An Open Markup Format for Mathematical Documents OMDoc [Version 1.6 (pre-2.0)]

December 15, 2015

Abstract:The OMDoc (Open Mathematical Documents) format is a content markup scheme for (collections of) mathematical documents including articles, textbooks, interactive books, and courses. OMDoc also serves as the content language for agent communication of mathematical services on a mathematical software bus.

This document is the specification of Version 1.6 of OMDoc of the OMDoc format, the first step towards OMDoc2. It defines the OMDoc language features and their meaning. The content of this part is normative for the OMDoc format; an OMDoc document is valid as an OMDoc document, iff it meets all the constraints imposed here. OMDoc applications will normally presuppose valid OMDoc documents and only exhibit the intended behavior on such.

Contents

Preface

The OMDoc (Open Mathematical Documents) format is a content markup scheme for (collections of) mathematical documents including articles, textbooks, interactive books, and courses. OMDoc also serves as the content language for agent communication of mathematical services on a mathematical software bus.

This document is the specification of Version 1.6 of OMDoc of the OMDoc format, the first step towards OMDoc2. It defines the OMDoc language features and their meaning. The content of this part is normative for the OMDoc format; an OMDoc document is valid as an OMDoc document, iff it meets all the constraints imposed here. OMDoc applications will normally presuppose valid OMDoc documents and only exhibit the intended behavior on such.

Part I

The OMDoc Format

In this chapter we will discuss issues that pertain to the general setup of the OMDoc format, before we present the respective modules in later chapters. OMDoc1.6 is the first step towards a second version of the OMDoc format.

Chapter 1

Dimensions of Representation in OMDoc

BNP:1

Strict vs. Pragmatic The OMDoc format is divided into two sublanguages: “Strict” OMDoc (in the lower half of Figure 1.1) and “Pragmatic” OMDoc (in the upper half[2]). EdN:2 The first subset uses a minimal set of elements representing the meaning of a mathematical expression in a uniform structure, while the second one tries to strike a pragmatic balance between verbosity and formality. Both forms of content expressions are legitimate and have their role in representing mathematics. The strict OMDoc format features a minimal set of conceptually orthogonal representational primitives, resulting in expressions with canonical structure, which simplifies the implementation of OMDoc processors as well as the comparison of content expressions. The pragmatic OMDoc format provides a large representational infrastructure that aims at being intuitive for humans to understand, read, and write.[3] EdN:3 In particular, the simplicity and conceptual clarity of strict OMDoc make allow to express structural well-formedness constraints, whereas the vocabulary of pragmatic OMDocis much nearer to mathematical practice and is thus easier to learn. It is a crucial design choice of the OMDocformat that the meaning fo pragmatic representations is defined entirely interms of strict representations[1]. Note that there may be multiple “pragmatic vocabularies” defined in terms of the strict core catering to different communities and their tastes.

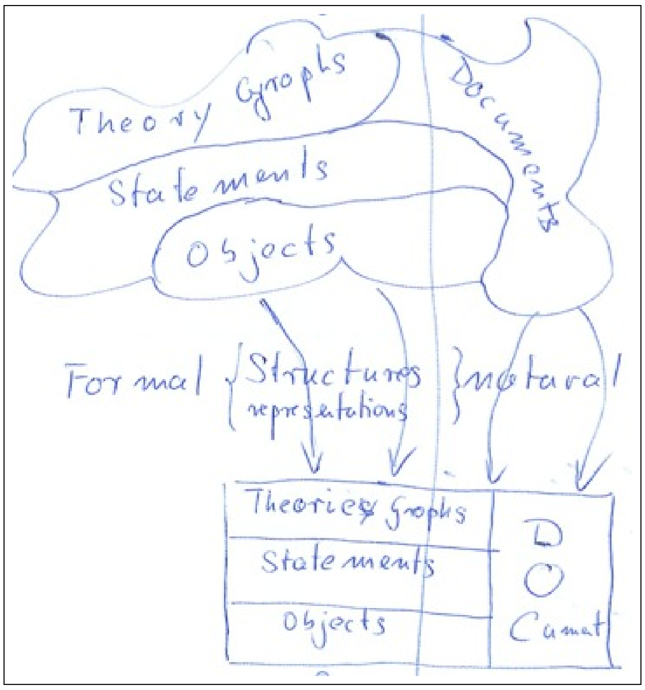

Figure 1.1: Dimensions of Representation in OMDoc

The introduction of strict OMDoc and the re-interpretation of pragmatic OMDoc in terms of it

[1]New Part: re-read and strengthen the argumentation
[2]EdNote: add the words “strict” and “pragmatic” to the picture
[3]EdNote: maybe state the numbers of elements in the end
[1]The strategy of dividing a markup format into a simple and structurally elegant core language and a larger set of pragmatic extensions which can be given a meaning by translating into the core was first pioneered by the author for content MathML3 [**CarlisleEd:MathML08**]

is radical redesign of the OMDoc format, which is new in OMDoc1.6. For this reason we consider OMDoc1.6 the first step into the direction of OMDoc2. With the development of strict OMDoc we aim to identify the representational primitives for representing mathematical documents, which can be given a simple and elegant semantics.

Formal vs. Informal One of the hallmarks of mathematical language is that is very rigorous in structure and usage in an attempt to fix the meaning of (mathematical) objects and statements about them. Indeed, the first decades of the last century established that mathematical language can in principle be expanded into logical form, where all objects and statements are fully identified by their syntactic form, and all reasoning steps are similarly justified by their form alone. we speak of "formal mathematics", when this is exercised and of "formal reasoning", when proofs are carried out in logical systems on this basis . In the last decades, significant parts of mathematical knowledge have been formalized and verified with the help of computers. But formalization and formal reasoning is still so costly and tedious that only a very small part of mathematics is formalized and verified in practice. Currently almost all mathematical documents consist of a mix of formal and informal (i.e. natural language) elements — certainly during the development of mathematical knowledge, but also in publications. Therefore are representation format for mathematical documents must allow this as well, consequently, OMDochas two sub-languages, "formal OMDoc" (on the left side of Figure 1.1) and "natural OMDoc" (on the right side).

OMDocoffers markup at three levels: objects, statements, and context.

objects are usually represented as *formulae* or *natural language phrases* in mathematical documents. In formal OMDocformulae are marked up according to their functional structure (as operator trees) and according to their layout in informal OMDoc(as layout trees). Note that any object can be represented in all three ways and all three ways of representation can be mixed at any level to account for mathematical practice, e.g. for mixed formulae like $\{n \in \mathbb{N} | n > 3 \text{ is prime}\}$.

statements are usually represented as *natural language sentences (with formulae)*[2] in informal settings and as (logical) formulae in formal ones. The discussion about the three ways of representation of objects applies analogously. Note that functional markup in formal OMDoconly addresses part of the requirements of formality, since their meaning depends on their context; we will explore this next.

theory graphs The context of objects (and the statements that contain them) is given by special statements (declarations). For conciseness and tractability, OMDocgroups declarations into "theories" and connects them by "theory morphisms" into "theory graphs". In a nutshell, every object (and thus every statement) has a "home theory", in which is meaningful. Theory morphisms make objects and statements available in their target theories.

As statements, theories and theory graphs are large objects, their informal representations (as mathematical text fragments and documents) usually carry linguistic cues to their discourse structure[4]. We discuss the relation between the discourse structure of informal representations and the formal structure of statements and theory graphs next. EdN:4

Discourse vs. Content Structure Mathematical Documents are very explicitly structured to help the reader grasp the complex objects, their relationships, and the flow of the argumentation in the proofs: Objects are often represented as formulae that reveal their structure, statements are labeled by indicators to their epistemic contribution to context (e.g. by labeling them as "definitions" or "theorems") and numbered for exact reference. The exposition of larger documents usually follows a topical structure with superimposed narrative structure driven by knowledge dependencies rather than e.g. a temporal dramaturgy driven by suspense. Even so, the structure of an informal document may be quite different from the formal structure of the knowledge it introduces. For instance, when we introduce a new concept in a course, we often first introduce a naive reduced approximation $\mathcal{N}$ of the real theory $\mathcal{F}$, only to show an example $\mathcal{E}_{\mathcal{N}}$ of where this

[2]or even larger text fragments made up of sentences like paragraphs

[4]EdNote: change the "documents" in Figure 1.1 to "discourse", at least in the strict box

is insufficient. Then we propose a first (straw-man) solution $\mathcal{S}$, and show an example $\mathcal{E}_\mathcal{S}$ of why this does not work. Based on the information we gleaned from this failed attempt, we build the eventual version $\mathcal{F}$ of the concept or theory and demonstrate that this works on $\mathcal{E}_\mathcal{F}$.

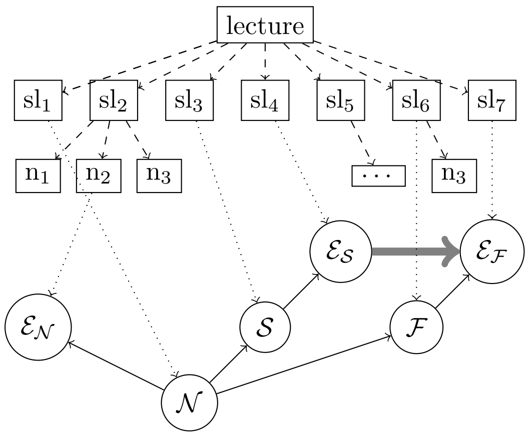

Figure 1.2: Content vs. Narrative Structures

The structure with the solid lines and boxes at the bottom of Figure 1.2 represents the content structure, where the circles $\mathcal{N}$, $\mathcal{E}_\mathcal{N}$, $\mathcal{S}$, $\mathcal{E}_\mathcal{S}$, $\mathcal{F}$, and $\mathcal{E}_\mathcal{F}$ signify theories for the content of the respective concepts and examples. The arrows represent the theory inheritance structure, e.g. Theory $\mathcal{F}$ imports theory $\mathcal{N}$. The top part of the diagram with the dashed lines stands for the narrative structure, where the arrows mark up the document structure. For instance, the slides sl_i are grouped into a lecture. In the example in Figure 1.2, the second slide of "lecture" presents the first example: the text fragment n_1 introduces it, and n_2 presents $\mathcal{E}_\mathcal{N}$ and n_2 might say something like "this did not work in the current situation, so we have to extend the conceptualization…". In a conventional setting, the narrative structure on the top and the content structure would be represented in different documents: The lecture slides and the formalization, and the equivalences (e.g. that n_2 verbalizes $\mathcal{E}_\mathcal{N}$; we have visualized these relations as dotted arrows in Figure 1.2) could not be taken advantage of, since they are not explicitly represented.

But these equivalences can be utilized to render services to the reader, for instance the imports relation in the theory graph on the lower half of Figure 1.2 induces a dependency relation that can be used to generate a minimal explanation (without the motivation) of $\mathcal{E}_\mathcal{F}$. For an example at the object level, consider for instance the formula $a(x+y^2)$, whose layout is ambiguous in two places: a could be a factor in a product (presented as juxtaposition) or a function that is applied to an argument. Likewise y^2 could be the variable y raised to the second power or the second element in the sequence $y^1, y^2, \ldots, y^n$. Humans can usually disambiguate this from the context, but a screen reader service needs access to the operator tree to read this as "a times [pause] x plus y squared" or "a applied to [pause] x plus y two".

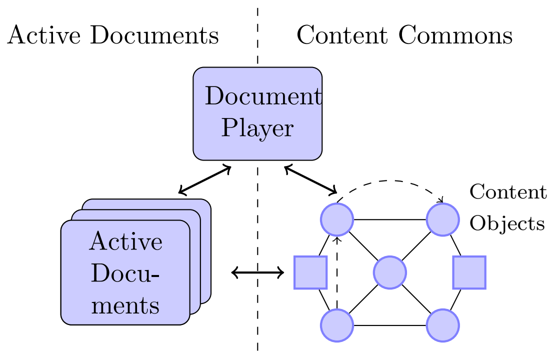

Figure 1.3: The Active Documents Paradigm

OMDocaims to reconcile the dichotomy between discourse structures (in informal mathematical documents which currently carry most of mathematical knowledge) and formal structures (that machines can operate upon) in one joint format. The central technique employed in OMDocis that of "parallel markup": The technique comes from MathML, where the `semantics` element is used to accomodate equivalent layout (presentation MathML) and operator trees (content MathML) and possibly foreign representations. Equivalence of nested sub-structures are represented by special cross-references. The MathML processor choses the one most adequate to its task — in the absence of distinguisthing information the first child.

OMDocextends this to the document level: The document contains elements whose children are alternative representations of the same object/statement/theory.[5] The significance of this is EdN:5
for that is Figure 1.3 shows the [6]. EdN:6

Just as for content-based systems on the formula level, there are now MKM systems that

[5]EdNote: implement this, and think about the cross-referencing, also need continuations to break tree overlaps, e.g.

generate presentation markup from content markup, based on general presentation principles, also on this level. For instance, the ACTIVEMATH system [**MelBue:krma03**] generates a simple narrative structure (the presentation; called a personalized book) from the underlying content structure (given in OMDoc) and a user model.

Coverage Currently our understanding of these primitives is largely limited to formal parts of mathematics, therefore strict OMDOC1.6 covers significantly less of informal mathematical documents than OMDOC1.2, so the meaning-giving translation from pragmatic OMDoc elements to strict OMDoc is partial. We plan to develop strict OMDoc into a system with greater coverage in the upcoming versions of OMDoc. OMDOC2.0 will be the first stable version where the coverage of strict OMDoc is complete. ENP:1

in content objects straddling slides.

[6] EDNOTE: talk about parallel markup, content documents and narrative documents and how to crosslink them and share structure

Chapter 2

OMDoc as a Modular Format

A modular approach to design is generally accepted as best practice in the development of any type of complex application. It separates the application's functionality into a number of "building blocks" or "modules", which are subsequently combined according to specific rules to form the entire application. This approach offers numerous advantages: The increased conceptual clarity allows developers to share ideas and code, and it encourages reuse by creating well-defined modules that perform a particular task. Modularization also reduces complexity by decomposition of the application's functionality and thus decreases debugging time by localizing errors due to design changes. Finally, flexibility and maintainability of the application are increased because single modules can be upgraded or replaced independently of others.

The OMDoc vocabulary has been split by thematic role, which we will briefly overview in Figure 2.1 before we go into the specifics of the respective modules in Part I to Part XIV. To avoid repetition, we will introduce some attributes already in this chapter that are shared by elements from all modules. In Part XV we will discuss the OMDoc document model and possible sub-languages of OMDoc that only make use of parts of the functionality (Chapter 48).

The first four modules in Figure 2.1 are required (mathematical documents without them do not really make sense), the other ones are optional. The document-structuring elements in module DOC have an attribute `modules` that allows to specify which of the modules are used in a particular document (see Part VI and Chapter 48).

Module	Title	Required?	Chapter
MOBJ	Mathematical Objects	yes	Part I
Formulae are a central part of mathematical documents; this module integrates the content-oriented representation formats OpenMath *and* MathML *into OMDoc*			
MTXT	Mathematical Text	yes	Part V
Mathematical vernacular, i.e. natural language with embedded formulae			
DOC	Document Infrastructure	yes	Part VI
A basic infrastructure for assembling pieces of mathematical knowledge into functional documents and referencing their parts			
DC	Metadata	yes	Part VII and Section 31.0
Contains bibliographical and licensing metadata ("data about data") which can be used to annotate many OMDoc elements by descriptive and administrative information that facilitates navigation and organization			
RT	Rich Text Structure	no	Chapter 21
Rich text structure in mathematical vernacular (lists, paragraphs, tables, . . .)			
ST	Mathematical Statements	no	Part IV
Markup for mathematical forms like theorems, axioms, definitions, and examples that can be used to specify or define properties of given mathematical objects and theories to group mathematical statements and provide a notion of context.			
PF	Proofs and proof objects	no	Part X
Structure of proofs and argumentations at various levels of details and formality			
ADT	Abstract Data Types	no	Chapter 32
Definition schemata for sets that are built up inductively from constructor symbols			
CTH	Complex Theories	no	Part XI
Theory morphisms; they can be used to structure mathematical theories			
DG	Development Graphs	no	Chapter 42
Infrastructure for managing theory inclusions, change management			
EXT	Applets, Code, and Data	no	Part XIII
Markup for applets, program code, and data (e.g. images, measurements, . . .)			
PRES	Presentation Information	no	Part XII
Limited functionality for specifying presentation and notation information for local typographic conventions that cannot be determined by general principles alone			
QUIZ	Infrastructure for Assessments	no	Part XIV
Markup for exercises integrated into the OMDoc document model			

Figure 2.1: The OMDoc Modules

Chapter 3

The OMDoc Namespaces

The namespace for the OMDoc2 format is the URI `http://omdoc.org/ns`. Note that the OMDoc namespace does not reflect the versions[1], this is done in the `version` attribute on the document root element `omdoc` (see Part VI). As a consequence, the OMDoc vocabulary identified by this namespace is not static, it can change with each new OMDoc version. However, if it does, the changes will be documented in later versions of the specification: the latest released version can be found at [**URL:omdocspec**].

In an OMDoc document, the OMDoc namespace must be specified either using a namespace declaration of the form `xmlns="http://omdoc.org/ns"` on the `omdoc` element or by prefixing the local names of the OMDoc elements by a namespace prefix (OMDoc customarily use the prefixes `omdoc:` or `o:`) that is declared by a namespace prefix declaration of the form `xmlns:o="http://omdoc.org/ns"` on some element dominating the OMDoc element in question (see for an introduction). OMDoc also uses the following namespaces[2]:

Format	namespace URI	see
Dublin Core	`http://purl.org/dc/elements/1.1/`	Part VII and Section 31.0
Creative Commons	`http://creativecommons.org/ns`	Part VIII
MathML	`http://www.w3.org/1998/Math/MathML`	Chapter 5
OpenMath	`http://www.openmath.org/OpenMath`	Chapter 4
XSLT	`http://www.w3.org/1999/XSL/Transform`	Part XII

Thus a typical document root of an OMDoc document looks as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<omdoc xml:id="test.omdoc" version="1.6"
  xmlns="http://omdoc.org/ns"
  xmlns:cc="http://creativecommons.org/ns"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:om="http://www.openmath.org/OpenMath"
  xmlns:m="http://www.w3.org/1998/Math/MathML">
...
</omdoc>
```

[1]The namespace is different from the OMDoc1 formats (versions 1.0, 1.1, and 1.2), which was `http://www.mathweb.org/omdoc`, but the OMDoc2 namespace will stay constant over all versions of the OMDoc2 format.

[2]In this specification we will use the namespace prefixes above on all the elements we reference in text unless they are in the OMDoc namespace.

Chapter 4

Common Attributes in OMDoc

Generally, the OMDoc format allows any attributes from foreign (i.e. non-OMDoc) namespaces on the OMDoc elements. This is a commonly found feature that makes the XML encoding of the OMDoc format extensible. Note that the attributes defined in this specification are in the default (empty) namespace: they do not carry a namespace prefix. So any attribute of the form `na:xxx` is allowed as long as it is in the scope of a suitable namespace prefix declaration.

Many OMDoc elements have optional `xml:id` attributes that can be used as identifiers to reference them. These attributes are of type `ID`, they must be unique in the document which is important, since many XML applications offer functionality for referencing and retrieving elements by `ID`-type attributes. Note that unlike other `ID`-attributes, in this special case it is the name `xml:id` [**XML:id05**] that defines the referencing and uniqueness functionality, not the type declaration in the DTD or XML schema (see for a discussion).

Note that in the OMDoc format proper, all ID type attributes are of the form `xml:id`. However in the older OpenMath and MathML standards, they still have the form `id`. The latter are only recognized to be of type `ID`, if a document type or XMLschema is present. Therefore it depends on the application context, whether a DTD should be supplied with the OMDoc document.

For many occasions (e.g. for printing OMDoc documents), authors want to control a wide variety of aspects of the presentation. OMDoc is a content-oriented format, and as such only supplies an infrastructure to mark up content-relevant information in OMDoc elements. To address this dilemma XML offers an interface to Cascading Style Sheets (CSS) [**BosHak:css98**], which allow to specify presentational traits like text color, font variant, positioning, padding, or frames of layout boxes, and even aural aspects of the text.

To make use of CSS, most OMDoc elements (all that have `xml:id` attributes) have `style` attributes that can be used to specify CSS directives for them. In the OMDoc fragment in Listing 4.1 we have used the `style` attribute to specify that the text content of the `omtext` element should be formatted in a centered box whose width is 80% of the surrounding box (probably the page box), and that has a 2 pixel wide solid frame of the specified RGB color. Generally CSS directives are of the form `A:V`, where `A` is the name of the aspect, and `V` is the value, several CSS directives can be combined in one `style` attribute as a semicolon-separated list (see [**BosHak:css98**] and the emerging CSS 3 standard).

Listing 4.1: Basic CSS Directives in a `style` Attribute

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type="text/css" href="http://example.org/style.css"?>
<omdoc xml:id="stylish">
  ...
  <omtext xml:id="t1" style="width:80%;align:center;border:2px #006699 solid">
    <h:p>Here comes something
      <h:span style="font-weight:bold;color:green" class="emphasize">stylish</h:span>!
    </h:p>
  </omtext>
  ...
</omdoc>
```

Note that many CSS properties of parent elements are inherited by the children, if they are not explicitly specified in the child. We could for instance have set the font family of all the children of the `omtext` element by adding a directive `font-family:sans-serif` there and then override it by a directive for the property `font-family` in one of the children.

Frequently recurring groups of CSS directives can be given symbolic names in CSS styles heets, which can be referenced by the `class` attribute. In Listing 4.1 we have made use of this with the class `emphasize`, which we assume to be defined in the style sheet `style.css` associated with the document in the "style sheet processing instruction" in the prolog[1] of the XML document (see [**Clark:assxd99**] for details). Note that an OMDoc element can have both `class` and `style` attributes, in this case, precedence is determined by the rules for CSS style sheets as specified in [**BosHak:css98**]. In our example in Listing 4.1 the directives in the `style` attribute take precedence over the CSS directives in the style sheet referenced by the `class` attribute on the `phrase` element. As a consequence, the word "stylish" would appear in green, bold italics.

[1]i.e. at the very beginning of the XML document before the document type declaration

Part II

Mathematical Objects (Module MOBJ)

A distinguishing feature of mathematics is its ability to represent and manipulate ideas and objects in symbolic form as mathematical formulae. OMDoc uses the OpenMath and Content-MathML formats to represent mathematical formulae and objects. Therefore, the OpenMath standard [**BusCapCar:2oms04**] and the MathML 2.0 recommendation (second edition) [**CarIon:MathML03**] are part of this specification. [7] EdN:7

We will review OpenMath objects in Chapter 4 and Content-MathML in Chapter 5, and specify an OMDoc element for entering mathematical formulae (element `legacy`) in Chapter 8.

Element	Attributes		Content
	Required	Optional	
`legacy`	`format`	`xml:id`, `formalism`	#PCDATA

Figure 4.1: Mathematical Objects in OMDoc

The recapitulation in the next two sections is not normative, please consult for a general introduction and history and the OpenMath standard and the MathML 2.0 Recommendation for details and clarifications.

[7] EdNote: discuss MathML3 and the relation between MathML and OpenMath and what that means for OMDoc

Chapter 5

OpenMath

[=OpenMath]

OpenMath is a markup language for mathematical formulae that concentrates on the meaning of formulae building on an extremely simple kernel (markup primitive for syntactical forms of content formulae), and adds an extension mechanism for mathematical concepts, the content dictionaries. **These** are machine-readable documents that define the meaning of mathematical concepts expressed by OpenMath symbols. The current released version of the OpenMath standard is OpenMath2, which incorporates many of the experiences of the last years, particularly with embedding OpenMath into the OMDoc format.

We will only review the XML encoding of OpenMath objects here, since it is most relevant to the OMDoc format. All elements of the XML encoding live in the namespace `http://www.openmath.org/OpenMath`, for which we traditionally use the namespace prefix `om:`. In OMDoc we embed OpenMath expressions without the enclosing `om:OMOBJ` element, since this does not seem to add anything.

Element	Attributes		Content
	Required	Optional	
om:OMS	cd, name	id, cdbase, class, style	EMPTY
om:OMV	name	id, class, style	EMPTY
om:OMA		id, cdbase, class, style	⟪*OMel*⟫*
om:OMBIND		id, cdbase, class, style	⟪*OMel*⟫,OMBVAR,⟪*OMel*⟫
om:OMBVAR		id, class, style	(OMV | OMATTR)+
om:OMFOREIGN		id, cdbase, class, style	ANY
om:OMATTR		id, cdbase, class, style	⟪*OMel*⟫
om:OMATP		id, cdbase, class, style	(OMS, (⟪*OMel*⟫|OMFOREIGN))+
om:OMI		id, class, style	[0-9]*
om:OMB		id, class, style	#PCDATA
om:OMF		id, class, style, dec, hex	#PCDATA
om:OME		id, class, style	⟪*OMel*⟫?
om:OMR	href		⟪*OMel*⟫?
where ⟪*OMel*⟫ is (OMS|OMV|OMI|OMB|OMSTR|OMF|OMA|OMBIND|OME|OMATTR)			

Figure 5.1: OpenMath Objects in OMDoc

5.1 The Representational Core of OpenMath

Definition 5.1.1 The central construct of the OpenMath is that of an OpenMath object, which has a tree-like representation made up of applications (**om:OMA**), binding structures (`om:OMBIND` using `om:OMBVAR` to tag bound variables), variables (**om:OMV**), and symbols (**om:OMS**).

`om:OMA`

`om:OMV`

`om:OMS`

The `om:OMA` element contains representations of the function and its argument in "prefix-" or

"Polish notation", i.e. the first child is the representation of the function and all the subsequent ones are representations of the arguments in order.

Objects and concepts that carry meaning independent of the local context (they are called **symbol** s in OpenMath) are represented as `om:OMS` elements, where the value of the `name` attribute gives the name of the symbol. The `cd` attribute specifies the relevant content dictionary, a document that defines the meaning of a collection of symbols including the one referenced by the `om:OMS`. This document can either be an original OpenMath content dictionary or an OMDoc document that serves as one (see Section 16.1 for a discussion). The optional `cdbase` on an `om:OMS` element contains a URI that can be used to disambiguate the content dictionary. Alternatively, the `cdbase` attribute can be given on an OpenMath element that is a parent to the `om:OMS` in question: The `om:OMS` inherits the `cdbase` of the nearest ancestor (inducing the usual XML scoping rules for declarations).[1]

The OpenMath2 standard proposes the following mechanism for determining a canonical identifying URI for the symbol declaration referenced by an OpenMath symbol of the form `<OMS cd="foo" name="bar"/>` with the `cdbase`-value e.g. `http://www.openmath.org/cd`: it is the URI reference `http://www.openmath.org/cd/foo#bar`, which by convention identifies an `omcd:CDDefinition` element with a child `omcd:Name` whose value is `bar` in a content dictionary resource `http://www.openmath.org/cd/foo.ocd` (see for a very brief introduction to OpenMath content dictionaries).

Variables are represented as `om:OMV` element. As variables do not carry a meaning independent of their local content, `om:OMV` only carries a `name` attribute (see Chapter 7 for further discussion).

For instance, the formula $\sin(x)$ would be modeled as an application of the sin function (which in turn is represented as an OpenMath symbol) to a variable:

```
<OMA xmlns="http://www.openmath.org/OpenMath"
     cdbase="http://www.openmath.org/cd">
  <OMS cd="transc1" name="sin"/>
  <OMV name="x"/>
</OMA>
```

In our case, the function sin is represented as an `om:OMS` element with name `sin` from the content dictionary `transc1`. The `om:OMS` inherits the `cdbase`-value `http://www.openmath.org/cd`, which shows that it comes from the OpenMath standard collection of content dictionaries from the `om:OMA` element above.[8] The variable x is represented in an `om:OMV` element with `name`-value `x`. EdN:8

For the `om:OMBIND` element consider the following representation of the formula $\forall x.\sin(x) \leq \pi$.

```
<OMBIND xmlns="http://www.openmath.org/cd">
  <OMS cd="quant1" name="forall"/>
  <OMBVAR><OMV name="x"/></OMBVAR>
  <OMA>
    <OMS cd="arith1" name="leq"/>
    <OMA><OMS cd="transc1" name="sin"/><OMV name="x"/></OMA>
    <OMS cd="nums1" name="pi"/>
  </OMA>
</OMBIND>
```

`om:OMBIND`

`om:OMBVAR`

Definition 5.1.2 The **om:OMBIND** element has exactly three children, the first one is a "binding operator"[2] — in this case the universal quantifier, the second one is a list of bound variables that must be encapsulated in an **om:OMBVAR** element, and the third is the body of the binding object, in which the bound variables can be used. OpenMath uses the `om:OMBIND` element to unambiguously specify the scope of bound variables in expressions: the bound variables in the `om:OMBVAR` element can be used only inside the mother `om:OMBIND` element, moreover they

[1] Note that while the `cdbase` inheritance mechanism described here remains in effect for OpenMath objects embedded in to the OMDoc format, it is augmented by one in OMDoc. As a consequence, OpenMath objects in OMDoc documents will usually not contain `cdbase` attributes; see Section 16.1 for a discussion.

[8] EdNote: MK: no, it does not, but from the theory in OMDoc, probably we should talk about this here.

[2] The binding operator must be a symbol which either has the role `binder` assigned by the OpenMath content dictionary (see [**BusCapCar:2oms04**] for details) or the symbol declaration in the OMDoc content dictionary must have the value `binder` for the attribute `role` (see Section 12.0).

can be systematically renamed without changing the meaning of the binding expression. As a consequence, bound variables in the scope of an `om:OMBIND` are distinct as OpenMath objects from any variables outside it, even if they share a name.

OpenMath offers an element for annotating (parts of) formulae with external information (e.g. MathML or LaTeX presentation):

Definition 5.1.3 The **om:OMATTR** element that pairs an OpenMath object with an attribute-value list. To annotate an OpenMath object, it is embedded as the second child in an `om:OMATTR` element. The attribute-value list is specified by children of the preceding **om:OMATP** (Attribute value Pair) element, which has an even number of children: children at odd positions must be `om:OMS` (specifying the attribute, they are called **key** s or **feature** s)[3], and children at even positions are the **value** s of the keys specified by their immediately preceding siblings. In the OpenMath fragment in Listing 5.1 the expression $x + \pi$ is annotated with an alternative representation and a color. Listing 7.1 has a more complex one involving types.

`om:OMATTR`

`om:OMATP`

Listing 5.1: Associating Alternate Representations with an OpenMath Object

```
<OMATTR>
  <OMATP>
    <OMS cd="alt−rep" name="ascii"/>
    <OMSTR>(x+1)</OMSTR>
    <OMS cd="alt−rep" name="svg"/>
    <OMFOREIGN encoding="application/svg+xml">
      <svg xmlns='http://www.w3.org/2000/svg'>...</svg>
    </OMFOREIGN>
    <OMS cd="pres" name="color"/>
    <OMS cd="pres" name="red"/>
  </OMATP>
  <OMA>
    <OMS cd="arith1" name="plus"/>
    <OMV name="x"/>
    <OMS cd="nums1" name="pi"/>
  </OMA>
</OMATTR>
```

A special application of the `om:OMATTR` element is associating non-OpenMath objects with OpenMath objects.

Definition 5.1.4 For this, OpenMath2 allows to use an **om:OMFOREIGN** element in the even positions of an `om:OMATP`. This element can be used to hold arbitrary XML content (in our example above SVG: Scalable Vector Graphics [**W3C:svg02**]), its required `encoding` attribute specifies the format of the content.

`om:OMFOREIGN`

We recommend a MIME type [**FreBor:MIME96**] (see ?spec@pres-bound? for an application).

5.2 Programming Extensions of OpenMath Objects

Definition 5.2.1 For representing objects in computer algebra systems OpenMath also provides other basic data types: **om:OMI** for integers, **om:OMB** for byte arrays, **om:OMSTR** for strings, and **om:OMF** for floating point numbers. These do not play a large role in the context of OMDoc, so we refer the reader to the OpenMath standard [**BusCapCar:2oms04**] for details.

`om:OMI`

`om:OMB`

`om:OMSTR`

`om:OMF`

Definition 5.2.2 The **om:OME** element is used for in-place error markup in OpenMath objects, it can be used almost everywhere in OpenMath elements. It has two children; the first

`om:OME`

[3]There are two kinds of keys in OpenMath distinguished according to the `role` value on their `symbol` declaration in the contentdictionary: `attribution` specifies that this attribute value pair may be ignored by an application, so it should be used for information which does not change the meaning of the attributed OpenMath object. The `role` is used for keys that modify the meaning of the attributed OpenMath object and thus cannot be ignored by an application.

one is an error operator[4], i.e. an OpenMath symbol that specifies the kind of error, and the second one is the faulty OpenMath object fragment. Note that since the whole object must be a valid OpenMath object, the second child must be a well-formed OpenMath object fragment.

As a consequence, the `om:OME` element can only be used for "semantic errors" like non-existing content dictionaries, out-of-bounds errors, etc. XML-well-formedness and DTD-validity errors will have to be handled by the XML tools involved. In the following example, we have marked up two errors in a faulty representation of $\sin(\pi)$. The outer error flags an arity violation (the function sin only allows one argument), and the inner one flags the typo in the representation of the constant π (we used the name `po` instead of `pi`).

```
<OME>
  <OMS cd="type−error" name="arity−violation"/>
  <OMA>
    <OMS cd="transc1" name="sin"/>
    <OME>
      <OMS cd="error" name="unexpected_symbol"/>
      <OMS cd="nums1" name="po"/>
    </OME>
    <OMV name="x"/>
  </OMA>
</OME>
```

As we can see in this example, errors can be nested to encode multiple faults found by an OpenMath application.

5.3 Structure Sharing in OpenMath

As we have seen above, OpenMath objects are essentially trees, where the leaves are symbols or variables. In many applications mathematical objects can grow to be very large, so that more space-efficient representations are needed. Therefore, OpenMath2 supports structure sharing[5] in OpenMath objects. In Figure 5.2 we have contrasted the tree representation of the object $1+1+1+1+1+1+1+1$ with the structure-shared one, which represents the formula as a directed acyclic graph (DAG). As any DAG can be exploded into a tree by recursively copying all sub-graphs that have more than one incoming graph edge, DAGs can conserve space by structure sharing. In fact the tree on the left in Figure 5.2 is exponentially larger than the corresponding DAG on the right.

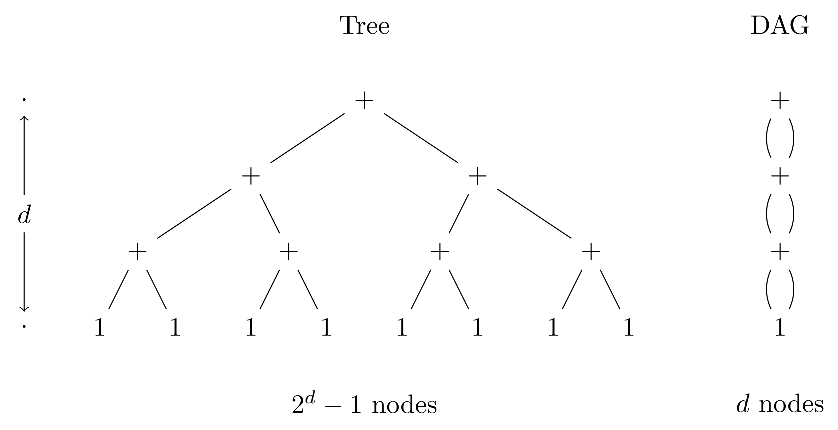

Figure 5.2: Structure Sharing by Directed Acyclic Graphs

[4]An error operator is like a binding operator, only the symbol has role `error`.

[5]Structure sharing is a well-known technique in computer science that tries to gain space efficiency in algorithms by re-using data structures that have already been created by pointing to them rather than copying.

Definition 5.3.1 To support DAG structures, OpenMath2 provides the (optional) attribute `id` on all OpenMath objects and an element **om:OMR** [6] for the purpose of cross-referencing. `om:OMR` The `om:OMR` element is empty and has the required attribute `href`; The OpenMath element represented by this `om:OMR` element is a copy of the OpenMath element pointed to in the `href` attribute.

Note that the representation of the `om:OMR` element is *structurally equal*, but not identical to the element it points to.

Using the `om:OMR` element, we can represent the OpenMath objects in Figure 5.2 as the XML representations in Figure 5.3.

Shared	Exploded
`<OMA>`	`<OMA>`
`  <OMS cd=”nat” name=”plus”/>`	`  <OMS cd=”nat” name=”plus”/>`
`  <OMA>`	`  <OMA id=”t1”>`
`    <OMS cd=”nat” name=”plus”/>`	`    <OMS cd=”nat” name=”plus”/>`
`    <OMA>`	`    <OMA id=”t11”>`
`      <OMS cd=”nat” name=”plus”/>`	`      <OMS cd=”nat” name=”plus”/>`
`      <OMI>1</OMI>`	`      <OMI>1</OMI>`
`      <OMI>1</OMI>`	`      <OMI>1</OMI>`
`    </OMA>`	`    </OMA>`
`    <OMA>`	`    <OMR href=”#t11”/>`
`      <OMS cd=”nat” name=”plus”/>`	
`      <OMI>1</OMI>`	
`      <OMI>1</OMI>`	
`    </OMA>`	
`  </OMA>`	`  </OMA>`
`  <OMA>`	`  <OMR href=”#t1”/>`
`    <OMS cd=”nat” name=”plus”/>`	
`    <OMA>`	
`      <OMS cd=”nat” name=”plus”/>`	
`      <OMI>1</OMI>`	
`      <OMI>1</OMI>`	
`    </OMA>`	
`    <OMA>`	
`      <OMS cd=”nat” name=”plus”/>`	
`      <OMI>1</OMI>`	
`      <OMI>1</OMI>`	
`    </OMA>`	
`  </OMA>`	
`</OMA>`	`</OMA>`

Figure 5.3: The OpenMath Objects from Figure 5.2 in XML Encoding

To ensure that the XML representations actually correspond to directed acyclic graphs, the occurrences of the `om:OMR` must obey the global acyclicity constraint below, where we say that an OpenMath element **dominate** s all its children and all elements they dominate; The `om:OMR` also dominates its **target** [7], i.e. the element that carries the `id` attribute pointed to by the `href` attribute. For instance, in the representation in Figure 5.3 the `om:OMA` element with `xml:id="t1"` and also the second `om:OMA` element dominate the `om:OMA` element with `xml:id="t11"`.

Axiom 5.3.2 (Acyclicity Constraint) An OpenMath element may not dominate itself.

