MathSemantifier - a Notation-based Semantification Study

Toloaca Ion
Supervisor: Michael Kohlhase
Jacobs University Bremen

April 20, 2016

Abstract

Mathematical formulae are a highly ambiguous content for which typesetting systems as LaTeX store only the representation. MathSemantifier is an open-source notation-based mathematical formula semantification system that attempts to tackle the problem of ambiguity in mathematical documents and produce knowledge-rich equivalents. The system extracts formulae (from formats such as LaTeX or MathML) and produces content-rich results (sTeX or Content MathML) that contain no semantic ambiguity. The disambiguation is achieved by matching the input formulae against a known database of notation definitions, which is aggregated into a Context Free Grammar. This paper outlines an implementation of MathSemantifier that focuses on helping researchers in semantifying their works, and the ultimate goal being a scalable implementation that would need next to no help from a human, and, therefore, could be used to semantify large collections of mathematical papers such as arXiV [2].

Contents

1 Introduction

The scientific community produces a large number of mathematical papers (approximately 120 thousand new papers per year), which raises the importance of machine based processing of such documents. Unfortunately, the most popular formats in which these papers are found (for instance, LaTeX) do not contain much information that would allow the computers to infer the complex knowledge graph behind each paper. Since, at this point, changing these formats is not practically possible, the other solution is to add a semantic flavor to the existing documents by translating them into a more suitable format, for instance, Content MathML.

1.1 History and Motivation

As a system as complex as the current scientific community was created, it went through a series of evolutions in the attempt to introduce the best method of writing scientific documents. This process was highly influenced by the invention and spreading of the internet. Scientists understood the necessity of a standard that could help them write and exchange their findings in an efficient way. A lot of effort has gone into translating books into digital documents.

Now, scientists have found ways to represent their knowledge in a machine comprehensible manner, some of which are Content MathML and OMDoc [15]. These new methods do not directly store the representation of the documents. Instead, what is actually stored is the knowledge graph hidden behind the ambiguity of the representation. Naturally, these documents can be used to also generate a human readable format, examples being Presentation MathML and LaTeX.

As previously mentioned, a lot of effort went into translating books into digital documents. Since the year of 1850, there have been produced approximately 3.5 million papers, and approximately 120 thousand new papers are written every year. Now, when all these digital documents need to be converted to these semantic representation, doing it by hand is not just unreasonable, but straight out impossible.

The next step in this evolution is translating digital documents into improved digital documents, that the computers can actually understand and not just store. This next step can only happen if a new, relatively painless, way of transition appears. As soon as the ease of transition and the benefits from doing it outweigh the difficulties associated with it, the scientific community will open the door into the world where computers can actively help researchers with more than just symbolic searches.

1.2 Content MathML and Presentation MathML

The two main formats this paper is focusing on are Content MathML (CMML) and Presentation MathML (PMML) [5].

PMML is used to describe the layout and structure of mathematical notations. PMML elements construct the basic kinds of symbols and expression-building structures present in traditional mathematical notations, containing also enough information for good renderings. The last part is exactly the motivation as to why MathML alone is not enough - because it only suggests specific ways of renderings, but does not require anything.

CMML, on the other hand, is used to provide an explicit encoding of the underlying mathematical meaning of mathematical expressions. This fact is important in this context because this implies that CMML contains no ambiguity, so, by choosing the final product to be CMML, it is indeed possible to achieve meaningful semantification. For example, considered "H multiplied by e". It can be often seen to be written as He in mathematics, however, this can be interpreted also as H applied to e in the context of lambda calculus, as well as a chemical.

An example of PMML and CMML code for the expression $x + y$ is provided below. Note how the **mrow** tag is used to delimit nested expressions, while the **mo** tag indicate mathematical objects (such as arithmetic operators) and **mi** is used for variables and similar objects. This provides a clear structured format, that will be easily parsed by MathSemantifier. In particular, PMML is such a small and convenient language that even building a parser for it as a part of MathSemantifier is certainly reasonable.

In the CMML part of the example, the **apply** tag is semantically the application of its first element (the arithmetic plus) on two identifiers x and y, which altogether means simply $x + y$.

PMML

```
<mrow>
    <mi>x</mi>
    <mo>+</mo>
    <mi>y</mi>
</mrow>
```

CMML

```
<apply>
    <csymbol cd="arith1">plus</csymbol>
    <ci>x</ci>
    <ci>y</ci>
</apply>
```

An important part of CMML is that all notations (like the arithmetic plus notations used in the example above), contain specifications of how these should be rendered into PMML, so the conversion is not a problem.

1.3 Ambiguity in Mathematical Documents

An important concept necessary in order to understand why semantification is a complex process is ambiguity. Mathematical documents are not a simple collection of symbols. The main use of these documents emerges only when the knowledge graph of a document is accessible. However, humans tend to be lazy in writing down the whole graph, but instead rely on implicit human knowledge

to decipher these documents. This is where ambiguity comes into play, when the author relies on the ability of the human to use the context of document in order to pinpoint the actual meaning an expression. Ambiguities can be largely divided into two: structural and idiomatic ambiguities.

1.3.1 Structural Ambiguities

A simple example that demonstrates the concept of structural ambiguities can be $f(a(b))$. It can mean one of the following:

1. Application of function f to a times b
2. Application of function f to the application of a to b
3. Multiplication of f, a and b
4. Multiplication of f and the application of a and b

Note how such a simple example with just three identifiers has at three different possible interpretations. It is easy to notice that the number of readings has an exponential complexity in the length of the expression. For instance, the expression above will have 2^{n-1} readings, where n is the number of identifiers, which is because we can considered each application as a multiplication too.

