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

Michael Kohlhase & Deyan Ginev & Rares Ambrus
Jacobs University, Bremen
http://kwarc.info/kohlhase

April 2, 2015

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 v1.3 (last revised 2013/12/12)

Contents

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 2.6).

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` The `modules` package takes two options: If we set `showmods`[1], then the views (see
EdN:1

[1] EdNote: This mechanism does not work yet, since we cannot disable it when importing modules and that leads to unwanted boxes. What we need to do instead is to tweak the sms utility to use an

qualifiedimports Section 2.10) 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 3.4.

noauxreq The option `noauxreq` prohibits the registration of `\@requiremodules` commands in the `aux` file. They are necessary for preloading the module signatures 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.

showmeta If the `showmeta` is set, then the metadata keys are shown (see [Koh15a] for details and customization options).

2.2 Semantic Macros

\symdef The is the main constructor for semantic macros in sTeX. 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 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.

internal version that never shows anything during sms reading.

`\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.

```
\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

`\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).

```
\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 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`.

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".

Semantic variables differ from semantic constants in two ways: *i*) they do not participate in the imports mechanism and *ii*) 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 `\vardef` `\op` for the (semigroup) operation via the `\vardef` macro. `\vardef` works exactly like, except *i*) semantic variables are local to the current TeX group and *ii*) 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>⎄</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

2.5 Symbol and Concept Names

`\termdef` `\capitalize` EdN:2 `\termref` `\symref`

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 beginning.[2]. 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 [Koh15b]. Together with the `hyperref` package [RO], this provide cross-referencing to the definitions of the symbols and concepts. As discussed in section 3.6, 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.

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

2.6 Modules and Inheritance

module Themodule 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 9 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 STEX 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. To enable situations like these, the `module` package uses auxiliary files called **STEX module signatures**. For any file, ⟨*file*⟩`.tex`, we generate a corresponding STEX module signature ⟨*file*⟩`.sms` with the `sms` utility (see also Limitation 3.1), which contains (copies of) all `\begin/\end{module}`, `\importmodule`, `\symdef`, and `\termdef` invocations in ⟨*file*⟩`.tex`. The value of an STEX module signature is that it can be loaded instead its corresponding STEX document, if we are only interested in the semantic macros. So `\importmodule[`load=⟨*filepath*⟩`]{`⟨*mod*⟩`}` will load the STEX module signature ⟨*filepath*⟩`.sms` (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 since ⟨*filepath*⟩`.sms` contains all `\importmodule` statements that ⟨*filepath*⟩`.tex` does, an `\importmodule` recursively loads all necessary
EdN:3 files to supply the semantic macros inherited by the current module.[3]

importmodulevia The `\importmodule` macro has a variant `\importmodulevia` that allows the specification of a theory morphism to be applied. `\importmodulevia{`⟨*thyid*⟩`}{`⟨*assignments*⟩`}` specifies the "source theory" via its identifier ⟨*thyid*⟩ and the morphism by ⟨*assignments*⟩. There are three kinds:

\vassign **symbol assignments** via `\vassign{`⟨*sym*⟩`}{`⟨*exp*⟩`}`, which defines the symbol ⟨*sym*⟩ introduced in the current theory by an expression ⟨*exp*⟩ in the source theory.

\tassign **term assignments** via `\tassign[`⟨*source-cd*⟩`]{`⟨*tname*⟩`}{`⟨*source-tname*⟩`}`, which

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

[3]EDNOTE: MK: document the other keys of `module`

assigns to the term with name ⟨*tname*⟩ in the current theory a term with name ⟨*source-tname*⟩ in the theory ⟨*source-cd*⟩ whose default value is the source theory.

`\ttassign` **term text assignments** via `\tassign{`⟨*tname*⟩`}{`⟨*text*⟩`}`, which defines a term with name ⟨*tname*⟩ in the current theory via a definitional text.

```
\begin{module}[id=ring]
\begin{importmodulevia}{monoid}
  \vassign{rbase}\magbase
  \vassign{rtimesOp}\magmaop
  \vassign{rone}\monunit
\end{importmodulevia}
\symdef{rbase}{G}
\symdef[name=rtimes]{rtimesOp}{\cdot}
\symdef{rtimes}[2]{\infix\rtimesOp{#1}{#2}}
\symdef{rone}{1}
\begin{importmodulevia}{cgroup}
  \vassign{rplus}\magmaop
  \vassign{rzero}\monunit
  \vassign{rinvOp}\cginvOp
\end{importmodulevia}
\symdef[name=rplus]{rplusOp}{+}
\symdef{rplus}[2]{\infix\rplusOp{#1}{#2}}
\symdef[name=rminus]{rminusOp}{-}
\symdef{rminus}[1]{\infix\rminusOp{#1}{#2}}
...
\end{module}
```

Example 8: A Module for Rings with inheritance from monoids and commutative groups

`\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 [**RabKoh:WSMSML10**] for discussion.

