Towards computer-assisted semantic markup of mathematical documents

CICM 2023 - Doctoral Programme

Luka Vrečar

Supervisors: Joe Wells, Fairouz Kamareddine

Heriot-Watt University

September 5, 2023

General research aims

- Trying to make computer representations of mathematical documents for human readers less ambiguous using semantic markup.
  - Textbooks, research papers, lecture notes, slides, etc.
  - Humans can disambiguate them well "on the fly", but computers cannot.
  - This could improve, e.g., screen readers for mathematics for the blind, interfacing with computer systems for mathematics, and teaching (through applications built on top of the semantic markup).
- We need to do this with LaTeX!
  - It is used by most authors to typeset mathematics.
  - Alternatives like MS Word or LibreOffice are not suitable.
  - HTML is a possible alternative, but not in the near future. If we want something now, it needs to be in LaTeX.

sTeX: a possible solution

Suppose we are trying to typeset the ambiguous $P \times Q$

- In plain LaTeX, this is achieved using `$P \times Q$`
- The $\times$ is overloaded. It can stand for matrix multiplication, the Cartesian product, etc.
- sTeX allows us to use specific macros to semantically mark up the formula using, e.g., `$\matrixtimes[x]{P, Q}$` or `$\cart{P, Q}$`

The new sTeX 3 provides a possible way to remove some ambiguity from mathematical documents, with relative ease. However, there is limited support for the creation of new documents, and none for converting the millions of existing ones. Our aim is to help facilitate the semantic markup of new and existing mathematical documents.

Implementing basic sTeX macros

In sTeX 3, defining new macros is very straightforward:

- We use `\symdef` to define the macros:
  `\symdef{mult}[name=multiplication, args=2]{#1 \cdot #2}`. Typing `$\mult{p}{q}$` yields $p \cdot q$
  - We use `\notation` to define different notations:
    `\notation{mult}[x]{#1 \times #2}`. Typing `$\mult[x]{p}{q}$` yields $p \times q$
- We use `\argsep` for *flexary* arguments:

  ```
  \symdef{mult}[name=multiplication, args=a]
  {\argsep{#1}{\cdot}}
  ```

  Typing `$\mult{p,q,r}$` yields $p \cdot q \cdot r$
  - Note that this `\symdef` overrides the definition above!

Starting out with sTeX

After hearing about the new and improved sTeX 3, I wanted to try it. I hoped to improve the course materials for the Foundations 1 course at Heriot-Watt, a course about the λ-calculus.
A quick aside on the λ-calculus:

- Each term is either a variable (x, y, z), an application of 2 terms (AB), or an abstraction over a variable ($\lambda x.A$)
- The three main operations we perform on terms are α-conversion, substitution, and β-reduction
- We don't represent terms as α-equivalence classes (and it's not obvious how to do that using sTeX). More specifically, we use variable names and not de Bruijn indices.
- Parentheses are used to disambiguate terms: $(\lambda x.A)B \neq \lambda x.(AB) = (\lambda x.(AB))$

sTeX macros for the λ-calculus

We can define 3 macros and use them to semantically mark up λ-terms as follows:

- `\symdef{app}[name=application, args=2]{(#1 #2)}`
- `\symdef{abs}[name=abstraction, args=Bi]{(\comp{\lambda}\argsep{#1}{ }\comp{.}(#2))}`
- `\symdef{var}[name=variable, args=1]{#1}`

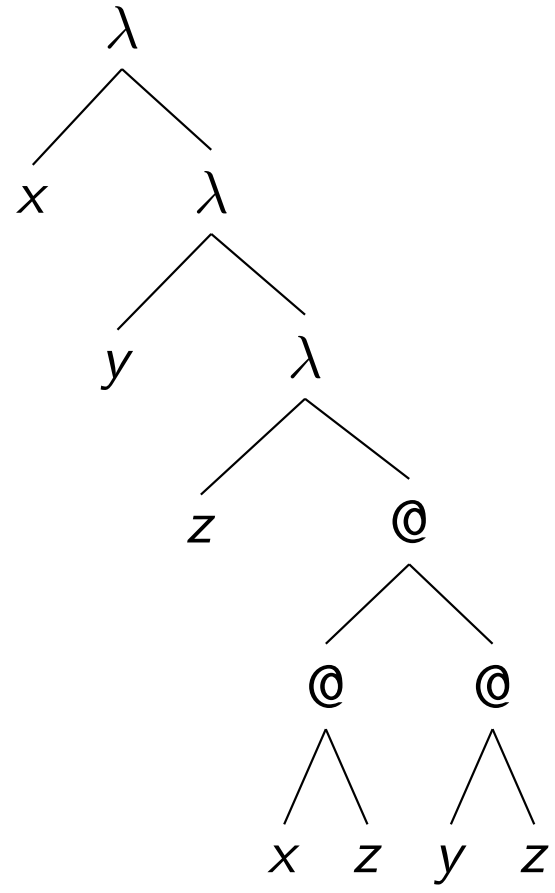
We can also define notations that give us precise control over what pairs of parentheses appear:

