Towards computer-assisted semantic markup of mathematical documents

Year 2 progression talk

Luka Vrečar

Heriot-Watt University

June 20, 2024

Introduction

- Documents written in LaTeX often contain ambiguous formulas (e.g., `$P \times Q$`).
- We can disambiguate them with sTeX (e.g., `$\cart{P}{Q}$`).
    - Other advantages - interaction with computer algebra systems, interactive theorem provers, screen readers, etc.
- Semantic markup via sTeX ("sTeX-ification") is more involved, so I hope to somewhat automate the process.

Proposed approach

For a given document we wish to sTEX-ify:

1. Identify which macros are needed and define any missing ones.
2. Generate a context-free grammar.
3. Parse all the formulas in the document with the grammar from step 2.
4. Disambiguate any ambiguous parses with a graphical user interface (GUI).
5. Create a copy of the original document, with formulas replaced by their sTEX counterparts.

New macros for λ-*terms*

- I designed some macros for λ-*terms* in Year 1.
- They have been improved using sTeX features like type and precedence
- There are now fewer notations, which makes them easier to use

```
\symdef{var}[name=variable, args=1,
↪   type=\varSet]{#1}
\symdef{abs}[name=abstraction, args=ai,
↪   prec=51;\infprec x\infprec,
↪   type=\funspace{\varSet,
↪   \setOfLambdas}{\setOfLambdas}]{\maincomp{⌟
↪   \lambda}\argsep{#1}{}\comp{.}#2}
\symdef{app}[name=application, args=2,
↪   prec=50;50x49, type=\funspace{\setOfLambdas,
↪   \setOfLambdas}{\setOfLambdas}]{#1 #2}
```

Grammar generation - initial approach

1. Find sTeX macro definitions and replace argument placeholders with a special nonterminal, `arg`.
2. Create a main rule, with `arg` on the LHS and all other nonterminals on the RHS.
3. Add a simple text-recognizing regex if all else fails

Macro definition	**Grammar rule**
`\symdef{var }[args=1]{#1}`	`var` → `arg`
`\symdef{app }[args=2]{#1 #2}`	`app` → `arg arg`
`\symdef{abs }[args=2]{ \lambda#1.#2}`	`abs` → "`\lambda`" `arg` "`dot`" `arg`
Main rule	`arg` → `var` | `app` | `abs` | `[a-z]+?`

Grammar generation - issues with the initial approach

- The grammars would *over-generate*, i.e., they produced many non-sensical trees
- Assuming anything can be an argument to any macro does not make sense mathematically
- For abstraction for example, the first argument should only be a variable

Grammar generation - general improvements

There are some improvements I made to the initial approach

- Information is extracted from semantic macros more reliably using `latexwalker`, a Python library for parsing LaTeX snippets
- The generation of rules is more systematic
  - Each semantic macro has its own "main" rule, which expands into all the individual notation rules (which then have argument placeholders replaced with `arg`)

Grammar generation - adding types

- Some macro definitions also contain *types*
- `\symdef{natplus}[args=2, type=\funspace{\Nat, \Nat}{\Nat}]{#1 + #2}`
- This macro has type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ - it takes in two natural numbers (*input types*) and returns a natural number (*output type*)
- We can restrict grammar rules by matching output types with arguments of the correct input type for each notation rule

$$
\begin{aligned}
\texttt{natplus} &\to \texttt{natArg1}\ +\ \texttt{natArg2} \\
\texttt{natArg1} &\to \texttt{natType} \\
\texttt{natArg2} &\to \texttt{natType} \\
\texttt{natType} &\to \texttt{natplus} \mid \ldots
\end{aligned}
$$

Grammar generation - adding types

- Not a lot of macros actually provide types, so we need a different solution
- Possibly, we can create an interface for editing grammars where users can select which macros can be arguments to other macros
- In this way we add types to macros in a more “loose” sense

Grammar generation - adding precedence

- We can add precedence to macros for things like automated bracketing
- We can use them as precedences during parsing, but they must be remapped first
- sTeX precedences go from -2^{32} (highest precedence) to 2^{32} (lowest precedence) with a default of 0
- `parglare` precedences are non-negative integers with a default of 10

