modules.sty: Semantic Macros and Module Scoping in $\LaTeX^*$

Michael Kohlhase
FAU Erlangen-Nürnberg
http://kwarc.info/kohlhase

Deyan Ginev
Authorea

Rares Ambrus
Jacobs University Bremen

March 20, 2019

Abstract

The modules package is a central part of the $\LaTeX^*$ collection, a version of $\LaTeX$ that allows to markup $\LaTeX$ documents semantically without leaving the document format, essentially turning $\LaTeX$ 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 $\LaTeX$ sources, or after translation.

*Version v1.4 (last revised 2019/03/20)
# 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 and Inheritance ................................ 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

3 Limitations & Extensions 11
   3.1 Perl Utility sms ........................................... 12
   3.2 Qualified Imports ......................................... 12
   3.3 Error Messages ........................................... 12
   3.4 Crossreferencing ......................................... 12
   3.5 No Forward Imports ....................................... 12

4 The Implementation 14
   4.1 Package Options .......................................... 14
   4.2 Modules and Inheritance ................................ 14
   4.3 Semantic Macros .......................................... 18
   4.4 Defining Math Operators ................................ 22
   4.5 Axiomatic Assumptions .................................. 22
   4.6 Semantic Macros for Variables ......................... 23
   4.7 Testing Semantic Macros ................................ 23
   4.8 Symbol and Concept Names .............................. 24
   4.9 Dealing with Multiple Files ............................. 25
   4.10 Loading Module Signatures .............................. 25
   4.11 Including Externally Defined Semantic Macros ...... 26
   4.12 Deprecated Functionality ............................... 26
   4.13 Experiments ............................................... 27
1 Introduction

Following general practice in the \TeX/La\TeX{} 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 \texttt{\def\Reals{\mathbb{R}}} for individual mathematical objects to more complex (functional) ones object constructors like \texttt{\def\SmoothFunctionsOn#1{\mathcal{C}^\infty(#1,#1)}}. Semantic macros are traditionally used to make \TeX/La\TeX{} code more portable. However, the \TeX/La\TeX{} 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 \cite{Koh08}.

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

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\texttt{\LaTeX{}} & \texttt{\SmoothFunctionsOn{\Reals}} & \mathcal{C}^\infty(\mathbb{R},\mathbb{R}) \\
\hline
\texttt{PDF/DVI} & \texttt{OPENMATH} & % & \texttt{\OMA} \\
& % & \texttt{\OMS cd="calculus" name="SmoothFunctionsOn"/>} \\
& % & \texttt{\OMS cd="calculus" name="Reals"/>} \\
& % & \texttt{\OMA} \\
\hline
\texttt{MathML} & % & \texttt{\apply} \\
& % & \texttt{\csymbol cd="calculus">SmoothFunctionsOn</csymbol> \\
& % & \texttt{\csymbol cd="calculus">Reals</csymbol> \\
& % & \texttt{\apply} \\
\hline
\end{tabular}
\caption{OPENMATH and MathML generated from Semantic Macros}
\end{table}

Example 1: OPENMATH and MathML generated from Semantic Macros

2 The User Interface

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

2.1 Package Options

\texttt{showmods} The \texttt{modules} package takes six options: If we set \texttt{showmods}¹, then the views (see

¹
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 3.2.

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.

If the showmeta option is set, then the metadata keys are shown (see Koh18b for details and customization options).

The mh option enables MathHub support; see Koh18a.

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

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

\symdef\[\langle keys\rangle\]{\langle cseq\rangle}\{\langle args\rangle\}\{\langle definiens\rangle\}

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{name}=xor on the definition of \xorOp.

A key local can be added to \(keys\) to specify that the symbol is local to the

---

1EdNote: 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 internal version that never shows anything during sms reading.
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.

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

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}
\end{module}

What is the sum of the first $\arbitraryn$ odd numbers, i.e. $\Sumfromto{i}1\arbitraryn{2i-1}$?