Listing 5.2: A Simple Cycle

```
<OMA id=”foo”>
```

[6] OpenMath1 and OMDoc1.0 did now know structure sharing, OMDoc1.1 added `xref` attributes to the OpenMath elements `om:OMA`, `om:OMBIND` and `om:OMATTR` instead of `om:OMR` elements. This usage is deprecated in OMDoc1.2, in favor of the `om:OMR`-based solution from the OpenMath2 standard. Obviously, both representations are equivalent, and a transformation from `xref`-based mechanism to the `om:OMR`-based one is immediate.

[7] The target of an OpenMath element with an `id` attribute is defined analogously

```
  <OMS cd="nat" name="divide"/>
  <OMI>1</OMI>
  <OMA><OMS cd="nat" name="plus"/>
    <OMI>1</OMI>
    <OMR href="#foo"/>
  </OMA>
</OMA>
```

In Listing 5.2 the `om:OMA` element with `xml:id="foo"` dominates its third child, which dominates the `om:OMR` with `href="foo"`, which dominates its target: the `om:OMA` element with `xml:id="foo"`. So by transitivity, this element dominates itself, and by the acyclicity constraint, it is not the XML representation of an OpenMath object. Even though it could be given the interpretation of the continued fraction

$$\frac{1}{1+\frac{1}{1+\cdots}}$$

this would correspond to an infinite tree of applications, which is not admitted by the OpenMath standard. Note that the acyclicity constraint is not restricted to such simple cases, as the example in Listing 5.3 shows. Here, the `om:OMA` with `xml:id="bar"` dominates its third child, the `om:OMR` element with `href="baz"`, which dominates its target `om:OMA` with `xml:id="baz"`, which in turn dominates its third child, the `om:OMR` with `href="bar"`, this finally dominates its target, the original `om:OMA` element with `xml:id="bar"`. So again, this pair of OpenMath objects violates the acyclicity constraint and is not the XML encoding of an OpenMath object.

Listing 5.3: A Cycle of Order Two

```
<OMA id="bar">                          <OMA id="baz">
  <OMS cd="nat" name="plus"/>             <OMS cd="nat" name="plus"/>
  <OMI>1</OMI>                            <OMI>1</OMI>
  <OMR href="#baz"/>                      <OMR href="#bar"/>
</OMA>                                  </OMA>
```

Chapter 6

Content MathML

[=cMathML]

Content-MathML is a content markup format that represents the abstract structure of formulae in trees of logical sub-expressions much like OpenMath. However, in contrast to that, Content-MathML provides a lot of primitive tokens and constructor elements for the K-14 fragment of mathematics (Kindergarten to 14th grade (i.e. undergraduate college level)).

The current released version of the MathML recommendation is the second edition of MathML 2.0 [**CarIon:MathML03**], a maintenance release for the MathML 2.0 recommendation [**CarIon:MathML01**] that cleans up many semantic issues in the content MathML part. We will now review those parts of MathML 2.0 that are relevant to OMDoc; for the full story see [**CarIon:MathML03**].

Even though OMDoc allows full Content-MathML, we will advocate the use of the Content-MathML fragment described in this section, which is largely isomorphic to OpenMath (see Section 6.1 for a discussion).

Element	Attributes		Content
	Required	Optional	
`m:math`		`id`, `xlink:href`	《*CMel*》+
`m:apply`		`id`, `xlink:href`	`m:bvar?`,《*CMel*》*
`m:csymbol`	`definitionURL`	`id`, `xlink:href`	EMPTY
`m:ci`		`id`, `xlink:href`	#PCDATA
`m:cn`		`id`, `xlink:href`	([0-9]—,—.)(*—e([0-9]—,—.)*)?
`m:bvar`		`id`, `xlink:href`	`m:ci`— `m:semantics`
`m:semantics`		`id`, `xlink:href`, `definitionURL`	《*CMel*》,(`m:annotation` — `m:annotation-xml`)*
`m:annotation`		`definitionURL`, `encoding`	#PCDATA
`m:annotation-xml`		`definitionURL`, `encoding`	ANY
where 《*CMel*》 is `m:apply`— `m:csymbol`— `m:ci`— `m:cn`—`m:semantics`			

Figure 6.1: Content-MathML in OMDoc

6.1 The Representational Core of Content-MathML

Definition 6.1.1 The top-level element of MathML is the **m:math** [1] element, see Figure 6.3 for an example. `m:math`

Like OpenMath, Content-MathML organizes the mathematical objects into a functional tree.

The basic objects (MathML calls them **token element** s) are

identifier s (element **m:ci**) corresponding to variables. The content of the `m:ci` element is `m:ci`

[1]For DTD validation OMDoc uses the namespace prefix "`m:`" for MathML elements, since the OMDoc DTD needs to include the MathML DTD with an explicit namespace prefix, as both MathML and OMDoc have a `selector` element that would clash otherwise (DTDs are not namespace-aware).

arbitrary Presentation-MathML, used as the name of the identifier.

m:cn

number s (element **m:cn**) for number expressions. The attribute `type` can be used to specify the mathematical type of the number, e.g. `complex`, `real`, or `integer`. The content of the `m:cn` element is interpreted as the value of the number expression.

m:csymbol

symbol s (element **m:csymbol**) for arbitrary symbols. Their meaning is determined by a `definitionURL` attribute that is a URI reference that points to a symbol declaration in a defining document. The content of the `m:csymbol` element is a Presentation-MathML representation that used to depict the symbol.

Apart from these generic elements, Content-MathML provides a set of about 80 empty content elements that stand for objects, functions, relations, and constructors from various basic mathematic fields.

m:apply
m:bvar

Definition 6.1.2 The **m:apply** element does double duty in Content-MathML: it is not only used to mark up applications, but also represents binding structures if it has an **m:bvar** child;

see Figure 6.3 below for a use case in a universal quantifier.

m:semantics
m:annotation-xml
m:annotation

Definition 6.1.3 The **m:semantics** element provides a way to annotate Content-MathML elements with arbitrary information. The first child of the `m:semantics` element is annotated with the information in the **m:annotation-xml** (for XML-based information) and **m:annotation** (for other information) elements that follow it. These elements carry `definitionURL` attributes that point to a "definition" of the kind of information provided by them. The optional `encoding` is a string that describes the format of the content.

6.2 OpenMath vs. Content MathML

OpenMath and MathML are well-integrated; there are semantics-preserving converters between the two formats. MathML supports the `m:semantics` element, that can be used to annotate MathML presentations of mathematical objects with their OpenMath encoding. Analogously, OpenMath supports the `presentation` symbol in the `om:OMATTR` element, that can be used for annotating with MathML presentation. OpenMath is the designated extension mechanism for MathML beyond K-14 mathematics: Any symbol outside can be encoded as a `m:csymbol` element, whose `definitionURL` attribute points to the OpenMath CD that defines the meaning of the symbol. Moreover all of the MathML content elements have counterparts in the OpenMath core contentdictionaries [**URL:omcd-core**]. For the purposes of OMDoc, we will consider the various representations following four representations of a content symbol in Figure 6.2 as equivalent. Note that the URI in the `definitionURL` attribute does not point to a specific file, but rather uses its base name for the reference. This allows a MathML (or OMDoc) application to select the format most suitable for it.

In Figure 6.3 we have put the OpenMath and content MathML encoding of the law of commutativity for the real numbers side by side to show the similarities and differences. There is an obvious line-by-line similarity for the tree constructors and token elements. The main difference is the treatment of types and variables.

[=omml-types]

```
<m:plus/>
```
Content-MathML token element

```
<m:plus definitionURL="http://www.openmath.org/cd/arith1#plus"/>
```
a

Content-MathML token element with explicit pointer

```
<m:csymbol definitionURL="http://www.openmath.org/cd/arith1#plus"/>
```
empty Content-MathML `m:csymbol`

```
<m:csymbol definitionURL="http://www.openmath.org/cd/arith1#plus">
  <m:mo>+</m:mo>
</m:csymbol>
```
Content-MathML `m:csymbol` with presentation

```
<OMS cdbase="http://www.openmath.org/cd" cd="arith1"
name="plus"/>
```
OpenMath symbol

Figure 6.2: Four equivalent Representations of a Content Symbol

OpenMath

```
<OMBIND>
 <OMS cd=”quant1” name=”forall”/>
 <OMBVAR>
  <OMATTR>
   <OMATP>
    <OMS cd=”sts” name=”type”/>
    <OMS cd=”setname1” name=”R”/>
   </OMATP>
   <OMV name=”a”/>

  </OMATTR>
  <OMATTR>
   <OMATP>
    <OMS cd=”sts” name=”type”/>
    <OMS cd=”setname1” name=”R”/>
   </OMATP>

   <OMV name=”b”/>
  </OMATTR>
 </OMBVAR>
  <OMA>
   <OMS cd=”relation” name=”eq”/>
   <OMA>
    <OMS cd=”arith1” name=”plus”/>
    <OMV name=”a”/>
    <OMV name=”b”/>
   </OMA>
   <OMA>
    <OMS cd=”arith1” name=”plus”/>
    <OMV name=”b”/>
    <OMV name=”a”/>
   </OMA>
 </OMA>
</OMBIND>
```

MathML

```
<m:apply>
 <m:forall/>
 <m:bvar>






  <m:ci type=”real”>a</m:ci>
 </m:bvar>






 <m:bvar>
  <m:ci type=”real”>b</m:ci>

 </m:bvar>
 <m:apply>
  <m:eq/>
  <m:apply>
   <m:plus/>
   <m:ci type=”real”>a</m:ci>
   <m:ci type=”real”>b</m:ci>
  </m:apply>
  <m:apply>
   <m:plus/>
   <m:ci type=”real”>b</m:ci>
   <m:ci type=”real”>a</m:ci>
  </m:apply>
 </m:apply>
</m:apply>
```

Figure 6.3: OpenMath vs. C-MathML for Commutativity

Chapter 7

Representing Types in Content-MathML and OpenMath

Types are representations of certain simple sets that are treated specially in (human or mechanical) reasoning processes. In typed representations variables and constants are usually associated with types to support more guided reasoning processes. Types are structurally like mathematical objects (i.e. arbitrary complex trees). Since types are ubiquitous in representations of mathematics, we will briefly review the best practices for representing them in OMDoc.

MathML supplies the `type` attribute to specify types that can be taken from an open-ended list of type names. OpenMath uses the `om:OMATTR` element to associate a type (in this case the set of real numbers as specified in the `setname1` content dictionary) with the variable, using the feature symbol `type` from the `sts` content dictionary. This mechanism is much more heavy-weight in our special case, but also more expressive: it allows to use arbitrary content expressions for types, which is necessary if we were to assign e.g. the type $(\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$ for functionals on the real numbers. In such cases, the second edition of the MathML2 Recommendation advises a construction using the `m:semantics` element (see [**DevKoh:stm03**] for details). Listings 7.1 and 7.2 show the realizations of a quantification over a variable of functional type in both formats.

Listing 7.1: A Complex Type in OpenMath

```
<OMBIND>
  <OMS cd="quant1" name="forall"/>
  <OMBVAR>
    <OMATTR>
      <OMATP>
        <OMS cd="sts" name="type"/>
        <OMA><OMS cd="sts" name="mapsto"/>
          <OMA><OMS cd="sts" name="mapsto"/>
            <OMS cd="setname1" name="R"/>
            <OMS cd="setname1" name="R"/>
          </OMA>
          <OMA><OMS cd="sts" name="mapsto"/>
            <OMS cd="setname1" name="R"/>
            <OMS cd="setname1" name="R"/>
          </OMA>
        </OMA>
      </OMATP>
      <OMV name="F"/>
    </OMATTR>
  </OMBVAR>
  ...
</OMBIND>
```

Note that we have essentially used the same URI (to the `sts` content dictionary) to identify the fact that the annotation to the variable is a type (in a particular type system).

Listing 7.2: A Complex Type in Content-MathML

```
<m:math>
```

```
 <m:apply>
   <m:forall/>
   <m:bvar>
     <m:semantics>
       <m:ci>F</m:ci>
       <m:annotation−xml definitionURL="http://www.openmath.org/cd/sts#type">
         <m:apply>
           <m:csymbol definitionURL="http://www.openmath.org/cd/sts#mapsto"/>
           <m:apply>
             <m:csymbol definitionURL="http://www.openmath.org/cd/sts#mapsto"/>
             <m:csymbol definitionURL="http://www.openmath.org/cd/setname1#real"/>
             <m:csymbol definitionURL="http://www.openmath.org/cd/setname1#real"/>
           </m:apply>
           <m:apply>
             <m:csymbol definitionURL="http://www.openmath.org/cd/sts#mapsto"/>
             <m:csymbol definitionURL="http://www.openmath.org/cd/setname1#real"/>
             <m:csymbol definitionURL="http://www.openmath.org/cd/setname1#real"/>
           </m:apply>
         </m:apply>
       </m:annotation−xml>
     </m:semantics>
   </m:bvar>
   . . .
 </m:apply>
</m:math>
```

[=omml-semvar]

Chapter 8

The Semantics of Variables in OpenMath and Content-MathML

A more subtle, but nonetheless crucial difference between OpenMath and MathML is the handling of variables, symbols, their names, and equality conditions. OpenMath uses the `name` attribute to identify a variable or symbol, and delegates the presentation of its name to other methods such as style sheets. As a consequence, the elements `om:OMS` and `om:OMV` are empty, and we have to understand the value of the `name` attribute as a pointer to a defining occurrence. In case of symbols, this is the symbol declaration in the content dictionary identified in the `cd` attribute. A symbol `<OMS cd="`$\langle\!\langle cd_1\rangle\!\rangle$`" name="`$\langle\!\langle name_1\rangle\!\rangle$`"/>` is equal to `<OMS cd="`$\langle\!\langle cd_2\rangle\!\rangle$`" name="`$\langle\!\langle name_2\rangle\!\rangle$`"/>`, iff $\langle\!\langle cd_1\rangle\!\rangle=\langle\!\langle cd_2\rangle\!\rangle$ and $\langle\!\langle name_1\rangle\!\rangle=\langle\!\langle name_2\rangle\!\rangle$ as XML simple names. In case of variables this is more difficult: if the variable is bound by an `om:OMBIND` element We say that an `om:OMBIND` element **binds** an OpenMath variable `<OMV name="x"/>`, iff this `om:OMBIND` element is the nearest one, such that `<OMV name="x"/>` occurs in (second child of the `om:OMATTR` element in) the `om:OMBVAR` child (this is the **defining occurrence** of `<OMV name="x"/>` here)., then we interpret all the variables `<OMV name="x"/>` in the `om:OMBIND` element as equal and different from any variables `<OMV name="x"/>` outside. In fact the OpenMath standard states that bound variables can be renamedo without changing the object (α-conversion). If `<OMV name="x"/>` is not bound, then the scope of the variable cannot be reliably defined; so equality with other occurrences of the variable `<OMV name="x"/>` becomes an ill-defined problem. We therefore discourage the use of unbound variables in OMDoc; they are very simple to avoid by using symbols instead, introducing suitable theories if necessary (see Chapter 15).

MathML goes a different route: the `m:csymbol` and `m:ci` elements have content that is Presentation-MathML, which is used for the presentation of the variable or symbol name.[1] While this gives us a much better handle on presentation of objects with variables than OpenMath (where we are basically forced to make due with the ASCII[2] representation of the variable name), the question of scope and equality becomes much more difficult: Are two variables (semantically) the same, even if they have different colors, sizes, or font families? Again, for symbols the situation is simpler, since the `definitionURL` attribute on the `m:csymbol` element establishes a global identity criterion (two symbols are equal, iff they have the same `definitionURL` value (as URI strings; see [**BerFie:uri98**]).) The second edition of the MathML standard adopts the same solution for bound variables: it recommends to annotate the `m:bvar` elements that declare the bound variable with an `id` attribute and use the `definitionURL` attribute on the bound occurrences of the `m:ci` element to point to those. The following example is taken from [**KohDev:bvm03**], which has more details.

[1] Note that surprisingly, the empty Content-MathML elements are treated more in the OpenMath spirit.

[2] In the current OpenMath standard, variable names are restricted to alphanumeric characters starting with a letter. Note that unlike with symbols, we cannot associate presentation information with variables via style sheets, since these are not globally unique (see ?spec@pres-bound? for a discussion of the OMDoc solution to this problem).

```
<m:lambda>
  <m:bvar><m:ci xml:id="the−boundvar">complex presentation</m:ci></m:bvar>
  <m:apply>
    <m:plus/>
    <m:ci definitionURL="#the−boundvar">complex presentation</m:ci>
    <m:ci definitionURL="#the−boundvar">complex presentation</m:ci>
  </m:apply>
</m:lambda>
```

For presentation in MathML, this gives us the best of both approaches, the `m:ci` content can be used, and the pointer gives a simple semantic equivalence criterion. For presenting OpenMath and Content-MathML in other formats OMDoc makes use of the infrastructure introduced in module PRES; see ?spec@pres-bound? for a discussion.

[=legacy]

Chapter 9

Legacy Representation for Migration

Sometimes, OMDoc is used as a migration format from legacy texts (see [**Kohlhase:OMDoc1.6primer**] for an example). In such documents it can be too much effort to convert all mathematical objects and formulae into OpenMath or Content-MathML form.

Definition 9.0.1 For this situation OMDoc provides the **legacy** element, which can contain arbitrary math markup[1]. The `legacy` element can occur wherever an OpenMath object or Content-MathML expression can and has an optional `xml:id` attribute for identification. The content is described by a pair of attributes:

`legacy`

- `format` (required) specifies the format of the content using URI reference. OMDoc does not restrict the possible values, possible values include `TeX`, `pmml`, `html`, and `qmath`.
- `formalism` is optional and describes the formalism (if applicable) the content is expressed in. Again, the value is unrestricted character data to allow a URI reference to a definition of a formalism.

For instance in the following `legacy` element, the identity function is encoded in the untyped λ-calculus, which is characterized by a reference to the relevant Wikipedia article.

```
<legacy format="TeX" formalism="http://en.wikipedia.org/wiki/Lambda_calculus">
  \lambda{x}{x}
</legacy>
```

[1]If the content is an XML-based, format like Scalable Vector Graphics [**W3C:svg02**], the DTD must be augmented accordingly for validation.

Part III

Strict OMDoc

In this chapter we will *strict OMDoc*, a subset of the language that uses a minimal set of elements representing the meaning of a mathematical expression in a uniform structure. Eventually all other parts of OMDoc we call them pragmatic OMDoc will be defined in terms of strict OMDoc (see Chapter 0 for details).

Element	Attributes		M	`Content`
	Required	Optional	D	
theory	`name, meta`	`xml:id,`	+	`theory* & object* & imports* & instance*`
object	`name, semrole`	`xml:id, synrole`	+	`type*,definition?`
imports	`from`	`xml:id`	+	`--`
instance	`name, from`	`xml:id`	+	`metamorphism?,(maps*&hides*)`
view	`name, from, to`	`xml:id`	+	`metamorphism?,(maps*&hides*)`
maps	`flatname`	`xml:id`	−	⟪*mobj*⟫
hides	`flatname`	`xml:id`	−	
metamorphism		`xml:id`	−	⟪*mobj*⟫
where ⟪*mobj*⟫ is `(OMOBJ |m:math |legacy)`				

Figure 9.1: Strict OMDoc Elements

[=genmeta]

Part IV

Metadata (Module DC)

BNP:9

Metadata is "data about data" — in the case of OMDoc data about documents fragments, such as titles, authorship, language usage, or administrative aspects like modification dates, distribution rights, identifiers, or version information. OMDoc documents also contain data about realations between mathematical objects, statements, and theories, that other applications would consider as metadata. To ensure interoperability with such applications and the Semantic Web, OMDoc supports the extraction of OMDoc-specific metadata to the RDF format[10] and the annotation of many OMDoc elements with web-ontology metadata. EdN:10

In this section we will introduce the metadata framework for strict omdoc, which provides a generic, extensible infrastructure for adding metadata based on the recently stabilized RDFa [**AdidaEtAl08:RDFa**] a standard for flexibly embedding metadata into X(HT)ML documents. This design decision allows us to separate the *syntax* (which is standardized in RDFa) from the *semantics*, which we externalize in metadata ontologies, which can be encoded in OMDoc; see [**Kohlhase:OMDoc1.6primer**].

The OMDoc format will incorporate various concrete metadata vocabularies as part of the document format. These are given as documented ontologies encoded as OMDoc theories and are normative parts of the format specification. Note that these metadata theories[11] can be thought of as the minimal specifications of the metadata: Refinements are admissible, if they are formulated as OMDoc theories that entertain views into the normative theories.[12] EdN:11 EdN:12

Element	Attributes		Content
	Req.	Optional	
`metadata`			`(meta|link)*`
`meta`	`property,content`	`datatype`	`ANY`
`link`	`rel`	`href`	`(resource|mlink|meta)*`
`resource`		`typeof, about`	`(meta|link)*`

Figure 9.2: Metadata in OMDoc

Definition 9.0.2 The **metadta** element contains content encodings for RDF triples and resources. [13] `metadta` EdN:13

[9]New Part: @CL, please re-read and expand
[10]EdNote: MK@CL write up something about processing in projects or processing and reference it here
[11]EdNote: MK@MK: reference theory framework (probably the strict one)
[12]EdNote: MK@MK: dream up a framework how to include the ontologies into the spec (probably as the DC and CC chapters; but do not forget the assertion type ontology, see Ticket 1511 in the OMDoc TRAC).
[13]EdNote: @CL: need to verbalize this.

EdN:14
EdN:15

[14] [15]

[14]EdNote: @CL: explain curies here.
[15]EdNote: @CL: need a simple example here, best DC metadata that we can take up later.

Chapter 10

Pragmatic to Strict Translation

Every OMDoc element that admits a `metadata` child can serve as a metadata subject, so we can offer the following pragmatic (abbreviative) syntax:

pragmatic	strict equivalent
<⟪*elt*⟫ rel="⟪*r*⟫" href="⟪*h*⟫"> ⟪*body*⟫ </⟪*elt*⟫>	<⟪*elt*⟫> <metadata> <link rel="⟪*r*⟫" href="⟪*h*⟫"/> </metadata> ⟪*body*⟫ </⟪*elt*⟫>
<⟪*elt*⟫ rel="⟪*r*⟫" href="⟪*h*⟫"> <metadata> ⟪*meta*⟫ </metadata> ⟪*body*⟫ </⟪*elt*⟫>	<⟪*elt*⟫> <metadata> <link rel="⟪*r*⟫" href="⟪*h*⟫"/> ⟪*meta*⟫ </metadata> ⟪*body*⟫ </⟪*elt*⟫>

OMDoc1.2 had some descriptions of metadata inhertance for Dublin Core Metadata. In the new metadata framework we do not need to explicitly present this, since it is part of the metadata ontology: If a metadatum can be inferred it is defined to be present. [16] EdN:16 ENP:9

[16]EdNote: @CL, make a simple concrete example.

Part V

Mathematical Statements (Module ST)

In this chapter we will look at the OMDoc infrastructure to mark up the *functional structure* of mathematical statements and their interaction with a broader mathematical context.

Chapter 11

Types of Statements in Mathematics

[=statementtypes]

In the last chapter we introduced mathematical statements as special text fragments that state properties of the mathematical objects under discussion and categorized them as definitions, theorems, proofs,. . . . A set of statements about a related set of objects make up the context that is needed to understand other statements. For instance, to understand a particular theorem about finite groups, we need to understand the definition of a group, its properties, and some basic facts about finite groups first. Thus statements interact with context in two ways: the context is built up from (clusters of) statements, and statements only make sense with reference to a context. Of course this dual interaction of statements with *context*[1] applies to any text and to communication in general. In mathematics, where the problem is aggravated by the load of notation and the need for precision for the communicated concepts and objects, contexts are often discussed under the label of mathematical theories. We will distinguish two classes of statements with respect to their interaction with theories: We view axioms and definitions as *constitutive* for a given theory, since changing this information will yield a different theory (with different mathematical properties, see the discussion in). Other mathematical statements like theorems or the proofs that support them are not constitutive, since they only illustrate the mathematical objects in the theory by explicitly stating the properties that are implicitly determined by the constitutive statements.

To support this notion of context OMDoc supports an infrastructure for theories using special `theory` elements, which we will introduce in Chapter 15 and extend in Part XI. Theory-constitutive elements must be contained as children in a `theory` element; we will discuss them in Section 12.3, non-constitutive statements will be defined in Chapter 12. They are allowed to occur outside a `theory` element in OMDoc documents (e.g. as top-level elements), however, if they do they must reference a theory, which we will call their **home theory** in a special `theory` attribute. This situates them into the context provided by this theory and gives them access to all its knowledge. The home theory of theory-constitutive statements is given by the theory they are contained in.

The division of statements into constitutive and non-constitutive ones and the encapsulation of constitutive elements in `theory` elements add a certain measure of safety to the knowledge management aspect of OMDoc. Since XML elements cannot straddle document borders, all constitutive parts of a theory must be contained in a single document; no constitutive elements can be added later (by other authors), since this would change the meaning of the theory on which other documents may depend on.

Before we introduce the OMDoc elements for theory-constitutive statements, let us fortify our intuition by considering some mathematical examples. *Axioms* are assertions about (sets of) mathematical objects and concepts that are assumed to be true. There are many forms of

[1]In linguistics and the philosophy of language this phenomenon is studied under the heading of “discourse theories”, see e.g. [**KamRey:fdtl93**] for a start and references.

axiomatic restrictions of meaning in mathematics. Maybe the best-known are the five Peano Axioms for natural numbers.

1. 0 is a natural number.
2. The successor $s(n)$ of a natural number n is a natural number.
3. 0 is not a successor of any natural number.
4. The successor function is one-one (i.e. injective).
5. The set $\mathbb{N}$ of natural numbers contains only elements that can be constructed by axioms 1. and 2.

Figure 11.1: The Peano Axioms

The Peano axioms in Figure 11.1 (implicitly) introduce three symbols: the number 0, the successor function s, and the set $\mathbb{N}$ of natural numbers. The five axioms in Figure 11.1 jointly constrain their meaning such that conforming structures exist (the natural numbers we all know and love) any two structures that interpret 0, s, and $\mathbb{N}$ and satisfy these axioms must be isomorphic. This is an ideal situation — the axioms are neither too lax (they allow too many mathematical structures) or too strict (there are no mathematical structures) — which is difficult to obtain. The latter condition (**inconsistent** theories) is especially unsatisfactory, since any statement is a theorem in such theories. As consistency can easily be lost by adding axioms, mathematicians try to keep axiom systems minimal and only add axioms that are safe.

Sometimes, we can determine that an axiom does not destroy consistency of a theory $\mathcal{T}$ by just looking at its form: for instance, axioms of the form $s = \mathbf{A}$, where s is a symbol that does not occur in $\mathcal{T}$ and $\mathbf{A}$ is a formula containing only symbols from $\mathcal{T}$ will introduce no constraints on the meaning of $\mathcal{T}$-symbols. The axiom $s = \mathbf{A}$ only constrains the meaning of the new symbol to be a unique object: the one denoted by $\mathbf{A}$. We speak of a **conservative extension** in this case. So, if $\mathcal{T}$ was a consistent theory, the extension of $\mathcal{T}$ with the symbol s and the axiom $s = \mathbf{A}$ must be one too. Thus axioms that result in conservative extensions can be added safely — i.e. without endangering consistency — to theories.

Generally an axiom $\mathcal{A}$ that results in a conservative extension is called a **definition** and any new symbol it introduces a **definiendum** (usually marked e.g. in boldface font in mathematical texts), and we call **definiens** the material in the definition that determines the meaning of the definiendum.

Chapter 12

Theory-Constitutive Statements in OMDoc

[=constitutive-statements]

The OMDoc format provides an infrastructure for four kinds of theory-constitutive statements: symbol declarations, type declarations, (proper) axioms, and definitions. We will take a look at all of them now.

Element	Attributes		M	Content
	Required	Optional	C	
symbol	`name`	`xml:id, role, scope, style, class`	+	`type*`
type		`xml:id, system, style, class`	–	`h:p*,`⟪*mobj*⟫
axiom	`name`	`xml:id, for, type, style, class`	+	`h:*,FMP*`
definition	`for`	`xml:id, type, style, class`	+	`h:p*,` ⟪*mobj*⟫`)?`
where ⟪*mobj*⟫ is `(OMOBJ |m:math |legacy)`				

Figure 12.1: Theory-Constitutive Elements in OMDoc

12.1 Symbol Declarations

The `symbol` element declares a symbol for a mathematical concept, such as 1 for the natural number "one", + for addition, = for equality, or `group` for the property of being a group. Note that we not only use the `symbol` element for mathematical objects that are usually written with mathematical symbols, but also for any concept or object that has a definition or is restricted in its meaning by axioms.

We will refer to the mathematical object declared by a `symbol` element as a "symbol", iff it is usually communicated by specialized notation in mathematical practice, and as a "concept" otherwise. The name "symbol" of the `symbol` element in OMDoc is in accordance with usage in the philosophical literature (see e.g. [**NewSim:cseisas81**]): A **symbol** is a *mental or physical* representation of a concept. In particular, a symbol may, but need not be representable by a (set of) glyphs (symbolic notation). The definiendum objects in Figure 32.1 would be considered as "symbols" while the concept of a "group" in mathematics would be called a "concept".

Definition 12.1.1 The **symbol** element has a required attribute `name` whose value uniquely identifies it in a theory. Since the value of this attribute will be used as an OpenMath symbol name, it must be an XML name[1] as defined in XML 1.1 [**xml1.1:04**]. The optional attribute `symbol`

[1]This limits the characters allowed in a name to a subset of the characters in Unicode 2.0; e.g. the colon

`scope` takes the values `global` and `local`, and allows a simple specification of visibility conditions: if the `scope` attribute of a `symbol` has value `local`, then it is not exported outside the theory; The `scope` attribute is deprecated, a formalization using the `hiding` attribute on the `imports` element should be used instead. Finally, the optional attribute `role` that can take the values[2]

`binder` The symbol may appear as a binding symbol of an binding object, i.e. as the first child of an `om:OMBIND` object, or as the first child of an `m:apply` element that has an `m:bvar` as a second child.

`attribution` The symbol may be used as key in an OpenMath `om:OMATTR` element, i.e. as the first element of a key-value pair, or in an equivalent context (for example to refer to the value of an attribution). This form of attribution may be ignored by an application, so should be used for information which does not change the meaning of the attributed OpenMath object.

`semantic-attribution` This is the same as `attribution` except that it modifies the meaning of the attributed OpenMath object and thus cannot be ignored by an application.

`error` The symbol can only appear as the first child of an OpenMath error object.

`application` The symbol may appear as the first child of an application object.

`constant` The symbol cannot be used to construct a compound object.

`type` The symbol denotes a sets that is used in a type systems to annotate mathematical objects.

`sort` The symbol is used for a set that are inductively built up from constructor symbols; see Chapter 32.

If the `role` is not present, the value `object` is assumed.

The children of the `symbol` element consist of a multi-system group of `type` elements (see Section 12.2 for a discussion). For this group the order does not matter. In Listing 12.1 we have a symbol declaration for the concept of a monoid. Keywords or simple phrases that describes the symbol in mathematical vernacular can be added in the `metadata` child of `symbol` as `dc:subject` and `dc:descriptions`; the latter have the same content model as the `h:p` elements, see the discussion in Part V). If the document containing their parent `symbol` element were stored in a data base system, it could be looked up via these metadata. As a consequence the symbol `name` need only be used for identification. In particular, it need not be mnemonic, though it can be, and it need not be language-dependent, since this can be done by suitable `dc:subject` elements.

Listing 12.1: An OMDoc `symbol` Declaration

```
<symbol name="monoid">
  <metadata>
    <dc:subject xml:lang="en">monoid</dc:subject>
    <dc:subject xml:lang="de">Monoid</dc:subject>
    <dc:subject xml:lang="it">monoide</dc:subject>
  </metadata>
  <type system="simply−typed">set[any] → (any → any → any) → any → bool</type>
  <type system="props">
    <OMS cd="arities" name="ternary−relation"/>
  </type>
</symbol>
```

: is not allowed. Note that this is not a problem, since the name is just used for identification, and does not necessarily specify how a symbol is presented to the human reader. For that, OMDoc provides the notation definition infrastructure presented in Part XII.

[2]The first six values come from the OpenMath2 standard. They are specified in content dictionaries; therefore OMDoc also supplies them.

12.2 Axioms

The relation between the components of a monoid would typically be specified by a set of axioms (e.g. stating that the base set is closed under the operation). For this purpose OMDoc uses the `axiom` element:

Definition 12.2.1 The **axiom** element contains mathematical vernacular as a sequence of `h:p` elements a `FMP` that expresses this as a logical formula. `axiom` elements may have a `generated-from` attribute, which points to another OMDoc element (e.g. an `adt`, see Chapter 32) which subsumes it, since it is a more succinct representation of the same mathematical content. Finally the `axiom` element has an optional `for` attribute to specify salient semantic objects it uses as a whitespace-separated list of URI references to symbols declared in the same theory, see Listing 12.2 for an example. Finally, the `axiom` element can have an `type` attribute, whose values we leave unspecified for the moment. `axiom`

Listing 12.2: An OMDoc `axiom`

```
<axiom xml:id="mon.ax" for="monoid">
  <h:p>If (M,∗) is a semigroup with unit e, then (M,∗,e) is a monoid.</h:p>
</axiom>
```

12.3 Type Declarations

Types (also called sorts in some contexts) are representations of certain simple sets that are treated specially in (human or mechanical) reasoning processes. A **type declaration** $e\colon t$ makes the information that a symbol or expression e is in a set represented by a type t available to a specified mathematical process. For instance, we might know that 7 is a natural number, or that expressions of the form $\sum_{i=1}^{n} a_i x^i$ are polynomials, if the a_i are real numbers, and exploit this information in mathematical processes like proving, pattern matching, or while choosing intuitive notations. If a type is declared for an expression that is not a symbol, we will speak of a **term declaration** .

Definition 12.3.1 OMDoc uses the **type** element for type declarations. The optional attribute `system` contains a URI reference that identifies the type system which interprets the content. There may be various sources of the set membership information conveyed by a type declaration, to justify it this source may be specified in the optional `just-by` attribute. The value of this attribute is a URI reference that points to an `assertion` or `axiom` element that asserts $\forall x_1, \ldots, x_n.e \in t$ for a type declaration $e\colon t$ with variables $x_1, \ldots, x_n$. If the `just-by` attribute is not present, then the type declaration is considered to be generated by an implicit axiom, which is considered theory-constitutive[3]. `type`

The `type` element contains one or two mathematical objects. In the first case, it represents a type declaration for a symbol (we call this a **symbol declaration**), which can be specified in the optional `for` attribute or by embedding the `type` element into the respective `symbol` element. A `type` element with two mathematical objects represents a term declaration $e\colon t$, where the first object represents the expression e and the second one the type t (see Listing 13.2 for an example). There the type declaration of `monoid` characterizes a monoid as a three-place predicate (taking as arguments the base set, the operation, and a neutral element).

As reasoning processes vary, information pertaining to multiple type systems may be associated with a single symbol and there can be more than one `type` declaration per expression and type system, this just means that the object has more than one type in the respective type system (not all type systems admit principal types).

[3]It is considered good practice to make the axiom explicit in formal contexts, as this allows an extended automation of the knowledge management process.

12.4 Definitions

Definitions are a special class axioms that completely fix the meaning of symbols.

definition **Definition 12.4.1** Therefore **definition** elements that represent definitions carry the required `for` attribute, which contain a whitespace-separated list of names of symbols in the same theory. We call these symbols **defined** and **primitive** otherwise. A `definition` contains mathematical vernacular as a sequen ce of `h:p` elements to describe the meaning of the defined symbols.

A `definition` element contains a mathematical object that can be substituted for the symbol specified in the `for` attribute of the definition. The `type` is fixed to `simple`[17]. Listing 12.3 gives an example of a simple definition of the number one from the successor function and zero. OMDoc treats the `type` attribute as an optional attribute. If it is not given explicitly, it defaults to `simple`. EdN:17

Listing 12.3: A Simple OMDoc `definition`.

```
<symbol name="one"/>
<definition xml:id="one.def" for="one" type="simple">
 <h:p><OMS cd="nat" name="one"/> is the successor of <om:OMS cd="nat" name="zero"/>.</h:p>
 <om:OMA>
   <om:OMS cd="int" name="suc"/>
   <om:OMS cd="nat" name="zero"/>
 </om:OMA>
</definition>
```

[17]EdNote: maybe better leave it out

Chapter 13

The Unassuming Rest

[=non-constitutive-statements]

The bulk of mathematical knowledge is in form of statements that are not theory-constitutive: statements of properties of mathematical objects that are entailed by the theory-constitutive ones. As such, these statements are logically redundant, they do not add new information about the mathematical objects, but they do make their properties explicit. In practice, the entailment is confirmed e.g. by exhibiting a proof of the assertion; we will introduce the infrastructure for proofs in Part X.

Element	Attributes		M	`Content`
	Required	Optional	D	
`assertion`		`xml:id, type, theory, class, style, status, just-by`	+	`h:p*, FMP*`
`type`	`system`	`xml:id, for, just-by, theory, class, style`	−	`h:p*,` ⟪*mobj*⟫`,`⟪*mobj*⟫
`example`	`for`	`xml:id, type, assertion, theory, class, style`	+	`h:p* |` ⟪*mobj*⟫`*`
`alternative`	`for, theory, entailed-by, entails, entailed-by-thm, entails-thm`	`xml:id, type, theory, class, style`	+	`h:p*, (FMP* | requation* |` ⟪*mobj*⟫`)`
where ⟪*mobj*⟫ is `(OMOBJ |m:math |legacy)`				

Figure 13.1: Assertions, Examples, and Alternatives in OMDoc

13.1 Assertions

Definition 13.1.1 OMDoc uses the **assertion** element for all statements (proven or not) about mathematical objects (see Listing 13.1) that are not axiomatic (i.e. constitutive for the meaning of the concepts or symbols involved). Traditional mathematical documents discern various kinds of these: theorems, lemmata, corollaries, conjectures, problems, etc. `assertion`

These all have the same structure (formally, a closed logical formula). Their differences are largely pragmatic (e.g. theorems are normally more important in some theory than lemmata) or proof-theoretic (conjectures become theorems once there is a proof). Therefore, we represent them in the general `assertion` element and leave the type distinction to a `type` attribute, which can have the values in Figure 13.2.

