Bachelor Thesis

Declaration Spotting in Mathematical Documents

Jan Frederik Schaefer
Supervisor: Prof. Kohlhase

May 8, 2016

Abstract

Automated understanding of mathematical documents is an ambitious goal. One approach to this problem are spotters. Spotters specialize in one particular kind of language phenomenon and search documents for all occurrences of this phenomenon. The results can be stored in a database and reused by future spotters. Eventually, this should lead to a good coverage of the language used by mathematicians, which could make automated understanding possible.

One of the interesting phenomena in modern mathematical documents are variables. Often, they are introduced in declarations such as "*Let p be a prime number*". Our goal is the implementation of a spotter that detects such declarations and extracts the information that "*p*" has been declared as a "*prime number*".

This information could for example assist search engines in finding applicable theorems. In the long term, another application could be proof verification tools, which require very comprehensive understanding of mathematical documents.

In this thesis, we will analyze how variables are declared; we will describe a framework for the implementation of spotters, and finally present a simple declaration spotter, which can be used as a base-line.

Contents

1 Introduction

Our goal is the implementation of a spotter for declarations in mathematical documents.

Declarations are a way of introducing variables in mathematical documents. For example in the sentence "*Let p be a prime number*", "p" is declared as a "*prime number*". We call "p" an *identifier* and the fact that it is a prime number a *restriction* of "p". A more detailed analysis of declarations can be found in section 3. For simplicity, some of the examples are constructed. A long list of real-world examples can be found in appendix A.

1.1 Motivation

One of the key requirements for semantic search in mathematical documents is an understanding of the mathematical formulae it contains. This is only possible if we understand the meaning of the symbols in a formula. A corpus study by Magdalena Wolska and Mihai Grigore [WG10] suggests that many symbols are indeed explicitly declared. Let us consider the following formula:

$$G \cong \langle \mathbb{Z}, + \rangle$$

With a simple heuristic we could probably guess that "G" should be a group. However, the formula only holds if "G" is infinite and cyclic. In a document we could, for example, find the following theorem:

Let G be an infinite cyclic group. Then $G \cong \langle \mathbb{Z}, + \rangle$

The spotter would extract the information that "G" is an "*infinite cyclic group*". This additional information is very useful. Let's say, for example, that we have some finite group and are searching for applicable theorems. Knowing that the formula is only applicable to *infinite* groups, a search engine could immediately discard this result.

Of course it is possible to manually annotate all symbols in a document, but it is a rather tedious and time-consuming task and could hardly be applied to the existing, huge corpora of mathematical documents. With an automated approach we could make them more accessible.

Semantic search in mathematical documents is a great application for a declaration spotter, but there are other, more ambitious applications. It brings us a small step closer to being able to generate formal representations of mathematical documents. Then we could, for example, transform informal proofs into a formal representation and use proof verification tools to verify their correctness.

Claus Zinn describes in his PhD thesis [Zin03] a framework for the verification of informal proofs. He uses Discourse Representation Theory (DRT) [Kam81] for a more formal representation of the informal proofs. While he provides many examples of sentences and their DRT representation, he does not provide a practical way to do this conversion automatically. In my opinion, mathematical vernacular is very complex and cannot be covered with a few hand-written rules. A more practical approach is to use spotters that are specialized in one