The concept above can be generalized as structural ambiguities being associated to different readings generated from different parse trees. This means that the document may provide some direct clue about what parse tree is the best, and the readings can be characterized by parse trees.

1.3.2 Idiomatic Ambiguities

Contrary to structural ambiguities, idiomatic ambiguities are not due to different parse trees. Given one single parse tree, some formulae allow for multiple readings. A standard example would be B_n. This could be:

1. The sequence of Bernoulli numbers
2. A user defined sequence
3. The vertex of one of a series of geometric objects

In other words, the same sequence of symbols, associated with the same parse tree can lead to multiple readings. The only feasible way at the moment of solving such ambiguities is having a large notation database and ultimately asking the user to choose from a list of possible readings. This is exactly how MathSemantifier works, which means that the final product is expect to excel at solving such tasks.

1.4 Extracting Semantics from Mathematical Documents

Ambiguity is the problem, so, now, a solution for it is required. Let us recall the example from the previous section $f(a(b))$. The first interpretation of the expression a human reader is likely to come up with is the application of f to the application of a to b. Moreover, the other readings may take the human reader by surprise, as he or she would assume that the first reading is the correct one.

Let us look into why this happens. First of all, f is a symbol humans usually use for functions, moreover, the brackets around $a(b)$ could be omitted if it were a multiplication rather than an application. Next, a and b are usually used for variables, but then again the brackets are useless if it is a multiplication. To sum up, the human throws away meanings that would imply useless work or usage of symbols in an unusual way. Imagine asking a mathematician which of the expressions is more natural, fa or ab, or, to be even more extreme, "Let $\epsilon > 0$" or "Let $\epsilon < 0$". All of these can be translated to heuristics that MathSemantifier can use to improve its results, or otherwise said, minimize the amount of work the human has to do by restricting the possible meanings as much as possible.

Up until now, different heuristics used were discussed. However, mathematics has better means of finding the one true meaning of the document. Imagine the formula $a = b$. As a human, we can deduce it may mean that some two objects are equal. We still have absolutely no understanding of what those objects are, and we just assume $=$ works as a relation on any 2 objects of the same type.

Now let's add a bit to the formula $(a = b) \wedge (b = 3)$. At this point, we deduce that $a = 3$, and $=$ is a relation of numbers by applying First Order Logic to the expression. Notice how more context reduced the ambiguity of the previous expression. Also, notice how we naturally assume First Order Logic is applicable to this situation. Notice also that we are not using just heuristics, but rather we are applying a deterministic approach to find out which meanings are impossible.

Let's get to a more interesting example $f(a+b)$. Normally, we would assume this is a function applied to $a+b$, but nothing in this context can help us decide against f multiplied by $a+b$. Adding an expression like $f : \mathrm{R} \to \mathrm{R}$ would certainly help to reason against the second interpretation, because it does not typecheck.

To sum up, humans read mathematical documents bit by bit and throw away impossible interpretations until there is, hopefully, only one left. In doing so, they mainly apply two strategies - heuristics and proof by contradiction. This proposal explores the possibility of MathSemantifier doing the same to some extent.

1.5 MathSemantifier in the Context of Disambiguation

In the endeavour of extracting semantics from mathematical documents MathSemantifier uses its notation database in order to attempt to explore the space of all possible readings, and the to throw away some using heuristics, ultimately letting the user decide which reading is the best.

This makes clear the parts the final product excels at, but also the main challenges related to that.

Regarding the good parts:

- Solving both structural and idiomatic ambiguities by exploring the space of all readings
- Notation addition can be easily added to the system, since the Context Free Grammar used is generated automatically
- Enhanced overall user experience when writing or semantifying existing mathematical documents by creating a dynamic, adaptive system by means of the above mentioned strengths

The main challenges are:

- Scalability in the context of such a system translates into finding the best ways to generate as few useless parse trees as possible
- Compiling a notation database into a Context Free Grammar when including multiple notation archives simultaneously
- The response time of a query to the system is a big issue, since, as mentioned before, everything related to ambiguities has an exponential time complexity. Finding a balance between exploring the space of all readings enough in order to find meaningful results while keeping the response time low is a very difficult challenge.

The rest of this paper will demonstrate in detail how MathSemantifier harnesses its strengths and solves or avoids the challenges described.

2 State of the Art

2.1 Semantification

Nowadays extracting the semantics of mathematical documents is regarded as an interesting field, however the number of researchers in it is limited. This is caused by several domain specific problems.

First of all, mathematics is usually represented in a format where it mixed with natural language. Research on extracting mathematical expressions from such contexts has not been incredibly successful so far. Another important problem is the lack of a well prepared set of data. This kind of data could be used for Machine Learning applications as training data, or simply as test data for complex systems. As software that has not been tested cannot be called particularly useful, this poses a great problem.

2.1.1 Restricted Natural Language

There are multiple projects that try to use a subset of the natural language that is still not that ambiguous but allows for intuitive document creation. Such projects are FMathL [18], MathLang [11], MathNat [10] and Naproche [6].

Most such approaches try to analyze the context free parts of mathematical expressions. We will go into more detail about Context Free Grammar applications in semantification in the next sections. It is worthy to note though that most such projects are not satisfied by commonly found grammar engines, and some, like FMathL, are developing their own specialized parsers. This paper does not outline custom parsers, however, if further work is to be done on the subject, implementing heuristics as to reduce the disambiguation choice list could require at some point a similar approach.

2.1.2 Format Conversions

Mathematical documents in their majority are written in TEX/LATEX, but for storage and processing purposes converting them to an XML representation is a good idea. Large scale mathematical document archives such as *arXMLiv* [21] are using the LATEXML [17] converter to produce MathML and OpenMath.