Example 2: Semantic Markup in a module Context

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

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

**Example 3:** Presentation Variants of a Semantic Macro

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

\symtest[\SetSt{a}{a>0}]

generates the output

**Example 4:** A Semantic Macro Definition with Test

\abbrtest[\SetSt{a}{a>0}]

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

\section{Axiomatic Assumptions}

In many ways, axioms and assumptions in definitions behave a lot like symbols (see \cite{RK13} for discussion). Therefore we provide the macro \assdef that can be used to mark up assumptions. Given a phrase \textlangle phrase\textrangle in a definition\footnote{\textit{EdNote}: only definitions?}, we can use \assdef\{\textlangle name\textrangle\}\{\textlangle phrase\textrangle\} to give this the symbol name \textlangle name\textrangle.\footnote{\textit{EdNote}: continue}

\section{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 \ref{fig:example}). 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 \ref{fig:example} 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 \textit{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:

1. they do not participate in the imports mechanism and
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 `\vardef` for the (semigroup) operation via the `\vardef` macro. `\vardef` works exactly like, except

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

<table>
<thead>
<tr>
<th>\LaTeX</th>
<th>\vardef{op}{[1]}{\assoc\circ{#1}}</th>
</tr>
</thead>
</table>
| OMDoc | \% <notation>
\% <prototype>
\% <OMA>
\% <OMV name="op"/>
\% <expr name="a1"/>
\% <expr name="a2"/>
\% </OMA>
\% </prototype>
\% <rendering>
\% <mrow>
\% <render name="a1"/>
\% <mcode \#x2384; </mcode>
\% <render name="a2"/>
\% </mrow>
\% </rendering>
\% </notation> |
| \LaTeX | \op{x,e} |
| OMDoc | \% <OMA>
\% <OMV name="op"/><OMV name="x"/><OMV name="e"/><OMA> |
| OpenMath | % <apply><ci>op</ci><ci>x</ci><ci>e</ci><ci>e</ci><ci>e</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[\langle keys \rangle]{\langle cseq \rangle}{\langle concept \rangle} to declare the macro \langle cseq \rangle that expands to \langle concept \rangle. See Figure 2 for an example, where we use the \capitalize macro to adapt \langle concept \rangle to the sentence beginning.\footnote{EdNote: continue, describe \langle keys \rangle, they will have to to with plurals, once implemented.} 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 \cite{koh18}. Together with the hyperref package \cite{ro}, this provide cross-referencing to the definitions of the symbols and concepts. As discussed in section 3.4, 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[\langle name=xor \rangle]{\langle xdisjunction \rangle}{\langle exclusive disjunction \rangle}
\capitalize\xdisjunction is commutative: $\xor{p}q=\xor{q}p$\footnote{Actually, in the current \TeX group, \importmodule should be placed directly after the \begin{module}.}

\begin{example}
\termdef[\langle name=xor \rangle]{\langle xdisjunction \rangle}{\langle exclusive disjunction \rangle}
\capitalize\xdisjunction is commutative: $\xor{p}q=\xor{q}p$
\end{example}

2.7 Modules and Inheritance

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=\textmod]{\textid} 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\textmod directive. If the module \textmod contains \importmodule\textmod directives of its own, these are also exported to the importing module.

Thus the \importmodule declarations induce the semantic inheritance relation. Figure 3 shows a module that imports the semantic macros from three others. In the simplest form, \importmodule\textmod will activate the semantic macros and concepts declared by \symdef and \termdef in module \textmod in the current module\cite{lib}. 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\textmod can only work, if the \TeX parser — which linearly goes through the \TeX sources — already came across the module \textmod. 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 \texttt{\LaTeX} module signatures. For any file, \texttt{(file).tex}, we generate a corresponding \texttt{\LaTeX} module signature \texttt{(file).sms} with the \texttt{sms} utility (see also Limitation \[3.1\]), which contains (copies of) all \texttt{\begin{module}}, \texttt{\importmodule}, \texttt{\symdef}, and \texttt{\termdef} invocations in \texttt{(file).tex}. The value of an \texttt{\LaTeX} module signature is that it can be loaded instead its corresponding \texttt{\LaTeX} document, if we are only interested in the semantic macros. So \texttt{\importmodule[load=(filepath)]{(mod)}} will load the \texttt{\LaTeX} module signature \texttt{(filepath).sms} (if it exists and has not been loaded before) and activate the semantic macros from module \texttt{(mod)} (which was supposedly defined in \texttt{(filepath).tex}). Note that since \texttt{(filepath).sms} contains all \texttt{\importmodule} statements that \texttt{(filepath).tex} does, an \texttt{\importmodule} recursively loads all necessary files to supply the semantic macros inherited by the current module.\footnote{\textbf{EdNote:} MK: document the other keys of \texttt{module}}

The \texttt{\metalanguage} macro is a variant of \texttt{\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 \cite{RK13} for discussion.

