`modules.sty`: Semantic Macros and Module Scoping in sTeX*

Michael Kohlhase, Dennis Müller
FAU Erlangen-Nürnberg
`http://kwarc.info/`

October 21, 2020

Abstract

The `modules` package is a central part of the sTeX collection, a version of TeX/LaTeX that allows to markup TeX/LaTeX documents semantically without leaving the document format, essentially turning TeX/LaTeX into a document format for mathematical knowledge management (MKM).

This package supplies a definition mechanism for semantic macros and a non-standard scoping construct for them, which is oriented at the semantic dependency relation rather than the document structure. This structure can be used by MKM systems for added-value services, either directly from the sTeX sources, or after translation.

*Version v1.6 (last revised 2020/10/14)

Contents

1 Introduction **3**

2 The User Interface **3**
2.1 Package Options . . . 3
2.2 Semantic Macros . . . 4
2.3 Testing Semantic Macros . . . 5
2.4 Axiomatic Assumptions . . . 6
2.5 Semantic Macros for Variables . . . 6
2.6 Symbol and Concept Names . . . 7
2.7 Modules, Inheritance, and sTeX Module Signatures . . . 8
2.8 Dealing with multiple Files . . . 9
2.9 Using Semantic Macros in Narrative Structures . . . 10
2.10 Including Externally Defined Semantic Macros . . . 11
2.11 Namespaces and Alignments . . . 11

3 Limitations & Extensions **12**
3.1 Qualified Imports . . . 12
3.2 Error Messages . . . 13
3.3 Crossreferencing . . . 13
3.4 No Forward Imports . . . 13

4 The Implementation **14**
4.1 Package Options . . . 14
4.2 Modules and Inheritance . . . 14
4.3 Semantic Macros . . . 20
4.4 Defining Math Operators . . . 23
4.5 Axiomatic Assumptions . . . 23
4.6 Semantic Macros for Variables . . . 23
4.7 Testing Semantic Macros . . . 24
4.8 Symbol and Concept Names . . . 24
4.9 Loading Modules . . . 25
4.10 Including Externally Defined Semantic Macros . . . 31
4.11 Namespaces and Alignments . . . 31
4.12 Deprecated Functionality . . . 31
4.13 Experiments . . . 31

1 Introduction

Following general practice in the TEX/LATEX community, we use the term "semantic macro" for a macro whose expansion stands for a mathematical object, and whose name (the command sequence) is inspired by the name of the mathematical object. This can range from simple definitions like `\def\Reals{\mathbb{R}}` for individual mathematical objects to more complex (functional) ones object constructors like `\def\SmoothFunctionsOn#1{\mathcal{C}^\infty(#1,#1)}`. Semantic macros are traditionally used to make TEX/LATEX code more portable. However, the TEX/LATEX scoping model (macro definitions are scoped either in the local group or until the rest of the document), does not mirror mathematical practice, where notations are scoped by mathematical environments like statements, theories, or such. For an in-depth discussion of semantic macros and scoping we refer the reader [Koh08].

The `modules` package provides a LATEX-based markup infrastructure for defining module-scoped semantic macros and LaTeXML bindings [LTX] to create OMDoc [Koh06] from STEX documents. In the STEX world semantic macros have a special status, since they allow the transformation of TEX/LATEX formulae into a content-oriented markup format like OpenMath [Bus+04] and (strict) content MathML [Aus+10]; see Figure 1 for an example, where the semantic macros above have been defined by the `\symdef` macros (see Section 2.2) in the scope of a `\begin{module}[id=calculus]` (see Section 2.7).

LATEX	`\SmoothFunctionsOn\Reals`
PDF/DVI	$\mathcal{C}^\infty(\mathbb{R},\mathbb{R})$
OpenMath	% <OMA> % <OMS cd="calculus" name="SmoothFunctionsOn"/> % <OMS cd="calculus" name="Reals"/> % </OMA>
MathML	% <apply> % <csymbol cd="calculus">SmoothFunctionsOn</csymbol> % <csymbol cd="calculus">Reals</csymbol> % </apply>

Example 1: OpenMath and MathML generated from Semantic Macros

2 The User Interface

The main contributions of the `modules` package are the `module` environment, which allows for lexical scoping of semantic macros with inheritance and the `\symdef` macro for declaration of semantic macros that underly the `module` scoping.

2.1 Package Options

`showmods` `qualifiedimports`

The `modules` package takes six options: If we set `showmods`, then the views (see Section **??**) are shown. If we set the `qualifiedimports` option, then qualified imports are enabled. Qualified imports give more flexibility in module inheritance,

but consume more internal memory. As qualified imports are not fully implemented at the moment, they are turned off by default see Limitation 3.1. The

noauxreq option `noauxreq` prohibits the registration of `\@requiremodules` commands in the `aux` file. They are necessary for preloading the modules so that entries in the table of contents can have semantic macros; but as they sometimes cause trouble the option allows to turn off preloading.[1]

EdN:1

showmeta If the `showmeta` option is set, then the metadata keys are shown (see [Koh20b] for details and customization options).

The `mh` option enables MathHub support; see [Koh20a].

trwarn Finally, if the `trwarn` is given, then the `modules` package only gives warnings instead of hard errors when term references are unknown.

2.2 Semantic Macros

\symdef The is the main constructor for semantic macros in LaTeX. A call to the `\symdef` macro has the general form

`\symdef[`⟨*keys*⟩`]{`⟨*cseq*⟩`}[`⟨*args*⟩`]{`⟨*definiens*⟩`}`

where ⟨*cseq*⟩ is a control sequence (the name of the semantic macro) ⟨*args*⟩ is a number between 0 and 9 for the number of arguments ⟨*definiens*⟩ is the token sequence used in macro expansion for ⟨*cseq*⟩. Finally ⟨*keys*⟩ is a keyword list that further specifies the semantic status of the defined macro.

The two semantic macros in Figure 1 would have been declared by invocations of the `\symdef` macro of the form:

```
\symdef{Reals}{\mathbb{R}}
\symdef{SmoothFunctionsOn}[1]{\mathcal{C}^\infty(#1,#1)}
```

Note that both semantic macros correspond to OpenMath or MathML "symbols", i.e. named representations of mathematical concepts (the real numbers and the constructor for the space of smooth functions over a set); we call these names the **symbol name** of a semantic macro. Normally, the symbol name of a semantic macro declared by a `\symdef` directive is just ⟨*cseq*⟩. The key-value pair

name `name=`⟨*symname*⟩ can be used to override this behavior and specify a differing name. There are two main use cases for this.

The first one is shown in Example 3, where we define semantic macros for the "exclusive or" operator. Note that we define two semantic macros: `\xorOp` and `\xor` for the applied form and the operator. As both relate to the same mathematical concept, their symbol names should be the same, so we specify `name=xor` on the definition of `\xorOp`.

local A key `local` can be added to ⟨*keys*⟩ to specify that the symbol is local to the module and is invisible outside. Note that even though `\symdef` has no advantage over `\def` for defining local semantic macros, it is still considered good style to

[1] EdNote: MK: is this still needed without sms files?

use \symdef and \abbrdef, if only to make switching between local and exported semantic macros easier.

primary Finally, the key primary (no value) can be given for primary symbols.

\abbrdef The \abbrdef macro is a variant of \symdef that is only different in semantics, not in presentation. An abbreviative macro is like a semantic macro, and underlies the same scoping and inheritance rules, but it is just an abbreviation that is meant to be expanded, it does not stand for an atomic mathematical object.

We will use a simple module for natural number arithmetics as a running example. It defines exponentiation and summation as new concepts while drawing on the basic operations like + and − from LaTeX. In our example, we will define a semantic macro for summation \Sumfromto, which will allow us to express an expression like $\sum i = 1^n x^i$ as \Sumfromto{i}1n{2i-1} (see Example 2 for an example). In this example we have also made use of a local semantic symbol for n, which is treated as an arbitrary (but fixed) symbol.

```
\begin{module}[id=arith]
  \symdef{Sumfromto}[4]{\sum_{#1=#2}^{#3}{#4}}
  \symdef[local]{arbitraryn}{n}
  What is the sum of the first $\arbitraryn$ odd numbers, i.e.
  $\Sumfromto{i}1\arbitraryn{2i-1}?$
\end{module}
```

What is the sum of the first n odd numbers, i.e. $\sum_{i=1}^{n} 2i - 1$?

Example 2: Semantic Markup in a module Context

\symvariant The \symvariant macro can be used to define presentation variants for semantic macros previously defined via the \symdef directive. In an invocation

\symdef[⟨*keys*⟩]{⟨*cseq*⟩}[⟨*args*⟩]{⟨*pres*⟩}
\symvariant{⟨*cseq*⟩}[⟨*args*⟩]{⟨*var*⟩}{⟨*varpres*⟩}

the first line defines the semantic macro \⟨*cseq*⟩ that when applied to ⟨*args*⟩ arguments is presented as ⟨*pres*⟩. The second line allows the semantic macro to be called with an optional argument ⟨*var*⟩: \⟨*cseq*⟩[var] (applied to ⟨*args*⟩ arguments) is then presented as ⟨*varpres*⟩. We can define a variant presentation for \xor; see Figure 3 for an example.

\resymdef Version 1.0 of the modules package had the \resymdef macro that allowed to locally redefine the presentation of a macro. But this did not interact well with the beamer package and was less useful than the \symvariant functionality. Therefore it is deprecated now and leads to an according error message.

