and related formal systems currently used in mathematical/logic-based software systems, focusing on modular development and trans-logic interoperability. The whole atlas is available online through the LATIN project [KMR09]. Our running example only comprises a small fragment consisting of algebra, first-order logic, set theory, and the Edinburgh Logical Framework.

A Generic Formal Module System. MMT is a generic, formal module system for mathematical knowledge. It is designed to be applicable to a large collection of declarative formal base languages, and all MMT notions are fully abstract in the choice of base language.

MMT is designed to be applicable to all base languages based on **theories**. Theories are the primary modules in the sense of Section 2. In the simplest case, they are defined by a set of typed **symbols** (the signature) and a set of **axioms** describing the properties of the symbols. A **signature morphism** σ from a theory S to a theory T translates or interprets the symbols of S in T.

If we have entailment relations for the formulas of S and T, a signature morphism is particularly interesting if it translates all theorems of S to theorems of T; this is called a **theory morphism**. Using the **Curry-Howard representation**, MMT drops the distinction between symbols and axioms and between signatures and theories altogether, and only uses theories. An axiom is a constant whose type is the asserted proposition, and a theorem is a defined constant whose definiens is the proof.

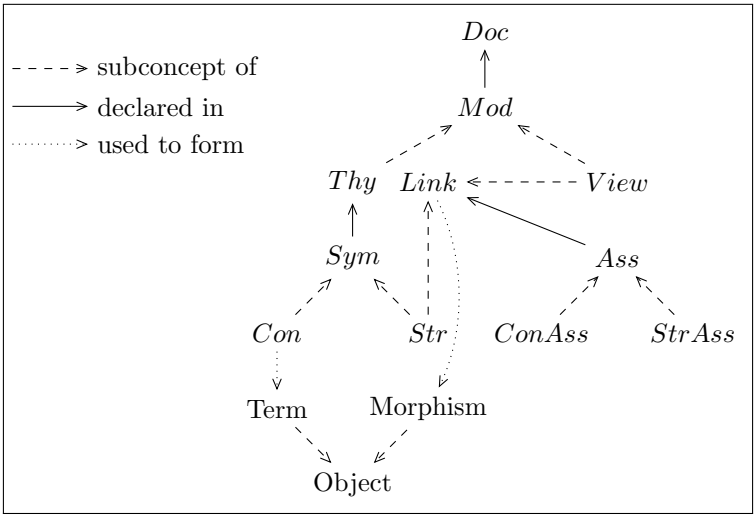


Figure 4: The MMT Ontology

The flat fragment of MMT provides a generic syntax for theories and theory morphisms (called *views* in MMT). A view from S to T is a list of **assignments** $c \mapsto \omega$ where c is an S-constant (axiom) and ω is a T-term (proof). Such a list of assignments induces a **homomorphic** translation of S-terms to T-terms (and S-proofs to T-proofs) by replacing every c with the corresponding ω. Such translations are often called *structural*, *recursive*, or *compositional*.

Full MMT adds the most general form of inheritance: interspersed named imports (called *structures* in MMT) carrying free, explicit, and partial instantiations. In particular, we choose named imports to avoid the problems caused by the diamond situation and import name clashes, which occur frequently in large-scale developments.

MMT has been designed in the tradition of the content-oriented OMDOC and OPENMATH languages to ensure easy machine processing, and MMT builds on their fragments for formal mathematics as the underlying concrete representation languages. We have designed and implemented an extension of MMT [Rab08b] with notation definitions that transform MMT-content representations into presentation-oriented formats like HTML+presentation MathML [ABC+10], but that is not part of this work.

A Simple Language Ontology. A scalable module system must be both expressive and simple, which forms a difficult trade-off. Therefore, MMT carefully picks only a few primitive language features: The ontology of MMT language features is so simple that it can be visualized in a single graph, see Figure 4. MMT concepts are distinguished into four levels: the document, module, symbol, and object level.

The expressions at the document level are the **documents**, which act as

open packages using logical identifiers. MMT systematically follows the intuition that documents are transparent to the semantics so that scalable knowledge management services can be implemented easily at the document level. In particular, MMT documents focus on namespace management and abstraction from physical locations.

MMT documents are open and logical: Every document may refer to every other document (as long as the dependency relation is acyclic) and the distribution of modules into documents and of documents into physical storage is left transparent.

As logical identifiers for all knowledge items, we introduce MMT URIs. The translation into physical identifiers – the URLs – is relegated to an extralinguistic catalog.

Documents contain **modules**, and MMT uses only two kinds of module declarations: **theories** and **views**. MMT does not need other module declarations because both grounded realizations and functors can be represented as views. Most declarative languages can be stated naturally as a category. The objects are sets of declarations and are represented as MMT theories. And the morphisms are translations between theories, which are represented as MMT views.

More precisely, MMT theories contain **symbol declarations**, and views contain **symbol assignments**. A view from theory S to theory T must realize all S-symbols in terms of T-objects. Consequently, for every kind of symbol declaration, there is a corresponding kind of objects. MMT uses only two kinds of symbol declarations: **Constants** represent all declarations of the base language, and **structures** represent named inheritance between theories. A constant assignment provides a T-term for an S-constant, and structure assignments provide a T-morphism for an S-structure.

Objects are complex expressions that represent mathematical expressions, formulas, etc. MMT only uses two kinds of objects: terms and morphisms. Constants occur as the atomic terms, and structures and views as the atomic morphisms. The MMT syntax for **terms** is motivated by the OPENMATH grammar [BCC+04], which uses generic constructs for application of arbitrary operators and binding by arbitrary binders. This is general enough to represent most mathematical languages. The semantics of MMT is generic in that it relegates the semantics of terms to an external definition, which we call the *foundation* below.

Morphisms from S to T are realization expressions representing realizations of S over T. We take the concept of **links** from development graphs [AHMS99] to unify the two atomic morphisms: Structures are morphisms induced by imports, views are morphisms declared (and proved) explicitly. Complex morphisms are formed by composition. The representation of realizations as morphisms has the advantage that MMT can easily provide concrete syntax for the two translations induced by a realization: The syntactic translation is given by applying morphisms to terms, and the semantic translation by composition of morphisms. Thus, MMT can capture the semantics of realizations while being parametric in the semantics of terms.

A Simple Semantics using Theory Graphs. The semantics of a collection of MMT documents is given as a **theory graph**, which serves as a compact specification of a collection of mathematical theories and their relations. The nodes of a theory graph are the theories; the edges are the links. Each path in a theory graph yields a theory morphism. In particular, if a declarative language is given as a category whose components are represented as MMT theories and morphisms, then diagrams in that category are represented as MMT theory graphs. It is a crucial observation that theory graphs are universal in the sense that they arise naturally and in the same way in any declarative language. Using theory graphs, MMT can capture the semantics of modular theories generically.

Example 1 (Running Example: Elementary Algebra) For a simple example, consider the theory graph on the right with nodes for the theories of monoids, commutative groups, and rings, and three structures between them. The theory `Monoid` might declare symbols for composition and unit, and axioms for associativity and neutrality. The theory of commutative groups is an extension of the theory of monoids: it arises by adding symbols and axioms to `Monoid`. Therefore, we only need to represent those added symbols and axioms in `CGroup` and add a structure `mon` importing from `Monoid`.

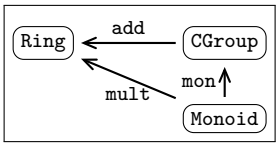


Figure 5 gives a more detailed view of the theory graph adding the symbols in the theory nodes, but eliding the axioms. `Ring` declares two structures for addition and multiplication, and the distinguish-semantics yields two different monoid operations for addition and multiplication.

Structures in MMT are always named and the distinguish-semantics is used in the case of diamonds. Qualified identifiers for the imported constants are formed by concatenating the structure name and the name of the imported symbols. For example, the theory `Ring` from Figure 5 can access the symbols `add/mon/comp` (addition), `add/mon/unit` (zero), `add/inv` (additive inverse), `mult/comp` (multiplication), and `mult/unit` (one).

Both structures and views from S to T are defined by a list of assignments σ that assigns T-objects to S-symbols, and both induce theory morphisms from S to T that map all S-objects to T-objects. This can be utilized to obtain the identify-semantics: sharing declarations are special cases of assignments in structures. MMT assignments support realization maps: structures can be mapped to morphisms.

Example 2 (Sharing) In Example 1, we assumed that all theories are written as theories of first-order logic `FOL` (which we will introduce formally in Example 5). Alternatively, we can use sorted first-order logic `SFOL`. The key difference is that `FOL` uses a single fixed universe, which is explicitly declared in `FOL` and thus automatically available in `FOL`-theories like `Monoid`. `SFOL`, on the other hand, permits an arbitrary set of universes, which are declared individually in `SFOL`-theories; consequently, `SFOL`-function/predicate symbols are sorted, and quantification is relative to a sort. A typical example is the `SFOL`-theory of vector spaces, which declares two sorts for scalars and vectors.

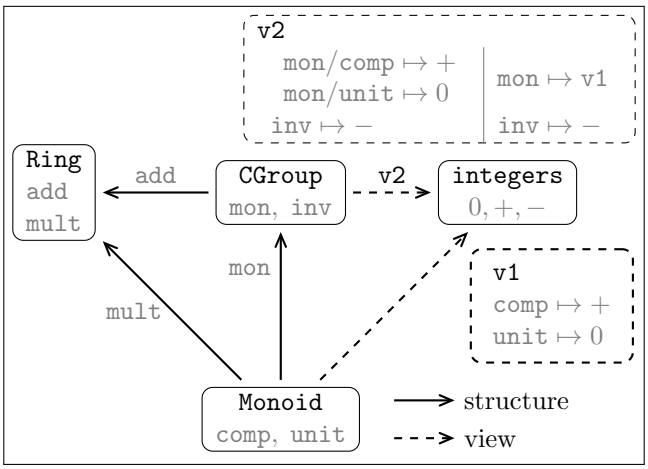


Figure 5: A Theory Graph for Elementary Algebra

Both `SFOL` and `FOL` are justifiable choices for the underlying logic of our running example (and Mmt permits integrating them by using views between logics). Using `SFOL` yields essentially the same theory graph except that all theories have to declare the universe explicitly. For example, the `SFOL`-theory `SMonoid` of monoids declares a sort `univ` for the universe. `SCGroup` and `SRing` arise accordingly, importing the universe declared in `SMonoid`.

However, we now need sharing declarations because `univ` is imported twice from `SMonoid` into `SRing`. Mmt uses asymmetric sharing declarations by first declaring the import `add` and then adding the assignment `univ` ↦ `add/mon/univ` to the structure `mult`. This has the effect of identifying the two copies of the universe. Alternatively, we obtain symmetric sharing declarations by declaring `univ` explicitly in `SRing` as well and adding the assignment `univ` ↦ `univ` to both structures.

Mmt also supports realization maps, which can be used to share whole structures in the same way. We will see examples for that in Section 5.6.

While a view relates two fixed theories without changing either one, structures from S to T occur within T and change T by including a copy of S. Thus, structures induce theory morphisms by definition, and views correspond to representation theorems.

Example 3 (Views (continued from Example 1)) The node on the right side of the graph in Figure 5 represents a theory for the integers declaring the constants 0, +, and −. The fact that the integers are a monoid is represented by the view `v1`. It is a theory morphism that explicitly gives the interpretations of all symbols: `comp` ↦ + and `unit` ↦ 0. If we had not omitted axioms, this view would also have to interpret all the axioms of `Monoid` as proof terms (see Example 4 for that).

The view `v2` is particularly interesting because there are two ways to represent the fact that the integers are a commutative group. In the first variant, all constants of `CGroup` are interpreted separately: `inv` as $-$ and the two imported constants `mon/comp` and `mon/unit` as $+$ and 0, respectively. In the second variant `v2` is constructed modularly by importing the existing view `v1`: The MMT structure assignment `mon` $\mapsto$ `v1` maps all symbol imported by `mon` according to `v1`. The intuition behind a structure assignment is that it makes the right triangle commute: `v2` is defined such that `v2` $\circ$ `mon` $=$ `v1`. Clearly, both variants lead to the same theory morphism; the second one is conceptually more complex but eliminates redundancy because it is structured.

Partial Morphisms. The assignments defining a structure may be (and typically are) partial whereas a view should be total. In order to treat structures and views uniformly, we admit **partial views** as well. This is not only possible, but in fact 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.

Example 4 (Partial Morphisms (continued from Example 3)) Consider for instance the situation in Figure 5 but this time taking axioms into account. Recall that under the Curry-Howard correspondence, axioms are just symbols whose types is given by the asserted formula. So we would have additional constants `assoc` and `neut` for associativity and the properties of the neutral element in `Monoid`, the constants `inv_ax` and `comm` for the properties of the inverse element and commutativity in `CGroup`, and finally the constant `dist` for distributivity in `Ring`.

Thus, the views `v1` and `v2` are clearly partial views, and the missing assignments for `assoc` and `neut` in `v1` and for `inv_ax` and `comm` in `v2` are proof obligations that need to be discharged by proving the translated axioms in theory `integers`. If these proof terms are known, they can be added to the views as assignments to the respective (axiom) constants. In this situation, the structured view `v2` shows its strength: It imports the constant assignments from `v1` that discharge proof obligations so that these proofs do not have to be repeated.

Partial morphisms also arise when representations are inherently partial. For example, we can give a one-sided inverse to the structure `mon` in Figure 5 by mapping `mon/comp` and `mon/unit` to `comp` and `unit`. However, only total morphisms induce functors in the sense of Section 2.3.

MMT introduces **filtering** to obtain a semantics for partial morphisms: All constants for which a view does not provide an assignment are implicitly filtered, i.e., are mapped to a special term $\top$. If a link l from S to T filters a S-constant that has a definiens, this is harmless because the filtered constant can be replaced with its definiens. But if definition-less constants are filtered, MMT enforces the strictness of filtering: All terms depending on a filtered constant, are also

filtered. In that case, we speak of filtered terms, which are also represented by $\top$.

A Foundation-Independent Semantics. Mathematical knowledge is described using very different foundations. Most of them can be grouped into set theory and type theory. Within each group there are numerous variants, e.g., Zermelo-Fraenkel or Gödel-Bernays set theory or set theories with or without the axiom of choice. Therefore, scalability across semantic domains requires a foundation-independent representation language. It is a unique feature of MMT to provide such a high level of genericity and still be able to give a rigorous semantics in terms of theory graphs and a foundation-independent **flattening theorem**.

The semantics of MMT is given proof theoretically by flattening in order to avoid a commitment to a particular model theory. This also makes MMT conservative over the base language so that we can combine MMT with arbitrary base languages without affecting their semantics. Therefore, we have to exclude non-conservative language features, but we have shown in [Rab12a, HR11] that despite the proof theoretical semantics of MMT, model theoretical module systems can be represented in MMT. Moreover, we have given an extension of MMT with hiding in [CHK+12a].

Foundation-independence is achieved by representing all logics, logical frameworks, and the foundational languages themselves simply as theories. For example, an MMT theory graph based on ZFC set theory starts with a theory that declares the symbols of ZFC such as $\in$ and $\subset$. Moreover, MMT does not prescribe a set of well-typed terms. Instead, MMT uses generic term formation operators, and any term may occur as the type of any other term.

We recover this loss of precision by formalizing the notion of *meta-languages*, which pervades mathematical discourse. Let us write M/T to express that we work in the object language T using the meta-language M. For example, most of mathematics is carried out in `FOL/ZFC`, i.e., first-order logic is the meta-language, in which set theory is defined. `FOL` itself might be defined in a logical framework such as *LF* [HHP93], and within *ZFC*, we can define the language of natural numbers, which yields `LF/FOL/ZFC/Nat`. In MMT, all of these languages are represented as theories. In many ways M/T behaves like an unnamed import from M to T, but using only an import would fail to describe the meta-relationship. Therefore, MMT uses a binary **meta-theory** relation between theories.

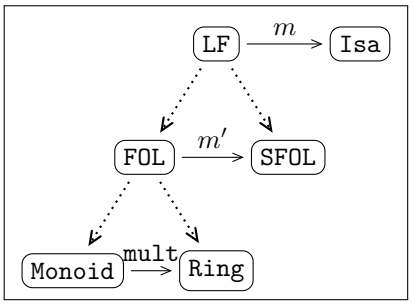


Figure 6: Meta-Theories

In the example in Figure 6 and generally in this paper, the meta-theory relation is visualized using dotted arrows. The theory `FOL` for first-order logic is the meta-theory for `Monoid` and `Ring`. And the theory `LF` for the logical framework LF is the meta-theory of `FOL` and `SFOL`. Note how the meta-theory can indicate

both to humans and to machines how T is to be interpreted. For example, interpretations of `Monoid` are always stated relative to a fixed interpretation of `FOL`.

The importance of meta-theories M/T in MMT is that M defines the semantics of T. More precisely, a **foundational theory** declares all primitive concepts and axioms of the foundational language and occurs as the uppermost meta-theory – like `LF` and `Isa` (Isabelle) in the example in Figure 6. The semantics of the foundational theory is called the **foundation**; it is given externally and assumed by MMT, and it induces the semantics of all other theories. Formally, MMT assumes that the foundation for the foundational theory M defines typing and equality judgments for arbitrary theories T with (possibly indirect) meta-theory M.

The choice of typing and equality is motivated by their universal importance in the formal languages of mathematics and computer science. Here we should clarify that, from an MMT perspective, languages like untyped set theory are in fact typed languages, if only coarsely-typed: For example, typical formalizations of set theory at least distinguish types for sets, propositions, and proofs, and a concise definition of axiom schemes naturally leads to a notion of function types.

As meta-theories are normal MMT theories, they are subject to the same module system. For example, views between logics can be used to move formalizations between logics. In Figure 6, the view m' indicates a logic translation that permits connecting the different theory graphs from Example 1 and 2.

Example 5 (Meta-Theories (continued from Example 4)) We can add meta-theories by adding a theory `FOL` for first-order logic, which occurs as the meta-theory of monoids, groups, and rings.

In particular, `FOL` declares a symbol ι for the fixed universe and symbols for the connectives, quantifiers, and inference rules. We use a theory for `ZFC` as the meta-theory of the integers. In that case the views `v1` and `v2` are only meaningful relative to an interpretation of first-order logic in set theory. In MMT, this interpretation is given as a view `FOLSem` from `FOL` to `ZFC` which is attached to `v1` and `v2` as a meta-morphism.

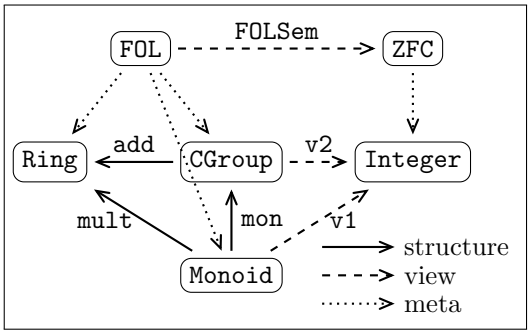


Figure 7: Meta-Theories in Example 5

`FOLSem` represents the inductive interpretation function that defines the semantics of first-order logic in set theory: In particular, it interprets all `FOL`-formulas as booleans, i.e., elements of the set $\{0, 1\}$. Note that `FOL` is also the meta-theory of `ZFC`, we will refine the example accordingly in Example 6.

Little Logics and Little Foundations. The little theories methodology [FGT92] strives to state every mathematical theorem in the theory with the smallest possible set of axioms in order to maximize theorem reuse. Using the **foundations-as-theories** approach, we can extend it to the **little logics** and even the **little foundations** methodology.

MMT provides a uniform module system for theories, logics, and foundational languages. Thus, we can use structures to represent inheritance at the level of logical foundations and views to represent formal translations between them. For example, the morphisms m and m' in Figure 6 indicate possible translations on the levels of logical frameworks and logics, respectively. Therefore, just like in the little theories approach, we can prove meta-logical results in the simplest logic or foundation that is expressive enough and then use views to move results between foundations.

Example 6 (Proof and Model Theory of First-Order Logic) In [HR11], we formalize the syntax, proof theory, and model theory and prove the soundness of first-order logic using a theory graph with `LF` as the ultimate meta-theory of all theories. A fragment of this theory graph is given in the commutative diagram in Figure 8, where we omit `LF` itself for readability.

The representation is more modular than the one sketched in Example 5 in that the theory `FOL` is split into two modules (see Figure 8). The syntax – i.e., the connectives and quantifiers – is defined in the theory `FOLSyn`; the proof theory is defined in the theory `FOLPf`, which imports `FOLSyn` via `syn` and adds constants for the rules of a calculus for first-order logic encoded via the Curry-Howard correspondence.

Moreover, the view `FOLSem` is decomposed into the two views `FOLSem1` and `FOLSem2`, whose composition corresponds to the view `FOLSem` from Example 5. `FOLSem1` interprets the syntax in a theory `FOLMod`. `FOLMod` defines first-order models using higher-order logic `HOL` as the meta-theory. This is sufficient to represent the soundness proof as a view `sound` that interprets all proof terms over `FOLPf` as valid statements over `FOLMod`. `sound` is given as a structured view: It imports the view `FOLSem1` using the structure assignment `syn` $\mapsto$ `FOLSem1`.

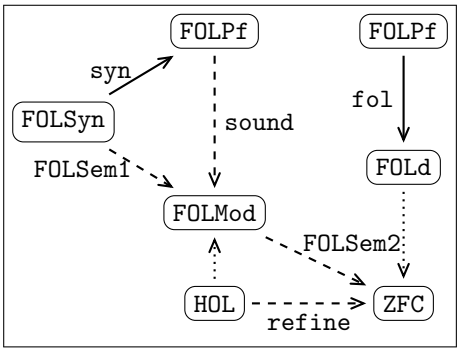


Figure 8: A Theory Graph for First-Order Logic with Proofs & Models

Using `HOL` and `FOLMod` as an intermediate theory for the representation of the model theory has two advantages. `HOL` serves as a little foundation that permits reusing the model theory and the soundness proof in any more expressive foundation such as `ZFC`. Moreover, carrying out the soundness proof is actually easier in `HOL` than in `ZFC` because it permits typed reasoning.

To reuse the model theory and soundness proof in `ZFC`, we use a view `refine`

that interprets HOL in ZFC. It represents the set-theoretical semantics of higher-order logic that interprets types as sets. We can think of this as a refinement from HOL to ZFC. Finally we use refine to give the morphism FOLSem2 that moves HOL-based first-order models to ZFC.

To establish the views with codomain ZFC, we must explicitly represent the axioms and proof system of ZFC. For that purpose, [HR11] uses a variant of first-order logic as the meta-theory of ZFC, namely FOLd. It arises by importing FOLPf via the import fol and then adding a definite description operator. Thus, we have two different interpretations of FOL in ZFC. Firstly, the meta-theory relation supplies ZFC with in particular propositions and connectives. Secondly, the view FOLSem maps FOL-propositions to ZFC-booleans, i.e., elements of the set $\{0, 1\}$. Therefore, we use two copies of the node FOLPf to keep the diagram commutative.

Moreover, [HR11] systematically uses our little logics approach, e.g., it uses separate theories for each connective of first-order logic. Proof theory, model theory, and soundness of each connective or quantifier are represented separately. Thus, they can be reused to define other logics as we do in the LATIN project [CHK+11].

Built-in Web-Scalability. Most module systems in mathematics and computer science are designed with the implicit assumption that all theories of a graph are retrieved from a single file system or server and are processed by loading them into the working memory of a single process. These assumptions are becoming increasingly unrealistic in the face of the growing size of both mathematical knowledge and formalized mathematical knowledge. Moreover, this mathematical knowledge is represented in different formal languages, which are processed with different implementations.

MMT is designed as a representation language that scales well to large interlinked document collections that are processed with a wide variety of systems across networks and implementation languages. Therefore, MMT offers integration support through web standards-compliance, incremental processing of large theory graphs, and an interchange-oriented fully disambiguated external syntax.

Scalable transport of MMT documents must be mediated by standardized protocols and formats. While the use of XML as concrete syntax is essentially orthogonal to the language design, the use of **URIs as identifiers** is not because it imposes subtle constraints that can be hard to meet a posteriori. In MMT, all constants, including imported ones, that are available in a theory have canonical **MMT URIs**. These are tripartite URIs *doc*?*mod*?*sym* formed from a document URI *doc*, a module name *mod*, and a qualified symbol name *sym*. For example, if the theory graph from Figure 5 is given in a document with URI http://cds.omdoc.org/mmt/paper/example, then the constant unit imported from Monoid into CGroup has the URI http://cds.omdoc.org/mmt/paper/example?CGroup?mon/unit.

MMT URIs are always well-formed URIs; in particular, the query part of the URI *doc*?*mod*?*sym* is *mod*?*sym*. The unusual separator ? was carefully chosen as a good combination of logical elegance and web standard-compliance.