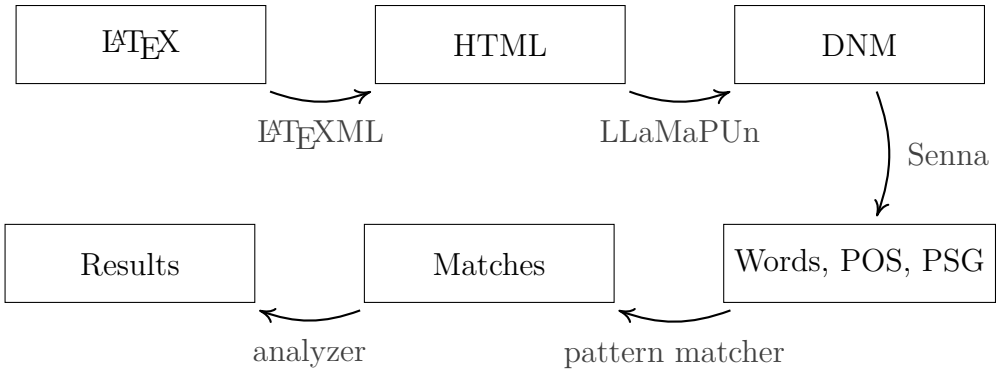


Figure 1: The document pipeline

particular kind of statements. Spotters could reuse results obtained from other spotters in order to detect more complex kinds of statements. Our long-term goal is to cover a large part of mathematical vernacular using many specialized spotters, making it practically feasible to generate a formal representation of the documents. This way, many existing proofs could be verified without the tedious work of manually formalizing them. Having a formal representation of the documents is of course a requirement for many other amazing applications too, such as the generation of a knowledge database. A different approach has been used by the MathLang project [KWZ08], which supports mathematicians in adding semantic information to their documents.

1.2 The Pipeline

In this section I will briefly mention the different processing steps, as shown in figure 1, before I will go over them in more detail. We start with TeX/LaTeX documents. In my experiments I will use documents from the Cornell e-print arXiv [ArX]. Since parsing TeX/LaTeX files is very difficult, the documents get converted into an HTML representation using LaTeXML (section 2.1).

For natural language processing (NLP) it is very convenient to have a plaintext representation. The LLaMaPUn library (section 2.2) provides tools for generating a Document Narrative Model (DNM) for this purpose. It contains a suitable plaintext representation of the HTML and mappings to the DOM.

We use Senna (section 2.3) to obtain part-of-speech (POS) tags and parse trees. With this information, we can run a pattern matcher (section 4) over the documents to find language patterns that correspond to declarations.

At last, we extract potentially declared identifiers and decide which identifiers have been declared in which declaration (section 5). Currently, we only visualize the results in KAT (section 2.4). In the long term, we want to store the results in a database as envisioned in [GJA+09].

Example

Before going into the details, let us consider a trivial real-world example from [10] to illustrate the steps of the pipeline. The LaTeX source contains the sentence

```
Let $r\geqslant 2$ be an integer.
```

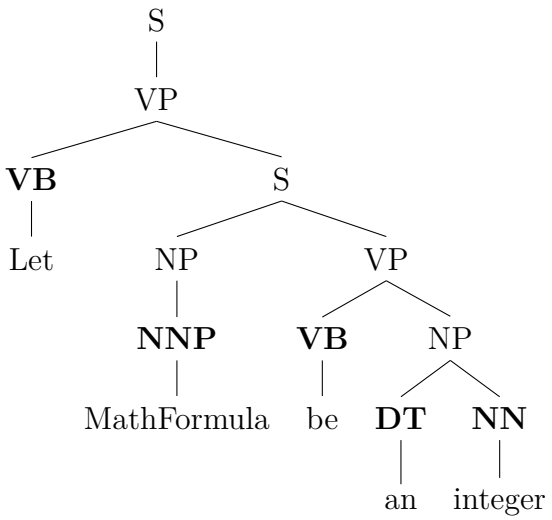
Using LaTeXML we get the following HTML representation:

```
<p xmlns="http://www.w3.org/1999/xhtml" id="p1.1" class="ltx_p">
    Let
    <math xmlns="http://www.w3.org/1998/Math/MathML" id="p1.1.m1.1" class="ltx_Math" alttext="r\
        geqslant 2" display="inline">
        <semantics id="p1.1.m1.1a">
            <mrow id="p1.1.m1.1.4" xref="p1.1.m1.1.4.cmml">
                <mi id="p1.1.m1.1.1" xref="p1.1.m1.1.1.cmml">r</mi>
                <mo id="p1.1.m1.1.2" xref="p1.1.m1.1.2.cmml">â‰¥</mo>
                <mn id="p1.1.m1.1.3" xref="p1.1.m1.1.3.cmml">2</mn>
            </mrow>
            <annotation-xml encoding="MathML-Content" id="p1.1.m1.1b">
                <apply id="p1.1.m1.1.4.cmml" xref="p1.1.m1.1.4">
                    <geq id="p1.1.m1.1.2.cmml" xref="p1.1.m1.1.2"/>
                    <ci id="p1.1.m1.1.1.cmml" xref="p1.1.m1.1.1">ðİŚ§</ci>
                    <cn type="integer" id="p1.1.m1.1.3.cmml" xref="p1.1.m1.1.3">2</cn>
                </apply>
            </annotation-xml>
            <annotation encoding="application/x-tex" id="p1.1.m1.1c">r\geqslant 2</annotation>
        </semantics>
    </math>
    be an integer.
</p>
```