Definition 13.1.2 The `assertion` element also takes an optional `xml:id` element that allows to reference it in a document, an optional `theory` attribute to specify the theory that provides

Value	Explanation
`theorem`, `proposition`	an important assertion with a proof
Note that the meaning of the `type` (in this case the existence of a proof) is not enforced by OMDoc applications. It can be appropriate to give an assertion the `type` `theorem`, if the author knows of a proof (e.g. in the literature), but has not formalized it in OMDoc yet.	
`lemma`	a less important assertion with a proof
The difference of importance specified in this `type` is even softer than the other ones, since e.g. reusing a mathematical paper as a chapter in a larger monograph, may make it necessary to downgrade a theorem (e.g. the main theorem of the paper) and give it the status of a lemma in the overall work.	
`corollary`	a simple consequence
An assertion is sometimes marked as a corollary to some other statement, if the proof is considered simple. This is often the case for important theorems that are simple to get from technical lemmata.	
`postulate`, `conjecture`	an assertion without proof or counter-example
Conjectures are assertions, whose semantic value is not yet decided, but which the author considers likely to be true. In particular, there is no proof or counter-example (see Chapter 13).	
`false-conjecture`	an assertion with a counter-example
A conjecture that has proven to be false, i.e. it has a counter-example. Such assertions are often kept for illustration and historical purposes.	
`obligation`, `assumption`	an assertion on which the proof of another depends
These kinds of assertions are convenient during the exploration of a mathematical theory. They can be used and proven later (or assumed as an axiom).	
`formula`	if everything else fails
This type is the catch-all if none of the others applies.	

Figure 13.2: Types of Mathematical Assertions

the context for this assertion, and an optional attribute `generated-from`, that points to a higher syntactic construct that generates these assertions, e.g. an abstract data type declaration given by an `adt` element (see Chapter 32).

Listing 13.1: An OMDoc Assertion About Semigroups

```
<assertion xml:id="ida.c6s1p4.l1" type="lemma">
  <h:p> A semigroup has at most one unit.</h:p>
  <FMP>∀S.sgrp(S) → ∀x, y.unit(x, S) ∧ unit(y, S) → x = y</FMP>
</assertion>
```

To specify its proof-theoretic status of an assertion `assertion` carries the two optional attributes `status` and `just-by`. The first contains a keyword for the status and the second a whitespace-separated list of URI references to OMDoc elements that justify this status of the assertion. For the specification of the status we adapt an ontology for deductive states of assertion from [**SutZimSch:tdefatpt04**] (see Figure 13.3). Note that the states in Figure 13.3 are not mutually exclusive, but have the inclusions depicted in Figure 13.4.

`status`	`just-by` points to	
`tautology`	Proof of $\mathcal{F}$	*All $\mathcal{T}$-interpretations satisfy $\mathcal{A}$ and some $\mathcal{C}_i$*
`tautologous-conclusion`	Proof of $\mathcal{F}_c$.	*All $\mathcal{T}$-interpretations satisfy some $\mathcal{C}_j$*
`equivalent`	Proofs of $\mathcal{F}$ and $\mathcal{F}^{-1}$	*$\mathcal{A}$ and $\mathcal{C}$ have the same $\mathcal{T}$-models (and there are some)*
`theorem`	Proof of $\mathcal{F}$	*All $\mathcal{T}$-models of $\mathcal{A}$ (and there are some) satisfy some $\mathcal{C}_i$*
`satisfiable`	Model of $\mathcal{A}$ and some $\mathcal{C}_i$	*Some $\mathcal{T}$-models of $\mathcal{A}$ (and there are some) satisfy some $\mathcal{C}_i$*
`contradictory-axioms`	Refutation of $\mathcal{A}$	*There are no $\mathcal{T}$-models of $\mathcal{A}$*
`no-consequence`	$\mathcal{T}$-model of $\mathcal{A}$ and some $\mathcal{C}_i$, $\mathcal{T}$-model of $\mathcal{A} \cup \overline{\mathcal{C}}$.	*Some $\mathcal{T}$-models of $\mathcal{A}$ (and there are some) satisfy some $\mathcal{C}_i$, some satisfy $\overline{\mathcal{C}}$*
`counter-satisfiable`	Model of $\mathcal{A} \cup \overline{\mathcal{C}}$	*Some $\mathcal{T}$-models of $\mathcal{A}$ (and there are some) satisfy $\overline{\mathcal{C}}$*
`counter-theorem`	Proof of $\overline{\mathcal{C}}$ from $\mathcal{A}$	*All $\mathcal{T}$-models of $\mathcal{A}$ (and there are some) satisfy $\overline{\mathcal{C}}$*
`counter-equivalent`	Proof of $\overline{\mathcal{C}}$ from $\mathcal{A}$ and proof of $\mathcal{A}$ from $\overline{\mathcal{C}}$	*$\mathcal{A}$ and $\overline{\mathcal{C}}$ have the same $\mathcal{T}$-models (and there are some)*
`unsatisfiable-conclusion`	Proof of $\overline{\mathcal{C}}$	*All $\mathcal{T}$-interpretations satisfy $\overline{\mathcal{C}}$*
`unsatisfiable`	Proof of $\neg\mathcal{F}$	*All $\mathcal{T}$-interpretations satisfy $\mathcal{A}$ and $\overline{\mathcal{C}}$*

Where $\mathcal{F}$ is an assertion whose `FMP` has `assumption` elements $\mathcal{A}_1, \ldots, \mathcal{A}_n$ and `conclusion` elements $\mathcal{C}_1, \ldots, \mathcal{C}_m$. Furthermore, let $\mathcal{A} := \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ and $\mathcal{C} := \{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$, and $\mathcal{F}^{-1}$ be the sequent that has the $\mathcal{C}_i$ as assumptions and the $\mathcal{A}_i$ as conclusions. Finally, let $\overline{\mathcal{C}} := \{\overline{\mathcal{C}_1}, \ldots, \overline{\mathcal{C}_m}\}$, where $\overline{\mathcal{C}_i}$ is a negation of $\mathcal{C}_i$.

Figure 13.3: Proof Status for Assertions in a Theory $\mathcal{T}$

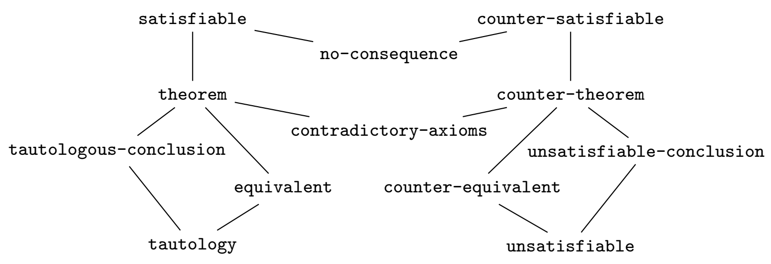

Figure 13.4: Relations of Assertion States

13.2 Type Assertions

In the last section, we have discussed the `type` elements in `symbol` declarations. These were axiomatic (and thus theory-constitutive) in character, declaring a symbol to be of a certain type, which makes this information available to type checkers that can check well-typedness (and thus plausibility) of the represented mathematical objects.

However, not all type information is axiomatic, it can also be deduced from other sources knowledge. We use the same `type` element we have discussed in Section 12.2 for such **type assertions** , i.e. non-constitutive statements that inform a type-checker. In this case, the `type` element can occur at top level, and even outside a `theory` element (in which case they have to specify their home theory in the `theory` attribute).

Listing 13.2 contains a type assertion $x + x \colon evens$, which makes the information that doubling an integer number results in an even number available to the reasoning process.

Listing 13.2: A Term declaration in OMDoc.

```
<type xml:id="double−even.td" system="#POST"
      theory="adv.int" for="plus" just−by="#double−even">
  <m:math>
    <m:apply><m:plus/>
      <m:ci type="integer">X</m:ci>
      <m:ci type="integer">X</m:ci>
    </m:apply>
  </m:math>
  <m:math>
    <m:csymbol definitionURL="http://cds.omdoc.org/integers/evens"/>
  </m:math>
</type>

<Assertion xml:id="double−even" type="lemma" theory="adv.int">
  <FMP>
    <m:math>
      <m:apply><m:forall/>
        <m:bvar><m:ci xml:id="x13" type="integer">X</m:ci></m:bvar>
        <m:apply><m:in/>
          <m:apply><m:plus/>
            <m:ci definitionURL="x13" type="integer">X</m:ci>
            <m:ci definitionURL="x13" type="integer">X</m:ci>
          </m:apply>
          <m:csymbol definitionURL="http://cds.omdoc.org/nat/evens"/>
        </m:apply>
      </m:apply>
    </m:math>
  </FMP>
</assertion>
```

The body of a type assertion contains two mathematical objects, first the type of the object and the second one is the object that is asserted to have this type.

13.3 Alternative Definitions

In contrast to what we have said about conservative extensions at the end of Section 12.3, mathematical documents often contain multiple definitions for a concept or mathematical object. However, if they do, they also contain a careful analysis of equivalence among them. OMDoc allows us to model this by providing the `alternative` element. Conceptually, an alternative definition or axiom is just a group of assertions that specify the equivalence of logical formulae. Of course, alternatives can only be added in a consistent way to a body of mathematical knowledge, if it is guaranteed that it is equivalent to the existing ones.

`alternative` **Definition 13.3.1** The `for` on the **alternative** points to the primary definition or assertion. Therefore, `alternative` has the attributes `entails` and `entailed-by`, that specify `assertions` that state the necessary entailments. It is an integrity condition of OMDoc that any `alternative` element references at least one `definition` or `alternative` element that entails it and one that it is entailed by (more can be given for convenience). The `entails-thm`, and `entailed-by-thm` attributes specify the corresponding assertions. This way we can always reconstruct equivalence of all definitions for a given symbol. As alternative definitions are not theory-constitutive, they can appear outside a `theory` element as long as they have a `theory` attribute.

13.4 Assertional Statements

There is another distinction for statements that we will need in the following. Some kinds of mathematical statements add information about the mathematical objects in question, whereas other statements do not. For instance, a symbol declaration only declares an unambiguous name for an object. We will call the following OMDoc elements **assertional** : `axiom` (it asserts central properties about an object), `type` (it asserts type properties about an object), `definition` (this asserts properties of a new object), and of course `assertion`.

The following elements are considered non-assertional: `symbol` (only a name is declared for an object), `alternative` (here the assertional content is carried by the `assertion` elements referenced in the structure-carrying attributes of `alternative`). For the elements introduced below we will discuss whether they are assertional or not in their context. In a nutshell, only statements introduced by the module ADT (see Chapter 32) will be assertional.

Chapter 14

Mathematical Examples in OMDoc

[=examples]

In mathematical practice examples play a great role, e.g. in concept formation as witnesses for definitions or as either supporting evidence, or as counter-examples for conjectures. Therefore examples are given status as primary objects in OMDoc. Conceptually, we model an example $\mathcal{E}$ as a pair $(\mathcal{W}, \mathbf{A})$, where $\mathcal{W} = (\mathcal{W}_1, \ldots, \mathcal{W}_n)$ is an n-tuple of mathematical objects and $\mathbf{A}$ is an assertion. If $\mathcal{E}$ is an example for a mathematical concept given as an OMDoc symbol $\mathbf{S}$, then $\mathbf{A}$ must be of the form $\mathbf{S}(\mathcal{W}_1, \ldots, \mathcal{W}_n)$.

If $\mathcal{E}$ is an example for a conjecture $\mathbf{C}$, then we have to consider the situation more carefully. We assume that $\mathbf{C}$ is of the form $\mathcal{Q}\mathbf{D}$ for some formula $\mathbf{D}$, where $\mathcal{Q}$ is a sequence $\mathcal{Q}_1 W_1, \ldots, \mathcal{Q}_m W_m$ of $m \geq n = \#\mathcal{W}$ quantifications of using quantifiers $\mathcal{Q}_i$ like $\forall$ or $\exists$. Now let $\mathcal{Q}'$ be a sub-sequence of $m - n$ quantifiers of $\mathcal{Q}$ and $\mathbf{D}'$ be $\mathbf{D}$ only that all the W_{i_j} such that the $\mathcal{Q}_{i_j}$ are absent from $\mathcal{Q}'$ have been replaced by $\mathcal{W}_j$ for $1 \leq j \leq n$. If $\mathcal{E} = (\mathcal{W}, \mathbf{A})$ supports $\mathbf{C}$, then $\mathbf{A} = \mathcal{Q}'\mathbf{D}'$ and if $\mathcal{E}$ is a counter-example for $\mathbf{C}$, then $\mathbf{A} = \neg\mathcal{Q}'\mathbf{D}'$.

Definition 14.0.1 OMDoc specifies this intuition in an **example** element that contains mathematical vernacular as a `h:p` elements for the description and n mathematical objects (the witnesses). It has the attributes

`example`

`for` specifying for which concepts or assertions it is an example. This is a reference to a whitespace-separated list of URI references to `symbol`, `definition`, or `assertion` elements.

`type` specifying the aspect, the value is one of `for` or `against`

`assertion` a reference to the assertion $\mathbf{A}$ mentioned above that formally states that the witnesses really form an example for the concept of assertion. In many cases even the statement of this is non-trivial and may require a proof.

`example` elements are considered non-assertional in OMDoc, since the assertional part is carried by the `assertion` element referenced in the `assertion` attribute.

Note that the list of mathematical objects in an `example` element does not represent multiple examples, but corresponds to the argument list of the symbol, they exemplify. In the example below, the symbol for monoid is a three-place relation (see the type declaration in Listing 12.1), so we have three witnesses.

Listing 14.1: An OMDoc representation of a mathematical example

```
<symbol name="strings−over"/>
<definition xml:id="strings.def" for="strings−over">. . . A* . . .</definition>
<symbol name="concat"/>
<definition xml:id="concat.def" for="concat">. . . :: . . .</definition>
<symbol name="empty−string"/>
```

```
<definition xml:id="empty−string.def" for="empty−string">... ϵ ...</definition>
...
<assertion xml:id="string.struct.monoid" type="lemma">
  <h:p>(A*, ::, ϵ) is a monoid.</h:p>
  <FMP>mon(A*, ::, ϵ)</FMP>
</assertion>
...
<example xml:id="mon.ex1" for="monoid" type="for"
        assertion="string.struct.monoid">
  <h:p>The set of strings with concatenation is a monoid.</h:p>
  <OMA id="nat−strings">
    <OMS cd="strings" name="strings"/>
    <OMS cd="setname1" name="N"/>
  </OMA>
  <OMS cd="strings" name="concat"/>
  <OMS cd="strings" name="empty−string"/>
</example>

<assertion xml:id="monoid.are.groups" type="false−conjecture">
 <h:p>Monoids are groups.</h:p>
 <FMP>∀S, o, e.mon(S, o, e) → ∃i.group(S, o, e, i)</FMP>
</assertion>

<example xml:id="mon.ex2" for="#monoids.are.groups" type="against"
        assertion="strings.isnt.group">
  <h:p>The set of strings with concatenation is not a group.</h:p>
  <OMR href="#nat−strings"/>
  <OMS cd="strings" name="strings"/>
  <OMS cd="strings" name="concat"/>
  <OMS cd="strings" name="empty−string"/>
</example>

<assertion xml:id="strings.isnt.group" type="theorem">
  <h:p>(A*, ::, ϵ) is a monoid, but there is no inverse function for it.</h:p>
</assertion>
```

In Listing 14.1 we show an example of the usage of an `example` element in OMDoc: We declare constructor symbols `strings-over`, that takes an alphabet A as an argument and returns the set A^* of stringss over A, `concat` for strings concatenation (which we will denote by $::$), and `empty-string` for the empty string ϵ. Then we state that $\mathcal{W} = (A^*, ::, \epsilon)$ is a monoid in an `assertion` with `xml:id="string.struct.monoid"`. The `example` element with `xml:id="mon.ex1"` in Listing 14.1 is an example for the concept of a monoid, since it encodes the pair $(\mathcal{W}, \mathbf{A})$ where $\mathbf{A}$ is given by reference to the assertion `string.struct.monoid` in the `assertion` attribute. Example `mon.ex2` uses the pair $(\mathcal{W}, \mathbf{A}')$ as a counter-example to the false conjecture `monoids.are.groups` using the assertion `strings.isnt.group` for $\mathbf{A}'$.

Chapter 15

Inline Statements

[=inline-statements]

Note that the infrastructure for statements introduced so far does its best to mark up the interplay of formal and informal elements in mathematical documents, and make explicit the influence of the context and their contribution to it. However, not all statements in mathematical documents can be adequately captured directly. Consider for instance the following situation, which we might find in a typical mathematical textbook.

> **Theorem 3.12***:* In a monoid M the left unit and the right unit coincide, we call it the **unit** of M.

The overt role of this text fragment is that of a mathematical theorem — as indicated by the cue word "**Theorem**", therefore we would be tempted represent it as an `omtext` element with the value `theorem` for the `type` attribute. But the relative clause is clearly a definition (the definiens is even marked in boldface). What we have here is an aggregated verbalization of two mathematical statements. In a simple case like this one, we could represent this as follows:

Listing 15.1: A Simple-Minded Representation of **Theorem 3.12**

```
<assertion type="theorem" style="display=flow">
  <h:p>In a monoid M, the left unit and the right unit coincide,</h:p>
</assertion>
<definition for="unit" style="display:flow">
  <h:p>we call it the <term role="definiendum" name="unit">unit</term> of M</h:p>
</definition>
```

But this representation remains unsatisfactory: the definition is not part of the theorem, which would really make a difference if the theorem continued after the inline definition. The real problem is that the inline definition is linguistically a phrase-level construct, while the `omtext` element is a discourse-level construct. However, as a phrase-level construct, the inline definition cannot really be taken as stand-alone, but only makes sense in the context it is presented in (which is the beauty of it; the re-use of context). With the `h:span` element and its `verbalizes`, we can do the following:

Listing 15.2: An Inline Definition

```
<assertion xml:id='unit-unique' type="theorem" >
  <h:p>In a monoid M, the left unit and the right unit coincide,
    <h:span verbalizes="#unit-def">we call it the unit of M</h:span>.</h:p>
</assertion>
<symbol name="unit"/>
<definition xml:id="unit-def" for="unit" just-by='#unit-unique'>
  <h:p>We call the (unique) element of a monoid M that acts as a left
    and right unit the <term role="definiendum" name="unit">unit</term> of M.</h:p>
</definition>
```

thus we would have the phrase-level markup in the proper place, and we would have an explicit version of the definition which is standalone[1], and we would have the explicit relation that states that the inline definition is an "abbreviation" of the standalone definition.[18] EdN:18

[1]Purists could use the CSS attribute `style` on the `definition` element with value `display:none` to hides it from the document; it might also be placed into another document altogether

[18]EdNote: we probably also need inline examples and inline assertions, see Ticket 1498 in the OMDoc TRAC.

Chapter 16

Theories as Structured Contexts

[=theories] OMDoc provides an infrastructure for mathematical theories as first-class objects that can be used to structure larger bodies of mathematics by functional aspects, to serve as a framework for semantically referencing mathematical objects, and to make parts of mathematical developments reusable in multiple contexts. The module ST presented in this chapter introduces a part of this infrastructure, which can already address the first two concerns. For the latter, we need the machinery for complex theories introduced in Part XI.

Definition 16.0.1 Theories are specified by the **theory** element in OMDoc, which has a required `xml:id` attribute for referencing the theory. Furthermore, the `theory` element can have the `cdbase` attribute that allows to specify the `cdbase` this theory uses for disambiguation on `om:OMS` elements (see Chapter 4 for a discussion). `theory`

Additional information about the theory like a title or a short description can be given in the `metadata` element. AfTer this, any top-level OMDoc element can occur, including the theory-constitutive elements introduced in Chapter 10 and Section 12.3, even `theory` elements themselves. Note that theory-constitutive elements may *only* occur in `theory` elements.

Definition 16.0.2 Theories can be structured like documents e.g. into sections and the like (see Chapter 25 for a discussion) via the **tgroup** element, which behaves exactly like the `omdoc` element introduced in Chapter 25 except that it also allows theory-constitutive elements, but does not allow a `theory` attribute, since this information is already given by the dominating `theory` element.[1] `tgroup`

Element	Attributes		M	Content
	Req.	Optional	D	
`theory`		`xml:id, class, style, cdbase, cdversion, cdrevision, cdstatus, cdurl, cdreviewdate`	+	`(`⟪*top+thc*⟫ `| imports)*`
`imports`	`from`	`id, type, class, style`	+	
`tgroup`		`xml:id, modules, type, class, style`	+	`(`⟪*top+thc*⟫`)*`
where ⟪*top+thc*⟫ stands for top-level and theory-constitutive elements				

Figure 16.1: Theories in OMDoc

16.1 Simple Inheritance

`theory` elements can contain `imports` elements (mixed in with the top-level ones) to specify inheritance: The main idea behind structured theories and specification is that not all theory-constitutive elements need to be explicitly stated in a theory; they can be inherited from other

[1]This element has been introduced to keep OMDoc validation manageable: We cannot directly use the `omdoc` element,since there is no simple, context-free way to determine whether an `omdoc` is dominated by a `theory` element.

theories. Formally, the set of theory-constitutive elements in a theory is the union of those that are explicitly specified and those that are imported from other theories. This has consequences later on, for instance, these are available for use in proofs. See Chapter 35 for details on availability of assertional statements in proofs and justifications.

`imports`

Definition 16.1.1 The meaning of the **imports** element is determined by two attributes:

`from` The value of this attribute is a URI reference that specifies the **source theory** , i.e. the theory we import from. The current theory (the one specified in the parent of the `imports` element, we will call it the **target theory**) inherits the constitutive elements from the source theory.

`type` This optional attribute can have the values `global` and `local` (the former is assumed, if the attribute is absent): We call constitutive elements **local** to the current theory, if they are explicitly defined as children, and else **inherited** . A **local import** (an `imports` element with `type="local"`) only imports the local elements of the source theory, a global import also the inherited ones.

The meaning of nested `theory` elements is given in terms of an implicit imports relation: The inner theory imports from the outer one. Thus

```
<theory xml:id="a.thy">
  <symbol name="aa"/>
  <theory xml:id="b.thy">
    <symbol name="cc"/>
    <definition xml:id="cc.def" for="cc" type="simple">
      <OMS cd="a.thy" name="af"/>
    </definition>
  </theory>
</theory>
```

is equivalent to

```
<theory xml:id="a.thy"><symbol name="aa"/></theory>
<theory xml:id="b.thy">
  <imports from="#a.thy" type="global"/>
  <symbol name="cc"/>
  <definition xml:id="cc.def" for="cc" type="simple">
     <OMS cd="a.thy" name="af"/>
  </definition>
</theory>
```

In particular, the symbol `cc` is visible only in theory `b.thy`, not in the rest of theory `a.thy` in the first representation.

Note that the inherited elements of the current theory can themselves be inherited in the source theory. For instance, in the Listing 16.2 the `left-inv` is the only local axiom of the theory `group`, which has the inherited axioms `closed`, `assoc`, `left-unit`.

In order for this import mechanism to work properly, the inheritance relation, i.e. the relation on theories induced by the `imports` elements, must be acyclic. There is another, more subtle constraint on the inheritance relation concerning multiple inheritance. Consider the situation in Listing 16.1: here theories `A` and `B` import theories with `xml:id="mythy"`, but from different URIs. Thus we have no guarantee that the theories are identical, and semantic integrity of the theory `C` is at risk. Note that this situation might in fact be totally unproblematic, e.g. if both URIs point to the same document, or if the referenced documents are identical or equivalent. But we cannot guarantee this by content markup alone, we have to forbid it to be safe.

Listing 16.1: Problematic Multiple Inheritance

```
<theory xml:id="A">
  <imports from="http://red.com/theories.omdoc#mythy"/>
</theory>
<theory xml:id="B">
  <imports from="http://blue.org/cd/all.omdoc#mythy"/>
</theory>
<theory xml:id="C"><imports from="#A"/><imports from="#B"/></theory>
```

Let us now formulate the constraint carefully, the **base URI** of an XML document is the URI that has been used to retrieve it. We adapt this to OMDoc theory elements: the base URI of an imported theory is the URI declared in the `cdbase` attribute of the `theory` element (if present) or the base URI of the document which contains it[2]. For theories that are imported along a chain of global imports, which include relative URIs, we need to employ URI normalization to compute the effective URI. Now the constraint is that any two imported theories that have the same value of the `xml:id` attribute must have the same base URI. Note that this does not imply a global unicity constraint for `xml:id` values of `theory` elements, it only means that the mapping of theory identifiers to URIs is unambiguous in the dependency cone of a theory.

In Listing 16.2 we have specified three algebraic theories that gradually build up a theory of groups importing theory-constitutive statements (symbols, axioms, and definitions) from earlier theories and adding their own content. The theory `semigroup` provides symbols for an operation `op` on a base set `set` and has the axioms for closure and associativity of `op`. The theory of monoids imports these without modification and uses them to state the `left-unit` axiom. The theory `monoid` then proceeds to add a symbol `neut` and an axiom that states that it acts as a left unit with respect to `set` and `op`. The theory `group` continues this process by adding a symbol `inv` for the function that gives inverses and an axiom that states its meaning.

Listing 16.2: A Structured Development of Algebraic Theories in OMDoc

```
<theory xml:id="semigroup">
  <symbol name="set"/><symbol name="op"/>
  <axiom xml:id="closed"> ... </axiom><axiom xml:id="assoc"> ... </axiom>
</theory>

<theory xml:id="monoid">
  <imports from="#semigroup"/>
  <symbol name="neut"/><symbol name="setstar"/>
  <axiom xml:id="left−unit">
    <h:p>neut is a left unit for op.</h:p><FMP>∀x ∈ set.op(x, neut) = x</FMP>
  </axiom>
  <definition xml:id="setstar.def" for="setstar" type="implicit">
    <h:p>·* subtracts the unit from a set </h:p><FMP>∀S.S* = S\{unit}</FMP>
  </definition>
</theory>

<theory xml:id="group">
  <imports from="#monoid"/>
  <symbol name="inv"/>
  <axiom xml:id="left−inv">
    <h:p>For every X ∈ set there is an inverse inv(X) wrt. op.</h:p>
  </axiom>
</theory>
```

The example in Listing 16.2 shows that with the notion of theory inheritance it is possible to re-use parts of theories and add structure to specifications. For instance it would be very simple to define a theory of Abelian semigroups by adding a commutativity axiom.

The set of symbols, axioms, and definitions available for use in proofs in the importing theory consists of the ones directly specified as `symbol`, `axiom`, and `definition` elements in the target theory itself (we speak of **local** axioms and definitions in this case and the ones that are inherited from the source theories via `imports` elements. Note that these symbols, axioms, and definitions (we call them **inherited**) can consist of the local ones in the source theories and the ones that are inherited there.

The local and inherited symbols, definitions, and axioms are the only ones available to mathematical statements and proofs. If a symbol is not available in the home theory (the one given by the dominating `theory` element or the one specified in the `theory` attribute of the statement), then it cannot be used since its semantics is not defined.

[2]Note that the base URI of the document is sufficient, since a valid OMDoc document cannot contain more than one `theory` element for a given `xml:id`

16.2 OMDoc Theories as Content Dictionaries

BOP:19

In Part I, we have introduced the OpenMath and Content-MathML representations for mathematical objects and formulae. One of the central concepts there was the notion that the representation of a symbol includes a pointer to a document that defines its meaning. In the OpenMath standard, these documents are identified as OpenMath content dictionaries, the MathML recommendation is not specific. In the examples above, we have seen that OMDoc documents can contain definitions of mathematical concepts and symbols, thus they are also candidates for "defining documents" for symbols. By the OpenMath2 standard [**BusCapCar:2oms04**] suitable classes of OMDoc documents can act as OpenMath content dictionaries (we call them OMDoc **content dictionaries** ; see Section 49.1). The main distinguishing feature of OMDoc content dictionaries is that they include `theory` elements with symbol declarations (see Section 12.3) that act as the targets for the pointers in the symbol representations in OpenMath and Content-MathML. The theory name specified in the `xml:id` attribute of the `theory` element takes the place of the `CDname` defined in the OpenMath content dictionary.

Furthermore, the URI specified in the `cdbase` attribute is the one used for disambiguation on `om:OMS` elements (see Chapter 4 for a discussion).

EdN:20 For instance the symbol declaration in Listing 12.1 can be referenced as[20]

```
<OMS cd="elAlg" name="monoid" cdbase="http://omdoc.org/algebra.omdoc"/>
```

if it occurs in a theory for elementary algebra whose `xml:id` attribute has the value `elAlg` and which occurs in a resource with the URI `http://omdoc.org/algebra.omdoc` or if the `cdbase` attribute of the `theory` element has the value `http://omdoc.org/algebra.omdoc`.

To be able to act as an OpenMath2 content dictionary format, OMDoc must be able to express content dictionary metadata (see Listing **??** for an example). For this, the `theory` element carries some optional attributes that allow to specify the administrative metadata of OpenMath content dictionaries.

The `cdstatus` attribute specifies the **content dictionary status** , which can take one of the following values: `official` (i.e. approved by the OpenMath Society), `experimental` (i.e. under development and thus liable to change), `private` (i.e. used by a private group of OpenMath users) or `obsolete` (i.e. only for archival purposes). The attributes `cdversion` and `cdrevision` jointly specify the **content dictionary version number** , which consists of two parts, a major **version** and a **revision** , both of which are non-negative integers. For details between the relation between content dictionary status and versions consult the OpenMath standard [**BusCapCar:2oms04**].

Furthermore, the `theory` element can have the following attributes:

`cdbase` for the content dictionary base which, when combined with the content dictionary name, forms a unique identifier for the content dictionary. It may or may not refer to an actual location from which it can be retrieved.

`cdurl` for a valid URL where the source file for the content dictionary encoding can be found.

`cdreviewdate` for the **review date** of the content dictionary, i.e. the date until which the content dictionary is guaranteed to remain unchanged.

EOP:19

[19] Old Part: The discussion here depends on the upcoming OM3 standard and MathML3 recommendation. The material is provisional on the expected outcome and may change in the future.

[20] EdNote: is this really the right cdbase?

Chapter 17

Strict Translations

We will now give the a formal[21] semantics of the ST elements in terms of strict OMDoc (see Part II).[22][23][24] EdN:21 EdN:22 EdN:23 EdN:24

pragmatic	strict
<axiom name=”⟪*n*⟫” xml:id=”⟪*i*⟫”> ⟪*body*⟫ </axiom>	<object name=”⟪*n*⟫” xml:id=”⟪*i*⟫”> ⟪*body*⟫ </object>
<symbol name=”⟪*n*⟫”> <type system=”⟪*s*⟫”>⟪*t*⟫</type> </symbol> <definition type=”simple” xml:id=”⟪*i*⟫” for=”⟪*n*⟫”> ⟪*body*⟫ </definition>	<object name=”⟪*n*⟫” xml:id=”⟪*i*⟫”> <type system=”⟪*s*⟫”>⟪*t*⟫</type> <definition>⟪*body*⟫</definition> </object>

[21]EdNote: do we really want to call it “formal”?
[22]EdNote: what do we do if there is both FMP and CMPs in an axiom?
[23]EdNote: what do we do if there is more than one symbol per definition?
[24]EdNote: what do we do for non-simple definitions

[=mtext]

Part VI

Mathematical Text (Module MTXT)

The everyday mathematical language used in textbooks, conversations, and written onto blackboards all over the world consists of a rigorous, slightly stylized version of natural language interspersed with mathematical formulae, that is sometimes called **mathematical vernacular** [1].

Element	Attributes		M	Content
	Required	Optional	D	
omtext		xml:id, type, for, class, style, verbalizes	+	h:p*
h:p		xml:id, style, class, index, verbalizes	+	⟪*math vernacular*⟫
h:span		type, for, relation, verbalizes	+	⟪*math vernacular*⟫
term	name	cd, cdbase, role, xml:id, class, style	–	⟪*math vernacular*⟫

Figure 17.1: The OMDoc Elements for Specifying Mathematical Properties

[1]The term "mathematical vernacular" was first introduced by Nicolaas Govert de Bruijn in the 1970s (see [**DeBruijn:tmv94**] for a discussion). It derives from the word "vernacular" used in the Catholic church to distinguish the language used by laymen from the official Latin.

Chapter 18

Mathematical Vernacular

OMDoc models mathematical vernacular as parsed text interspersed with content-carrying elements. Most prominently, the OpenMath objects, Content-MathML expressions, and `legacy` elements are used for mathematical objects, see Part I. The text structure is marked up with the inline fragment of XHTML 1.0 [**W3C:xhtml2000**].

In Figure 18.1 we have given an overview over the ones described in this book. The last two modules in Figure 18.1 are optional (see Chapter 48). Other (external or future) OMDoc modules can introduce further elements; natural extensions come when OMDoc is applied to areas outside mathematics, for instance computer science vernacular needs to talk about code fragments (see Chapter 44 and [**Kohlhase:codemlspec**]), chemistry vernacular about chemical formulae (e.g. represented in Chemical Markup Language [**CML:web**]).

Module	Elements	Comment	see
XHTML	`h:p` and inline Elements	extended by MTXT	[**W3C:xhtml2000**]
MOBJ	`om:OM*`, `m:*`, `legacy`	mathematical Objects	Part I
MTXT	`h:span`, `term`, `note`, `idx`, `citation`	phrase-level markup	below
DOC	`ref`, `ignore`	document structure	Part VI
EXT	`omlet`	for applets, images, ...	Definition ????

Figure 18.1: OMDoc Modules Contributing to Mathematical Vernacular

As we have explicated above, all mathematical documents state properties of mathematical objects — informally in mathematical vernacular or formally (as logical formulae), or both. OMDoc uses the `omtext` element to mark up text passages that form conceptual units, e.g. paragraphs, statements, or remarks.

Definition 18.0.1 `omtext` elements have an optional `xml:id` attribute, so that they can be cross-referenced, the intended purpose of the text fragment in the larger document context can be described by the optional attribute `type`.

Chapter 19

Rhetoric/Mathematical Roles of Text Fragments

BOP:25
This can take e.g. the values `abstract`, `introduction`, `conclusion`, `comment`, `thesis`, `antithesis`, `elaboration`, `motivation`, `evidence`, `transition` with the obvious meanings. In the last five cases `omtext` also has the extra attribute `for`, and in the last one, also an attribute `from`, since these are in reference to other OMDoc elements. EOP:25 The content of an `omtext` element is mathematical vernacular contained in a sequence of `h:p` elements. This can be preceded by a `metadata` element that can be used to specify authorship, give the passage a title, etc. (see Part VII).

We have used the `type` attribute on `omtext` to classify text fragments by their rhetoric role. This is adequate for much of the generic text that makes up the narrative and explanatory text in a mathematical textbook. But many text fragments in mathematical documents directly state properties of mathematical objects (we will call them mathematical statements; see Part IV for a more elaborated markup infrastructure). These are usually classified as definitions, axioms, etc. Moreover, they are of a form that can (in principle) be formalized up to the level of logical formula; in fact, mathematical vernacular is seen by mathematicians as a more convenient form of communication for mathematical statements that can ultimately be translated into a foundational logical system like axiomatic set theory [**Bernays:ast91**]. For such text fragments, OMDoc reserves the following values for the `type` attribute:

`axiom` (fixes or restricts the meaning of certain symbols or concepts.) An axiom is a piece of mathematical knowledge that cannot be derived from anything else we know.

`definition` (introduces new concepts or symbols.) A definition is an axiom that introduces a new symbol or construct, without restricting the meaning of others.

`example` (for or against a mathematical property).

`proof` (a proof), i.e. a rigorous — but maybe informal — argument that a mathematical statement holds.

`hypothesis` (a local assumption in a proof that will be discharged later) for text fragments that come from (parts of) proofs.

`derive` (a step in a proof), we will specify the exact meanings of this and the two above in Part X and present more structured counterparts.

For the first four values, `omtext` also provides the attribute `for`, as they point to other mathematical aspects such as symbols, assertions, definitions, axioms or alternatives.

[25] OLD PART: The rhethorical relations will be completely reworked taking the SALT ontology into account. For the moment we liberalize the RNC schema to allow `xsd:anyURI` here

Finally, OMDoc also reserves the values `theorem`, `proposition`, `lemma`, `corollary`, `postulate`, `conjecture`, `false-conjecture`, `formula`, `obligation`, `assumption`, `rule` and `assertion` for statements that assert properties of mathematical objects (see Figure 13.2 in Section 13.0 for explanations). Note that the differences between these values are largely pragmatic or proof-theoretic (conjectures become theorems once there is a proof). Further types of text can be specified by providing a URI that points to a description of the text type (much like the `definitionURL` attribute on the `m:csymbol` elements in Content-MathML).

Of course, the `type` only allows a rough classification of the mathematical statements at the text level, and does not make the underlying content structure explicit or reveals their contribution and interaction with mathematical context. For that purpose OMDoc supplies a set of specialized elements, which we will discuss in Part IV. Thus `omtext` elements will be used to give informal accounts of mathematical statements that are better and more fully annotated by the infrastructure introduced in Part IV. However, in narrative documents, we often want to be informal, while maintaining a link to the formal element. For this purpose OMDoc provides the optional `verbalizes` attribute on the `omtext` element. Its value is a whitespace-separated list of URI references to formal representations (see Chapter 14 for further discussion).

Chapter 20

Phrase-Level Markup of Mathematical Vernacular

To make the sentence-internal structure of mathematical vernacular more explicit, OMDoc provides an infrastructure to mark up natural language phrases in sentences. Linguistically, a **phrase** is a group of words that functions as a single unit in the syntax of a sentence. Examples include "noun phrases, verb phrases, or prepositional phrases". In OMDoc we use the `h:span` element from XHTML a general wrapper for sentence-level phrases that allows to mark their specific properties with special attributes and a `metadata` child. The `term` element is naturally restricted to phrases by construction.

Definition 20.0.1 The **h:span** element has the optional attribute `xml:id` for referencing the text fragment and the CSS attributes `style` and `class` to associate presentation information with it (see the discussion in Chapter 3 and ?spec@omstyle?). `h:span`

The semantics of the `h:span` element is defined by mapping to the SALT Rhetorical Ontology [**Groza:SALT07**] i.e. for example we define a `nucleus` phrase to be an instance of `http://salt.semanticauthoring.org/onto/rhetorical-ontology#nucleus`. The `type` attribute serves a linguistic purpose. A `h:span` denoting a part of a sentence that plays an important role in the understanding of the entire text or is simply basic information essential to the author's purpose takes the value of a `nucleus`. A `h:span` that plays a secondary role in the text, i.e. that serves primarily to further explain or support the `nucleus` with additional information takes the value of a `satellite`. The main difference between these two concepts is that a `nucleus` can be comprehended in a context of a text by iself, while on the other hand a `satellite` is incomprehensible without its corresponding `nucleus` phrase. In order to further clarify and annotate this dependence, if a `h:span` element has a value `satellite` for the `type` attribute, it also has the optional attributes `for` and `relation`, which are explained below.

The `relation` attribute gives the type of dependency relation connecting the `satellite` phrase with its corresponding `nucleus` phrase. It can take one of the following values: `antithesis`, `circumstance`, `concession`, `condition`, `evidence`, `means`, `preparation`, `purpose`, `cause`, `consequence`, `elaboration`, `restatement` and `solutionhood`. We go through each of these terms separately to further clarify their role and meaning.

`antithesis` is a relation where the author has a positive regard for the `nucleus`. The `nucleus` and the `satellite` are in contrast i.e. both can not be true, and the intention of the author is to increase the reader's positive regard towards the `nucleus`.