2.7 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. One way to do this would be to have LaTeX read the prerequisite documents without producing output. For efficiency reasons, sTeX chooses a different route. It comes with a utility `sms` (see Section 3.1) that exports the modules and macros defined inside them from a particular document and stores them inside `.sms` files. This way we can avoid overloading LaTeX with useless information, while retaining the important information which can then be imported in a more efficient way.

`\importmodule` For such situations, the `\importmodule` macro can be given an optional first argument that is a path to a file that contains a path to the module file, whose module definition (the `.sms` file) is read. Note that the `\importmodule` macro 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.

```
\begin{module}[id=foo]
\importmodule[load=../other/bar]{bar}
\importmodule[load=../mycolleaguesmodules]{baz}
\importmodule[load=../other/bar]{foobar}
  ...
\end{module}
```

Example 9: Self-contained Modules via `importmodule`

In Example 9, 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.

Sometimes we want to import an existing OMDoc theory[2] $\widehat{\mathcal{T}}$ into (the OMDoc document $\widehat{\mathcal{D}}$ generated from) a sTeX document $\mathcal{D}$. Naturally, we have to provide an sTeX stub module $\mathcal{T}$ that provides `\symdef` declarations for all symbols we

`\importOMDocmodule` use in $\mathcal{D}$. In this situation, we use`\importOMDocmodule[`⟨*spath*⟩`]{`⟨*OURI*⟩`}{`⟨*name*⟩`}`, where ⟨*spath*⟩ is the file system path to $\mathcal{T}$ (as in `\importmodule`, this argument must not contain the file extension), ⟨*OURI*⟩ is the URI to the OMDoc module (this time with extension), and ⟨*name*⟩ is the name of the theory $\widehat{\mathcal{T}}$ and the module in $\mathcal{T}$ (they have to be identical for this to work). Note that since the ⟨*spath*⟩ argument is optional, we can make "local imports", where the stub $\mathcal{T}$ is in $\mathcal{D}$ and only contains the `\symdef`s needed there.

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

[2]OMDoc theories are the counterpart of sTeX modules.

the TEX formatter (it is usually set to something like 15[3]). Therefore, it may be necessary to circumvent the recursive load pattern providing (logically spurious) `\importmodule` commands. Consider for instance module `bar` in Example 9, 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.sms` 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`

`\sinput`

In all of the above, we do not want to load an `sms` file, if the corresponding file has already been loaded, since the semantic macros are already in memory. Therefore the `modules` package supplies a semantic variant of the `\input` macro, which records in an internal register that the modules in the file have already been loaded. Thus if we consistently use `\sinput` instead of `\input` or `\include` for files that contain modules[4], we can prevent double loading of files and therefore gain efficiency. The `\sinputref` macro behaves just like `\sinput` in the LATEX workflow, but in the LATEXML conversion process creates a reference to the transformed version of the input file instead. `\inputref` and `\sincluderef` also only create references in the LATEXML workflow but do not register loaded files or avoid duplicate loading.

`\sinputref` `\inputref` `\includeref`

Finally, the separation of documents into multiple modules often profits from a symbolic management of file paths. To simplify this, the `modules` package supplies the `\defpath` macro: `\defpath[`⟨*baseURI*⟩`]{`⟨*cname*⟩`}{`⟨*path*⟩`}` defines a command, so that `\`⟨*csname*⟩`{`⟨*name*⟩`}` expands to ⟨*path*⟩`/`⟨*name*⟩. So we could have used

`\defpath`

```
\defpath{OPaths}{../other}
\importmodule[load=\OPahts{bar}]{bar}
```

instead of the second line in Example 9. The variant `\OPaths` has the big advantage that we can get around the fact that TEX/LATEX does not set the current directory in `\input`, so that we can use systematically deployed `\defpath`-defined path macros to make modules relocatable by defining the path macros locally. The optional parameter ⟨*baseURI*⟩ is for the LATEXML transformation, which (if ⟨*baseURI*⟩ is specified) resolves ⟨*path*⟩ to an absolute URI according to [BFM05, section 5.2].

2.8 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

[3]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

[4]files without modules should be treated by the regular LATEX input mechanism, since they do not need to be registered.

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 situation for most parts of mathematical documents. For that sTeX provides the