- `\notation{app}[nb]{#1 #2}`, typeset as AB
- `\notation{abs}[nob]{\argsep{#1}{}.(#2)}`, typeset as $\lambda xyz.(A)$
- `\notation{abs}[nib]{(\argsep{#1}{}.#2)}`, typeset as $(\lambda xyz.A)$
- `\notation{abs}[nb]{\argsep{#1}{}.#2}`, typeset as $\lambda xyz.A$

Using the macros

The macros and notations we have just defined allow us to typeset terms like $\lambda xyz.(xz(yz))$. Examining the source code shows how it resembles the abstract syntax tree of the term:

```
\abs[nb]
    {x,y,z}
    {\app[nb]
        {\app[nb]{x}{z}}
        {\app{y}{z}}
    }
```



What semantic markup can facilitate

In the case of the λ-calculus one could, for example

- Perform substitution or β-reduction on λ-terms within documents at compilation time. This can be done with, e.g., Lua.
- A way to display the tree structure of λ-terms on hover. This can be done with, e.g., JavaScript in PDF/HTML.

These are just examples, the tip of the iceberg of possible applications.

- Performing manipulations of terms directly inside LaTeX documents could enable easy generation of multiple worked examples for lecture notes, authors' work could be checked for soundness "on the fly" (the sTeX IDE already has some support for checking type correctness), etc.
- If we use a standardised way of semantic markup (such as sTeX), we can develop standardised interfaces with screen readers, CAS, automated theorem provers, etc.

How do we get there?

- Apart from adapting document authoring workflows to use sTeX, we need to add semantic markup to the millions of existing mathematical documents.
- Initially I tried manually constructing a context-free grammar to parse λ-terms written in LaTeX, and representing their abstract syntax tree using sTeX macros.
  - This was time consuming and as such would not scale to other areas of mathematics easily
- I started exploring the idea of somehow converting existing sTeX macros into CFG rules, and parsing documents with those.

Automatic grammar generation: idea

Example: from
`\symdef{abs}[args=ai]{\lambda \argsep{#1}{} . #2}` the following rules are generated:

$$\texttt{abs} \rightarrow \texttt{"\lambda"}\ \texttt{abs1list}\ \texttt{"."}\ \texttt{arg}$$

$$\texttt{abs1list} \rightarrow \texttt{abs1}\ \texttt{abs1list} \mid \texttt{abs1}$$

Rule generation

- Each argument placeholder is replaced by `arg`
- Each `\argsep{FOO}{sep}` is turned into a separate rule of the form $\texttt{FOOlist} \rightarrow \texttt{FOO}\ \texttt{sep}\ \texttt{FOOlist} \mid \texttt{FOO}$

Automatic grammar generation: supporting infrastructure

I am implementing it in Python making use of `parglare`, a GLR parsing library. To work, the grammar generation needs some "supporting infrastructure". This was all done as quick mockups rather than general solutions.

- Formula extraction, for extracting text between math mode delimiters (currently only $s) from LaTeX documents.
  - This does not handle LaTeX code in math mode for typesetting purposes, e.g., `$\textcolor{red}{x} + y$` to typeset $\textcolor{red}{x} + y$, or laying formulas out using tables
- Macro extraction, for finding sTeX macro definitions (regular expressions matching the `\symdef` and `\notation` keywords).
- Python objects for storing information about encountered macros and generated grammar rules.

Automatic grammar generation: where am I right now?

The approach is still in initial development. So far, I only tested it on my macros for the λ-calculus.

- The grammars overgenerate.
  - The λ-term $ABCD$ has 8 valid parses according to the grammar (only 1 correct since application is left-associative).
  - They could possibly be restricted using other features of sTeX macros, such as types, precedence, and associativity
- It is hard to determine what sTeX modules we would need to create a grammar for parsing a given document.
  - Maybe ML can be used to generate a list of suggestions for the user.

What I need to do

Issues with the grammar generation need to be addressed.
Some form of user interface is needed for the parsing and disambiguation.

- First we need to figure out what the most suitable interface is (web application, IDE extension, standalone program, ...)
- Then we need to develop, test, and release it
- We should work closely with working mathematicians to suit their use cases best

Apart from semantically marking up existing documents, we can also explore ways of making authoring new semantically marked up documents easier. We can possibly use some of the techniques we use for marking up existing documents.
Then, we can turn our attention to developing other applications that make use of semantic markup such as screen readers, those we mentioned earlier, etc.

Questions