LATEXML can actually already produce Content MathML, however the result produced in such a way from a knowledge poor setting contains only one of the possible readings, and, therefore, is not satisfactory from a semantic point of view as it not need satisfy the actual semantics of the document. Enhancing LATEXML's ability of converting LATEXinto knowledge rich XML format variations by exploring the space of possible readings is one of the main goal of this proposal.

2.2 Using Grammars for Semantification

2.2.1 Combinatory Categorial Grammars

The MSc Thesis of Deyan Ginev [9] studies a similar approach that uses Combinatory Categorial Grammars as a part of a larger analysis pipeline that is applied to handpicked set of mathematical texts. The results show a close to perfect recall rate with a low degree of ambiguity.

This approach, however, uses a manually crafted grammar as opposed to the generated grammar used by MathSemantifier. While writing a grammar by hand can lead to good results as mentioned above, this approach does not scale, since it is hard to find someone who will write such grammars every time a new notation is needed. Notations, on the other hand, are much faster and easier to write. In other words, scalability and extensibility are the reasons behind MathSemantifier.

2.2.2 S-graph grammars

The research of Alexander Koller of University of Potsdam exhibits an interesting way of utilizing grammars for semantic construction. S-graph grammars [16] constitute a new grammar formalism for computing graph based semantic representations. What distinguishes this line of research from the common data-driven systems trained for such purposes is that S-graph grammars use graphs as semantic representations in a way that is consistent with more classical views on semantic construction.

S-graph grammars are introduced as a synchronous grammar formalism that describes relations between strings and graphs, which can be used for a graph based compositional semantic construction, which is essentially what this paper outlines in simpler terms - using Context Free Grammars.

2.2.3 Marpa Grammar Engine

As the idea of the proposal is to use a Grammar Engine, we need to introduce a concrete solution. Text parsing using Context Free Grammars is quite popular, and, therefore, there is quite a number of solutions, which can, however, make the choice of the best grammar engine for a particular purpose difficult. After comparing several such tools, the conclusion reached was that the Marpa Grammar Engine [12] stands out as a flexible, powerful and efficient tool suited for the purpose of semantification.

To understand better the claim about efficiency, below a comparison between multiple regex parsers and Marpa made by the creator of Marpa is provided.[14]

Number of parens in test string	Executions per second				
	libmarpa (pure C)	Marpa::XS (mixed C and Perl)	Regexp:: Common:: balanced	tchrist's regex	Marpa::PP (Pure Perl)
10	4524.89	111.71	3173.30	33429.33	47.39
100	1180.64	58.96	62.09	197.25	15.35
500	252.40	19.50	2.43	7.58	4.09
1000	117.16	10.28	0.53	1.84	2.14
2000	56.07	5.47	0.12	0.34	1.08
3000	36.35	3.72	0.05	0.13	0.74

The results are presented in executions per second. We can see that standard solutions may be faster for a small nesting level, but they go quadratic as it rises.

Another important point is that Marpa Grammar Engine has a quite simple interface, allowing for extensive manipulations during the parsing process. Each rules can be given an "Action" - a Perl routine - that can define what the engine is supposed to do when the rules is used.

As it was mentioned in the previous sections, the grammar generated will most likely be highly ambiguous, and the Marpa Engine allows for ambiguous grammar of any kind that could be required for this application.

2.3 Extracting formulae from mathematical documents

As MathSemantifier as described in this paper need not be restricted to processing just MathML formulae, but also mathematical documents written in LaTeX, it is natural that a method of extracting MathML formulae from mathematical documents is provided. A simple way of achieving that would be by using the same method as MathWebSearch [20] is using in order to extract MathML formulae from documents containing MathML formulae and other information. MathWebSearch accepts MwsHarvest [4] as crawled data. These harvests are in MathML, therefore perfectly suited for this application.

3 Preliminaries

3.1 The Notation Database of MMT

MathSemantifier uses a Notation Database to create the grammar it uses. Therefore, it needs to be explained where those notations come from. MMT [19] (Module system for Mathematical Theories) is a language developed as a scalable representation and interchange language for mathematical knowledge. It permits natural representations of the syntax and semantics of virtually all declarative languages. The decisive factor about MMT is that there is already a large database of notations written in sTeX, which is transformed and stored in MMT in an original format that MathSemantifier is processing in order to generate a Context Free Grammar.

SMGloM [8] is a part of the notation database of MMT that is especially relevant to the MathSemantifier. It contains notations from vastly different topics such as:

- Algebra
- Calculus
- Geometry
- Graphs
- Number fields
- Topology
- Set Theory

3.2 MMT Notation Storage Format

The original format that MMT uses to store the notations is of high relevance, since it the raw material from which the Context Free Grammar is generated. The format consists of a Scala **case class** tree. Several examples are provided below.

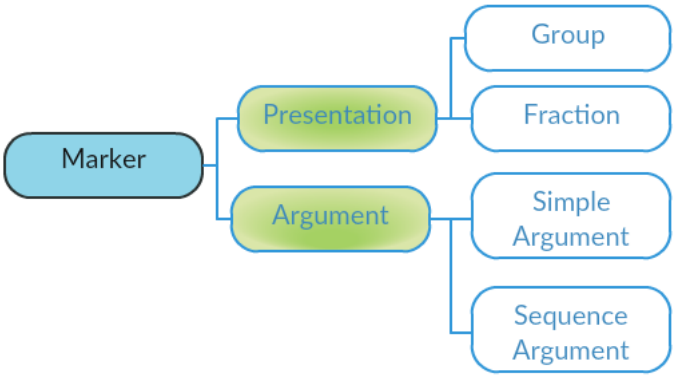


Figure 1: MMT Notation Markers (severely simplified)

Since there are about 40 types of different Markers, the tree above is meant to simply give an idea of the structure, rather than explain it fully. The **Marker** class is the tree of all possible notation components. The most basic types of Markers include **Presentation Markers** and **Argument Markers.** They have a great diversity of subtypes. **Simple Argument** is used for single arguments, while **Sequence Argument** is a placeholder for a sequence with a specified delimiter. The most basic example of a sequence argument is, if considering the example $2 + 3 + 5$, the sequence $2, 3, 5$ and the delimiter $+$.