\usemodule `\usemodule` macro, which takes the same arguments as `\importmodule`, but is treated differently in the sTeX module signatures. A typical situation is shown in Figure 10, where we open the module `ring` (see Figure 8) 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[../algebra/rings.tex]{ring}
  We $R$ be a ring $(\rbase,\rplus,\rzero,\rminusOp,\rtimes,\rone)$, ...
\end{omtext}
```

Example 10: Using Semantic Macros in Narrative Structures

\adoptmodule Still another import-like relation is the adoption of a module (see [Koh13] for details). We use the `\adoptmodule` macro for that.[4]

EdN:4

2.9 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.

2.10 Views

A view is a mapping between modules, such that all model assumptions (axioms) of the source module are satisfied in the target module. [5]

EdN:5

[4] EdNote: elaborate this, so that the documentation becomes (more) self-contained.
[5] EdNote: Document and make Examples

2.11 Support for MathHub

Much of the sTeX content is hosted on MathHub (`http://MathHub.info`), a portal and archive for flexiformal mathematics. MathHub offers GIT repositories (public and private escrow) for mathematical documentation projects, online and offline authoring and document development infrastructure, and a rich, interactive reading interface. The `modules` package supports repository-sensitive operations on MathHub.

Note that MathHub has two-level repository names of the form ⟨*group*⟩/⟨*repo*⟩, where ⟨*group*⟩ is a MathHub-unique repository group and ⟨*repo*⟩ a repository name that is ⟨*group*⟩-unique. The file and directory structure of a repository is arbitrary – except that it starts with the directory `source` because they are Math Archives in the sense of [Hor+11]. But this structure can be hidden from the sTeX author with MathHub-enabled versions of the `modules` macros.

`\importmhmodule` The `importmhmodule` macro is a variant of `\importmodule` with repository support. Instead of writing

```
\defpath{MathHub}{/user/foo/lmh/MathHub}
\importmodule[load=\MathHub{fooMH/bar/source/baz/foobar}]{foobar}
```

we can simply write (assuming that `\MathHub` is defined as above)

```
\importmhmodule[repos=fooMH/bar,path=baz/foobar]{foobar}
```

Note that the `\importmhmodule` form is more semantic, which allows more advanced document management features in MathHub.

If `baz/foobar` is the "current module", i.e. if we are on the MathHub path ...`MathHub/fooMH/bar`..., then stating the repository in the first optional argument is redundant, so we can just use

```
\importmhmodule[path=baz/foobar]{foobar}
```

if no file needs to loaded, `\importmhmodule` is the same as `\importmodule`.

Of course, neither LaTeX nor LaTeXML know about the repositories when `\mhcurrentrepos` they are called from a file system, so we can use the `\mhcurrentrepos` macro to tell them. But this is only needed to initialize the infrastructure in the driver file. In particular, we do not need to set it in in each module, since the `\importmhmodule` macro sets the current repository automatically.

`\usemhmodule` `\adoptmhmodule` The `\usemhmodule` and `\adoptmhmodule` macros are the analogs to `\usemodule` and `\adoptmodule`.

Caveat if you want to use the MathHub support macros (let's call them mh-variants), then every time a module is imported or a document fragment is included from another repos, the mh-variant `\importmhmodule` must be used, so that the "current repository" is set accordingly. To be exact, we only need to use mh-variants, if the imported module or included document fragment use mh-variants.

\mhinputref
\mhinput

For this, the `modules` package supplies the mh-variants `\mhinputref` and `\mhinput` of the `\inputref` macro introduced above and normal LaTeX `\input` macro.

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 TRAC [sTeX].

3.1 Perl Utility sms

Currently we have to use an external perl utility `sms` to extract sTeX module signatures from sTeX files. This considerably adds to the complexity of the sTeX installation and workflow. If we can solve security setting problems that allows us to write to sTeX module signatures outside the current directory, writing them from sTeX may be an avenue of future development see [sTeX, issue #1522] for a discussion.

3.2 Module Signatures loaded even if Modules are

Currently, the module signature ⟨*filepath*⟩`.sms` is loaded even if ⟨*filepath*⟩`.tex` has already been loaded. The Problem is that the `\input` `\inputref` macros do not register the files. I guess, for `\input` we may not want that, but for `\inputref` we should; and maybe we should also have a registering variant `\rinput` that does, then we can leave the choice to the user.

3.3 Only one level of path simplification

It seems that module simplification only covers one level, in particular `foo/bar/../..` is only simplified to `foo/..`, which is not enough in practice. As a consequence, module signatures are loaded unnecessary in the presence of relative paths.

3.4 Qualified Imports

qualifiedimports

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.

3.5 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.

3.6 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 [Koh15b]. A way around this limitation would be to import the current module from the STEX module signature (see Section 2.6) via the `\importmodule` declaration.

3.7 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 `\sinput{mod}`, as in

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

then the `\importmodule` command can read `mod.sms` (created via the `sms` utility) 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).

```
⟨*package⟩
\DeclareOption{showmeta}{\PassOptionsToPackage{\CurrentOption}{metakeys}}
\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}
```

Finally, we need to declare the end of the option declaration section to LaTeX.

```
\ProcessOptions
```

LaTeXML does not support module options yet, so we do not have to do anything here for the LaTeXML bindings. We only set up the Perl packages (and tell `emacs` about the appropriate mode for convenience

The next measure is to ensure that the `sref` and `xcomment` packages are loaded (in the right version). For LaTeXML, we also initialize the package inclusions.

```
\RequirePackage{sref}
\RequirePackage{xspace}
\RequirePackage{mdframed}
\RequirePackage{xstring}
```

4.2 Modules and Inheritance

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

`module:cd` This `KeyVal` key is only needed for LaTeXML at the moment; use this to specify a content dictionary name that is different from the module name.