\section*{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 reused, 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 \texttt{\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 \texttt{\LaTeX} read the prerequisite documents without producing output. For efficiency reasons, \texttt{\LaTeX} chooses a different route. It comes with a utility \texttt{sms} (see Section \[3.1\]) that exports the modules and macros defined inside them from a particular document and stores them inside \texttt{.sms} files. This way we can avoid overloading \texttt{\LaTeX} with useless information, while retaining the important information which can then be imported in a more efficient way.

For such situations, the \texttt{\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 \texttt{.sms} file) is read. Note that the \texttt{\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 \texttt{\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 8: Self-contained Modules via \texttt{importmodule}

In Example 8, we have shown the typical setup of a module file. The \texttt{importmodule} macro takes great care that files are only read once, as \TeX 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 \(\tilde{T}\) into (the OMDoc document \(\tilde{D}\) generated from) a \TeX document \(D\). Naturally, we have to provide an \TeX stub module \(T\) that provides \texttt{symdef} declarations for all symbols we use in \(D\). In this situation, we use \texttt{importOMDocmodule}\((spath)\){\(\langle OURI\rangle\)}{\(\langle name\rangle\)}, where \((spath)\) is the file system path to \(T\) (as in \texttt{importmodule}, this argument must not contain the file extension), \(\langle OURI\rangle\) is the URI to the OMDoc module (this time with extension), and \(\langle name\rangle\) is the name of the theory \(\tilde{T}\) and the module in \(T\) (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 \(T\) is in \(D\) and only contains the \texttt{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 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) \texttt{importmodule} commands. Consider for instance module \texttt{bar} in Example 8, say that \texttt{bar} already has load depth 15, then we cannot naively import it in this way. If module \texttt{bar} depended say on a module \texttt{base} on the critical load path, then we could add a statement \texttt{requiremodules\(\ldots/base\)} in the second line. This would load the modules from \(\ldots/base.smx\) 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 \texttt{foo}. Solving the load depth problem.

The \texttt{inputref} macro behaves just like \texttt{input} in the \LaTeX workflow, but in the \LaTeXX conversion process creates a reference to the transformed version of the input file instead.

2.9 Using Semantic Macros in Narrative Structures

The \texttt{importmodule} macro establishes the inheritance relation, a transitive relation among modules that governs visibility of semantic macros. In particular, it

\footnote{OMDoc theories are the counterpart of \TeX modules.}
\footnote{If you have sufficient rights to change your \TeX installation, you can also increase the variable \texttt{max_in_open} in the relevant \texttt{texmf.cnf} file. Setting it to 50 usually suffices}
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 situation for most parts of mathematical documents. For that \STEX provides the \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 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. 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.

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 \[\TeX].
3.1 Perl Utility sms

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

3.2 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 \TeX{} issue #1505. Until this is implemented the infrastructure is turned off by default, but we have already introduced the `qualifiedimports` option for the future.

3.3 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.

3.4 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 [Koh18c]. A way around this limitation would be to import the current module from the \TeX{} module signature (see Section 2.7) via the `\importmodule` declaration.