`circumstance` is a relation where the situation presented in the `satellite` is unrealized. It simply provides a framework in the subject matter within which the reader is to interpret the `nucleus`.

`concession` is a relation where the author yet again wants to increase the reader's positive regard for the `nucleus`. This time, by acknowledging a potential or apparent incompatibility between the `nucleus` and the `satellite`.

`condition` is a relation in which the `satellite` represents a hypothetical future, i.e. unrealized situation and the realization of the statement given in the `nucleus` phrase depends on the realization of the situation described in the `satellite` phrase.

`evidence` is a relation where the author wants to increase the reader's belief in the `nucleus` by providing the `satellite`, which is something that the reader believes in or will find credible.

`means` is a relation where the `satellite` represents a method or an instrument which tends to make the realization of the situation presented in the `nucleus` phrase more likely. For instance: **The Gaussian algorithm solves a linear system of equations**, by first reducing the given system to a triangular or echelon form using elementary row operations and then using a back substitution to find the solution.

`preparation` is a relation where the `satellite` precedes the `nucleus` in the text, and tends to make the reader more ready, interested or oriented for reading what is to be stated in the `nucleus` phrase.

`purpose` is a relation where the author wants the reader to recognize that the activity described in the `nucleus` phrase is initiated in order to realize what is described in the `satellite` phrase.

`cause` is a relation where the author wants the reader to recognize that the `satellite` is the cause for the action described in the `nucleus` phrase.

`consequence` is a relation where the author wants the reader to recognize that the action described in the `nucleus` is to have a result or consequences, as described in the `satellite` phrase.

`elaboration` is a relation where the `satellite` phrase provides additional detail for the `nucleus`. For instance: **In elementary number theory, integers are studied without the use of techniques from other mathematical fields.** Questions of divisibility, factorization into prime numbers, investigation of perfect numbers, use of the Euclidean algorithm to compute the GCD and congruences belong here.

`restatement` is a relation where the `satellite` simply restates what is said in the `nucleus` phrase. However the `nucleus` is more central to the authors purposes than the `satellite` is. For instance: The somewhat older term arithmetic is also used to refer to number theory, but is no longer as popular as it once was...... **Number theory used to be called "the higher arithmetic", but this is dropping out of use.**

`solutionhood` is a relation in which the `nucleus` phrase represents a solution to the problem(s) presented in the `satellite` phrase.

Listing 20.1: Phrases and their attribute usage

```
<omtext>
  <h:span id="sat1.2" type="satellite" relation="concession" for="#nuc1.1">Although it
   still shows up in the names of mathematical fields such as arithmetic functions,
  arithmetic of eliptic curves, fundamental theorem of arithmetic
  </h:span>
  <h:span id="nuc1.1" type="nucleus"> the word arithmetic is no longer as popular as it once was
  </h:span>
</omtext>
```

The `for` attribute, available when a `h:span` is denoted to be a `satellite`, is there to link the `h:span` to its corresponding `nucleus` phrase i.e. serves only for referential purposes, holding the value of the URI of the `nucleus` phrase. Thus having the phrases uniquely identified by the `xml:id` attribute is highly encouraged due to its great relevance for weaving the semantics of a text at this granularity level.

Furthermore, the `h:span` element allows the attribute `index` for parallel multilingual markup: Recall that sibling `omtext` elements form multilingual groups of text fragments. We can use the `h:span` element to make the correspondence relation on text fragments more fine-grained: `h:span` elements in sibling `omtext`s that have the same `index` value are considered to be equivalent. Of course, the value of an `index` has to be unique in the dominating `omtext` element (but not beyond). Thus the `index` attributes simplify manipulation of multilingual texts, see Listing 22.2 for an example at the discourse level.

Finally, the `h:span` element can carry a `verbalizes` attribute whose value is a whitespace-separated list of URI references that act as pointers to other OMDoc elements. This has two applications: the first is another kind of parallel markup where we can state that a phrase corresponds to (and thus “verbalizes”) a part of formula in a sibling `FMP` element.[26] EdN:26

Listing 20.2: Parallel Markup between Formal and Informal

```
<h:p>
  If <h:span verbalizes="#isaG">⟨G, ∘⟩ is a group</h:span>, then of course
     <h:span verbalizes="#isaM">it is a monoid</h:span> by construction.
</h:p>
<FMP>
  <OMA><OMS cd="logic1" name="implies"/>
    <OMA id="isaG"><OMS cd="algebra" name="group"/>
      <OMA id="GG"><OMS cd="set" name="pair">
        <OMV name="G"/><OMV name="op"/>
      </OMA>
    </OMA>
    <OMA xml:id="isaM"><OMS cd="algebra" name="monoid"/>
      <OMR href="GG"/>
    </OMA>
  </OMA>
</FMP>
```

Another important application of the `verbalizes` is the case of inline mathematical statements, which we will discuss in Chapter 14.

[26]EdNote: MK: this needs to be done differently, rework this example

Chapter 21

Technical Terms

In OMDoc we can give the notion of a **technical term** a very precise meaning: it is a span representing a concept for which a declaration exists in a content dictionary (see Section 12.0). In this respect it is the natural language equivalent for an OpenMath symbol or a Content-MathML token[1]. Let us consider an example: We can equivalently say "$0 \in \mathbb{N}$" and "the number zero is a natural number". The first rendering in a formula, we would cast as the following OpenMath object:

```
<OMA><OMS cd="set1" name="in"/>
  <OMS cd="nat" name="zero"/>
  <OMS cd="nat" name="Nats"/>
</OMA>
```

with the effect that the components of the formula are disambiguated by pointing to the respective content dictionaries. Moreover, this information can be used by added-value services e.g. to cross-link the symbol presentations in the formula to their definition (see), or to detect logical dependencies. To allow this for mathematical vernacular as well, we provide the `term` element: in our example we might use the following markup.

```
...<term cd="nat" name="zero">the number zero</term> is an
<term cd="nat" name="Nats">natural number</term>...
```

Definition 21.0.1 The **term** element has one required attribute: `name` and two optional ones: `cd` and `cdbase`. Together they determine the meaning of the phrase just like they do for `om:OMS` elements (see the discussion in Chapter 4 and Section 16.1). The `term` element also allows the attribute `xml:id` for identification of the phrase occurrence, the CSS attributes for styling and the optional `role` attribute that allows to specify the role the respective phrase plays. We reserve the value `definiens` for the defining occurrence of a phrase in a definition. This will in general mark the exact point to point to when presenting other occurrences of the same[2] phrase. Other attribute values for the `role` are possible, OMDoc does not fix them at the current time. Consider for instance the following text fragment from Figure **??** in [**Kohlhase:OMDoc1.6primer**].

`term`

> **Definition 1.** Let E be a set. A mapping of $E \times E$ is called a **law of composition** on E. The value $f(x, y)$ of f for an ordered pair $(x, y) \in E \times E$ is called the **composition of** x and y under this law. A set with a law of composition is called a magma.

Here the first boldface term is the definiens for a "law of composition", the second one for the result of applying this to two arguments. It seems that this is not a totally different concept that is defined here, but is derived systematically from the concept of a "law of composition" defined before. Pending a thorough linguistic investigation we will mark up such occurrences with `definiens-applied`, for instance in

[1]and is subject to the same visibility and scoping conditions as those; see Chapter 15 for details

[2]We understand this to mean with the same `cd` and `name` attributes.

Listing 21.1: Marking up the Technical Terms

```
Let $E$ be a set. A mapping of $E \times E$ is called a
<term cd="magmas" name="law_of_comp" role="definiendum">law of composition</term> on $E$.
The value $f(x,y)$ of $f$ for an ordered pair $(x,y) \in E \times E$ is called the
<term cd="magmas"name="law_of_comp" role="definiendum−applied">composition of</term>
$x$ and $y$ under this law.
```

There are probably more such systematic correlations; we leave their categorization and modeling in OMDoc to the future.

Chapter 22

Index and Bibliography

[27] EdN:27

Element	Attributes		MD	Content
idx		(xml:id|xref)	−	idt?, ide+
ide		index, sort-by, see, seealso, links	−	idp*
idt		style, class	−	⟨⟨*math vernacular*⟩⟩
idp		sort-by, see, seealso, links	−	⟨⟨*math vernacular*⟩⟩
note		type, xml:id, style, class, index, verbalizes	+	⟨⟨*math vernacular*⟩⟩
citation	ref		text	

Figure 22.1: Rich Text Format OMDoc

Definition 22.0.1 (Index Markup) The **idx** element is used for index markup in OMDoc. It contains an optional **idt** element that contains the index text, i.e. the phrase that is indexed. The remaining content of the index element specifies what is entered into various indexes. For every index this phrase is registered to there is one **ide** element (index entry); the respective entry is specified by name in its optional `index` attribute. The `ide` element contains a sequence of index phrases given in **idp** elements. The content of an `idp` element is regular mathematical text. Since index entries are usually sorted, (and mathematical text is difficult to sort), they carry an attribute `sort-by` whose value (a sequence of Unicode characters) can be sorted lexically [**Unicode:collation**]. Moreover, each `idp` and `ide` element carries the attributes `see`, `seealso`, and `links`, that allow to specify extra information on these. The values of the first ones are references to `idx` elements, while the value of the `links` attribute is a whitespace-separated list of (external) URI references. The formatting of the index text is governed by the attributes `style` and `class` on the `idt` element. The `idx` element can carry either an `xml:id` attribute (if this is the defining occurrence of the index text) or an `xref` attribute. In the latter case, all the `ide` elements from the defining `idx` (the one that has the `xml:id` attribute) are imported into the referring `idx` element (the one that has the `xref` attribute).

idx idt ide idp

Listing 22.1: An Example of Rich Text Structure

```
<omtext>
  <h:p style="color:red" xml:id="p1">All <idx><idt>animals are dangerous</idt>
    <idp>dangerous</idp><idp seealso="creature">animal</idp></idx>!
    (which is a highly <em>unfounded</em> statement).
    In reality only some animals are, for instance:</h:p>
  <h:ul id="l1">
    <h:li>sharks (they bite) and </h:li>
    <h:li>bees (they sting).</h:li>
  </h:ul>
  <h:p>If we measure danger by the number of deaths, we obtain</h:p>
  <table xmlns="http://www.w3.org/1999/xhtml">
```

[27] EdNote: introduce idx and citation, and also describe makeindex and bibliography in doc!

```
    <tr>              <th>Culprits</th> <th>Deaths</th> <th>Action</th></tr>
    <tr>              <td>sharks</td> <td>312</td> <td>bite</td></tr>
    <tr xml:id="bn"> <td>bees</td> <td>23</td> <td>sting</td></tr>
    <tr>              <td>cars</td>  <td>7500</td> <td>crash</td></tr>
  </table>
  <h:p>So, if we do the numbers <note xml:id="n1" type="ednote">check the
  numbers again</note> we see that animals are dangerous, but they are
  less so than cars but much more photogenic as we can see
  <h:a href="http://www.yellowpress.com/killerbee.jpg">here</h:a>.</h:p>

  <note type="footnote">From the International Journal of Bee−keeping; numbers only
  available for 2002.</note>
</omtext>
```

note

Definition 22.0.2 (Notes) The **note** element is the closest approximation to a footnote or endnote, where the kind of note is determined by the `type` attribute. OMDoc supplies `footnote` as a default value, but does not restrict the range of values. Its `for` attribute allows it to be attached to other OMDoc elements externally where it is not allowed by the OMDoc document type. In our example, we have attached a footnote by reference to a table row, which does not allow `note` children.

BOP:28

All elements in the RT module carry an optional `xml:id` attribute for identification and an `index` attribute for parallel multilingual markup (e.g. Chapter 19 for an explanation and Listing 22.2 for a translation example).

Listing 22.2: Multilingual Parallel Markup

```
<docalt xml:id="animals.overview">
  <omtext>
    <h:p index="intro">Consider the following animals:</h:p>
    <h:ul index="animals">
      <h:li index="first">a tiger,</h:li>
      <h:li index="second">a dog.</h:li>
    </h:ul>
  </omtext>
  <omtext xml:lang="de">
    <h:p index="intro">Betrachte die folgenden Tiere:</h:p>
    <h:ul index="animals">
      <h:li index="first">Ein Tiger</h:li>
      <h:li index="second">Ein Hund</h:li>
    </h:ul>
  </omtext>
</docalt>
```

EOP:28

[28] Old Part: do we want to support parallel path markup in the future? It has not been used that much. Also, there is another place where this is explained.

Part VII

Document Infrastructure (Module DOC)

[=document-types] Mathematical knowledge is largely communicated by way of a specialized set of documents (e.g. e-mails, letters, pre-prints, journal articles, and textbooks). These employ special notational conventions and visual representations to convey the mathematical knowledge reliably and efficiently.

When marking up mathematical knowledge, one always has the choice whether to mark up the structure of the document itself, or the structure of the mathematical knowledge that is conveyed in the document. Even though in most documents, the document structure is designed to help convey the structure of the knowledge, the two structures need not be the same. To frame the discussion we will distinguish two aspects of mathematical documents. In the *knowledge-centered view* we organize the mathematical knowledge by its function, and do not care about a way to present it to human recipients. In the *narrative-centered view* we are interested in the structure of the argument that is used to convey the mathematical knowledge to a human user.

We will call a document **knowledge-structured** and **narrative-structured** , based on which of the two aspects is prevalent in the organization of the material. Narrative-structured documents in mathematics are generally directed at human consumption (even without being in presentation markup). They have a general narrative structure: text interleaving with formal elements like assertions, proofs, ... Generally, the order of presentation plays a role in their effectiveness as a means of communication. Typical examples of this class are course materials or introductory textbooks. Knowledge-structured documents are generally directed at machine consumption or for referencing. They do not have a linear narrative spine and can be accessed randomly and even re-ordered without information loss. Typical examples of these are formula collections, OpenMath content dictionaries, technical specifications, etc.

The distinction between knowledge-structured and narrative-structured documents is reminiscent of the presentation vs. content distinction discussed in , but now it is on the level of document structure. Note that mathematical documents are often in both categories: a mathematical textbook can be read from front to end, but it can also be used as a reference, accessing it by the index and the table of contents. The way humans work with knowledge also involves a change of state. When we are taught or explore a mathematical domain, we have a linear/narrative path through the material, from which we abstract more and more, finally settling for a semantic representation that is relatively independent from the path we acquired it by. Systems like ActiveMath (see [**Kohlhase:OMDoc1.6projects**]) use the OMDoc format in exactly that way playing on the difference between the two classes and generating narrative-structured representations from knowledge-structured ones on the fly.

So, maybe the best way to think about this is that the question whether a document is narrative- or knowledge-structured is not a property of the document itself, but a property of the application processing this document.

OMDoc provides markup infrastructure for both aspects. In this chapter, we will discuss the infrastructure for the narrative aspect — for a working example we refer the reader to [**Kohlhase:OMDoc1.6primer**]. We will look at markup elements for knowledge-structured documents in Chapter 15.

Even though the infrastructure for narrative aspects of mathematical documents is somewhat presentation-oriented, we will concentrate on content-markup for it. In particular, we will not concern ourselves with questions like font families, sizes, alignment, or positioning of text fragments. Like in most other XML applications, this kind of information can be specified in the CSS `style` and `class` attributes described in Chapter 3.

Chapter 23

The Document Root

[=omdoc-root]

Definition 23.0.1 The XML root element of the OMDoc format is the **omdoc** element, it contains all other elements described here. We call an OMDoc element a **top-level element** , if it can appear as a direct child of the `omdoc` element. The `omdoc` element has an optional attribute `xml:id` that can be used to reference the whole document. The optional attribute `version` is used to specify the version of the OMDoc format the contens of the element conforms to. It is fixed to the string `1.6` by this specification. This will prevent validation with a different version of the DTD or schema, or processing with an application using a different version of the OMDoc specification. The (optional) attribute `modules` allows to specify the OMDoc modules that are used in this element. The value of this attribute is a whitespace-separated list of module identifiers (e.g. MOBJ the left column in Figure 2.1), OMDoc sub-language identifiers (see Figure 49.1), or URI references for externally given OMDoc modules or sub-language identifiers.[1] The intention is that if present, the `modules` specifies the list of all the modules used in the document (fragment). If a `modules` attribute is present, then it is an error, if the content of this element contains elements from a module that is not specified; spurious module declarations in the `modules` attributes are allowed.

omdoc

Here and in the following we will use tables as the one in Figure 23.1 to give an overview over the respective OMDoc elements described in a chapter or section. The first column gives the element name, the second and third columns specify the required and optional attributes. We will use the fourth column labeled "MD" to indicate whether an OMDoc element can have a `metadata` child (see Definition ????), which will be described in the next section. Finally the fifth column describes the content model — i.e. the allowable children — of the element. For this, we will use a form of Backus Naur form notation also used in the DTD: `#PCDATA` stands for "parsed character data", i.e. text intermixed with legal OMDoc elements.) A synopsis of all elements is provided in ?spec@table?.

Element	Attributes		M	Content
	Required	Optional	D	
omdoc		xml:id, type, class, style, version, modules	+	(⟪*top-level*⟫)*
ref	xref	type, class, style	–	
ignore		type, comment	–	ANY
where ⟪*top-level*⟫ stands for top-level OMDoc elements				

Figure 23.1: OMDoc Elements for Specifying Document Structure.

[1]Allowing these external module references keeps the OMDoc format extensible. Like in the case with namespace URIs OMDoc do not mandate that these URI references reference an actual resource. They merely act as identifiers for the modules.

Chapter 24

Metadata

[=metadata]

The World Wide Web was originally built for human consumption, and although everything on it is machine-readable, most of it is not machine-understandable. The accepted solution is to provide metadata (data about data) to describe the documents on the web in a machine-understandable format that can be processed automatically. Metadata commonly specifies aspects of a document like title, authorship, language usage, and administrative aspects like modification dates, distribution rights, and identifiers.

In general, metadata can either be embedded in the respective document, or be stated in a separate one. The first facilitates maintenance and control (metadata is always at your fingertips, and it can only be manipulated by the document's authors), the second one enables inference and distribution. OMDoc allows to embed metadata into the document, from where it can be harvested for external metadata formats, such as the XML resource description format (RDF [**LasSwi:rdf99**]). We use one of the best-known metadata schemata for documents – the *Dublin Core* (cf. Part VII and Section 31.0). The purpose of annotating metadata in OMDoc is to facilitate the administration of documents, e.g. digital rights management, and to generate input for metadata-based tools, e.g. RDF-based navigation and indexing of document collections. Unlike most other document formats OMDoc allows to add metadata at many levels, also making use of the metadata for document-internal markup purposes to ensure consistency.

Definition 24.0.1 The **metadata** element contains elements for various metadata formats including bibliographic data from the Dublin Core vocabulary (as mentioned above), licensing information from the Creative Commons Initiative (see Part VIII), as well as information for OpenMath content dictionary management. Application-specific metadata elements can be specified by adding corresponding OMDoc modules that extend the content model of the `metadata` element. `metadata`

The OMDoc `metadata` element can be used to provide information about the document as a whole (as the first child of the `omdoc` element), as well as about specific fragments of the document, and even about the top-level mathematical elements in OMDoc. This reinterpretation of bibliographic metadata as general data about knowledge items allows us to extract document fragments and re-assemble them to new aggregates without losing information about authorship, source, etc.

Chapter 25

Document Comments

[=comments]

Many content markup formats rely on commenting the source for human understanding; in fact source comments are considered a vital part of document markup. However, as XML comments (i.e. anything between "`<!--`" and "`-->`" in a document) need not even be read by some XML parsers, we cannot guarantee that they will survive any XML manipulation of the OMDoc source.

Therefore, anything that would normally go into comments should be modeled with an `omtext` element (`type comment`, if it is a text-level comment; see Chapter 18) or with the `ignore` element for persistent comments, i.e. comments that survive processing.

Definition 25.0.1 The content of the **ignore** element can be any well-formed OMDoc, it can occur as an OMDoc top-level element or inside mathematical texts (see Part V). `ignore`

This element should be used if the author wants to comment the OMDoc representation, but the end user should not see their content in a final presentation of the document, so that OMDoc text elements are not suitable, e.g. in

```
<ignore type="todo" comment="this does not make sense yet, rework">
  <assertion xml:id="heureka">...</assertion>
</ignore>
```

Of course, `ignore` elements can be nested, e.g. if we want to mark up the comment text (a pure string as used in the example above is not enough to express the mathematics). This might lead to markup like

```
<ignore type="todo" comment="rework">
  <ignore type="todo−comment">
    <h:p>This does not make sense yet, in particular, the equation
      ... cannot be true, think of ...
    </h:p>
  </ignore>
  <assertion xml:id="heureka">...</assertion>
</ignore>
```

BOP:29

Example 25.0.2 Another good use of the `ignore` element is to use it as an analogon to the in-place error markup in OpenMath objects (see Section 5.1). In this case, we use the `type` attribute to specify the kind of error and the content for the faulty OMDoc fragment. Note that since the whole object must be XML valid. As a consequence, the `ignore` element can only be used for "mathematical errors" like sibling `CMP` or `FMP` elements that do not have the same meaning as in Listing 25.1. XML-well-formedness and validity errors will have to be handled by the XML tools involved.

Listing 25.1: Marking up Mathematical Errors Using `ignore`

[29] Old Part: MK: cannot make this example any more, think of a new one

```
<ignore type="CMP−lang−error"
        comment="multilingual CMPs are not translations of each other">
  <assertion xml:id="ass1">
    <CMP>The proof is trivial</CMP>
    <CMP xml:lang="de">Der Beweis ist extrem schwer</CMP>
  </assertion>
</ignore>
```

For another use of the `ignore` element, see Figure 27.1 in Chapter 26.

EOP:29

Chapter 26

Document Structure

[=sectioning]

Like other documents mathematical ones are often divided into units like chapters, sections, and paragraphs by tags and nesting information. OMDoc makes these document relations explicit by using the `omdoc` element with an optional attribute `type`. It can take the values

`sequence` for a succession of paragraphs. This is the default, and the normal way narrative texts are built up from paragraphs, mathematical statements, figures, etc. Thus, if no `type` is given the type `sequence` is assumed.

`itemize` for unordered lists. The children of this type of `omdoc` will usually be presented to the user as indented paragraphs preceded by a bullet symbol. Since the choice of this symbol is purely presentational, OMDoc use the CSS `style` or `class` attributes on the children to specify the presentation of the bullet symbols (see Chapter 3).

`enumeration` for ordered lists. The children of this type of `omdoc` are usually presented like unordered lists, only that they are preceded by a running number of some kind (e.g. "1.", "2."... or "a)", "b)"...; again the `style` or `class` attributes apply).

`sectioning` The children of this type of `omdoc` will be interpreted as sections. This means that the children will be usually numbered hierarchically, and their metadata will be interpreted as section heading information. For instance the `metadata/dc:title` information (see Part VII for details) will be used as the section title. Note that OMDoc does not provide direct markup for particular hierarchical levels like "chapter", "section", or "paragraph", but assumes that these are determined by the application that presents the content to the human or specified using the CSS attributes.

Other values for the `type` attribute are also admissible, they should be URI references to documents explaining their intension.

We consider the `omdoc` element as an implicit `omdoc`, in order to allow plugging together the content of different OMDoc documents as `omdoc`s in a larger document. Therefore, all the attributes of the `omdoc` element also appear on `omdoc` elements and behave exactly like those.

[=sharing]

Chapter 27

Sharing and Referring to Document Parts

Definition 27.0.1 As the document structure need not be a tree in hypertext documents, `omdoc` elements also allow empty **ref** elements whose `xref` attribute can be used to reference OMDoc elements defined elsewhere. The optional `xml:id` (its value must be document-unique) attribute identifies it and can be used for building reference labels for the included parts. Even though this attribute is optional, it is highly recommended to supply it. The `type` attribute can be used to describe the reference type. Currently OMDoc supports two values: `include` (the default) for in-text replacement and `cite` for a proper reference. The first kind of reference requires the OMDoc application to process the document as if the `ref` element were replaced with the OMDoc fragment specified in the `xref`. The processing of the type `cite` is application specific. It is recommended to generate an appropriate label and (optionally) supply a hyper-reference. There may be more supported values for `type` in time. `ref`

BOP:30

Let R be a `ref` element of type `include`. We call the element the URI in the `xref` points to its **target** unless it is an `omdoc` element; in this case, the target is an `omdoc` element which has the same children as the original `omdoc` element[1]. We call the process of replacing a `ref` element by its target in a document **ref reduction** and the document resulting from the process of systematically and recursively reducing all the `ref` elements the **ref-normal form** of the source document. Note that `ref`-normalization may not always be possible, e.g. if the ref-targets do not exist or are inaccessible — or worse yet, if the relation given by the `ref` elements is cyclic. Moreover, even if it is possible to `ref`-normalize, this may not lead to a valid OMDoc document, e.g. since `ID` type attributes that were unique in the target documents are no longer in the `ref`-reduced one. We will call a document **ref reducible** , iff its `ref`-normal form exists, and **ref valid** , iff the `ref` normal form exists and is a valid OMDoc document. EOP:30

Note that it may make sense to use documents that are not `ref`-valid for narrative-centered documents, such as courseware or slides for talks that only allude to, but do not fully specify the knowledge structure of the mathematical knowledge involved. For instance the slides discussed in [**Kohlhase:OMDoc1.6primer**] do not contain the `theory` elements that would be needed to make the documents `ref`-valid.

The `ref` elements also allow to "flatten" the tree structure in a document into a list of leaves and relation declarations (see Figure 27.1 for an example). It also makes it possible to have more than one view on a document using `omdoc` structures that reference a shared set of OMDoc elements. Note that we have embedded the `ref`-targets of the top-level `omdoc` element into an

[30] Old Part: reconsider this, we may change the omgroup element to omdoc

[1] This transformation is necessary, since OMDoc does not allow to nest `omdoc` elements, which would be the case if we allowed verbatim replacement for `omdoc` elements. As we have stated above, the `omdoc` has an implicit `omdoc` element, and thus behaves like one.

`ignore` comment, so that an OMDoc transformation (e.g. to text form) does not encounter the same content twice.

```
<omdoc xml:id="text"
         type="sequence">
  <omtext xml:id="t1">$T_1$</omtext>
  <omdoc xml:id="enum"
            type="enumeration">
    <omtext xml:id="t2">$T_2$</omtext>
    <omtext xml:id="t3">$T_3$</omtext>
  </omdoc>
  <omtext xml:id="t4">$T_4$</omtext>
</omdoc>
```

```
<omdoc xml:id="text" type="sequence">
  <ref xref="#t1"/>
  <ref xref="#enum"/>
  <ref xref="#t4"/>
</omdoc>

<ignore type="targets"
         comment="already referenced">
  <omtext xml:id="t1">$T_1$</omtext>
  <omtext xml:id="t2">$T_2$</omtext>
  <omtext xml:id="t3">$T_3$</omtext>
  <omtext xml:id="t4">$T_4$</omtext>

  <omdoc xml:id="enum"
            type="enumeration">
    <ref xref="#t2"/>
    <ref xref="#t3"/>
  </omdoc>
</ignore>
```

Figure 27.1: Flattening a Tree Structure

While the OMDoc approach to specifying document structure is a much more flexible (database-like) approach to representing structured documents[2] than the tree model, it puts a much heavier load on a system for presenting the text to humans. In essence the presentation system must be able to recover the left representation from the right one in Figure 27.1. Generally, any OMDoc element defines a fragment of the OMDoc it is contained in: everything between the start and end tags and (recursively) those elements that are reached from it by following the cross-references specified in `ref` elements. In particular, the text fragment corresponding to the element with
EdN:31 `xml:id="text"` in the right OMDoc of Figure 27.1 is just the one on the left.[31]

In Chapter 3 we have introduced the CSS attributes `style` and `class`, which are present on all OMDoc elements. In the case of the `ref` element, there is a problem, since the content of these can be incompatible. In general, the rule for determining the style information for an element is that we treat the replacement element as if it were a child of the `ref` element, and then determine the values of the CSS properties of the `ref` element by inheritance.

[=docalt]

[2]The simple tree model is sufficient for simple markup of existing mathematical texts and to replay them verbatim in a browser, but is insufficient e.g. for generating individualized presentations at multiple levels of abstractions from the representation. The OMDoc text model — if taken to its extreme — allows to specify the respective role and contributions of smaller text units, even down to the sub-sentence level, and to make the structure of mathematical texts machine-understandable. Thus, an advanced presentation engine like the ActiveMath system [**SieBen:acgap00**] can — for instance — extract document fragments based on the preferences of the respective user.

[31]EdNote: make the model of first normalizing and then presentation and what this entais. cf. the TRAC issues.

Chapter 28

Abstract Documents

Definition 28.0.1 To be able to support abstract documents that can be concretized, OMDoc supplies the **docalt** [32] element that groups alternative document fragments so that the presentation process can chose among them.

docalt

EdN:32

Example 28.0.2 One very simple example is to group language variants[1] using the optional `xml:lang` attribute to specify the language they are written in. Conforming with the XML recommendation, we use the ISO 639 two-letter country codes (`de` $\widehat{=}$ German, `en` $\widehat{=}$ English, `fr` $\widehat{=}$ French, `nl` $\widehat{=}$ Dutch, ...). If no `xml:lang` is given, then `en` is assumed as the default value.

Listing 28.1: A Multilingual Group of `CMP` Elements

```
<omtext>
  Let <om:OMV id="set"name="V"/> be a set.
  A <term role="definiendum">unary operation</term> on
  <om:OMR href="#set"/> is a function <om:OMV id="fun" name="F"/> with
 <om:OMA id="im">
    <om:OMS cd="relations1" name="eq"/>
    <om:OMA><om:OMS cd="fns1" name="domain"/><om:OMV name="F"/></om:OMA>
    <om:OMV name="V"/>
  </om:OMA>
  and
 <om:OMA id="ran">
    <om:OMS cd="relations1" name="eq"/>
    <om:OMA><om:OMS cd="fns1" name="range"/><om:OMV name="F"/></om:OMA>
    <om:OMV name="V"/>
  </om:OMA>.
</omtext>
<omtext xml:lang="de">
  Sei <om:OMR href="#set"/> eine Menge.
  Eine <term role="definiendum">unäre Operation</term>
  ist eine Funktion <om:OMR href="#fun"/>, so dass
  <om:OMR href="#im"/> und <om:OMR href="#ran"/>.
</omtext>
<omtext xml:lang="fr">
  Soit <om:OMR href="#set"/> un ensemble.
  Une <term role="definiendum">opération unaire</term> sûr
  <om:OMR href="#set"/> est une fonction <om:OMR href="#fun"/>
  avec <om:OMR href="#im"/> et <om:OMR href="#ran"/>.
</omtext>
```

Listing 28.1 shows an example of a multilingual group. Here, the OpenMath extension by DAG representation (see Chapter 4) facilitates multi-language support: Only the language-dependent parts of the text have to be rewritten, the (language-independent) formulae can simply be re-used by cross-referencing.

[32]EdNote: name is provisional, Christine will develop this more fully

[1]i.e. all the document fragments in this group are direct translations of each other.

Chapter 29

Frontmatter and Backmatter

[33] EdN:33

[33]EdNote: describe index and tableofcontents, and backport it to OMDoc1.3

[=dc-elements]

Part VIII

Dublin Core Metadata in OMDoc

The most commonly used metadata standard is the Dublin Core vocabulary, which is supported in some form by most formats. OMDoc uses this vocabulary for compatibility with other metadata applications and extends it for document management purposes in OMDoc. Most importantly OMDoc extends the use of metadata from documents to other (even mathematical) elements and document fragments to ensure a fine-grained authorship and rights management. BNP:34

[34] New Part: MK@CL, please discuss

Chapter 30

Dublin Core Ontology

Chapter 31

Pragmatic Dublin Core Elements

In the following we will describe the variant of Dublin Core metadata elements used in OMDoc. Here, the `metadata` element can contain any number of instances of any Dublin Core elements described below in any order. In fact, multiple instances of the same element type (multiple `dc:creator` elements for example) can be interspersed with other elements without change of meaning. OMDoc extends the Dublin Core framework with a set of roles (from the MARC relator set [**Marc:relators03**]) on the authorship elements and with a rights management system based on the Creative Commons Initiative.

The descriptions in this section are adapted from [**DCMI:dmt03**], and augmented for the application in OMDoc where necessary. All these elements live in the Dublin Core namespace `http://purl.org/dc/elements/1.1/`, for which we traditionally use the namespace prefix`dc:`.

Element	Attributes		Content
	Req.	Optional	
dc:creator		xml:id, class, style, role	ANY
dc:contributor		xml:id, class, style, role	ANY
dc:title		xml:lang	⟪*math vernacular*⟫
dc:subject		xml:lang	⟪*math vernacular*⟫
dc:description		xml:lang	⟪*math vernacular*⟫
dc:publisher		xml:id, class, style	ANY
dc:date		action, who	ISO 8601
dc:type			fixed: "Dataset" or "Text"
dc:format			fixed: "application/omdoc+xml"
dc:identifier		scheme	ANY
dc:source			ANY
dc:language			ISO 639
dc:relation			ANY
dc:rights			ANY
for ⟪*math vernacular*⟫ see Part V			

Figure 31.1: Dublin Core Metadata in OMDoc

Definition 31.0.1 (Titles) The title of the element — note that OMDoc metadata can be specified at multiple levels, not only at the document level, in particular, the Dublin Core **dc:title** element can be given to assign a title to a theorem, e.g. the "Substitution Value Theorem". `dc:title`

The `dc:title` element can contain mathematical vernacular (see Part V). Multiple `dc:title` elements inside a `metadata` element are assumed to be translations of each other.[35] EdN:35

Definition 31.0.2 (Creators) A primary creator or author of the publication. Additional contributors whose contributions are secondary to those listed in **dc:creator** elements should be named in `dc:contributor` elements. Documents with multiple co-authors should provide multiple `dc:creator` elements, each containing one author. The order of `dc:creator` elements is presumed to define the order in which the creators' names should be presented. `dc:creator`

[35] EdNote: do we still want this?

As markup for names across cultures is still un-standardized, OMDoc recommends that the content of a `dc:creator` element consists in a single name (as it would be presented to the user). The `dc:creator` element has an optional attribute `dc:id` so that it can be cross-referenced and a `role` attribute to further classify the concrete contribution to the element. We will discuss its values in Section 31.0.

Definition 31.0.3 (Contributors) A party whose contribution to the publication is secondary to those named in `dc:creator` elements. Apart from the significance of contribution, the semantics of the **dc:contributor** is identical to that of `dc:creator`, it has the same restriction content and carries the same attributes plus a `dc:lang` attribute that specifies the target language in case the contribution is a translation.

`dc:contributor`

Definition 31.0.4 (Subjects) This element contains an arbitrary phrase or keyword, the attribute `dc:lang` is used for the language. Multiple instances of the **dc:subject** element are supported per `dc:lang` for multiple keywords.

`dc:subject`

Definition 31.0.5 (Descriptions) A text describing the containing element's content; the attribute `dc:lang` is used for the language. As description of mathematical objects or OMDoc fragments may contain formulae, the content of this element is of the form "mathematical text" described in Part V.

Definition 31.0.6 (Publishers) The entity for making the document available in its present form, such as a publishing house, a university department, or a corporate entity. The **dc:publisher** element only applies if the `metadata` is a direct child of the root element (`omdoc`) of a document.

`dc:publisher`

Definition 31.0.7 (Dates) The date and time a certain action was performed on the element that contains this. The content is in the format defined by XML Schema data type `dateTime` (see [**BirMal:XMLSchema:Datatypes**] for a discussion), which is based on the ISO 8601 norm for dates and times.

Concretely, the format is ⟪*YYYY*⟫-⟪*MM*⟫-⟪*DD*⟫T⟪*hh*⟫:⟪*mm*⟫:⟪*ss*⟫ where ⟪*YYYY*⟫ represents the year, ⟪*MM*⟫ the month, and ⟪*DD*⟫ the day, preceded by an optional leading "`-`" sign to indicate a negative number. If the sign is omitted, "`+`" is assumed. The letter "`T`" is the date/time separator and ⟪*hh*⟫, ⟪*mm*⟫, ⟪*ss*⟫ represent hour, minutes, and seconds respectively. Additional digits can be used to increase the precision of fractional seconds if desired, i.e the format ⟪*ss*⟫.⟪*sss...*⟫ with any number of digits after the decimal point is supported. The **dc:date** element has the attributes `action` and `who` to specify who did what: The value of `who` is a reference to a `dc:creator` or `dc:contributor` element and `action` is a keyword for the action undertaken. Recommended values include the short forms `updated`, `created`, `imported`, `frozen`, `review-on`, `normed` with the obvious meanings. Other actions may be specified by URIs pointing to documents that explain the action.

`dc:date`

Definition 31.0.8 (Types) Dublin Core defines a vocabulary for the document types in [**DCMI:dtv03**]. The best fit values for OMDoc are

`Dataset` defined as "*information encoded in a defined structure (for example lists, tables, and databases), intended to be useful for direct machine processing.*"

`Text` "*a resource whose content is primarily words for reading. For example – books, letters, dissertations, poems, newspapers, articles, archives of mailing lists. Note that facsimiles or images of texts are still of the genre text.*"

`Collection` defined as "*an aggregation of items. The term collection means that the resource is described as a group; its parts may be separately described and navigated*".

`dc:type`

The more appropriate should be selected for the element that contains the **dc:type** . If it consists mainly of formal mathematical formulae, then `Dataset` is better, if it is mainly given as text, then `Text` should be used. More specifically, in OMDoc the value `Dataset` signals that the order of children in the parent of the `metadata` is not relevant to the meaning. This is the case for instance in formal developments of mathematical theories, such as the specifications in Part XI.

Definition 31.0.9 (Formats) The physical or digital manifestation of the resource. Dublin Core suggests using MIME types [**FreBor:MIME96**]. Following [**MurLau:xmt01**] we fix the content of the **dc:format** element to be the string `application/omdoc+xml` as the MIME type for OMDoc. `dc:format`

Definition 31.0.10 (Identifiers) A string or number used to uniquely identify the element. The **dc:identifier** element should only be used for public identifiers like ISBN or ISSN numbers. The numbering scheme can be specified in the `scheme` attribute. `dc:identifier`

Definition 31.0.11 (Sources) Information regarding a prior resource from which the publication was derived. We recommend using either a URI or a scientific reference including identifiers like ISBN numbers for the content of the **dc:source** element. `dc:source`

Definition 31.0.12 (Relations) Relation of this document to others. The content model of the **dc:relation** element is not specified in the OMDoc format. `dc:relation`

Definition 31.0.13 (Languages) If there is a primary language of the document or element, this can be specified here. The content of the **dc:language** element must be an ISO 639 norm two-letter language specifier, like `en` $\widehat{=}$ English, `de` $\widehat{=}$ German, `fr` $\widehat{=}$ French, `nl` $\widehat{=}$ Dutch, ... `dc:language`

Definition 31.0.14 (Rights) Information about rights held in and over the document or element content or a reference to such a statement. Typically, a **dc:rights** element will contain a rights management statement, or reference a service providing such information. `dc:rights` information often encompasses Intellectual Property rights (IPR), Copyright, and various other property rights. If the `dc:rights` element is absent (and no `dc:rights` information is inherited), no assumptions can be made about the status of these and other rights with respect to the document or element. `dc:rights`

OMDoc supplies specialized elements for the Creative Commons licenses to support the sharing of mathematical content. We will discuss them in Part VIII.