Recall that theories are containers for declarations and that constructions like imports define the identifiers that are available in a given theory. Therefore, if every available constant has a canonical identifier, the syntax of identifiers is inherently connected to the possible relations between theories. Consequently, and maybe surprisingly, defining the canonical identifiers is almost as difficult as defining the semantics of the whole language.

All MMT definitions and algorithms are designed with incremental processing in mind. In particular, MMT is decomposable and order-invariant. For example, the declaration $T = \{s_1 : \tau_1,\ s_2 : \tau_2\}$ of a theory T with two typed symbols yields the atomic declarations $T = \{\}$, $T?s_1 : \tau_1$, and $T?s_2 : \tau_2$. Documents, views, and structures are decomposed accordingly. This "unnesting" of declarations is possible because every declaration has a canonical URI so that declarations can be taken out of context for transport and storage and re-assembled later.

The understanding of structures and their induced declarations is crucial to achieve web-scalability. Languages with imports and instantiations tend to be much more complex than flat ones making them harder to specify and implement. Therefore, the semantics of modularity must often remain opaque to generic knowledge management services, an undesirable situation. Because MMT has a simple and foundation-independent flattening semantics, modularity can be made transparent whenever a system is unable to process it.

Moreover, the flattening of MMT is lazy: Every structure declaration can be eliminated individually without recursively flattening the imported theory. Thus, systems gain the flexibility to flatten MMT documents partially and on demand.

4. Related Work

In this section, we survey the state of the art in module systems for formal languages using the terminology developed in Section 2 and relate MMT to them. Figure 9 gives an overview of the discussed systems.

Mathematical Language. Even though mathematical knowledge can vary greatly in its presentation as well as its level of formality and rigor, there is a level of deep semantic structure that is common to all forms of mathematics. This large-scale structure of mathematical knowledge is much less apparent than that of formulas and is usually implicit in informal representations. Experienced mathematicians are nonetheless aware of it, and use it for navigating in and communicating mathematical knowledge.

Much of this structure can be found in networks of theories such as those in a monograph "Introduction to Group Theory" or a chapter in a textbook. The relations among such theories are described in the text, sometimes supported by mathematical statements called "representation theorems". We can observe that mathematical texts can only be understood with respect to a particular mathematical context given by a theory which the reader can usually infer from the document, e.g., from the title or the specialization of the author. The intuitive notion of meta-theory is well-established in mathematics, but again

	OpenMath (CD)	OMDoc (theory)	OBJ (theory)	ASL (specification)	dev. graphs (node)	CASL (specification)	IMPS (theory)	PVS (theory)	Isabelle (locale)	Nuprl	Coq (module type)	Agda (module)	SML (signature)	Java (class)	MMT (theory)	Twelf (signature)	Internalized
formality[1]	pc	pc	c	c	c	c	c	c	c	c	c	c	c	c	c	c	
packages[2]	l	l	s			p	s	p	s		p	p	s	l	l	l	
package imports[3]	o	o				c		c	c		o	c		o	o	o	
named inheritance[4]			s						s		i	i	i	i	i	i	i
instantiation[5]			iit						fep		fep	iit	fep	iit	fep	fep	iit
renaming												+				+	
hiding[6]											s	s	s	s	f		
unnamed inheritance[4]		i	s	a	a	i	a	i	s		i		i	s		i	
diamond semantics[7]		i	i			i		a	a		e		d	i		i	
name clash resolution[8]		q	i			i		q	i		e		s	i		q	
instantiation[5]		fep		fet	fet	iep		iit	fep		fep		fep	iit			
renaming			+			+			+							+	
hiding[6]		f		c	c	s		s			s		s	s			
realization maps			+								+		+	+	+	+	+
grounded realizations									+		+		+	+	+	+	+
views/functors		+	+	+	+	+	+		+		+		+	+	+	+	+
higher-order																	+
translations[9]				y	y						e		e	e	ye	e	e
semantics[10]			m	m	m	m	e	m	e		e	e	e	e	e	e	
internalized[11]										*	*	*		+			
logic-independent			+	+	+	+			+						+	+	
foundation-independent	+	+													+		
URIs as identifiers	+	+												+	+	+	
XML syntax	+	+													+		

[1] p = presentation, c = content
[2] p = physical, l = logical, s = single package
[3] o = open, c = closed
[4] i = interspersed, s = separated, a = axiom-inheritance
[5] i/f = interfaced/free, e/i = explicit/implicit, t/p = total/partial
[6] s = simple, c = complex, f = filtering
[7] i = identify, d = distinguish, a = identify iff instantiations or types agree, e = error
[8] i = overload/identify, q = qualified names, s = shadowing, e = error
[9] explicit syntax for the translation along views/functors: y = syntactic, e = semantic
[10] m = model theory, e = elaboration
[11] + = internalized, ∗ = additionally an internalized module system as in right-most column

Figure 9: Features of Module Systems

it is mainly used informally. Formal definitions are found in the area of logic where a logic is used as the meta-language of a logical theory.

Mathematical theories have been studied by mathematicians and logicians in the search of a rigorous foundation for mathematical practice. They have usually been formalized as collections of symbol declarations and axioms. Mathematical

reasoning often involves several related mathematical theories, and it is desirable to exploit these relationships by moving theorems between theories. A major systematic, large-scale application of this technique in mathematics is found in the works by Bourbaki [Bou68, Bou74], which tried to prove every theorem in the theory with the smallest possible set of axioms.

This technique was formalized in [FGT92], which introduced the *little theories* approach. Theories are studied as formal objects. And structural relationships between them are represented as theory morphisms, which serve as conduits for passing information (e.g., definitions and theorems) between theories (see [Far00]).

Web Scale Languages. The challenge in putting mathematics on the World Wide Web is to capture both notation and meaning in a way that documents can utilize the human-oriented notational forms of mathematics and provide machine-supported interactions at the same time. The W3C recommendation for mathematics on the web is the MathML language [ABC+10]. It provides two sublanguages: **presentation** MathML permits the specification of notations and layout for mathematical formulas, and **content** MathML is geared towards specifying the meaning in a machine-processable way. The latter is structurally equivalent to OpenMath. In particular, both formats represent the structure of mathematical formulas as OpenMath objects, i.e. tree-like expressions built up from constants, variables, and primitive data types via function applications and bindings.

Mmt constants correspond to symbols in MathML and OpenMath and Mmt theories to OpenMath **content dictionaries** (CDs). CDs are machine-readable and web-accessible documents that provide a very basic way to declare mathematical objects for the communication over the WWW and to attach meaning to them. Meaning can be expressed in the form of axioms or types given as OpenMath objects representing logical formulas or in the form of informal mathematical text.

OpenMath provides a certain communication safety over traditional mathematics: It can no longer be the case that the author writes $\mathbb{N}$ for the set of natural numbers with 0, and the reader understands the set of natural number without 0, as the two notions of "natural numbers" — even though presented identically — are represented by different symbols (probably from different CDs). Thus, the service offered by the OpenMath/MathML approach is one of disambiguation as a base for further machine support.

In Mmt terms, the productions for constants, variables, application, and binding correspond closely to OpenMath. Mmt adds morphism application and the special term $\top$, and we omit the primitive data types (integers, strings, floats, ...). We use typed and defined variables in analogy to Mmt constant declarations and do not use the attributions of OpenMath.

OpenMath CDs enable formula disambiguation and web scale communication, but the lack of machine-understandable intra-CD knowledge structure and inter-CD relations preclude higher-level machine support. Therefore, OMDoc [Koh06] represents mathematical knowledge at the levels of objects, state-

ments, theories, and documents: OpenMath and content MathML are subsumed to represent objects. Statements are symbols, axioms, definitions, theorems, proofs and occur as declarations within theories. Moreover, theories may declare unnamed, interspersed, free instantiations, and structured theory morphisms can be declared as in development graphs. Documents provide a basic content-oriented infrastructure for communication and archival.

Syntactically, OMDoc and OpenMath are distinguished from purely formal representation languages by the fact that all formal mathematical elements of the language can be augmented or replaced by natural language text fragments; OMDoc even allows text with interspersed OpenMath objects. Semantically, OMDoc and OpenMath are distinguished because they do not supplement the formal syntax with a formal semantics.

Some implementations of purely formal representation languages have made use of XML, OpenMath/MathML, or OMDoc as primary or secondary representation formats. For example, Mizar [TB85] uses XML as the primary internal format, and Matita [ACTZ06] uses content and presentation MathML; Coq [CH88, BC04] provides an OMDoc export, and Isabelle [Pau94] a partial XML export. Web-scale languages can in principle serve as standardized interchange formats between such systems. Some examples of interoperability mediated by OMDoc and OpenMath are [CHK+12b, CO00, HR09]. But applications have so far been limited due to the lack of an interchange format with a standardized semantics.

Mmt provides such a semantics. It keeps OMDoc's leveled representation but restricts attention to a subset for which a formal semantics can be developed. Syntactically, the main addition of Mmt is the use of named imports and of theory morphisms as objects.

OpenMath and OMDoc use URIs [BLFM05] to identify symbols by triples of symbol name, CD id, and CD base, which corresponds to the triples in Mmt URIs. In particular, the CD base is a URI acting as a namespace identifier. But in OpenMath, the formation of symbol URIs out of those triples is achieved only by imposing a one-CD-one-file restriction, which is too restrictive in general. Moreover, the formation of symbol URIs in OpenMath and OMDoc uses the *fragment* components of URIs. Therefore, fragment access does not scale well because clients have to download a complete document and then execute the fragment access locally. Mmt avoids this by using the *query* component of the URI.

Algebraic Specification Languages. In algebraic specification, theories are used to specify the behavior of programs and software components, and realizations (corresponding to morphisms in Mmt) are used to enable reuse of components (structures in Mmt) and to formalize refinements of specifications (views in Mmt).

In this setting, implementations can be regarded as refinements into executable specifications, which we have called *grounded realizations*. This approach naturally leads to a regime of specification and implementation co-development, where initial, declarative specifications are refined to take op-

erational issues into account. Implementations are adapted to changing specifications, and verification conditions and their proofs have to be adapted as programming errors are found and fixed. This has been studied extensively, and a number of systems have been developed. We will discuss OBJ [GWM+93], ASL [SW83, ST88], CASL [CoF04], and development graphs [AHMS99, MAH06] as representative examples.

OBJ refers to a family of languages based on variants of sorted first-order logic. It was originally developed in the 1970s based on the Clear programming language and pioneered many ideas of modular specifications, in particular the use of initial model semantics [GTW78]. The most important variant is OBJ3; Maude [CELM96] is a closely related system based on rewriting logic. OBJ is a single-package system. Theories and views are similar to Mmt. OBJ permits unnamed imports without instantiation and with identify-semantics, and named imports with interfaced, implicit, and total instantiations. All imports are separated. Realization maps instantiate named imports with realization expressions, which are formed from views.

ASL is a generic module system over an arbitrary institution [GB92] with a model theoretical semantics. Similar to institutions, the focus is on abstract modeling rather than concrete syntax. Modules are called "specifications" and are formed using the operations of union (which can be encoded in Mmt via concatenation of theories), imports, and complex hiding (which was introduced by ASL). Imports between specifications are unnamed and do not use instantiations but only axiom-inheritance and renaming. Unnamed views are used to express refinement theorems. We gave a representation of ASL in Mmt in [CHK+12a], which uses an extension of Mmt to accommodate hiding. ASL+ [Asp94] is an extension of ASL that supports higher-order functors.

The **development graph** language is an extension of ASL specifically designed for the management of change. The central data structure is a theory graph formed from theories and two kinds of links, which corresponds to the theory graphs of Mmt. (Global) "definitional links" are unnamed imports like in ASL and provide axiom-inheritance; (global) "theorem links" are partial views where the missing instantiations are treated as proof obligations that are to be discharged by theorem proving systems. ASL style hiding is supported by *hiding links*. The Maya system [AHMS02] implements development graphs for first-order logic. Like the Mmt implementation and contrary to most other systems discussed here, Maya does not flatten the specification while reading it in. Thus, the modular information, in particular the theory graph, is available in the internal data structures. This is much more robust against changes in the underlying modules and provides a good basis for theorem reuse and management of change.

The development graph calculus pioneers "local links". From the Mmt perspective, a local link is a link which filters all but the local constants of its domain. A global theorem link can be decomposed into a set of commuting local theorem links. By finding these local theorem links individually and reusing them where possible, development graphs can avoid redundancy and maximize

reuse. From the MMT perspective, a decomposed global theorem link is simply a total view without deep assignments, i.e., views where all structures are mapped to morphisms. Thus, MMT provides not only a representation format for development graphs and decomposed theorem links, but also for intermediate development graphs in which theorem links have been partially decomposed or where local theorem links are postulated but have not been found yet.

In general, module-level reasoning in MMT can often utilize such decompositions. Typically, a judgment about all constants in S (including imported ones) is decomposed into separate judgments about the local constants and about the structures declared in S.

The common algebraic specification language (**CASL**) was initiated in 1994 in an attempt to unify and standardize existing specification languages. As such, it was strongly influenced by other languages such as OBJ and ASL. The CASL logics are centered around partial subsorted first-order logic, and specific logics are obtained by specializing (e.g., total functions, no subsorting) or extending (e.g., modal logic or higher-order logic). CASL uses closed physical packages based on files and called "libraries". The modules are called "specifications", the imports are unnamed and interspersed, permit renaming, and use the identify-semantics. The overload/identify-semantics is used to handle import name clashes. Instantiations are interfaced, explicit, and total, and map constants to constants. In parametric specifications, special separated imports are used that can be instantiated with views. CASL offers simple hiding.

In HetCASL [Mos05] and the Hets system [MML07], CASL is extended to heterogeneous specifications using different logics and logic morphisms. Imports and views may go across logics if logic morphisms are attached. This is a very similar to the use of meta-theories and meta-morphisms in MMT. Contrary to MMT, the logics and logic morphisms are implemented in the underlying programming language and not declared within the formal language itself. Hets implements the development graph calculus for heterogeneous specifications. In [CHK+12b], we designed an extension of Hets that can understand logics declared as MMT theories.

Type Theories. Type theories and related formal languages utilize strong logical systems to express both mathematical statements and proofs as mathematical objects. Some systems like AutoMath [dB70], Isabelle [Pau94], or Twelf [PS99] even allow the specification of the logical language itself, in which the reasoning takes place. Semi-automated theorem proving systems have been used to formalize substantial parts of mathematics and mechanically verify many theorems in the respective areas.

These systems usually come with a module system that manages and structures the body of knowledge formalized in the system and a library containing a large set of modules. We will consider the module systems of IMPS [FGT93], PVS [ORS92, OS97], Isabelle [Pau94], Coq [CH88, BC04], Agda [Nor05], and Nuprl [CAB+86]. We will describe the module system of Twelf, which was designed based on MMT, in Section 10.3.

IMPS was the first theorem proving system that systematically exploited the "little theories approach" of separating theories into small modules and moving theorems along theory morphisms. It was initiated in 1990 and is built around a higher-order logic with partial functions. It is a single-package system, the imports are unnamed and separated without instantiations; there is no renaming. Modules can be related via views, which map symbols to symbols.

PVS is an interactive theorem prover for a variant of classical higher-order logic with a rich, undecidable type system. The PVS packages are called "libraries" and are physical packages based on directories. Unnamed, interspersed imports have interfaced, total, and implicit instantiations, which map symbols to terms. Unnamed imports of the same module are identified if the instantiations agree. There is no renaming, and the import name clash problem is handled using the overload/identify semantics. Simple hiding is supported by export declarations that determine which names become available upon import.

Isabelle is an interactive theorem prover based on simple type theory [Chu40] with a structured high-level proof language.

Isabelle "theories" mix features of packages and primary modules. One can see the theories as the primary modules of a single-package system. In particular, they support the typical declarations (constants, axioms, etc.) as well as imports and hiding. However, we will consider Isabelle theories as packages and "locales" as the primary modules. This is more in line with our terminology with the only specialty that the use of primary modules in Isabelle remains optional.

Isabelle theories are closed packages with physical identifiers based on files. Isabelle provides two generic module systems. Originally, only axiomatic type classes were used as modules. They permit only inheritance via unnamed, separated imports without instantiations. Type class ascriptions to type variables and overloading resolution are used to access the symbols of a type class. Later, locales were introduced as modules in [KWP99] and gradually extended. In the current release, locales offer unnamed, separated imports with free instantiations; renaming is possible. Type classes are recovered as a special case.

Realizations are treated differently depending on whether they are grounded or not and whether the domain is a type class or a locale: Views between locales are called "sublocale" and "subclass" declarations, and grounded realizations are called "interpretation" for locales and "instantiation" for type classes.

Isabelle assigns the semantics of a modular theory by elaboration. Locales are internalized by locale predicates that abstract over all symbols and assumptions of the locale; every theorem proved in the locale is relativized by the locale predicate and exported to the toplevel. Thus, instantiation is reduced to β-reduction.

Nuprl is an interactive theorem prover based on a rich undecidable type theory. It does not provide an explicit module system. However, its type theory is so expressive that it can in principle be used to define an internalized module system as shown in [CH00]. Then modules, grounded realizations, and higher-order functors can be defined using Nuprl types, terms, and function terms,

respectively. Named and unnamed imports are defined using intersection and dependent sum types. But Nuprl does not provide specific module system-like syntax for these notions.

Coq is an interactive theorem prover based on the calculus of constructions [CH88]. Physical open packages are called "libraries" and correspond to directories and files.

The Coq module system is modeled after the SML module system (see below). SML signatures, structures, and functors correspond to Coq module types, modules without parameters, and modules with parameters, respectively. Contrary to SML, no shadowing is used, and errors are signaled instead. In addition, Coq can be used with an internalized higher-order module system using record types. As for Nuprl, this yields modules, grounded realizations, and higher-order functors. Both module systems are used independently. The standard library mainly uses the former. The latter is used systematically in [GGMR09].

Agda is a functional programming language based on Martin-Löf's dependent type theory [ML74]. It uses dependent record types to internalize certain theories. In addition, the notion of "modules" combines aspects of what we call packages and modules. These modules are physical closed packages based on files and are used mainly for namespace management. Named interspersed imports between modules are possible using nested module declarations where the inner one is defined in terms of a parametric module. These imports carry interfaced, implicit, and total instantiations that map symbols to terms. Named imports may not occur as parameters so that this does not yield a notion of functors.

Programming Languages. Programming languages differ from the languages mentioned above in that they focus on aspects of execution including input/output and state. But if we ignore those aspects, we find the same module system patterns as in the other languages. We discuss the functional language SML [MTHM97] and the object-oriented language Java [GJJ96] as examples.

SML uses a single-package system that permits the modular design of primary modules (called "signatures") and realizations (called "structures").

Imports between signatures are interspersed and can be named (called "structure declarations") or unnamed (called "inclusions"). Both kinds of imports carry free, explicit, and partial instantiations that map symbols to symbols or structures to realizations. If unnamed imports lead to a diamond situation or a name clash, the later declarations always shadow the previous ones. Views are restricted to inclusion morphisms between signatures (called "structural subtyping"); these views are implicit and inferred by implementations.

Realizations can themselves be given modularly using functors, which give a realization of a signature that is parametric in symbols or structure declarations. Imports between realizations are possible by declaring a structure and defining it to be equal to the result of a functor application. Consequently, these imports

are named and interspersed, and the instantiations are interfaced, explicit, and total and map symbols to symbols and structures to realizations. Structures are typed structurally by signatures, which permits simple hiding.

From an MMT perspective, SML signatures, structures, and functors can be unified conceptually. Signatures correspond to MMT theories in which no constant has a definition; structures to MMT theories in which all constants have definitions; and functors to MMT theories where only a few declarations at the beginning (the interface of the functor) have no definition. Both the structural subtyping relation between signatures and the typing relation between structures and signatures correspond to an inclusion view between the respective MMT theories.

Java uses open packages with optional imports. Package names are the authority components of URIs [BLFM05]. Packages are provided in `jar` archive files, and implementations provide a catalog to locate packages that is based on the `classpath`. Java packages are very close to MMT documents. Similar to MMT, Java identifiers are logical and formed from three hierarchical components: package URI, class name, and field name. However, Java uses "." as a separator character both between and within these components and resolves ambiguities dynamically; MMT uses "?" and "/" so that MMT URIs can be interpreted statically.

Java modules are called "classes". There are two kinds of imports. Firstly, unnamed, separated imports without renaming are called "class inheritance"; a class may only inherit from one other class though. Secondly, named, interspersed imports are called "object instantiation", and the resulting structures "objects". Instantiations are interfaced, implicit, and total, but a class may provide multiple interfaces (called "constructors"), which map symbols to expressions or objects to objects. As constructors may execute code, the expressions passed to the constructor do not have to correspond to symbols or objects declared in the class. Views are restricted to inclusion morphisms out of special modules (called "interfaces"). Simple hiding is realized via private declarations.

Java internalizes its module system, and functors are subsumed by the concept of methods.

5. Syntax

We will now develop the abstract syntax of MMT, our formal module system that realizes the features described in Section 3.

5.1. Grammar

The MMT **grammar** is given in Figure 10 where $^+$, |, and [−] denote nonempty repetition, alternative, and optional parts, respectively. Note that several non-terminal symbols correspond directly to concepts of the MMT ontology given in Section 3. In order to state the flattening theorem below, we also introduce the flat MMT syntax; it arises by removing the productions given in

Theory graph	γ	::=	$\cdot \mid \gamma,\ Thy \mid \gamma,\ View$
Theory	Thy	::=	$T \stackrel{[M]}{=} \{\vartheta\}$
View	$View$	::=	$l : S \to T \stackrel{[\mu]}{=} \{\sigma\} \mid l : S \to T = \mu$
Theory body	ϑ	::=	$\cdot \mid \vartheta,\ Con \mid \vartheta,\ Str$
Constant	Con	::=	$c : \omega = \omega \mid c : \omega \mid c = \omega \mid c$
Structure	Str	::=	$s : S \stackrel{[\mu]}{=} \{\sigma\} \mid s : S = \mu$
Link body	σ	::=	$\cdot \mid \sigma,\ ConAss \mid \sigma,\ StrAss$
Ass. to constant	$ConAss$	::=	$c \mapsto \omega$
Ass. to structure	$StrAss$	::=	$s \mapsto \mu$
Variable context	Υ	::=	$\cdot \mid \Upsilon,\ x[: \omega][= \omega]$
Term	ω	::=	$\top \mid T?c \mid x \mid \omega^{\mu} \mid @(\omega, \omega^+)$ $\mid \beta(\omega; \Upsilon; \omega)$
Morphism	μ	::=	$id_T \mid l \mid \mu\,\mu$
Document identifier	g	::=	URI, no query, no fragment
Module identifier	S, T, M, l	::=	$g?I$
Symbol identifier			$T?I$
Local identifier	c, s, I	::=	$i[/i]^+$
Names	i, x	::=	pchar$^+$
	URI, pchar		see RFC 3986 [BLFM05]

Figure 10: The Grammar for MMT Expressions

gray boxes. We will call any MMT concept **flat**, iff it can be expressed in the flat MMT syntax.

The **meta-variables** we will use are given in Figure 11. References to named MMT knowledge items are Latin letters, MMT objects and lists of knowledge items are Greek letters. We will occasionally use _ as an unnamed meta-variable for irrelevant values.

In the following we describe the syntax of MMT and its intended semantics in a bottom-up manner, i.e., identifiers, object level, symbol level, and module level. Alternatively, the following subsections can be read in top-down order.

Level	Declaration	Expression
Module	theory T, S, R, M link l	theory graph γ (set of modules)
Symbol	constant c structure r, s	theory body ϑ (set of symbols) link body σ (set of assignments)
Object	variable x	term ω morphism μ

Figure 11: Meta-Variables

5.2. Identifiers

All MMT identifiers are URIs and the productions for URIs given in RFC 3986 [BLFM05] are part of the MMT grammar. We distinguish identifiers of documents, modules, and symbols.

Document identifiers g are URIs without queries or fragments (The query and fragment components of a URI are those starting with the special characters ? and #, respectively.).

Module identifiers are formed by pairing a document identifier g with a local module identifier I that is declared in that document. We use ? as a separating character. Similarly, symbol identifiers $T?c$ arise by pairing a theory identifier with an local identifier that is induced in that theory.

Local identifiers may be qualified and are thus lists of names separated by /. Finally, names are non-empty strings of pchars. pchar is defined in RFC 3986 and produces any Unicode character where certain reserved characters must be %-encoded; reserved characters are ?/#[]% and all characters generally illegal in URIs.