```
\addmetakey{module}{cd}% no longer used
\addmetakey{module}{load}% ignored
\addmetakey*{module}{title}
\addmetakey*{module}{creators}
\addmetakey*{module}{contributors}
```

`module:id` For a module with `[id=`⟨*name*⟩`]`, we have a macro `\module@defs@`⟨*name*⟩ 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@`⟨*name*⟩; we define it first and then initialize `\module@defs@`⟨*name*⟩ 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 `\mod@id` and the module path in `\`⟨*name*⟩`@cd@file@base` which we add to `\module@defs@`⟨*name*⟩, so that we can use it in the importing module.

```
\define@key{module}{id}{%
\edef\this@module{\expandafter\noexpand\csname module@defs@#1\endcsname}%
\global\@namedef{module@defs@#1}{}%
\ifmod@qualified
\edef\this@qualified@module{\expandafter\noexpand\csname module@defs@qualified@#1\endcsname}%
\global\@namedef{module@defs@qualified@#1}{}%
\fi
\def\mod@id{#1}}
```

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

```
\newcounter{module}[section]
\newcommand\module@heading{\stepcounter{module}%
\ifmod@show%
\noindent{\textbf{Module} \thesection.\themodule [\mod@id]}%
\sref@label@id{Module \thesection.\themodule [\mod@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. All the work has already been done in the keyval bindings, so this is very simple.

```
\newenvironment{module}[1][]{\begin{@module}[#1]%
\@ifundefined{mod@id}{}% only define if components are!
{\@ifundefined{mod@path}{}{\csxdef{\mod@id @cd@file@base}{\mod@path}}}%
\module@heading}
{\end{@module}}
\ifmod@show\surroundwithmdframed{module}\fi
```

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

```
\newenvironment{@module}[1][]{\metasetkeys{module}{#1}}{}
```

`\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{%
\@ifundefined{module@#1@activated}{\csname module@defs@#1\endcsname}{}%
\@namedef{module@#1@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*⟩.[6]

EdN:6

```
\def\export@defs#1{\@ifundefined{mod@id}{}%
{\expandafter\expandafter\expandafter\g@addto@macro\expandafter%
\this@module\expandafter{\activate@defs{#1}}}}
```

Now we come to the implementation of `\importmodule`, but before we do, we define two conditionals:

`\if@export` `\if@export` distinguishes the `\importmodule` and `\usemodule` macros defined below, they differ in whether they re-export macros they import.

```
\newif\if@export
```

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

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

`\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 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 `\imported@modules` register and an internal one (`\@importmodule`) that does the actual work.

```
\gdef\imported@modules{}
\srefaddidkey{importmodule}
\addmetakey{importmodule}{load}
\addmetakey[sms]{importmodule}{ext}
\newcommand\importmodule[2][]{\metasetkeys{importmodule}{#1}%
\ifx\imported@modules\@empty\edef\imported@modules{#2}%
\else\edef\imported@modules{#2,\imported@modules}\fi%
\@exporttrue\@importmodule[\importmodule@load]{#2}{\importmodule@ext}\ignorespaces}
```

`\@importmodule` `\@importmodule[`⟨*file*⟩`]{`⟨*mod*⟩`}{`⟨*ext*⟩`}` loads ⟨*file*⟩`.`⟨*ext*⟩ (if it is given) and activates the module ⟨*mod*⟩. It also remembers the file name and extension in `\mod@path` and `\mod@ext` so that `\begin{module}` can pick them up later. There is a slight inefficiency in that modules can be recorded multiply, but since `\activate@defs` is efficient about not activating twice, this is OK.

```
\newcommand\@importmodule[3][]{%
{\@importingtrue% to shut up macros while in the group opened here
\edef\@load{#1}\ifx\@load\@empty\else%
\@ifundefined{#2@cd@file@base}{\requiremodules{#1}{#3}}%
{\edef\@path{\expandafter\csname #2@cd@file@base\endcsname}
\IfStrEq\@load\@path% if the known path is the same as the requested one
{}% do nothing, it has already been loaded, else signal an error
{\PackageError{modules}
{Module Name Clash\MessageBreak A module with name #2 was already loaded under the path
```

[6] EdNote: MK: I have the feeling that we may be exporting modules multiple times here, is that a problem?

```
  "\@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}% \@load empty and end importing group
\activate@defs{#2}\if@export\export@defs{#2}\fi}
```

`\usemodule` \usemodule acts like \importmodule for the LaTeX side, except that the sms utility does not transfer it to the module signatures and it does not export the symdefs.

```
\newcommand\usemodule[2][]{\@exportfalse\importmodule[#1]{#2}}
```