Presentation Markers have yet another purpose of simulating the PMML structure. **Group Marker** stands for the **mrow** MathML tag, **Fraction Marker** - for **mfrac** and so on. MMT has Markers for every possible MathML tag, which goes to explain the total number of Markers to an extent.

3.3 LaTeXML Pre-processing

Since sTeX, as an extension of LaTeX, is just as complex to parse, MMT uses LaTeXML in order to convert sTeX to OMDoc [15]. OMDoc is then processed and stored in MMT.

Below is an example of the arithmetic plus notation written in sTeX and the equivalent in MathML.

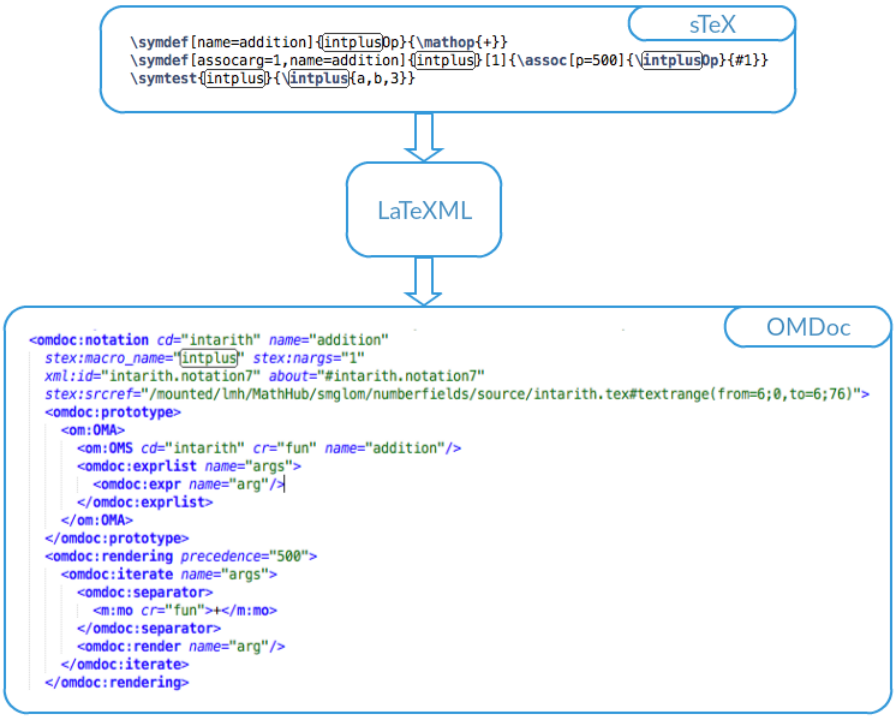


Figure 2: sTEX to OMDoc conversion

3.4 Marpa Context Free Grammar Parser

MathSemantifier will take the an auto-generated CFG and input from the **Web UI** (discussed in more details in further sections) in order to produce possible parse trees of the input. Marpa Grammar Engine is the proposed. It was already discussed above, so let us summarize the important points that matter for this particular application.

Marpa is[13]

- Fast - it parses grammars in linear time. This is critical since the size of the grammar is likely to be at least linear in the number of notations
- Powerful - it can parse left, right and middle recursions. Since it is unclear what kind of recursions can occur in generated grammars, and it would be a pity if the parse tree generator would need to be changed midway because of it, it is better to choose something that can handle all kinds of recursion
- Convenient - Parser generation is unrestricted and exact, all that needs to be provided is a CFG in BNF form. Also, if several alternatives may

yield a parse, all of them are considered. This is critical, since this directly implies that it can deal with ambiguous grammars

- Flexibility - it provides control over the parsing process, giving out information about which rules have been recognized so far and their locations. This is again extremely important because otherwise, once again, semantification would be simply impossible

Obviously, since the CFG will be ambiguous this is the most computationally intensive part. The scalability of the final product will mostly depend on the efficiency of this step.

As a side note, Marpa does not have a Scala API, so, instead, the Perl API will be used. This implies that the result parse trees will need to be transported somehow to the core application in MMT. For that purpose the intended solution is LWP [3]. LWP is a set of Perl modules which provide a simple and consistent API to the World Wide Web. After experimenting with the API, the conclusion was that the API is simple and powerful enough for the current purpose.