Next, the HTML gets loaded into the LLaMaPUn library. It creates a DNM containing a plaintext representation of the HTML. In this case, it could for example be

```
Let MathFormula be an integer.
```

Note that we replaced the `<math>` node by "*MathFormula*". Next, we use Senna to obtain POS tags and a parse tree for the plaintext. In this case, we would get



The pattern matcher should find in this case that this sentence might be a declaration, where "*an integer*" is a restriction of an identifier in the formula. At last, the analyzer would look into the formula and determine that "r" is the declared identifier and that it is additionally restricted by the condition "$r \geqslant 2$".

2 Preliminaries

2.1 LaTeXML

LaTeXML [Mil] can be used to convert TeX/LaTeX documents into HTML files. It aims at preserving both semantics and the visual representation of the original documents. Formulae are represented with MathML [ABC+10]. To give the reader an idea of the LaTeXML output, let us consider the following LaTeX snippet:

```
\newtheorem{theorem}{Theorem}
% ...
\begin{theorem}
    Let $A$ and $B$ be sets. Then $A \cup B = B \cup A$.
\end{theorem}
```

The corresponding (truncated) LaTeXML output would be:

```
<h6 xmlns="http://www.w3.org/1999/xhtml" class="ltx_title ltx_runin ltx_font_bold ltx_title_theorem">
  <span class="ltx_tag ltx_tag_theorem">Theorem 1</span>.
</h6>
<div xmlns="http://www.w3.org/1999/xhtml" id="Thmtheorem1.p1" class="ltx_para">
<p id="Thmtheorem1.p1.1" class="ltx_p">
  <span id="Thmtheorem1.p1.1.1" class="ltx_text ltx_font_italic">
    Let
    <math xmlns="http://www.w3.org/1998/Math/MathML" id="Thmtheorem1.p1.1.1.m1.1" class="
        ltx_Math" alttext="A" display="inline">
      <semantics id="Thmtheorem1.p1.1.1.m1.1a">
```

```
        <mi id="Thmtheorem1.p1.1.1.m1.1.1" xref="Thmtheorem1.p1.1.1.m1.1.1.cmml">
          A
        </mi>
        <annotation-xml encoding="MathML-Content" id="Thmtheorem1.p1.1.1.m1.1b">
          <ci id="Thmtheorem1.p1.1.1.m1.1.1.cmml" xref="Thmtheorem1.p1.1.1.m1.1.1">
            ðlŘt
          </ci>
        </annotation-xml>
        <annotation encoding="application/x-tex" id="Thmtheorem1.p1.1.1.m1.1c">
          A
        </annotation>
      </semantics>
    </math>
    and
    ...
  </span>
</p>
```

2.2 LLaMaPUn Library

The LLaMaPUn (Language and Mathematics Processing and Understanding) library [GJA+09] contains many tools required for natural language processing of mathematical documents. Its current version is implemented in the Rust programming language [MK14], which makes it very fast and light-weight. We have a specialized library, because mathematical vernacular is different from normal English. Several attempts have been made to analyze it (see e.g. [Gan09, Wol13, Zin03]). In particular, it is interspersed with mathematical formulae, which can have various functions in a sentence. Another, related problem is that most conventional natural language processing tools cannot deal with the complex HTML files generated by LaTeXML and require simple plaintext as input without any formulae.

The dnmlib, which is part of the LLaMaPUn library, tackles this problem by generating a Document Narrative Model (DNM).

The DNM

A DNM contains a plaintext representation of the HTML file and supports mappings between the plaintext and the original Document Object Model (DOM).

The plaintext generation is very configurable. It supports among other things unicode normalization and word stemming. Also, it can be specified how certain nodes should be handled. In our case, for example, we want to replace all `<math>` nodes with a string like "*MathFormula*". Also, we want to skip the bibliography.