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

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

`\importmodulevia` The importmodulevia environment just calls \importmodule, but to get around the group, we first define a local macro \@@doit, which does that and can be called with an \aftergroup to escape the environment grouping introduced by importmodulevia.

```
\newenvironment{importmodulevia}[2][]{\gdef\@@doit{\importmodule[#1]{#2}}%
\ifmod@show\par\noindent importing module #2 via \@@doit\fi}
{\aftergroup\@@doit\ifmod@show end import\fi}
```

`vassign`

```
\newcommand\vassign[3][]{\ifmod@show\ensuremath{#2\mapsto #3}, \fi}
```

`tassign`

```
\newcommand\tassign[3][]{\ifmod@show #2\ensuremath{\mapsto} #3, \fi}
```

`ttassign`

```
\newcommand\ttassign[3][]{\ifmod@show #2\ensuremath{\mapsto} ‘‘#3’’, \fi}
```

`\importOMDocmodule` for the LaTeX side we can just re-use \importmodule, for the LaTeXML side we have a full URI anyways. So things are easy.[7]

EdN:7

```
\newcommand\importOMDocmodule[3][]{\importmodule[#1]{#3}}
```

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

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

`\adoptmodule` \adoptmodule macro behaves exactly like \importmodule for formatting. For LaTeXML.

```
\let\adoptmodule=\importmodule
```

[7] EdNote: MK@DG: this macro is seldom used, maybe I should just switch arguments.

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}
\newcommand\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}
```

EdN:8 [8]

`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}{primary}[true]{}
\define@key{symdef}{assocarg}{}
\define@key{symdef}{bvars}{}
\define@key{symdef}{bargs}{}
\addmetakey{symdef}{name}
\addmetakey*{symdef}{title}
\addmetakey*{symdef}{description}
\addmetakey{symdef}{subject}
\addmetakey*{symdef}{display}
```

EdN:9 [9]

`\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]}}
```

[8] EdNote: MK@DG: maybe we need to add DefKeyVals here?

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

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

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

`\@@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}%
```

First, using `\mod@newcommand` we initialize the intermediate macro `\module@`$\langle sym\rangle$`@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 $\langle opt\rangle$ calls `\modules@`$\langle sym\rangle$`@pres@`$\langle opt\rangle$ 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{\mod@id}{#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%
\@ifundefined{mod@id}{}{%
```

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@`$\langle sym\rangle$`@pres@`.

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

Then we add add the definition of `\`$\langle sym\rangle$ which calls the intermediate function and handles the optional argument.

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

We also add `\mod@symref@`$\langle sym\rangle$ macro to the macro pool so that the `\symref` macro can pick it up.

```
\expandafter\g@addto@macro\csname  module@defs@\mod@id\expandafter\endcsname\expandafter%
{\expandafter\@mod@nc\csname mod@symref@#2\expandafter\endcsname\expandafter%
{\expandafter\mod@termref\expandafter{\mod@id}{#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\this@qualified@module%
{\expandafter\mod@newcommand\csname modules@#2@pres@qualified\endcsname[#3]{#4}}%
\expandafter\g@addto@macro\this@qualified@module%
{\expandafter\def\csname#2atqualified\endcsname{\csname modules@#2@pres@qualified\endcsname}}%
\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}% mod@show
\def\symdef@type{Symbol}
```

`\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`.[10]

EdN:10

```
\def\symvariant#1{\@ifnextchar[{\@symvariant{#1}}{\@symvariant{#1}[0]}}
\def\@symvariant#1[#2]#3#4{%
\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`.

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

`\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,\MessageBreak
    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

EdN:11 `\DefMathOp` [11]

```
\define@key{DefMathOp}{name}{\def\defmathop@name{#1}}
```

[10] EdNote: MK@DG: this needs to be implemented in LaTeXML

[11] EdNote: MK@MK,DG: DefMathOp needs to be documented above, what can we do with it?

```
\newcommand\DefMathOp[2][]{%
\setkeys{DefMathOp}{#1}%
\symdef[#1]{\defmathop@name}{#2}}
```

4.5 Semantic Macros for Variables

`\vardef` We do the argument parsing like in `\symdef` above, but add the `local` key. All the other changes are in the LaTeXML binding exclusively.

```
\def\vardef{\@ifnextchar[{\@vardef}{\@vardef[]}}
\def\@vardef[#1]#2{\@ifnextchar[{\@@vardef[#1]{#2}}{\@@vardef[#1]{#2}[0]}}
\def\@@vardef[#1]#2[#3]#4{\def\@test{#1}%
\ifx\@test\@empty\@@symdef[local]{#2}[#3]{#4}\else\symdef[local,#1]{#2}[#3]{#4}\fi}
```

4.6 Testing Semantic Macros

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

```
\addmetakey{symtest}{name}
\addmetakey{symtest}{variant}
\newcommand\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}
```

`\abbrtest`

```
\addmetakey{abbrtest}{name}
\newcommand\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}
```

4.7 Symbol and Concept Names

`\mod@path` the `\mod@path` macro is used to remember the local path, so that the `module` environment can set it for later cross-referencing of the modules. If `\mod@path` is empty, then it signifies the local file.

```
\def\mod@path{}
```