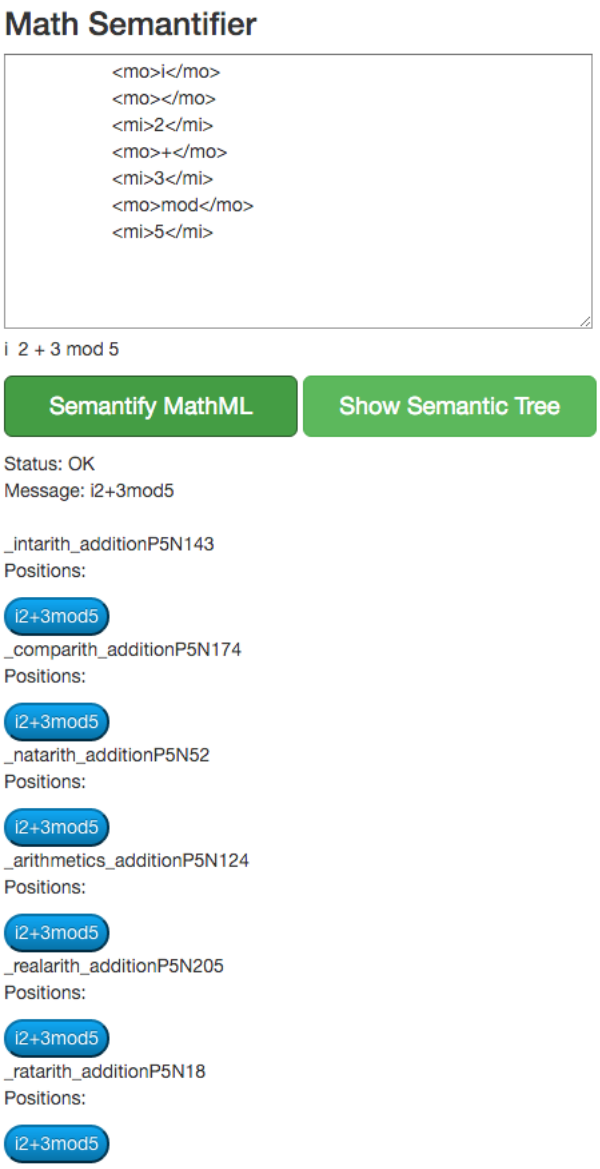
4 Use Cases

MathSemantifier has two major use cases.

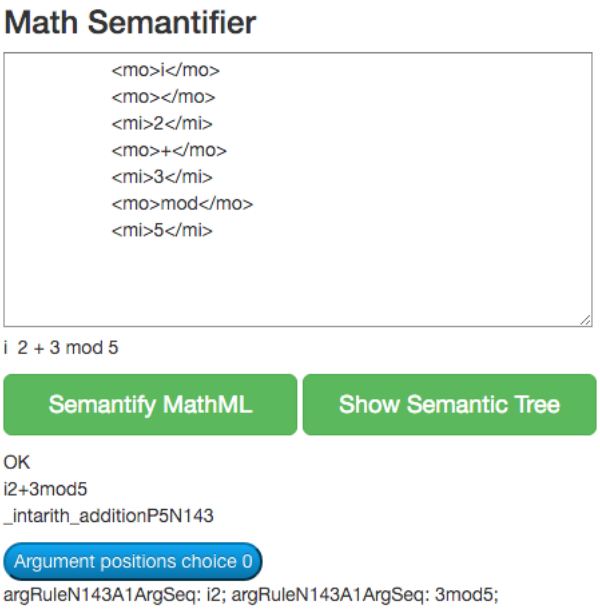
- The user can guide the semantification of the top symbol directly, by choosing the correct matching range, notation and argument positions.
- The other option is to ask for all the possibilities and get all the semantic trees.

4.1 User Guided Semantification

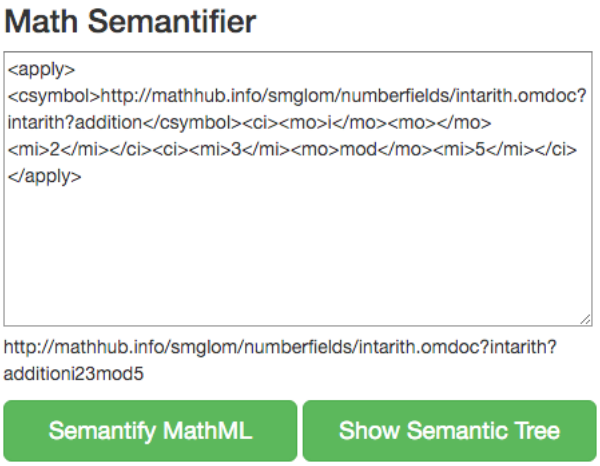
The user provides Presentation MathML as input to the system, then uses the **Semantify** button to reveal a list of top level notations as shown below.



After determining which notation is the correct one, the user needs to make sure that the arguments were detected properly.



Finally, the resulting Content MathML will be displayed.



Note that while the workflow is certainly not ideal, this study is intended to be a proof of concept and an aid to further similar research.

4.2 Semantic Tree Generation

The easier but computationaly more expensive alternative is to simply generate all the possible parse trees at once and display them. The user simply needs to click the **Show Semantic Tree** option.

Math Semantifier

```
<mo>i</mo>
<mo></mo>
<mi>2</mi>
<mo>+</mo>
<mi>3</mi>
<mo>mod</mo>
<mi>5</mi>
```

i 2 + 3 mod 5

Semantify MathML | Show Semantic Tree

Reading 1)
<apply id="11" ><csymbol>http://mathhub.info/smglom/numberfields/comparith.omdoc?comparith?addition</csymbol><ci>
<apply id="2" ><csymbol>http://mathhub.info/smglom/numberfields/comparith.omdoc?comparith?multiplication</csymbol><ci>
<csymbol id="1" >http://mathhub.info/smglom/numberfields/complexnumbers.omdoc?complexnumbers?imaginary-unit</csymbol></ci><ci><mi>2</mi></ci></apply></ci><ci><apply id="9" >
<csymbol>http://mathhub.info/smglom/numberfields/intarith.omdoc?intarith?mod</csymbol><ci><mi>3</mi></ci><ci>
<mi>5</mi></ci></apply></ci></apply>

Reading 2)
<apply id="57" ><csymbol>http://mathhub.info/smglom/numberfields/arithmetics.omdoc?arithmetics?addition</csymbol><ci>
<share href="#7"/></ci><ci><share href="#8"/></ci></apply>