3.5 No Forward Imports

\TeX{} 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

---

\textit{EdNote: usemodule should work here; revise}
\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 \texttt{\input{mod}}. As in

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

then the \texttt{\importmodule} command can read \texttt{mod.sms} (created via the \texttt{sms} utility) without having to wait for the module \(mod\) to be defined.
4 The Implementation

The modules package generates two files: the \texttt{\LaTeX} package (all the code between \texttt{\{\texttt{*package}\} and \texttt{\{/package\}}) and the \texttt{LaTeXML} bindings (between \texttt{\{\texttt{*ltxml}\} and \texttt{\{/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 \texttt{xxx} will just set the appropriate switches to true (otherwise they stay false). The options we are not using, we pass on to the \texttt{sref} package we require next.

\begin{verbatim}
\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}
\newif\if@mmt\@mmtfalse
\DeclareOption{mmt}{\@mmttrue}
\DeclareOption*{\PassOptionsToPackage{\CurrentOption}{sref}}
\ProcessOptions
\end{verbatim}

\LaTeX{} does not support module options yet, so we do not have to do anything here for the \texttt{LaTeXML} bindings. We only set up the \texttt{PERL} packages (and tell \texttt{emacs} about the appropriate mode for convenience).

The next measure is to ensure that the \texttt{sref} and \texttt{xcomment} packages are loaded (in the right version). For \texttt{LaTeX{}}, we also initialize the package inclusions.

\begin{verbatim}
\RequirePackage{sref}
\if@modules@mh@\RequirePackage{modules-mh}\fi
\if@mmt\RequirePackage{mmt}\fi
\RequirePackage{xspace}
\RequirePackage{mdframed}
\RequirePackage{pathsuris}
\end{verbatim}

4.2 Modules and Inheritance

We define the keys for the \texttt{module} environment and the actions that are undertaken, when the keys are encountered.

\texttt{module:cd} This \texttt{KeyVal} key is only needed for \LaTeX{} at the moment; use this to specify a content dictionary name that is different from the module name.
For a module with \texttt{id=\langle \textit{name} \rangle}, we have a macro \texttt{\module@defs\langle \textit{name} \rangle} that acts as a repository for semantic macros of the current module. I will be called by \texttt{importmodule} to activate them. We will add the internal forms of the semantic macros whenever \texttt{\symdef} is invoked. To do this, we will need an unexpended form \texttt{this\module} that expands to \texttt{\module@defs\langle \textit{name} \rangle}; we define it first and then initialize \texttt{\module@defs\langle \textit{name} \rangle} as empty. Then we do the same for qualified imports as well (if the \texttt{qualifiedimports} option was specified). Furthermore, we save the module name in the token register \texttt{\mod@id}.

\begin{verbatim}
\define@key{module}{id}{\edef\this@module{\expandafter\noexpand\csname module@defs@#1\endcsname}}%\csgdef{module@defs@#1}{}%\ifmod@qualified\edef\this@qualified@module{\expandafter\noexpand\csname module@defs@#1\endcsname}\csgdef{module@defs@qualified@#1}{}\fi%\def\mod@id{#1}%
\end{verbatim}

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

\begin{verbatim}
\ifdef{\thesection}{\newcounter{module}\[section]\}{\newcounter{module}}%\newrobustcmd\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%\def\mod@id{id}(\#1)%
\end{verbatim}

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 \texttt{\module@\langle \textit{name} \rangle@path} and \texttt{\module@\langle \textit{name} \rangle@ext}, so that we can use them later. The source of these two macros, \texttt{\mod@path} and \texttt{\mod@ext}, are defined in \texttt{\requiremodules}. 

\begin{verbatim}

\end{verbatim}
\newenvironment{module}[1][]{%
 \begin{module}[#1]%
 \ifsundef{mod@id}{% only define if components are!
 \ifsundef{mod@path}{%\csxdef{module@mod@id @path}{{\mod@path}}%
 \ifsundef{mod@ext}{%\csxdef{module@mod@id @ext}{{\mod@ext}}}%
 }{}
 \if@importing\else\mmtheory{mod@id}{{????}}\fi\fi% make the headings
 \ignorespaces}{%
 \if@importing\else\mmtheoryend\fi\fi%
 \end{module}%
 \ignorespacesafterend%
 \ifmod@show\surroundwithmdframed{module}\fi%
\activate@defs
 To activate the \symdef{}s from a given module \langle mod \rangle, 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@#1@activated}{\csname module@defs@#1\endcsname}{}%
 \@namedef{module@#1@activated}{true}%
 }%
\export@defs
\export@defs{⟨ mod ⟩} exports all the \symdef{}s from module \langle mod \rangle to the current module (if it has the name \langle currmod \rangle), by adding a call to \module@defs@{}⟨ mod ⟩ to the registry \module@defs@{}⟨ currmod ⟩.78