2.3 Testing Semantic Macros

One of the problems in managing large module graphs with many semantic macros, so the module package gives an infrastructure for unit testing. The first macro is

\symtest \symtest, which allows the author of a semantic macro to generate test output (if the symtest option is set) see figure 4 for a "tested semantic macro definition". Note that the language in this purely generated, so that it can be adapted (tbd).

```
\begin{module}[id=xbool]
  \symdef[name=xor]{xorOp}{\oplus}
  \symvariant{xorOp}{uvee}{\underline{\vee}}
  \symdef{xor}[2]{#1\xorOp #2}
  \symvariant{xor}[2]{uvee}{#1\xorOp[uvee] #2}
  Exclusive disjunction is commutative: $\xor{p}q=\xor{q}p$\\
  Some authors also write exclusive or with the $\xorOp[uvee]$ operator,
  then the formula above is $\xor[uvee]{p}q=\xor[uvee]{q}p$
\end{module}
```

Exclusive disjunction is commutative: $p \oplus q = q \oplus p$
Some authors also write exclusive or with the $\underline{\vee}$ operator, then the formula above is $p \underline{\vee} q = q \underline{\vee} p$

Example 3: Presentation Variants of a Semantic Macro

```
\symdef[name=setst]{SetSt}[2]{\{#1\,\vert\,#2\}}
\symtest[name=setst]{SetSt}{\SetSt{a}{a>0}}
```

generates the output
Symbol `setst` with semantic macro `\SetSt`: used e.g. in $\{a \,|\, a > 0\}$

Example 4: A Semantic Macro Definition with Test

`\abbrtest` The `\abbrtest` macro gives the analogous functionality for `\abbrdef`.

2.4 Axiomatic Assumptions

`\assdef` In many ways, axioms and assumptions in definitions behave a lot like symbols (see [RK13] for discussion). Therefore we provide the macro `\assdef` that can be used to mark up assumptions. Given a phrase ⟨*phrase*⟩ in a definition[2], we can use `\assdef{`⟨*name*⟩`}{`⟨*phrase*⟩`}` to give this the symbol name ⟨*name*⟩.[3]

EdN:2
EdN:3

2.5 Semantic Macros for Variables

Up to now, the semantic macros generated OpenMath and MathML markup where the heads of the semantic macros become constants (the `OMS` and `csymbol` elements in Figure 1). But sometimes we want to have semantic macros for variables, e.g. to associate special notation conventions. For instance, if we want to define mathematical structures from components as in Figure 5, where the semigroup operation $\circ$ is a variable epistemologically, but is a n-ary associative operator – we are in a semigroup after all. Let us call such variables **semantic variables** to contrast them from **semantic constants** generated by `\symdef` and `\symvariant`.

Semantic variables differ from semantic constants in two ways:

1. they do not participate in the imports mechanism and

[2] EdNote: only definitions?
[3] EdNote: continue

Definition 3.17 Let $\langle G, \circ\rangle$ be a semigroup, then we call $e \in G$ a **unit**, iff $e \circ x = x \circ e = x$. A semigroup with unit $\langle G, \circ, e\rangle$ is called a **monoid**.

Example 5: A Definition of a Structure with "semantic variables".

2. they generate markup with variables.

In the case of Figure 5 we (want to) have the XML markup in Figure 6. To associate the notation to the variables, we define semantic macros for them, here the macro `\op` for the (semigroup) operation via the `\vardef` macro. `\vardef` works exactly like, except

`\vardef`

1. semantic variables are local to the current TEX group and
2. they generate variable markup in the XML

STEX	`\vardef{op}[1]{\assoc\circ{#1}}`

OMDOC	`% <notation>`
	`%  <prototype>`
	`%  <OMA>`
	`%  <OMV name="op"/>`
	`%  <expr name="a1"/>`
	`%  <expr name="a2"/>`
	`%  </OMA>`
	`%  </prototype>`
	`%  <rendering>`
	`%  <mrow>`
	`%  <render name="a1"/>`
	`%  <mo>&#x2384;</mo>`
	`%  <render name="a2"/>`
	`%  </mrow>`
	`%  </rendering>`
	`% </notation>`

LATEX	`\op{x,e}`
PDF/DVI	$x \circ e$
OPENMATH	`% <OMA><OMV name="op"/><OMV name="x"/><OMV name="e"/></OMA>`
MATHML	`% <apply><ci>op</ci><ci>x</ci><ci>e</ci></apply>`

Example 6: Semantic Variables in OPENMATH and MATHML

2.6 Symbol and Concept Names

Just as the `\symdef` declarations define semantic macros for mathematical symbols, the `modules` package provides an infrastructure for *mathematical concepts* that are expressed in mathematical vernacular. The key observation here is that concept names like "finite symplectic group" follow the same scoping rules as mathematical symbols, i.e. they are module-scoped. The `\termdef` macro is an analogue to `\symdef` that supports this: use `\termdef[`⟨*keys*⟩`]{`⟨*cseq*⟩`}{`⟨*concept*⟩`}` to declare the macro `\`⟨*cseq*⟩ that expands to ⟨*concept*⟩. See Figure 7 for an example, where we use the `\capitalize` macro to adapt ⟨*concept*⟩ to the sentence

`\termdef`

`\capitalize`

EdN:4 beginning.[4]. The main use of the \termdef-defined concepts lies in automatic cross-referencing facilities via the \termref and \symref macros provided by the statements package [Koh20c]. Together with the hyperref package [RO], this provide cross-referencing to the definitions of the symbols and concepts. As discussed in section 3.3, the \symdef and \termdef declarations must be on top-level in a module, so the infrastructure provided in the modules package alone cannot be used to locate the definitions, so we use the infrastructure for mathematical statements for that.

\termref
\symref

```
\termdef[name=xor]{xdisjunction}{exclusive disjunction}
\capitalize\xdisjunction is commutative: $\xor{p}q=\xor{q}p$
```

Example 7: Extending Example 3 with Term References

2.7 Modules, Inheritance, and SᴛEX Module Signatures

module The module environment takes an optional KeyVal argument. Currently, only the id key is supported for specifying the identifier of a module (also called the module name). A module introduced by \begin{module}[id=foo] restricts the scope the semantic macros defined by the \symdef form to the end of this module given by the corresponding \end{module}, and to any other module environments that import them by a \importmodule{foo} directive. If the module foo contains \importmodule directives of its own, these are also exported to the importing module.

\importmodule Thus the \importmodule declarations induce the semantic inheritance relation. Figure 8 shows a module that imports the semantic macros from three others. In the simplest form, \importmodule{⟨*mod*⟩} will activate the semantic macros and concepts declared by \symdef and \termdef in module ⟨*mod*⟩ in the current module[1]. To understand the mechanics of this, we need to understand a bit of the internals. The module environment sets up an internal macro pool, to which all the macros defined by the \symdef and \termdef declarations are added; \importmodule only activates this macro pool. Therefore \importmodule{⟨*mod*⟩} can only work, if the TEX parser — which linearly goes through the SᴛEX sources — already came across the module ⟨*mod*⟩. In many situations, this is not obtainable; e.g. for "semantic forward references", where symbols or concepts are previewed or motivated to knowledgeable readers before they are formally introduced or for modularizations of documents into multiple files. We come to this next: [5]

EdN:5

\metalanguage The \metalanguage macro is a variant of importmodule that imports the meta language, i.e. the language in which the meaning of the new symbols is expressed. For mathematics this is often first-order logic with some set theory; see [RK13] for discussion.

[4] EdNote: continue, describe ⟨*keys*⟩, they will have to to with plurals,... once implemented

[1] Actually, in the current TEX group, therefore \importmodule should be placed directly after the \begin{module}.

[5] EdNote: MK: document the other keys of module

2.8 Dealing with multiple Files

The infrastructure presented above works well if we are dealing with small files or small collections of modules. In reality, collections of modules tend to grow, get re-used, etc, making it much more difficult to keep everything in one file. This general trend towards increasing entropy is aggravated by the fact that modules are very self-contained objects that are ideal for re-used. Therefore in the absence of a content management system for LaTeX document (fragments), module collections tend to develop towards the "one module one file" rule, which leads to situations with lots and lots of little files.

Moreover, most mathematical documents are not self-contained, i.e. they do not build up the theory from scratch, but pre-suppose the knowledge (and notation) from other documents. In this case we want to make use of the semantic macros from these prerequisite documents without including their text into the current document.

`\importmodule` The `\importmodule` macro can be given an optional first keyword argument that can be used to specify which sTeX modules to load.

`load` `\importmodule[load=`⟨*filepath*⟩`]{`⟨*mod*⟩`}` will read the sTeX file at ⟨*filepath*⟩`.tex` without producing output (if it exists and has not been loaded before) and activate the semantic macros from module ⟨*mod*⟩ (which was supposedly defined in ⟨*filepath*⟩`.tex`). Note that an `\importmodule` recursively loads all necessary files to supply the semantic macros inherited by the current module. `\importmhmodule` does not produce output, it only uses the side effects of the symbol declarations `\symi*` and definitions (`\symdef` and `\symvariant`) the module environment (`\begin`/`\end{module}`, as well as the `\import`/`\usemodule` directives – we jointly call this information the **sTeX module signatures**.

Thus `\importmodule` can be used to make module files truly self-contained. To arrive at a file-based content management system, it is good practice to reuse the module identifiers as module names and to prefix module files with corresponding `\importmodule` statements that pre-load the corresponding module files. But this leads to tedious duplication: We see imports of the form

```
\importmhmodule[path=foo/en/very-long-name]{very-long-name}
```

`dir` To avoid this, `\importmhmodule` allows a second key: `dir`, which specifies the directory of the sTeX module. So we can write

```
\importmhmodule[dir=foo/en]{very-long-name}
```

instead when the module name and file name coincide. This also avoids the maintenance problems (typos) induced by duplication. If both `path` and `dir` are given, the latter takes precedence.