Note that Dublin Core also defines a `Coverage` element that specifies the place or time which the publication's contents addresses. This does not seem appropriate for the mathematical content of OMDoc, which is largely independent of time and geography.

31.1 Roles in Dublin Core Elements

Because the Dublin Core metadata fields for `dc:creator` and `dc:contributor` do not distinguish roles of specific parties (such as author, editor, and illustrator), we will follow the Open eBook specification [**OpenEBook:oeps99**] and use an optional `role` attribute for this purpose, which is adapted for OMDoc from the MARC relator code list [**Marc:relators03**].

`aut` (author) Use for a person or corporate body chiefly responsible for the intellectual content of an element. This term may also be used when more than one person or body bears such responsibility.

`ant` (bibliographic/scientific antecedent) Use for the author responsible for a work upon which the element is based.

`clb` (collaborator) Use for a person or corporate body that takes a limited part in the elaboration of a work of another author or that brings complements (e.g., appendices, notes) to the work of another author.

`edt` (editor) Use for a person who prepares a document not primarily his/her own for publication, such as by elucidating text, adding introductory or other critical matter, or technically directing an editorial staff.

`ths` (thesis advisor) Use for the person under whose supervision a degree candidate develops and presents a thesis, memoir, or text of a dissertation.

`trc` (transcriber) Use for a person who prepares a handwritten or typewritten copy from original material, including from dictated or orally recorded material. This is also the role (on the `dc:creator` element) for someone who prepares the OMDoc version of some mathematical content.

`trl` (translator) Use for a person who renders a text from one language into another, or from an older form of a language into the modern form. The target language can be specified by `dc:lang`.

As OMDoc documents are often used to formalize existing mathematical texts for use in mechanized reasoning and computation systems, it is sometimes subtle to specify authorship. We will discuss some typical examples to give a guiding intuition. Listing 31.1 shows metadata for a situation where editor R gives the sources (e.g. in LaTeX) of an element written by author A to secretary S for conversion into OMDoc format.

Listing 31.1: A Document with Editor (`edt`) and Transcriber (`trc`)

```
<metadata>
  <dc:title>The Joy of Jordan C* Triples</dc:title>
  <dc:creator role="aut">A</dc:creator>
  <dc:contributor role="edt">R</dc:contributor>
  <dc:contributor role="trc">S</dc:contributor>
</metadata>
```

In Listing 31.2 researcher R formalizes the theory of natural numbers following the standard textbook B (written by author A). In this case we recommend the first declaration for the whole document and the second one for specific math elements, e.g. a definition inspired by or adapted from one in book B.

Listing 31.2: A Formalization with Scientific Antecedent (`ant`)

```
<omdoc xml:id="NNat" version="1.6" xmlns:dc="http://purl.org/dc/elements/1.1/">
  <metadata><dc:title>Natural Numbers</dc:title></metadata>
  ...
  <theory xml:id="NNat.thy">
    <metadata>
      <dc:title>Natural Numbers</dc:title>
      <dc:creator role="aut">R</dc:creator>
      <dc:contributor role="ant">A</dc:contributor>
      <dc:source>B</dc:source>
    </metadata>
    ...
  </theory>
  ...
</omdoc>
```

ENP:34

[=cc]

Part IX

Managing Rights by Creative Commons Licenses

The Dublin Core vocabulary provides the `dc:rights` element for information about rights held in and over the document or element content, but leaves the content model unspecified. While it is legally sufficient to describe this information in natural language, a content markup format like OMDoc should support a machine-understandable format. As one of the purposes of the OMDoc format is to support the sharing and re-use of mathematical content, OMDoc provides markup for rights management via the Creative Commons (CC) licenses. Digital rights management (DRM) and licensing of intellectual property has become a hotly debated topic in the last years. We feel that the Creative Commons licenses that encourage sharing of content and enhance the (scientific) public domain while giving authors some control over their intellectual property establish a good middle ground. Specifying rights is important, since in the absence of an explicit or implicit (via inheritance) `dc:rights` element no assumptions can be made about the status of the document or fragment. Therefore OMDoc adds another child to the `metadata` element.

This `cc:license` element is a symbolic representation of the Creative Commons legal framework, adapted to the OMDoc setting: The Creative Commons Metadata Initiative specifies various ways of embedding CC metadata into documents and electronic artefacts like pictures or MP3 recordings. As OMDoc is a source format, from which various presentation formats are generated, we need a content representation of the CC metadata from which the end-user representations for the respective formats can be generated.

Element	Attributes		Content
	Req.	Optional	
cc:license		jurisdiction	permissions, prohibitions, requirements,h:p*
cc:permissions		reproduction, distribution, derivative_works	h:p*
cc:prohibitions		commercial_use	h:p*
cc:requirements		notice, copyleft, attribution	h:p*

Figure 31.2: The OMDoc Elements for Creative Commons Metadata

cc:license

Definition 31.1.1 The Creative Commons Metadata Initiative [**URL:creativecommons**] divides the license characteristics in three types: **permissions** , **prohibitions** and **requirements** , which are represented by the three elements, which can occur as children of the **cc:license** element. After these, a a natural language explanation of the license grant in a math text (see Part V). The `cc:license` element has two optional arguments:

`jurisdiction` which allows to specify the country in whose jurisdiction the license will be enforced[1]. It's value is one of the top-level domain codes of the "Internet Assigned Names Authority (IANA)" [**IANA:TLD**]. If this attribute is absent, then the original US version of the license is assumed.

`version` which allows to specify the version of the license. If the attribute is not present, then the newest released version is assumed (version 2.0 at the time of writing this book)

The following three elements can occur as children of the `cc:license` element; their attribute specify the rights bestowed on the user by the license. All these elements have the namespace `http://creativecommons.org/ns`, for which we traditionally use the namespace prefix `cc:`. All three elements can contain a natural language explanation of their particular contribution to the license grant in a sequence of `h:p` elements.

cc:permissions

Definition 31.1.2 (Permissions) cc:permissions are the rights granted by the license, to model them the element has three attributes, which can have the values `permitted` (the permission is granted by the license) and `prohibited` (the permission isn't):

Attribute	Permission	Default
`reproduction`	the work may be reproduced	`permitted`
`distribution`	the work may be distributed, publicly displayed, and publicly performed	`permitted`
`derivative_works`	derivative works may be created and reproduced	`permitted`

cc:prohibitions

Definition 31.1.3 (Prohibitions) cc:prohibitions are the things the license prohibits.

Attribute	Prohibition	Default
`commercial_use`	stating that rights may be exercised for commercial purposes.	`permitted`

cc:requirements

Definition 31.1.4 (Requirements) cc:requirements are restrictions imposed by the license.

Attribute	Requirement	Default
`notice`	copyright and license notices must be kept intact	`required`
`attribution`	credit must be given to copyright holder and/or author	`required`
`copyleft`	derivative works, if authorized, must be licensed under the same terms as the work	`required`

This vocabulary is directly modeled after the Creative Commons Metadata [**URL:creativecommonsMetadata**] which defines the meaning, and provides an RDF [**LasSwi:rdf99**] based implementation. As we have discussed in Chapter 23, OMDoc follows an approach that specifies metadata in the document itself; thus we have provided the elements described here. In contrast to many other situations in OMDoc, the rights model is not extensible, since only the current model is backed by legal licenses provided by the creative commons initiative.

Listing 31.3 specifies a license grant using the Creative Commons "share-alike" license: The copyright is retained by the author, who licenses the content to the world, allowing others to reproduce and distribute it without restrictions as long as the copyright notice is kept intact. Furthermore, it allows others to create derivative works based on the content as long as it attributes the original work of the author and licenses the derived work under the identical license (i.e. the Creative Commons "share-alike" as well).

[1]The Creative Commons Initiative is currently in the process of adapting their licenses to jurisdictions other than the USA, where the licenses originated. See [**URL:creativecommonsworldwide**] for details and to check for progress.

Listing 31.3: A Creative Commons License

```
<metadata>
  <dc:rights>Copyright (c) 2004 Michael Kohlhase</dc:rights>
  <license jurisdiction="de" xmlns="http://creativecommons.org/ns">
    <permissions reproduction="permitted" distribution="permitted"
                 derivative_works="permitted"/>
    <prohibitions commercial_use="permitted"/>
    <requirements notice="required" copyleft="required" attribution="required"/>
  </license>
</metadata>
```

Part X

Derived Statements

[=derived-defs]

Chapter 32

Derived Definition Forms

BOP:36

We say that a definiendum is **well-defined** , iff the corresponding definiens uniquely determines it; adding such definitions to a theory always results in a conservative extension.

Definiens	Definiendum	Type
The number 1	$1 {:=} s(0)$ (1 is the successor of 0)	simple
The exponential function $e^{\cdot}$	The **exponential function** $e^{\cdot}$ is the solution to the differential equation $\partial f = f$ [where $f(0) = 1$].	implicit
The addition function +	**Addition** on the natural numbers is defined by the equations $x + 0 = x$ and $x + s(y) = s(x + y)$.	recursive

Figure 32.1: Some Common Definitions

Definitions can have many forms, they can be

- equations where the left hand side is the defined symbol and the right hand side is a term that does not contain it, as in our discussion above or the first case in Figure 32.1. We call such definitions **simple** .
- general statements that uniquely determine the meaning of the objects or concepts in question, as in the second definition in Figure 32.1. We call such definitions **implicit** ; the Peano axioms are another example of this category.

  Note that this kind of definitions requires a proof of unique existence to ensure well-definedness. Incidentally, if we leave out the part in square brackets in the second definition in Figure 32.1, the differential equation only characterizes the exponential function up to additive real constants. In this case, the "definition" only restricts the meaning of the exponential function to a set of possible values. We call such a set of axioms a **loose** definition.
- given as a set of equations, as in the third case of Figure 32.1, even though this is strictly a special case of an implicit definition: it is a sub-case, where well-definedness can be shown by giving an argument why the systematic applications of these equations terminates, e.g. by exhibiting an ordering that makes the left hand sides strictly smaller than the right-hand sides. We call such a definition **inductive** .

In Figure 32.1 we have seen that there are many ways to fix the meaning of a symbol, therefore OMDoc `definition` elements are more complex than `axioms`. In particular, the `definition` element supports several kinds of definition mechanisms with specialized content models specified in the `type` attribute (cf. the discussion at the end of Chapter 10):
[=implicit-defs]

[36] Old Part: fit into the picture here

Element	Attributes		M	Content
	Required	Optional	C	
definition	`for`	`xml:id, type, style, class, uniqueness, existence`	+	`h:p*` 《*mobj*》
definition	`for`	`xml:id, type, style, class, consistency, exhaustivity`	+	`h:p*, requation+, measure?, ordering?`
requation		`xml:id, style, class`	–	《*mobj*》`,`《*mobj*》
measure		`xml:id, style, class`	–	《*mobj*》
ordering		`xml:id, style, class`	–	《*mobj*》
where 《*mobj*》 is `(OMOBJ |m:math |legacy)`				

Figure 32.2: Theory-Constitutive Elements in OMDoc

32.1 Implicit Definitions

This kind of definition is often (more accurately) called "*definition by description*", since the definiendum is described so accurately, that there is exactly one object satisfying the description. The "description" of the defined symbol is given as a multi-system `FMP` group whose content uniquely determines the value of the symbols that are specified in the `for` attribute of the `definition` element with `type implicit`. The necessary statement of unique existence can be specified in the `existence` and `uniqueness` attribute, whose values are URI references to to assertional statements (see Section 13.3) that represent the respective properties. We give an example of an implicit definition in Listing 32.1.

Listing 32.1: An Implicit Definition of the Exponential Function

```
<definition xml:id="exp−def" for="#exp" type="implicit"
            uniqueness="#exp−unique" existence="#exp−exists">
  <FMP>exp' = exp ∧ exp(0) = 1</FMP>
</definition>
<assertion xml:id="exp−unique">
  <h:p>
    There is at most one differentiable function that solves the
     differential equation in definition <ref type="cite" xref="#exp−def"/>.
  </h:p>
</assertion>
<assertion xml:id="exp−exists">
  <h:p>
    The differential equation in <ref type="cite" xref="#exp−def"/> is solvable.
  </h:p>
</assertion>
```

[=inductive-defs]

32.2 Inductive Definitions

`requation`

This is a variant of the `implicit` case above. It defines a recursive function by a set of recursive equations in **requation** elements whose left and right hand sides are specified by the two children. The first one is called the **pattern** , and the second one the **value** . The intended meaning of the defined symbol is, that the value (with the variables suitably substituted) can be substituted for a formula that matches the pattern element. In this case, the `definition` element carries a `type` with value `inductive` and the optional attributes `exhaustivity` and `consistency`, which point to `assertion`s stating that the cases spanned by the patterns are exhaustive (i.e. all cases are considered), or that the values are consistent (where the cases overlap, the values are equal).

Listing 32.2 gives an example of a a recursive definition of the addition on the natural numbers.

Listing 32.2: A recursive definition of addition

```
<definition xml:id="plus.def" for="#plus" type="inductive"
            consistency="#s−not−0" exhaustivity="#s−or−0">
  <metadata><dc:subject>addition</dc:subject></metadata>
  <h:p>Addition is defined by recursion on the second argument.</h:p>
  <requation>x + 0 ⇝ x</requation>
  <requation>x + s(y) ⇝ s(x + y)</requation>
```

</definition>

To guarantee termination of the recursive instantiation (necessary to ensure well-definedness), it is possible to specify a measure function and well-founded ordering by the optional `measure` and `ordering` elements which contain mathematical objects. The elements contain mathematical objects.

Definition 32.2.1 The content of the **measure** element specifies a measure function, i.e. a function from argument tuples for the function defined in the parent `definition` element to a space with an ordering relation which is specified in the **ordering** element. This element also carries an optional attribute `terminating` that points to an `assertion` element that states that this ordering relation is a terminating partial ordering. `measure` `ordering`

Definition 32.2.2 Pattern definitions are a special degenerate cases of the recursive definition. A function is defined by a set of `requation` elements, but the defined function does not occur in the second children.

This form of definition is often used instead of `simple` in logical languages that do not have a function constructor. It allows to define a function by its behavior on patterns of arguments. Since termination is trivial in this case, no `measure` and `ordering` elements appear in the body of a `definition` element whose `type` has value `pattern`. EOP:36

[=adt]

Chapter 33

Abstract Data Types (Module ADT)

[=adt] Most specification languages for mathematical theories support definition mechanisms for sets that are inductively generated by a set of constructors and recursive functions on these under the heading of abstract data types. Prominent examples of abstract data types are natural numbers, lists, trees, etc. The module ADT presented in this chapter extends OMDoc by a concise syntax for abstract data types that follows the model used in the CASL (Common Abstract Specification Language [**CoFI:2004:CASL-RM**]) standard.

Conceptually, an abstract data type declares a collection of symbols and axioms that can be used to construct certain mathematical objects and to group them into sets. For instance, the Peano axioms (see Figure 11.1) introduce the symbols 0 (the number zero), s (the successor function), and $\mathbb{N}$ (the set of natural numbers) and fix their meaning by five axioms. These state that the set $\mathbb{N}$ contains exactly those objects that can be constructed from 0 and s alone (these symbols are called **constructor symbol** s and the representations **constructor term** s). Optionally, an abstract data type can also declare **selector symbol** s, for (partial) inverses of the constructors. In the case of natural numbers the predecessor function is a selector for s: it "selects" the argument n, from which a (non-zero) number $s(n)$ has been constructed.

Following CASL we will call sets of objects that can be represented as constructor terms **sort** s. A sort is called **free** , iff there are no identities between constructor terms, i.e. two objects represented by different constructor terms can never be equal.

The sort $\mathbb{N}$ of natural numbers is a free sort. An example of a sort that is not free is the theory of finite sets given by the constructors $\emptyset$ and the set insertion function ι , since the set $\{a\}$ can be obtained by inserting a into the empty set an arbitrary (positive) number of times; so e.g. $\iota(a,\emptyset) = \iota(a,\iota(a,\emptyset))$. This kind of sort is called **generated** , since it only contains elements that are expressible in the constructors.

An abstract data type is called **loose** , if it contains elements besides the ones generated by the constructors. We consider free sorts more **strict** than generated ones, which in turn are more strict than loose ones.

Definition 33.0.1 In OMDoc, we use the **adt** element to specify abstract data types possibly consisting of multiple sorts. It is a theory-constitutive statement and can only occur as a child of a `theory` element (see Chapter 10 for a discussion). An `adt` element contains one or more `sortdef` elements that define the sorts and specify their members and it can carry a `parameters` attribute that contains a whitespace-separated list of parameter variable names. If these are present, they declare type variables that can be used in the specification of the new sort and constructor symbols see [**Kohlhase:OMDoc1.6projects**] for an example. `adt`

We will use an augmented representation of the abstract data type of natural numbers as a running example for introduction of the functionality added by the ADT module; Listing 33.1

Element	Attributes		M	Content
	Req.	Optional	D	
adt		xml:id, class, style, parameters	+	sortdef+
sortdef	name	type, role, scope, class, style	+	(constructor | insort)*, recognizer?
constructor	name	type, scope, class, style	+	argument*
argument			+	type, selector?
insort	for		−	
selector	name	type, scope, role, total, class, style	+	EMPTY
recognizer	name	type, scope, role, class, style	+	

Figure 33.1: Abstract data types in OMDoc

contains the listing of the OMDoc encoding. In this example, we introduce a second sort $\mathbb{P}$ for positive natural numbers to make it more interesting and to pin down the type of the predecessor function.

sortdef

Definition 33.0.2 A **sortdef** element is a highly condensed piece of syntax that declares a sort symbol together with the constructor symbols and their selector symbols of the corresponding sort. It has a required `name` attribute that specifies the symbol name, an optional `type` attribute that can have the values `free`, `generated`, and `loose` with the meaning discussed above. A `sortdef` element contains a set of **constructor** and **insort** elements. The latter are empty elements which refer to a sort declared elsewhere in a `sortdef` with their `for` attribute: An `insort` element with `for="`⟪*URI*⟫`#`⟪*name*⟫`"` specifies that all the constructors of the sort ⟪*name*⟫ are also constructors for the one defined in the parent `sortdef`. Furthermore, the type of a sort given by a `sortdef` element can only be as strict as the types of any sorts included by its `insort` children.

constructor

insort

Listing 33.1 introduces the sort symbols `pos-nats` (positive natural numbers) and `nats` (natural numbers) , the symbol names are given by the required `name` attribute. Since a constructor is in general an n-ary function, a `constructor` element contains n `argument` children that specify the argument sorts of this function along with possible selector functions.

argument

Definition 33.0.3 The argument sort is given as the first child of the **argument** element: a `type` element as described in Section 12.2.

Note that n may be 0 and thus the constructor element may not have `argument` children (see for instance the `constructor` for `zero` in Listing 33.1). The first `sortdef` element there introduces the constructor symbol `succ@Nat` for the successor function. This function has one argument, which is a natural number (i.e. a member of the sort `nats`).

Sometimes it is convenient to specify the inverses of a constructors that are functions. For this OMDoc offers the possibility to add an empty `selector` element as the second child of an `argument` child of a `constructor`.

selector

Definition 33.0.4 The **selector** element has a required attribute `name` specifies the symbol name, the optional `total` attribute of the `selector` element specifies whether the function represented by this symbol is total (value `yes`) or partial (value `no`). In Listing 33.1 the `selector` element in the first `sortdef` introduces a selector symbol for the successor function `succ`. As `succ` is a function from `nats` to `pos-nats`, `pred` is a total function from `pos-nats` to `nats`.

recognizer

Definition 33.0.5 Finally, a `sortdef` element can contain a **recognizer** child that specifies a symbol for a predicate that is true, iff its argument is of the respective sort. The name of the predicate symbol is specified in the required `name` attribute.

Listing 33.1 introduces such a **recognizer predicate** as the last child of the `sortdef` element for the sort `pos-nats`.

Note that the `sortdef`, `constructor`, `selector`, and `recognizer` elements define symbols of the name specified by their `name` element in the theory that contains the `adt` element. To govern the visibility, they carry the attribute `scope` (with values `global` and `local`) and the attribute `role` (with values `type`, `sort`, `object`).

Listing 33.1: The natural numbers using `adt` in OMDoc

```
<theory xml:id="Nat">
  <adt xml:id="nat-adt">
    <metadata>
      <dc:title>Natural Numbers as an Abstract Data Type.</dc:title>
      <dc:description>The Peano axiomatization of natural numbers.</dc:description>
    </metadata>

    <sortdef name="pos-nats" type="free">
      <metadata>
        <dc:description>The set of positive natural numbers.</dc:description>
      </metadata>
      <constructor name="succ">
        <metadata><dc:description>The successor function.</dc:description></metadata>
        <argument>
          <type><OMS cd='Nat' name="nats"/></type>
          <selector name="pred" total="yes">
            <metadata><dc:description>The predecessor function.</dc:description></metadata>
          </selector>
        </argument>
      </constructor>
      <recognizer name="positive">
        <metadata>
          <dc:description>
            The recognizer predicate for positive natural numbers.
          </dc:description>
        </metadata>
      </recognizer>
    </sortdef>

    <sortdef name="nats" type="free">
      <metadata><dc:description>The set of natural numbers</dc:description></metadata>
      <constructor name="zero">
        <metadata><dc:description>The number zero.</dc:description></metadata>
      </constructor>
      <insort for="#pos-nats"/>
    </sortdef>
  </adt>
</theory>
```

To summarize Listing 33.1: The abstract data type `nat-adt` is free and defines two sorts `pos-nats` and `nats` for the (positive) natural numbers. The positive numbers (`pos-nats`) are generated by the successor function (which is a constructor) on the natural numbers (all positive natural numbers are successors). On `pos-nats`, the inverse `pred` of `succ` is total. The set `nats` of all natural numbers is defined to be the union of `pos-nats` and the constructor `zero`. Note that this definition implies the five well-known Peano Axioms: the first two specify the constructors, the third and fourth exclude identities between constructor terms, while the induction axiom states that `nats` is generated by `zero` and `succ`. The document that contains the `nat-adt` could also contain the symbols and axioms defined implicitly in the `adt` element explicitly as `symbol` and `axiom` elements for reference. These would then carry the `generated-from` attribute with value `nat-adt`.

Chapter 34

Strict Translations

We will now give the a formal[37] semantics of the ST elements in terms of strict OMDoc (see Part II).[38][39][40] EdN:37 EdN:38 EdN:39 EdN:40

Implicit definitions are licensed by a description operator in the meta-theory. In a theory $\langle\!\langle t\rangle\!\rangle$, whose meta-theory $\langle\!\langle m\rangle\!\rangle$ contains a description operator $\langle\!\langle that\rangle\!\rangle$ we have[41][42] EdN:41 EdN:42

pragmatic	strict
<symbol name="$\langle\!\langle n\rangle\!\rangle$">	<object name="$\langle\!\langle n\rangle\!\rangle$" xml:id="$\langle\!\langle i\rangle\!\rangle$">
<type system="$\langle\!\langle s\rangle\!\rangle$">$\langle\!\langle t\rangle\!\rangle$</type>	<type system="$\langle\!\langle s\rangle\!\rangle$">$\langle\!\langle t\rangle\!\rangle$</type>
</symbol>	<definition>
<definition type="implicit"	<OMBIND>
xml:id="$\langle\!\langle i\rangle\!\rangle$" for="$\langle\!\langle f\rangle\!\rangle$"	<OMS cd="$\langle\!\langle m\rangle\!\rangle$" name="$\langle\!\langle that\rangle\!\rangle$"/>
uniqueness="$\langle\!\langle u\rangle\!\rangle$" existence="$\langle\!\langle e\rangle\!\rangle$">	<OMBVAR><OMV name="$\langle\!\langle x\rangle\!\rangle$"/></OMBVAR>
<FMP>$\langle\!\langle \Phi[n]\rangle\!\rangle$</FMP>	$\boxed{\Phi[x]}$
</definition>	</OMBIND>
	</definition>
	</object>

where $\Phi[x]$ is obtained from $\Phi[n]$ by replacing all `<OMS cd="`$\langle\!\langle t\rangle\!\rangle$`'' name="`$\langle\!\langle n\rangle\!\rangle$`"/>` by `<OMV name="`$\langle\!\langle n\rangle\!\rangle$`''/>`.

Similarly inductive definitions are licensed by a recursion operator $\langle\!\langle rec\rangle\!\rangle$:

pragmatic	strict
<symbol name="$\langle\!\langle n\rangle\!\rangle$">	<object name="$\langle\!\langle n\rangle\!\rangle$" xml:id="$\langle\!\langle i\rangle\!\rangle$">
<type system="$\langle\!\langle s\rangle\!\rangle$">$\langle\!\langle t\rangle\!\rangle$</type>	<type system="$\langle\!\langle s\rangle\!\rangle$">$\langle\!\langle t\rangle\!\rangle$</type>
</symbol>	<definition>
<definition type="inductive"	<OMBIND>
xml:id="$\langle\!\langle i\rangle\!\rangle$" for="$\langle\!\langle f\rangle\!\rangle$"	<OMS cd="$\langle\!\langle m\rangle\!\rangle$" name="$\langle\!\langle rec\rangle\!\rangle$"/>
consistency="$\langle\!\langle c\rangle\!\rangle$" exhaustivity="$\langle\!\langle e\rangle\!\rangle$">	<OMBVAR><OMV name="$\langle\!\langle x\rangle\!\rangle$"/></OMBVAR>
<requation>$\langle\!\langle \Phi_1[n]\rangle\!\rangle\langle\!\langle \Psi_1[n]\rangle\!\rangle$</requation>	$\boxed{\Phi_1[x]}\ \boxed{\Psi_2[x]}\ \dots\ \boxed{\Phi_m[x]}\ \boxed{\Psi_m[x]}$
...	</OMBIND>
<requation>$\langle\!\langle \Phi_m[n]\rangle\!\rangle\langle\!\langle \Psi_m[n]\rangle\!\rangle$</requation>	</definition>
</definition>	</object>

[43][44] EdN:43 EdN:44

[37]EdNote: do we really want to call it "formal"?

[38]EdNote: what do we do if there is both FMP and CMPs in an axiom?

[39]EdNote: what do we do if there is more than one symbol per definition?

[40]EdNote: what do we do for non-simple definitions

[41]EdNote: point to a theory with description operator or similar functionality. Maybe give a special theory D and allow all meta-theories $\langle\!\langle t\rangle\!\rangle$ that have a view from D.

[42]EdNote: do we want a description operator that takes the existence and uniqueness proofs as arguments? Can we point to `proof` elements somehow? What do we really do with the attributes?

[43]EdNote: what is the correct form of the recursion operator?

[44]EdNote: similar question with the attributes here.

Part XI

Representing Proofs (Module PF)

[=proofs-intro] Proofs form an essential part of mathematics and modern sciences. Conceptually, a **proof** is a representation of uncontroversial evidence for the truth of an assertion.

The question of what exactly constitutes a proof has been controversially discussed (see e.g. [**BarCoh:ecm01**]). The clearest (and most radical) definition is given by theoretical logic, where a proof is a sequence, or tree, or directed acyclic graph (DAG) of applications of inference rules from a formally defined logical calculus, that meets a certain set of well-formedness conditions. There is a whole zoo of logical calculi that are optimized for various applications. They have in common that they are extremely explicit and verbose, and that the proofs even for simple theorems can become very large. The advantage of having formal and fully explicit proofs is that they can be very easily verified, even by simple computer programs. We will come back to this notion of proof in Chapter 37.

In mathematical practice the notion of a proof is more flexible, and more geared for consumption by humans: any line of argumentation is considered a proof, if it convinces its readers that it could in principle be expanded to a formal proof in the sense given above. As the expansion process is extremely tedious, this option is very seldom carried out explicitly. Moreover, as proofs are geared towards communication among humans, they are given at vastly differing levels of abstraction. From a very informal proof idea for the initiated specialist of the field, who can fill in the details herself, down to a very detailed account for skeptics or novices which will normally be still well above the formal level. Furthermore, proofs will usually be tailored to the specific characteristics of the audience, who may be specialists in one part of a proof while unfamiliar to the material in others. Typically such proofs have a sequence/tree/DAG-like structure, where the leaves are natural language sentences interspersed with mathematical formulae (or mathematical vernacular).

Let us consider a proof and its context (Figure 34.1) as it could be found in a typical elementary math. textbook, only that we have numbered the proof steps for referencing convenience. Figure 34.1 will be used as a running example throughout this chapter.

Since proofs can be marked up on several levels, we will introduce the OMDoc markup for proofs in stages: We will first concentrate on proofs as structured texts, marking up the discourse structure in example Figure 34.1. Then we will concentrate on the justifications of proof steps, and finally we will discuss the scoping and hierarchical structure of proofs.

The development of the representational infrastructure in OMDoc has a long history: From the beginning the format strived to allow structural semantic markup for textbook proofs as well as accommodate a wide range of formal proof systems without over-committing to a particular system. However, the proof representation infrastructure from Version 1.1 of OMDoc turned

Theorem: *There are infinitely many prime numbers.*
Proof: We need to prove that the set P of all prime numbers is not finite.

1. We proceed by assuming that P is finite and reaching a contradiction.
2. Let P be finite.
3. Then $P = \{p_1, \ldots, p_n\}$ for some p_i.
4. Let $q := p_1 \cdots p_n + 1$.
5. Since for each $p_i \in P$ we have $q > p_i$, we conclude $q \notin P$.
6. We prove the absurdity by showing that q is prime:
7. For each $p_i \in P$ we have $q = p_i k + 1$ for some natural number k, so p_i can not divide q;
8. q must be prime as P is the set of all prime numbers.
9. Thus we have contradicted our assumption (2)
10. and proven the assertion. □

Figure 34.1: A Theorem with a Proof.

out not to be expressive enough to represent the proofs in the HELM library [**AspPad:hsmw01**]. As a consequence, the PF module has been redesigned [**AspKohSac:dtdop03**] as part of the MOWGLI project [**AspKoht:mimp02**]. The current version of the PF module is an adaptation of this proposal to be as compatible as possible with earlier versions of OMDoc. It has been validated by interpreting it as an implementation of the $\overline{\lambda}\mu\tilde{\mu}$ calculus [**SacerdotiCoen:enlt05**] proof representation calculus.

[=proof-structure]

Chapter 35

Proof Structure

In this section, we will concentrate on the structure of proofs apparent in the proof text and introduce the OMDoc infrastructure needed for marking up this aspect. Even if the proof in Figure 34.1 is very short and simple, we can observe several characteristics of a typical mathematical proof. The proof starts with the thesis that is followed by nine main "steps" (numbered from 1 to 10). A very direct representation of the content of Figure 34.1 is given in Listing 35.1.

Listing 35.1: An OMDoc Representation of Figure 34.1.

```
<assertion xml:id="a1">
  <h:p>There are infinitely many prime numbers.</h:p>
</assertion>
<proof xml:id="p" for="#a1">
  <omtext xml:id="intro">
    <h:p>We need to prove that the set $P$ of all prime numbers is not finite.</h:p>
  </omtext>
  <derive xml:id="d1">
    <h:p>We proceed by assuming that $P$ is finite and reaching a contradiction.</h:p>
    <method>
      <proof xml:id="p1">
        <hypothesis xml:id="h2"><h:p>Let $P$ be finite.</h:p></hypothesis>
        <derive xml:id="d3">
          <h:p>Then $P = \{p_1, \ldots, p_n\}$ for some $p_i$.</h:p>
          <method><premise xref="#h2"/></method>
        </derive>
        <symbol name="q"/>
        <definition xml:id="d4" for="q" type="informal">
          <CMP>Let $q \overset{def}{=} p_1 \cdots p_n + 1$</CMP>
        </definition>
        <derive xml:id="d5">
          <h:p> Since for each $p_i \in P$ we have $q > p_i$, we conclude $q \notin P$.</h:p>
        </derive>
        <omtext xml:id="c6">
          <h:p>We prove the absurdity by showing that $q$ is prime:</h:p>
        </omtext>
        <derive xml:id="d7">
          <h:p>For each $p_i \in P$ we have $q = p_i k + 1$ for some
            natural number $k$, so $p_i$ can not divide $q$;</h:p>
          <method><premise xref="#d4"/></method>
        </derive>
        <derive xml:id="d8">
          <h:p>$q$ must be prime as $P$ is the set of all prime numbers.</h:p>
          <method><premise xref="#d7"/></method>
        </derive>
        <derive xml:id="d9">
          <h:p>Thus we have contradicted our assumption</h:p>
          <method><premise xref="#d5"/><premise xref="#d8"/></method>
        </derive>
      </proof>
    </method>
  </derive>
  <derive xml:id="d10" type="conclusion">
    <h:p>This proves the assertion.</h:p>
  </derive>
</proof>
```

proof

Definition 35.0.1 Proofs are specified by **proof** elements in OMDoc that have the optional attributes `xml:id` and `theory` and the required attribute `for`. The `for` attribute points to the assertion that is justified by this proof (this can be an `assertion` element or a `derive` proof step (see below), thereby making it possible to specify expansions of justifications and thus hierarchical proofs).

Note that there can be more than one proof for a given assertion.

Element	Attributes		M	Content
	Req.	Optional	D	
proof	for	theory, xml:id, class, style	+	(omtext | derive | hypothesis | symbol | definition)*
proofobject		xml:id, for, class, style, theory	+	(OMOBJ |m:math |legacy)
hypothesis		xml:id, class, style, inductive	–	CMP*, FMP*
derive		xml:id, class, style, type	–	CMP*, FMP*, method?
method		xref	–	(OMOBJ |m:math |legacy | premise | proof | proofobject)*
premise	xref		–	EMPTY

Figure 35.1: The OMDoc Proof Elements

The content of a proof consists of a sequence of proof steps, whose DAG structure is given by cross-referencing. These proof steps are specified in four kinds of OMDoc elements:

`omtext` OMDoc allows this element to allow for intermediate text in proofs that does not have to have a logical correspondence to a proof step, but e.g. guides the reader through the proof. Examples for this are remarks by the proof author, e.g. an explanation why some other proof method will not work. We can see another example in Listing 35.1 in lines 5-7, where the comment gives a preview over the course of the proof.

`derive` elements specify normal proof steps that derive a new claim from already known ones, from assertions or axioms in the current theory, or from the assumptions of the assertion that is under consideration in the proof. See for example lines $12ff$ in Listing 35.1 for examples of `derive` proof steps that only state the local assertion. We will consider the specification of justifications in detail in Chapter 35 below. The **derive** element carries an optional `xml:id` attribute for identification and an optional `type` to single out special cases of proofs steps.

derive

The value `conclusion` is reserved for the concluding step of a proof[1], i.e. the one that derives the assertion made in the corresponding theorem.

The value `gap` is used for proof steps that are not justified (yet): we call them **gap steps** . Note that the presence of gap steps allows OMDoc to specify incomplete proofs as proofs with gap steps.

`hypothesis` elements allow to specify local assumptions that allow the hypothetical reasoning discipline needed for instance to specify proof by contradiction, by case analysis, or simply to show that A implies B, by assuming A and then deriving B from this local hypothesis. The scope of an hypothesis extends to the end of the `proof` element containing it. In Listing 35.1 the classification of step 2 from Figure 34.1 as the **hypothesis** element `h2` forces us to embed it into a `derive` element with a `proof` grandchild, making a structure apparent that was hidden in the original.

hypothesis

An important special case of hypothesis is the case of "inductive hypothesis", this can be flagged by setting the value of the attribute `inductive` to `yes`; the default value is `no`.

[1]As the argumentative structure of the proof is encoded in the justification structure to be detailed in Chapter 35, the concluding step of a proof need not be the last child of a proof element.

`symbol`/`definition` These elements allow to introduce new local symbols that are local to the containing `proof` element. Their meaning is just as described in Section 12.3, only that the role of the `axiom` element described there is taken by the `hypothesis` element. In Listing 35.1 step 4 in the proof is represented by a `symbol`/`definition` pair. Like in the `hypothesis` case, the scope of this symbol extends to the end of the `proof` element containing it.

These elements contain an informal (natural language) representation of the proof step in a multilingual `CMP` group and possibly an `FMP` element that gives a formal representation of the claim made by this proof step. A `derive` element can furthermore contain a `method` element that specifies how the assertion is derived from already-known facts (see the next section for details). All of the proof step elements have an optional `xml:id` attribute for identification and the CSS attributes.

As we have seen above, the content of any proof step is essentially a Gentzen-style sequent; see Listing 36.2 for an example. This mixed representation enhances multi-modal proof presentation [**Fiedler:tape97**], and the accumulation of proof information in one structure. Informal proofs can be formalized [**Baur:susmt99**]; formal proofs can be transformed to natural language [**HuangFiedler:pmfp96**]. The first is important, since it will be initially infeasible to totally formalize all mathematical proofs needed for the correctness management of the knowledge base.

[=justifications]

Chapter 36

Proof Step Justifications

So far we have only concerned ourselves with the linear structure of the proof, we have identified the proof steps and classified them by their function in the proof. A central property of the `derive` elements is that their content (the local claim) follows from statements that we consider true. These can be earlier steps in the proof or general knowledge. To convince the reader of a proof, the steps are often accompanied with a **justification** . This can be given either by a logical inference rule or higher-level evidence for the truth of the claim. The evidence can consist in a proof method that can be used to prove the assertion, or in a separate subproof, that could be presented if the consumer was unconvinced. Conceptually, both possibilities are equivalent, since the method can be used to compute the subproof (called its **expansion**). Justifications are represented in OMDoc by the `method` children of `derive` elements[1] (see Listing 36.1 for an example):

Definition 36.0.1 The **method** element contains a structural specification of the justification of the claim made in the `FMP` of a `derive` element. `method`

So the `FMP` together with the `method` element jointly form the counterpart to the natural language content of the `CMP` group, they are sibling to: The `FMP` formalizes the local claim, and the `method` stands for the justification. In Listing 36.1 the formula in the `CMP` element corresponds to the claim, whereas the part "By . . . , we have" is the justification. In other words, a `method` element specifies a proof method or inference rule with its arguments that justifies the assertion made in the `FMP` elements. It has an optional `xref` attribute whose target is an OMDoc definition of an inference rule or proof method.[2] A method may have OpenMath objects, Content-MathML expressions, `legacy`, `premise`, `proof`, and `proofobject`[3] children. These act as parameters to the method, e.g. for the repeated universal instantiation method in Listing 36.1 the parameters are the terms to instantiate the bound variables.