Naive understanding of this code: \#1 be will be expanded first, then \this@module, then \active@defs, then \g@addto@macro.
\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 conditional and an auxiliary macro:
\if@importing \if@importing can be used to shut up macros in an import situation.
\newif\if@importing\ifimportingfalse
\update@used@modules
 This updates the register \used@modules

\begin{EdNote}
MK: I have the feeling that we may be exporting modules multiple times here, is that a problem?
\end{EdNote}
\begin{EdNote}
Jinbo: This part of code is extremely easy to generate bugs, cautiously edit this part of code.
\end{EdNote}
The \texttt{importmodule} macro is an interface macro that loads \texttt{file} and activates and re-exports the \texttt{symdef}s from module \texttt{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 \texttt{used@modules} register and an internal one (\texttt{@importmodule}) that does the actual work.

\begin{verbatim}
\gdef\used@modules{}
\srefaddidkey{importmodule}
\addmetakey{importmodule}{load}
\addmetakey{importmodule}{ext}
\addmetakey{importmodule}{conservative}[true]
\newcommand\importmodule[2][]{% 
\metasetkeys{importmodule}{#1}% 
\update@used@modules{#2}% 
\@importmodule[\importmodule@load]{#2}{\importmodule@ext}{export}% 
\ignorespacesandpars%
}
\@importmodule
\@importmodule[\importmodule@load]{#2}{\importmodule@ext}{export}% 
\ignorespacesandpars%
\end{verbatim}

\texttt{@importmodule} \texttt{[\texttt{filepath}]}\texttt{\{\texttt{mod}\}}\texttt{\{\texttt{ext}\}\{\texttt{export}\}} loads \texttt{filepath} \texttt{.\texttt{ext}} (if it is given) and activates the module \texttt{\texttt{mod}}. If \texttt{\texttt{export}} is \texttt{export}, then it also re-exports the \texttt{symdef}s from \texttt{\texttt{mod}}.

First \texttt{@load} will store the base file name with full path, then check if \texttt{module@\texttt{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 \texttt{requiremodules}.

\begin{verbatim}
\newcommand\importmodule[4][]{% 
  \if@importingtrue% to shut up macros while in the group opened here
  \edef\@load{#1}%
  \ifx\@load\@empty%
    \relax%
  \else%
    \ifcsundef{module@#2@path}{{\texttt{requiremodules}{#1}{#3}%}}{% 
      \edef\@path{\csname module@#2@path\endcsname}%
      \IfStrEq\@load\@path{% if the known path is the same as the requested one
        \relax% 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 "\@path"\MessageBreak
          }\end{verbatim}
The imported path "@load" is probably a different module with the same name; this is dangerous -- not importing

{Check whether the Module name is correct}%

\activate@defs{#2}% activate the module
\edef\@export{#4}\def\@@export{export}% prepare comparison
\ifx\@export\@@export\export@defs{#2}\fi% export the module
\if@importing\else\if@mmt\mmtinclude{#1}{#2}\fi\fi%

\usemodule \usemodule acts like \importmodule, except that the sms utility does not transfer it to the module signatures and it does not re-export the symdefs.
\newcommand\usemodule[2]{% \metasetkeys{importmodule}{#1}
\update@used@modules{#2}
@importmodule{importmodule@load}{#2}{importmodule@ext}{noexport}
\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 for the \LaTeX side we can just re-use \importmodule, for the \LaTeXML side we have a full URI anyways. So things are easy.\footnote{EdNote: MK@DG: this macro is seldom used, maybe I should just switch arguments.}
\newrobustcmd\importOMDocmodule[3]{\importmodule[#1]{#3}}%

\metalanguage \metalanguage behaves exactly like \importmodule for formatting. For \LaTeX, 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][]{%
    \ifx\conceptdef@display\st@flow\else\stDMemph{\conceptdef@type} #2:\fi%
    \ifx\conceptdef@title\@empty\else~\stDMemph{\conceptdef@title}\par\fi%
}