Example 7 (Continued from Example 1) We assume that the MMT theory graph for the running example is located in a document with some URI e. Then the MMT URIs of theories and views are for example $e?$`Ring` and $e?$`v1`. The MMT URIs of the constants available in the theory $e?$`Ring` are

- $e?$`Ring?add/mon/comp`,
- $e?$`Ring?add/mon/unit`,
- $e?$`Ring?add/inv`,
- $e?$`Ring?mult/comp`,
- $e?$`Ring?mult/unit`.

and the MMT URIs of the structures available in the theory $e?$`Ring` are

- $e?$`Ring?add`,
- $e?$`Ring?add/mon`,
- $e?$`Ring?mult`,

Structures are special because they may be considered both as symbol level objects available in the theory $e?$`Ring` and as module level objects available globally. This is reflected in MMT by giving structures two identifiers. Consider the structure that imports `Monoid` into `Ring`: If we want to emphasize its nature as a declaration within `Ring`, we use the symbol identifier $e?$`Ring?mult`; if we want to emphasize its nature as a morphism, we use the module identifier $e?$`Ring/mult`.

This has the effect that the non-terminal symbol l for links can produce the identifiers of both views and structures. This is important when studying or implementing MMT because it permits unifying the cases for views and structures.

5.3. The Object Level

Following the OpenMath approach, MMT objects are distinguished into terms and morphisms. **Terms** ω are formed from:

- **constants** $T?c$ referring to constant c declared in theory T,
- **variables** x declared in an enclosing binder,
- **applications** $@(\omega, \omega_1, \ldots, \omega_n)$ of ω to arguments ω_i,
- **bindings** $\beta(\omega_1; \Upsilon; \omega_2)$ by a binder ω_1 of a list of variables Υ with body ω_2,
- **morphism applications** ω^μ of μ to ω,
- a **special term** $\top$ for filtered terms (see below).

Variable contexts are lists of variable declarations. Parallel to constant declarations, variables carry an optional type and an optional definiens. The scope of a bound variable consists of the types and definitions of the succeeding variable declarations and the body of the binder.

For every occurrence of a term, there is a **home theory** against which the term is checked. For occurrences in constant declarations, this is the containing theory. For occurrences in assignments, this is the codomain of the containing link. Relative to a home theory T, we speak of **terms over** T. Terms over T may use $T?c$ to refer to a previously declared T-constant c. And if s is a previously declared structure instantiating S, and c is a constant declared in S, then T may use $T?s/c$ to refer to the copy of c induced by s. Here we assumed that the declarations occur in an order that respects their dependencies; we will see later in Theorem 43 that the precise order chosen does not matter. MMT does not impose a specific typing relation between terms. In particular, well-formed terms may be untyped or may have multiple types.

Example 8 (Continued from Example 7) The running example only contains constants. Complex terms arise when types and axioms are covered. For example, the type of the inverse in a commutative group is $@(\to, \iota, \iota)$. Here $\to$ represents the function type constructor and ι the carrier set. These two constants are not declared in the example. Instead, we will add them in Example 13 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:

$$\omega_e \ := \ \beta(\forall; x : \iota; @(=, @(e?\texttt{Monoid}?\texttt{comp}, e?\texttt{Monoid}?\texttt{unit}, x), x)).$$

Here $\forall$ and $=$ are further constants that are inherited from the meta-theory.

To avoid case distinctions when dealing with declarations $c : \tau = \delta$, we will occasionally write $\tau = \bot$ to express that a constant does not provide a type τ. Accordingly, we write $\delta = \bot$ if it does not provide a definition δ. However, this is only notational convenience, and we do not adopt $\bot$ as a term of the language.

Morphisms are built up from links and compositions. If s is a structure declared in T that imports from S, then T/s is a link from S to T. Similarly, every view m from S to T is a link. Composition is written $\mu\,\mu'$ where μ is applied before μ', i.e., composition is in diagrammatic order. The identity morphism of the theory T is written id_T. 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.

Just like a structure declared in T is both a symbol of T and a link into T, a morphism from S to T can be regarded as a composed object over T. To

stress this often fruitful perspective, we also call the codomain of a morphism its **home theory**, and the domain its **type**. Then morphism composition $\mu' \mu$ can be regarded as the application of μ to μ': It takes a morphism μ' with home theory S and type R and returns a morphism with home theory T of the same type.

Example 9 (Continued from Example 8) In the running example, a morphism is

$$\mu_e \ := \ e?\texttt{CGroup/mon}\, e?\texttt{v2}.$$

It has domain $e?\texttt{Monoid}$ and codomain $e?\texttt{integers}$. 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.,

$$\beta(\forall; x : \iota; @(=, @(+, 0, x), x)).$$

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 by using the identity as a meta-morphism.

We define a straightforward abbreviation for the application of morphisms to whole contexts:

Definition 10 We define Υ^μ by

$$\cdot^\mu \ := \ \cdot \qquad \text{and} \qquad \big(\Upsilon, x : \tau = \delta\big)^\mu \ := \ \Upsilon^\mu,\ x : \tau^\mu = \delta^\mu$$

Here we assume $\bot^\mu = \bot$ to avoid case distinctions.

The analogy between terms and morphisms is summarized in Figure 12.

	Atomic object	Complex object	Type	Checked relative to
Terms	constant	term	term	home theory
Morphisms	link	morphism	domain	codomain

Figure 12: The Object Level

5.4. The Symbol Level

We distinguish four symbol level concepts as given in Figure 13: constants and structures, and assignments to them.

	Declaration	Assignment
Terms	of a constant Con	of a term ω to a constant: $c \mapsto \omega$
Morphisms	of a structure Str	of a morphism μ to a structure: $s \mapsto \mu$

Figure 13: The Symbol Level

A **constant declaration** of the form $c : \tau = \delta$ declares a constant 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 $\bot$ for an omitted type or definition.

Recall that via the Curry-Howard representation, a theorem can be declared as a constant with the asserted proposition as the type and the proof as the definiens. Similarly, (derived) inference rules are declared as (defined) constants.

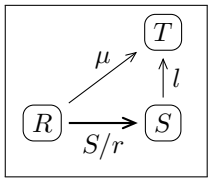
MMT constant declarations do not cover many advanced definition principles, e.g., implicitly defined function or predicate symbols in first-order logic or type definitions in higher-order logic [GP93]. This is necessary because such definition principles are typically not foundation-independent, e.g., an implicit function symbol definition like "f is such that $\forall x.P(x, f(x))$" requires the universal quantifier $\forall$. In [HKR12], we give an extension of MMT that permits the meta-theory to declare definition principles that can then be instantiated by individual constant definitions.

A **structure declaration** of the form $s : S \stackrel{[\mu]}{=} \{\sigma\}$ in a theory T declares a structure s instantiating the theory S defined by assignments σ. Such structures can have an optional meta-morphism μ (see below). Alternatively, a structure may be introduced as an abbreviation for an existing morphism: $s : S = \mu$. This corresponds to constants defined by terms.

While the domain of a structure is given explicitly (in the style of a type), the codomain is the theory in which the structure is declared. Consequently, if $s : S = \mu$ is declared in T, μ must be a morphism from S to T.

Just like symbols are the constituents of theory bodies, assignments are the constituents of link bodies. Let l be a link from S to T. An **assignment to a constant** of the form $c \mapsto \omega$ in the body of l expresses that l maps the constant c of S to the term ω over T. Assignments of the form $c \mapsto \top$ are special: They express that the constant c is **filtered**, i.e., l is definition-less for c.

Assignments must type-check to ensure that typing is preserved by theory morphisms. This means that the term ω must type-check against τ^l where τ is the type of c declared in S.



If r is a structure importing R into S and l a link from S to T, then an **assignment to a structure** of the form $r \mapsto \mu$ in l expresses that l maps r to the morphism μ. Framed in terms of realizations, this means that S declares a realization r of R, and l provides a realization map of r to the realization expression μ over T. Consequently, μ must type-check in the sense that it is a morphism from R to T. Framed in terms of theory graphs, this means that the triangle $S/r\ \ l = \mu$ in the diagram on the right commutes.

Induced Symbols. Intuitively, the semantics of a structure s with domain S declared in T is that all symbols of S are copied into T. For example, if S contains a constant c, then an induced constant s/c is available in T, and so on recursively. In other words, / is used as the operator that dereferences structures.

Similarly, every assignment to a structure induces assignments to constants. Continuing the above example, if a link with domain T contains an assignment to s, this induces assignments to the induced constants s/c. Furthermore, assignments may be **deep** in the following sense: If c is a constant of S, a link with domain T may also contain assignments to induced constants, like s/c. Of course, this can lead to clashes if a link contains assignments for both s and s/c; links with such clashes will have to be rejected by the definition of well-formedness.

Example 11 (Continued from Example 9) The (non-axiom) symbol declarations in the theory `CGroup` are written formally like this:

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

The former is a structure declaration and induces the constants $e?\texttt{CGroup}?\texttt{mon/comp}$ and $e?\texttt{CGroup}?\texttt{mon/unit}$. `Ring` contains only the two structures

$$\texttt{add} : e?\texttt{CGroup} = \{\} \quad \text{and} \quad \texttt{mult} : e?\texttt{Monoid} = \{\}.$$

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 assignments for the induced constants $e?\texttt{CGroup}?\texttt{mon/comp}$ as well as $e?\texttt{CGroup}?\texttt{mon/unit}$. For example, $e?\texttt{CGroup}?\texttt{mon/comp}$ is mapped to $e?\texttt{Monoid}?\texttt{comp}^{e?\texttt{v1}}$, i.e., $e?\texttt{integers}?+$.

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

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

Example 12 (Continued from Example 2) If we use the `SFOL`-based formalization of algebra, we have a theory `SMonoid` declaring `univ`. In that case `SRing` inherits two instances of `univ`, which must be shared. Therefore, `SRing` would contain the two structures

$$\texttt{add} : e?\texttt{SCGroup} = \{\}$$
$$\texttt{mult} : e?\texttt{SMonoid}' = \{\texttt{mon/univ} \mapsto e?\texttt{SRing}?\texttt{add/mon/univ}\}$$

5.5. The Module Level

On the module level a **theory declaration** of the form $T \stackrel{[M]}{=} \{\vartheta\}$ declares a theory T defined by a list of symbol declarations ϑ, which we call the **body** of T. Theories have an optional meta-theory M. A **View declarations** of the form $m : S \to T \stackrel{[\mu]}{=} \{\sigma\}$ declares a view m from S to T defined by a list of assignments σ and by an optional meta-morphism μ. Just like structures, views may also be defined by an existing morphism: $m : S \to T = \mu$.

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 theory graph. If M is the meta-theory of T, then T may use all symbols of M. M provides the syntactic material that T can use to define the semantics of its symbols.

Because a theory S with meta-theory M implicitly imports 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 from M to T (or any meta-theory of T).

Example 13 (Continued from Example 11) We can now combine the situations from Example 5 and 6 in one big MMT theory graph. In a document with URI m, we declare an MMT theory for the logical framework as

$$m?\texttt{LF} = \{\texttt{type}, \to, \ldots\}$$

where we only list the constants that are relevant for our running example: `type` represents the kind of types, and $\to$ is the function type constructor. We will assume $\to$ to be n-ary and right-associative, e.g., we write $@(m?\texttt{LF}?\to, \omega_1, \omega_2, \omega_3)$ for $@(m?\texttt{LF}?\to, \omega_1, @(m?\texttt{LF}?\to, \omega_2, \omega_3))$.

We declare a theory for first-order logic in a document with URI f like this:

$$f?\texttt{FOLSyn} \stackrel{m?\texttt{LF}}{=} \left\{ \begin{array}{l} \iota : m?\texttt{LF}?\texttt{type},\ \texttt{o} : m?\texttt{LF}?\texttt{type}, \\ \texttt{equal} : @(m?\texttt{LF}?\to, f?\texttt{FOLSyn}?\iota, f?\texttt{FOLSyn}?\iota, f?\texttt{FOLSyn}?\texttt{o}), \\ \ldots \end{array} \right\}$$

Here we restrict ourselves to a few constant declarations again: The types ι and `o` represent terms and formulas, and the equality operation takes two terms and returns a formula.

Then the theories `Monoid`, `CGroup`, and `Ring` are declared using $f?\texttt{FOLSyn}$ as their meta-theory. For example, the declaration of the theory `CGroup` finally looks like this:

$$e?\texttt{CGroup} \stackrel{f?\texttt{FOLSyn}}{=} \left\{ \begin{array}{l} \texttt{mon} : e?\texttt{Monoid} \stackrel{id_{f?\texttt{FOLSyn}}}{=} \{\}, \\ \texttt{inv} : @(m?\texttt{LF}?\to, f?\texttt{FOLSyn}?\iota, f?\texttt{FOLSyn}?\iota) \end{array} \right\}$$

Here the structure `mon` must have a meta-morphism translating from the meta-theory of `Monoid` to the current theory, and that is simply the identity morphism of $f?\texttt{FOLSyn}$ because `Monoid` and $e?\texttt{CGroup}$ have the same meta-theory. If the meta-theory of `integers` is `ZFC`, then the meta-morphism of `v1` and `v2` is `FOLSem`.

Example 13 shows that the genericity of MMT can make the notation in concrete examples hard to read. In practical user interfaces, users will write ι in external syntax. Moreover, relative URIs (see Section 9.2) or CURIEs [BM09] can be used to shorted URIs.

5.6. Secondary Modules

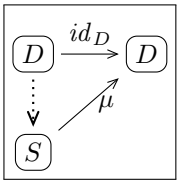
MMT uses only views as secondary modules and does not provide separate primitives for grounded realizations or functors as discussed in Section 2.3. Instead, MMT expresses them in terms of theory morphisms.

Realizations. The central observation underlying the representation of realizations in MMT is that grounded realizations are never absolutely grounded – they are always relative to an implicit global environment. For example, in logic, the realizations are models and thus relative to the underlying language of mathematics, e.g., ZFC set theory. Similarly, in programming languages, the realizations are implementations and thus relative to the underlying implementation language. Using MMT's foundation-independent representation paradigm, this implicit environment is represented as an MMT theory D itself.

Consequently, MMT does not need a new primitive for grounded realizations: The grounded realizations of S relative to the environment D are just the morphisms from S to D, which double as realization expressions. Moreover, MMT naturally provides concrete syntax for both the syntactic and the semantic translations: The syntactic translation is obtained by morphism application, the semantic one by morphism composition.

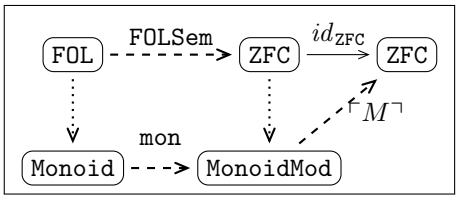
We will consider examples from logic and programming languages. In the former case, D is a theory for ZFC set theory; in the latter case, we use SML and D is a theory for the global environment of SML. To simplify the notation, we will drop the document part of the MMT URIs everywhere, e.g., we will write `Monoid` instead of the technically correct e?`Monoid` (where e is the URI introduced in Example 13).

Definition 14 (Grounded Realizations) An MMT-theory with meta-theory D is called a "D-theory". Then a **grounded realization** of the D-theory S is a morphism μ from S to D that is the identity on D, i.e., that makes the diagram on the right commute.



Example 15 (Realizations in Logic (continued from Example 13)

In the theory graph on the right, the `ZFC`-theory `MonoidMod` arises as the pushout of `Monoid` along `FOLSem` over `FOL` (compare Figure 7). In MMT, this pushout can be expressed easily:



$$\texttt{MonoidMod} \overset{\texttt{ZFC}}{=} \left\{\texttt{mon} : \texttt{Monoid} \overset{\texttt{FOLSem}}{=} \{\}\right\}$$

Thus, `MonoidMod` declares the same local symbols as `Monoid` but translated along `FOLSem`.

Then we can represent models M, i.e., monoids, as grounded realizations $\ulcorner M \urcorner$ of `MonoidMod`. Indeed, a monoid M provides one value for every declaration of `Monoid`, just like an MMT-morphism.

We give a systematic representation of the syntax and semantics of FOL along these lines in [HR11]. Because `FOL` declares the constant ι for the universe, this representation requires a family of views `FOLSem`(U), each of which interprets the universe as the set U. Here the `SFOL`-based formalization of algebra from Example 2 leads to a more compelling formalization of models:

Example 16 (Realizations in `SFOL`) In sorted first-order logic from Example 2, where each theory declares a sort for its domain, the analogue to Example 15 is slightly more intuitive because `SMonoid` and thus also `SMonoidMod` have constant declarations for the universe `univ` as well as for `comp`, `unit`, `assoc`, and `neut`. Thus, a monoid $M = (U, \circ, e)$ is encoded as the view $\ulcorner M \urcorner$ that contains assignments `base` $\mapsto \ulcorner U \urcorner$, `comp` $\mapsto \ulcorner \circ \urcorner$, `unit` $\mapsto \ulcorner e \urcorner$, `assoc` $\mapsto P$, and `neut` $\mapsto Q$. Here $\ulcorner U \urcorner$, $\ulcorner \circ \urcorner$, and $\ulcorner e \urcorner$ are the MMT-terms over `ZFC` that represent the objects U, $\circ$, and e. Moreover, P and Q are the terms representing the necessary proofs that show that M is indeed a monoid.

Example 17 (Realizations in SML) The theory `SML` contains declarations for all primitives of the simple type theory underlying SML, such as `->`, `fn`, and `type`. These constants are untyped and definition-less. `SML` is used as the meta-theory of the theory $D =$ `SMLLib`, which extends `SML` with typed constants for all declarations of the SML basis library [SML97].

Now we represent SML signatures S as `SMLLib`-theories with name $\ulcorner S \urcorner$ and SML structures s realizing S as grounded realizations of $\ulcorner S \urcorner$. For example, consider the simple SML signature S and the structure s realizing it given on the right of Figure 14. Their representations in MMT are given in the commutative theory graph on the left side of the same figure (which duplicates `SMLLib` to be commutative). $\ulcorner S \urcorner$ contains one MMT constant declaration for every declaration in S. These constants have a type according to S but no definiens. The view $\ulcorner s \urcorner$ maps every declaration of $\ulcorner S \urcorner$ to its value given by s.

More generally, SML structures s may also contain declarations that do not correspond to declarations present in S. In that case, an auxiliary theory T with meta-theory `SMLLib` is used that contains one declaration for every declaration in s. Then the view $\ulcorner s \urcorner$ arises as the partial view from T to `SMLLib`.

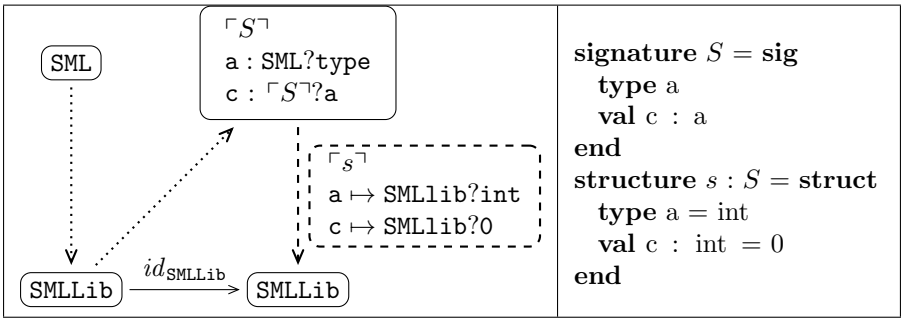


Figure 14: Implementations in SML

Note that the right hand side of Figure 14 shows the external syntax of SML and the left-hand side the internal representation of MMT. The latter makes the URIs and the view $\ulcorner s \urcorner$ explicit that are implicit in the former.

In Example 17, a single meta-theory `SMLLib` is used because both SML signatures and SML structures may use the SML basis library. A common alternative is that the specification and the implementation language are separated into two different languages. We encounter this, for example, in logic where theories are written using only the syntax of the logic whereas models are given in terms of the semantic domain – in our example ZFC set theory.

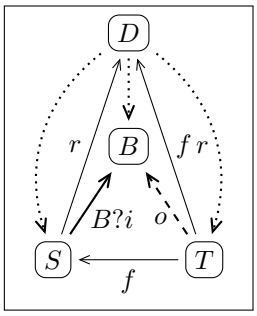
We can strengthen the above representations considerably by using an additional meta-theory: A foundational theory for a logical framework that occurs as the meta-theory of the environment D. For example, we can use `LF` as the meta-theory of `ZFC` and `SML`. Then the constants occurring in `ZFC` and `SML` can be typed using the type theory of LF. In [IR11], we show how to formalize `ZFC` and other foundations of mathematics in LF. A corresponding representation of the semantics of `SML` in LF can be found in [LCH07].

In that case, we can represent not only models of FOL theories and implementations of SML signatures as MMT morphisms; we can also represent the "conforms-to" relation between theory and model as well as between specification and implementation by the well-formedness condition on MMT morphisms. This observation combines the `LF` type system with the property that, as we will see in Section 7, MMT morphisms are guaranteed to preserve typing.

Functors. First-order functors can be represented in MMT similarly. In particular, functors are special cases of views, and functor application is a special case of morphism composition. MMT deliberately does not provide higher-order functors to remain conservative.

Definition 18 (Functors) Given two D-theories S and T, a **functor** from S to T is a morphism from T to S that is the identity on D. Given such a functor f and a grounded realization r of S, the **functor application** is defined as the grounded realization $f\,r$.

It is often convenient to give such a functor as a triple $F = (B, i, o)$ as in the diagram on the right. Here the theory B is the **body** of the functor; it contains a structure declaration $i : S \stackrel{id_D}{=} \{\}$ followed by arbitrary constant declarations all of which have a definiens. The intuition is that B imports the theory S, which represents the **input interface** of the functor, and then implements the intended output interface T. Finally, the view o is the **output interface** of the functor: It determines how T is implemented by B.



Let i^{-1} denote the view from B to S, which inverts i, i.e., it maps every constant induced by the structure i to the corresponding constant of S. Because all local constant declarations of B have a definiens,

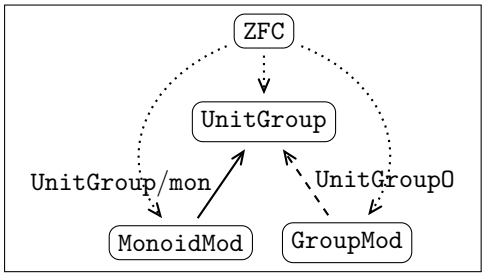
i^{-1} is total. Then we obtain the intended functor as the composition $f = o\, i^{-1}$. Given a grounded realization r of S, functor application is simply the composition $o\, i^{-1}\, r$.

Note that we are flexible whether the intelligence of the functor is given in B or in o. B may contain defined constants for all declarations of T already so that o is just an inclusion. The opposite extreme arises if B contains no declaration besides i and the assignments in o give the body of the functor.

We will now use the (B, i, o)-style of defining functors to expand on the examples for realizations.

Example 19 (Logic (continued from Example 15))

As an example for a functor between models of logical theories, consider the well-known functor from algebra that maps a monoid $M = (U, \circ, e)$ to its group of units (whose universe is the set $\{u \in U | \exists v \in U. u \circ v = v \circ u = e\}$). We represent it as a triple $(\texttt{UnitGroup}, \texttt{mon}, o)$ as in the diagram on the right. We declare



$$\texttt{UnitGroup} \overset{\texttt{ZFC}}{=} \left\{ \texttt{mon} : \texttt{MonoidMod} \overset{id_{\texttt{ZFC}}}{=} \{\} \right\}$$

$$\texttt{UnitGroupO} : \texttt{GroupMod} \to \texttt{UnitGroup} \overset{id_{\texttt{ZFC}}}{=} \{\sigma\}$$

Here σ contains the assignments that realize a group in terms of set theory and an assumed monoid `mon`. For example, σ contains an assignment

$$\texttt{univ} \mapsto @(C, \texttt{UnitGroup?mon/univ}, I)$$

where we assume that C is defined in `ZFC` such that $@(C, s, p)$ represents the set $\{x \in s \,|\, p(x)\}$, and we use I to represent the property of having an inverse element.

Example 20 (SML (continued from Example 17)) An SML functor

functor f(**struct** i : S) : T = **struct** Σ **end**

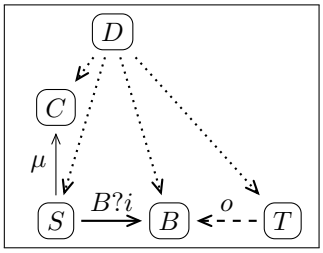
can be represented directly as a triple (B, i, o) where B is the theory

$$B \overset{id_{\texttt{SMLLib}}}{=} \left\{ i : \ulcorner S \urcorner \overset{id_{\texttt{SMLLib}}}{=} \{\}, \ulcorner \Sigma \urcorner \right\}$$

and the view o from $\ulcorner T \urcorner$ to B is an inclusion.

Functors with multiple arguments can be represented by first declaring an auxiliary theory that collects all the arguments of the functor.