In Example 8, we have shown the typical setup of a module file. The `\importmodule` macro takes great care that files are only read once, as sTeX allows multiple inheritance and this setup would lead to an exponential (in the module inheritance depth) number of file loads.

```
\documentclass{article}
\usepackage{stex}
\begin{document}
...
\begin{module}[id=foo]
\importmodule[dir=../other]{bar}
\importmodule[load=../mycolleaguesmodules]{baz}
\importmodule[load=../other/bar]{foobar}
  ...
\end{module}
...
\end{document}
```

Example 8: Self-contained Modules via `importmodule`

Note that – as sTEX uses the `standalone` package [Sch], we can make the sTEX modules "standalone", i.e. with a `\documentclass`, preamble and `\begin`/`\end{document}` that can directly be formatted with `pdflatex`, but also included with `\include`/`\usemodule`. This is very convenient for distributing functionality into sTEX modules.

Note that the recursive (depth-first) nature of the file loads induced by this setup is very natural, but can lead to problems with the depth of the file stack in the TEX formatter (it is usually set to something like 15[2]). Therefore, it may be necessary to circumvent the recursive load pattern providing (logically spurious) `\importmodule` commands. Consider for instance module `bar` in Example 8, say that `bar` already has load depth 15, then we cannot naively import it in this way. If module `bar` depended say on a module `base` on the critical load path, then we could add a statement `\requiremodules{../base}` in the second line. This would load the modules from `../base.tex` in advance (uncritical, since it has load depth 10) without activating them, so that it would not have to be re-loaded in the critical path of the module `foo`. Solving the load depth problem.

`\requiremodules`

`\inputref`

The `\inputref` macro behaves just like `\input` in the LATEX workflow, but in the LaTeXML conversion process creates a reference to the transformed version of the input file instead. Moreover, spacing can be customized by the `\inputref@preskip` and `\inputref@postskip` macros, which default to nothing, but could be customized e.g. to `\medskip`.

`\inputref@preskip`

`\inputref@postskip`

2.9 Using Semantic Macros in Narrative Structures

The `\importmodule` macro establishes the inheritance relation, a transitive relation among modules that governs visibility of semantic macros. In particular, it can only be used in modules (and has to be used at the top-level, otherwise it is hindered by LATEX groups). In many cases, we only want to *use* the semantic macros in an environment (and not re-export them). Indeed, this is the normal

[2] If you have sufficient rights to change your TEX installation, you can also increase the variable `max_in_open` in the relevant `texmf.cnf` file. Setting it to 50 usually suffices

situation for most parts of mathematical documents. For that sTeX provides the
\usemodule \usemodule macro, which takes the same arguments as \importmodule, but the semantic macros the module imports are not re-exported from the current module. A typical situation is shown in Figure 9, where we open the module ring (see Figure ??) and use its semantic macros (in the omtext environment). In earlier versions of sTeX, we would have to wrap the omtext environment in an anonymous module environment to prevent re-export.

```
\begin{omtext}
  \usemodule[load=../algebra/rings.tex]{ring}
  We $R$ be a ring $(\rbase,\rplus,\rzero,\rminusOp,\rtimes,\rone)$, ...
\end{omtext}
```

Example 9: Using Semantic Macros in Narrative Structures

2.10 Including Externally Defined Semantic Macros

In some cases, we use an existing LaTeX macro package for typesetting objects that have a conventionalized mathematical meaning. In this case, the macros are "semantic" even though they have not been defined by a \symdef. This is no problem, if we are only interested in the LaTeX workflow. But if we want to e.g. transform them to OMDoc via LaTeXML, the LaTeXML bindings will need to contain references to an OMDoc theory that semantically corresponds to the LaTeX package. In particular, this theory will have to be imported in the generated OMDoc file to make it OMDoc-valid.

\requirepackage To deal with this situation, the modules package provides the \requirepackage macro. It takes two arguments: a package name, and a URI of the corresponding OMDoc theory. In the LaTeX workflow this macro behaves like a \usepackage on the first argument, except that it can — and should — be used outside the LaTeX preamble. In the LaTeXML workflow, this loads the LaTeXML bindings of the package specified in the first argument and generates an appropriate imports element using the URI in the second argument.

2.11 Namespaces and Alignments

We often want to align the content of sTeX modules to formalizations, e.g. to take advantage of type declarations there. For this, we extend the keys of the module
align environment and the symdef macro with a key align whose value is an external MMT theory or symbol name respectively. Note that symbols can only be aligned in aligned modules.

As full MMT URIs are of the form $\langle URI\rangle$?$\langle theory\rangle$?$\langle name\rangle$, we need a way to specify the $\langle URI\rangle$. We adopt the system of **namespaces** of MMT [MNS]:
\namespace the macro declares a namespace URI. If the optional argument is given, then this is a namespace abbreviation declaration, which can be used later to reference theories/modules from other namespaces.

The example below shows off all possibilities. We first declare the namespace of the document (which places all theories and their symbols into this namespace). Then we add two more a namespace abbreviation: `sets:` and `moresets:` that we will use to for the alignments in the module. We use the `ns` and `align` keys in the `module` environment to specify that the external theory `sets:?ESet` is the default alignment target, i.e. any symbol that in the `emptyset` module is aligned by default to the symbol with the same name in the external `sets:?ESet` theory.

```
\namespace{http://mathhub.info/smglom/sets}
\namespace[sets]{http://mathhub.info/MitM/smglom/sets}
\namespace[moresets]{http://mathhub.info/more/sets}
\begin{module}[creators=miko,ns=sets,align=ESet]{emptyset}
  \importmodule{set}
  \symdef[assocarg=1]{set}[1]{\{#1\}}
  \symdef[align=empty]{eset}{\emptyset}
  \symdef[align=AEset?eset]{aeset}{\emptyset^+}
  \symdef[ns=moresets,align=fuzzy?eset]{feset}{\emptyset^f}
  \symdef[noalign]{neset}{\emptyset^*}
\end{module}
```

The first `\symdef` aligns the symbol `emptyset?set` with `sets:?ESet?set` via default alignment. This breaks down for the symbol `eset`, so we specify an alignment to the symbol `sets:?empty` via the `align` key on the `\symdef`. If we want to align with a different theory we can just use the `?` notation as for `aeset`. A different namespace can be specified by the `ns` key, and finally, we can indicate that a symbol should not be aligned via the `noalign` key.

3 Limitations & Extensions

In this section we will discuss limitations and possible extensions of the `modules` package. Any contributions and extension ideas are welcome; please discuss ideas, requests, fixes, etc. on the sTeX issue tracker at [sTeX].

3.1 Qualified Imports

In an earlier version of the `modules` package we used the `usesqualified` for importing macros with a disambiguating prefix (this is used whenever we have conflicting names for macros inherited from different modules). This is not accessible from the current interface. We need something like a `\importqualified` macro for this; see [sTeX, issue #1505]. Until this is implemented the infrastructure is turned off by default, but we have already introduced the `qualifiedimports` option for the future.

`qualifiedimports`

3.2 Error Messages

The error messages generated by the `modules` package are still quite bad. For instance if `thyA` does note exists we get the cryptic error message

```
! Undefined control sequence.
\module@defs@thyA ...hy
                      \expandafter \mod@newcomma...
l.490 ...ortmodule{thyA}
```

This should definitely be improved.

3.3 Crossreferencing

Note that the macros defined by `\symdef` are still subject to the normal TEX scoping rules. Thus they have to be at the top level of a module to be visible throughout the module as intended. As a consequence, the location of the `\symdef` elements cannot be used as targets for crossreferencing, which is currently supplied by the `statement` package [Koh20c]. A way around this limitation would be to import the current module from the STEX module (see Section 2.7) via the `\importmodule` declaration.

3.4 No Forward Imports

STEX allows imports in the same file via `\importmodule{`⟨*mod*⟩`}`, but due to the single-pass linear processing model of TEX, ⟨*mod*⟩ must be the name of a module declared *before* the current point. So we cannot have forward imports as in

```
\begin{module}[id=foo]
  \importmodule{mod}
  ...
\end{module}
...
\begin{module}[id=mod]
  ...
\end{module}
```

a workaround, we can extract the module ⟨*mod*⟩ into a file mod.tex and replace it with `\inputref{mod}`, as in

```
\begin{module}[id=foo]
  \importmodule[load=mod]{mod}
  ...
\end{module}
...
\inputref{mod}
```

then the `\importmodule` command can read `mod.tex` without having to wait for the module ⟨*mod*⟩ to be defined.

4 The Implementation

The `modules` package generates two files: the LaTeX package (all the code between ⟨*package⟩ and ⟨/package⟩) and the LaTeXML bindings (between ⟨*ltxml⟩ and ⟨/ltxml⟩). We keep the corresponding code fragments together, since the documentation applies to both of them and to prevent them from getting out of sync.