Reading 3)
<apply id="37" ><csymbol>http://mathhub.info/smglom/numberfields/natarith.omdoc?natarith?addition</csymbol><ci><share href="#3"/></ci><ci><share href="#8"/></ci></apply>

Reading 4)
<apply id="52" ><csymbol>http://mathhub.info/smglom/numberfields/arithmetics.omdoc?arithmetics?addition</csymbol><ci>
<share href="#4"/></ci><ci><share href="#9"/></ci></apply>

Reading 5)
<apply id="19" ><csymbol>http://mathhub.info/smglom/numberfields/comparith.omdoc?comparith?addition</csymbol><ci>
<apply id="6" ><csymbol>http://mathhub.info/smglom/numberfields/intarith.omdoc?intarith?multiplication</csymbol><ci><share href="#1"/></ci><ci><mi>2</mi></ci></apply></ci><ci><apply id="9" >
<csymbol>http://mathhub.info/smglom/numberfields/intarith.omdoc?intarith?mod</csymbol><ci><mi>3</mi></ci><ci>
<mi>5</mi></ci></apply></ci></apply>

For this particular example, there is a total of 72 different readings. This can be easily explained, since **Invisible Times** and **Arithmetic Plus** have each 6 different notations, and **Mod** has 2. $6 \cdot 6 \cdot 2 = 72$ explains exactly where the 72 readings come from, since each notation can be combined with any other notation without any restrictions.

5 Implementation

5.1 Project Goals and Challenges

The main goals of the project are:

- Generating the correct set of parses efficiently and effectively
- Providing opportunities for improvement for further research in the area

The challenges on the way of achieving a way of generating dynamically the full set of semantic parse trees are:

- Aggregating the mathematical notation database that the MMT system into a CFG
- Dealing with scalability issues which are entailed by ambiguity
- The set of correct parses should be a subset of the set of generated parses
- The set of generated parses should ideally be a subset of the set of correct parses too
- Handle character encodings correctly

This following subsections how **MathSemantifier** achieves the above goals.

Section 6 shows to what extent the challenges described are overcome and how.

5.2 **MathSemantifier** Architecture

The architecture can be roughly divided into four parts:

1. **Web UI**
2. **MMT Backend**
3. **Parsing Backend**
4. **MMT Notations DB**

The diagram below shows the subparts of each components. The components will be discussed in more details in the further subsections.

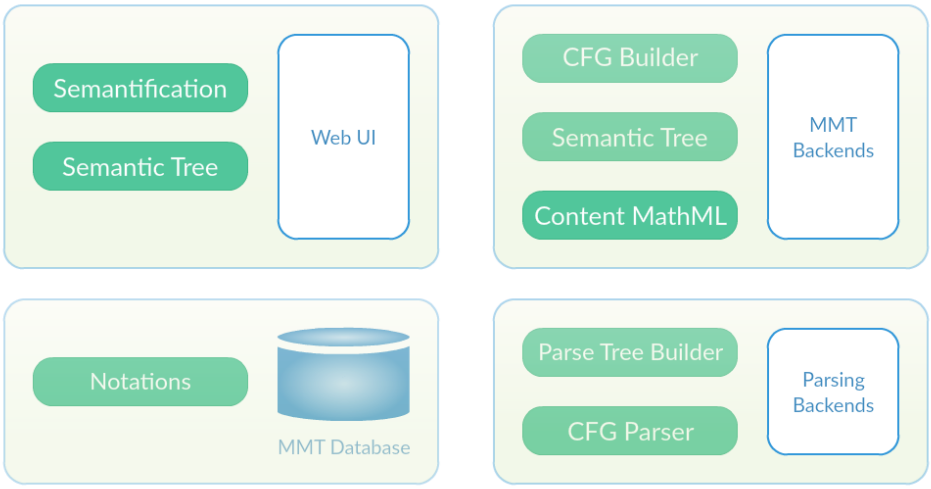


Figure 3: **MathSemantifier** Architecture

5.3 Component communication

Regarding the communication between parts:

- MMT Backends and MMT Notation DB are both part of the MMT Server Application
- All the other connections are implemented using the **REST** protocol, using simple **POST** requests

This also implies that the all the different components are not intended to run on one machine, especially the **Web UI** and the reset of the components.

The frontend / backend separation reveals opportunities for distributed solutions of such a system, in case scalability becomes an issue.

5.4 Web User Interface

The **Web UI** is the simplest of the components of **MathSemantifier.** It is intended to be a lightweight solution that queries a server for the results of more computationally intensive tasks.

The interface consists of an input area, where **MathML** needs to be inputted, and two options:

1. Semantify
2. Show Semantic Tree

Since the second option is technically more powerful that the first one, since it generates all the possibilities at once, it is perfect to demo the results the system can give.

A series of example in **MathML** is provided to show how the number of produced parse trees scales when varying the length and complexity of the input.

5.5 MMT Backend

The MMT Backend is a **Server Extension** that is part of MMT.

It is the core of the system, that assembles the results from all the other parts together into **Semantic Trees.**

Its subcomponents are discussed below.

5.5.1 Context Free Grammar Generator

The Grammar Generator aggregates all the knowledge contained in the MMT Notations into one Context Free Grammar. The grammar will be shaped into the normal form accepted by the Marpa Grammar Engine. To achieve this, the format used to store the notations in MMT will need to be decomposed into CFG rules. Otherwise said, the tree-like structure of each formula, that is stored as nested applications of **MMT Markers** (discussed previously) needs to be serialized into CFG rules.