Sometimes it is not desirable to use the view i^{-1} because applying i^{-1} to a B-term involves expanding all the definitions of B. In that case, we can use structure assignments to represent functor application as a pushout. Assume we are writing a D-theory C and we already have a morphism μ from S to C, μ is a realization of S available in C. We want to apply the functor (B, i, o) to μ. We can do that by extending C to C' by adding the following structure declaration



$$\texttt{apply} : B \stackrel{id_D}{=} \{i \mapsto \mu\}$$

This makes C' the pushout of μ and $B?i$, and the composed morphism $o\,C'?\texttt{apply}$ is the realization of T that is the result of applying (B, i, o) to μ. Note that these functor applications are generative, not applicative; in particular, we have to add a declaration to C to apply the functor in this way.

6. Well-formed Expressions

In this section we define the well-formed MMT theory graphs (also called valid theory graphs) by means of an inference system. Only the well-formed graphs are meaningful.

We will first define some auxiliary judgments in Section 6.1. Operationally, these judgments have the intuition of lookup judgments, i.e., they retrieve declarations from the environment (in our case: from the theory graph). In non-modular languages where the environment is simply a list of declarations, this lookup can be handled by a single rule. However, in a module system, the lookup involves the computation of all induced declarations. Because this lookup of induced declarations is of central importance and also foundation-independent, the respective judgments are given by a set of self-contained rules in Section 6.1.

Afterwards, we define the main judgments in Section 6.2 and give an inference system for them in Section 6.3 to 6.5. These judgments are parametric in an arbitrary foundation, resulting in the genericity of MMT.

6.1. Induced Declarations

In the following we define the declarations induced by a theory graph. They arise by adding all induced symbols and assignments to those already physically present. This corresponds to the flattening semantics of structures that eliminates structures and transforms MMT theory graphs into flat ones.

The judgments for induced declarations are given in Figure 15. All of them are parametrized by a theory graph γ. The first four judgments are functional in the sense that they take identifiers as input and return declarations as output. In particular, these functions return the parameters in the third column. The mutually recursive definitions of all judgments are given below.

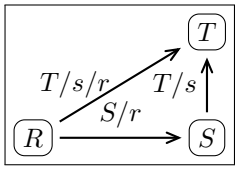
Judgment	Intuition: in theory graph γ ...	Inferred
$\gamma > T = \{\vartheta\}$	T is a theory with body ϑ.	ϑ
$\gamma \gg l : S \to T = B$	l is a link from S to T with definiens B.	S, T, B
$\gamma >_T c : \tau = \delta$	$c : \tau = \delta$ is an induced constant of T.	τ, δ
$\gamma \gg_l c \mapsto \delta$	$c \mapsto \delta$ is an induced constant assignment of l.	δ
$M \hookrightarrow T$	M is the meta-theory of T.	
$\mu \hookrightarrow l$	μ is the meta-morphism of l.	

Figure 15: Judgments for Induced Declarations

Induced Modules. Firstly, the judgments $\gamma > T = \{\vartheta\}$ and $\gamma \gg l : S \to T = B$ define the structure of the MMT theory graph, i.e., the induced module level identifiers. Here B is of the form $\{\sigma\}$ or μ according to whether l is defined by a link body or a morphism. Moreover, we write $M \hookrightarrow T$ and $\mu \hookrightarrow l$ to give the meta-theory and meta-morphism of a theory T or a link l. These judgments are somewhat trivial because they hold iff a meta-theory or meta-morphism is provided explicitly in the syntax of the theory graph.

The first five rules in Figure 16 are straightforward: They simply cover the declaration of a theory, and the two possible ways each to declare a view or a structure. We use square brackets to denote the optional meta-theories or meta-morphisms, and we give the cases for $M \hookrightarrow T$ and $\mu \hookrightarrow l$ as second conclusions of a rule.

The only non-trivial rule is *ind_str*, which covers the case of induced structures: $T/s/r$ identifies the structure induced when a structure declaration s instantiates S and S itself has a structure r. The induced structure is defined to be equal to the composition of the two structures, which formalizes the intended semantics of induced structures.



Induced Symbols. For every theory or link of γ, we define the symbol level identifiers it induces. If $\gamma > T = \{\_\}$, we write $\gamma >_T c : \tau = \delta$ if $c : \tau = \delta$ is an induced constant declaration of T. To avoid case distinctions, we write $\bot$ for τ or δ if they are omitted. If $\gamma \gg l : \_ \to \_ = \_$, we write $\gamma \gg_l c \mapsto \delta$ if $c \mapsto \delta$ is an induced assignment of l.

The induced constants of a theory are defined by the rules in Figure 17. The rule *con* simply handles explicit constant declarations. The remaining rules handle induced constants that arise by translating a declaration $c : \tau = \delta$ along a structure T/s. In all cases, the type of the induced constant is determined by translating τ along T/s. To avoid case distinctions, we assume $\bot^{T/s} = \bot$, i.e., untyped constants induce untyped constants.

But three cases are distinguished to determine the definiens of the induced constant. Firstly, rule *ind_con_def* applies if the constant c already has a definiens $\delta \neq \bot$. Then the induced constant has the translation of δ along T/s as its definiens. Otherwise, there are two further cases depending on the

$$\frac{T \overset{[M]}{=} \{\vartheta\} \ \text{ in } \ \gamma}{\gamma > T = \{\vartheta\} \quad [M \hookrightarrow T]}\ \mathit{thy}$$

$$\frac{l : S \to T = \mu \ \text{ in } \ \gamma}{\gamma \gg l : S \to T = \mu}\ \mathit{viewdef} \qquad \frac{l : S \to T \overset{[\mu]}{=} \{\sigma\} \ \text{ in } \ \gamma}{\gamma \gg l : S \to T = \{\sigma\} \quad [\mu \hookrightarrow l]}\ \mathit{view}$$

$$\frac{\gamma > T = \{\vartheta\} \quad s : S = \mu \ \text{ in } \ \vartheta}{\gamma \gg T/s : S \to T = \mu}\ \mathit{strdef}$$

$$\frac{\gamma > T = \{\vartheta\} \quad s : S \overset{[\mu]}{=} \{\sigma\} \ \text{ in } \ \vartheta}{\gamma \gg T/s : S \to T = \{\sigma\} \quad [\mu \hookrightarrow T/s]}\ \mathit{str}$$

$$\frac{\gamma > T = \{\vartheta\} \quad s : S = \_ \ \text{ in } \ \vartheta \quad \gamma \gg S/r : R \to S = \_}{\gamma \gg T/s/r : R \to T = S/r \ \ T/s}\ \mathit{ind\_str}$$

Figure 16: Induced Modules [and Meta-Theories/Morphisms]

assignment provided by the structure T/s (see the respective rules in Figure 18). If T/s provides an explicit definiens δ, it becomes the definiens of the induced constant (rule *ind_con_ass*). If T/s provides the default assignment $T?s/c$ the induced constant has no definiens (rule *ind_con_dflt*).

The induced assignments of a link l are defined by the rules in Figure 18. The rule *def_link_ass* defines the assignments of a link that is defined as μ: Every definition-less constant is translated along μ.

For links that are defined by a list of assignments, four cases must be distinguished. Firstly, the rule *ass* applies if there is an explicit assignment $c \mapsto \omega$ in l. Secondly, rule *ind_ass* creates induced assignments to s/c, which arise if there is an assignment of a morphism μ to the structure r in a link l. Since r/c identifies the constant c imported along r, the induced assignment arises by translating c along μ.

Finally, it is possible that neither rule *ass* nor rule *ind_ass* applies to c – namely if the body of l contains neither an explicit nor an induced assignment for c. We abbreviate that by "c not covered by σ". In that case the rules *dflt_ass_str* and *dflt_ass_view* define default assignments depending on whether l is a structure or a view. If l is a structure, c is mapped to the induced constant $T?s/c$ in rule *dflt_ass_str*. If l is a view, c is filtered via rule *dflt_ass_view*.

Clash-Freeness. It is easy to prove that if $\gamma >_S c : \_ = \bot$ and $\gamma \gg l : S \to T = \_$, then always $\gamma \gg_l c \mapsto \delta$ for some δ, but δ is not necessarily unique.

$$\frac{\gamma > T = \{\vartheta\} \quad c : \tau = \delta \text{ in } \vartheta}{\gamma >_T c : \tau = \delta}\ con$$

$$\frac{\gamma \gg T/s : S \to T = \_ \quad \gamma >_S c : \tau = \delta \quad \delta \neq \bot}{\gamma >_T s/c : \tau^{T/s} = \delta^{T/s}}\ ind\_con\_def$$

$$\frac{\gamma \gg T/s : S \to T = \_ \quad \gamma >_S c : \tau = \bot \quad \gamma \gg_{T/s} c \mapsto \delta}{\gamma >_T s/c : \tau^{T/s} = \delta}\ ind\_con\_ass$$

$$\frac{\gamma \gg T/s : S \to T = \_ \quad \gamma >_S c : \tau = \bot \quad \gamma \gg_{T/s} c \mapsto T?s/c}{\gamma >_T s/c : \tau^{T/s} = \bot}\ ind\_con\_dflt$$

Figure 17: Induced Constants

$$\frac{\gamma \gg l : S \to T = \mu \quad \gamma >_S c : \_ = \bot}{\gamma \gg_l c \mapsto (S?c)^\mu}\ def\_link\_ass$$

$$\frac{\gamma \gg l : S \to T = \{\sigma\} \quad c \mapsto \omega \text{ in } \sigma}{\gamma \gg_l c \mapsto \omega}\ ass$$

$$\frac{\gamma \gg l : S \to T = \{\sigma\} \quad \begin{array}{c}\gamma \gg S/r : R \to S = \_ \\ \gamma >_S r/c : \_ = \bot\end{array} \quad r \mapsto \mu \text{ in } \sigma}{\gamma \gg_l r/c \mapsto (R?c)^\mu}\ ind\_ass$$

$$\frac{\gamma \gg l : S \to T = \{\sigma\} \quad \gamma >_S c : \_ = \bot \quad \begin{array}{c}l \text{ structure} \\ c \text{ not covered by } \sigma\end{array}}{\gamma \gg_l c \mapsto T?s/c}\ dflt\_ass\_str$$

$$\frac{\gamma \gg l : S \to T = \{\sigma\} \quad \gamma >_S c : \_ = \bot \quad \begin{array}{c}l \text{ view} \\ c \text{ not covered by } \sigma\end{array}}{\gamma \gg_l c \mapsto \top}\ dflt\_ass\_view$$

Figure 18: Induced Assignments

More generally, the judgments for induced declarations do not necessarily define functions from qualified identifiers to induced declarations. For example, a theory graph might declare the same module name twice or a theory might declare the same symbol name twice. To exclude theory graphs with such name clashes, we use the following definition:

Definition 21 A theory graph γ is called *clash-free* if all of the following hold:

- γ contains no two module declarations for the names I and J such that $I = J$ or such that J is of the form I/J' and the body of I contains a declaration for the name J'.
- There is no module in γ whose body contains two declarations for the names I and J such that $I = J$ or J is of the form I/J'.

Here I, J, and J' are any local identifiers.

This definition is a bit complicated because it covers theory graphs and theories that explicitly declare qualified identifiers such as in a constant declaration $s/c : \tau = \delta$. In most languages, such declarations are forbidden. But such declarations are introduced when flattening the theory graph, and we want the flat theory graph to be well-formed as well. It is natural to solve this problem by assuming that the flattening algorithm can always generate fresh names for the induced constants. However, such a non-canonical choice of identifiers prevents interoperability.

Therefore, MMT permits declarations that introduce qualified identifiers. This is in fact quite natural because deep assignments in links introduce assignments to qualified identifiers already. The definition of clash-freeness handles both theories and links uniformly: Theories may not explicitly declare both a structure s and a constant s/c, and links may not provide both an assignment for a structure s and a deep assignment for an induced constant s/c.

More precisely, we have:

Lemma 22. *If a theory graph γ is clash-free, then the judgments of Figure 15 are well-defined functions that infer the parameters given in the third column if the remaining parameters are provided.*

Proof. This follows by a simple induction over the derivations of the elaboration judgments. □

Example 23 (Continued from Example 13) In our running example, we have the theory $\gamma > e?\texttt{CGroup} = \{\ldots\}$ and the structure

$$\gamma \gg e?\texttt{CGroup}/\texttt{mon} : e?\texttt{Monoid} \to e?\texttt{CGroup} = \{\}$$

This structure has the induced assignment

$$\gamma \gg_{e?\texttt{CGroup}/\texttt{mon}} \texttt{comp} \mapsto e?\texttt{CGroup}?\texttt{mon}/\texttt{comp}$$

according to rule *dflt_ass_str*. And we have the induced constant

$$\gamma >_{e?\texttt{CGroup}} \texttt{mon}/\texttt{comp} : @(m?\texttt{LF}?\to, f?\texttt{FOL}?\iota, f?\texttt{FOL}?\iota, f?\texttt{FOL}?\iota)^{e?\texttt{CGroup}/\texttt{mon}} = \bot$$

according to rule *ind_con_dflt*.

Judgment	Intuition	
$\triangleright \gamma$	γ is a well-formed theory graph.	$*$
$\gamma; \Upsilon \triangleright_T \omega$	ω is structurally well-formed over γ, T, and Υ.	
$\gamma; \Upsilon \triangleright_T \omega : \omega'$	ω is well-typed with type ω' over γ, T, and Υ.	$*$
$\gamma; \Upsilon \triangleright_T \omega \equiv \omega'$	ω and ω' are equal over γ, T, and Υ.	$*$
$\gamma \triangleright \mu : S \to T$	μ is a well-typed morphism from S to T.	$*$
$\gamma \triangleright \mu \equiv \mu' : S \to T$	μ and μ' are equal as morphisms from S to T.	$*$

$*$: foundation-dependent

Figure 19: Typing Judgments

The judgments for MMT are given in Figure 19. For the **structural levels**, the inference system uses a single judgment $\triangleright \gamma$ for well-formed theory graphs. For the **object level**, we use judgments for typing and equality of terms and morphisms. Because MMT is generic in the base language, most judgments are relative to a fixed foundation that defines the semantics of the base language. However, we suppress the foundation in the notations.

$\gamma; \Upsilon \triangleright_T \omega : \omega'$ and $\gamma; \Upsilon \triangleright_T \omega \equiv \omega'$ express **typing and equality of terms** in context Υ and theory T. These judgments are not defined generically by MMT; instead, they are defined by the foundation:

Definition 24 A **foundation** is a definition of the judgments $\gamma; \Upsilon \triangleright_T \omega : \omega'$ and $\gamma; \Upsilon \triangleright_T \omega \equiv \omega'$. In order to avoid case distinctions, we require foundations to define these judgments also for the cases where ω or ω' are $\bot$.

As before, we will occasionally write $\bot$ when the optional type or definition of a constant or variable is not present. For that case, it is convenient to extend the equality and typing judgment to $\bot$. We write $\gamma; \Upsilon \triangleright_T \omega : \bot$ to express that ω is a well-formed untyped value, and $\gamma; \Upsilon \triangleright_T \bot : \omega$ to express that ω is a well-formed type, i.e., a term that may occur on the right hand side of :. Moreover, we assume that $\gamma \triangleright_T \bot \equiv \bot$.

In theoretical accounts, foundations can be given, for example, as an inference system or a decision procedure, or via a denotational semantics. In the MMT implementation (see Section 10.1), foundations are realized as oracles that are provided by plugins.

In addition to the two foundational judgments for terms, MMT provides one foundation-independent judgment about terms. $\gamma; \Upsilon \triangleright_T \omega$ expresses that a term is **structurally well-formed**. Intuitively, this is the strongest well-formedness on ω that can be defined foundation-independently. In all three judgments for terms, we omit Υ when it is empty.

Inspecting the rules of MMT shows that MMT only needs the special case of these three judgments where Υ is the empty context. But it is useful to require the general case to permit future extensions of MMT; moreover, for most foundations, the use of an arbitrary context makes the definitions easier anyway.

Contrary to the judgments for terms, all judgments for **typing and equality of morphisms** are defined generically by MMT (relative to the foundation). $\gamma \rhd \mu : S \to T$ expresses that μ is a well-formed morphism from S to T. Similarly, $\gamma \rhd \mu \equiv \mu' : S \to T$ expresses equality. This notation emphasizes the category theoretic intuition of morphisms with domain and codomain. Readers who prefer the type theoretic intuition of realizations as typed objects can use the alternative notation $\gamma \rhd_T \mu : S$ (speak: μ is a well-typed realization of S over T) instead of $\gamma \rhd \mu : S \to T$ as well as the according notation for equality.

6.3. Inference Rules for the Structural Levels

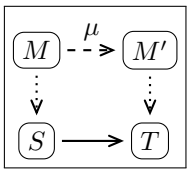
The inference rules define how well-formed theory graphs are extended incrementally. There are three kinds of extensions of a theory graph γ:

- add a module at the end of γ – see the rules in Figure 20,
- add a symbol at the end of the last module of γ (which must be a theory) – see the rules in Figure 21,
- add an assignment to the last link of γ (which may be a view if γ ends in that view, or a structure if γ ends in a theory which ends in that structure) – see the rules in Figure 22.

When theories or links are added, their body is empty initially and populated incrementally by adding symbols and assignments, respectively. This has the effect that there is exactly one inference rule for every theory, view, symbol, or assignment, i.e., for every URI-bearing knowledge item.

We assume for simplicity that all theory graphs are clash-free. It is straightforward to extend all inference rules with additional newness hypotheses for identifiers such that eventually $\rhd \gamma$ implies that γ is clash-free. The reference implementation in Section 10 takes care of this, but we omit this here to simplify the notation.

The rules in Figure 20 define theory graphs as lists of modules. The rule *Start* starts with an empty theory graph, and the rules *Thy* and *View* add modules with empty bodies (that will be filled incrementally). Rule *View* unifies the cases whether S has a meta-theory or not by using square brackets for optional parts; whether T has a meta-theory, is irrelevant. Finally *ViewDef* adds a view defined by a morphism.



In rule *View*, one might intuitively expect the assumption $\left[M \hookrightarrow S \quad M' \hookrightarrow T \quad \gamma \rhd \mu : M \to M'\right]$ which is the situation depicted in the diagram on the right. *View* subsumes that case because μ is also a morphism from M to T due to rule $\mathcal{M}_{covar}$ described in Section 6.4. In addition, *View* covers situations where there is no M'.

The rules in Figure 21 add symbols to theories. There are three cases corresponding to the three kinds of symbols: constants, structures defined by a morphism, and structures defined by a list of assignments. The rule *Con* says that constant declarations $c : \tau = \delta$ can be added if δ has type τ. Recall that this includes the cases where $\tau = \bot$ or $\delta = \bot$.

Note also that $\gamma \rhd_T \delta$ and $\gamma \rhd_T \tau$ are necessary even though we require $\gamma \rhd_T \delta : \tau$. Indeed in most type systems, the latter would entail the former

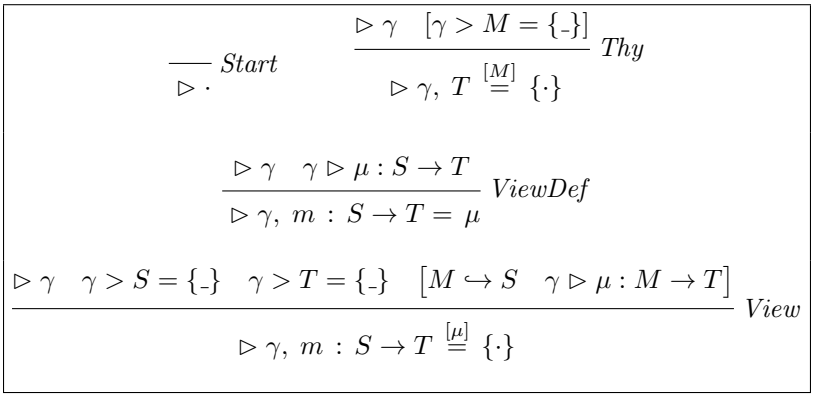


Figure 20: Adding Modules

two, but in MMT the typing judgment is given by the foundation as an oracle, so we cannot be sure.

The rules *Str* and *StrDef* are completely analogous to the rules *View* and *ViewDef* from Figure 20. Again square brackets are used in *Str* to unify the two cases where S has a meta-theory or not. In all three rules it is irrelevant whether T has a meta-theory or not; we indicate that by giving this optional meta-theory in gray boxes.

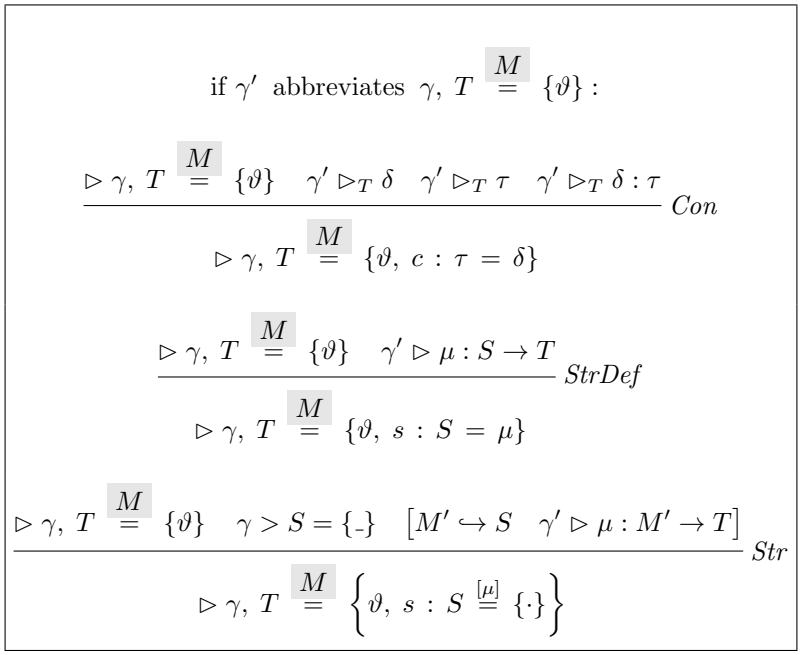


Figure 21: Adding Symbols

The addition of assignments to a link l is more complicated because assignments can be added to views or structures. MMT treats both cases in the same way, which we want to stress by unifying the rules. Therefore, let $\gamma \gg^{\texttt{last}} l : S \to T$ denote that l is a link occurring at the end of γ, i.e., either

- l refers to a view and $\gamma = \ldots,\ l : S \to T \stackrel{[\mu]}{=} \{\sigma\}$ or
- $l = T/s$ refers to a structure and $\gamma = \ldots,\ T \stackrel{[M]}{=} \left\{\ldots,\ s : S \stackrel{[\mu]}{=} \{\sigma\}\right\}$,

and in that case let $\gamma\ +\ Ass$ be the theory graph arising from γ by replacing σ with σ, Ass.

Then the rules in Figure 22 add assignments to a link. *ConAss* adds an assignment $c \mapsto \delta$ for a definition-less constant c of S. Such assignments are well-typed if δ is typed by the translation of the type of c along l. Again we assume $\bot^l := \bot$ to avoid case distinctions. This rule includes the case of $\delta = \top$, i.e., definition-less constants can be filtered by mapping them to $\top$.

The rule *ConFlt* handles the corresponding case where c has a definition. Here no assignment is necessary because it will be induced. Therefore, only the optional filtering must be covered by the rule.

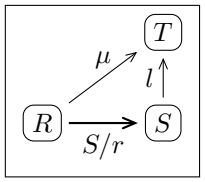
$$\frac{\rhd\gamma \quad \gamma \gg^{\texttt{last}} l : S \to T \quad \gamma >_S c : \tau = \bot \quad \gamma \rhd_T \delta \quad \gamma \rhd_T \delta : \tau^l}{\rhd\gamma\ +\ c \mapsto \delta}\ \mathit{ConAss}$$

$$\frac{\rhd\gamma \quad \gamma \gg^{\texttt{last}} l : S \to T \quad \gamma >_S c : \_ = \delta \quad \delta \neq \bot}{\rhd\gamma\ +\ c \mapsto \top}\ \mathit{ConFlt}$$

$$\frac{\begin{array}{c}\rhd\gamma \quad \gamma \gg^{\texttt{last}} l : S \to T \quad \gamma \gg S/r : R \to S = \_ \quad \gamma \rhd \mu : R \to T \\ [M \hookrightarrow R \quad \mu' \hookrightarrow S/r \quad \gamma \rhd \mu \equiv \mu'\, l : M \to T] \\ \gamma \rhd_T \delta^l \equiv (R?c)^\mu \ \text{ whenever } \ \gamma >_S r/c : \_ = \delta,\ \delta \neq \bot\end{array}}{\rhd\gamma\ +\ r \mapsto \mu}\ \mathit{StrAss}$$

Figure 22: Adding Assignments

The rule *StrAss* is similar to *ConAss* except that adding assignments for structures is a bit more complicated. The first three hypotheses ara analogous to the ones in *ConAss*. The guiding intuition for the remaining hypotheses is that an assignment $r \mapsto \mu$ for a structure r in S should make the diagram on the right commute. From this intuition, we can immediately derive the typing requirements that μ must be a well-typed morphism from R to T.