4.1 Package Options

We declare some switches which will modify the behavior according to the package options. Generally, an option `xxx` will just set the appropriate switches to true (otherwise they stay false). The options we are not using, we pass on to the `sref` package we require next.

```
⟨*package⟩
\newif\if@modules@html@\@modules@html@true
\DeclareOption{omdocmode}{\@modules@html@false}
\newif\if@modules@mh@\@modules@mh@false
\DeclareOption{mh}{\@modules@mh@true}
\newif\ifmod@show\mod@showfalse
\DeclareOption{showmods}{\mod@showtrue}
\newif\ifaux@req\aux@reqtrue
\DeclareOption{noauxreq}{\aux@reqfalse}
\newif\ifmod@qualified\mod@qualifiedfalse
\DeclareOption{qualifiedimports}{\mod@qualifiedtrue}
\newif\if@trwarn\@trwarnfalse
\DeclareOption{trwarn}{\@trwarntrue}
\DeclareOption*{\PassOptionsToPackage{\CurrentOption}{sref}}
\ProcessOptions

\RequirePackage{stex-base}
\RequirePackage{sref}
\RequirePackage{pathsuris}
\RequirePackage{currfile}
\RequirePackage{standalone}
\if@modules@mh@\RequirePackage{modules-mh}\fi
\RequirePackage{xspace}
\if@latexml\else\ifmod@show\RequirePackage{mdframed}\fi\fi
```

4.2 Modules and Inheritance

We define the keys for the `module` environment and the actions that are undertaken, when the keys are encountered.

```
\addmetakey*{module}{title}
\addmetakey*{module}{id}
\addmetakey*{module}{creators}
\addmetakey*{module}{contributors}
\addmetakey*{module}{srccite}
\addmetakey*{module}{align}[WithTheModuleOfTheSameName]
\addmetakey*{module}{ns}
```

```
\addmetakey*{module}{narr}
\addmetakey*{module}{noalign}[true]
```

`module@heading` We make a convenience macro for the module heading. This can be customized.

```
\ifdef{\thesection}{\newcounter{module}[section]}{\newcounter{module}}%
\newrobustcmd\module@heading{%
  \stepcounter{module}%
  \ifmod@show%
  \noindent{\textbf{Module} \thesection.\themodule [\module@id]}%
  \sref@label@id{Module \thesection.\themodule [\module@id]}%
    \ifx\module@title\@empty :\quad\else\quad(\module@title)\hfill\\\fi%
  \fi%
}% mod@show
```

`module` Finally, we define the begin module command for the module environment. Much of the work has already been done in the keyval bindings, so this is quite simple. We store the file name (without extension) and extension of the module file in the global macros `\module@`⟨*name*⟩`@path` and `\module@`⟨*name*⟩`@ext`, so that we can use them later. The source of these two macros, `\mod@path` and `\mod@ext`, are defined in `\requiremodules`.

```
\newenvironment{module}[1][]{%
  \begin{@module}[#1]%
    \ifcsundef{mod@path}{}{\csxdef{module@\module@id @path}{\mod@path}}%
    \ifcsundef{mod@ext}{}{\csxdef{module@\module@id @ext}{\mod@ext}}%
  \module@heading% make the headings
  \ignorespacesandpars\usemodule@maybesetcodes}{%
  \end{@module}%
  \ignorespacesafterend%
}%
\ifmod@show\surroundwithmdframed{module@om@common}\fi%
```

`@module` A variant of the `module` environment that does not create printed representations (in particular no frames).

For a module with uri ⟨*uri*⟩, we have a macro `\module@defs@`⟨*uri*⟩ that acts as a repository for semantic macros of the current module. I will be called by `\importmodule` to activate them. We will add the internal forms of the semantic macros whenever `\symdef` is invoked. To do this, we will need an unexpended form `\this@module` that expands to `\module@defs@`⟨*uri*⟩; we define it first and then initialize `\module@defs@`⟨*uri*⟩ as empty. Then we do the same for qualified imports as well (if the `qualifiedimports` option was specified). Furthermore, we save the module name in the token register `\module@id`.

To compute the ⟨*uri*⟩ of a module, `\set@default@ns` computes the namespace, if none is provided as an optional argument, as follows:

If the file of the module is `/some/path/file.tex` and we are not in a `MathHub` repository, the namespace is `file:///some/path`.

If the file of the module is `/some/path/in/mathhub/repo/sitory/source/sub/file.tex` and `repo/sitory` is an archive in the `MathHub` root, and the `MANIFEST.MF`

of `repo/sitory` declares a namespace `http://some.namespace/foo`, then the namespace of the module is `http://some.namespace/foo/sub`.

```
\newif\ifarchive@ns@empty@\archive@ns@empty@false
\def\set@default@ns{
  \edef\@module@ns@temp{\currfiledir}
  \if@iswindows@\edef\@module@ns@temp{\windows@to@path\@module@ns@temp}\fi
  \archive@ns@empty@false
  \unless\ifcsname mh@currentrepos\endcsname
    \archive@ns@empty@true
  \else
    \expandafter\ifx\csname currentrepos@ns@\mh@currentrepos\endcsname\@empty\archive@ns@empty@t
  \fi
  \ifarchive@ns@empty@
    \edef\@module@ns@temp{file\@Colon\@Slash\@Slash\@module@ns@temp}
    \setkeys{module}{ns=\@module@ns@temp}
  \else
    \edef\@module@filepath@temppath{\@module@ns@temp}
    \edef\@module@ns@tempuri{\csname currentrepos@ns@\mh@currentrepos\endcsname}
    \edef\@module@archivedirpath{\csname currentrepos@dir@\mh@currentrepos\endcsname\@Slash sou
    \edef\@module@archivedirpath{\expandafter\detokenize\expandafter{\@module@archivedirpath}}
    \IfBeginWith\@module@filepath@temppath\@module@archivedirpath{
      \StrLen\@module@archivedirpath[\ns@temp@length]
      \StrGobbleLeft\@module@filepath@temppath\ns@temp@length[\@module@filepath@temprest]
      \edef\@module@ns@tempuri{\@module@ns@tempuri\@module@filepath@temprest}
    }{}
    \setkeys{module}{ns=\@module@ns@tempuri}
  \fi
}
```

If the module is not given an `id`, `\set@next@moduleid` computes one by enumeration, e.g. `module0`, `module1`, etc.

```
\def\set@next@moduleid{
  \unless\ifcsname namespace@\module@ns @unnamedmodules\endcsname
      \csgdef{namespace@\module@ns @unnamedmodules}{0}
  \fi
  \edef\namespace@currnum{\csname namespace@\module@ns @unnamedmodules\endcsname}
  \edef\module@temp@setidname{\noexpand\setkeys{module}{id=module\namespace@currnum}}
  \module@temp@setidname
  \csxdef{namespace@\module@ns @unnamedmodules}{\the\numexpr\namespace@currnum+1}
}
```

Finally, the `@module` environment does the actual work, i.e. setting metakeys, computing namespace/id, defining `\this@module`, etc.

```
\newenvironment{@module}[1][]{
  \metasetkeys{module}{#1}
  \ifx\module@ns\@empty\set@default@ns\fi
  \ifx\module@narr\@empty
    \setkeys{module}{narr=\module@ns}
  \fi
  \ifcsname module@id\endcsname
```

```
    \ifx\module@id\@empty\set@next@moduleid\fi
  \else\set@next@moduleid\fi
  %\seturi[module@uri@]{\module@ns\@QuestionMark\module@id}
  \edef\module@uri@uri{\module@ns\@QuestionMark\module@id} % faster, and at this point equivale
  \csxdef{\module@uri@uri}{\noexpand\@invoke@module{\module@uri@uri}}
  \csxdef{moduleid@\module@id}{\noexpand\@invoke@module{\module@uri@uri}}
  \csxdef{Module\module@id}{\noexpand\@invoke@module{\module@uri@uri}}
  \edef\this@module{%
    \expandafter\noexpand\csname module@defs@\module@uri@uri\endcsname%
  }%
  \csxdef{module@defs@\module@uri@uri}{%
    \expandafter\def\expandafter\noexpand\csname Module\module@id\endcsname%
      {\noexpand\@invoke@module{\module@uri@uri}}%
  }%
  \ifmod@qualified%
    \edef\this@qualified@module{%
      \expandafter\noexpand\csname module@defs@\module@uri@uri\endcsname%
    }%
    \csxdef{module@defs@qualified@\module@uri@uri}{%
      \expandafter\def\expandafter\noexpand\csname Module\module@id\endcsname%
      {\noexpand\@invoke@module{\module@uri@uri}}%
    }%
  \fi%
}{}%
```

A module with URI ⟨*uri*⟩ and id ⟨*id*⟩ creates two macros \⟨*uri*⟩ and `\Module`⟨*id*⟩, that ultimately expand to `\@invoke@module{`⟨*uri*⟩`}`. Currently, the only functionality is `\@invoke@module{`⟨*uri*⟩`}\@URI`, which expands to the full uri of a module (i.e. via `\Module`⟨*id*⟩`\@URI`). In the future, this macro can be extended with additional functionality, e.g. accessing symbols in a macro for overloaded (macro-)names.

```

\def\@URI{uri}

\def\@invoke@module#1#2{%
  \ifx\@URI#2%
    #1%
  \else%
    % TODO something else
    #2%
  \fi%
}

```

`\activate@defs` To activate the `\symdef`s from a given module ⟨*mod*⟩, we call the macro `\module@defs@`⟨*mod*⟩. But to make sure that every module is activated only once, we only activate if the macro `\module@defs@`⟨*mod*⟩ is undefined, and define it directly afterwards to prohibit further activations.

```
\def\activate@defs#1{%
```

```
  \ifcsundef{module@\csname moduleid@#1\endcsname\@URI @activated}%
    {\csname module@defs@\csname moduleid@#1\endcsname\@URI\endcsname}{}%
  \@namedef{module@\csname moduleid@#1\endcsname\@URI @activated}{true}%
}%
```

`\export@defs` `\export@defs{`⟨*mod*⟩`}` exports all the `\symdef`s from module ⟨*mod*⟩ to the current module (if it has the name ⟨*currmod*⟩), by adding a call to `\module@defs@`⟨*mod*⟩ to the registry `\module@defs@`⟨*currmod*⟩.