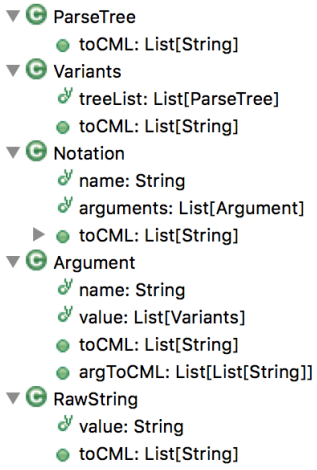
This is done in several steps:

1. Break apart the **MMT Marker** trees into level by level representations
2. Transform the intermediate representation into valid CFG rules
3. Optimize if possible (will be discussed in detail in the subsequent sections)

5.5.2 Semantic Tree Generator

The **Semantic Tree Generator** works by recursively querying the **Parsing Backend** and using the result to construct a the tree of possible meanings.

The parse trees stored in MMT have the following structure:



This simple structure allows for a flexible way of parse tree storage.

- **Variants -** represents a list of possible readings. It will always be the top node in any parse tree.
- **Notation -** a notation detected in the input. It contains its name and arguments.
- **Argument -** an argument of a notation. The plugin is recursively called on it to construct its meaning subtree as well.
- **RawString -** the ground term representation

This structure is important because, in case another application wants to access the parse trees and process them, this structure will need to be dealt with.

5.5.3 Content MathML Generator

The final part of any of the semantification processes is converting the internal represenation to a standard one, which is CMML in this case. MMT provides a simple API which requires:

1. The notation path
2. The argument maps

Both of which are available in my representation structure. The argument path is obtained by extacting the argument number from the argument name, and looking up in a map of paths created at the time of grammar creation. This implies the grammar rule names are overloaded with meaning, however, the possibilities are very limited in this aspect since the parsing framework used does not give complete control over the parsing process.

5.6 Parsing Backend

The parsing backend consists of two parts.

5.6.1 Context Free Grammar Parser

First of all, the the CFG needs to be queried and parsed. This is implemented using lazy evaluation, which means that it is only done when a request actually comes.

The serialized CFG is unpacked and feed to the Marpa Parser Generator.

5.6.2 Parse Tree Generator

The more complex of the two parts is actually going throught the parse trees and extracting useful information. Note that going through all the parse trees is not practical, so only the first N (currently 1000) parse trees are processed. This still gives the correct results in most cases since the grammar rules are optimized for giving preference to parse trees that are more likely to be correct (discussed in more detail in section 6).

5.7 Encoding issues

Since **MathSemantifier** deals with unicode, and has parts written in Javascript, Scala and Perl, which treat unicode symbols differently, it needed a solution for this problem. Javascript and Scala have **UTF-16** strings, while Perl has **UTF-8** strings. The implemented solution simply passes around the string encoded using **encodeURIComponent** in Javascript or its equivalents in other languages, and only the parsing backend in Perl actually deals meaningfuly with substrings. The result of the parsing backend contains the complete subtrings for the matched rules and arguments. This also represents an opportunity for improvement, since passing around positions in a string is more efficient that substrings.

6 Optimizations and Heuristics

This section describes the attempts made in order to deal with the overwhelming ambiguity of mathematical notations and produce actual results.

1. Only notations on the whole input are matched. Subterms are matched directly by the **Semantic Tree** plugin.

2. The number of parse trees is limited to $N = 1000$. The number of parse trees was empirically determined to include the set of correct parses for the examples used (see the **Evaluation** section**).**

3. The non-empty alternatives in the CFG are put first. The Grammar Parser has a predictable behaviour of going through the alternatives left

to right, and since empty rules match **always**, it is imperative that other alternative are attempted first. Note that only after this optimization any kind of results were possible to achieve on non-trivial input.

4. The CFG includes precedences correctly. This weeds out a large number of incorrect readings.

5. Sub-term sharing is part of the **MathML** standard, and within one request, the response will try to share its terms as much as possible in order to compress the output.

6. **The Semantic Tree** plugin makes use of memoziation to reduce one dimension of the exponential blowup as explained on the following example. $2 + 3$ has 6 possible readings, because $+$ is defined 6 times, between natural, integer, real numbers and so on. However, 2 and 3 - these subterms do not contain any notations, yet, the plugin will need to know that for every possibility of $+$. Memoization allows to omit a large number of **POST** requests even in this simple example, which results into a signficant speedup of a factor of approximately 4 (13 **POST** requests vs 3 **POST** requests). Also, slight modification of the input benefit of a signifcant speed-up, as well as using a previous input as a subterm in a new input.

7 Further work

7.1 Suggestions for further optimizations

1. The CFG could be checked for unused rules, and, more importantly, converted to BNF, for instance

2. Theory based optimization - either let the user specify which theories to create the grammar from, or take into consideration that notations from the same theory are likely to be close in the input (for instance, $-$ and $+$ are more likely to both be used as arithmetic sybmols in an expression, than only one of them)

3. Sequence arguments (for instance, $2 + 3 + 4 + 5 + 6$) currently generate a very high number of possible parses. If more control over parsing were possible, parsing of sequence arguments should be greedy - attempting to take in as many delimiter argument pairs as there are.

7.2 Requirements for a more suitable Notation Database

1. The most important optimization would be adding types to the notation input arguments and output. This would allow for type based parsing, which would certainly be more efficient and generate more relevant results

2. Grouping similar notations within a theory. For example, arithmetic plus on natural, integer or real number should be possible to connect somehow.

If the notations have types, the it will naturally occur by grouping the notations with the same types into single rules.

3. For the **csymbol** used in an **apply** tag, the notation should cross-reference it with the part of the representation it corresponds to, since it is not otherwise clearly specified.