However, this is not sufficient yet to make the diagram commute. In general, the link l already contains some assignments and possibly a meta-morphism so

that the semantics of the composition $S/r\,l$ is already partially determined. Therefore, μ must agree with $S/r\,l$ whenever the latter is already determined.

This is easy for a possible meta-morphism of $\gamma \rhd \mu' : M \to S$ of S/r. The composition of μ' and l must agree with the restriction of μ to M. Additionally, for all constants r/c of S that have a definiens δ, the translation of δ along l must be equal to the translation of $R?c$ along μ.

The rule *StrAss* is in fact inefficient because it requires to flatten S, i.e., to compute all induced constants r/c of S. However, it is important for scalability to avoid flattening whenever possible. We will come back to this in Section 7.2.

6.4. Inference Rules for Morphisms

Figure 23 gives the typing rules for morphisms. The rule $\mathcal{M}_{link}$ handles links. $\mathcal{M}_{ident}$ and $\mathcal{M}_{comp}$ give identity and composition of morphisms. Meta-theories behave like inclusions with regard to composition of morphisms: The rules $\mathcal{M}_{covar}$ and $\mathcal{M}_{contravar}$ give the usual co- and contravariance rules.

Finally, we define the equality of morphisms in rule $\mathcal{M}_{\equiv}$. We use an extensional equality that identifies two morphisms if they map the same argument to equal terms. This is equivalent to the special case where the morphisms agree for all definition-less constants. If the domain has a meta-theory M, the meta-morphisms must be equal as well. This is checked recursively by requiring $\gamma \rhd \mu \equiv \mu' : M \to T$.

$$\frac{\gamma \gg l : S \to T = \_}{\gamma \rhd l : S \to T}\,\mathcal{M}_{link}$$

$$\frac{\gamma > T = \{\_\}}{\gamma \rhd id_T : T \to T}\,\mathcal{M}_{ident} \qquad \frac{\gamma \rhd \mu : R \to S \quad \gamma \rhd \mu' : S \to T}{\gamma \rhd \mu\,\mu' : R \to T}\,\mathcal{M}_{comp}$$

$$\frac{M \hookrightarrow S \quad \gamma \rhd \mu : S \to T}{\gamma \rhd \mu : M \to T}\,\mathcal{M}_{contravar} \qquad \frac{\gamma \rhd \mu : S \to M \quad M \hookrightarrow T}{\gamma \rhd \mu : S \to T}\,\mathcal{M}_{covar}$$

$$\frac{\begin{array}{c}\gamma \rhd \mu : S \to T \qquad \gamma \rhd \mu' : S \to T \\ \gamma \rhd_T S?c^{\mu} \equiv S?c^{\mu'} \text{ whenever } \gamma >_S c : \_ = \bot \\ [M \hookrightarrow S \qquad \gamma \rhd \mu \equiv \mu' : M \to T]\end{array}}{\gamma \rhd \mu \equiv \mu' : S \to T}\,\mathcal{M}_{\equiv}$$

Figure 23: Morphisms

6.5. Inference Rules for Terms

As noted above, MMT relegates the judgments

$$\gamma; \Upsilon \rhd_T \omega \equiv \omega' \qquad \text{and} \qquad \gamma; \Upsilon \rhd_T \omega : \omega'$$

for typing and equality of terms to the foundation. MMT only defines the judgment $\gamma; \Upsilon \rhd_T \omega$ for **structurally well-formed** terms. Structural well-formedness guarantees in particular that only constants and variables are used that are in scope.

This judgment is axiomatized by the rules in Figure 24. First we define an auxiliary judgment $\gamma \rhd_T \Upsilon$ for well-formed contexts using the rules $\mathcal{T}_\cdot$ and $\mathcal{T}_\Upsilon$. These are such that every variable may occur in the types and definitions of subsequent variables. The rules $\mathcal{T}_x$, $\mathcal{T}_c$, $\mathcal{T}_\top$, $\mathcal{T}_@$, and $\mathcal{T}_{[]}$ are straightforward. $\mathcal{T}_{[]}$ is such that a bound variable may occur in the type or definition of subsequent variables in the same binder.

Finally, $\mathcal{T}_\mu$ and $\mathcal{T}_\hookrightarrow$ formalize the cases relevant for the MMT module system. $\mathcal{T}_\mu$ moves closed terms along morphisms, and $\mathcal{T}_\hookrightarrow$ moves terms along the meta-theory relation. Note that ω^μ is well-formed independent of whether ω is filtered by μ. This is important because the decision whether ω is filtered is expensive if the theory graph has not been flattened yet. Moreover, we permit only closed terms ω in rule $\mathcal{T}_\mu$. This substantially simplifies checking well-formedness (and the more general formulation, in which ω has free variables, is admissible if the involved terms are normalized as defined in Section 7.1).

It is easy to prove a subexpression property for structural well-formedness:

Lemma 25. *If $\gamma; \Upsilon \rhd_T \omega$ then all subexpressions of ω are well-formed in the respective context.*

7. Formal Properties

Now that we have established the grammar and well-formedness conditions for MMT, we can analyze the properties of well-formed theory graphs.

First, in Section 7.1, we introduce a normalization function for terms. Then in Section 7.2, we introduce the notion of regular foundations and establish several important properties of theory graphs that are well-formed relative to a regular foundation. In Section 7.3 we introduce the concept of structural well-formedness, a computationally motivated compromise between MMT-well-formedness and grammatical well-formedness.

Finally, in Section 7.4 and 7.5 we examine the operation of flattening (i.e. eliminating the modular aspects of MMT theory graphs) as a semantics-giving operation of theory graphs. In particular, we show that, in MMT, flattening can be done incrementally, which is important for computational tractability and scalability.

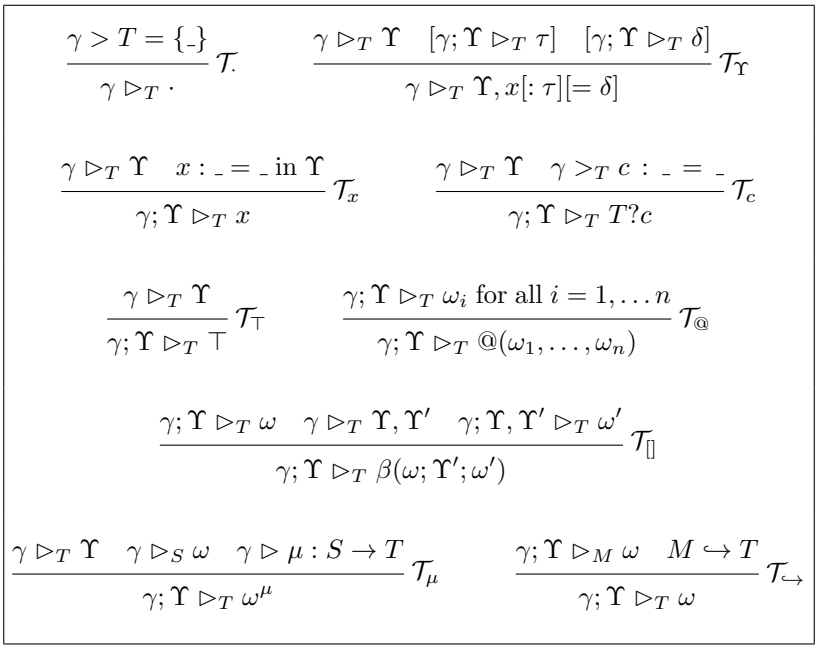


Figure 24: Structurally Well-formed Terms

7.1. Normal Terms

Because MMT is foundation-independent, the equality relation on terms is transparent to MMT. However, some concepts of MMT influence the equality between terms. In particular, the result of a morphism application ω^μ can be computed by homomorphically replacing all constants in ω with their assignments under μ. In the sequel, we define this equality relation to the extent that it is imposed by MMT.

We define a normal form $\overline{\omega}$ that eliminates all morphism applications, expands all definitions, and enforces the strictness of filtering. The latter means that a term with a filtered subterm is also filtered.

Definition 26 Given a theory graph γ and a term ω, the **normal form** $\overline{\omega}^\gamma$ of ω is defined by induction on ω (using sub-inductions for the case of morphism application) as in Figure 25. The same figure defines $\overline{\Upsilon}$, the straightforward extension of normalization to contexts. As before, we assume $\bot^l = \bot$ to avoid case distinctions.

Whenever clear from the context, we suppress γ in the notation and write $\overline{\omega}$ instead of $\overline{\omega}^\gamma$.

Among the cases in Figure 25, the case $(D?c)^l$ is the most interesting because it actually looks into l to apply it to a constant. It distinguishes three subcases:

1. If $D?c$ has a definiens $\delta \neq \bot$, it is expanded before applying l – firstly

$$
\begin{array}{rcl}
\overline{\top} & := & \top \\
\overline{x} & := & x \\
\overline{T?c} & := & \begin{cases} \overline{\delta} & \text{if } \gamma >_T c : \_ = \delta \text{ and } \delta \neq \bot \\ T?c & \text{otherwise} \end{cases} \\
\overline{@(\omega_1, \ldots, \omega_n)} & := & \begin{cases} @(\overline{\omega_1}, \ldots, \overline{\omega_n}) & \text{if } \overline{\omega_i} \neq \top \text{ for all } i \\ \top & \text{otherwise} \end{cases} \\
\overline{\beta(\omega_0; \Upsilon; \omega_1)} & := & \begin{cases} \beta(\overline{\omega_0}; \overline{\Upsilon}; \overline{\omega_1}) & \text{if } \overline{\omega_i} \neq \top \text{ for all } i, \overline{\Upsilon} \neq \top \\ \top & \text{otherwise} \end{cases} \\
\overline{\omega^{id_T}} & := & \overline{\omega} \\
\overline{\omega^{\mu\,\mu'}} & := & \overline{(\omega^{\mu})^{\mu'}} \\
\\
\overline{\top^l} & := & \top \\
\overline{x^l} & := & x \\
\overline{@(\omega_1, \ldots, \omega_n)^l} & := & \overline{@({\omega_1}^l, \ldots, {\omega_n}^l)} \\
\overline{\beta(\omega_0; \Upsilon; \omega_1)^l} & := & \overline{\beta({\omega_0}^l; \Upsilon^l; {\omega_1}^l)} \\
\overline{(\omega^{\mu})^l} & := & \overline{\overline{\omega^{\mu}}^l} \\
\overline{(D?c)^l} & := & \begin{cases} \overline{\delta^l} & \text{if } \delta \neq \bot \\ \overline{(D?c)^{\mu}} & \text{if } \delta = \bot,\ D \neq S,\ \mu \hookrightarrow l \\ \overline{\delta'} & \text{if } \delta = \bot,\ D = S,\ \gamma \gg_l c \mapsto \delta' \end{cases} \\
 & & \text{where } \gamma \gg l : S \to T = \_ \text{ and } \gamma >_D c : \_ = \delta \\
\hline
\overline{\cdot} & := & \cdot \\
\overline{\Upsilon, x : \tau = \delta} & := & \begin{cases} \overline{\Upsilon}, x : \overline{\tau} = \overline{\delta} & \text{if } \overline{\Upsilon} \neq \top,\ \overline{\tau} \neq \top \text{ and } \overline{\delta} \neq \top \\ \top & \text{otherwise} \end{cases}
\end{array}
$$

Figure 25: Normalization

because l should not have to give assignments for defined constants, and secondly because l might filter the name c.

2. Otherwise, we consider the case $D \neq S$ where S is the domain of l. Below we will see that in well-formed theory graphs this is only possible if D is a possibly indirect meta-theory of S. In that case, l must have a meta-morphism $\mu \hookrightarrow l$, which is applied to $D?c$.
3. Otherwise, if $D = S$, then l must provide an assignment $\gamma \gg_l c \mapsto \delta'$.

We use a functional notation $\overline{\omega}$ for the normal form. But technically, as γ is arbitrary, the normal form does not always exist (e.g., if we try to apply a view that is not declared in γ) or does not exist uniquely (e.g., if γ is not clash-free). We will show in Lemma 28 that $\overline{\omega}$ exists uniquely if γ is well-formed, which justifies our notation.

Example 27 (Continued from Example 23)

Consider the term $@(e?\texttt{Monoid}?\texttt{comp}, e?\texttt{Monoid}?\texttt{unit}, e?\texttt{Monoid}?\texttt{unit})$ over the theory `Monoid` and the morphism $e?\texttt{CGroup/mon}$ that moves it from `Monoid` to `Group`. Then we have:

$$\overline{@(e?\texttt{Monoid}?\texttt{comp}, e?\texttt{Monoid}?\texttt{unit}, e?\texttt{Monoid}?\texttt{unit})^{e?\texttt{CGroup/mon}}} = @(\overline{e?\texttt{Monoid}?\texttt{comp}^{e?\texttt{CGroup/mon}}}, \overline{e?\texttt{Monoid}?\texttt{unit}^{e?\texttt{CGroup/mon}}}, \overline{e?\texttt{Monoid}?\texttt{unit}^{e?\texttt{CGroup/mon}}})$$

And using $\gamma \gg_{e?\texttt{CGroup/mon}} \texttt{comp} \mapsto e?\texttt{CGroup}?\texttt{mon/comp}$ and $\gamma \gg_{e?\texttt{CGroup/mon}} \texttt{unit} \mapsto e?\texttt{CGroup}?\texttt{mon/unit}$, we obtain finally

$$@(e?\texttt{CGroup}?\texttt{mon/comp}, e?\texttt{CGroup}?\texttt{mon/unit}, e?\texttt{CGroup}?\texttt{mon/unit})$$

To finish the formal definition of normalization, we must show the well-definedness of normalization:

Lemma 28. *Assume* i) $\rhd\ \gamma$ *for a clash-free* γ *and* ii) $\gamma \rhd_T \omega$. *Then* $\overline{\omega}$ *is well-defined,* $\gamma \rhd_T \overline{\omega}$, *and* $\overline{\omega}$ *is flat.*

Proof. Inspecting the definition of $\overline{\omega}$, we see there is exactly one case for every possible term ω. Technically, this observation uses *i)* to deduce that all lookups occurring during the normalization are well-defined. It also uses *ii)* to conclude that whenever the case $\overline{D?c^l}$ occurs, D is either the domain of l or a possibly indirect meta-theory of it.

Furthermore, a straightforward induction over the structure of ω shows that if the normal form is well-defined, it does not contain morphism applications, i.e., $\overline{\omega}$ is flat.

Therefore, the only thing that must be proved is the well-foundedness of the recursive definition. Firstly, we show by induction on μ that $\overline{\omega^\mu}$ always leads to a term of the form

$$\overline{\overline{\omega^{l_1}}^{\cdot^{\cdot^{l_n}}}}$$

where $l_1, \ldots, l_n$ is the (possibly empty) list of links comprising μ (modulo associativity and identity morphism). Consequently, we can assume without loss of generality that all morphism applications are for a single link l. Then, secondly, the well-foundedness follows by joint induction on γ and ω, i.e., using the lexicographic ordering formed from the size of γ followed by the size of ω. Only some cases warrant closer attention:

- $\overline{T?c}$. This case may increase the size of the involved terms when a constant is replaced with its definiens. But due to the well-formedness of γ, the definiens must be structurally well-formed over a theory graph smaller than γ. (In particular, there are no cyclic dependencies between definitions in well-formed theory graphs.)
- $\overline{S?c\,^l}$. Similar to the previous case, this case may increase the size of the involved terms when $S?c\,^l$ is replaced with the assignment l provides for $S?c$. The same argument applies.

Finally, $\gamma \rhd_T \overline{\omega}$ follows in the same way by joint induction on γ and ω. In particular, whenever a term $T?c$ is replaced with a definiens or a term $S?c^{\,l}$ with the respective assignment, the definiens/assignment is structurally well-formed over a smaller theory graph. □

7.2. Regular Foundations

While the details of the foundation are transparent to MMT, it is useful to impose a regularity condition on foundations that captures some intuitions of typing and equality. First we need an auxiliary definition for the declaration-wise equality of contexts:

Definition 29 For two contexts $\Upsilon^j = \left(x : \tau_1^j = \delta_1^j, \ldots, x : \tau_n^j = \delta_n^j\right)$ for $j = 1, 2$, we write $\gamma; \Upsilon^0 \rhd_T \Upsilon^1 \equiv \Upsilon^2$ iff for all $i = 1, \ldots, n$

$$\gamma; \Upsilon^0, \Upsilon^1_{i-1} \rhd_T \tau_i^1 \equiv \tau_i^2 \qquad \text{and} \qquad \gamma; \Upsilon^0, \Upsilon^1_{i-1} \rhd_T \delta_i^1 \equiv \delta_i^2$$

where $\Upsilon_i^1 := x : \tau_1^1 = \delta_1^1, \ldots, x : \tau_i^1 = \delta_i^1$. Recall that we assume $\gamma \rhd_T \bot \equiv \bot$ to avoid case distinctions.

Definition 30 (Regular Foundation) A foundation is called **regular** if it satisfies the following conditions where γ, T, and all terms are arbitrary:

1. The equality judgment respects normalization:

$$\gamma \rhd_T \omega \equiv \overline{\omega}$$

2. The equality relation induced by $\gamma; \Upsilon \rhd_T \omega \equiv \omega'$ is an equivalence relation for every Υ and satisfies the following congruence laws (where i runs over the respective applicable indices):

$$\gamma; \Upsilon \rhd_T \omega_i \equiv \omega_i' \text{ implies } \gamma; \Upsilon \rhd_T @(\omega_0, \ldots, \omega_n) \equiv @(\omega_0', \ldots, \omega_n')$$

$$\gamma; \Upsilon_0 \rhd_T \omega_i \equiv \omega_i' \text{ and } \gamma; \Upsilon_0 \rhd_T \Upsilon \equiv \Upsilon' \text{ implies } \gamma; \Upsilon_0 \rhd_T \beta(\omega_0; \Upsilon; \omega_1) \equiv \beta(\omega_0'; \Upsilon'; \omega_1')$$

$$\gamma; \Upsilon \rhd_T \omega_i \equiv \omega_i' \text{ and } \gamma; \Upsilon \rhd_T \omega_1 : \omega_2 \text{ implies } \gamma; \Upsilon \rhd_T \omega_1' : \omega_2'$$
$$\gamma \rhd_T \Upsilon \equiv \Upsilon' \text{ and } \gamma; \Upsilon \rhd_T \omega_1 \equiv \omega_2 \text{ implies } \gamma; \Upsilon' \rhd_T \omega_1 \equiv \omega_2$$
$$\gamma \rhd_T \Upsilon \equiv \Upsilon' \text{ and } \gamma; \Upsilon \rhd_T \omega_1 : \omega_2 \text{ implies } \gamma; \Upsilon' \rhd_T \omega_1 : \omega_2$$

   Note that we do not impose a congruence law for morphism application at this point.

3. The foundation preserves typing and equality along flat morphisms. To state this precisely, assume flat theories S and T. Moreover assume a mapping f of constant identifiers to terms such that: Whenever $D = S$ or D is a possibly indirect meta-theory of S and $c : \tau = \delta$ is declared in D, then $\gamma \rhd_T f(D?c) : f(\tau)$ and $\gamma \rhd_T f(D?c) \equiv f(\delta)$.

Then we require that for two flat terms $\gamma \rhd_S \omega_i$

$$\gamma \rhd_S \omega_1 : \omega_2 \text{ implies } \gamma \rhd_T f(\omega_1) : f(\omega_2)$$
$$\gamma \rhd_S \omega_1 \equiv \omega_2 \text{ implies } \gamma \rhd_T f(\omega_1) \equiv f(\omega_2)$$

where $f(\omega)$ arises by replacing every constant $D?c$ in ω with $f(D?c)$.

Most formal languages can be expressed as foundations in this sense, e.g., all pure type systems [Bar92]. We will give detailed examples in Section 8.

Regular foundations are uniquely determined by their action on flat terms so that the module system is transparent to the foundation:

Lemma 31. *For every regular foundation and arbitrary γ, T, ω, ω':*

$$\gamma \rhd_T \omega \equiv \omega' \text{ iff } \gamma \rhd_T \overline{\omega} \equiv \overline{\omega'}$$
$$\gamma \rhd_T \omega : \omega' \text{ iff } \gamma \rhd_T \overline{\omega} : \overline{\omega'}$$

Proof. The first equivalence follows easily using property (1) of Definition 30, symmetry, and transitivity. The second equivalence follows easily using property (1) of Definition 30, symmetry, and the last one of the congruence properties. □

Note that the typing and equality judgments are only assumed for the foundational theories. For all other theories, the typing and equality judgments are inherited from the respective meta-theory. For example, a foundation for `SML` must specify the typing and equality relations of SML expressions. And a foundation for `ZFC` must specify the well-formedness and provability of propositions, both of which we consider as special cases of typing.

It is no coincidence that exactly these two judgments form the interface between Mmt and the foundation: They are closely connected to the syntax of Mmt constant declarations, which may carry types and definitions. If types can be declared for the constants of a language, then the typing relation should be extended to all complex expressions. This is necessary, for example, to check that theory morphisms preserve types. Similarly, if the constants may carry definitions, an equality relation for complex expressions becomes necessary. If foundations are realized as algorithms that check typing and equality, the Mmt constant declarations provide these algorithms with the base cases.

With this bureaucracy out of the way, we can prove some intended properties of morphisms. First we show that morphisms behave as expected. In fact, the presence of filtering makes some of these theorems quite subtle. Therefore, we use the following definition:

Definition 32 A morphism $\gamma \rhd \mu : S \to T$ is **total** if its metamorphism (if there is one) is total and $\overline{S?c\,^{\mu}} \not\equiv \top$ whenever $\gamma >_S c : \_ = \_$. A theory graph is total if all its links are total morphisms.

Note that a morphism that filters only defined constants is still total because the normalization expands definitions. A morphism is not total if it filters

definition-less constants. Partial (i.e., non-total) morphisms often behave badly because they do not preserve truth: Consider a view from S to T that does not provide an assignment for an axiom a, maybe because that axiom is not provable in T at all. Then clearly we cannot expect all theorems of S to be translated to theorems of T. However, this property is also what makes partial morphisms interesting in practice: For example, a partial morphism can be used to represent a translation from a higher-order axiomatization of the real numbers to a first-order one: Such a translation would only translate the first-order-expressible parts, which is still useful in practice. Alternatively, a partial view can be used to conjecture a total one: Such a view typically translates all constants except for those representing axioms.

First, we prove the following intuitively obvious, but technically difficult lemma.

Lemma 33. *If $\gamma \rhd_S \omega$ and $\gamma \rhd \mu : S \to T$, then $\overline{\omega^\mu} = \overline{\overline{\omega}^\mu}$.*

Proof. This is proved by a straightforward but technical induction on the structure of ω^μ. A notable subtlety is that the primary induction is on γ and μ using the statement for arbitrary ω as the induction hypothesis. Then the case where μ is a link uses a sub-induction on ω. We give some example cases where `Def` refers to the definition of normalization and `IH` refers to the induction hypothesis:

- case for a composed morphism in the induction on μ:

$$\overline{\omega^{\mu\,\mu'}} \overset{\texttt{Def}}{=} \overline{(\omega^\mu)^{\mu'}} \overset{\texttt{IH}\,\mu'}{=} \overline{\overline{\omega^\mu}^{\mu'}} \overset{\texttt{IH}\,\mu}{=} \overline{\overline{\overline{\omega}^\mu}^{\mu'}} \overset{\texttt{IH}\,\mu'}{=} \overline{(\overline{\omega}^\mu)^{\mu'}} \overset{\texttt{Def}}{=} \overline{\overline{\omega}^{\mu\,\mu'}}$$

- case for ω^l for a single link l, proved by a sub-induction on ω:
  - case for application if $\overline{\omega_i} \neq \top$ for $i = 1 \ldots, n$:

$$\overline{@(\omega_1, \ldots, \omega_n)^l} \overset{\texttt{Def}}{=} \overline{@({\omega_1}^l, \ldots, {\omega_n}^l)} \overset{\texttt{IH}}{=} \overline{@(\overline{\omega_1}^l, \ldots, \overline{\omega_n}^l)} \overset{\texttt{Def}}{=}$$
$$\overline{@(\overline{\omega_1}, \ldots, \overline{\omega_n})^l} \overset{\texttt{Def}}{=} \overline{\overline{@(\omega_1, \ldots, \omega_n)}^l}$$

    If $\overline{\omega_i} = \top$, the case is trivial.
  - case for a constant $D?c$: If $\gamma >_D c : \_ = \bot$, the statement is trivial because $\overline{D?c} \overset{\texttt{Def}}{=} D?c$. If $\gamma >_D c : \_ = \delta$ for $\delta \neq \bot$, then

$$\overline{D?c^l} \overset{\texttt{Def}}{=} \overline{\delta^l} \overset{\texttt{IH}}{=} \overline{\overline{\delta}^l} \overset{\texttt{Def}}{=} \overline{\overline{D?c}^l}$$

□

Then we have the main technical results about theory graphs and morphisms.