Definition 36.0.2 The **premise** elements are used to refer to already established assertions: other proof steps or statements — e.g. ones given as `assertion`, `definition`, or `axiom` elements — the method was applied to to obtain the local claim of the proof step. The `premise` elements are empty and carry the required attribute `xref`, which contains the URI of the assertion. `premise`

[1]The structural and formal justification elements discussed in this section are derived from hierarchical data structures developed for semi-automated theorem proving (satisfying the logical side). They allow natural language representations at every level (allowing for natural representation of mathematical vernacular at multiple levels of abstraction). This proof representation (see [**BenzmuellerEtAl:otama97**] for a discussion and pointers) is a DAG of nodes which represent the proof steps.

[2]At the moment OMDoc does not provide markup for such objects, so that they should best be represented by `symbols` with `definition` where the inference rule is explained in the `CMP` (see the lower part of Listing 36.1), and the `FMP` holds a content representation for the inference rule, e.g. using the content dictionary [**CD:inference-rules**]. A good enhancement is to encapsulate system-specific encodings of the inference rules in `private` or `code` elements and have the `xref` attribute point to these.

[3]This object is an alternative representation of certain proofs, see Chapter 37.

Thus the `premise` elements specify the DAG structure of the proof. Note that even if we do not mark up the method in a justification (e.g. if it is unknown or obvious) it can still make sense to structure the argument in `premise` elements. We have done so in Listing 35.1 to make the dependencies of the argumentation explicit.

If a `derive` step is a logically (or even mathematically) complex step, an expansion into sub-steps can be specified in a `proof` or `proofobject` element embedded into the justifying `method` element. An embedded proof allows us to specify generic markup for the hierarchic structure of proofs. Expansions of nodes justified by method applications are computed, but the information about the method itself is not discarded in the process as in tactical theorem provers like Isabelle [**Paulson:iagtp94**] or NuPrL [**Constable86**]. Thus, proof nodes may have justifications at multiple levels of abstraction in an hierarchical proof data structure. Thus the `method` elements allow to augment the linear structure of the proof by a tree/DAG-like secondary structure given by the `premise` links. Due to the complex hierarchical structure of proofs, we cannot directly utilize the tree-like structure provided by XML, but use cross-referencing. The `derive` step in Listing 36.1 represents an inner node of the proof tree/DAG with three children (the elements with identifiers `A2`, `A4`, and `A5`).

Listing 36.1: A `derive` Proof Step

```
<proof xml:id="proof.2.1.2.proof.D2.1" for="#assertion.2.1.2">
  ...
  <derive xml:id="D2.1">
    <h:p>By <ref type="cite" xref="#A2"/>, <ref type="cite" xref="#A4"/>, and
        <ref type="cite" xref="#A5"/> we have z + (a + (−a)) = (z + a) + (−a).</h:p>
    <FMP>z + (a + (−a)) = (z + a) + (−a)</FMP>
    <method xref="nk−sorts.omdoc#NK−Sorts.forallistar">
      <OMV name="z"/>
      <OMV name="a"/>
      −a
      <premise xref="#A2"/><premise xref="#A4"/><premise xref="#A5"/>
    </method>
  </derive>
  ...
</proof>
...
<theory xml:id="NK−Sorts">
  <metadata>
    <dc:title>Natural Deduction for Sorted Logic</dc:title>
  </metadata>

  <symbol name="forallistar">
    <metadata>
      <dc:description>Repeated Universal Instantiation></dc:description>
    </metadata>
  </symbol>
  <definition xml:id="forallistar.def" for="forallistar" type="informal">
    <CMP>Given n parameters, the inference rule ∀I* instantiates
      the first n universal quantifications in the antecedent with them.</CMP>
  </definition>
  ...
</theory>
```

In OMDoc the `premise` elements must reference proof steps in the current proof or statements (`assertion` or `axiom` elements) in the scope of the current theory: A statement is in **scope** of the current theory, if its home theory is the current theory or imported (directly or indirectly) by the current theory.

Furthermore note that a proof containing a `premise` element is not self-contained evidence for the validity of the `assertion` it proves. Of course it is only evidence for the validity at all (we call such a proof grounded), if all the statements that are targets of `premise` references have grounded proofs themselves[4] and the reference relation does not contain cycles. A grounded proof can be made self-contained by inserting the target statements as `derive` elements before the referencing `premise` and embedding at least one `proof` into the `derive` as a justification.

Let us now consider another proof example (Listing 36.2) to fortify our intuition.

[4]For `assertion` targets this requirement is obvious. Obviously, `axioms` do not need proofs, but certain forms of definitions need well-definedness proofs (see Section 12.3). These are included in the definition of a grounded proof.

Listing 36.2: An OMDoc Representation of a Proof by Cases

```
<assertion xml:id="t1" theory="sets">
  <h:p>If a ∈ U or a ∈ V, then a ∈ U ∪ V.</h:p>
  <FMP>
    <assumption xml:id="t1_a">a ∈ U ∨ a ∈ V</assumption>
    <conclusion xml:id="t1_c">a ∈ U ∪ V</conclusion>
  </FMP>
</assertion>
<proof xml:id="t1_p1" for="#t1" theory="sets">
  <omtext xml:id="t1_p1_m1">
    <h:p> We prove the assertion by a case analysis.</h:p>
  </omtext>
  <derive xml:id="t1_p1_l1">
    <h:p>If a ∈ U, then a ∈ U ∪ V.</h:p>
    <FMP>
      <assumption xml:id="t1_p1_l1_a">a ∈ U</assumption>
      <conclusion xml:id="t1_p1_l1_c">a ∈ U ∪ V</conclusion>
    </FMP>
    <method xref="sk.omdoc#SK.by_definition">∪</method>
  </derive>
  <derive xml:id="t1_p1_l2">
    <h:p>If a ∈ V, then a ∈ U ∪ V.</h:p>
    <FMP>
      <assumption xml:id="t1_p1_l2_a">a ∈ V</assumption>
      <conclusion xml:id="t1_p1_l2_c">a ∈ U ∪ V</conclusion>
    </FMP>
    <method xref="sk.omdoc#SK.by_definition">∪</method>
  </derive>
  <derive xml:id="t1_p1_c">
    <h:p> We have considered both cases, so we have a ∈ U ∪ V.</h:p>
  </derive>
</proof>
```

This proof is in sequent style: The statement of all local claims is in self-contained `FMP`s that mark up the statement in `assumption`/`conclusion` form, which makes the logical dependencies explicit. In this example we use inference rules from the calculus "SK",Gentzen's sequent calculus for classical first-order logic [**Gentzen:uudlsiii35**], which we assume to be formalized in a theory `SK`. Note that local assumptions from the `FMP` should not be referenced outside the `derive` step they were made in. In effect, the `derive` element serves as a grouping device for local assumptions.

Note that the same effect as embedding a `proof` element into a `derive` step can be obtained by specifying the `proof` at top-level and using the optional `for` attribute to refer to the identity of the enclosing proof step (given by its optional `xml:id` attribute), we have done this in the proof in Listing 36.3, which expands the `derive` step with identifier `t1_p1_l1` in Listing 36.2.

Listing 36.3: An External Expansion of Step `t_1_p1_l1` in Listing 36.2

```
<definition xml:id="union.def" for="union">
  ∀P, Q, x.x ∈ P ∪ Q ⇔ x ∈ P ∨ x ∈ Q
</definition>

<proof xml:id="t1_p1_l1.exp" for="#t1_p1_l1">
  <derive xml:id="t1_p1_l1.d1">
    <FMP>
      <assumption xml:id="t1_p1_l1.d1.a">a ∈ U</assumption>
      <conclusion xml:id="t1_p1_l1.d1.c">a ∈ U</conclusion>
    </FMP>
    <method xref="sk.omdoc#SK.axiom"/>
  </derive>
  <derive xml:id="t1_p1_l1.l1.d2">
    <FMP>
      <assumption xml:id="t1_p1_l1.d2.a">a ∈ U</assumption>
      <conclusion xml:id="t1_p1_l1.d2.c">a ∈ U ∨ a ∈ V</conclusion>
    </FMP>
    <method xref="sk.omdoc#SK.orR"><premise xref="#t1_p1_l1.d1"/></method>
  </derive>
  <derive xml:id="t1_p1_l1.d3">
    <FMP>
      <assumption xml:id="t1_p1_l1.d3.a">a ∈ U ∨ a ∈ V</assumption>
      <conclusion xml:id="t1_p1_l1.d3.c">a ∈ U ∪ V</conclusion>
    </FMP>
    <method xref="sk.omdoc#SK.definition−rl">U, V, a
      <premise xref="#unif.def"/>
    </method>
  </derive>
  <derive xml:id="t1_p1_l1.d4">
    <FMP>
      <assumption xml:id="t1_p1_l1.d3.a">a ∈ U</assumption>
      <conclusion xml:id="t1_p1_l1.d3.c">a ∈ U ∪ V</conclusion>
    </FMP>
    <method xref="sk.omdoc#SK.cut">
      <premise xref="#t1_p1_l1.d2"/>
      <premise xref="#t1_p1_l1.d3"/>
    </method>
  </derive>
</proof>
```

[=scoping-proofs]

Chapter 37

Scoping and Context in a Proof

Unlike the sequent style proofs we discussed in the last section, many informal proofs use the natural deduction style [**Gentzen:uudlsiii35**], which allows to reason from local assumptions. We have already seen such hypotheses as `hypothesis` elements in Listing 35.1. The main new feature is that hypotheses can be introduced at some point in the proof, and are discharged later. As a consequence, they can only be used in certain parts of the proof. The hypothesis is inaccessible for inference outside the nearest ancestor `proof` element of the `hypothesis`.

Let us now reconsider the proof in Figure 34.1. Some of the steps (2, 3, 4, 5, 7) leave the thesis unmodified; these are called **forward reasoning** or **bottom-up proof step** s, since they are used to derive new knowledge from the available one with the aim of reaching the conclusion. Some other steps (1, 6) are used to conclude the (current) thesis by opening new subproofs, each one characterized with a new local thesis. These steps are called **backward reasoning** or **top-down proof step** s steps, since they are used to reduce a complex problem (proving the thesis) to several simpler problems (the subproofs). In our example, both backward reasoning steps open just one new subproof: Step 1 reduces the goal to proving that the finiteness of P implies a contradiction; step 5 reduces the goal to proving that q is prime.

Step 2 is used to introduce a new hypothesis, whose scope extends from the point where it is introduced to the end of the current subproof, covering also all the steps inbetween and in particular all subproofs that are introduced in these. In our example the scope of the hypothesis that P is finite (step 2 in Figure 34.1) are steps 3 – 8. In an inductive proof, for instance, the scope of the inductive hypothesis covers only the proof of the inductive step and not the proof of the base case (independently from the order adopted to present them to the user).

Step 4 is similar, it introduces a new symbol q, which is a local declaration that has scope over lines 4 – 9. The difference between a hypothesis and a local declaration is that the latter is used to introduce a variable as a new element in a given set or type, whereas the former, is used to locally state some property of the variables in scope. For example, *“let n be a natural number”* is a declaration, while *“suppose n to be a multiple of 2”* is a hypothesis. The introduction of a new hypothesis or local declaration should always be justified by a proof step that discharges it. In our example the declaration P is discharged in step 10. Note that in contrast to the representation in Listing 35.1 we have chosen to view step 6 in Figure 34.1 as a top-down proof step rather than a proof comment.

To sum up, every proof step is characterized by a current thesis and a *context*, which is the set of all the local declarations, hypotheses, and local definitions in scope. Furthermore, a step can either introduce a new hypothesis, definition, or declaration or can just be a forward or backward reasoning step. It is a forward reasoning `derive` step if it leaves the current thesis as it is. It is a backward reasoning `derive` step if it opens new subproofs, each one characterized by a new thesis and possibly a new context.

Listing 37.1: A top-down Representation of the Proof in Figure 34.1.

```
<assertion xml:id="a1">
```

```
  <h:p>There are infinitely many prime numbers.</h:p>
</assertion>
<proof for="#a1">
  <omtext xml:id="c0">
    <h:p>We need to prove that the set P of all prime numbers is not finite.</h:p>
  </omtext>
  <derive xml:id="d1">
    <h:p> We proceed by assuming that P is finite and reaching a contradiction.</h:p>
    <method xref="nk.omdoc#NK.by−contradiction">
      <proof>
        <hypothesis xml:id="h2"><h:p>Let P be finite.</h:p></hypothesis>
        <derive xml:id="d3"><h:p>Then P = {p_1, . . . , p_n} for some n</h:p></derive>
        <symbol name="q"/>
        <definition xml:id="d4" for="q" type="informal">
          <CMP>Let q =^def p_1 · · · p_n + 1</CMP>
        </definition>
        <derive xml:id="d5a">
          <h:p>For each p_i ∈ P we have q > p_i</h:p>
          <method xref="#Trivial"><premise xref="#d4"/></method>
        </derive>
        <derive xml:id="d5b">
          <h:p>q ∉ P</h:p>
          <method xref="#Trivial"><premise xref="#d5"/></method>
        </derive>
        <derive xml:id="d6">
          <h:p>We show absurdity by showing that q is prime</h:p>
          <FMP>⊥</FMP>
          <method xref="#Contradiction">
            <premise xref="#d5b"/>
            <proof>
              <derive xml:id="d7a">
                <h:p>
                  For each p_i ∈ P we have q = p_i k + 1 for a given natural number k.
                </h:p>
                <method xref="#By_Definition"><premise xref="#d1"/></method>
              </derive>
              <derive xml:id="d7b">
                <h:p>Each p_i ∈ P does not divide q</h:p>
              </derive>
              <derive xml:id="d8">
                <h:p>q is prime</h:p>
                <method xref="#Trivial">
                  <premise xref="#h2"/>
                  <premise xref="#p4"/>
                </method>
              </derive>
            </proof>
          </method>
        </derive>
      </proof>
    </method>
  </derive>
</proof>
```

`proof` elements are considered to be non-assertional in OMDoc, since they do not make assertions about mathematical objects themselves, but only justify such assertions. The assertional elements inside the proofs are governed by the scoping mechanisms discussed there, so that using them in a context where assertional elements are needed, can be forbidden.

[=proofobjects]

Chapter 38

Formal Proofs as Mathematical Objects

In OMDoc, the notion of fully formal proofs is accommodated by the `proofobject` element. In logic, the term **proof object** is used for term representations of formal proofs via the Curry/Howard/DeBruijn Isomorphism (see e.g. [**Thompson91**] for an introduction and Figure 38.1 for an example). λ-terms are among the most succinct representations of calculus-level proofs as they only document the inference rules. Since they are fully formal, they are very difficult to read and need specialized proof presentation systems for human consumption. In proof objects inference rules are represented as mathematical symbols, in our example in Figure 38.1 we have assumed a theory `PLOND` for the calculus of natural deduction in propositional logic which provides the necessary symbols (see Listing 38.1).

Definition 38.0.1 The **proofobject** element contains an optional multilingual group of `h:p` elements which describes the formal proof as well as a proof object which can be an OpenMath object, Content-MathML expression, or `legacy` element.

`proofobject`

Note that using OMDoc symbols for inference rules and mathematical objects for proofs reifies them to the object level and allows us to treat them at par with any other mathematical objects. We might have the following theory for natural deduction in propositional logic as a reference target for the second inference rule in Figure 38.1.

Listing 38.1: A Theory for Propositional Natural Deduction

```
<theory xml:id="PL0ND">
  <metadata>
    <dc:description>The Natural Deduction Calculus for Propositional Logic</dc:description>
  </metadata>
  ...
  <symbol name="andI">
    <metadata><dc:subject>Conjunction Introduction</dc:subject></metadata>
    <type system="prop−as−types">A → B → (A ∧ B)</type>
  </symbol>

  <definition xml:id="andI.def" for="andi">
    <h:p>Conjunction introduction, if we can derive A and B,
      then we can conclude A ∧ B.</h:p>
  </definition>
  ...
</theory>
```

In particular, it is possible to use a `definition` element to define a derived inference rule by simply specifying the proof term as a definiens:

```
<symbol name="andcom">
  <metadata><dc:description>Commutativity for ∧</dc:description></metadata>
  <type system="prop−as−types">(A ∧ B) → (B ∧ A)</type>
</symbol>
```

$$\dfrac{\dfrac{\dfrac{[A \wedge B]}{B}\wedge E_r \quad \dfrac{[A \wedge B]}{A}\wedge E_l}{B \wedge A}\wedge I}{A \wedge B \Rightarrow B \wedge A}\Rightarrow I$$

```
<proofobject xml:id="ac.p" for="#and−comm">
 <metadata>
  <dc:description>
   Assuming A ∧ B we have B and A
   from which we can derive B ∧ A.
  </dc:description>
 </metadata>
 <OMBIND id="andcom.pf">
  <OMS cd="PL0ND" name="impliesI"/>
  <OMBVAR>
   <OMATTR>
    <OMATP>
     <OMS cd="PL0ND" name="type"/>
     A ∧ B
    </OMATP>
    <OMV name="X"/>
   </OMATTR>
  </OMBVAR>
  <OMA>
   <OMS cd="PL0ND" name="andI"/>
   <OMA>
    <OMA>
     <OMS cd="PL0ND" name="andEr"/>
     <OMV name="X"/>
    </OMA>
    <OMA>
     <OMS cd="PL0ND" name="andEl"/>
     <OMV name="X"/>
    </OMA>
   </OMA>
  </OMA>
 </OMBIND>
</proofobject>
```

The schema on the left shows the proof as a natural deduction proof tree, the OMDoc representation gives the proof object as a λ term. This term would be written as the following term in traditional (mathematical) notation: $\Rightarrow I(\lambda X : A \wedge B. \wedge I(\wedge E_r(X), \wedge E_l(X)))$

Figure 38.1: A Proof Object for the Commutativity of Conjunction

```
<definition xml:id="andcom.def" for="#andcom" type="simple">
  <OMR href="#andcom.pf"/>
</definition>
```

Like `proofs`, `proofobjects` elements are considered to be non-assertional in OMDoc, since they do not make assertions about mathematical objects themselves, but only justify such assertions.

Part XII

Complex Theories (Modules CTH and DG)

In Chapter 15 we have presented a notion of theory and inheritance that is sufficient for simple applications like content dictionaries that informally (though presumably rigorously) define the static meaning of symbols. Experience in e.g. program verification has shown that this infrastructure is insufficient for large-scale developments of formal specifications, where reusability of formal components is the key to managing complexity. For instance, for a theory of rings we cannot simply inherit the same theory of monoids as both the additive and multiplicative structure.

In this chapter, we will generalize the inheritance relation from Chapter 15 to that of "theory inclusions", also called "theory morphisms" or "theory interpretations" elsewhere [**Farmer93**]. This infrastructure allows to structure a collection of theories into a complex theory graph that particularly supports modularization and reuse of parts of specifications and theories. This gives rise to the name "complex theories" of the OMDoc module.

Element	Attributes		M	Content
	Required	Optional	D	
`theory`		`xml:id, class, style`	+	`(`⟪*top-level*⟫` | imports | inclusion)*`
`imports`	`from`	`xml:id, type, class, style, conservativity, conservativity-just`	+	`morphism?`
`morphism`		`xml:id, base, class, style, type, hiding, consistency, exhaustivity`	−	`requation*, measure?, ordering?`
`inclusion`	`via`	`xml:id, conservativity, conservativity-just`	−	`EMPTY`
`theory-inclusion`	`from, to`	`xml:id, class, style, conservativity, conservativity-just`	+	`(CMP*,FMP*, morphism, obligation*)`
`axiom-inclusion`	`from, to`	`xml:id, class, style, conservativity, conservativity-just`	+	`morphism?, obligation*`

Figure 38.2: Complex Theories in OMDoc

Chapter 39

Inheritance via Translations

[=morphisms]

Literal inheritance of symbols is often insufficient to re-use mathematical structures and theories efficiently. Consider for instance the situation in the elementary algebraic hierarchy: for a theory of rings, we should be able to inherit the additive group structure from the theory `group` of groups and the structure of a multiplicative monoid from the theory `monoid`: A ring is a set R together with two operations $+$ and $*$, such that $(R,+)$ is a group with unit 0 and inverse operation $-$ and $(R^*,*)$ is a monoid with unit 1 and base set $R^*{:=}\{r\in R|r\neq 0\}$. Using the literal inheritance regime introduced so far, would lead us into a duplication of efforts as we have to define theories for semigroups and monoids for the operations $+$ and $*$ (see Figure 39.1).

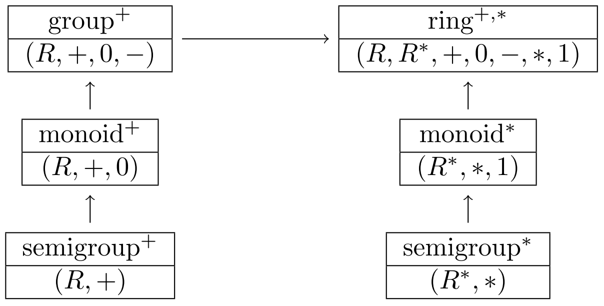

Figure 39.1: A Theory of Rings via Simple Inheritance

This problem[1] can be alleviated by allowing theory inheritance via translations. Instead of literally inheriting the symbols and axioms from the source theory, we involve a symbol mapping function (we call this a **morphism**) in the process. This function maps source formulae (i.e. built up exclusively from symbols visible in the source theory) into formulae in the target theory by translating the source symbols.

Figure 39.2 shows a theory graph that defines a theory of rings by importing the monoid axioms via the morphism σ. With this translation, we do not have to duplicate the `monoid` and `semigroup` theories and can even move the definition of $\cdot^*$ operator into the theory of monoids, where it intuitively belongs[2].

Formally, we extend the notion of inheritance given in Chapter 15 by allowing a target theory to import another a source theory **via a morphism** : Let $\mathcal{S}$ be a theory with theory-constitutive

[1]which seems negligible in this simple example, but in real life, each instance of multiple inheritance leads to a *multiplication* of all dependent theories, which becomes an exponentially redundant management nightmare.

[2]On any monoid $M=(S,\circ,e)$, we have the $\cdot^*$ operator, which converts a set $S\subseteq M$ in to $S^*{:=}\{r\in S|r\neq e\}$

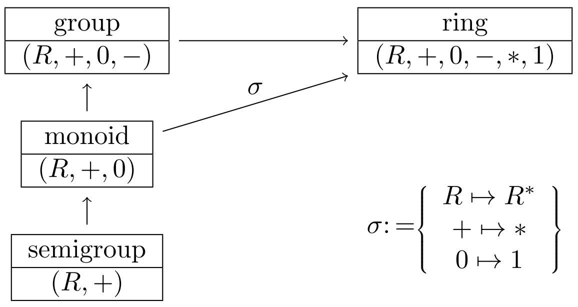
group
$(R,+,0,-)$
ring
$(R,+,0,-,*,1)$
σ
monoid
$(R,+,0)$
semigroup
$(R,+)$
$\sigma\colon=\left\{\begin{array}{c} R\mapsto R^* \\ +\mapsto * \\ 0\mapsto 1\end{array}\right\}$

Figure 39.2: A Theory of Rings via Morphisms

elements[3] $t_1,\ldots,t_n$ and $\sigma\colon\mathcal{S}\to\mathcal{T}$ a morphism, if we declare that $\mathcal{T}$ imports $\mathcal{S}$ via σ, then $\mathcal{T}$ **inherit** s the theory-constitutive statements $\sigma(t_i)$ from $\mathcal{S}$. For instance, the theory of rings inherits the axiom $\forall x.x+0=x$ from the theory of monoids as $\sigma(\forall x.x+0=x)=\forall x.x*1=x$.

`morphism`

Definition 39.0.1 To specify the formula mapping function, module CTH extends the `imports` element by allowing it to have a child element **morphism** , which specifies a formula mapping by a set of recursive equations using the `requation` element described in Section 12.3. The optional attribute `type` allows to specify whether the function is really recursive (value `recursive`) or pattern-defined (value `pattern`).

As in the case of the `definition` element, termination of the defined function can be specified using the optional child elements `measure` and `ordering`, or the optional attributes `uniqueness` and `existence`, which point to uniqueness and existence assertions. Consistency and exhaustivity of the recursive equations are specified by the optional attributes `consistency` and `exhaustivity`.

Listing 39.1 gives the OMDoc representation of the theory graph in Figure 39.2, assuming the theories in Listing 16.2.

Listing 39.1: A Theory of Rings by Inheritance Via Renaming

```
<theory xml:id="ring">
  <symbol name="times"/><symbol name="one"/>
  <imports xml:id="add.import" from="#group" type="global"/>
  <imports xml:id="mult.import" from="#monoid" type="global">
    <morphism>
      <requation>
        <OMS cd="monoid" name="set"/>
        <OMA><OMS cd="monoid" name="setstar"/>
          <OMS cd="semigroup" name="set"/>
        </OMA>
      </requation>
      <requation>
        <OMS cd="monoid" name="op"/>
        <OMS cd="ring" name="times"/>
      </requation>
      <requation>
        <OMS cd="monoid" name="neut"/>
        <OMS cd="ring" name="one"/>
      </requation>
    </morphism>
  </imports>
  <axiom xml:id="ring.distribution">
    <CMP><OMS cd="semigroup" name="op"/> distributes over
      <OMS cd="ring" name="times"/>
    </CMP>
  </axiom>
</theory>
```

To conserve space and avoid redundancy, OMDoc morphisms need only specify the values of symbols that are translated; all other symbols are inherited literally. Thus the set of symbols

[3]which may in turn be inherited from other theories

inherited by an `imports` element consists of the symbols of the source theory that are not in the domain of the morphism. In our example, the symbols R, $+$, 0, $-$, $*$, 1 are visible in the theory of rings (and any other symbols the theory of semigroups may have inherited). Note that we do not have a name clash from multiple inheritance.

Finally, it is possible to hide symbols from the source theory by specifying them in the `hiding` attribute. The intended meaning is that the underlying signature mapping is defined (total) on all symbols in the source theory except on the hidden ones. This allows to define symbols that are local to a given theory, which helps achieve data protection. Unfortunately, there is no simple interpretation of hiding in the general case in terms of formula translations, see [**CoFI:2004:CASL-RM**; **MAH-06-a**] for details. The definition of hiding used there is more general. The variant used here arises as the special case where the hiding morphism, which goes against the import direction, is an inclusion; then the symbols that are not in the image are the hidden ones. If we restrict ourselves to hiding defined symbols, then the situation becomes simpler to understand: A morphism that hides a (defined) symbol s will translate the theory-constitutive elements of the source theory by expanding definitions. Thus s will not be present in the target theory, but all the contributions of the theory-constitutive elements of the source theory will have been inherited. Say, we want to define the concept of a sorting function, i.e. a function that — given a list L as input — returns a returns a permutation L' of L that is ordered. In the situation depicted in Figure 41.1, we would the concept of an ordering function (a function that returns a permutation of the input list that is ordered) with the help of predicates `perm` and `ordered`. Since these are only of interest in the context of the definition of the latter, they would typically be hidden in order to refrain from polluting the name space.

As morphisms often contain common prefixes, the `morphism` element has an optional `base` attribute, which points to a chain of morphisms, whose composition is taken to be the base of this morphism. The intended meaning is that the new morphism coincides as a function with the base morphism, wherever the specified pattern do not match, otherwise their corresponding values take precedence over those in the base morphism. Concretely, the `base` contains a whitespace-separated list of URI references to `theory-inclusion`, `axiom-inclusion`, and `imports` elements. Note that the order of the references matters: they are ordered in order of the path in the local chain, i.e if we have `base="#`⟪*ref1*⟫`...#`⟪*refn*⟫`"` there must be theory inclusions σ_i with `xml:id="`⟪*refi*⟫`"`, such that the target theory of σ_{i-1} is the source theory of σ_i, and such that the source theory of σ_1 and the target theory of σ_n are the same as those of the current theory inclusion.

Finally, the CTH module adds two the optional attributes `conservativity` and `conservativity-just` to the `imports` element for stating and justifying conservativity (see the discussion below).

Chapter 40

Postulated Theory Inclusions

[=theory-morphisms]

We have seen that inheritance via morphisms provides a powerful mechanism for structuring and re-using theories and contexts. It turns out that the distinguishing feature of theory morphisms is that all theory-constitutive elements of the source theory are valid in the target theory (possibly after translation). This can be generalized to obtain even more structuring relations and thus possibilities for reuse among theories. Before we go into the OMDoc infrastructure, we will briefly introduce the mathematical model (see e.g. [**Hutter:mocsv00**] for details).

A **theory inclusion** from a **source theory** $\mathcal{S}$ to a **target theory** $\mathcal{T}$ is a mapping σ from $\mathcal{S}$ objects[1] to those of $\mathcal{T}$, such that for every theory-constitutive statement $\mathbf{S}$ of $\mathcal{S}$, $\sigma(\mathbf{S})$ is provable in $\mathcal{T}$ (we say that $\sigma(\mathbf{S})$ is a **$\mathcal{T}$-theorem**).

In OMDoc, we weaken this logical property to a structural one: We say that a theory-constitutive statement $\mathbf{S}$ in theory $\mathcal{S}$ is **structurally included** in theory $\mathcal{T}$ via σ, if there is an assertional element $\mathbf{T}$ in $\mathcal{T}$, such that the content of $\mathbf{T}$ is $\sigma(\mathbf{S})$. Note that strictly speaking, σ is only defined on formulae, so that if a statement $\mathbf{S}$ is only given by a `CMP`, $\sigma(\mathbf{S})$ is not defined. In such cases, we assume $\sigma(\mathbf{S})$ to contain a `CMP` element containing suitably translated mathematical vernacular.

Definition 40.0.1 In this view, a **structural theory inclusion** from $\mathcal{S}$ to $\mathcal{T}$ is a morphism $\sigma\colon\mathcal{S}\to\mathcal{T}$, such that every theory-constitutive element is structurally included in $\mathcal{T}$.

Note that an `imports` element in a theory $\mathcal{T}$ with source theory $\mathcal{S}$ as discussed in Chapter 38 induces a theory inclusion from $\mathcal{S}$ into $\mathcal{T}$[2] (the theory-constitutive statements of $\mathcal{S}$ are accessible in $\mathcal{T}$ after translation and are therefore structurally included trivially). We call this kind of theory inclusion **definitional** , since it is a theory inclusion by virtue of the definition of the target theory. For all other theory inclusions (we call them **postulated theory inclusion** s), we have to establish the theory inclusion property by proving the translations of the theory-constitutive statements of the source theory (we call these translated formulae **proof obligation**).

The benefit of a theory inclusion is that all theorems, proofs, and proof methods of the source theory can be used (after translation) in the target theory (see Chapter 41). Obviously, the transfer approach only depends on the theorem inclusion property, and we can extend its utility by augmenting the theory graph by more theory morphisms than just the definitional ones (see [**FaGu93**] for a description of the IMPS theorem proving system that makes heavy use of this idea). We use the infrastructure presented in this chapter to structure a collection of theories as a graph — the **theory graph** — where the nodes are theories and the links are theory inclusions (definitional and postulated ones).

[1] Mathematical objects that can be represented using the only symbols of the source theory $\mathcal{S}$.

[2] Note that in contrast to the inheritance relation induced by the `imports` elements the relation induced by general theory inclusions may be cyclic. A cycle just means that the theories participating in it are semantically equivalent.

We call a theory inclusion $\sigma\colon\mathcal{S}\to\mathcal{T}$ **conservative** , iff $\mathbf{A}$ is already a $\mathcal{S}$-theorem for all $\mathcal{T}$-theorems of the from $\sigma(\mathbf{A})$. If the morphism σ is the identity, then this means the local axioms in $\mathcal{T}$ only affect the local symbols of $\mathcal{T}$, and do not the part inherited from $\mathcal{S}$. In particular, conservative extensions of consistent theories cannot be inconsistent. For instance, if all the local theory-constitutive elements in $\mathcal{T}$ are symbol declarations with definitions, then conservativity is guaranteed by the special form of the definitions. We can specify conservativity of a theory inclusion via the `conservativity`. The values `conservative` and `conservative` are used for the two cases discussed above. There is a third value: `conservative`, which we will not explain here, but refer the reader to [**MAH-06-a**].

`theory-inclusion`

Definition 40.0.2 OMDoc implements the concept of postulated theory inclusions in the top-level **theory-inclusion** element. It has the required attributes `from` and `to`, which point to the source- and target theories and contains a `morphism` child element as described above to define the translation function. A subsequent (possibly empty) set of `obligation` elements can be used to mark up proof obligations for the theory-constitutive elements of the source theory.

`obligation`

Definition 40.0.3 An **obligation** is an empty element whose `assertion` attribute points to an `assertion` element that states that the theory-constitutive statement specified by the `induced-by` (translated by the morphism in the parent `theory-inclusion`) is provable in the target theory. Note that a `theory-inclusion` element must contain `obligation` elements for all theory-constitutive elements (inherited or local) of the source theory to be correct.

Listing 40.1 shows a theory inclusion from the theory `group` defined in Listing 16.2 to itself. The morphism just maps each element of the base set to its inverse. A good application for this kind of theory morphism is to import claims for symmetric (e.g. with respect to the function `inv`, which serves as an involution in `group`) cases via this theory morphism to avoid explicitly having to prove them (see Chapter 41).

Listing 40.1: A Theory Inclusion for Groups

```
<assertion xml:id="conv.assoc">∀x, y, z ∈ M.z ∘ (y ∘ x) = (z ∘ y) ∘ x</assertion>
<assertion xml:id="conv.closed" theory="semigroup">∀x, y ∈ M.y ∘ x ∈ M</assertion>
<assertion xml:id="left.unit" theory="monoid">∀x ∈ M.e ∘ x = x</assertion>
<assertion xml:id="conv.inv" theory="group">∀x, y ∈ M.x ∘ x^{-1} = e</assertion>
<theory-inclusion xml:id="grp-conv-grp" from="#group" to="#group">
  <morphism><requation>X ∘ Y ⇝ Y ∘ X</requation></morphism>
  <obligation assertion="#conv.closed" induced-by="#closed.ax"/>
  <obligation assertion="#conv.assoc" induced-by="#assoc.ax"/>
  <obligation assertion="#left.unit" induced-by="#unit.ax"/>
  <obligation assertion="#conv.inv" induced-by="#inv.ax"/>
</theory-inclusion>
```

Chapter 41

Local- and Required Theory Inclusions

[=restinf]

In some situations, we need to pose well-definedness conditions on theories, e.g. that a specification of a program follows a certain security model, or that a parameter theory used for actualization satisfies the assumptions made in the formal parameter theory; (see [**Kohlhase:OMDoc1.6primer**] for a discussion). If these conditions are not met, the theory intuitively does not make sense. So rather than simply stating (or importing) these assumptions as theory-constitutive statements — which would make the theory inconsistent, when they are not met — they can be stated as well-definedness conditions. Usually, these conditions can be posited as theory inclusions, so checking these conditions is a purely structural matter, and comes into the realm of OMDoc's structural methods.

Definition 41.0.1 OMDoc provides the empty **inclusion** element for this purpose. It can occur anywhere as a child of a `theory` element and its `via` attribute points to a theory inclusion, which is required to hold in order for the parent theory to be well-defined.

`inclusion`

If we consider for instance the situation in Figure 41.1[1]. There we have a theory `OrdList` of lists that is generic in the elements (which is assumed to be a totally ordered set, since we want to talk about ordered lists). We want to to instantiate `OrdList` by applying it to the theory `NatOrd` of natural numbers and obtain a theory `NatOrdList` of lists of natural numbers by importing the theory `OrdList` in `NatOrdList`. This only makes sense, if `NatOrd` is a totally ordered set, so we add an `inclusion` element in the statement of theory `NatOrdList` that points to a theory inclusion of `TOSet` into `OrdNat`, which forces us to verify the axioms of `TOSet` in `OrdNat`.

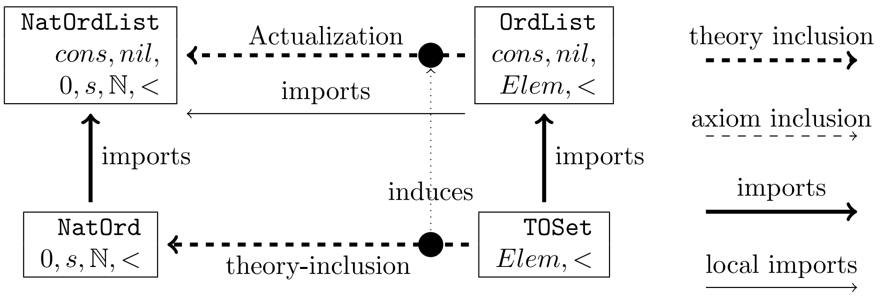

Figure 41.1: A Structured Specification of Lists (of Natural Numbers)

[1]This example is covered in detail in [**Kohlhase:OMDoc1.6primer**].

Furthermore note, that the inclusion of `OrdList` into `NatOrdList` should not include the `TOSet` axioms on orderings, since this would defeat the purpose of making them a precondition to well-definedness of the theory `NatOrdList`. Therefore OMDoc follows the "development graph model" put forward in [**Hutter:mocsv00**] and generalizes the notion of theory inclusions even further: A formula mapping between theories $\mathcal{S}$ and $\mathcal{T}$ is called a **local theory inclusion** or **axiom inclusion** , if the theory inclusion property holds for the local theory-constitutive statements of the source theory.

To distinguish this from the notion of a proper theory inclusion — where the theory inclusion property holds for all theory constitutive statements of $\mathcal{S}$ (even the inherited ones) — we call the latter one **global** . Of course all global theory inclusions are also local ones, so that the new notion is a true generalization. Note that the structural inclusions of an axiom inclusion are not enough to justify translated source theorems in the target theory.

To allow for a local variant of inheritance, the CTH module adds an attribute `type` to the `imports` element. This can take the values `global` (the default) and `local`. In the latter case, only the theory-constitutive statements that are local to the source theory are imported.

`axiom-inclusion` **Definition 41.0.2** Furthermore, the CTH module introduces the **axiom-inclusion** element for local theory inclusions. This has the same attributes as `theory-inclusion`: `from` to specify source theory, `to` for the target theory. It also allows `obligation` elements as children.

Chapter 42

Induced Assertions and Expositions

[=assertionvia]

The main motivation of theory inclusions is to be able to transport mathematical statements from the source theory to the target theory. In OMDoc, this operation can be made explicit by the attributes `generated-from` and `generated-via` that the module CTH adds to all mathematical statements. On a statement **T**, the second attribute points to a theory inclusion σ whose target is (imported into the) current theory, the first attribute points to a statement **S** in that theory which is of the same type (i.e. has the same OMDoc element name) as **T**. The content of **T** must be (equivalent to) the content of **S** translated by the morphism of σ.