Naive understanding of this code: `#1` be will be expanded first, then `\this@module`, then `\active@defs`, then `\g@addto@macro`.

```
\def\g@addto@macro@safe#1#2{\ifx#1\relax\def#1{}\fi\g@addto@macro#1{#2}}
\def\export@defs#1{\@ifundefined{module@id}{}{%
\expandafter\expandafter\expandafter\g@addto@macro@safe%
\expandafter\this@module\expandafter{\activate@defs{#1}}}}%
```

Now we come to the implementation of `\importmodule`, but before we do, we define conditional and an auxiliary macro:

`\if@importing` `\if@importing` can be used to shut up macros in an import situation.

```
\newif\if@importing\@importingfalse
```

`\update@used@modules` This updates the register `\used@modules`

```
\newcommand\update@used@modules[1]{%
  \ifx\used@modules\@empty%
    \edef\used@modules{#1}%
  \else%
    \edef\used@modules{\used@modules,#1}%
  \fi}
```

`\importmodule` The `\importmodule[`⟨*file*⟩`]{`⟨*mod*⟩`}` macro is an interface macro that loads ⟨*file*⟩ and activates and re-exports the `\symdef`s from module ⟨*mod*⟩. As we will (probably) need to keep a record of the currently imported modules (top-level only), we divide the functionality into a user-visible macro that records modules in the `\used@modules` register and an internal one (`\@importmodule`) that does the actual work.

```
\gdef\used@modules{}
\srefaddidkey{importmodule}
\addmetakey{importmodule}{load}
\addmetakey{importmodule}{dir}
\addmetakey[false]{importmodule}{conservative}[true]
\newcommand\importmodule[2][]{%
\metasetkeys{importmodule}{#1}%
\usemodule@maybesetcodes
\update@used@modules{#2}%
\ifx\importmodule@dir\@empty
\@importmodule[\importmodule@load]{#2}{export}%
\else\@importmodule[\importmodule@dir/#2]{#2}{export}\fi%
\ignorespacesandpars}
```

`\@importmodule` `\@importmodule[`⟨*filepath*⟩`]{`⟨*mod*⟩`}{`⟨*export?*⟩`}` loads ⟨*filepath*⟩`.tex` and activates the module ⟨*mod*⟩. If ⟨*export?*⟩ is `export`, then it also re-exports the `\symdef`s from ⟨*mod*⟩.

First `\@load` will store the base file name with full path, then check if `\module@`⟨*mod*⟩`@path` is defined. If this macro is defined, a module of this name has already been loaded, so we check whether the paths coincide, if they do, all is fine and we do nothing otherwise we give a suitable error. If this macro is undefined we load the path by `\requiremodules`.

```
\newcommand\@importmodule[3][]{%
{\@importingtrue% to shut up macros while in the group opened here
\edef\@load{#1}%
\edef\@load{\expandafter\detokenize\expandafter{\@load}}%
\ifx\@load\@empty\relax\else%
\ifcsundef{module@#2@path}{\requiremodules{#1}}%
{\edef\@path{\csname module@#2@path\endcsname}%
\IfStrEq\@load\@path{\relax}% if the known path is the same as the requested one do nothing
{\PackageError{modules}% else signal an error
{Module Name Clash\MessageBreak
A module with name #2 was already loaded under the path "\@path"\MessageBreak
The imported path "\@load" is probably a different module with the\MessageBreak
same name; this is dangerous -- not importing}%
{Check whether the Module name is correct}}}%
\fi}%
\activate@defs{#2}% activate the module
\edef\@export{#3}\def\@@export{export}%prepare comparison
\ifx\@export\@@export\export@defs{#2}\fi% export the module
}%
```

`\usemodule` `\usemodule` acts like `\importmodule`, except that it does not re-export the semantic macros in the modules it loads.

```
\newcommand\usemodule[2][]{%
\metasetkeys{importmodule}{#1}%
\update@used@modules{#2}%
\ifx\importmodule@dir\@empty
\@importmodule[\importmodule@load]{#2}{noexport}%
\else\@importmodule[\importmodule@dir/#2]{#2}{noexport}\fi%
\ignorespacesandpars}
```

`\withusedmodules` This variant just imports all the modules in a comma-separated list (usually `\used@modules`)

```
\newcommand\withusedmodules[2]{{\@for\@I:=#1\do{\activate@defs\@I}{#2}}}%
```

`\importOMDocmodule` this is now deprecated.

```
\newrobustcmd\importOMDocmodule[3][]{\PackageError{modules}%
  {The \protect\importOMDocmodule macro is deprecated}
  {use \protect\importmodule instead!}}%
```

`\metalanguage` `\metalanguage` behaves exactly like `\importmodule` for formatting. For LaTeXML, we only add the `type` attribute.

```
\let\metalanguage=\importmodule%
```

4.3 Semantic Macros

\mod@newcommand We first hack the LaTeX kernel macros to obtain a version of the \newcommand macro that does not check for definedness.

```
\let\mod@newcommand=\providerobustcmd%
```

Now we define the optional KeyVal arguments for the \symdef form and the actions that are taken when they are encountered.

conceptdef

```
\srefaddidkey{conceptdef}%
\addmetakey*{conceptdef}{title}%
\addmetakey{conceptdef}{subject}%
\addmetakey*{conceptdef}{display}%
\def\conceptdef@type{Symbol}%
\newrobustcmd\conceptdef[2][]{%
  \metasetkeys{conceptdef}{#1}%
  \ifx\conceptdef@display\st@flow\else{\stDMemph{\conceptdef@type} #2:}\fi%
  \ifx\conceptdef@title\@empty~\else~(\stDMemph{\conceptdef@title})\par\fi%
}%
```

symdef:keys The optional argument local specifies the scope of the function to be defined. If local is not present as an optional argument then \symdef assumes the scope of the function is global and it will include it in the pool of macros of the current module. Otherwise, if local is present then the function will be defined only locally and it will not be added to the current module (i.e. we cannot inherit a local function). Note, the optional key local does not need a value: we write \symdef[local]{somefunction}[0]{some expansion}. The other keys are not used in the LaTeX part.

```
\newif\if@symdeflocal%
\srefaddidkey{symdef}%
\define@key{symdef}{local}[true]{\@symdeflocaltrue}%
\define@key{symdef}{noverb}[all]{}%
\define@key{symdef}{align}[WithTheSymbolOfTheSameName]{}%
\define@key{symdef}{specializes}{}%
\addmetakey*{symdef}{noalign}[true]
\define@key{symdef}{primary}[true]{}%
\define@key{symdef}{assocarg}{}%
\define@key{symdef}{bvars}{}%
\define@key{symdef}{bargs}{}%
\addmetakey{symdef}{ns}%
\addmetakey{symdef}{name}%
\addmetakey*{symdef}{title}%
\addmetakey*{symdef}{description}%
\addmetakey{symdef}{subject}%
\addmetakey*{symdef}{display}%
```

EdN:6 [6]

\symdef The the \symdef, and \@symdef macros just handle optional arguments.

```
\def\symdef{\@ifnextchar[{\@symdef}{\@symdef[]}}%
\def\@symdef[#1]#2{\@ifnextchar[{\@@symdef[#1]{#2}}{\@@symdef[#1]{#2}[0]}}%
```

next we locally abbreviate \mod@newcommand to simplify argument passing.

```
\def\@mod@nc#1{\mod@newcommand{#1}[1]}%
```

and we copy a very useful piece of code from http://tex.stackexchange.com/questions/23100/looking-for-an-ignorespacesandpars, it ignores spaces and following implicit paragraphs (double newlines), explicit \pars are respected however

```
\def\ignorespacesandpars{\begingroup\catcode13=10\@ifnextchar\relax{\endgroup}{\endgroup}}
```

and more adapted from http://tex.stackexchange.com/questions/179016/ignore-spaces-and-pars-after-an-environment

```
\def\ignorespacesandparsafterend#1\ignorespaces\fi{#1\fi\ignorespacesandpars}
\def\ignorespacesandpars{\ifhmode\unskip\fi\@ifnextchar\par{\expandafter\ignorespacesandpars\@g
```

\@@symdef now comes the real meat: the \@@symdef macro does two things, it adds the macro definition to the macro definition pool of the current module and also provides it.

```
\def\@@symdef[#1]#2[#3]#4{%
```

We use a switch to keep track of the local optional argument. We initialize the switch to false and set all the keys that have been provided as arguments: name, local.

```
  \@symdeflocalfalse%
  \metasetkeys{symdef}{#1}%
  \usemodule@maybesetcodes%
```

First, using \mod@newcommand we initialize the intermediate macro \module@⟨*sym*⟩@pres@, the one that can be extended with \symvariant

```
  \expandafter\mod@newcommand\csname modules@#2@pres@\endcsname[#3]{#4}%
```

and then we define the actual semantic macro, which when invoked with an optional argument ⟨*opt*⟩ calls \modules@⟨*sym*⟩@pres@⟨*opt*⟩ provided by the \symvariant macro.

```
  \expandafter\mod@newcommand\csname #2\endcsname[1][]%
  {\csname modules@#2@pres@##1\endcsname}%
```

Finally, we prepare the internal macro to be used in the \symref call.

```
\expandafter\@mod@nc\csname mod@symref@#2\expandafter\endcsname\expandafter%
{\expandafter\mod@termref\expandafter{\module@uri@uri}{#2}{##1}}%
```

We check if the switch for the local scope is set: if it is we are done, since this function has a local scope. Similarly, if we are not inside a module, which we could export from.

```
  \if@symdeflocal%
  \else%
    \ifcsundef{module@id}{}{%
```

[6] EdNote: MK@MK: we need to document the binder keys above.