Theorem 34 (Morphisms)**.** *Assume a fixed regular foundation. Then*

1. *For fixed γ, the binary relation on morphisms induced by $\gamma \rhd \mu \equiv \mu' : S \to T$ is an equivalence relation.*

2. *If* $\gamma \rhd \mu_1 \equiv \mu_1' : R \to S$ *and* $\gamma \rhd \mu_2 \equiv \mu_2' : S \to T$ *and* μ_2 *and* μ_2' *are total, then* $\gamma \rhd \mu_1\,\mu_2 \equiv \mu_1'\,\mu_2' : R \to T$.

3. *When morphism composition is well-formed, it is associative and* id_T *is a neutral element.*

4. *The identity morphism and the composition of total morphisms are total. In particular, every well-formed total theory graph induces a category of theories and – modulo equality – morphisms.*

5. *If* $\gamma \rhd_R \omega$ *and* $\gamma \rhd \mu\,\mu' : R \to T$, *then* $\gamma \rhd_T \omega^{\mu\,\mu'} \equiv (\omega^{\mu})^{\mu'}$.

6. *If* $\gamma \rhd_S \omega$ *and* $\gamma \rhd \mu \equiv \mu' : S \to T$, *then* $\gamma \rhd_T \omega^{\mu} \equiv \omega^{\mu'}$.

7. *If* $\gamma \rhd \mu : S \to T$, $\gamma \rhd_S \omega$, $\gamma \rhd_S \omega'$, *then*
   - *if* $\overline{\omega} = \overline{\omega'}$, *then* $\overline{\omega^{\mu}} = \overline{\omega'^{\mu}}$, *and*
   - *if* $\gamma \rhd_S \omega : \omega'$ *and* μ *is total, then* $\gamma \rhd_T \omega^{\mu} : \omega'^{\mu}$.
   - *if* $\gamma \rhd_S \omega \equiv \omega'$ *and* μ *is total, then* $\gamma \rhd_T \omega^{\mu} \equiv \omega'^{\mu}$.

Proof. 1. Reflexivity, symmetry, and transitivity follow immediately from the corresponding properties for terms using rule $\mathcal{M}_{\equiv}$ (see Figure 23).

2. This is proved by induction on the number of meta-theories of R. If there is none, the result follows using rule $\mathcal{M}_{\equiv}$ and applying (5) and twice (7). If $M \hookrightarrow R$, the same argument applies with M instead of R.

3. Because of rule $\mathcal{M}_{\equiv}$, the equality of two morphisms is equivalent to a set of judgments of the form $\gamma \rhd_D c^{\mu} \equiv c^{\mu'}$ for all constants c of S or one of its meta-theories. Because the foundation is regular, every such judgment is equivalent to $\gamma \rhd_T \overline{c^{\mu}} \equiv \overline{c^{\mu'}}$. Then the conclusion follows from the definition of normalization.

4. The totality properties are easy to prove. A category is obtained by taking the theories $\gamma > T = \{\_\}$ as the objects, and the quotient

$$\{\mu \mid \gamma \rhd \mu : S \to T\} \,/\, \{(\mu, \mu') \mid \gamma \rhd \mu \equiv \mu' : S \to T\}$$

as the set of morphisms from S to T. Identity and composition are induced by id_T and $\mu\,\mu'$. (See [Mac98] for the notion of a *category*.)

5. Because the foundation is regular, the conclusion is equivalent to $\gamma \rhd_T \overline{\omega^{\mu\,\mu'}} \equiv \overline{(\omega^{\mu})^{\mu'}}$. And this follows directly from the definition of normalization.

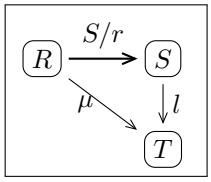
6. Using regularity, it is sufficient to show $\overline{\omega^{\mu}} = \overline{\omega^{\mu'}}$. Using Lemma 33, this reduces to the case where ω is flat. For flat ω, we use induction first on γ and then on ω. All cases for ω except the one for constants follow from the definition of normalization and property (2) in Definition 30. The case for constants with definiens follows by applying the induction hypothesis to the definiens. Finally, the case for definition-less constants follows from $\gamma \rhd \mu \equiv \mu' : S \to T$ due to the premises of Rule $\mathcal{M}_{\equiv}$.

7. The first statement follows immediately from Lemma 33. Using Lemma 31 and 33, the conclusions of the second and third statement reduce to $\gamma \rhd_T \overline{\omega}^\mu : \overline{\omega'}^\mu$ and $\gamma \rhd_T \overline{\omega}^\mu \equiv \overline{\omega'}^\mu$, respectively, i.e., it is sufficient to consider the cases where ω and ω' are flat. And those cases follow using property (3) in Definition 30 and the type-preservation of well-formed total morphisms guaranteed by rule *ConAss*.

□

The restriction that μ must be total in part (7) of Theorem 34 is necessary. To see why, assume $\omega = @(\pi_1, @(\texttt{pair}, a, b))$. A foundation might define $\gamma \rhd_S \omega \equiv a$. Now if μ filters b, then $\overline{\omega^\mu} = \top$ but not necessarily $\overline{a^\mu} = \top$. An even trickier example arises when S contains an axiom $a : @(\texttt{true}, @(\texttt{equal}, \omega, \omega'))$ and the foundation uses a to derive $\gamma \rhd_S \omega \equiv \omega'$. If μ filters a, then it is possible that $\gamma \rhd_T \omega^\mu \equiv \omega'^\mu$ does not hold even when μ filters neither ω nor ω'.

The following theorem establishes a central property of MMT theory graphs that plays a crucial role in adequacy proofs. In the diagram on the right, S is a theory with a structure r instantiating R, and l is a link from S to T that assigns μ to r. The theorem states that the triangle commutes. This means that assignments to structures can be used to represent commutativity conditions on diagrams.



Theorem 35. *Assume $\rhd$ γ relative to a fixed regular foundation. If we have $\gamma \rhd \mu : R \to T$ and $\gamma \gg l : S \to T = \{\sigma\}$ such that σ contains the assignment $r \mapsto \mu$, then*

$$\gamma \rhd S/r\ l \equiv \mu : R \to T.$$

Proof. By rule $\mathcal{M}_\equiv$, we have to show $\gamma \rhd_T R?c^{\,S/r\ l} \equiv R?c^{\,\mu}$ for all constants c with $\gamma >_R c : \_ = \bot$. Using the regularity (and Lemma 33), it is enough to show equality after normalization. A first normalization step reduces the left hand side to $\overline{S?r/c^{\,l}}$. Now there are two cases differing by whether r/c has a definiens in S or not.

- If $\gamma >_S r/c : \_ = \bot$, then $\gamma \gg_l r/c \mapsto R?c^{\,\mu}$, and the left hand side normalizes to $\overline{R?c^{\,\mu}}$.
- If $\gamma >_S r/c : \_ = \delta$ for some $\delta \neq \bot$, a further normalization step reduces the left hand side to $\overline{\delta^l}$. And rule *StrAss* guarantees that in this case $\gamma \rhd_T \delta^l \equiv R?c^{\,\mu}$.

Furthermore, we have to show that the morphisms agree on the meta-theory of R if there is one. This follows due to the premises of rule *StrAss*. □

The extensional definition of the equality of morphisms by rule $\mathcal{M}_\equiv$ is mathematically elegant but operationally inefficient because it requires the full flattening of the domain theory. However, MMT permits avoiding the flattening by using the following theorem:

Theorem 36. *The following rule is admissible:*

$$\frac{\begin{array}{c}\gamma \rhd \mu : S \to T \quad \gamma \rhd \mu' : S \to T \\ \gamma \rhd_T S?c^{\mu} \equiv S?c^{\mu'} \text{ whenever } c : \_ \text{ in } S \\ \gamma \rhd S/s\ \mu \equiv S/s\ \mu' : R \to T \text{ whenever } s : R \doteq \{\_\} \text{ in } S \\ [M \hookrightarrow S \quad \gamma \rhd \mu \equiv \mu' : M \to T]\end{array}}{\gamma \rhd \mu \equiv \mu' : S \to T}\ \mathcal{M}'_{\equiv}$$

Proof. Both $\mathcal{M}_{\equiv}$ and $\mathcal{M}'_{\equiv}$ require $\gamma \rhd_T S?c^{\mu} \equiv S?c^{\mu'}$ for the constants of S. But in $\mathcal{M}_{\equiv}$, this is required for all constants, including the ones induced by structures. $\mathcal{M}'_{\equiv}$, on the other hand, only requires it for the local constants of S, which can be verified without elaboration. For the induced constants of S, rule $\mathcal{M}'_{\equiv}$ recursively checks equality of morphisms for every structure declared in S. By unraveling the recursion, it is easy to see that $\mathcal{M}'_{\equiv}$ eventually checks the same prerequisites as $\mathcal{M}_{\equiv}$. □

A variant of *StrAss* that does not require elaboration can be obtained in a similar way.

Because all nodes and edges in the theory graph have URIs, and morphisms are paths in the theory graph, i.e., lists of URIs, the representation of morphisms is very easy. Then the combination of Theorem 35 and 36 permits very efficient reasoning about the equality of morphisms by reducing it to reasoning about paths in theory graphs. We call this module-level reasoning because it forgets all details about the bodies of theories and links and only uses the theory graph. Naturally module-level reasoning about equality of morphisms is sound but not complete; moreover, the equational theory of paths in the theory graph is not necessarily decidable. However, in our experience, module-level reasoning succeeds in the many practically important cases.

7.3. Structural Well-Formedness

MMT is intended as a basis for mathematical knowledge management systems. For this we need a well-formedness condition that is stronger than that of the context-free grammars used in current representation languages like MATHML or OMDOC as well as weaker than the full semantic validation performed by current symbolic systems.

Context-free validation is simple and widely implemented, but it is very weak and accepts many meaningless expressions. For example, documents containing references to non-existent knowledge items pass context-free validation. For many knowledge management applications, this is too weak. Semantic validation on the other hand accepts only meaningful expressions. It checks a theory graph using a type system or an interpretation function, i.e., it depends on the foundation. Therefore, it is complex, and often only one implementation is available for a specific formal language, which cannot easily be reused by other applications.

Therefore, we introduce structural well-formedness as an intermediate condition. It checks that all references to modules, symbols, or variables exist and

are in scope, and that all morphisms have the right domains and co-domains (i.e., are well-typed). Structural well-formedness is foundation-independent and therefore easy to implement once and for all. We claim that the added strength of full validation is not necessary as a precondition for many web scale algorithms such as browsing, information retrieval, or versioning.

Structural well-formedness of terms has already been defined in Section 6.5. The structural well-formedness of arbitrary theory graphs can be defined using a special foundation:

Definition 37 The *structural foundation* is the foundation where $\gamma \rhd_T \omega : \omega'$ and $\gamma \rhd_T \omega \equiv \omega'$ always hold. A theory graph γ is **structurally well-formed** if it is clash-free and $\rhd \gamma$ holds relative to the structural foundation.

Obviously, the structural foundation is regular.

Clearly, the structural foundation is not a reasonable mathematical foundation, but it is useful because it is maximal or most permissive among all foundations. It is also easy to implement and can be used as a default foundation when the actual foundation is not known or an implementation for it not available.

Structural well-formedness is foundation-independent in the following sense:

Theorem 38. *If a theory graph is well-formed relative to any foundation, then it is structurally well-formed. If γ is structurally well-formed, then $\gamma \rhd_T \omega$ is independent of the foundation.*

Proof. The first statement holds because the use of the structural foundation simply amounts to removing the typing and equality hypotheses in the rules *Con* and *ConAss*. The second statement holds because the rules for the judgment $\gamma \rhd_T \omega$ do not refer to any other judgment. □

7.4. Structural Equivalence

Corresponding to the notions of structural and semantic validation, we can define structural and semantic equivalence of theory graphs:

Definition 39 Relative to a fixed foundation, two well-formed theory graphs γ and γ' are called **structurally equivalent** if the following holds:

- $\gamma > T = \{\_\}$ iff $\gamma' > T = \{\_\}$, and in that case T has meta-theory M in γ iff it does so in γ',
- $\gamma \gg l : S \to T = \_$ iff $\gamma' \gg l : S \to T = \_$,
- $\gamma >_T c : \_ = \_$ iff $\gamma' >_T c : \_ = \_$ whenever $\gamma > T = \{\_\}$.

The intuition behind structural equivalence is that structurally equivalent theory graphs declare the same names: they have the same theories, the same constants, and the same links. It leaves open whether a constant of name s/c is declared or whether a constant c is imported via a structure s. It also leaves open whether a link is a structure or a view.

The value of structural equivalence is that it imposes no requirements on the foundation. Furthermore, structural equivalence is sufficiently strong an

invariant for many applications such as indexing or cross-referencing. This is formalized in the following next theorem.

Theorem 40. *Assume two structurally equivalent theory graphs γ and γ'. Then for all theories S and T of γ:*

- $\gamma \rhd_T \omega$ iff $\gamma' \rhd_T \omega$,
- $\gamma \rhd \mu : S \to T$ iff $\gamma' \rhd \mu : S \to T$.

Proof. This follows by a straightforward induction on the derivations of well-formed terms and morphisms. □

In structurally equivalent theory graphs, the same constant might have different types. Semantic equivalence refines this:

Definition 41 Two structurally equivalent theory graphs γ and γ' are called **semantically equivalent** if the following holds:

- If $\gamma > T = \{\_\}$, $\gamma >_T c : \tau = \delta$, and $\gamma' >_T c : \tau' = \delta'$, then $\overline{\delta}^{\gamma} = \overline{\delta'}^{\gamma'}$ and $\overline{\tau}^{\gamma} = \overline{\tau'}^{\gamma'}$.
- For all $\gamma \gg l : S \to T = \_$, if $\gamma \gg_l c \mapsto \delta$ and $\gamma' \gg_l c \mapsto \delta'$, then $\overline{\delta}^{\gamma} = \overline{\delta'}^{\gamma'}$.

Intuitively, if two theory graphs are semantically equivalent, then they have the same constant declarations and the same assignments. Another way to put it, is that the theory graphs are indiscernable in the following sense:

Theorem 42. *Assume two semantically equivalent theory graphs γ and γ' and a regular foundation. Then for all module declarations Mod:*

$$\rhd \gamma,\ Mod \quad \text{iff} \quad \rhd \gamma',\ Mod.$$

Proof. First of all, due to the structural equivalence, $\gamma,\ Mod$ is clash-free iff $\gamma',\ Mod$ is. Now assume a well-formedness derivation D for $\rhd \gamma,\ Mod$. Let D' arise from D by replacing every occurrence of γ with γ', and replacing the subtree of D deriving $\rhd \gamma$ with some derivation of $\rhd \gamma'$. We claim that every subtree of D' is a well-formedness derivation for its respective root. Then in particular, D' is a well-formedness derivation for $\rhd \gamma',\ Mod$. This is shown by induction on Mod. All induction steps are simple because in most rules the theory graph only occurs as a fixed parameter. Those rules that "look into" the theory graph do so via the judgments given in Section 6.1, and the semantic equivalence of γ and γ' guarantees that these judgments agree up to normalization, and normalization is respected by a regular foundation. □

This provides systems working with MMT theory graphs with an invariant for foundation-independent and semantically indiscernible transformations. Systems maintaining theory graphs can apply such transformations to increase the efficiency of storage or lookup in a way that is transparent to other applications. Moreover, it provides an easily implementable criterion to analyze the impact of a change, which is exploited in [IR12a].

Of course, Definition 41 is just a sufficient criterion for semantic indiscernability. If a foundation adds equalities between terms, then theory graphs that

are distinguished by Definition 41 become equivalent with respect to that foundation. But the strength of Definition 41 and Theorem 42 is that they are foundation-independent. Therefore, it can be implemented easily and generically.

The most important examples of semantical equivalence are reordering and flattening (see Section 7.5).

Theorem 43. *If γ and γ' are well-formed theory graphs that differ only in the order of modules, symbols, or assignments, then they are semantically equivalent.*

Proof. Clear since the elaboration judgments are insensitive to reorderings that preserve well-formedness. □

Note that not all reorderings preserve the well-formedness of theory graphs – there is a partial order on declarations that the linearization in the theory graph must respect. For example, constants must be declared before they are used. Theorem 43 justifies using the following relaxed definition of well-formed theory graphs:

Definition 44 A theory graph γ is called **effectively well-formed** if some reordering of it is well-formed.

This is extremely valuable in practice because it permits applications to forget the order and thus to store theory graphs more efficiently, e.g., in hash tables. It is also relevant for distributed developments where keeping track of the order is often not feasible.

7.5. Flattening

The representation of theory graphs introduced in the last section is geared towards expressing mathematical knowledge in its most general form and with the least redundancy: constants can be shared by inheritance (i.e., via imports), and terms can be moved between theories via morphisms. This style of writing mathematics has been cultivated by the Bourbaki group [Bou68, Bou74] and lends itself well to a systematic development of theories.

However, it also has drawbacks: Items of mathematical knowledge are often not where or in the form in which we expect them, as they have been generalized to a different context. For example, a constant c need not be explicitly represented in a theory T, if it is induced as the image of a constant c' under some import into T.

In this section, we show that for every theory graph there is an equivalent flat one. This involves adding all induced knowledge items to every theory thus making all theories self-contained (but hugely redundant between theories). For a given MMT theory graph γ, we can view the flattening of γ as its semantics because flattening eliminates the specific MMT-representation infrastructure of structures and morphisms and expresses the knowledge purely in the base language (i.e. the MMT module system is conservative).

Theorem 45. *Given a fixed regular foundation, every well-formed theory graph is semantically equivalent to a flat one.*

Proof. Given $\triangleright\ \gamma$, we construct a flat theory graph γ' as follows:

1. Theories:
   - For every T with $\gamma > T = \{\_\}$, there is a theory declaration T in γ'. It has the same meta-theory (if any) in γ' as in γ.
   - For every $\gamma >_T c : \tau = \delta$, the theory T of γ' contains a constant declaration $c : \overline{\tau} = \overline{\delta}$.
2. Links with definiens: For every $\gamma \gg l : S \to T = \mu$, γ' contains a view $l : S \to T = \mu$.
3. Links with assignments:
   - For every $\gamma \gg l : S \to T = \{\_\}$, γ' contains a view from S to T. It has the same meta-morphism (if any) in γ' as in γ.
   - For every $\gamma \gg_l c \mapsto \delta$, the view l of γ' contains a constant assignment $c \mapsto \overline{\delta}$.

It is easy to see that these declarations can be arranged in some way that makes γ' structurally well-formed. Furthermore, it is clear from the construction of γ' that γ' is flat and that γ and γ' are semantically equivalent. The only property that is not obvious is that γ' is well-formed. For that, we must show in particular that all assignments in all views in γ' satisfy the typing assumption of rule *ConAss*. This follows from the construction of γ and property (1) of regular foundations (which is the only property of regular foundations needed for this proof). □

Example 46 (Continued from Example 1) The flattening of the theory graph of our running example contains the module declarations in Figure 26, where we omit all types for simplicity

Two features of MMT are not eliminated in the flattening: meta-theories and filtering.

Regarding meta-theories, the definitions and results in this section could be easily extended to elaborate meta-theories as well. For example, a meta-theory M can be reduced to a structure that instantiates M and has some reserved name. In fact, that is what we did in an earlier version of MMT [Rab08a]. However, this is not desirable because both humans and machines can use meta-theories to relate MMT theories to their semantics. In particular, constants of the meta-theory are often treated differently than the others; for example, their semantics might be hard-coded in an implementation.

Regarding filtering, the situation is more complicated. Imagine a constant declaration $c : \tau = \top$ in the flat theory graph. This is particularly intuitive if we think of c as a theorem stating τ. Then $c : \tau = \top$ means that the theorem holds but its proof is filtered because it relies on a filtered assumption.

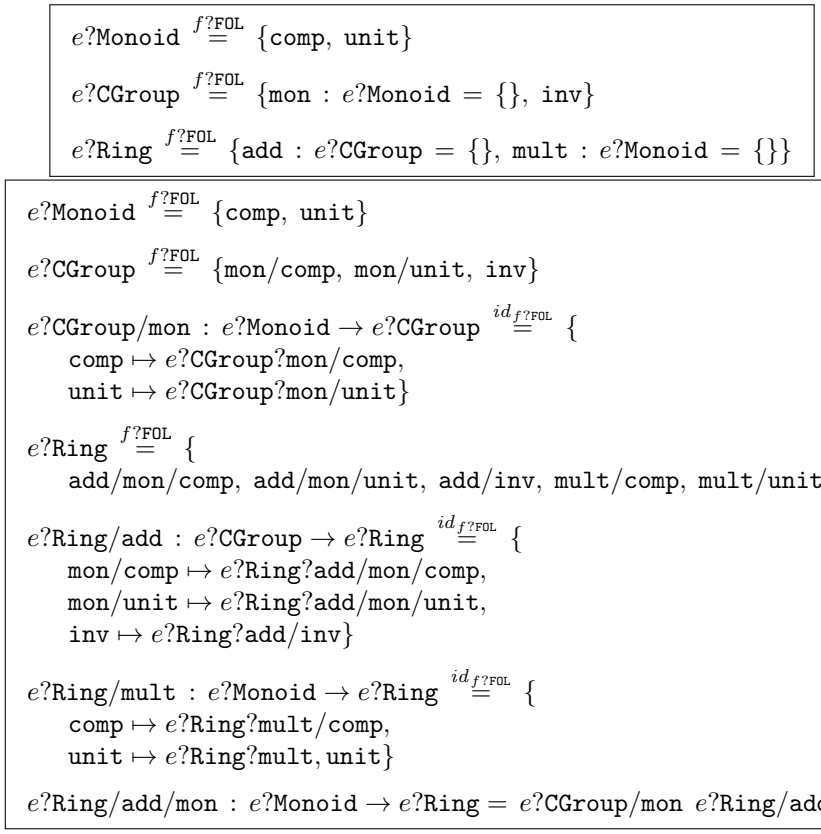


Figure 26: Declarations of the running example (top) and their flattening (bottom)

It is now a foundational question how to handle this case. One possibility is to delete the declaration of c. This is especially appealing from a type/proof theoretical perspective where constant declarations are what defines the existence of objects and their meaning. This community might argue that if the proof is filtered, then the theorem is useless because it can never be applied or verified. Consequently, it can just as well be removed. Another possibility is to replace c with the declaration $c : \tau$, i.e., to turn it into an axiom. This is appealing from a set/model theoretical perspective where constant declarations merely introduce names for objects that exist in the models. This community might argue that it is irrelevant whether the proof is filtered or not as long as we know that there is one.

In order to stay neutral to this foundational issues, we do not elaborate filtering. Instead, we leave all filtered declarations in the flattened signature and leave it to the foundation to decide whether they are used or not.

The most important practical aspect of the flattening in MMT is not its existence: The property that flattening can be applied incrementally is significantly more difficult to achieve than the flattening itself. Consider a theory graph

$$\gamma_0,\ T \overset{M}{=} \{\vartheta_0,\ s : S = \{\sigma\},\ \vartheta_1\},\ \gamma_1.$$

We would like to flatten only the structure T/s. Then the structure can be replaced with a translated copy of the body of S.

For example $c : \tau = \bot$ is translated to $s/c : \tau' = \bot$, where τ' is the translation of τ. In particular, in τ' all names referring to constant of S must be prefixed with s. If s has an assignment $c \mapsto \delta'$, then the declaration is translated to $s/c : \tau' = \delta'$.

We obtain incrementality if structure declarations in S are not flattened recursively. This is possible in MMT. For example, for a structure $r : R$ in the body of S, a structure $s/r : R = S/r\,T/s$ can be added to T rather than adding all induced constants $T?s/r/c$. Individual assignments to structures can be flattened similarly.

8. Specific Foundations

To define a specific foundation (recall Definition 24), we need to define the judgments $\gamma; \Upsilon \rhd_T \omega \equiv \omega'$ and $\gamma; \Upsilon \rhd_T \omega : \omega'$.