EdN:10

\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 L\TeX 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}{name}%
\addmetakey*{symdef}{title}%
\addmetakey*{symdef}{description}%
\addmetakey{symdef}{subject}%
\addmetakey*{symdef}{display}%

EdN:11

\symdef The the \symdef, and \@symdef macros just handle optional arguments.
\def\symdef\@ifnextchar[{{\@symdef}{\@symdef[]}}%
\def\@symdef[#1]#2\@ifnextchar[{{\@symdef[#1]}{\@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 \url{http://tex.stackexchange.com/questions/23100/looking-for-an-ignorespacesandpars} it ignores spaces and following implicit paragraphs (double newlines), explicit \par are respected however

\begin{itemize}
\item[10] EdNote: MK\@DG: maybe we need to add DefKeyVals here?
\item[11] EdNote: MK\@MK: we need to document the binder keys above.
\end{itemize}
\def\ignorespacesandpars{%
  \begingroup
  \catcode13=10
  \@ifnextchar\relax{%
    \endgroup}%
  \endgroup%
}

and more adapted from \url{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\@gobble}%
  {}%
}

\@@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}%

  If the mmt option is set and we are not importing, then we write out the constant declaration for this symdef\footnote{\textbf{EdNote}: eventually we may want to do something about the notations. This would pass #4 to MMT via a macro that makes the # (argumentmarkers) active and empty. I am not clear how well this works, so we leave out notations.}

  \if@mmt\if@importing\else%
  \ifx\symdef@name\@empty\mmtconstdec{#2}\else\mmtconstdec{\symdef@name}\fi%
  \fi\fi%

  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@nc\csname mod@symref@#2\expandafter\endcsname\expandafter%
  \expandafter{\expandafter\mod@termref\expandafter{\mod@id}{#2}{##1}}%

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

  \expandafter\@mod@nc\csname mod@symref@#2\expandafter\endcsname[#3][]{#4}%
  \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.

\begin{verbatim}
\if@symdeflocal\%
\else\%
\ifsundef{mod@id}\%
\else\%
\ifsundef{mod@id}\%
\else\%
\if@symdeflocal\%
\else\%
\ifsundef{mod@id}\%
\else\%
\end{verbatim}

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@.

\begin{verbatim}
\expandafter\g@addto@macro\this@module\%
\expandafter\g@addto@macro\this@module\%
\end{verbatim}

Then we add the definition of (sym) which calls the intermediate function and handles the optional argument.

\begin{verbatim}
\expandafter\g@addto@macro\this@module\%
\expandafter\g@addto@macro\this@module\%
\end{verbatim}

We also add \mod@symref@(sym) macro to the macro pool so that the \symref macro can pick it up.

\begin{verbatim}
\expandafter\g@addto@macro\this@module\%
\expandafter\g@addto@macro\this@module\%
\end{verbatim}

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

\begin{verbatim}
\ifmod@qualified\%
\expandafter\g@addto@macro\this@module\%
\expandafter\g@addto@macro\this@module\%
\end{verbatim}

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

\begin{verbatim}
\ifmod@show\%
\ifsymdef@display\st@flow\else\noindent\stDMemph{symdef@type} #2:\fi\%
\ifsymdef@display\st@flow\else\noindent\stDMemph{symdef@type} #2:\fi\%
\end{verbatim}

\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 \modules@(sym)@pres@ induced by the \symdef.\footnote{\textsc{EdNote}: MK@DG: this needs to be implemented in \LaTeXML}
and if we are in a named module, then we need to export the function \modules\(\langle\text{sym}\rangle\pres\langle\text{opt}\rangle\) just as we have done that in \symdef.

\Resymdef This is now deprecated.

\Abbrdef The \Abbrdef macro is a variant of \Symdef that does the same on the \LaTeX level.