Otherwise, we add three functions to the module's pool of defined macros using \g@addto@macro. We first add the definition of the intermediate function \modules@⟨*sym*⟩@pres@.

```
      \expandafter\g@addto@macro@safe\this@module%
      {\expandafter\mod@newcommand\csname modules@#2@pres@\endcsname[#3]{#4}}%
```

Then we add add the definition of \⟨*sym*⟩ which calls the intermediate function and handles the optional argument.

```
      \expandafter\g@addto@macro@safe\this@module%
      {\expandafter\mod@newcommand\csname #2\endcsname[1][]%
      {\csname modules@#2@pres@##1\endcsname}}%
```

We also add \mod@symref@⟨*sym*⟩ macro to the macro pool so that the \symref macro can pick it up.

```
        \expandafter\expandafter\expandafter\g@addto@macro@safe\expandafter\this@module\expanda
        {\expandafter\@mod@nc\csname mod@symref@#2\expandafter\endcsname\expandafter%
        {\expandafter\mod@termref\expandafter{\module@uri@uri}{#2}{##1}}}%
```

Finally, using \g@addto@macro we add the two functions to the qualified version of the module if the qualifiedimports option was set.

```
      \ifmod@qualified%
        \expandafter\g@addto@macro@safe\this@qualified@module%
        {\expandafter\mod@newcommand\csname modules@#2@pres@qualified\endcsname[#3]{#4}}%
        \expandafter\g@addto@macro@safe\this@qualified@module%
        {\expandafter\def\csname#2@qualified\endcsname{\csname modules@#2@pres@qualified\endcsna
      \fi%
    }% mod@qualified
\fi% symdeflocal
```

So now we only need to show the data in the symdef, if the options allow.

```
  \ifmod@show%
    \ifx\symdef@display\st@flow\else{\noindent\stDMemph{\symdef@type} #2:}\fi%
    \ifx\symdef@title\@empty~\else~(\stDMemph{\symdef@title})\par\fi%
  \fi%
  \ignorespacesandpars%
}% mod@show
\def\symdef@type{Symbol}%
\providecommand{\stDMemph}[1]{\textbf{#1}}
```

\symvariant `\symvariant{`⟨*sym*⟩`}[`⟨*args*⟩`]{`⟨*var*⟩`}{`⟨*cseq*⟩`}` just extends the internal macro \modules@⟨*sym*⟩@pres@ defined by `\symdef{`⟨*sym*⟩`}[`⟨*args*⟩`]{...}` with a variant \modules@⟨*sym*⟩@pres@⟨*var*⟩ which expands to ⟨*cseq*⟩. Recall that this is called by the macro \⟨*sym*⟩[⟨*var*⟩] induced by the \symdef.

```
\def\symvariant#1{%
  \@ifnextchar[{\@symvariant{#1}}{\@symvariant{#1}[0]}%
  }%
\def\@symvariant#1[#2]#3#4{%
  \usemodule@maybesetcodes
  \expandafter\mod@newcommand\csname modules@#1@pres@#3\endcsname[#2]{#4}%
```

and if we are in a named module, then we need to export the function `\modules@⟨sym⟩@pres@⟨opt⟩` just as we have done that in `\symdef`.

```
  \ifcsundef{module@id}{}{%
    \expandafter\g@addto@macro\this@module%
    {\expandafter\mod@newcommand\csname modules@#1@pres@#3\endcsname[#2]{#4}}%
  }%
\ignorespacesandpars}%
```

`\resymdef` This is now deprecated.

```
\def\resymdef{%
  \@ifnextchar[{\@resymdef}{\@resymdef[]}%
}%
\def\@resymdef[#1]#2{%
  \@ifnextchar[{\@@resymdef[#1]{#2}}{\@@resymdef[#1]{#2}[0]}%
}%
\def\@@resymdef[#1]#2[#3]#4{%
  \PackageError{modules}%
  {The \protect\resymdef macro is deprecated}{use the \protect\symvariant instead!}%
}%
```

`\abbrdef` The `\abbrdef` macro is a variant of `\symdef` that does the same on the LaTeX level.

```
\let\abbrdef\symdef%
```

4.4 Defining Math Operators

`\DefMathOp` `\DefMathOp[`⟨*key pair*⟩`]{definition}` will take 2 arguments. ⟨*key pair*⟩ should be something like `[name=...]`, for example, `[name=equal]`. Though `\setkeys`, `\defmathop@name` will be set. Further definition will be done by `\symdef`.

```
\define@key{DefMathOp}{name}{%
  \def\defmathop@name{#1}%
}%
\newrobustcmd\DefMathOp[2][]{%
  \setkeys{DefMathOp}{#1}%
  \symdef[#1]{\defmathop@name}{#2}%
}%
```

4.5 Axiomatic Assumptions

`\assdef` We fake it for now, not clear what we should do on the LaTeX side.

```
\newcommand\assdef[2][]{#2}
```

4.6 Semantic Macros for Variables

`\vardef` From the LaTeX point of view `\vardef` is just an abbreviation

```
\let\vardef\abbrdef
```

4.7 Testing Semantic Macros

\symtest Allows to test a \symdef in place, this shuts up when being imported.

```
\addmetakey{symtest}{name}%
\addmetakey{symtest}{variant}%
\newrobustcmd\symtest[3][]{%
  \if@importing%
  \else%
    \metasetkeys{symtest}{#1}%
    \par\noindent \textbf{Symbol}~%
    \ifx\symtest@name\@empty\texttt{#2}\else\texttt{\symtest@name}\fi%
    \ifx\symtest@variant\@empty\else\ (variant \texttt{\symtest@variant})\fi%
    \ with semantic macro %
    \texttt{\textbackslash #2\ifx\symtest@variant\@empty\else[\symtest@variant]\fi}%
    : used e.g. in \ensuremath{#3}%
  \fi%
  \ignorespacesandpars%
}%
```

\abbrtest

```
\addmetakey{abbrtest}{name}%
\newrobustcmd\abbrtest[3][]{%
  \if@importing%
  \else%
    \metasetkeys{abbrtest}{#1}%
    \par\noindent \textbf{Abbreviation}~%
    \ifx\abbrtest@name\@empty\texttt{#2}\else\texttt{\abbrtest@name}\fi%
    : used e.g. in \ensuremath{#3}%
  \fi%
  \ignorespacesandpars}%
```

4.8 Symbol and Concept Names

\termdef

```
\def\mod@true{true}%
\addmetakey[false]{termdef}{local}%
\addmetakey{termdef}{name}%
\newrobustcmd\termdef[3][]{%
  \metasetkeys{termdef}{#1}%
  \expandafter\mod@newcommand\csname#2\endcsname[0]{#3\xspace}%
  \ifx\termdef@local\mod@true%
  \else%
    \ifcsundef{module@id}{}{%
      \expandafter\g@addto@macro\this@module%
      {\expandafter\mod@newcommand\csname#2\endcsname[0]{#3\xspace}}%
    }%
  \fi%
}%
```

\capitalize

```
\def\@capitalize#1{\uppercase{#1}}%
\newrobustcmd\capitalize[1]{\expandafter\@capitalize #1}%
```

\module@component This macro computes the module component identifier for external links on term references. It is initially empty, but can be redefined later (e.g.in the `smultiling` package).

```
\newcommand\mod@component[1]{}
```

\mod@termref `\mod@termref{`⟨*module*⟩`}{`⟨*name*⟩`}{`⟨*nl*⟩`}` determines whether the macro `\module@`⟨*module*⟩`@path` is defined. If it is, we make it the prefix of a URI reference in the local macro `\@uri`, which we compose to the hyper-reference, otherwise we give a warning.[7]

EdN:7

```
\newcommand\mod@termref[3]{\def\@test{#3}%
  \@ifundefined{module@defs@#1}{%
    \protect\G@refundefinedtrue%
    \if@trwarn
      \PackageWarning{modules}{`\protect\termref' with unidentified cd "#1":\MessageBreak
        the cd key must reference an active module}%
    \else
      \PackageError{modules}{`\protect\termref' with unidentified cd "#1"}
      {the cd key must reference an active module}%
    \fi}%
  {\def\@label{sref@#2@#1\mod@component{#1}@target}%
    \@ifundefined{module@#1@path}% local reference
    {\sref@hlink@ifh{\@label}{\ifx\@test\@empty #2\else #3\fi}%
%      \footnote{sTeX mod@termref: local reference to\\ \@label}
    }%
    {\def\@uri{\csname module@#1@path\endcsname\mod@component{#1}.pdf\#\@label}%
      \sref@href@ifh{\@uri}{\ifx\@test\@empty #2\else #3\fi}%
%      \footnote{sTeX mod@termref: external reference to \\\@uri}
}%
  }}%
```

4.9 Loading Modules

4.9.1 Selective Inclusion

The next great goal is to establish the `\requiremodules` macro, which reads an STEX file and processes all the module signature information in them, but does not produce any output. This is a tricky business, as we need to "parse" the modules and treat the module signature macros specially (we refer to this as "**sms mode**", since it is equivalent to what the – now deprecated – `sms` utility did).

In the following we introduce a lot of auxiliary functionality before we can define `\requiremodules`.

\usemodule@allow* The first step is setting up a functionality for registering `\sTeX` macros and environments as part of a module signature.