For a fixed theory graph, the **meta-relation** between theory identifiers is the transitive closure $<$ of the relation "X has meta-theory Y". We use the $\leq$ notation accordingly for the reflexive-transitive closure. Then the foundational theories are the $<$-maximal ones; and for every other theory T, there is a unique foundational theory M with $T < M$, which we call **the foundational theory** of T. A specific foundation is typically coupled with a certain foundational theory M and only defines $\gamma \rhd_T \omega \equiv \omega'$ and $\gamma \rhd_T \omega : \omega'$ using a definition that is parametric in an arbitrary theory $T < M$.

Foundational theories and foundations can be given for a wide variety of formal languages. As examples, we give them for three very different languages: OPENMATH, LF, and ZFC.

8.1. OpenMath

OPENMATH [BCC+04] is used for the communication of mathematical objects over the internet. OPENMATH content dictionaries can be viewed to correspond to MMT theories, so that MMT yields a module system for OPENMATH content dictionaries. Pure OPENMATH is an untyped language, in which α-conversion of bound variables is the only non-trivial equality relation. Clearly, this foundation is very easy to implement.

The foundational theory for OPENMATH is empty because OPENMATH does not use any predefined constant names. Thus the standard content dictionaries can be introduced as MMT theories with that meta-theory. We can define a foundation for OPENMATH as follows.

Firstly, $\gamma \rhd_T \omega : \omega'$ holds iff one of the following holds:

1. $\omega = \bot$ and $\omega' = \bot$,
2. $\gamma \rhd_T \omega'$ and $\omega = \bot$.

To understand why this characterizes OpenMath, consider how it affects the rule *Con*. According to rule *Con*, a constant declaration $c : \tau = \delta$ is only well-formed if $\gamma \rhd_T \delta : \tau$. Thus, each of the above cases leads to one kind of constant declaration: The first case is used to declare an OpenMath symbol. These are untyped and definition-less. The second case is used to declare an OpenMath formal mathematical property. Here ω' is the asserted property (encoded as an Mmt type corresponding to the Curry-Howard correspondence). In both cases, there is no definiens because OpenMath does not consider definitions.

Secondly, $\gamma \rhd_T \omega \equiv \omega'$ is the smallest relation on structurally well-formed terms that

- is reflexive,
- is closed under substitution of equals,
- is closed under α-renaming,
- respects normalization, i.e, $\gamma \rhd_T \omega \equiv \overline{\omega}$.

Theorem 47. *The foundation for* OpenMath *is regular.*

Proof. All properties can be verified directly. □

8.2. The Edinburgh Logical Framework (LF)

LF [HHP93] is a logical framework based on dependent type theory. Being a logical framework, it represents both logics and theories as LF signatures. Mmt subsumes this approach by also representing LF as a (foundational) Mmt theory. As for OpenMath, Mmt yields a module system for LF.

The foundational theory for LF is given by:

$$\texttt{LF} = \{\texttt{type}, \texttt{kind}, \texttt{lambda}, \texttt{Pi}\}.$$

`type` is the kind of all types. `kind` is the universe of kinds; it does not occur in concrete syntax for LF, but is needed as the Mmt type of all well-formed LF kinds. `Pi` is the dependent type constructor, and `lambda` its introductory form. The application of Mmt can be used as the eliminatory form of `Pi`.

If $T = \texttt{LF}$, only the typing judgment $\gamma \rhd_{\texttt{LF}} \bot : \bot$ holds. This is needed to make the untyped constants in the theory `LF` well-formed (see rule *ConAss*). $\gamma \rhd_{\texttt{LF}} \omega \equiv \omega'$ holds iff $\overline{\omega} = \overline{\omega'}$.

Otherwise, typing and equality are defined according to the LF type theory:

- For constants, $\gamma \rhd_T D?c : \tau$ holds if $T < \texttt{LF}$ and $\gamma >_D c : \tau = \_$.
- For other terms, $\gamma \rhd_T \omega : \omega'$ holds if $\overline{\omega}$ is a well-formed LF-term of type $\overline{\omega'}$ or a well-formed LF-type family of kind $\overline{\omega'}$ in theory T. The details are as in [HHP93] except that the rule for constants is not needed.
- $\gamma \rhd_T \bot : \omega$ holds if $\overline{\omega}$ is a well-formed LF-type or a well-formed LF-kind in theory T. This permits declarations of typed or kinded constants.

Similarly, the judgment $\gamma \rhd_T \omega \equiv \omega'$ is defined by the rules given in the properties (1) and (2) of Definition 30 and the equality rules for LF given in [HHP93].

Theorem 48. *The foundation for LF is regular.*

Proof. The properties (1) and (2) are built into the definition. Property (3) follows from the results in [HST94] after observing that every type-preserving mapping from S to T yields an LF signature morphisms from $\overline{S}$ to $\overline{T}$. Here $\overline{S}$ denotes the union of the bodies of all theories D with $T \leq D < \texttt{LF}$. □

Regular foundations for any pure type system (see, e.g., [Bar92]) and other type theories can be given in the same way.

8.3. Set Theory (ZFC)

In Example 6, we have already sketched how ZFC can be represented as an MMT theory with meta-theory `FOLd`. This in turn can be represented using `LF` as the foundational theory. The details of this representation can be found in [IR11]. Consequently, no separate foundation for ZFC is needed, and the semantics of ZFC-theories can be inherited from the foundation for LF.

However, it is instructive to give a direct foundation for ZFC in addition. There are many different ways to do that, corresponding to the different formulations of axiomatic set theory. In any case, the foundational theory `ZFC` for ZFC is big, declaring constants for all concepts that are required in a minimal axiomatic set theory. Therefore, in practice, this theory will be highly modular.

A typical definition of `ZFC` contains

- constants `set` and `prop` for the basic concepts of sets and propositions,
- constants `empty`, `powerset`, `union`, `comprehension`, etc. for the primitives occurring in the axioms,
- constants `forall`, `conjunction`, etc. to construct first-order formulas,
- one constant for each axiom of ZFC, where the asserted formulas is given as the type.

The typing judgment $\gamma \rhd_T \omega : \omega'$ holds if

- $\omega' = \texttt{ZFC?set}$ and ω is a well-formed set expression,
- $\omega' = \texttt{ZFC?prop}$ and ω is a well-formed formula, or
- ω' is a well-formed set expression and ω is an element of ω'.

In particular, $\omega \neq \bot$ so that only defined constants can be added in `ZFC`-theories.

The equality judgment $\gamma \rhd_T \omega \equiv \omega'$ holds whenever ω and ω' are provably equal sets in the sense of ZFC and in all cases required by regularity.

It is straightforward to complete this definition in a way that makes the resulting foundation regular.

9. Web-Scalability

MMT documents are systematically transparent to the semantics. Therefore, we have deliberately ignored them so far and introduce them only now. They play a central role for web-scalability because they permit the packaging and distribution of theory graphs and the global sharing of knowledge items. We will discuss them in Section 9.1.

Besides documents, the basis for web-scalability is web standards-compliance. We have developed a concrete XML syntax for MMT that serves as the basis for our implementations. It builds on the OMDOC format [Koh06], which already

integrates some of the primitive notions of MMT including the OPENMATH and MATHML 3 syntax for terms (interpreted as OPENMATH objects). The XML encoding is a straightforward compositional mapping; we omit it here and direct the reader to [RK12] for details.

Less trivial is the definition of a URI-based representation for all identifiers, which we discuss in Section 9.2. This is already implicit in the MMT grammar, but we will add relative URI references that are indispensable for scalability.

9.1. Documents and Libraries

Recall that our syntax uses two-partite module identifiers $g?I$. g is a URI that identifies a package, called **document** in MMT. We use the syntax

$$Doc ::= g = \{\gamma\}$$

to declare a document g containing the theory graph γ. A document is called **primary** if all modules declared within γ have module identifiers of the form $g?I$. Non-primary documents arise when documents are aggregated dynamically using fragments from different documents, i.e., as the result of a search query; we call those **virtual** documents in [KRZ10].

Within MMT documents, we define two relaxations of the MMT syntax that are important for scalability and that can be easily elaborated into the official syntax: relative identifiers and remote references. **Relative identifiers** and their resolution into absolute ones are defined in Section 9.2. We speak of **remote references** if a document refers to a module that is declared in some other document. Technically, according to the rules of MMT, such a non-self-contained theory graph would be invalid. Therefore, we make documents with remote references self-contained by adding all referenced remote modules in some valid order at the beginning. This is always possible if there is no cyclic dependency between documents.

The semantics of remote references is well-defined because MMT identifiers are URIs and thus globally unique. However, they are not necessarily URLs and thus do not necessarily indicate physical locations from which the remote module could be retrieved. Therefore, we make use of a **catalog** that translates MMT URIs into URLs, which give the physical locations. This way applications are free to retrieve content from a variety of backends, such as file systems, databases, or local working copies, in a way that is transparent to the MMT semantics.

We call a collection of documents together with a catalog an MMT **library**. A library's document collection can be anything from a self-contained document to (the MMT-relevant subset of) the whole internet. The central component is the catalog that defines the meaning of identifiers in terms of physical locations. Adding a document to a library may include the upload of a physical document, but may also simply consist in adding some catalog entries.

Well-formedness of libraries is checked incrementally by checking individual documents when they are added. A document $g = \{\gamma\}$ is **well-formed** relative to a library L if the following hold:

- If a module identifier declared in γ already exists in L, then the two modules must be identical.
- γ_L is a well-formed theory graph where γ_L arises by prepending all remotely referenced modules according to their resolution in L.

It is easy to prove that if we only ever add well-formed documents to an initially empty library, all modules in the library can be arranged into a single well-formed theory graph. This can be realized, for example, by implementing L as a database that rejects the commit of ill-formed content (see Section 10). Thus, libraries provide a safe and scalable way of building large theory graphs.

9.2. URI-based Addressing

As defined in the MMT grammar, an **absolute identifier** of an MMT knowledge item is a document URI G, a module identifier $G?M$, or a symbol identifier $G?M?S$. It is convenient to unify these three cases by assuming $M = \varepsilon$ and/or $S = \varepsilon$ if the respective component is not present. Then absolute references are always triples (G, M, S).

Similarly, a **relative identifier** is a triple (g, m, s). g is a relative document reference, i.e., a URI reference as defined in RFC 3986 [BLFM05] but without query or fragment. Note that this includes the case $g = \varepsilon$. m and s are usually of the form I, i.e., slash-separated (possibly empty) sequences of non-empty names. For completeness, we mention that MMT also permits m and s to be relative: If $g = \varepsilon$, m may be of the form $/I$, which is a module reference that is interpreted relative to the current module; and if $g = m = \varepsilon$, s may also be of the form $/I$, which is a symbol reference that is interpreted relative to the current symbol.

In particular triples (g, T, c) correspond to the $(cdbase, cd, name)$ triples of the OPENMATH standard [BCC+04]. This triple-based addressing model takes up an idea (called "reference by context") from OMDOC 1.1 that was dropped in OMDOC 1.2 because its semantics could not be rigorously defined without the MMT concepts.

We use $g?m?s$ as the URI encoding of an identifier (g, m, s) – even if g, m, or s are empty – and adopt the convention that trailing but not leading ? characters can be dropped. For example, we encode

- (g, m, ε) as $g?m$,
- (ε, m, s) as $?m?s$,
- $(\varepsilon, \varepsilon, s)$ as $??s$,

This encoding can be parsed back uniquely into triples.

Definition 49 (Relative URI Resolution) The **resolution** of a relative identifier $R = (g, m, s)$ is defined relative to an absolute identifier $B = (G, M, S)$, which serves as the base of the resolution. The result is the following absolute

identifier:

$$resolve(B,R) := \begin{cases} (G+g,m,s) & \text{if } g \neq \varepsilon \\ (G,M+m,s) & \text{if } g = \varepsilon, m \neq \varepsilon \\ (G,M,S+s) & \text{if } g = m = \varepsilon, s \neq \varepsilon \\ (G,M,S) & \text{if } g = m = s = \varepsilon \end{cases}$$

where $G+g$ denotes the resolution of the URI reference g relative to the URI G as defined in RFC 3986 [BLFM05]. Furthermore, $M+m$ resolves m relative to M: If $m = /m'$, then $M+m$ arises by appending m' to M; otherwise $M+m=m$. $S+s$ is defined accordingly.

The above definition yields the ill-formed result (G,ε,s) when resolving a symbol level reference $R=(\varepsilon,\varepsilon,s)$ against a document level base $B=(G,\varepsilon,\varepsilon)$. We forbid that pathological case, which would correspond to a symbol being declared outside a theory.

To resolve relative identifiers within a document, we need the following:

Definition 50 (Base URI) Let γ a theory graph, then we define a **base URI** for MMT expressions occurring in γ:

1. The base of a module declaration or a remote reference to a module is the URI of the containing document.
2. The base of symbol declaration is the URI of the containing theory.
3. The base of an assignment to a symbol is the URI of the codomain theory.
4. If μ is a morphism with domain S, then the bases of ω in ω^μ, and μ' in $\mu'\,\mu$, are S.
5. In all other cases, the base of an expression is the base of the parent node in the syntax tree.

MMT URIs are the main MMT-related data structure needed for cross-application scalability, and our experience shows that they must be implemented by almost every peripheral system, even those that do not implement MMT itself. Already at this point, we had to implement them in SML [RS09], JavaScript [GLR09], XQuery [ZKR10], Haskell (for Hets, [MML07]), and Bean Shell (for a jEdit plugin) [IR12b] – in addition to the Scala-based reference API presented in Section 10.1.

This was only possible because MMT-URIs constitute a well-balanced trade-off between mathematical rigor, feasibility, and URI-compatibility: In particular, due to the use of the two separators / and ? (rather than only one), they can be parsed locally, i.e., without access to or understanding of the surrounding MMT document.

10. Implementations

The design of the MMT language has been driven by a tight feedback loop between theoretical analysis of knowledge structures and practical implementation efforts. In particular, we have evaluated the space of possible module systems along the classifications developed in Section 2 in terms of expressivity, computational tractability, and scalability in a variety of case studies. These efforts led to three implementations, which we will present here. All of them are open source, and can be obtained from the respective home pages.

10.1. The MMT Reference API

The MMT implementation provides a Scala-based [OSV07] (and thus fully Java-compatible) open-source implementation for the MMT data structures.[3] The central part of the API acts as an MMT library with atomic add and retrieve methods for knowledge items identified by MMT URIs. This is supplemented by persistent storage that maintains MMT theory graphs as XML documents in file systems or databases.

Adding Knowledge Items. We distinguish three levels of **validation** with varying strictness. Firstly, context-free XML validation is quick but cannot guarantee MMT-well-formedness. Secondly, structural validation guarantees the structural well-formedness according to Definition 37. Thirdly, semantic validation refines structural validation by additionally validating foundation-specific typing and equality constraints by using plugins for individual foundations (see below).

Structural validation decomposes a theory graph into a sequence of atomic declarations that are validated and added incrementally. Every constant declaration or assignment is an atomic declaration. Declarations of documents, theories, views, and structures are atomic if the body is empty. For example, a view is decomposed into the declaration of an empty view and one declaration for each constant assignment.

The MMT inference system is designed such that this incremental addition is possible. In particular, later atomic declarations can never invalidate earlier ones.

Retrieving Knowledge Items. To retrieve knowledge items from a library, we use the **atomic queries** given in Figure 27. These take an MMT URI and return the induced MMT declaration as defined in Section 6.1.

Atomic queries permit not only the retrieval of all declarations of the original documents in the library, but also of all induced declarations. Note that there are two ways to combine a structure URI $g?T/s$ and a constant c: The query $g?T/s?c$ retrieves an assignment provided by s for c (defaulting to $c \mapsto g?T?s/c$ if there is none); and the query $g?T?s/c$ retrieves the induced constant declaration of T.

[3] See https://trac.kwarc.info/MMT.

URI	Definedness condition	Result
g	$g = \{\gamma\}$ in L	$g = \{\gamma\}$
T	$\gamma_L > T = \{\vartheta\}, \quad [M \hookrightarrow T]$	$T \overset{[M]}{=} \{\vartheta\}$
l	$\gamma_L \gg l : S \to T = \{\sigma\}, [\mu \hookrightarrow l]$	$l : S \to T \overset{[\mu]}{=} \{\sigma\}$
l	$\gamma_L \gg l : S \to T = \mu$	$l : S \to T = \mu$
$T?c$	$\gamma_L >_T c : \tau = \delta$	$c : \tau = \delta$
$l?c$	$\gamma_L \gg_l c \mapsto \delta$	$c \mapsto \delta$

Figure 27: Atomic Queries

Example 51 (Atomic Queries) Examples for atomic queries were already indicated in Example 23.

atomic query	returns:	comment
e?CGroup/mon?comp	mon/comp $\mapsto$ e?CGroup?mon/comp	the default assignment for lack of an explicit assignment
e?CGroup?mon/comp	mon/comp : $\tau^{e?\texttt{CGroup/mon}} = \bot$	where τ is as in Example 23

Atomic queries are relatively easy to implement and provide a sufficient interface for higher knowledge management layers to implement many additional services. For example, we can use them to implement **local validation**. Given a library L and an implementation of atomic queries, we can validate documents and document fragments relative to L without having to read all of L. Instead, the respective atomic query is sent to L whenever a reference to an unknown knowledge item is encountered.

Atomic queries also yield an easy implementation of **flattening** because they already return all declarations of induced constants and assignments that occur in the flattened theory graph. Therefore, to implement flattening, we only have to know the URIs of the induced declarations. This information can be cached by the library, and applications can aggregate the flattened theory graph without having to implement MMT.

Indexing Knowledge Items. As a by-product of structural validation, a **relational representation** of the validated document is generated, which corresponds to an ABox in the MMT ontology (from Figure 4). The individuals of this ontology are the MMT URIs of induced declarations. Unary predicates give the type of each MMT URI (i.e., theory, view, constant, structure, assignment to constant, assignment to structure). And the binary predicates between MMT URIs include for example "Constant c_1 occurs in the type of constant c_2." or "View v has domain S.". This information is cached, and the implementation supports the compositional **query language** we presented in [Rab12b]. The combination of atomic and relational queries is a simple but powerful interface to MMT libraries.

The ABox also includes the dependency relation between declarations, which is essential for change management. The MMT implementation supports the **change management** solution we presented in [IR12a].

Interfaces. The MMT API includes various back and front ends. The back ends implement a catalog that translates MMT URIs into physical locations. The back ends retrieve documents from remote MMT libraries (such as another instance of the MMT API or an SVN repository) via HTTP or from local file systems. This catalog is fully transparent to the data structures and the applications built on top.

The front ends provide users and systems access to the library. The current implementation includes a **shell** and a **web server** front end. The shell is scriptable and can be used to explicitly retrieve, validate, and query MMT documents. The interaction with the web server proceeds like with the shell except that input and output are passed via HTTP. In particular, the web server can easily be run as a local proxy that provides MMT functionality and file system abstraction to local applications. An example instance of the web server is serving the content of the TNTBASE repository of the LATIN project [KMR09].

Foundations as Plugins. The foundation-specific validation algorithm provides a plugin interface for foundations. Currently, one such plugin exists for the foundation for LF from Section 8.

Every plugin must identify a theory M, and implement functions that decide (or attempt to prove) instances of the typing and equality judgments for any theory $T < M$. Foundations should be regular, and due to Lemma 31, they only need to consider flat instances of these judgments. Moreover, due to Theorem 45, they can assume that T is flat. Thus, existing algorithms for the formal system represented by the foundational theory M can easily be reused to obtain plugins for M.

Notations. Going beyond the scope of this paper, the MMT language also provides a **notation language** inspired by [KMR08] with a simple declarative syntax to define renderings of MMT content in arbitrary human- or machine-oriented formats. This can produce, e.g., XHTML+presentation MathML that is **interactively browsable** by integrating the MMT-aware JOBAD JavaScript library [GLR09]. Alternatively, it can produce other system's concrete input syntax so that MMT can serve as an interchange language. Notations can be grouped into *styles*, which are themselves subject to the MMT module system.

10.2. TNTbase – a Scalable MMT*-Compliant Database*

The TNTBASE system [ZK09] is an open-source versioned XML **database** developed at Jacobs University.[4] It was obtained by integrating Berkeley DB XML [Ora10] into the Subversion Server [Apa00], is intended as a basis for the collaborative editing and sharing of XML-based documents, and integrates versioning and access of document fragments. We have extended TNTBASE with an MMT **plugin** that makes it MMT-aware [KRZ10, ZKR10].

[4] See http://tntbase.org/.

The most important aspect of this plugin is validation-upon-commit. Using the `tntbase:validate` property, folders and files can be configured to require validation. Thus, users can choose between no, XML-based, structural, or foundational validation of MMT files. Since the commit of ill-formed files can be rejected, TNTBASE can guarantee that it only contains well-formed documents. Thus, other systems can use MMT-enriched TNTBASE for the long term storage of their system libraries and can trust in the correctness of documents retrieved from the database.

Moreover, the plugin computes the relational representation of a committed well-formed document, which TNTBASE stores as an XML file along with every MMT file. TNTBASE exposes the relational representation of MMT documents via an XQuery interface, and we have implemented a variety of custom queries in an XQuery module that is integrated into TNTBASE. TNTBASE indexes the files containing the relational representation so that such queries scale very well. For example, our XQuery module includes a function that computes the transitive closure of the structural dependency relation between MMT modules and dynamically generates a self-contained MMT document that includes all dependencies of a given module. Even for small libraries and even if TNTBASE runs on a remote server, this query outperforms the straightforward implementation based on local files.

Moreover, using the virtual documents of TNTBASE, such generated documents are editable; see [ZK10] for details. TNTBASE keeps track of how a document was aggregated and propagates the necessary patches when a changed version of the virtual document is committed.

10.3. Twelf – an MMT*-Compliant Logical Framework*

The MMT implementation from Section 10.1 starts with a generic MMT implementation and adds a plugin for a specific formal language F. Alternatively, an invasive implementation is possible, which starts with an implementation of F and adds the MMT module system to it. Such implementations are restricted to theory graphs with a single foundational theory for F, but can reuse special features for F such as user interfaces and type inference. We have implemented this for LF in [RS09] based on the Twelf implementation [PS99] of LF.[5]

The effect of adding MMT to Twelf is that Twelf becomes a tool for authoring theory graphs with LF as the single foundational theory. A major advantage of this approach is that authors can benefit from the advanced Twelf features, in particular infix parsing, type reconstruction, and implicit arguments. This implementation was used successfully to generate large case studies of MMT theory graphs in [DHS09], [HR11], and [IR11].

Twelf uses MMT URIs for namespace management, and a catalog service [Iac11] provides the translation of logical URIs into physical URLs. Twelf also supports several advanced language features that are part of MMT but were not mentioned in this paper. In particular, this includes nested theories and

[5] See `https://trac.kwarc.info/MMT/wiki/Twelf`.

unnamed imports between theories and links. Furthermore, fixity and precedence declarations of Twelf are preserved as MMT notations that are used when rendering the MMT theory graph.

Twelf can produce MMT documents in MMT's XML syntax from its input that are guaranteed to be well-formed. In [CHK+12b], we showed how logics written in Twelf can be exported in MMT concrete syntax and imported into and used in the Hets system [MML07].

11. Conclusion and Future Work

Formal knowledge is at the core of mathematics, logic, and computer science, 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 this knowledge, often in the form of logical formulas. Unfortunately, these systems have differing domains of applications, foundational assumptions, and input languages, which makes them non-interoperable and difficult to compare and relate 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 easily share knowledge representations.

In this work, we contributed to the solution of this problem by providing a scalable representation language for mathematical knowledge. We have focused on the modular organization of formal, explicitly represented mathematical knowledge. We have developed a classification of modular knowledge representation languages and evaluated the space of possible module systems in terms of expressivity, computational tractability, and scalability. We have distilled our findings into one particularly well-behaved system – MMT – discussed its properties, and described a set of loosely coupled implementations.

11.1. The MMT *Language*

MMT is a foundationally unconstrained module system that serves as a web-scalable interface layer between computational systems working with formally represented knowledge.

MMT integrates successful features of existing paradigms

- reuse along theory morphisms from the "little theories" approach,
- the theory graph abstraction from algebraic specification languages,
- categories of theories and logics from model theoretical logical frameworks,
- the logics-as-theories representation from proof theoretical logical frameworks,
- declarations of constants and named realizations from type theory,
- the Curry-Howard correspondence from type/proof theory,
- URIs as logical namespace identifiers from OpenMath/OMDoc and Java,

- standardized XML-based concrete syntax from web-oriented representation languages,

and makes them available in a single, coherent representational system for the first time.

The combination of these features is reduced to a small set of carefully chosen, orthogonal primitives in order to obtain a simple and extensible language design. In fact, some of the primitives combine so many intuitions that it was rather difficult to name them.