`\termdef`

```
\def\mod@true{true}
\addmetakey[false]{termdef}{local}
\addmetakey{termdef}{name}
```

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

`\capitalize`

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

`\mod@termref` `\mod@termref{`⟨*module*⟩`}{`⟨*name*⟩`}{`⟨*nl*⟩`}` determines whether the macro `\`⟨*module*⟩`@cd@file@base` 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.

```
\def\mod@termref#1#2#3{\def\@test{#3}%
\@ifundefined{#1@cd@file@base}%
{\protect\G@refundefinedtrue%
\@latex@warning{\protect\termref with unidentified cd "#1": the cd key must reference an active
\def\@label{sref@#2 @target}}%
{\def\@label{sref@#2@#1@target}}%
\ifx\csname #1@cd@file@base\endcsname\@empty% local reference
\sref@hlink@ifh{\@label}{\ifx\@test\@empty #2\else #3\fi}\else%
\def\@uri{\csname #1@cd@file@base\endcsname.pdf\#\@label}%
\sref@href@ifh{\@uri}{\ifx\@test\@empty #2\else #3\fi}\fi}
```

4.8 Dealing with Multiple Files

Before we can come to the functionality we want to offer, we need some auxiliary functions that deal with path names.

EdN:12 `\mod@canonicalize` the path simplifier `\mod@canonicalize` is not implemented at the moment[12]

```
\newcommand\mod@canonicalize[1]{#1}
```

We directly test the simplification:

source	result	should be
aaa	aaa	aaa
../../aaa	../../aaa	../../aaa
aaa/bbb	aaa/bbb	aaa/bbb
aaa/..	aaa/..	
../../aaa/bbb	../../aaa/bbb	../../aaa/bbb
../aaa/../bbb	../aaa/../bbb	../bbb
../aaa/bbb	../aaa/bbb	../aaa/bbb
aaa/bbb/../ddd	aaa/bbb/../ddd	aaa/ddd
aaa/bbb/../..	aaa/bbb/../..	

`\defpath`

```
\newcommand\defpath[3][]{\expandafter\newcommand\csname #2\endcsname[1]{#3/##1}}
```

[12] EdNote: MK: Jinbo's task; integrate it somehow

4.9 Loading Module Signatures

4.9.1 Selective Inclusion

`\requiremodules` this macro loads the modules in a file. It does quite a lot of work to make sure that they are only loaded once (actually this does not quite work so they are loaded only twice). `\requiremodules{`⟨*file*⟩`}{`⟨*ext*⟩`}` also deposits a call to `\@requiremodules{`⟨*file*⟩`}{`⟨*ext*⟩`}` to the `aux` file to make sure that the semantic macros are already loaded for the table of contents.

```
\newcommand\requiremodules[2]{%
{\mod@showfalse% save state and ensure silence while reading sms
\edef\mod@path{#1}\edef\mod@ext{#2}% set up path/ext
\input{#1.#2}}}
```

`\@requiremodules` the internal macro that actually does the loading. We disable it at the end of the document, so that when the aux file is read again, nothing is loaded.

```
\newcommand\@requiremodules[2]{\if@tempswa\mod@showfalse\@importingtrue%
\edef\mod@path{#1}\edef\mod@ext{#2}% set up path/ext
\input{#1.#2}\fi}
```

`\sinput`

```
\def\sinput#1{
{\mod@updatedpre{#1}% add the new file to the already existing path
\let\mod@savedprefix\mod@prefix% add the path to the new file to the prefix
\mod@updatedpost{#1}%
\def\mod@blaaaa{}% macro used in the simplify function (remove .. from the prefix)
\mod@simplify{\mod@savedprefix}% remove |xxx/..| from the path (in case it exists)
\mod@reguse{\mod@savedprefix}%
\let\newreg\mod@reg% use to compare, in case the .sms file was loaded before
\mod@search{\mod@savedprefix}% update registry
\ifx\newreg\mod@reg%\message{This file has been previously introduced}
\else\input{\mod@savedprefix}%
\fi}}
```

EdN:13 [13]

```
\let\sinputref=\sinput
\newcommand\inputref[1]{\edef\mod@path{#1}\edef\mod@ext{tex}\input{#1}}
\let\includeref=\include
```

4.10 Including Externally Defined Semantic Macros

`\requirepackage`

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

[13] EdNote: the sinput macro is just faked, it should be more like requiremodules, except that the tex file is inputted; I wonder if this can be simplified.