[7] EdNote: MK: this should be rethought, in particular the local reference does not work!

```
\newif\if@smsmode\@smsmodefalse
\def\usemodule@escapechar@allowed{true}
\def\usemodule@allow#1{
  \expandafter\let\csname usemodule@allowedmacro@#1\endcsname\usemodule@escapechar@allowed
}
\def\usemodule@allowenv#1{
  \expandafter\let\csname usemodule@allowedenv@#1\endcsname\usemodule@escapechar@allowed
}
\def\usemodule@escapechar@beginstring{begin}
\def\usemodule@escapechar@endstring{end}
```

and now we use that to actually register all the STEX functionality as relevant for `sms` mode.

```
\usemodule@allow{symdef}
\usemodule@allow{abbrdef}
\usemodule@allow{importmodule}
\usemodule@allowenv{module}
\usemodule@allow{importmhmodule}
\usemodule@allow{gimport}
\usemodule@allowenv{modsig}
\usemodule@allowenv{mhmodsig}
\usemodule@allowenv{mhmodnl}
\usemodule@allowenv{modnl}
\usemodule@allow{symvariant}
\usemodule@allow{symi}
\usemodule@allow{symii}
\usemodule@allow{symiii}
\usemodule@allow{symiv}
```

To read external modules without producing output, `\requiremodules` redefines the \-character to be an *active* character that, instead of executing a macro, checks whether a macro name has been registered using `\usemodule@allow` before selectively executing the corresponding macro or ignoring it. To produce the relevant code, we therefore define a macro `\@active@slash` that produces a \-character with category code 13 (*active*), as well as `\@open@brace` and `\@close@brace`, which produce open and closing braces with category code 12 (*other*).

```
\catcode`\.=0
.catcode`.\=13
.def.@active@slash{\}
.catcode`.<=1
.catcode`.>=2
.catcode`.{=12
.catcode`.}=12
.def.@open@brace<{>
.def.@close@brace<}>
.catcode`.\=0
\catcode`\.=12
\catcode`\{=1
```

```
\catcode`\}=2
\catcode`\<=12
\catcode`\>=12
```

The next two macros set and reset the category codes before/after `sms` mode.

`\set@usemodule@catcodes`

```
\def\set@usemodule@catcodes{%
    \global\catcode`\\=13%
    \global\catcode`\#=12%
    \global\catcode`\{=12%
    \global\catcode`\}=12%
    \global\catcode`\$=12%$
    \global\catcode`\^=12%
    \global\catcode`\_=12%
    \global\catcode`\&=12%
    \expandafter\let\@active@slash\usemodule@escapechar%
}
```

`\reset@usemodule@catcodes`

```
\def\reset@usemodule@catcodes{%
    \global\catcode`\\=0%
    \global\catcode`\#=6%
    \global\catcode`\{=1%
    \global\catcode`\}=2%
    \global\catcode`\$=3%$
    \global\catcode`\^=7%
    \global\catcode`\_=8%
    \global\catcode`\&=4%
}
```

`\usemodule@maybesetcodes` Before a macro is executed in `sms`-mode, the category codes will be reset to normal, to ensure that all macro arguments are parsed correctly. Consequently, the macros need to set the category codes back to `sms` mode after having read all arguments iff the macro got executed in `sms` mode. `\usemodule@maybesetcodes` takes care of that.

```
\def\usemodule@maybesetcodes{%
  \if@smsmode\set@usemodule@catcodes\fi%
}
```

`\requiremodules` This macro loads the module signatures in a file using the `\requiremodules@smsmode` above. We set the flag `\mod@showfalse` in the local group, so that the macros know now to pollute the result.

```
\newrobustcmd\requiremodules[1]{%
  \mod@showfalse%
  \edef\mod@path{#1}%
  \edef\mod@path{\expandafter\detokenize\expandafter{\mod@path}}%
  \requiremodules@smsmode{#1}%
}%
```

`\requiremodules@smsmode` this reads sTEX modules by setting the category codes for `sms` mode, `\input`ting the required file and wrapping it in a `\vbox` that gets stored away and ignored, in order to not produce any output. It also sets `\hbadness`, `\hfuzz` and friends to values that suppress overfull and underfull hbox messages.

```
\newbox\modules@import@tempbox
\def\requiremodules@smsmode#1{
  \setbox\modules@import@tempbox\vbox{%
    \@smsmodetrue%
    \set@usemodule@catcodes%
    \hbadness=100000\relax%
    \hsize=10pt%
    \hfuzz=10000pt\relax%
    \input{#1.tex}%
    \reset@usemodule@catcodes%
    }%
    \usemodule@maybesetcodes
}
```

`\usemodule@escapechar` This macro gets called whenever a \-character occurs in `sms` mode. It is split into several macros that parse and store characters in `\usemodule@escape@currcs` until a character with category code $\neq 11$ occurs (i.e. the macro name is complete), check whether the macro is allowed in `sms` mode, and then either ignore it or execute it after setting category codes back to normal. Special care needs to be taken to make sure that braces have the right category codes (1 and 2 for open and closing braces, respectively) when delimiting macro arguments.

Entry point:

```

\def\usemodule@escapechar{%
    \def\usemodule@escape@currcs{}%
    \usemodule@escape@parse@nextchar@%
}%
```

The next macro simply reads the next character and checks whether it has category code 11. If so, it stores it in `\usemodule@escape@currcs`. Otherwise, the macro name is complete, it stores the last character in `\usemodule@last@char` and calls `\usemodule@escapechar@checkcs`.

```
\long\def\usemodule@escape@parse@nextchar@#1{%
    \ifcat a#1 %
        \edef\usemodule@escape@currcs{\usemodule@escape@currcs#1}%
        \let\usemodule@do@next\usemodule@escape@parse@nextchar@%
    \else%
      \def\usemodule@last@char{#1}%
      \def\usemodule@do@next{\usemodule@escapechar@checkcs}%
    \fi%
    \usemodule@do@next%
}
```

The next macro checks whether the currently stored macroname is allowed in `sms` mode. There are four cases that need to be considered: `\begin`, `\end`, allowed

macros, and others. In the first two cases, we reinsert `\usemodule@last@char` and continue with `\usemodule@escapechar@checkbeginenv` or `\usemodule@escapechar@checkendenv` respectively, to check whether the environment being openend/closed is allowed in `sms` mode. In both cases, `\usemodule@last@char` is an open brace with category code 12. In the third case, we need to check whether `\usemodule@last@char` is an open brace, in which case we call `\usemodule@converttoproperbraces`, otherwise, we set category codes to normal and execute the macro. In the fourth case, we just reinsert `\usemodule@last@char` and continue.

```
\def\usemodule@escapechar@checkcs{
    \ifx\usemodule@escape@currcs\usemodule@escapechar@beginstring%
        \edef\usemodule@do@next{\noexpand\usemodule@escapechar@checkbeginenv\usemodule@last@char
    \else%
        \ifx\usemodule@escape@currcs\usemodule@escapechar@endstring%
          \edef\usemodule@do@next{\noexpand\usemodule@escapechar@checkendenv\usemodule@last@char
        \else%
            \expandafter\ifx\csname usemodule@allowedmacro@\usemodule@escape@currcs\endcsname%
                \usemodule@escapechar@allowed%
              \ifx\usemodule@last@char\@open@brace%
                \expandafter\let\expandafter\usemodule@do@next@ii\csname\usemodule@escape@currcs
                \edef\usemodule@do@next{\noexpand\usemodule@converttoproperbraces\@open@brace}
              \else%
                \reset@usemodule@catcodes%
                \edef\usemodule@do@next{\expandafter\noexpand\csname\usemodule@escape@currcs\end
              \fi%
            \else\def\usemodule@do@next{\relax\usemodule@last@char}\fi%
        \fi%
    \fi%
    \usemodule@do@next%
}
```

This macro simply takes an argument in braces (with category codes 12), reinserts it with "proper" braces (category codes 1 and 2), sets category codes back to normal and calls `\usemodule@do@next@ii`, which has been `\let` as the macro to be executed.

```
\expandafter\expandafter\expandafter\def%
\expandafter\expandafter\expandafter\usemodule@converttoproperbraces%
\expandafter\@open@brace\expandafter#\expandafter1\@close@brace{%
  \reset@usemodule@catcodes%
  \usemodule@do@next@ii{#1}%
}
```

The next two macros apply in the `\begin` and `\end` cases. They check whether the environment is allowed in `sms` mode, if so, open/close the environment, and otherwise do nothing.

Notably, `\usemodule@escapechar@checkendenv` does not set category codes back to normal, since `\end{environment}` never takes additional arguments that need to be parsed anyway.

```
\expandafter\expandafter\expandafter\def%
```

```
\expandafter\expandafter\expandafter\usemodule@escapechar@checkbeginenv%
\expandafter\@open@brace\expandafter#\expandafter1\@close@brace{%
    \expandafter\ifx\csname usemodule@allowedenv@#1\endcsname\usemodule@escapechar@allowed%
        \reset@usemodule@catcodes%
        \def\usemodule@do@next{\begin{#1}}%
    \else%
        \def\usemodule@do@next{#1}%
    \fi%
    \usemodule@do@next%
}
\expandafter\expandafter\expandafter\def%
\expandafter\expandafter\expandafter\usemodule@escapechar@checkendenv%
\expandafter\@open@brace\expandafter#\expandafter1\@close@brace{%
    \expandafter\ifx\csname usemodule@allowedenv@#1\endcsname\usemodule@escapechar@allowed%
        %\reset@usemodule@catcodes%
        \def\usemodule@do@next{\end{#1}}%
    \else%
      \def\usemodule@do@next{#1}%
    \fi%
    \usemodule@do@next%
}
```

`\@requiremodules` the internal version of `\requiremodules` for use in the `*.aux` file. We disable it at the end of the document, so that when the `aux` file is read again, nothing is loaded.

```
\newrobustcmd\@requiremodules[1]{%
  \if@tempswa\requiremodules{#1}\fi%
}%
```

`\inputref@*skip` hooks for spacing customization, they are empty by default.

```
\def\inputref@preskip{}
\def\inputref@postskip{}
```

`\inputref` `\inputref{`⟨*path to the current file without extension*⟩`}` supports both absolute path and relative path, meanwhile, records the path and the extension (not for relative path).

```
\newrobustcmd\inputref[1]{%
  \def\@Slash{/}
  \edef\@load{#1}%
  \StrChar{\@load}{1}[\@testchar]
  \inputref@preskip%
  \ifx\@testchar\@Slash%
    \edef\mod@path{#1}%
    \edef\mod@path{\expandafter\detokenize\expandafter{\mod@path}}%
    \input{#1}%
  \else%
    \@cpath{#1}\input{\@CanPath.tex}%
  \fi%
  \inputref@postskip%
}%
```