For example, the following LaTeX snippet

```
Let $x, y \in \mathbf{R}$ with $x, y \ge 0$,
then we have $\frac{x+y}{2} \ge \sqrt{xy}$ \cite{lorem:ipsum}.
```

which corresponds to the following (shortened) HTML

Let $...$ with $...$, then we have
$...$ <**cite** ...></**cite**>.

could be transformed into the following plaintext:

Let **MathFormula** with **MathFormula**, then we have **MathFormula CitationElement**.

As a light-weight data structure, we introduced so-called *DNMRanges* that mark ranges in a DNM. During DNM generation, we essentially recurse through the DOM and generate plaintext for the nodes we encounter. In this process, we can keep track of the offsets and store for each node the range it corresponds to. This provides us with an easy node to DNMRange mapping. Mapping DNMRanges back to the original DOM has not been implemented yet (as there was no demand for it so far). It can be done by storing for each plaintext character the node it corresponds to and, in the case of text nodes, the offset. The tricky part is that unicode normalization and word-stemming can add or remove characters. In the future, we hope to have an infrastructure that allows us to serialize DNMRanges in an easy way so that we can use them to store our annotations.

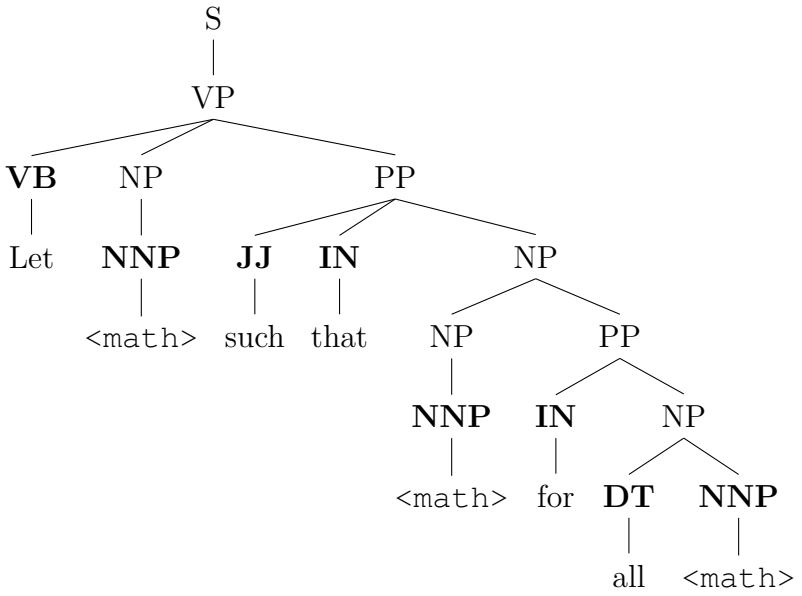
2.3 Senna

I am not aware of any good part-of-speech (POS) taggers that are trained on mathematical documents, though attempts have been made. For example, [SS14] uses the Stanford POS tagger [MSB+14] by replacing math with random strings. Mathematical vernacular is quite different from general English, which makes the usage of conventional NLP tools rather error-prone. We decided to use Senna [RCK11, Col11] for POS tagging and syntactic parsing, as it is very light-weight and could easily be integrated. In my opinion, it works surprisingly well with the DNMs of our documents.

For example,

Let $e \in G$ *such that* $ex = x$ *for all* $x \in G$.

would give the following result (with the period omitted):

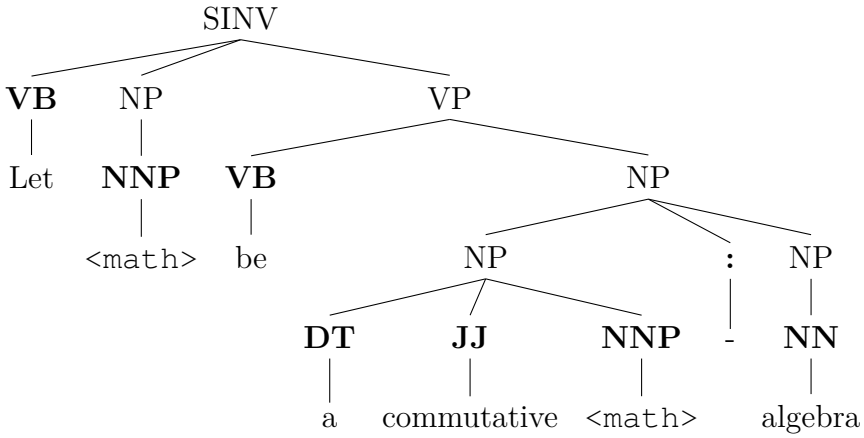


which provides us useful information about the structure of the sentence. An example where senna fails, would be

Let A be a commutative $\mathbb{Q}$*-algebra*

.