4.11 Views

We first prepare the ground by defining the keys for the `view` environment.

```
\srefaddidkey{view}
\addmetakey*{view}{title}
\addmetakey{view}{display}
\addmetakey{view}{from}
\addmetakey{view}{to}
\addmetakey{view}{creators}
\addmetakey{view}{contributors}
\addmetakey[sms]{view}{ext}
```

`\view@heading` Then we make a convenience macro for the view heading. This can be customized.

```
\newcounter{view}[section]
\newcommand\view@heading[4]{\if@importing\else%
\stepcounter{view}%
\edef\@display{#3}\edef\@title{#4}%
\noindent\ifx\@display\st@flow\else
{\textbf{View} {\thesection.\theview} from \textsf{#1} to \textsf{#2}}%
\sref@label@id{View \thesection.\theview}%
\ifx\@title\@empty\quad\else\quad(\@title)\fi%
\par\noindent%
\fi\ignorespaces\fi}%ifmod@show
```

`view` The `view` environment relies on the `@view` environment (used also in the sTeX module signatures) for module bookkeeping and adds presentation (a heading and a box) if the `showmods` option is set.

```
\newenvironment{view}[3][]%
{\metasetkeys{view}{#1}\sref@target%
\begin{@view}{#2}{#3}\view@heading{#2}{#3}{\view@display}{\view@title}}
{\end{@view}\ignorespaces}
\ifmod@show\surroundwithmdframed{view}\fi
```

`@view` The `@view` does the actual bookkeeping at the module level.

```
\newenvironment{@view}[2]{%from, to
\@importmodule[\view@from]{#1}{\view@ext}%
\@importmodule[\view@to]{#2}{\view@ext}}
{}
```

`viewsketch` The `viewsketch` environment behaves like `view`, but only has text contents.

```
\newenvironment{viewsketch}[3][]%
{\metasetkeys{view}{#1}\sref@target%
\begin{@view}{#2}{#3}\view@heading{#2}{#3}{\view@display}{\view@title}}
{\end{@view}}
\ifmod@show\surroundwithmdframed{viewsketch}\fi
```

4.12 Support for **MathHub**

`\importmhmodule` The `\importmhmodule` saves the current value of `\mh@currentrepos` in a local macro `\mh@@repos`, resets `\mh@currentrepos` to the new value if one is given in the optional argument, and after importing resets `\mh@currentrepos` to the old value in `\mh@@repos`. We do all the `\ifx` comparison with an `\expandafter`, since the values may be passed on from other key bindings.

```
\srefaddidkey{importmhmodule}
\addmetakey{importmhmodule}{repos}
\addmetakey{importmhmodule}{path}
\addmetakey[sms]{importmhmodule}{ext}
\newcommand\importmhmodule[2][]{\metasetkeys{importmhmodule}{#1}%
\ifx\importmhmodule@path\@empty
\importmodule[ext=\importmhmodule@ext,id=\importmhmodule@id]{#2}\else
\edef\mh@@repos{\mh@currentrepos}%
\ifx\importmhmodule@repos\@empty\else\mhcurrentrepos{\importmhmodule@repos}\fi%
\importmodule[load=\MathHub{\mh@currentrepos/source/\importmhmodule@path},
              ext=\importmhmodule@ext,id=\importmhmodule@id]{#2}%
\mhcurrentrepos\mh@@repos\fi\ignorespaces}
```

and now the analogs

`\usemhmodule`

```
\newcommand\usemhmodule[2][]{\metasetkeys{importmhmodule}{#1}%
\ifx\importmhmodule@path\@empty
\usemodule[ext=\importmhmodule@ext,id=\importmhmodule@id]{#2}\else
\edef\mh@@repos{\mh@currentrepos}%
\ifx\importmhmodule@repos\@empty\else\mhcurrentrepos{\importmhmodule@repos}\fi%
\usemodule[load=\MathHub{\mh@currentrepos/source/\importmhmodule@path},
           ext=\importmhmodule@ext,id=\importmhmodule@id]{#2}%
\mhcurrentrepos\mh@@repos\fi\ignorespaces}
```

`\adoptmhmodule`

```
\newcommand\adoptmhmodule[2][]{\metasetkeys{importmhmodule}{#1}%
\ifx\importmhmodule@path\@empty
\adoptmodule[ext=\importmhmodule@ext,id=\importmhmodule@id]{#2}\else
\edef\mh@@repos{\mh@currentrepos}%
\ifx\importmhmodule@repos\@empty\else\mhcurrentrepos{\importmhmodule@repos}\fi%
\adoptmodule[load=\MathHub{\mh@currentrepos/source/\importmhmodule@path},
             ext=\importmhmodule@ext,id=\importmhmodule@id]{#2}%
\mhcurrentrepos\mh@@repos\fi\ignorespaces}
```

`\mhinputref`

```
\newcommand\mhinputref[2][]{\def\@repos{#1}%
\edef\mh@@repos{\mh@currentrepos}%
\ifx\@repos\@empty\else\mhcurrentrepos{#1}\fi%
\inputref{\MathHub{\mh@currentrepos/source/#2}}%
\mhcurrentrepos\mh@@repos\ignorespaces}
```