7.3 Requirements for a more suitable Parsing Framework

1. Full control over the parsing process is a requirement for further work
2. The ability to handle types, precedences and associativity
3. The ability to handle greedy parsing - most CFG parsing frameworks have the counted rule $Expression ::= Expression+$, which has a similar meaning to the + found in a **regex.** However, **regexes** actually will do a greedy match, the more appropriate match being then $Expression+?$, the non-greedy version. This is critical to reduce the number of useless parse trees when dealing with Sequence Arguments

8 Evaluation (Incomplete)

This section presents an evaluation of **MathSemantifier** from the point of view of how effective and efficient both guided semantification and semantic tree generation are. Next, the interoperability of the system with other possible applications is examined, which mostly depends on the backend APIs.

8.1 Guided semantification

The user interface for top level symbol guided semantification is simple and intuitive, and it detects all the possible readings for the listed examples.

8.2 Semantic Tree Generation

For the provided list of examples of depth 2-3, all the possible parse trees are produced.

8.3 Performance of the Semantic Tree Generation

It is important to note that there are two kinds of figures we may want to look at: one run or several subsequent related runs. Because of memoization, in the long run, if the terms or evens just subterms repeat, the queries become signficantly faster.

8.4 The backend APIs

It is important that the MMT backend provides a simple **HTTP** endpoint, which can be accessed with a **POST** request that contains just the input in a suitable encoding. The backend will then return a **JSON** object that contains all the possible readings. The backend could be modified to accept parameters to fine tune the system, like the maximum number of processed parse trees, but even then accessing it would be just as simple.

9 Conclusions

The conclusion of this study is the proof of concept architecture and implementation of a system capable of converting **Presentation MathML** to all the possible meanings, which is a list of **Content MathML**. The testing revealed that the system is able to recognize correctly single top level symbols, as well as the whole set of readings of expressions of nesting depth up to 3 (this is not the limit, but not enough testing in section 8 is done beyond that). This shows that it is certainly possible to aggregate the knowledge from the MMT Notations and to use it for parsing purposes. However, both the Notations and the Parsing Framework need significant improvements in order for the system to be scalable beyond what is presented in the previous section. The most important part of this study is, therefore, the optimizations and heuristics used, and other ones that further research could benefit from.

References

[1] Apache Camel, URL: http://camel.apache.org/, seen November 2015. 2015.

[2] ArXiV Online. *URL, as seen November 2015, http://arxiv.org*, 2015.

[3] LWP - The World-Wide Web library for Perl, URL: http://search.cpan.org/dist/libwww-perl/lib/LWP.pm, seen November 2015. 2015.

[4] MwsHarvest, URL: https://trac.mathweb.org/ MWS/wiki/MwsHarvest (seen November 2015). 2015.

[5] Ron Ausbrooks, Stephen Buswell, David Carlisle, Stephane Dalmas, Stan Devitt, Angel Diaz, Max Froumentin, Roger Hunter, Patrick Ion, Michael Kohlhase, Robert Miner, Nico Poppelier, Bruce Smith, Neil Soiffer, Robert Sutor, and Stephen Watt. Mathematical Markup Language (MathML) version 2.0 (second edition). *W3C recommendation, World Wide Web Consortium (W3C)*, 2003.

[6] Marcos Cramer, Peter Koepke, and Bernhard Schroder. Parsing and disambiguation of symbolic mathematics in the naproche system.

[7] Catalin David, Constantin Jucovschi, Andrea Kohlhase, and Michael Kohlhase. SAlly:A Framework for Semantic Allies.

[8] Deyan Ginev, Mihnea Iancu, Constantin Jucovschi, Andrea Kohlhase, Michael Kohlhase, Heinz Kroger, Jurgen Schefter, and Wolfram Sperber. The SMGloM Project and System.

[9] Deyan Ginev, Michael Kohlhase, and Magdalena Wolska. The Structure of Mathematical Expressions. 2011.

[10] Muhammad Humayoun and Christophe Raffalli. Mathnat - mathematical text in a controlled natural language. *Special issue: Natural Language Processing and its Applications*.

[11] Fairouz Kamareddine, Manuel Maarek, and Joe B. Wells. MathLang: An experience driven language of mathematics. *Electronic Notes in Theoretical Computer Science 93C*.

[12] Jeffrey Kegler. The Marpa parser. Web Manual at https://jeffreykegler.github.io/Marpa-web-site/, seen November 2015. .

[13] Jeffrey Kegler. The Marpa Parser. https://jeffreykegler.github.io/Marpa-web-site/, . [Online; accessed 19-April-2016].

[14] Jeffrey Kegler. Marpa v. Perl regexes: some numbers. http://blogs.perl.org/users/jeffrey_kegler/2011/11/marpa-v-perl-regexes-some-numbers.html, 2011. [Online; accessed 19-April-2016].

[15] Michael Kohlhase. OMDoc - An open markup format for mathematical documents (Version 1.2).

[16] Alexander Koller. Semantic construction with S-graph grammars.

[17] Bruce Miller. LaTeXML: A LaTeX to XML converter. Web Manual at http://dlmf.nist.gov/LaTeXML, seen November 2015.

[18] Arnold Neumaier and Peter Schodl. A framework for representing and processing arbitrary mathematics. *In Proceedings of the International Conference on Knowledge Engineering and Ontology Development*.

[19] Florian Rabe. The MMT API: A Generic MKM System.

[20] Hambasan Radu, Corneliu C. Prodescu, and Michael Kohlhase. MathWebSearch at NTCIR-11. 2014.

[21] Heinrich Stamerjohanns, Michael Kohlhase, Deyan Ginev, Catalin David, and Bruce Miller. Transforming large collections of scientific publications to XML. *Mathematics in Computer Science*, 2010.