4.4 Defining Math Operators

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

4.5 Axiomatic Assumptions

\Assdef We fake it for now, not clear what we should do on the \LaTeX side.
4.6 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 \LaTeXXML binding exclusively.
\begin{verbatim}
238 \def\vardef{\@ifnextchar[\@vardef}{\@vardef[]}
239 \def\@vardef[#1]{\@ifnextchar[\@@vardef[#1]{#2}}
240 \def\@@vardef[#1]{#2}{\@vardef[#1]}
241 \def\@test{\@empty}
242 \symdef[local]{#2}{#3}{#4}
243 \else
244 \symdef[local]{#2}{#3}{#4}
245 \fi
246 \ignorespacesandpars}
\end{verbatim}

4.7 Testing Semantic Macros

\symtest Allows to test a \symdef in place, this shuts up when being imported.
\begin{verbatim}
249 \addmeta{symtest}{name}
250 \addmeta{symtest}{variant}
251 \newrobustcmd{\symtest}[3][{}]
252 \if@importing
253 \else
254 \metasetkeys{symtest}{#1}
255 \par\noindent \textbf{Symbol} \ifx\symtest@name\@empty\texttt{#2}\else\texttt{\symtest@name}\fi
256 \ifx\symtest@variant\@empty\else (variant \texttt{\symtest@variant})\fi
257 \with semantic macro \texttt{\textbackslash #2}\ifx\symtest@variant\@empty\else[\symtest@variant]\fi
258 \fi
259 \ignorespacesandpars
\end{verbatim}

\abbrtest \addmeta{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
\with semantic macro \texttt{\textbackslash #2}\ifx\abbrtest@variant\@empty\else[\abbrtest@variant]\fi
\fi
\ignorespacesandpars
\end{verbatim}
4.8 Symbol and Concept Names

\texttt{\termdef}

\begin{verbatim}
274 \def\mod@true{true}\%
275 \addmetakey[false]{termdef}{local}\%
276 \addmetakey{termdef}{name}\%
277 \newrobustcmd{\termdef}[3][]{\%
278 \metasetkeys{termdef}{#1}\%
279 \expandafter{\mod@newcommand{\csname #2\endcsname}[0]{\#3}\xspace}\%
280 \ifx\termdef@local\mod@true\%
281 \else\%
282 \ifsundef{mod@id}{\%\%
283 \expandafter{\g@addto@macro\this@module%}
284 \{\expandafter{\mod@newcommand{\csname #2\endcsname}[0]{\#3}\xspace}\%
285 \}\%
286 \fi\%\%
287 \}%\%
\capitalize
288 \def\@capitalize#1{\uppercase{#1}}\%
289 \newrobustcmd{\capitalize}[1]{\expandafter\@capitalize #1}\
\end{verbatim}

\texttt{\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 \texttt{smultiling} package).

\begin{verbatim}
290 \newcommand{\mod@component}[1]{\%
\end{verbatim}

\texttt{\mod@termref}

\begin{verbatim}
\mod@termref{(module)}{(name)}{(nl)} determines whether the macro \texttt{\module@(module)@path} is defined. If it is, we make it the prefix of a URI reference in the local macro \texttt{@uri}, which we compose to the hyper-reference, otherwise we give a warning.\footnote{EdNote: MK: this should be rethought, in particular the local reference does not work!}
\end{verbatim}

\begin{verbatim}
291 \newcommand{\mod@termref}[3][\def\test{#3}]{\%
292 \edef\@test{#3}\%
293 \ifx\@test\@empty\%
294 \PackageWarning{modules}{$\protect\termref'$ with unidentified cd "#1":\MessageBreak
295 the cd key must reference an active module}\%
296 \else\%
297 \PackageError{modules}{$\protect\termref'$ with unidentified cd "#1"}\%
298 \{the cd key must reference an active module}\%
299 \fi\%\%
300 \mod@label{sref@#2@#1@target}\%
301 \edef\@test{#3}\%
302 \ifx\@test\@empty\%
303 \PackageWarning{modules}{$\protect\termref'$ with unidentified cd "#1":\MessageBreak
304 the cd key must reference an active module}\%
305 \else\%
306 \PackageError{modules}{$\protect\termref'$ with unidentified cd "#1"}\%
307 \{the cd key must reference an active module}\%
308 \fi\%\%
309 \mod@uri{\csname module@#1@path\endcsname\mod@component{#1}.pdf#\@label}\%
310 \mod@href{\@uri}{\ifx\@test\@empty#2\else#3\fi}\%
\end{verbatim}

\begin{verbatim}
14 EdNote: MK: this should be rethought, in particular the local reference does not work!\footnote{EdNote: MK: this should be rethought, in particular the local reference does not work!}
\end{verbatim}
4.9 Dealing with Multiple Files

We use the \pathsuris package to deal with the canonicalization of paths. \@cpath will canonicalize a path and store the result into \@CanPath. To print a canonicalized path, simply use \cpath{⟨path⟩}.