In the context of the theory inclusion in Listing 40.1, we might have the following situation:

Listing 42.1: Translating a Statement via a Theory Inclusion

```
<assertion xml:id="foo" type="theorem">...</assertion>
<proof xml:id="foo.pf" for="#foo">...</proof>
<assertion xml:id="target" induced-by="#foo" induced-via="#grp-conv-grp">
  ...
</assertion>
```

Here, the second assertion is induced by the first one via the theory inclusion in Listing 40.1, the statement of the theorem is about the inverses. In particular, the proof of the second theorem comes for free, since it can also be induced from the proof of the first one.

In particular we see that in OMDoc documents, not all statements are automatically generated by translation e.g. the proof of the second assertion is not explicitly stated. Mathematical knowledge management systems like knowledge bases might choose to do so, but at the document level we do not mandate this, as it would lead to an explosion of the document sizes. Of course we could cache the transformed proof giving it the same "cache attribute state".

Note that not only statements like assertions and proofs can be translated via theory inclusions, but also whole documents: Say that we have course materials for elementary algebra introducing monoids and groups via left units and left inverses, but want to use examples and exercises from a book that introduces them using right units and right inverses. Assuming that both are formalized in OMDoc, we can just establish a theory morphism much like the one in Listing 40.1. Then we can automatically translate the exercises and examples via this theory inclusion to our own setting by just applying the morphism to all formulae in the text[1] and obtain exercises and examples that mesh well with our introduction. Of course there is also a theory inclusion in the other direction, which is an inverse, so our colleague can reuse our course materials in his right-leaning setting.

[1]There may be problems, if mathematical statements are verbalized; this can currently not be translated directly, since it would involve language processing tools much beyond the content processing tools described in this book. For the moment, we assume that the materials are written in a controlled subset of mathematical vernacular that avoids these problems.

Another example is the presence of different normalization factors in physics or branch cuts in elementary complex functions. In both cases there is a plethora of definitions, which all describe essentially the same objects (see e.g. [**BraCor:raefca02**] for an overview over the branch cut situation). Reading materials that are based on the “wrong” definition is a nuisance at best, and can lead to serious errors. Being able to adapt documents by translating them from the author theory to the user theory by a previously established theory morphism can alleviate both.

Mathematics and science are full of such situations, where objects can be viewed from different angles or in different representations. Moreover, no single representation is “better” than the other, since different views reveal or highlight different aspects of the object (see [**KohKoh:esmk05**] for a systematic account). Theory inclusions seem uniquely suited to formalize the structure of different views in mathematics and their interplay, and the structural markup for theories in OMDoc seems an ideal platform for offering added-value services that feed on these structures without committing to a particular formalization or foundation of mathematics.

Chapter 43

Development Graphs (Module DG)

[=dgraph]

The OMDoc module DG for development graphs complements module CTH with high-level justifications for the theory inclusions. Concretely, the module provides an infrastructure for dealing efficiently with the proof obligations induced by theory inclusions and forms the basis for a managementoftheory change. We anticipate that the elements introduced in this chapter will largely be hidden from the casual user of mathematical software systems, but will form the basis for high-level document- and mathematical knowledge management services.

43.1 Introduction

As we have seen in the example in Listing 40.1, the burden of specifying an `obligation` element for each theory-constitutive element of the source theory can make the establishment of a theory inclusion quite cumbersome — theories high up in inheritance hierarchies can have a lot (often hundreds) of inherited, theory-constitutive statements. Even more problematically, such obligations are a source of redundancy and non-local dependencies, since many of the theory-constitutive elements are actually inherited from other theories.

Consider for instance the situation in Figure 43.1, where we are interested in the top theory inclusion Γ. On the basis of theories $\mathcal{T}_1$ and $\mathcal{T}_2$, theory $\mathcal{C}_1$ is built up via theories $\mathcal{A}_1$ and $\mathcal{B}_1$. Similarly, theory $\mathcal{C}_2$ is built up via $\mathcal{A}_2$ and $\mathcal{B}_2$ (in the latter, we have a non-trivial non-trivial morphism σ). Let us assume for the sake of this argument that for $\mathcal{X}_i \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}\}$ theories $\mathcal{X}_1$ and $\mathcal{X}_2$ are so similar that axiom inclusions (they are indicated by thin dashed arrows in Figure 43.1 and have the formula-mappings α, β, and γ) are easy to prove[1].

To justify Γ, we must prove that the Γ-translations of all the theory-constitutive statements of $\mathcal{C}_1$ are provable in $\mathcal{C}_2$. So let statement $\mathbf{B}$ be theory-constitutive for $\mathcal{C}_1$, say that it is local in $\mathcal{B}_1$, then we already know that $\beta(\mathbf{B})$ is provable in $\mathcal{B}_2$ since β is an axiom inclusion. Moreover, we know that $\sigma(\beta(\mathbf{B}))$ is provable in $\mathcal{C}_2$, since σ is a (definitional, global) theory inclusion. So, if we have $\Gamma = \sigma \circ \beta$, then we are done for $\mathbf{B}$ and in fact for all local statements of $\mathcal{B}_1$, since the argument is independent of $\mathbf{B}$. Thus, we have established the existence of an axiom inclusion from $\mathcal{B}_1$ to $\mathcal{C}_2$ simply by finding suitable inclusions and checking translation compatibility.

Definition 43.1.1 We will call a situation, where a theory $\mathcal{T}$ can be reached by an axiom inclusion with a subsequent chain of theory inclusions a **local chain** (with morphism $\tau := \sigma_n \circ \cdots \circ \sigma_1 \circ \sigma$),

[1]A common source of situations like this is where the $\mathcal{X}_2$ are variants of the $\mathcal{X}_1$ theories. Here we might be interested whether $\mathcal{C}_2$ still proves the same theories (and often also in the converse theory inclusion Γ^{-1} that would prove that the variants are equivalent).

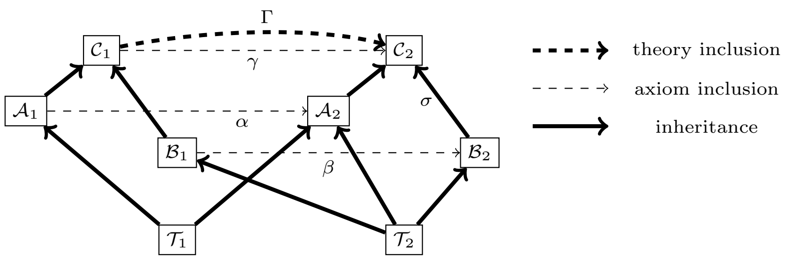

Figure 43.1: A Development Graph with Theory Inclusions

if $\mathcal{S} \xrightarrow{\sigma} \mathcal{T}_1$ is an axiom inclusion or (local theory import) and $\mathcal{T}_i \xrightarrow{\sigma_i} \mathcal{T}_{i+1}$ are theory inclusions (or local theory import).

$$\begin{tikzcd}
\mathcal{S} \arrow[r, dashed, "\sigma"'] \arrow[rrrrr, dashed, bend left, "\tau = \sigma_n \circ \cdots \circ \sigma_1 \circ \sigma"] & \mathcal{T}_1 \arrow[r, dashed, "\sigma_1"'] & \mathcal{T}_2 \arrow[r, dashed, "\sigma_2"'] & \cdots \arrow[r, dashed, "\sigma_{n-1}"'] & \mathcal{T}_n \arrow[r, dashed, "\sigma_n"'] & \mathcal{T}
\end{tikzcd}$$

Note that by an argument like the one for **B** above, a local chain justifies an axiom inclusion from $\mathcal{S}$ into $\mathcal{T}$: all the τ-translations of the local theory-constitutive statements in $\mathcal{S}$ are provable in $\mathcal{T}$.

In our example in Figure 43.1 — given the obvious compatibility assumptions on the morphisms which we have not marked in the figure, — we can justify four new axiom inclusions from the theories $\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{A}_1$, and $\mathcal{B}_1$ into $\mathcal{C}_2$ by the following local chains[2].

$$\begin{tikzcd}
\mathcal{T}_2 \arrow[r] & \mathcal{B}_2 \arrow[r, "\sigma"] & \mathcal{C}_2 & \mathcal{B}_1 \arrow[r, dashed, "\beta"] & \mathcal{B}_2 \arrow[r, "\sigma"] & \mathcal{C}_2 \\
\mathcal{T}_1 \arrow[r] & \mathcal{A}_2 \arrow[r] & \mathcal{C}_2 & \mathcal{A}_1 \arrow[r, dashed, "\alpha"] & \mathcal{A}_2 \arrow[r] & \mathcal{C}_2
\end{tikzcd}$$

Thus, for each theory $\mathcal{X}$ that $\mathcal{C}_1$ inherits from, there is an axiom inclusion into $\mathcal{C}_2$. So for any theory-constitutive statement in $\mathcal{C}_1$ (it must be local in one of the $\mathcal{X}$) we know that it is provable in $\mathcal{C}_2$; in other words Γ is a theory inclusion if it is compatible with the morphisms of these axiom inclusions. We have depicted the situation in Figure 43.2.

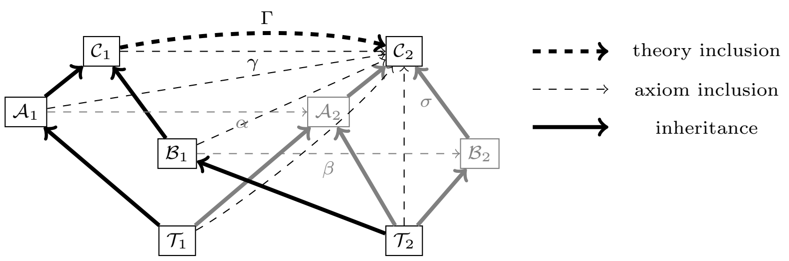

Figure 43.2: A Decomposition for the theory inclusion Γ

We call a situation where we have a formula mapping $\mathcal{S} \xrightarrow{\sigma} \mathcal{T}$, and an axiom inclusion $\mathcal{X} \xrightarrow{\sigma_{\mathcal{X}}} \mathcal{T}$ for every theory $\mathcal{X}$ that $\mathcal{S}$ inherits from a **decomposition** for σ, if the $\sigma_{\mathcal{X}}$ and σ are compatible. As we have seen in the example above, a decomposition for σ can be used to justify that σ a theory inclusion: all theory-constitutive elements in $\mathcal{S}$ are local in itself or one of the theories $\mathcal{X}$ it inherits from. So if we have axiom inclusions from all of these to $\mathcal{T}$, then all obligations induced by them are justified and σ is indeed a theory inclusion.

[2]Note for the leftmost two chains use the fact that theory inclusions (in our case definitional ones) are also axiom inclusions by definition.

43.2 An OMDoc Infrastructure for Development Graphs (Module DG)

decomposition

Definition 43.2.1 The DG module provides the **decomposition** element to model justification by decomposition situations. This empty element can occur at top-level or inside a `theory-inclusion` element.

The `decomposition` element can occur as a child to a `theory-inclusion` element and carries the required attribute `links` that contains a whitespace-separated list of URI references to the `axiom-` and `theory-inclusion` elements that make up the decomposition situation justifying the parent `theory-inclusion` element. Note that the order of references in `links` is irrelevant. If the `decomposition` appears on top-level, then the optional `for` attribute must be used to point to the `theory-inclusion` it justifies. In this situation the `decomposition` element behaves towards a `theory-inclusion` much like a `proof` for an `assertion`.

Element	Attributes		M	Content
	Required	Optional	D	
decomposition	links		–	EMPTY
path-just	local, globals	for	–	EMPTY
theory-inclusion	from, to, by	xml:id, class, style	+	(CMP*,FMP*, morphism, (decomposition* | obligation*))
axiom-inclusion	from, to	xml:id, class, style	+	morphism?, (path-just* | obligation*)

Figure 43.3: Development Graphs in OMDoc

Furthermore module DG provides `path-just` elements as children to the `axiom-inclusion` elements to justify that this relation holds, much like a `proof` element provides a justification for an `assertion` element for some property of mathematical objects.

path-just

Definition 43.2.2 A **path-just** element justifies an `axiom-inclusion` by reference to other `axiom-inclusion` or `theory-inclusion` elements. **Local chains** are encoded in the empty `path-just` element via the required attributes `local` (for the first `axiom-inclusion`) and the attribute `globals` attribute, which contains a whitespace-separated list of URI references to `theory-inclusions`. Note that the order of the references in the `globals` matters: they are ordered in order of the path in the local chain, i.e if we have `globals="... #ref1 #ref2 ..."` there must be theory inclusions σ_i with `xml:id="ref`i`"`, such that the target theory of σ_1 is the source theory of σ_2.

Like the `decomposition` element, `path-just` can appear at top-level, if it specifies the `axiom-inclusion` it justifies in the (otherwise optional) `for` attribute.

Let us now fortify our intuition by casting the situation in Listings 43.1 to 43.2 in OMDoc syntax. Another — more mathematical — example is carried out in detail in [**Kohlhase:OMDoc1.6primer**].

Listing 43.1: The OMDoc representation of the theories in Figure 43.1.

```
<theory xml:id="t1">...</theory>          <theory xml:id="t2">...</theory>

<theory xml:id="a1">                         <theory xml:id="b1">
  <imports xml:id="ima1" from="#t1"/>          <imports xml:id="imb1" from="#t2"/>
  <axiom xml:id="axa11">...</axiom>             <axiom xml:id="axb11">...</axiom>
  <axiom xml:id="axa12">...</axiom>           </theory>
</theory>

<theory xml:id="a2">                         <theory xml:id="b2">
  <imports xml:id="im1a2" from="#t1"/>          <imports xml:id="imb2" from="#t2"/>
  <imports xml:id="im2a2" from="#t2"/>
  <axiom xml:id="axa21">...</axiom>               <axiom xml:id="axb21">...</axiom>
</theory>                                    </theory>

<theory xml:id="c1">                         <theory xml:id="c2">
```

```
  <imports xml:id="im1c1" from="#a1"/>        <imports xml:id="im1c2" from="#a2"/>
  <imports xml:id="im2c1" from="#b1"/>        <imports xml:id="im2c2" from="#b2"/>
  <axiom xml:id="axc11">...</axiom>             <axiom xml:id="axc21">...</axiom>
</theory>                                    </theory>
```

Here we set up the theory structure with the theory inclusions given by the `imports` elements (without morphism to simplify the presentation). Note that these have `xml:id` attributes, since we need them to construct axiom- and theory inclusions later. We have also added axioms to induce proof obligations in the axiom inclusions:

Listing 43.2: The OMDoc Representation of the Inclusions in Figure 43.1.

```
<axiom−inclusion xml:id="aia" from="#a1" to="#a2">
  <obligation induced−by="#axa11" assertion="#th−axa11"/>
  <obligation induced−by="#axa12" assertion="#th−axa12"/>
</axiom−inclusion>

<axiom−inclusion xml:id="bib" from="#b1" to="#b2">
  <obligation induced−by="#axb11" assertion="#th−axb1"/>
</axiom−inclusion>

<axiom−inclusion xml:id="cic" from="#c1" to="#c2">
  <obligation induced−by="#axc11" assertion="#th−axc1"/>
</axiom−inclusion>
```

We leave out the actual assertions that justify the `obligation`s to conserve space. From the axiom inclusions, we can now build four more via path justifications:

Listing 43.3: The Induced Axiom Inclusions in Figure 43.1.

```
<axiom−inclusion xml:id="t1ic" from="#t1" to="#c2">
 <path−just local="#im1a2" globals="#im1c2"/>
</axiom−inclusion>

<axiom−inclusion xml:id="t2ic" from="#t2" to="#c2">
 <path−just local="#imb2" globals="#im2c2"/>
</axiom−inclusion>

<axiom−inclusion xml:id="aic" from="#a1" to="#c2">
 <path−just local="#aia" globals="#im1c2"/>
</axiom−inclusion>

<axiom−inclusion xml:id="bic" from="#b1" to="#c2">
 <path−just local="#bib" globals="#im2c2"/>
</axiom−inclusion>
```

Note that we could also have justified the axiom inclusion `t2ic` with two local paths: via the theory $\mathcal{A}_2$ and via $\mathcal{B}_2$ (assuming the translations work out). These alternative justifications make the development graph more robust against change; if one fails, the axiom inclusion still remains justified. Finally, we can assemble all of this information into a decomposition that justifies the theory inclusion Γ:

```
<theory−inclusion xml:id="tcic" from="#c1" to="#c2">
  <decomposition links="#t1ic #t2ic #aic #bic #cic"/>
</theory−inclusion>
```

Part XIII

Notation and Presentation (Module PRES)

[=presintro] As we have seen, OMDoc is concerned mainly with the content and structure of mathematical documents, and offers a complex infrastructure for dealing with that. However, mathematical texts often carry typographic conventions that cannot be determined by general principles alone. Moreover, non-standard presentations of fragments of mathematical texts sometimes carry meanings that do not correspond to the mathematical content or structure proper. In order to accommodate this, OMDoc provides a limited functionality for embedding style information into the document. BOP:45

Element	Attributes		Content
	Required	Optional	
`omstyle`	`element`	`for, xml:id, xref, class, style`	`(style|xslt)*`

Figure 43.4: The OMDoc Elements for Notation Information

The normal (but of course not the only) way to generate presentation from XML documents is to use XSLT style sheets (see for other applications). XSLT [**Clark:xslt99**] is a general transformation language for XML. XSLT programs (often called **style sheet** s) consist of a set of **template** s (rules for the transformation of certain nodes in the XML tree). These templates are recursively applied to the input tree to produce the desired output.

The general approach to presentation and notation in OMDoc is not to provide general-purpose presentational primitives that can be sprinkled over the document, since that would distract the author from the mathematical content, but to support the specification of general style information for OMDoc elements and mathematical symbols in separate elements.

In the case of a single OMDoc document it is possible to write a specialized style sheet that transforms the content-oriented markup used in the document into mathematical notation. However, if we have to deal with a large collection of OMDoc representations, then we can either write a specialized style sheet for each document (this is clearly infeasible to do by hand), or we can develop a style sheet for the whole collection (such style sheets tend to get large and unmanageable).

[45] OLD PART: All the material in this chapter is obsolete and will be replaced by the new notation definition system presented in [**KMR:NoLMD08**]

The OMDoc format allows to generate specialized style sheets that are tailored to the presentation of (collections of) OMDoc documents. The mechanism will be discussed in , here we only concern ourselves with the OMDoc primitives for representing the necessary data. In the next section, we will address the specification of style information for OMDoc elements by `omstyle` elements, and then the question of the specification of notation for mathematical symbols in `presentation` elements. EOP:45

Chapter 44

Defining Notations

We propose to encode the presentational characteristics of mathematical objects declaratively in *notation definitions* [46], which are part of the representational infrastructure and consist of "prototypes" [47] (patterns that are matched against content representations) and "renderings" (that are used to construct the corresponding presentations). Note that since we have reified the notations, we can now devise a flexible management process for notations. For example, we can capture the notation preferences of authors, aggregators, and readers and adapt documents to these. We propose an elaborated mechanism to collect notations from various sources and specify notation preferences below. EdN:46 EdN:47

44.1 Syntax of Notation Definitions

We will now present an abstract version of the presentation starting from the observation that in content markup formalisms for mathematics formulae are represented as "formula trees". Concretely, we will concentrate on OpenMath objects, the conceptual data model of OpenMath representations, since it is sufficiently general, and work is currently under way to unify the semantics of MathML with the one of OpenMath. Furthermore, we observe that the target of the presentation process is also a tree expression: a layout tree made of layout primitives and glyphs, e. g., a presentation MathML or LaTeX expression. [48] EdN:48

To specify notation definitions, we use the one given by the abstract grammar from Figure 44.1. Here $|$, $[-]$, $-^*$, and $-^+$ denote alternative, bracketing, and non-empty and possibly empty repetition, respectively. The non-terminal symbol ω is used for patterns ϕ that do not contain jokers. Throughout this article, we will use the non-terminal symbols of the grammar as meta-variables for objects of the respective syntactic class.

Intuitions The intuitive meaning of a *notation definition* $ntn = \phi_1, \ldots, \phi_r \vdash (\lambda_1\colon \rho_1)^{p_1}, \ldots, (\lambda_s\colon \rho_s)^{p_s}$ is the following: If an object matches one of the patterns ϕ_i, it is rendered by one of the renderings ρ_i. Which rendering is chosen, depends on the active rendering context, which is matched against the context annotations λ_i; context annotations are usually lists of key-value pairs and

[46]EdNote: TODO: change "notation specification" to "notation definition" in the whole paper

[47]EdNote: This is the only occurrence of the term "prototype" in the theoretical part. This doesn't match our implementation well. I'd use the term "prototype" more often. ... we use two terms ("pattern" and "prototype"). Using two terms might confuse the reviewers/readers. –CL
FR: I think "prototype" is the wrong word anyway and should be replaced with, e.g., "pattern" everywhere. I think it's OK for the paper though.
CM: I agree with Florian (for the first submission)

[48]EdNote: @FR: We should not mention LaTeX if our implementations don't support it. –CL; but we can support in in theory, can't we? Isn't it rather a question of input data? –CM
See JOMDoc mailing list –CL FR: JOMDoc is supposed to support it; the theory does. Anyway I need an ASCII based language to give readable examples.

Notation declarations	ntn	$::=$	$\phi^+ \vdash [(\lambda\colon \rho)^p]^+$
Patterns	ϕ	$::=$	
Symbols			$\sigma(n,n,n)$
Variables		$\vert$	$\upsilon(n)$
Applications		$\vert$	$@(\phi[,\phi]^+)$
Binders		$\vert$	$\beta(\phi,\Upsilon,\phi)$
Attributions		$\vert$	$\alpha(\phi,\sigma(n,n,n)\mapsto\phi)$
Symbol/Variable/Object/List jokers		$\vert$	$\underline{s} \vert \underline{v} \vert \underline{o}[\phi] \vert \underline{o} \vert \underline{l}(\phi)$
Variable contexts	Υ	$::=$	ϕ^+
Match contexts	M	$::=$	$[q\mapsto X]^*$
Matches	X	$::=$	$\omega^* \vert S^* \vert (X)$
Empty match contexts	μ	$::=$	$[q\mapsto H]^*$
Holes	H	$::=$	$\_ \vert$ “” $\vert (H)$
Context annotation	λ	$::=$	C^*
Renderings	ρ	$::=$	
XML elements			$\langle S\rangle\rho^*\langle/\rangle$
XML attributes		$\vert$	$S=$ ”ρ^*”
Texts		$\vert$	S
Symbol or variable names		$\vert$	$\underline{q}$
Matched objects		$\vert$	$\underline{q}^p$
Matched lists		$\vert$	$\texttt{for}(\underline{q},I,\rho^*)\{\rho^*\}$
Precedences	p	$::=$	$-\infty \vert I \vert \infty$
Names	n,s,v,l,o	$::=$	C^+
Integers	I	$::=$	integer
Qualified joker names	q	$::=$	$l/q \vert s \vert v \vert o \vert l$
Strings	S	$::=$	C^*
Characters	C	$::=$	character except /

Table 44.1: The Grammar for Notation Definitions

their precise syntax is given in Section **??**. The integer values p_i give the output precedences of the renderings, which are used to dynamically determine the placement of brackets.

The *patterns* ϕ_i are formed from a formal grammar for a subset of OpenMath objects extended with named jokers. The jokers $\underline{o}[\phi]$ and $\underline{l}(\phi)$ correspond to \(φ\) (or \(.\) if ϕ is omitted) and \(φ\)$^+$ in Posix regular expression syntax ([**posix**]) – except that our patterns are matched against the list of children of an OpenMath object instead of against a list of characters. Here underlined variables denote names of jokers that can be referred to in the renderings ρ_i. We need two special jokers $\underline{s}$ and $\underline{v}$, which only match OpenMath symbols and variables, respectively. The *renderings* ρ_i are formed by a formal syntax for simplified XML extended with means to refer to the jokers used in the patterns. When referring to object jokers, input precedences are given that work together with the output precedences of renderings. [49] EdN:49

Match contexts are used to store the result of matching a pattern against an object. Due to list jokers, jokers may be nested; therefore, we use qualified joker names in the match contexts (which are transparent to the user). Empty match contexts are used to store the structure of a match context induced by a pattern: They contain holes that are filled by matching the pattern against an object.

[49] EdNote: Note, we realized that the JOMDoc implementation is not compatible with this specification of jokers. It only supports a generic “object” joker, as well as a list joker. –CL FR: Yes. In my opinion JOMDoc should be changed. Alternatively, if someone's willing to change the specification in this article, they're welcome. CM@FR: Please open a discussion on this in the trac, for the article, we should leave it as given.

Example We will use a multiple integral as an example that shows all aspects of our approach in action.

$$\int_{a_1}^{b_1} \dots \int_{a_n}^{b_n} \sin x_1 + x_2 \, dx_n \dots dx_1.$$

Let *int*, *iv*, *lam*, *plus*, and *sin* abbreviate symbols for integration, closed real intervals, lambda abstraction, addition, and sine. We intend *int*, *lam*, and *plus* to be flexary symbols, i. e., symbols that take an arbitrary finite number of arguments. Furthermore, we assume symbols *color* and *red* from a content dictionary for style attributions. We want to render into LaTeX the OpenMath object

$$\begin{array}{l} @\big(\,int, @(iv, a_1, b_1), \dots, @(iv, a_n, b_n), \\ \quad \beta\big(\,lam, \upsilon(x_1), \dots, \upsilon(x_n),\ \alpha(@(plus, @(sin, \upsilon(x_1)), \upsilon(x_2)), color \mapsto red)\big)\big) \end{array}$$

as `\int_{`a_1`}^{`b_1`}...\int_{`a_n`}^{`b_n`}\color{red}{\sin `x_1`+`x_2`}d`x_n`...d`x_1

We can do that with the following notations:

$$\begin{array}{l} @(int, \underline{\texttt{ranges}}(@(iv, \underline{\texttt{a}}, \underline{\texttt{b}})), \beta(lam, \underline{\texttt{vars}}(\underline{\texttt{x}}), \underline{\texttt{f}})) \\ \vdash\ ((format = latex) : \\ \quad \texttt{for}(\underline{\texttt{ranges}})\{\backslash\texttt{int\_}\{\ \underline{\texttt{a}}^{\infty}\ \}\texttt{\^{}}\{\ \underline{\texttt{b}}^{\infty}\ \}\}\ \underline{\texttt{f}}^{\infty}\ \texttt{for}(\underline{\texttt{vars}}, -1)\{\texttt{d}\ \underline{\texttt{x}}^{\infty}\})^{-\infty} \\ \alpha(\underline{\texttt{a}}, color \mapsto \underline{\texttt{col}})\ \vdash\ ((format = latex) :\ \{\backslash color\{\ \underline{\texttt{col}}\ \}\ \underline{\texttt{a}}^{\infty}\ \})^{-\infty} \\ @(plus, \underline{\texttt{args}}(\underline{\texttt{arg}}))\ \vdash\ ((format = latex) :\ \texttt{for}(\underline{\texttt{args}}, +)\{\underline{\texttt{arg}}\})^{10} \\ @(sin, \underline{\texttt{arg}})\ \vdash\ ((format = latex) :\ \backslash sin\ \underline{\texttt{arg}})^{0} \end{array}$$

The first notation matches the application of the symbol *int* to a list of ranges and a lambda abstraction binding a list of variables. The rendering iterates first over the ranges, rendering them as integral signs with bounds, then recurses into the function body $\underline{\texttt{f}}$, then iterates over the variables, rendering them in reverse order prefixed with *d*. The second notation is used when $\underline{\texttt{f}}$ recurses into the presentation of the function body $\alpha(@(plus, @(sin, \upsilon(x_1)), \upsilon(x_2)), color \mapsto red)$. It matches an attribution of *color*, which is rendered using the LaTeX color package. The third notation is used when $\underline{\texttt{a}}$ recurses into the attributed object $@(plus, @(sin, \upsilon(x_1)), \upsilon(x_2))$. It matches any application of *plus*, and the rendering iterates over all arguments, placing the separator $+$ in between. Finally, *sin* is rendered in a straightforward way. We omit the notation that renders variables by their name.

The output precedence $-\infty$ of *int* makes sure that the integral as a whole is never bracketed. And the input precedences ∞ makes sure that the arguments of *int* are never bracketed. Both are reasonable because the integral notation provides its own fencing symbols, namely $\int$ and *d*. The output precedences of *plus* and *sin* are 10 and 0, which means that *sin* binds stronger; therefore, the expression $\sin x$ is not bracketed either. However, an inexperienced user may wish to display these brackets. Therefore, our rendering does not completely suppress them. Rather, we annotate them with an "elision level", which is computed as the difference of the two precedences. [50] This information can then be used by active documents (see section **??**). EdN:50

Well-formed Notations A notation definition $\phi_1, \dots, \phi_r \vdash (\lambda_1\colon \rho_1)^{p_1}, \dots, (\lambda_s\colon \rho_s)^{p_s}$ is well-formed if all ϕ_i are well-formed patterns that induce the same empty match contexts, and all ρ_i are well-formed renderings with respect to that empty match context.

Every pattern ϕ generates an *empty match context* $\mu(\phi)$ as follows:

- For an object joker $\underline{o}[\phi]$ or $\underline{o}$ occurring in ϕ but not within a list joker, $\mu(\phi)$ contains $o \mapsto \_$.
- For a symbol or variable with name n occurring in ϕ but not within a list joker, $\mu(\phi)$ contains $n \mapsto$ "".

[50] EdNote: CL@FR: Actually it is not strictly the difference. FR: I made it the difference in section ??

- For a list joker $\underline{l}(\phi')$ occurring in ϕ, $\mu(\phi)$ contains
  - $l \mapsto (\_)$, and
  - $l/n \mapsto (H)$ for every $n \mapsto H$ in $\mu(\phi')$.

In an empty match context, a hole $\_$ is a placeholder for an object, "" for a string, $(\_)$ for a list of objects, $((\_))$ for a list of lists of objects, and so on. Thus, symbol, variable, or object joker in ϕ produce a single named hole, and every list joker and every joker within a list joker produces a named list of holes (H). For example, the empty match context induced by the pattern in the notation for *int* above is

$$\texttt{ranges} \mapsto (\_),\ \texttt{ranges/a} \mapsto (\_),\ \texttt{ranges/b} \mapsto (\_),\ \texttt{f} \mapsto \_,$$
$$\texttt{vars} \mapsto (\_),\ \texttt{vars/x} \mapsto (\text{``''})$$

A pattern ϕ is well-formed if it satisfies the following conditions:

- There are no duplicate names in $\mu(\phi)$.
- No jokers occur within object jokers.
- List jokers may not occur as direct children of binders or attributions.
- At most one list joker may occur as a child of the same application, and it may not be the first child.
- At most one list joker may occur in the same variable context.

These restrictions guarantee that matching an OpenMath object against a pattern is possible in at most one way. In particular, no backtracking is needed in the matching algorithm.

Assume an empty match context μ. We define well-formed renderings with respect to μ as follows:

- $\langle S\rangle\rho_1,\ldots,\rho_r\langle/\rangle$ is well-formed if all ρ_i are well-formed.
- $S = "\rho_1,\ldots,\rho_r"$ is well-formed if all ρ_i are well-formed and are of the form S' or $\underline{n}$. Furthermore, $S = "\rho_1,\ldots,\rho_r"$ may only occur as a child of an XML element rendering.
- S is well-formed.
- $\underline{n}$ is well-formed if $n \mapsto$ "" is in μ.
- $\underline{o}^p$ is well-formed if $o \mapsto \_$ is in μ.
- $\texttt{for}(\underline{l}, I, \mathit{sep})\{\mathit{body}\}$ is well-formed if $l \mapsto (\_)$ or $l \mapsto (\text{``''})$ is in μ, all renderings in *sep* are well-formed with respect to μ, and all renderings in *body* are well-formed with respect to μ^l. The step size I and the separator *sep* are optional, and default to 1 and the empty string, respectively, if omitted.

Here μ^l is the empty match context arising from μ if every $l/q \mapsto (H)$ is replaced with $q \mapsto H$ and every previously existing hole named q is removed. Replacing $l/q \mapsto (H)$ means that jokers occurring within the list joker l are only accessible within a corresponding rendering $\texttt{for}(\underline{l}, I, \rho^*)\{\rho^*\}$. And removing the previously existing holes means that in $@(\underline{o}, \underline{l}(\underline{o}))$, the inner object joker shadows the outer one.

44.2 Semantics of Notation Definitions

The rendering algorithm has two levels. The high-level takes as input a document *Doc*, a notation database *DB*, and a rendering context Λ. [51] The intuition of *DB* is that it is essentially a set of notation definitions. In practice this set of notation definitions will be large and highly structured; therefore, we assume a database maintaining it. The rendering context Λ is a list of context annotations that the database uses to select notations, e. g., the requested output format and language. The algorithm outputs *Doc* with OpenMath objects in it replaced with their rendering. It does so in three steps: EdN:51

1. In a preprocessing step, *Doc* is scanned and notation definitions given inside *Doc* are collected. If *Doc* imports or includes other documents, these are retrieved and processed recursively. And if *Doc* references external notation definitions, these are retrieved as well. Together with the notations already present in the database, these notations form the notation context Π.
2. In a second preprocessing step, Π is normalized by grouping together renderings of the same patterns. Then for each pattern, the triples $(\lambda\colon\rho)^p$ pertaining to it are filtered and ordered according to how well λ matches Λ. This process can also scan for and store arbitrary other information in *Doc* or in *DB*: For example, *Doc* can contain what we call notation tags (see Section **??**) that explicitly relate an OpenMath object and a rendering.
3. *Doc* is traversed, and the low-level algorithm is invoked on every OpenMath object found.

The preprocessing steps are described in detail in Sec. **??**. In the following, we describe the low-level algorithm.

The low-level algorithm takes as input an OpenMath object ω and the notation context Π. It returns the rendering of ω. If the low-level algorithm is invoked recursively to render a subobject of ω, it takes an input precedence p, which is used for bracket placement, as an additional argument.

It computes its output in two steps.

1. ω is matched against the patterns in the notation definitions in Π until a matching pattern ϕ is found. [52] The notation definition in which ϕ occurs induces a list $(\lambda_1\colon\rho_1)^{p_1},\ldots,(\lambda_n\colon\rho_n)^{p_n}$ of context-annotations, renderings, and output precedences. The first one of them is chosen unless Π contains a notation tag that selects a different one for ω. EdN:52
2. The output is ${\rho_j}^{M(\phi,\omega)}$, the rendering of ρ_j in context $M(\phi,\omega)$ as defined below. Additionally, if $p_j > p$, the output is enclosed in brackets.

Semantics of Patterns The semantics of patterns is that they are matched against OpenMath objects. Naturally, every OpenMath object matches against itself. Symbol, variable, and object jokers match in the obvious way[53]. A list joker $\underline{l}(\phi)$ matches against a non-empty list of objects all matching ϕ. EdN:53

Let ϕ be a pattern and ω a matching OpenMath object. We define a match context $M(\phi,\omega)$ as follows.

- For a symbol or variable joker with name n that matched against the sub-object ω' of ω, $M(\phi,\omega)$ contains $n \mapsto S$ where S is the name of ω'.
- For an object joker $\underline{o}[\phi']$, $M(\phi,\omega)$ contains $o \mapsto \phi'$.

[51]EdNote: CL@FR: Is there any intuition behind calling these Λ and Π? FR: Upper case lambda is matched against the lower case lambdas. I think the pi is from Christine.
[52]EdNote: FR@all: What happens if ω matches two patterns?
CL: Shouldn't be a problem. Then we just take the renderings associated with those two patterns and prioritize among them.
[53]EdNote: See above: Our implementation only knows object jokers. –CL; @FR: See above, please create a ticket in the trac.

- For an object joker $\underline{o}$ that matched against the sub-object ω' of ω, $M(\phi, \omega)$ contains $o \mapsto \omega$.
- If a list joker $\underline{l}(\phi')$ matched a list $\omega_1, \ldots, \omega_r$, then $M(\phi, \omega)$ contains
  - $l \mapsto (\omega_1, \ldots, \omega_r)$, and
  - for every l/q in $\mu(\phi)$: $l/q \mapsto (X_1, \ldots, X_r)$ where $q \mapsto X_i$ in $M(\phi', \omega_i)$.

We omit the precise definition of what it means for a pattern to match against an object. It is, in principle, well-known from regular expressions. Since no backtracking is needed, the computation of $M(\phi, \omega)$ is straightforward. We denote by $M(q)$, the lookup of the match bound to q in a match context M.

Semantics of Renderings If ϕ matches against ω and the rendering ρ is well formed with respect to $\mu(\phi)$, the intuition of $\rho^{M(\phi,\omega)}$ is that the joker references in ρ are replaced according to $M(\phi, \omega) =: M$. Formally, ρ^M is defined as follows.

The rendering is either a string or a sequence of XML elements. We will use $O + O'$ to denote the concatenation of two outputs O and O'. By that, we mean a concatenation of sequences of XML elements or of strings if O and O' have the same type (string or XML). Otherwise, $O + O'$ is a sequence of XML elements treating a string as an XML text node. This operation is associative since consecutive text nodes can always be merged.

- $\langle S\rangle \rho_1 \ldots \rho_r \langle/\rangle$ is rendered as an XML element with name S. The attributes are those ${\rho_i}^M$ that are rendered as attributes. The children are the concatenation of the remaining ${\rho_i}^M$ preserving their order.
- $S = "\rho_1 \ldots \rho_r"$ is rendered as an attribute with label S and value ${\rho_1}^M + \ldots + {\rho_n}^M$ (which has type text due to the well-formedness).
- S is rendered as the text S.
- $\underline{s}$ and $\underline{v}$ are rendered as the text $M(s)$ or $M(v)$, respectively.
- $\underline{o}^p$ is rendered by applying the rendering algorithm recursively to $M(o)$ and p.
- $\texttt{for}(\underline{l}, I, \rho_1 \ldots \rho_r)\{\rho'_1 \ldots \rho'_s\}$ is rendered by the following algorithm:
  1. Let $sep := {\rho_1}^M + \ldots + {\rho_r}^M$ and t be the length of $M(l)$.
  2. For $i = 1, \ldots, t$, let $R_i := {\rho'_1}^{M^l_i} + \ldots + {\rho'_s}^{M^l_i}$.
  3. If $I = 0$, return nothing and stop. If I is negative, reverse the list R, and invert the sign of I.
  4. Return $R_I + sep + R_{2*I} \ldots + sep + R_T$ where T is the greatest multiple of I smaller than or equal to t.

Here the match context M^l_i arises from M as follows

- replace $l \mapsto (X_1 \ldots X_t)$ with $l \mapsto X_i$,
- for every $l/q \mapsto (X_1 \ldots X_t)$ in M: replace it with $q \mapsto X_i$, and remove a possible previously defined match for q.