4.10 Including Externally Defined Semantic Macros

\requirepackage

```
\def\requirepackage#1#2{\makeatletter\input{#1.sty}\makeatother}%
```

4.11 Namespaces and Alignments

\namespace

```
\newcommand\namespace[2][]{\ignorespacesandpars}
```

4.12 Deprecated Functionality

\sinput*

```
\newrobustcmd\sinput[1]{%
  \PackageError{modules}%
  {The `\protect\sinput' macro is deprecated}{use the \protect\input instead!}%
}%
\newrobustcmd\sinputref[1]{%
  \PackageError{modules}%
  {The \protect\sinputref macro is deprecated}{use the \protect\inputref instead!}%
}%
```

In this section we centralize old interfaces that are only partially supported any more.

module:uses

For each the module name xxx specified in the uses key, we activate their symdefs and we export the local symdefs.[8]

EdN:8

```
\define@key{module}{uses}{\PackageError{modules}%
  {The 'uses' key on {module} macro is deprecated}{}}
```

module:usesqualified

This option operates similarly to the module:uses option defined above. The only difference is that here we import modules with a prefix. This is useful when two modules provide a macro with the same name.

```
\define@key{module}{usesqualified}{\PackageError{modules}%
  {The 'usesqualified' key on {module} macro is deprecated}{}}
```

\coolurion/off

```
\def\coolurion{\PackageWarning{modules}{coolurion is obsolete, please remove}}%
\def\coolurioff{\PackageWarning{modules}{coolurioff is obsolete, please remove}}%
```

4.13 Experiments

In this section we develop experimental functionality. Currently support for complex expressions, see `https://svn.kwarc.info/repos/stex/doc/blue/comlex_semmacros/note.pdf` for details.

[8] EdNote: this issue is deprecated, it will be removed before 1.0.

\csymdef For the LaTeX we use \symdef and forget the last argument. The code here is just needed for parsing the (non-standard) argument structure.

```
\def\csymdef{\@ifnextchar[{\@csymdef}{\@csymdef[]}}%
\def\@csymdef[#1]#2{%
  \@ifnextchar[{\@@csymdef[#1]{#2}}{\@@csymdef[#1]{#2}[0]}%
}%
\def\@@csymdef[#1]#2[#3]#4#5{%
  \@@symdef[#1]{#2}[#3]{#4}%
}%
```

\notationdef For the LaTeX side, we just make \notationdef invisible.

```
\def\notationdef[#1]#2#3{}
```

The code for avoiding duplicate loading is very very complex and brittle (and does not quite work). Therefore I would like to replace it with something better. It has two parts:

- keeping a registry of file paths, and only loading when the file path has not been mentioned in that, and
- dealing with relative paths (for that we have to string together prefixes and pass them one)

For the first problem, there is a very nice and efficient solution using etoolbox which I document below. If I decide to do away with relative paths, this would be it.

\reqmodules We keep a file path registry \@register and only load a module, if it is not in there.

```
\newrobustcmd\reqmodules[2]{%
  \ifinlist{#1}{\@register}{}{\listadd\@register{#1}\input{#1.#2}}%
}%
```

for the relative paths, I have to find out the directory prefix and the file name. Here are two helper functions, which work well, but do not survive being called in an \edef, which is what we would need. First some preparation: we set up a path parser

```
\newcounter{@pl}
\DeclareListParser*{\forpathlist}{/}
```

\file@name \file@name selects the filename of the file path: \file@name{/foo/bar/baz.tex} is baz.tex.

```
\def\file@name#1{%
  \setcounter{@pl}{0}%
  \forpathlist{\stepcounter{@pl}\listadd\@pathlist}{#1}
  \def\do##1{%
    \ifnumequal{\value{@pl}}{1}{##1}{\addtocounter{@pl}{-1}}
  }%
  \dolistloop{\@pathlist}%
}%
```

\file@path `\file@path` selects the path of the file path `\file@path{/foo/bar/baz.tex}` is `/foo/bar`

```
\def\file@path#1{%
  \setcounter{@pl}{0}%
  \forpathlist{\stepcounter{@pl}\listadd\@pathlist}{#1}%
  \def\do##1{%
    \ifnumequal{\value{@pl}}{1}{}{%
      \addtocounter{@pl}{-1}%
      \ifnumequal{\value{@pl}}{1}{##1}{##1/}%
    }%
  }%
  \dolistloop{\@pathlist}%
}%
```

⟨/package⟩

what I would really like to do in this situation is

\NEWrequiremodules but this does not work, since the `\file@name` and `\file@path` do not survive the `\edef`.

```
\def\@NEWcurrentprefix{}
\def\NEWrequiremodules#1{%
  \def\@pref{\file@path{#1}}%
  \ifx\@pref\@empty%
  \else%
    \xdef\@NEWcurrentprefix{\@NEWcurrentprefix/\@pref}%
  \fi%
  \edef\@input@me{\@NEWcurrentprefix/\file@name{#1}}%
  \message{requiring \@input@me}\reqmodule{\@input@me}%
}%
```

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in roman refer to the code lines where the entry is used.

LaTeXML, 3, 10, 11, 14, 19
MathML, 3, 4, 6, 7
module
  name, 8
name
  module, 8
OMDoc, 3, 7, 11
OpenMath, 3, 4, 6, 7
relation
  inheritance (semantic), 8
semantic
  inheritance relation, 8
Xml, 7

Change History

v0.9
General: First Version with Documentation 1
v0.9a
General: Completed Documentation 1
v0.9b
General: Complete functionality and Updated Documentation . . 1
v0.9c
General: more packaging 1
v0.9d
General: fixing double loading of .tex and .sms 1
v0.9e
General: fixing LaTeXML 1
v0.9f
General: remove unused options uses and usesqualified 1
v0.9g
General: adding importOMDocmodule 1
adding resymdef functionality . . 1
v0.9h
General: adding `\metalanguage` . . 1
using `\mod@newcommand` instead of `\providecommand` for more intuitive inheritance. 1
v1.0
General: minor fixes 1
v1.1
General: adding additional keys for the `\symdef` macro and exporting them to OMDoc . . . 1
adding optional arguments to semantic macros for display variants. The resymdef functionality introduced in 0.9g is now deprecated. It was hardly used. 1
exporting requiremodules to the `aux` file, so that they are preloaded (pre-required) so semantic macros in section titles can work. 1
Moving LaTeXML bindings into `modules.sty.ltxml` and disabling generation 1
v1.2
General: No longer loading the aux file at the end of the document 1
v1.3
General: adding MathHub support 1
v1.4
General: Completely revamped importing modules this is much faster now, but can no longer do relative paths. 1
deprecated `\sinput` and `\sinputref` 1
v1.5
General: “unidentified cd” in termref is now an error. 1
adding dir attribute to import/usemodule 1
Moved MH Versions to a separate `mathhub` package 1
v1.6
General: deprecating importOMDocmodule 1
getting rid of `sms` files 1

References

[Aus+10] Ron Ausbrooks et al. *Mathematical Markup Language (MathML) Version 3.0*. Tech. rep. World Wide Web Consortium (W3C), 2010. URL: http://www.w3.org/TR/MathML3.

[Bus+04] Stephen Buswell et al. *The Open Math Standard, Version 2.0*. Tech. rep. The OpenMath Society, 2004. URL: http://www.openmath.org/standard/om20.

[Koh06] Michael Kohlhase. *OMDoc – An open markup format for mathematical documents [Version 1.2]*. LNAI 4180. Springer Verlag, Aug. 2006. URL: http://omdoc.org/pubs/omdoc1.2.pdf.

[Koh08] Michael Kohlhase. "Using LaTeX as a Semantic Markup Format". In: *Mathematics in Computer Science* 2.2 (2008), pp. 279–304. URL: https://kwarc.info/kohlhase/papers/mcs08-stex.pdf.

[Koh20a] Michael Kohlhase. *MathHub Support for sTeX*. Tech. rep. 2020. URL: https://github.com/sLaTeX/sTeX/raw/master/sty/mathhub/mathhub.pdf.

[Koh20b] Michael Kohlhase. *metakeys.sty: A generic framework for extensible Metadata in LaTeX*. Tech. rep. 2020. URL: https://github.com/sLaTeX/sTeX/raw/master/sty/metakeys/metakeys.pdf.

[Koh20c] Michael Kohlhase. *statements.sty: Structural Markup for Mathematical Statements*. Tech. rep. 2020. URL: https://github.com/sLaTeX/sTeX/raw/master/sty/statements/statements.pdf.

[LTX] Bruce Miller. *LaTeXML: A LaTeX to XML Converter*. URL: http://dlmf.nist.gov/LaTeXML/ (visited on 03/12/2013).

[MNS] *Documents and Namespaces*. URL: https://uniformal.github.io/doc/language/namespaces (visited on 11/23/2019).

[RK13] Florian Rabe and Michael Kohlhase. "A Scalable Module System". In: *Information & Computation* 0.230 (2013), pp. 1–54. URL: https://kwarc.info/frabe/Research/mmt.pdf.

[RO] Sebastian Rahtz and Heiko Oberdiek. *Hypertext marks in LATEX: a manual for hyperref*. URL: http://tug.org/applications/hyperref/ftp/doc/manual.pdf (visited on 01/28/2010).

[Sch] Martin Scharrer. *standalone – Compile TeX pictures stand-alone or as part of a document*. URL: https://ctan.org/pkg/standalone (visited on 10/16/2020).

[sTeX] *sTeX: A semantic Extension of TeX/LaTeX*. URL: https://github.com/sLaTeX/sTeX (visited on 05/11/2020).