\@rinput \@rinput{⟨path to the current file without extension⟩}{⟨extension⟩} allows loading modules with relative path. For example, \@rinput{foo/bar/B}{tex} will load foo/bar/B.tex.

\begin{verbatim}
\def\CurrentDir{}\%
\newrobustcmd{\@rinput}{2}{\@cpath{\CurrentDir#1}\% 
\StrCut[\value{RealAddrNum}]{/}\@CanPath{/}{\@TempPath}{/}{\@Rubbish}\% 
\StrCut[1]{\@TempPath{/}{\@Rubbish}{\@DirPath}\% 
\edef\CurrentDir{\@DirPath}\%
\edef\mod@path{}\%
\edef\mod@ext{}\%
\input{\@CanPath.#2}\%
\def\CurrentDir{}\%
\end{verbatim}

4.10 Loading Module Signatures

4.10.1 Selective Inclusion

\requiremodules this macro loads the modules in a file and makes sure that no text is deposited (we set the flags \mod@showfalse and \@importingtrue in the local group). It also remembers the file name and extension in \mod@path and \mod@ext so that \begin{module} can pick them up later.

\begin{verbatim}
\newrobustcmd\requiremodules{2}{\% 
\mod@showfalse\%
\@importingtrue\% save state and ensure silence while reading sms 
\edef\mod@path{#1}\%
\edef\mod@ext{#2}\% set up path/ext 
\input{#1.#2}\%
\end{verbatim}

\@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.

\begin{verbatim}
\newrobustcmd\@requiremodules{2}{\%
\if@tempswa\requiremodules{#1}{#2}\fi\%
\end{verbatim}

\footnote{\textbackslash TeX mod@termref: external reference to \textbackslash@uri}
\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).}\\footnote{EdN:16}

\newrobustcmd\inputref\{2\}\{\%
\def\@Slash{/}
\edef\@load{#2}\%
\StrChar{\@load}{1}{\@testchar}
\ifx\@testchar\@Slash\%
\edef\mod@path{#2}\%
\edef\mod@ext{tex}\%
\input{#2}\%
\else\%
\@rinput{#2}{tex}\%
\fi\%
\}

4.11 Including Externally Defined Semantic Macros
\requirepackage
\newrobustcmd\requirepackage\{1\#2\{\makeatletter\input{#1.sty}\makeatother\}%

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.

\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.

\footnote{EdN:17 EdNote: this issue is deprecated, it will be removed before 1.0.}

\footnote{EdNote: MK: the first (optional) argument is not used. Maybe do something with a non-standard (i.e. non-tex) extension with an optional argument?}
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](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.

\notationdef

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

\reqmodules

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

<table>
<thead>
<tr>
<th>Entry</th>
<th>Page Numbers</th>
<th>Related Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>LaTeXML</td>
<td>3</td>
<td>10, 18, 23</td>
</tr>
<tr>
<td>module,</td>
<td>8</td>
<td>inheritance</td>
</tr>
<tr>
<td>semantic</td>
<td></td>
<td>relation</td>
</tr>
<tr>
<td>MathML</td>
<td>3, 4, 6, 7</td>
<td>OMDoc, 3, 7, 11, 10</td>
</tr>
<tr>
<td>module,</td>
<td>8</td>
<td>semantic</td>
</tr>
<tr>
<td>name,</td>
<td></td>
<td>inheritance</td>
</tr>
<tr>
<td>Perl,</td>
<td>14</td>
<td>XML, 7</td>
</tr>
<tr>
<td>relation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
   Moved MH Versions to a
   separate mathhub package ............. 1
References


[sTeX] KWARC/s\( \text{T}\LaTeX \). url: [https://github.com/KWARC/sTeX](https://github.com/KWARC/sTeX) (visited on 05/15/2015).