Example Consider the example introduced in Section **??**. There we have

$$\begin{aligned}\omega = {} & @\big(\mathit{int}, @(\mathit{iv}, a_1, b_1), \ldots, @(\mathit{iv}, a_n, b_n),\\ & \beta\big(\mathit{lam}, \upsilon(x_1), \ldots, \upsilon(x_n),\ \alpha(@(\mathit{plus}, @(\mathit{sin}, \upsilon(x_1)), \upsilon(x_2)), \mathit{color} \mapsto \mathit{red})\big)\big)\end{aligned}$$

And Π is the given list of notation definitions. Let $\Lambda = (format = latex)$. Matching ω against the patterns in Π succeeds for the first notation definitions and yields the following match context M:

$$\texttt{ranges} \mapsto (@(\mathit{iv}, a_1, b_1), \ldots, @(\mathit{iv}, a_n, b_n)),\ \texttt{ranges/a} \mapsto (a_1, \ldots, a_n),$$

$$\texttt{ranges/b} \mapsto (b_1, \ldots, b_n),\ \texttt{f} \mapsto \alpha(@(\mathit{plus}, @(\mathit{sin}, \upsilon(x_1)), \upsilon(x_2)), \mathit{color} \mapsto \mathit{red}),$$

$$\texttt{vars} \mapsto (\upsilon(x_1), \ldots, \upsilon(x_n)),\ \texttt{vars/x} \mapsto (x_1, \ldots, x_n)$$

In the second step, a specific rendering is chosen. In our case, there is only one rendering, which matches the required rendering context Λ, namely

$$\rho = \texttt{for}(\underline{\texttt{ranges}})\{\texttt{\textbackslash int\_\{ }\underline{\texttt{a}}^{\infty}\texttt{ \}\^{}\{ }\underline{\texttt{b}}^{\infty}\texttt{ \}\}}\ \ \underline{\texttt{f}}^{\infty}\ \ \texttt{for}(\underline{\texttt{vars}}, -1)\{\texttt{d }\underline{\texttt{x}}^{\infty}\})^{-\infty}$$

To render ρ in match context M, we have to render the three components and concatenate the results. Only the iterations are interesting. In both iterations, the separator *sep* is empty; in the second case, the step size I is -1 to render the variables in reverse order.

[=ext]

Part XIV

Auxiliary Elements (Module EXT)

Up to now, we have been mainly concerned with providing elements for marking up the inherent structure of mathematical knowledge in mathematical statements and theories. Now, we interface OMDoc documents with the Internet in general and mathematical software systems in particular. We can thereby generate presentations from OMDoc documents where formulae, statements or even theories that are active components that can directly be manipulated by the user or mathematical software systems. We call these documents **active document** s. For this we have to solve two problems: an abstract interface for calls to external (web) services[1] and a way of storing application-specific data in OMDoc documents (e.g. as arguments to the system calls).

The module EXT provides a basic infrastructure for these tasks in OMDoc. The main purpose of this module is to serve as an initial point of entry. We envision that over time, more sophisticated replacements will be developed driven by applications.

Element	Attributes		M	Content
	Req.	Optional	D	
private		xml:id, for, theory, requires, type, reformulates, class, style	+	data+
code		xml:id, for, theory, requires, type, class, style	+	input?, output?, effect?, data+
input		xml:id, style, class	+	h:p*
output		xml:id, style, class	+	h:p*
effect		xml:id, style, class	+	h:p*
data		format, href, size, original, pto, pto-version	−	<![CDATA[...]]>

Figure 44.1: The OMDoc Auxiliary Elements for Non-XML Data

[1]Compare [**Kohlhase:OMDoc1.6primer**] in the OMDoc Primer.

Chapter 45

Non-XML Data and Program Code in OMDoc

The representational infrastructure for mathematical knowledge provided by OMDoc is sufficient as an output- and library format for mathematical software systems like computer algebra systems, theorem provers, or theory development systems. In particular, having a standardized output- and library format like OMDoc will enhance system interoperability, and allows to build and deploy general storage and library management systems (see [**Kohlhase:OMDoc1.6projects**] for an OMDoc example). In fact this was one of the original motivations for developing the format.

However, most mathematical software systems need to store and communicate system-specific data that cannot be standardized in a general knowledge-representation format like OMDoc. Examples of this are pieces of program code, like tactics or proof search heuristics of tactical theorem provers or linguistic data of proof presentation systems. Only if these data can be integrated into OMDoc, it will become a full storage and communication format for mathematical software systems. One characteristic of such system-specific data is that it is often not in XML syntax, or its format is not fixed enough to warrant for a general XML encoding.

Definition 45.0.1 For this kind of data, OMDoc provides the **private** and **code** elements. As the name suggests, the latter is intended for program code[1] and the former for system-specific data that is not program code.

`private`

`code`

The attributes of these elements are almost identical and contain metadata information identifying system requirements and relations to other OMDoc elements. We will first describe the shared attributes and then describe the elements themselves.

`xml:id` for identification.

`theory` specifies the mathematical theory (see Chapter 15) that the data is associated with.

`for` allows to attach data to some other OMDoc element. Attaching `private` elements to OMDoc elements is the main mechanism for system-specific extension of OMDoc.

`requires` specifies other data this element depends upon as a whitespace-separated list of URI references. This allows to factor private data into smaller parts, allowing more flexible data storage and retrieval which is useful for program code or private data that relies on program code. Such data can be broken up into procedures and the call-hierarchy can be encoded in `requires` attributes. With this information, a storage application based on OMDoc can always communicate a minimal complete code set to the requesting application.

[1]There is a more elaborate proposal for treating program code in the OMDoc arena at [**Kohlhase:codemlspec**], which may be integrated into OMDoc as a separate module in the future, for the moment we stick to the basic approach.

`reformulates` (`private` only) specifies a set of OMDoc elements whose knowledge content is reformulated by the `private` element as a whitespace-separated list of URI references. For instance, the knowledge in the assertion in Listing 45.1 can be used as an algebraic simplification rule in the ANALYTICA theorem prover [**ClaKoh:sda03**] based on the MATHEMATICA computer algebra system.

The `private` and `code` elements contain an optional `metadata` element and a set of `data` elements that contain or reference the actual data.

Listing 45.1: Reformulating Mathematical Knowledge

```
  <assertion xml:id="ALGX0">
    <h:p>If a, b, c, d are numbers, then we have a + b(c + d) = a + bc + bd.</h:p>
  </assertion>
  <private xml:id="alg−expr−1" pto="Analytica" reformulates="ALGX0">
   <data format="mathematica−5.0">
     <![CDATA[SIMPLIFYRULES[a_ + b_*(c_ + d_) :> a + b*c + b*d /; NumberQ[b]]]]>
   </data>
  </private>
```

`data`

Definition 45.0.2 The **data** element contains the data in a `CDATA` section. Its `pto` attribute contains a whitespace-separated list of URI references which specifies the set of systems to which the data are related. The intention of this field is that the data is visible to all systems, but should only manipulated by a system that is mentioned here. The `pto-version` attribute contains a whitespace-separated list of version number strings; this only makes sense, if the value of the corresponding `pto` is a singleton. Specifying this may be necessary, if the data or even their format change with versions.

If the content of the `data` element is too large to store directly in the OMDoc or changes often, then the `data` element can be augmented by a link, specified by a URI reference in the `href` attribute. If the `data` element is non-empty and there is a `href`[2], then the optional attribute `original` specifies whether the `data` content (value `local`) or the external resource (value `external`) is the original. The optional `size` attribute can be used to specify the content size (if known) or the resource identified in the `href` attribute. The `data` element has the (optional) attribute `format` to specify the format the data are in, e.g. `image/jpeg` or `image/gif` for image data, `text/plain` for text data, `binary` for system-specific binary data, etc. It is good practice to use the MIME types [**FreBor:MIME96**] for this purpose whenever applicable. Note that in a `private` or `code` element, the `data` elements must differ in their `format` attribute. Their order carries no meaning.

In Listing 45.2 we use a `private` element to specify data for an image[3] in various formats, which is useful in a content markup format like OMDoc as the transformation process can then choose the most suitable one for the target.

Listing 45.2: A `private` Element for an Image

```
<private xml:id="legacy">
  <metadata>
    <dc:title>A fragment of Bourbaki's Algebra</dc:title>
    <dc:creator role="trl">Michael Kohlhase</dc:creator>
    <dc:date action="created">2002−01−03T0703</dc:date>
    <dc:description>A fragment of Bourbaki's Algebra</dc:description>
    <dc:source>Nicolas Bourbaki, Algebra, Springer Verlag 1974</dc:source>
    <dc:type>Text</dc:type>
  </metadata>
  <data format="application/x−latex" href="legacy.tex"/>
  <data format="image/jpg" href="legacy.jpeg"/>
  <data format="application/postscript" href="legacy.ps"/>
  <data format="application/pdf" href="legacy.pdf"/>
</private>
```

[2]e.g. if the `data` content serves as a cache for the data at the URI, or the `data` content fixes a snapshot of the resource at the URI

[3]actually Figure **??** from [**Kohlhase:OMDoc1.6primer**]

Definition 45.0.3 The `code` element is used for embedding pieces of program code into an OMDoc document. It contains the documentation elements **input** , **output** , and **effect** that specify the behavior of the procedure defined by the code fragment. The `input` element describes the structure and scope of the input arguments, `output` the outputs produced by calling this code on these elements, and `effect` any side effects the procedure may have. They contain a mathematical vernacular marked up in a sequence of `h:p` elements[54]. If any of these elements are missing it means that we may not make any assumptions about them, not that there are no inputs, outputs or effects.[55]

input

output

effect

EdN:54

EdN:55

For instance, to specify that a procedure has no side-effects we need to specify something like

```
<effect><h:p>None.</h:p></effect>
```

These documentation elements are followed by a set of `data` elements that contain or reference the program code itself. Listing 46.2 shows an example of a `code` element used to store Java code for an applet.

Listing 45.3: The Program Code for a Java Applet

```
<code xml:id="callMint" requires="org.riaca.cas">
  <metadata>
    <dc:description>
      The multiple integrator applet. It puts up a user interface, queries the user for a
      function, which it then integrates by calling one of several computer algebra systems.
    </dc:description>
  </metadata>
  <data format="application/x−java−applet">
    <![CDATA[... 《the callMint code goes here》 ...]]>
  </data>
  <input><h:p>None: the applet handles input itself.</h:p></input>
  <output><h:p>The result of the integration.</h:p></output>
  <effect><h:p>None.</h:p></effect>
</code>
```

BOP:56

[54]EdNote: OMDoc1.2 had the possibility to express FMPs in there. The latter may be used for program verification purposes. Do we want to enable this again?
[55]EdNote: Maybe
[56]Old Part: MK: this to be rethought and possibly integrated with the `h:object` element, see Ticket 1547 in the OMDoc TRAC for a discussion

Chapter 46

Applets and External Objects in OMDoc

Web-based text markup formats like HTML have the concept of an external object or "applet", i.e. a program that can in some way be executed in the browser or web client during document manipulation. This is one of the primary format-independent ways used to enliven parts of the document. Other ways are to change the document object model via an embedded programming language (e.g. JavaScript). As this method (dynamic HTML) is format-dependent[1], it seems difficult to support in a content markup format like OMDoc.

The challenge here is to come up with a format-independent representation of the applet functionality, so that the OMDoc representation can be transformed into the specific form needed by the respective presentation format. Most user agents for these presentation formats have built-in mechanisms for processing common data types such as text and various image types. In some instances the user agent may pass the processing to an external application ("plug-ins"). These need information about the location of the object data, the MIME type associated with the object data, and additional values required for the appropriate processing of the object data by the object handler at run-time.

Element	Attributes		M	Content
	Req.	Optional	D	
omlet	data,	xml:id, action, show, actuate, class, style	+	h:p* & param* & data*
param	name	value, valuetype	-	EMPTY

Figure 46.1: The OMDoc Elements for External Objects

Definition 46.0.1 In OMDoc, we use the **omlet** element for applets. It generalizes the HTML applet concept in two ways: The computational engine is not restricted to plug-ins of the browser (we do not know what the result format and presentation engine will be) and the program code can be included in the OMDoc document, making document-centered computation easier to manage. omlet

Like the `xhtml:object` tag, the `omlet` element can be used to wrap any text. In the OMDoc context, this means that the children of the `omlet` element can be any elements or text that can occur in the `h:p` element together with `param` elements to specify the arguments. The main presentation intuition is that the applet reserves a rectangular space of a given pre-defined size (specified in the CSS markup in the `style` attribute; see Listing 46.2) in the result document presentation, and hands off the presentation and interaction with the document in this space to the applet process. The data for the external object is referenced in two possible ways. Either via

[1]In particular, the JavaScript references the HTML DOM, which in our model is created by a presentation engine on the fly.

the data attribute, which contains a URI reference that points to an OMDoc code or private element that is accessible (e.g. in the same OMDoc) or by embedding the respective code or private elements as children at the end of the omlet element. This indirection allows us to reuse the machinery for storing code in OMDocs. For a simple example see Listing 46.2.

The behavior of the external object is specified in the attributes action, show and actuate attributes[2].

The action specified the intended action to be performed with the data. For most objects, this is clear from the MIME type. Images are to be displayed, audio formats will be played, and application-specific formats are passed on to the appropriate plug-in. However, for the latter (and in particular for program code), we might actually be interested to display the data in its raw (or suitably presented) form. The action addresses this need, it has the possible values execute (pass the data to the appropriate plug-in or execute the program code), display (display it to the user in audio- or visual form), and other (the action is left unspecified).

The show attribute is used to communicate the desired presentation of the ending resource on traversal from the starting resource. It has one of the values new (display the object in a new document), replace (replace the current document with the presentation of the external object), embed (replace the omlet element with the presentation of the external object in the current document), and other (the presentation is left unspecified).

The actuate attribute is used to communicate the desired timing of the action specified in the action attribute. Recall that OMDoc documents as content representations are not intended for direct viewing by the user, but appropriate presentation formats are derived from it by a "presentation process" (which may or may not be incorporated into the user agent). Therefore the actuate attribute can take the values onPresent (when the presentation document is generated), onLoad (when the user loads the presentation document), onRequest (when the user requests it, e.g. by clicking in the presentation document), and other (the timing is left unspecified).

The simplest form of an omlet is just the embedding of an external object like an image as in Listing 46.1, where the data attribute points to the private element in Listing 45.2. For presentation, e.g. as XHTML in a modern browser, this would be transformed into an xhtml:object element [**W3C:xhtml2000**], whose specific attributes are determined by the information in the omlet element here and those data children of the private element specified in the data attribute of the omlet that are chosen for presentation in XHTML. If the action specified in the action attribute is impossible (e.g. if the contents of the data target cannot be presented), then the content of the omlet element is processed as a fallback.

Listing 46.1: An omlet for an Image

```
<omlet data="#legacy" show="embed">A Fragment of Bourbaki's Algebra</omlet>
```

In Listing 46.2 we present an example of a conventional Java applet in a mathematical text: the data attribute points to a code element, which will be executed (if the value of the action attribute were display, the code would be displayed).

Listing 46.2: An omlet that Calls the Java Applet from Listing 45.3.

```
<omtext xml:id="monp_1">
 <h:p>Let practice integration!</h:p>
  <h:p><omlet data="#callMint" action="execute" style="width:320;height:200">
        No plug−in found for callMint!
     </omlet></h:p>
</omtext>
```

In this example, the Java applet did not need any parameters (compare the documentation in the input element in Listing 45.3).

In the applet in Listing 46.3 we assume a code fragment or plug-in (in a code element whose xml:id attribute has the value sendtoTP, which we have not shown) that processes a set of named arguments (parameter passing with keywords) and calls the theorem prover, e.g. via a web-service as described in [**Kohlhase:OMDoc1.6primer**].

[2]These latter two attributes are modeled after the XLink [**DeRMal:xlink01**] attributes show and actuate.

Listing 46.3: An `omlet` for Connecting to a Theorem Prover

```
<h:p> Let us prove it interactively:
  <omlet data="#sendtoTP" action="display">
    <param name="timeout" value="30" valuetype="data"/>
    <param name="performative" value="prove"/>
    <param name="problem" value="#ALGX0" valuetype="object"/>
    <param name="description" value="http://example.org/prob17.html" valuetype="ref"/>
    <param name="instance">
     <om:OMA><OMS name="root" cd="arith1"/>
         <om:OMI>3</om:OMI><om:OMI>3</om:OMI>
       </om:OMA>
   </param>
    Sorry, no theorem prover available!
  </omlet>
</h:p>
```

Definition 46.0.2 For parameter passing, we use the **param** elements which specify a set of values that may be required to process the object data by a plug-in at run-time. Any number of `param` elements may appear in the content of an `omlet` element. Their order does not carry any meaning. The `param` element carries the attributes

`param`

`name` This required attribute defines the name of a run-time parameter, assumed to be known by the plug-in. Any two `param` children of an `omlet` element must have different `name` values.

`value` This attribute specifies the value of a run-time parameter passed to the plug-in for the key `name`. Property values have no meaning to OMDoc; their meaning is determined by the plug-in in question.

`valuetype` This attribute specifies the type of the `value` attribute. The value `data` (the default) means that the value of the `value` will be passed to the plug-in as a string. The value `ref` specifies that the value of the `value` attribute is to be interpreted as a URI reference that designates a resource where run-time values are stored. Finally, the value `object` specifies that the `value` value points to a `private` or `code` element that contains a multi-format collection of `data` elements that carry the data.

If the `param` element does not have a `value` attribute, then it may contain a list of mathematical objects encoded as OpenMath, content MathML, or `legacy` elements.

EOP:56

[=quiz]

Part XV

Exercises (Module QUIZ)

Exercises and study problems are vital parts of mathematical documents like textbooks or exams, in particular, mathematical exercises contain mathematical vernacular and pose the same requirements on context like mathematical statements. Therefore markup for exercises has to be tightly integrated into the document format, so OMDoc provides a module for them.

Note that the functionality provided in this module is very limited, and largely serves as a placeholder for more pedagogically informed developments in the future (see [**Kohlhase:OMDoc1.6projects**] and [**GogMelUllCai:psmmee03**] for an example in the OMDoc framework).

Element	Attributes		M	Content
	Req.	Optional	D	
`exercise`		`xml:id, class, style`	+	`h:p*,hint?,(solution*|mc*)`
`hint`		`xml:id, class, style`	+	`h:p*`
`solution`		`xml:id, for, class, style`	+	⟪*top-level element*⟫
`mc`		`xml:id, for, class, style`	−	`choice, hint?, answer`
`choice`		`xml:id, class, style`	+	`h:p*`
`answer`	`verdict`	`xml:id, class, style`	+	`h:p*`

Figure 46.2: The OMDoc Auxiliary Elements for Exercises

Definition 46.0.3 The QUIZ module provides the top-level elements **exercise** , `hint`, and `solution`. The first one is used for exercises and assessments. The question statement is represented as mathematical vernacular in a sequence of `h:p` elements. This information can be augmented by hints (using the `hint` element) and a solution/assessment block (using the `solution` and `mc` elements). `exercise`

The `hint` and `solution` elements can occur as children of `exercise`; or outside, referencing it in their optional `for` attribute. This allows a flexible positioning of the hints and solutions, e.g. in separate documents that can be distributed separately from the `exercise` elements.

Definition 46.0.4 The **hint** element contains mathematical vernacular as a sequence of `h:p` elements for the hint text. The **solution** element can contain any number of OMDoc top-level elements to explain and justify the solution. This is the case, where the question contains an assertion whose proof is not displayed and left to the reader. Here, the `solution` contains a proof. `hint` `solution`

Listing 46.4: An Exercise from the TEXBook

```
<exercise xml:id="TeXBook−18−22">
  <h:p>Sometimes the condition that defines a set is given as a fairly long
    English description; for example consider ‘{p|p and p+2 are prime}’. An
    hbox would do the job:</h:p>

  <h:p style="display:block;font−family:fixed">
    $\{\,p\mid\hbox{$p$ and $p+2$ are prime}\,\}$
  </h:p>

  <h:p>but a long formula like this is troublesome in a paragraph, since an hbox cannot
    be broken between lines, and since the glue inside the
    <h:span style="font−family:fixed">\hbox</h:span> does not vary with the inter−word
    glue in the line that contains it. Explain how the given formula could be
    typeset with line breaks.</p>
  </h:p>
  <hint>
    <h:p>Go back and forth between math mode and horizontal mode.</h:p>
  </hint>
  <solution>
    <h:p>
      <h:span style="font−family:fixed">
       $\{\,p\mid p$~and $p+2$ are prime$\,\}$
      </h:span>,
      assuming that <h:span style="font−family:fixed">\mathsurround</h:span> is
      zero. The more difficult alternative ’<h:span style="font−family:fixed">
      $\{\,p\mid p\\ {\rm and}\ p+2\rm\ are\ prime\,\}$</h:span>’
      is not a solution, because line breaks do not occur at
      <h:span style="font−family:fixed">\␣</h:span> (or at glue of any
      kin) within math formulas. Of course it may be best to display a formula like
      this, instead of breaking it between lines.
    </h:p>
  </solution>
</exercise>
```

Multiple-choice exercises (see Listing 46.5) are represented by a group of `mc` elements inside an `exercise` element.

`mc`

Definition 46.0.5 An **mc** element represents a single choice in a multiple choice element. It contains the elements below (in this order).

`choice` for the description of the choice (the text the user gets to see and is asked to make a decision on). The **choice** element carries the `xml:id`, `style`, and `class` attributes and contains mathematical vernacular in a sequence of `h:p` elements.

`hint` (optional) for a hint to the user, see above for a description.

`answer` for the feedback to the user. This can be the correct answer, or some other feedback (e.g. another hint, without revealing the correct answer). The `verdict` attribute specifies the truth of the answer, it can have the values `true` or `false`. This element is required, inside a `mc`, since the `verdict` is needed. It can be empty if no feedback is available. Furthermore, the **answer** element carries the `xml:id`, `style`, and `class` attributes and contains mathematical vernacular as a sequence of `h:p` elements.

Listing 46.5: A Multiple-Choice Exercise in OMDoc

```
<exercise for="#ida.c6s1p4.l1" xml:id="ida.c6s1p4.mc1">
  <h:p>
    What is the unit element of the semi−group Q with operation a ∗ b = 3ab?
  </h:p>
  <mc>
    <choice><h:p><OMI>1</OMI></h:p></choice>
    <answer verdict="false"><h:p>No, 1 ∗ 1 = 3 and not 1</h:p></answer>
  </mc>
  <mc>
    <choice><h:p>1/3</h:p></choice>
    <answer verdict="true"></answer>
  </mc>
  <mc>
    <choice><h:p>It has no unit.</h:p></choice>
    <answer verdict="false"><h:p>No, try another answer</h:p></answer>
  </mc>
</exercise>
```

[=document-model]

Part XVI

Document Models for OMDoc

In almost all XML applications, there is a tension between the document view and the object view of data; after all, XML is a document-oriented interoperability framework for exchanging data objects. The question, which view is the correct one for XML in general is hotly debated among XML theorists. In OMDoc, actually both views make sense in various ways. Mathematical documents are the objects we try to formalize, they contain knowledge about mathematical objects that are encoded as formulae, and we arrive at content markup for mathematical documents by treating knowledge fragments (statements and theories) as objects in their own right that can be inspected and reasoned about.

In Part I to Part XIV, we have defined what OMDoc documents look like and motivated this by the mathematical objects they encode. But we have not really defined the properties of these documents as objects themselves (we will speak of the OMDoc **document object model** (OM-DOM)). To get a feeling for the issues involved, let us take stock of what we mean by the object view of data. In mathematics, when we define a class of mathematical objects (e.g. vector spaces), we have to say which objects belong to this class, and when they are to be considered equal (e.g. vector spaces are equal, iff they are isomorphic). When defining the intended behavior of operations, we need to care only about objects of this class, and we can only make use of properties that are invariant under object equality. In particular, we cannot use properties of a particular realization of a vector space that are not preserved under isomorphism. For document models, we do the same, only that the objects are documents.

Chapter 47

XML Document Models

[=XMLDom]

XML supports the task of defining a particular class of documents (e.g. the class of OMDoc documents) with formal grammars such as the document type definition (DTD) or an XML schema, that can be used for mechanical document validation. Surprisingly, XML leaves the task of specifying document equality to be clarified in the (informal) specifications, such as this OMDoc specification. As a consequence, current practice for XML applications is quite varied. For instance, the OpenMath standard (see [**BusCapCar:2oms04**] and Chapter 4) gives a mathematical object model for OpenMath objects that is specified independently of the XML encoding. Other XML applications like e.g. presentation MathML [**CarIon:MathML03**] or XHTML [**W3C:xhtml2000**] specify models in form of the intended screen presentation, while still others like the XSLT [**Clark:xslt99**] give the operational semantics.

For a formal definition let $\mathcal{K}$ be a set of documents. We take a **document model** to be a partial equivalence relation[1]. In particular, a relation $\mathcal{X}$ is an equivalence relation on $\mathcal{K}$. For a given document model $\mathcal{X}$, let us say that two documents d and d' are **$\mathcal{X}$-equal**, iff $d\mathcal{X}d'$. We call a property p **$\mathcal{X}$-invariant** , iff for all $d\mathcal{X}d'$, p holds on d whenever p holds on d'.

A possible source of confusion is that documents can admit more than one document model (see [**KohKoh:esmk05**] for an exploration of possible document models for mathematics). Concretely, OMDoc documents admit the OMDoc document model that we will specify in section Chapter 47 and also the following four XML document models that can be restricted to OMDoc documents (as a relation).[2]

The binary document model interprets files as sequences of bytes. Two documents are equal, iff they are equal as byte sequence. This is the most concrete and fine-grained (and thus weakest) document model imaginable.

The lexical document model interprets binary files as sequences of Unicode characters [**Unicode:tuc03**] using an encoding table. Two files may be considered equal by this document model even though they differ as binary files, if they have different encodings that map the byte sequences to the same sequence of Unicode characters.

The XML syntax document model interprets Unicode Files as sequences consisting of an XML declaration, a DOCTYPE declaration, tags, entity references, character references, CDATA sections, PCDATA comments, and processing instructions. At this level, for instance, whitespace characters between XML tags are irrelevant, and XML documents may be considered the same, if they are different as Unicode sequences.

[1]A partial equivalence relation is a symmetric transitive relation. We will use $[d]_{\mathcal{X}}$ for the **equivalence class** of d, i.e. $[d]_{\mathcal{X}} := \{e|d\mathcal{X}e\}$ $\mathcal{X}$ on documents, such that $\{d|d\mathcal{X}d\} = \mathcal{K}$

[2]Here we follow Eliotte Rusty Harold's classification of layers of XML processing in [**Harold:ex03**], where he distinguishes the binary, lexical, sequence, structure, and semantic layer, the latter being the document model of the XML application

The XML structure document model interprets documents as XML trees of elements, attributes, text nodes, processing instructions, and sometimes comments. In this document model the order of attribute declarations in XML elements is immaterial, double and single quotes can be used interchangeably for strings, and XML comments (`<!--`...`-->`) are ignored.

Each of these document models, is suitable for different applications, for instance the lexical document model is the appropriate one for Unicode-aware editors that interpret the encoding string in the XML declaration and present the appropriate glyphs to the user, while the binary document model would be appropriate for a simple ASCII editor. Since the last three document models are refinements of the XML document model, we will recap this in the next section and define the OMDoc document model in Chapter 47.

To get a feeling for the issues involved, let us compare the OMDoc elements in Listings 47.1 to 48.1 below. For instance, the serialization in Listing 47.2 is XML-equal to the one in Listing 47.1, but not to the one in Listing 48.1.

Listing 47.1: An OMDoc Definition

```
<docalt>
  <definition xml:id="comm.def.en" for="comm">
    <h:p xml:lang="en">
      An operation <OMV name="op" id="op"/> is called commutative, iff
      <OMA id="comm1"><OMS cd="relation1" name="eq"/>
        <OMA><OMV name="op"/><OMV name="X"/><OMV name="Y"/></OMA>
        <OMA><OMV name="op"/><OMV name="Y"/><OMV name="X"/></OMA>
      </OMA>
      for all <OMV id="x" name="X"/> and <OMV id="y" name="Y"/>.
    </h:p>
  </definition
  <definition xml:id="comm.def.de" for="#comm">
    <h:p xml:lang="de">
      Eine Operation <OMR href="#op"/> heißt kommutativ, falls
      <OMR href="#comm1"/> für alle <OMR href="#x"/> und <OMR href="#y"/>.
    </h:p>
  </definition>
</docalt>
```

Listing 47.2: An XML-equal serialization for Listing 47.1

```
<definition for="comm" xml:id="comm.def.de" >
 <h:p xml:lang='de'> <!-- Note the unabbreviated empty element -->
   Eine Operation <OMR href="#op"/> heißt
   kommutativ, falls <OMR href='comm1'/> für alle
   <OMR href="#x"/> und <OMR href='y'/>.
  </h:p>
</definition>
```

Chapter 48

The OMDoc Document Model

[=omdom]

The OMDoc document model extends the XML structure document model in various ways. We will specify the equality relation in the table below, and discuss a few general issues here.

The OMDoc document model is guided by the notion of content markup for mathematical documents. Thus, two document fragments will only be considered equal, if they have the same abstract structure. For instance, the order of children of an `docalt` element is irrelevant, since they form a multilingual group which form the base for multilingual text assembly. Other facets of the OMDoc document model are motivated by presentation-independence, for instance the distribution of whitespace is irrelevant even in text nodes, to allow formatting and reflow in the source code, which is not considered to change the information content of a text.

Listing 48.1: An OMDoc-Equal Representation for Listings 47.1 and 47.2

```
<docalt>
  <definition xml:id="comm.def.de" for="comm">
    <h:p xml:lang="de">Eine Operation <OMR href="#op"/>
      heißt kommutativ, falls
      <OMA id="comm1"><OMS cd="relation1" name="eq"/>
        <OMA><OMV name="op"/><OMV name="X"/><OMV name="Y"/></OMA>
        <OMA><OMV name="op"/><OMV name="Y"/><OMV name="X"/></OMA>
      </OMA>
      für alle <OMR href="#x"/> und <OMR href="#y"/>.
  </h:p>
  </definition>
  <definition xml:id="comm.def.en" for="#comm">
    <h:p xml:lang="en">
      An operation <OMV id="op" name="op"/> is called commutative,
      iff <OMR href="#comm1"/> for all <OMV id="x" name="X"/> and
      <OMV id="y" name="Y"/>.
    </h:p>
  </definition>
</docalt>
```

Compared to other document models, this is a rather weak (but general) notion of equality. Note in particular, that the OMDoc document model does *not* use mathematical equality here, which would make the formula $X + Y = Y + X$ (the `om:OMA` with `xml:id="comm1"` in Listing 48.1 instantiated with addition for `op`) mathematically equal to the trivial condition $X + Y = X + Y$, obtained by exchanging the right hand side $Y+X$ of the equality by $X+Y$, which is mathematically equal (but not OMDoc-equal).

Let us now specify (part of) the equality relation by the rules in the table in Figure 48.1. We have discussed a machine-readable form of these equality constraints in the XML schema for OMDoc in [**KohAng:tccmvc03**].

The last rule in Figure 48.1 is probably the most interesting, as we have seen in Part VI, OMDoc documents have both formal and informal aspects, they can contain *narrative* as well as *narrative-structured* information. The latter kind of document contains a formalization of a mathematical theory, as a reference for automated theorem proving systems. There, logical dependencies play a

#	Rule	comment	elements
1	unordered	The order of children of this element is irrelevant (as far as permitted by the content model). For instance only the order of `obligation` elements in the `axiom-inclusion` element is arbitrary, since the others must precede them in the content model.	`adt` `axiom-inclusion` `metadata` `symbol` `code` `private` `presentation` `omstyle`
2	multi-group	The order between siblings elements does not matter, as long as the values of the key attributes differ.	`requation` `dc:description` `sortdef` `data` `dc:title` `solution`
3	DAG encoding	Directedacyclicgraphs built up using `om:OMR` elements are equal, iff their tree expansions are equal.	`om:OMR` `ref`
4	Dataset	If the content of the `dc:type` element is `Dataset`, then the order of the siblings of the parent `metadata` element is irrelevant.	`dc:type`

Figure 48.1: The OMDoc Document Model

much greater role than the order of serialization in mathematical objects. We call such documents **content OMDoc** and specify the value `Dataset` in the `dc:type` element of the OMDoc metadata for such documents. On the other extreme we have human-oriented presentations of mathematical knowledge, e.g. for educational purposes, where didactic considerations determine the order of presentation. We call such documents **narrative-structured** and specify this by the value `Text` (also see the discussion in Part VII)

Chapter 49

OMDoc Sub-Languages

[=sub-languages]

In the last chapters we have described the OMDoc modules. Together, they make up the OMDoc document format, a very rich format for marking up the content of a wide variety of mathematical documents. (see [**Kohlhase:OMDoc1.6primer**] for some worked examples). Of course not all documents need the full breadth of OMDoc functionality, and on the other hand, not all OMDoc applications (see [**Kohlhase:OMDoc1.6projects**] for examples) support the whole language.

One of the advantages of a modular language design is that it becomes easy to address this situation by specifying sub-languages that only include part of the functionality. We will discuss plausible OMDoc sub-languages and their applications that can be obtained by dropping optional modules from OMDoc. Figure 49.1 visualizes the sub-languages we will present in this chapter. The full language OMDoc is at the top, at the bottom is a minimal sub-language OMDoc Basic, which only contains the required modules (mathematical documents without them do not really make sense). The arrows signify language inclusion and are marked with the modules acquired in the extension.

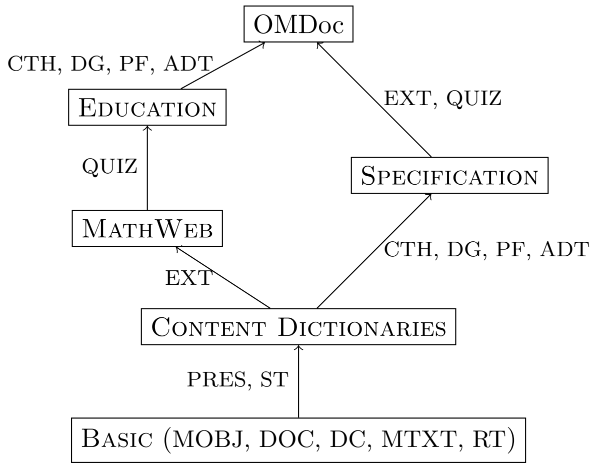

Figure 49.1: OMDoc Sub-Languages and Modules

The sub-language identifiers can be used as values of the `modules` attribute on the `omdoc` and `omdoc` elements. Used there, they abbreviate the list of modules these sub-languages contain.

49.1 Basic OMDoc

Basic OMDoc is sufficient for very simple mathematical documents that do not introduce new symbols or concepts, or for early (and non-specific) stages in the migration process from legacy representations of mathematical material (see ?spec@top-level?). This OMDoc sub-language consists of five modules: we need module MOBJ for mathematical objects and formulae, which are present in almost all mathematical documents. Module DOC provides the document infrastructure, and in particular, the root element `omdoc`. We need DC for titles, descriptions, and administrative metadata, and module MTXT so we can state properties about the mathematical objects in `omtext` element. Finally, module RT allows to structured text below the `omtext` level. This module is not strictly needed for basic OMDoc, but we have included it for convenience.

49.2 OMDoc Content Dictionaries

Content Dictionaries are used to define the meaning of symbols in the OpenMath standard [**BusCapCar:2oms04**], they are the mathematical documents referred to in the `cd` attribute of the `om:OMS` element. To express contentdictionaries in OMDoc, we need to add the module ST to Basic OMDoc. It provides the possibility to specify the meaning of basic mathematical objects (symbols) by axioms and definitions together with the infrastructure for inheritance, and grouping, and allows to reference the symbols defined via their home theory (see the discussion in Chapter 15).

With this extension alone, OMDoc content dictionaries add support for multilingual text, simple inheritance for theories, and document structure to the functionality of OpenMath content dictionaries. Furthermore, OMDoc content dictionaries allow the conceptual separation of mathematical properties into constitutive ones and logically redundant ones. The latter of these are not strictly essential for content dictionaries, but enhance maintainability and readability, they are included in OpenMath content dictionaries for documentation and explanation.

The sub-language for OMDoc content dictionaries also allows the specification of notations for the introduced symbols (by module PRES). So the resulting documents can be used for referencing (as in OpenMath) and as a resource for deriving presentation information for the symbols defined here. To get a feeling for this sub-language, see the example in the OMDoc variant of the OpenMath content dictionary `arith1` in ?spec@omcds?, which shows that the OpenMath content dictionary format is (isomorphic to) a subset of the OMDoc format. In fact, the OpenMath2 standard only presents the content dictionary format used here as one of many encodings and specifies abstract conditions on content dictionaries that the OMDoc encoding below also meets. Thus OMDoc is a valid content dictionary encoding.

49.3 Specification OMDoc

OMDoc content dictionaries are still a relatively lightweight format for the specification of meaning of mathematical symbols and objects. Large scale formal specification efforts, e.g. for program verification need more structure to be practical. Specification languages like Casl (Common Algebraic Specification Language [**CoFI:2004:CASL-RM**]) offer the necessary infrastructure, but have a syntax that is not integrated with web standards.

The Specification OMDoc sub-language adds the modules ADT and CTH to the language of OMDoc content dictionaries. The resulting language is equivalent to the Casl standard, see [**AutHut:tefsduc00**; **Hutter:mocsv00**; **MAH-06-a**] for the necessary theory.

The structured definition schemata from module ADT allow to specify abstract data types, sets of objects that are inductively defined from constructor symbols. The development graph structure built on the theory morphisms from module CTH allow to make inclusion assertions about theories that structure fragments of mathematical developments and support a Management of change.

49.4 MathWeb OMDoc

OMDoc can be used as a content-oriented basis for web publishing of mathematics. Documents for the web often contain images, applets, code fragments, and other data, together with mathematical statements and theories.

The OMDoc sub-language MathWeb OMDoc extends the language for OMDoc content dictionaries by the module EXT, which adds infrastructure for images, applets, code fragments, and other data.

49.5 Educational OMDoc

OMDoc is currently used as a content-oriented basis for various systems for mathematics education (see e.g. [**Kohlhase:OMDoc1.6primer**] for an example and discussion). The OMDoc sub-language Educational OMDoc extends MathWeb OMDoc by the module QUIZ, which adds infrastructure for exercises and assessments.

49.6 Reusing OMDoc modules in other formats

Another application of the modular language design is to share modules with other XML applications. For instance, formats like DocBook [**WalMue:dtdg99**] or XHTML [**W3C:xhtml2000**] could be extended with the OMDoc statement level. Including modules MOBJ, DC, and (parts of) MTXT, but not RT and DOC would result in content formats that mix the document-level structure of these formats. Another example is the combination of XML-RPC envelopes and OMDoc documents used for interoperability in [**Kohlhase:OMDoc1.6primer**].