`\mhinput`

```
\let\mhinput\mhinputref
```

\mhcurrentrepos \@mhcurrentrepos

`\mhcurrentrepos` is used to initialize the current repository. If the repos has changed, it writes a call to the internal macro `\@mhcurrentrepos` for the aux file and calls it. So that the `\importmodule` calls there work with the correct repos.

```
\newcommand\mhcurrentrepos[1]{\edef\@test{#1}%
\ifx\@test\mh@currentrepos\else%
\protected@write\@auxout{}{\string\@mhcurrentrepos{#1}}\fi%
\@mhcurrentrepos{#1}}
\newcommand\@mhcurrentrepos[1]{\edef\mh@currentrepos{#1}}
```

importmhmodulevia

```
\newenvironment{importmhmodulevia}[3][]{\gdef\@@doit{\importmhmodule[#1]{#2}{#3}}%
\ifmod@show\par\noindent importing module #2 via \@@doit\fi}
{\aftergroup\@@doit\ifmod@show end import\fi}
```

```
\srefaddidkey{mhview}
\addmetakey{mhview}{display}
\addmetakey{mhview}{creators}
\addmetakey{mhview}{contributors}
\addmetakey*{mhview}{title}
\addmetakey{mhview}{fromrepos}
\addmetakey{mhview}{torepos}
\addmetakey{mhview}{frompath}
\addmetakey{mhview}{topath}
\addmetakey[sms]{mhview}{ext}
```

mhview

the MathHub version

```
\newenvironment{mhview}[3][]% keys, from, to
{\metasetkeys{mhview}{#1}\sref@target%
\begin{@mhview}{#2}{#3}\view@heading{#2}{#3}{\mhview@display}{\mhview@title}}
{\end{@mhview}\ignorespaces}
\ifmod@show\surroundwithmdframed{mhview}\fi
```

@mhview

The `@mhview` does the actual bookkeeping at the module level.

```
\newenvironment{@mhview}[2]{%from, to
\importmhmodule[repos=\mhview@fromrepos,path=\mhview@frompath,ext=\mhview@ext]{#1}%
\importmhmodule[repos=\mhview@torepos,path=\mhview@topath,ext=\mhview@ext]{#2}}
{}
```

mhviewsketch

The `mhviewsketch` environment behaves like `mhview`, but only has text contents.

```
\newenvironment{mhviewsketch}[3][]%
{\metasetkeys{mhview}{#1}\sref@target%
\begin{@mhview}{#2}{#3}\view@heading{#2}{#3}{\mhview@display}{\mhview@title}}
{\end{@mhview}\ignorespaces}
\ifmod@show\surroundwithmdframed{mhviewsketch}\fi
```

EdN:14 \obligation

The `\obligation` element does not do anything yet on the latexml side.[14]

```
\newcommand\obligation[3][]{\if@importing\else Axiom #2 is proven by \sref{#3}\fi}
```

[14] EdNote: document above

4.13 Deprecated Functionality

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.[15] EdN:15

```
\define@key{module}{uses}{%
\@for\module@tmp:=#1\do{\activate@defs\module@tmp\export@defs\module@tmp}}
```

`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}{%
\@for\module@tmp:=#1\do{\activate@defs{qualified@\module@tmp}\export@defs\module@tmp}}
```

`\coolurion/off`

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

4.14 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.

`\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.

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

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

```
\newcommand\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.

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.

[BFM05] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986. Internet Engineering Task Force (IETF), 2005. URL: http://www.ietf.org/rfc/rfc3986.txt.

[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.

[Hor+11] Fulya Horozal et al. “Combining Source, Content, Presentation, Narration, and Relational Representation”. In: *Intelligent Computer Mathematics*. Ed. by James Davenport et al. LNAI 6824. Springer Verlag, 2011, pp. 212–227. ISBN: 978-3-642-22672-4. URL: http://kwarc.info/frabe/Research/HIJKR_dimensions_11.pdf.

[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://svn.kwarc.info/repos/stex/doc/mcs08/stex.pdf.

[Koh13] Michael Kohlhase. “Organizing Symbols between Slides, Lecture Notes, Encyclopedias, and Original Papers in Semantic Digital Libraries”. KWARC Blue Note. 2013. URL: https://svn.omdoc.org/repos/omdoc/doc/blue/slides-notes/note.pdf.

[Koh15a] Michael Kohlhase. *metakeys.sty: A generic framework for extensible Metadata in LaTeX*. Tech. rep. Comprehensive TeX Archive Network (CTAN), 2015. URL: http://www.ctan.org/tex-archive/macros/latex/contrib/stex/metakeys/metakeys.pdf.

[Koh15b] Michael Kohlhase. *statements.sty: Structural Markup for Mathematical Statements*. Tech. rep. Comprehensive TeX Archive Network (CTAN), 2015. URL: http://www.ctan.org/tex-archive/macros/latex/contrib/stex/statements/statements.pdf.

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

[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).

[sTeX] *Semantic Markup for LaTeX*. Project Homepage. URL: http://trac.kwarc.info/sTeX/ (visited on 02/22/2011).