In addition, MMT contributes three new features:

Canonical identifiers By making morphisms named objects, MMT can provide globally unique, web-scalable identifiers for all knowledge items. Even in the presence of modularity and reuse, all induced knowledge items become addressable via URIs. Moreover, identifiers are invariant under MMT operations such as flattening.

Meta-theories The logical foundations of domain representations of mathematical knowledge can be represented as modules themselves and can be structured and interlinked via meta-morphisms. Thus, the different foundations of systems can be related and the systems made interoperable. The explicit representation of epistemic foundations also benefits systems whose mathematical knowledge is only implicitly embedded into the algorithms: The explicit representation can serve as a documentation of the system interface as well as a basis for verification or testing attempts.

Foundation-independence The design, implementation, and maintenance of large scale logical knowledge management services will realistically only pay off if the same framework can be reused for different foundations of mathematics. Therefore, MMT does not commit to a particular foundation and provides an interface layer between the logical-mathematical core of a mathematical foundation and knowledge management services. Thus, the latter can respect the semantics of the former without knowing or implementing the foundation.

MMT is **web-scalable** in the sense that it supports the distribution of resources (theories, proofs, etc.) over the internet thus permitting their collaborative development and application. We can encapsulate MMT-based or MMT-aware systems as web-services and use MMT as a universal interface language. At the same time MMT is **fully formal** in the sense that its semantics is specified rigorously in a self-contained formal system, namely using the type-theoretical style of judgments and inference rules. Such a level of formality is rare among module systems, SML being one of the few examples.

We contend that the dream of formalizing large parts of mathematics to make them machine-understandable can only be reached based on a system with both these features. However, in practice, they are often in conflict, and their combination makes MMT unique. In particular, it is easy to write large scale implementations in MMT, and it is easy to verify and trust them.

11.2. Beyond MMT

We have designed MMT as the simplest possible language that combines foundation-independence, modularity, web-scalability, and formality. Future work can now build on MMT and add individual orthogonal language features – in each case preserving these four qualities. In particular, for each feature, we have to define grammar and inference rules, the induced knowledge items and their URIs, and their behavior under theory morphisms. In fact, we have already developed some of these features but excluded them in this paper to focus on a minimal core language.

In the following we list some language features that we plan to add MMT in the future:

Unnamed Imports In addition to the described named imports with distinguish-semantics, MMT is designed to provide also unnamed imports with identify-semantics. They are already part of the MMT API, and the main reason to omit them here was to simplify the presentation of the formal semantics of MMT.

Cyclic Imports Cyclic unnamed imports are common when modules are not used for encapsulation as in MMT but for namespace management; this is permitted, e.g., between OWL ontologies or OPENMATH content dictionaries. But interestingly, inspecting the flattening theorem reveals that cyclic named imports are not as harmful for MMT as one might think: They can be elaborated easily if we permit theories with infinitely many constant declarations. In particular, cyclic named imports can permit elegant representations of languages with an infinite hierarchy of universes, coinductive data types, or an infinite hierarchy of reflection.

Nested Theories Nested theories will provide a scalable mechanism for representing hierarchic scopes and visibility. Many language features naturally suggest such a nesting of scopes such as mutual recursion, local functions, record types, or proofs with local definitions.

Intuitively, if S is a subtheory of T, the declarations of T occurring before S are implicitly imported into S via an unnamed import, and the declarations of T succeeding S can refer to S, e.g., by importing it. The main difficulty here is to add nested theories in a way that preserves the order-invariance of declarations.

Inductive Data Types and Record Types [Dum12] develops an extension of MMT that uses theories to represent inductive and record types. To represent an inductive data type, we define its constructors in a theory I using a distinguished type t over I. Now we can define an extension of MMT, in which for any theory T, the T-terms of the inductive type induced by (I, t) are the closed terms ω such that $\gamma \rhd_I \omega : t$. The foundation for I can be chosen easily by simply not equating terms unless required by MMT. Similarly, some inductive functions out of this type can be represented as theory morphisms from I to T that map t to the intended return type.

Similarly, to represent a record type over T, we define its fields in a theory R and give a distinguished partial morphism m from R to T. Now we can define the T-terms of the induced type as the total morphisms from R to T that agree with m. Then the selection of fields of a record can be represented as morphism application to the respective R-constant.

Theory Expressions Some of the module systems we discussed, e.g., CASL or Isabelle, provide complex theory expressions. For example, $S \cup T$ can denote the union of the theories S and T. Other examples are the translation of a theory along a morphism, the extension of a theory with some declarations, or the pushout along certain morphisms. Similarly, we can add further productions for morphism expression, e.g., for the mediating morphism out of a pushout.

The main difficulty here is that these complex theories and consequently their declarations do not have canonical identifiers. Indeed, most systems handle theory expressions by decomposing them internally and generating fresh internal names for the involved subexpressions. Similarly, all of these constructions can be expressed in MMT already by introducing auxiliary theories as we showed in [CHK+12a]. But certain theory expressions – most importantly unions and pushouts along unnamed imports – can be added to MMT in a way that preserves canonical identifiers without using generated names.

Sorting The components of a constant declaration – type and definiens – correspond to the base judgments provided by the foundations – typing and equality. In particular, MMT uses the constant declarations to provide the axioms of the inference systems used in specific foundations. It is natural but not necessary to consider exactly typing and equality. For example, we can extend MMT with constant declarations $c <: \tau$ that declare c as a *sort* refining τ. Examples are subtypes (refining types), type classes (refining the kind of types), and set theoretical classes (refining the universe of sets). This extension would go together with a subsorting judgment $\gamma \rhd_T \omega <: \omega'$ in the foundation.

Conservative Extensions [HKR12] develops an extension of MMT with a foundation-independent notion of language extension. It subsumes many important extension principles including many conservative extension principles and definition principles (e.g., case-based or implicit definitions).

Conservative extensions, in particular, can be represented using a theory S declaring the primitive concepts (e.g., axioms) and then another theory T that imports S and adds only defined (or definable) constants – the derived concepts (e.g., theorems). In that case, it is desirable to make this kind of conservativity of T explicit in order to exploit it later. For example, if T is conservative over S, then a theory importing S should implicitly also gain access to T.

Minimal Foundations Not all language features can be defined foundation-independently. Consider Mizar-style [TB85] implicit definitions of the form

$$\texttt{func}\ c\ \texttt{means}\ F(c);\ \texttt{correctness}\ P;$$

where P is a proof of $\exists^! x.F(x)$ and c is defined as that unique value. Such a definition is meaningful iff the foundational theory can express the quantifier $\exists^!$ of unique existence. Moreover, in that case it can be elaborated into the two declarations c and $c\_def : F(c)$ (which is in fact what Mizar and most other systems are doing).

The approach of [HKR12] also yields a way to add such pragmatic language features to MMT together with the minimal foundations needed to define their semantics. If an individual foundational theory M imports one of these distinguished minimal foundations, the corresponding pragmatic feature becomes available in theories with meta-theory M.

Further pragmatic declarations include, for example, function declarations (possible if M can express λ-abstraction) and constants with multiple types (possible if M can express intersection types). The above-mentioned features of sorting and inductive data types as well as the Curry-Howard representation of axioms, theorems, and proof rules can become special cases of pragmatic features as well. We can even generalize the notion of foundations and then recover the type and definiens of a constant as pragmatic features that are possible if M can express typing and equality.

Hiding and Filtering In [RKS11], we showed how a slight extension of the semantics of filtering yields a substantial increase in expressivity. In particular, it becomes possible to safely relax the strictness of filtering. The key idea is that foundations do not only say "yes" when confirming a typing or equality relation but also return a list of dependencies, which MMT maintains and uses to propagate filtering. We use a syntactically similar but semantically different extension of MMT in [CHK+12a] to extend MMT with model theoretical hiding. We expect that further research will permit the unification of these two features.

Logical Relations The notions of theory and theory morphisms between theories can be extended with logical relations between theory morphisms. MMT logical relations will be purely syntactical notions that correspond to the well-known semantic ones. A preliminary account was given in [Soj10]. They will permit natural representations of relations between realizations – such as model morphisms – as well as of extensional equality relations.

Computation MMT is currently restricted to declarative languages thus excluding the important role of computation, e.g., in computer algebra systems, decision procedures, and programs extracted from proofs. Generating code from appropriate MMT theories is relatively simple. But we also want to permit literal code snippets in the definiens of a constant. This will provide a formal interface between a formal semantics and scalable implementations.

Aliases MMT avoids the introduction of new names for symbols; instead, canonical qualified identifiers are formed. But this often leads to long unfriendly identifiers. Aliases for individual identifiers or identifier prefixes are

a simple syntactic device for providing human-friendly names, e.g., by declaring the aliases + and ∗ for `add/mon/comp` and `mult/comp` in the theory `Ring`. Moreover, such names can be used to make the modular structure of a theory transparent. This is already part of our implementation.

Declaration Patterns and Functors A common feature of declarative languages is that the declarations in a theory T with meta-theory M must follow one out of several patterns. For example, if M is first-order logic, then T should contain only declarations of function symbols, predicate symbol, and axioms. We can capture this foundation-independently in MMT by declaring such patterns in M and then pattern-checking the declarations in T against them.

Patterns also permit adding a notion of functors to MMT whose input is an arbitrary well-patterned theory T with meta-theory M. The output is a theory defined by induction on the list of declarations in T. This permits concise representations of functors between categories of theories, e.g., the functor that takes a sorted first-order theory and returns its translation to unsorted first-order logic by relativization of quantifiers. This can be extended to functors between categories of diagrams.

Narrative and Informal Representations One motivation behind MMT has been to give a formal semantics to OMDOC 1.2, and the present work does this for the OMDOC fragment concerned with formal theory development. It omits narrative aspects (e.g., document structuring, notations, examples, citations) as well as informal and semi-formal representations. We will extend MMT towards all of OMDOC, and this effort will culminate in the OMDOC 2 language. As a first step, we have included sectioning and notations in the MMT API. Many other features of OMDOC 1.2 will be recovered as pragmatic features in the above sense.

11.3. Applying MMT

The development of MMT and its implementations has been driven by our ongoing and intended applications. Most importantly, we have evaluated MMT on the logic atlas built in the LATIN project as described in Section 10. Here, MMT is applied in two ways.

Firstly, MMT provides the ontology used to organize the highly interlinked theories in the logic graph. In particular, the MMT principles of meta-theories and foundation-independence provide a clean separation of concerns between the logical framework (LF in the case of LATIN), the logics, and the domain theories written in these logics.

Secondly, MMT serves as the scalable interface language between the various MMT-aware software systems used in LATIN. Twelf [PS99] is used to write logics, TNTBASE [ZK09] for persistent storage, the MMT API for presentation and indexing, JOBAD [GLR09] for interactive browsing, and Hets [MML07] for institution-based cross-logic proof management, and we are currently adding

sTeXIDE [JK10] for semantic authoring support. MMT is crucial to communicate the content and its semantics between both the heterogeneous platforms and the respective developers. In particular, the canonical MMT identifiers have proved pivotal for the integration of software systems.

Building on the LATIN atlas, we are creating an "Open Archive of FlexiForms" (OAFF). It will store flexiformal (i.e., represented at flexible degrees of formality) representations of mathematical knowledge and supply them with MMT-base knowledge management services. OAFF will contain the domain theories and libraries written in the logics that are part of the LATIN atlas. Using MMT, it becomes possible to represent libraries developed in different foundational systems in one uniform formalism. Since MMT can also represent relations between the underlying foundational systems, this provides a base for practical reliable system integration. Besides the LF-based LATIN library, we have imported in this way the Mizar [TB85] library [IKRU12], the TPTP [SS98] library, and OWL [W3C09] ontologies [Hor12].

Acknowledgments. The work reported in this paper has evolved in many versions over the period of 5 years. During this time, our understanding of the pertinent issues has grown with the many case studies cited in this paper, and we gratefully acknowledge the contributions of the respective co-authors. We want to specifically mention Mihai Codescu, Fulya Horozal, Alin Iacob, Mihnea Iancu, and Till Mossakowski, who have collaborated in the LATIN project. Our understanding of meta-logical frameworks have profited from interactions with Frank Pfenning and Carsten Schürmann, and that of internalized module systems from interactions with Claudio Sacerdoti Coen. Christoph Lange has helped with understanding the issues of web scalability and the connection to linked open data and the semantic web. The list of possible extensions currently under investigation, in particular computation and theory expressions, has benefited from two joint workshops with the members of the MathScheme project, i.e., Jacques Carette, William Farmer, and Russell O'Connor.

Finally, we want to thank Alin Iacob and two anonymous referees for their meticulous and insightful comments, which triggered many local improvements and a structural re-organization of the material in the last revision of the paper.

Some aspects of the reported work were partially funded by the German Research Council (DFG). The first author has been supported by the German Academic Exchange Service (DAAD), the Logosphere project, and the German Merit Foundation during his Ph.D. time, which laid the early foundations of MMT.

[ABC+10] R. Ausbrooks, S. Buswell, D. Carlisle, G. Chavchanidze, S. Dalmas, S. Devitt, A. Diaz, S. Dooley, R. Hunter, P. Ion, M. Kohlhase, A. Lazrek, P. Libbrecht, B. Miller, R. Miner, C. Rowley, M. Sargent, B. Smith, N. Soiffer, R. Sutor, and S. Watt. Mathematical Markup Language (MathML) Version 3.0. Technical report, World Wide Web Consortium, 2010. See http://www.w3.org/TR/MathML3.

[ACTZ06] A. Asperti, C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli. Crafting a Proof Assistant. In T. Altenkirch and C. McBride, editors, *TYPES*, pages 18–32. Springer, 2006.

[AHMS99] S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an Evolutionary Formal Software-Development Using CASL. In D. Bert, C. Choppy, and P. Mosses, editors, *WADT*, volume 1827 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 1999.

[AHMS02] S. Autexier, D. Hutter, T. Mossakowski, and A. Schairer. The Development Graph Manager Maya (System Description). In H. Kirchner and C. Ringeissen, editors, *Algebraic Methods and Software Technology, 9th International Conference*, pages 495–502. Springer, 2002.

[Apa00] Apache Software Foundation. Apache Subversion, 2000. see http://subversion.apache.org/.

[Asp94] D. Aspinall. Types, Subtypes, and ASL+. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification*, pages 116–131. Springer, 1994.

[Bar92] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.

[BC04] Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[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] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, Internet Engineering Task Force, 2005.

[BM09] M. Birbeck and S. McCarron. A syntax for expressing Compact URIs. Technical report, World Wide Web Consortium, 2009. See http://www.w3.org/TR/2009/CR-curie-20090116/.

[Bou68] N. Bourbaki. *Theory of Sets*. Elements of Mathematics. Springer, 1968.

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

[CAB+86] R. Constable, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. Sasaki, and S. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986.

[CELM96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4, pages 65–89, 1996.

[CH88] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[CH00] R. Constable and J. Hickey. Nuprl's Class Theory and Its Applications. In F. Bauer and R. Steinbruggen, editors, *Foundations of Secure Computation*, pages 91–115. IOS Press, 2000.

[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*, pages 289–291. Springer, 2011.

[CHK+12a] M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, and F. Rabe. A Proof Theoretic Interpretation of Model Theoretic Hiding. In T. Mossakowski and H. Kreowski, editors, *Recent Trends in Algebraic Development Techniques 2010*, pages 118–138. Springer, 2012.

[CHK+12b] M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, F. Rabe, and K. Sojakova. Towards Logical Frameworks in the Heterogeneous Tool Set Hets. In T. Mossakowski and H. Kreowski, editors, *Recent Trends in Algebraic Development Techniques 2010*, pages 139–159. Springer, 2012.

[Chu40] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(1):56–68, 1940.

[CO00] O. Caprotti and M. Oostdijk. On Communicating Proofs in Interactive Mathematical Documents. In J. Campbell and E. Roanes-Lozano, editors, *AISC*, pages 53–64. Springer, 2000.

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

[dB70] N. de Bruijn. The Mathematical Language AUTOMATH. In M. Laudet, editor, *Proceedings of the Symposium on Automated Demonstration*, volume 25 of *Lecture Notes in Mathematics*, pages 29–61. Springer, 1970.

[DHS09] S. Dumbrava, F. Horozal, and K. Sojakova. A Case Study on Formalizing Algebra in a Module System. In F. Rabe and C. Schürmann, editors, *Workshop on Modules and Libraries for Proof Assistants*, pages 11–18. ACM, 2009.

[DS05] M. Duerst and M. Suignard. Internationalized Resource Identifiers (IRIs). RFC 3987, Internet Engineering Task Force, 2005.

[Dum12] S. Dumbrava. A Type Theory based on Reflection. Master's thesis, Jacobs University Bremen, 2012.

[End72] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[Far00] W. Farmer. An Infrastructure for Intertheory Reasoning. In D. McAllester, editor, *Conference on Automated Deduction*, pages 115–131. Springer, 2000.

[FGT92] W. Farmer, J. Guttman, and F. Thayer. Little Theories. In D. Kapur, editor, *Conference on Automated Deduction*, pages 467–581, 1992.

[FGT93] W. Farmer, J. Guttman, and F. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11(2):213–248, 1993.

[GB92] J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.

[GGMR09] F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 327–342. Springer, 2009.

[GJJ96] J. Gosling, W. Joy, and G. Steele Jr. *The Java Language Specification*. Addison-Wesley, 1996.

[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*, pages 279–293. Springer, 2009.

[GP93] M. Gordon and A. Pitts. The HOL Logic. In M. Gordon and T. Melham, editors, *Introduction to HOL, Part III*, pages 191–232. Cambridge University Press, 1993.

[GTW78] J. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology*, volume 4, pages 80–149. Prentice Hall, 1978.

[GWM+93] J. Goguen, Timothy Winkler, J. Meseguer, K. Futatsugi, and J. Jouannaud. Introducing OBJ. In J. Goguen, D. Coleman, and R. Gallimore, editors, *Applications of Algebraic Specification using OBJ*. Cambridge, 1993.

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

[HKR12] F. Horozal, M. Kohlhase, and F. Rabe. Extending MKM Formats at the Statement Level. In J. Campbell, J. Carette, G. Dos Reis, J. Jeuring, P. Sojka, V. Sorge, and M. Wenzel, editors, *Intelligent Computer Mathematics*, pages 64–79. Springer, 2012.

[Hor12] F. Horozal. Management of Change in the Web Ontology Language. Master's thesis, Jacobs University Bremen, 2012.

[HR09] P. Horn and D. Roozemond. OpenMath in SCIEnce: SCSCP and POPCORN. In J. Carette, L. Dixon, C. Sacerdoti Coen, and S. Watt, editors, *Intelligent Computer Mathematics*, pages 474–479. Springer, 2009.

[HR11] F. Horozal and F. Rabe. Representing Model Theory in a Type-Theoretical Logical Framework. *Theoretical Computer Science*, 412(37):4919–4945, 2011.

[HST94] R. Harper, D. Sannella, and A. Tarlecki. Structured presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994.

[Iac11] A. Iacob. Towards Project-Based Workflows in Twelf. Master's thesis, Jacobs University Bremen, 2011.

[IKRU12] M. Iancu, M. Kohlhase, F. Rabe, and J. Urban. The Mizar Mathematical Library in OMDoc: Translation and Applications. *Journal of Automated Reasoning*, 2012. to appear.

[IR11] M. Iancu and F. Rabe. Formalizing Foundations of Mathematics. *Mathematical Structures in Computer Science*, 21(4):883–911, 2011.

[IR12a] 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*, pages 325–340. Springer, 2012.

[IR12b] M. Iancu and F. Rabe. (Work-in-Progress) An MMT-Based User-Interface. In *Workshop on User Interfaces for Theorem Provers*, 2012.

[JK10] C. Jucovschi and M. Kohlhase. sTeXIDE: An Integrated Development Environment for sTeX Collections. In S. Autexier, J. Calmet, D. Delahaye, P. Ion, L. Rideau, R. Rioboo, and A. Sexton, editors, *Intelligent Computer Mathematics*, number 6167 in Lecture Notes in Artificial Intelligence. Springer, 2010.

[KMR08] M. Kohlhase, C. Müller, and F. Rabe. Notations for Living Mathematical Documents. In S. Autexier, J. Campbell, J. Rubio, V. Sorge, M. Suzuki, and F. Wiedijk, editors, *Mathematical Knowledge Management*, pages 504–519. Springer, 2008.

[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*, pages 370–384. Springer, 2010.

[KWP99] F. Kammüller, M. Wenzel, and L. Paulson. Locales – a Sectioning Concept for Isabelle. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics*, pages 149–166. Springer, 1999.

[Law63] F. Lawvere. *Functional Semantics of Algebraic Theories*. PhD thesis, Columbia University, 1963.

[LCH07] D. Lee, K. Crary, and R. Harper. Towards a mechanized metatheory of Standard ML. In M. Hofmann and M. Felleisen, editors, *Symposium on Principles of Programming Languages*, pages 173–184. ACM, 2007.

[Mac98] S. Mac Lane. *Categories for the working mathematician*. Springer, 1998.

[MAH06] T. Mossakowski, S. Autexier, and D. Hutter. Development graphs - Proof management for structured specifications. *J. Log. Algebr. Program*, 67(1–2):114–145, 2006.

[ML74] P. Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In *Proceedings of the '73 Logic Colloquium*, pages 73–118. North-Holland, 1974.

[MML07] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editor, *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522, 2007.

[Mos05] T. Mossakowski. Heterogeneous Specification and the Heterogeneous Tool Set, 2005. Habilitation thesis, see http://www.informatik.uni-bremen.de/~till/.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML,* Revised edition. MIT Press, 1997.

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

[Odl95] A. Odlyzko. Tragic loss or good riddance? The impending demise of traditional scholarly journals. *International Journal of Human-Computer Studies*, 42:71–122, 1995.

[Ora10] Oracle. Oracle berkeley db xml, 2010. see http://www.oracle.com/us/products/database/berkeley-db/xml/index.html.

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

[OS97] S. Owre and N. Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, SRI International, 1997.

[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.

[Rab08a] F. Rabe. *Representing Logics and Logic Translations*. PhD thesis, Jacobs University Bremen, 2008. see http://kwarc.info/frabe/Research/phdthesis.pdf.

[Rab08b] F. Rabe. The MMT System, 2008. see https://trac.kwarc.info/MMT/.

[Rab12a] F. Rabe. A Logical Framework Combining Model and Proof Theory. *Mathematical Structures in Computer Science*, 2012. to appear; see http://kwarc.info/frabe/Research/rabe_combining_10.pdf.

[Rab12b] 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*, pages 142–157. Springer, 2012.

[RK12] F. Rabe and M. Kohlhase. An XML Syntax for MMT. Technical report, OMDoc Report, 2012.

[RKS11] F. Rabe, M. Kohlhase, and C. Sacerdoti Coen. A Foundational View on Integration Problems. In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 107–122. Springer, 2011.

[RS09] F. Rabe and C. Schürmann. A Practical Module System for LF. In J. Cheney and A. Felty, editors, *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*, pages 40–48. ACM Press, 2009.

[SML97] Standard ML Basis Library, 1997. See http://www.standardml.org/Basis/.

[Soj10] K. Sojakova. Mechanically Verifying Logic Translations. Master's thesis, Jacobs University Bremen, 2010.

[Sol95] R. Solomon. On Finite Simple Groups and Their Classification. *Notices of the AMS*, pages 231–239, 1995.

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

[ST88] D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Control*, 76:165–210, 1988.

[SW83] D. Sannella and M. Wirsing. A Kernel Language for Algebraic Specification and Implementation. In M. Karpinski, editor, *Fundamentals of Computation Theory*, pages 413–427. Springer, 1983.

[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.

[W3C98] W3C. Extensible Markup Language (XML), 1998. http://www.w3.org/XML.

[W3C99] W3C. XML Path Language, 1999. http://www.w3.org/TR/xpath/.

[W3C07] W3C. XQuery 1.0: An XML Query Language, 2007. http://www.w3.org/TR/xquery/.

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

[ZBM31] Zentralblatt MATH (ZBMATH), 1931. http://www.zentralblatt-math.org.

[ZK09] V. Zholudev and M. Kohlhase. TNTBase: a Versioned Storage for XML. In *Proceedings of Balisage: The Markup Conference 2009*, volume 3 of *Balisage Series on Markup Technologies*. Mulberry Technologies, Inc., 2009.

[ZK10] V. Zholudev and M. Kohlhase. Scripting Documents with XQuery: Virtual Documents in TNTBase. In *Proceedings of Balisage: The Markup Conference*, Balisage Series on Markup Technologies. Mulberry Technologies, Inc., 2010.

[ZKR10] V. Zholudev, M. Kohlhase, and F. Rabe. A [insert XML Format] Database for [insert cool application]. In *XMLPrague 2010*. XMLPrague.cz, 2010.