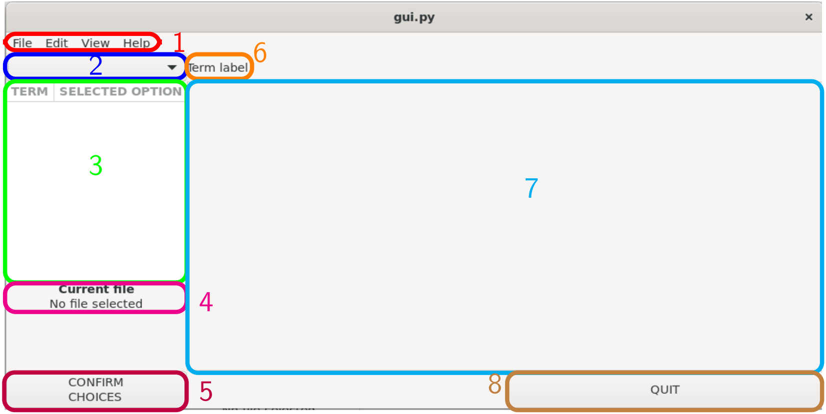
Grammar generation - issues and improvements

- Grammars sometimes contain cycles, which `parglare` cannot work with
  - We can address this with a different parser, like DynGenPar
- There is currently no way to generate a grammar from more than one sTeX archive at a time - addressed in future work
- Grammars must sometimes be manually edited
  - Improving the code might solve this to some extent
  - Developing an interface for creating/merging/editing grammars will also help

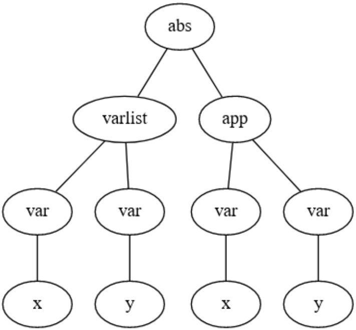
A GUI for disambiguation during parsing - motivation

- Formulas may parse ambiguously, and comparing terminal printouts is not easy
- We can visualise all parses side by side in a nicer way
- This tool can then evolve into a program for all steps of sTeX-ification, from grammar generation to producing the actual sTeX-ified documents

A GUI for disambiguation during parsing - design



A GUI for disambiguation during parsing - tree visualisation




```
abs             abs
   varlist         varlist
      var             var
         x               x
      var             var
         y               y
   app             app
      var             var
         x               x
      var             var
         y               y
```

A GUI for disambiguation during parsing - example

I will now show the GUI in practice on a small example file

A GUI for disambiguation during parsing - improvements

- Currently, it is hard to use it with large complex formulas
  - Adding more compact visualisations
  - Joining parse trees as much as possible
- "α-equivalent" formulas must be disambiguated separately (e.g., $\lambda x.xy$ and $\lambda y.yz$)
- Context is important, but the GUI just shows formulas
  - Show a PDF with highlighted ambiguous formulas that users can click on to show parse trees

DynGenPar - introduction

- Developed by Kevin Kofler as part of the FMathL project
- Studied it for my BSc
  - The C++ implementation is very different from the description in Kofler's PhD thesis
  - The description was for a non-deterministic algorithm that made random choices during parsing
  - The implementation used continuation-passing style to concurrently explore all possible parses
  - I extracted a minimal core and produced a more formal description of the implementation
- It could be useful for cyclic grammars, so I translated it fom C++ to Python

DynGenPar - comparison to GLR

- Similar to GLR, but replaces parsing tables with an *initial graph*
- The graph connects symbols of a grammar based on whether a rule connects them

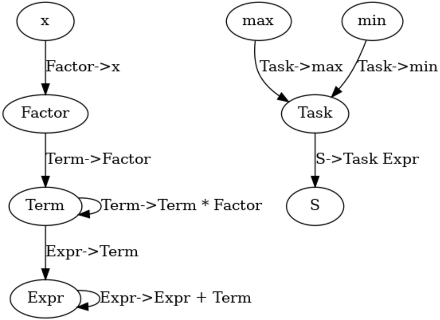
S $\to$ Task Expr

Task $\to$ *min* | *max*

Expr $\to$ Expr + Term | Term

Term $\to$ Term * Factor | Factor

Factor $\to$ *x*



DynGenPar - translation into Python

- Needed a translation from C++ to Python
  - It interfaces easier with my other code
  - My understanding of the algorithm has improved
  - Possible formalisation in the future?
- It was not trivial
  - Started with the minimal implementation
  - I removed some more things that were not necessary (a parent StackItem class, for example)
  - Had to replace GOTOs by restructuring some parts of the code
  - Python does not have pointers, so I had issues with memory sharing, which I solved with deep copies
    - This affects performance, but not noticeably enough for a program which requires human interaction

DynGenPar - improvements

- The parser is missing some features that `parglare` has, like precedences and parse actions
- I want to add more ways to provide tokens to the parser, and a tokenizer (for our particular use case, this could be done with `latexwalker`)