Here, we would get



Again the period is omitted for compactness. This particularly case can be fixed by using "`mathformula`" instead of "`MathFormula`" to replace `<math>` nodes. I tried both versions, and in some cases one of them works better, and in other cases the other one works better.

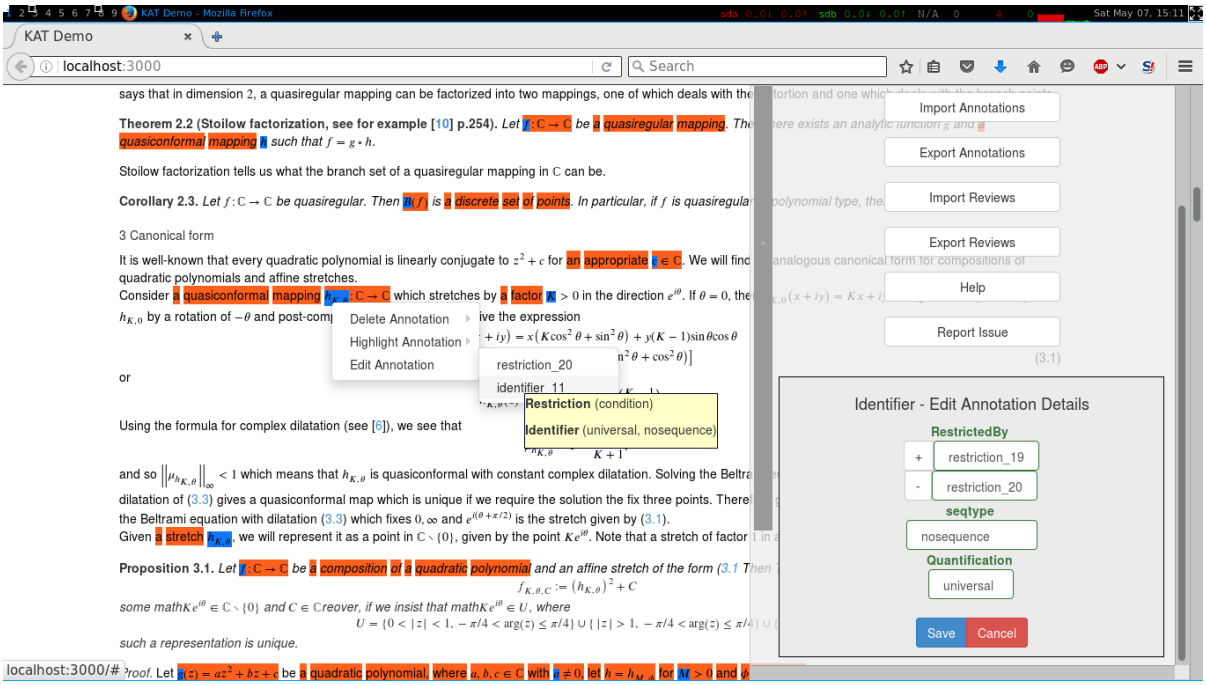


Figure 2: The KAT user interface - here with one of the documents used for evaluation ([4])

2.4 Kwarc Annotation Tool

The Kwarc Annotation Tool (KAT) [DGK+14] is an annotation tool for STEM (Science, Technology, Engineering and Mathematics) documents. We can import the results of our spotter into KAT to visualize and assess them.

KAT is still under development. Currently, only ranges from node to node can be annotated, which means that we have to wrap the words in the HTML representation into nodes. There are on-going efforts to also support string offsets in text nodes using XPointers. The other problem is that not all the information is visualized: There is no good solution for the visualization of overlapping ranges yet, and other information is not immediately visible. For example, in our case you cannot see whether a declaration is marked as universal without hovering over it. Nonetheless, it turned out to be a very valuable tool. A screenshot is provided in figure 2.

3 Declarations

A declaration introduces identifiers and restricts them.

3.1 Restrictions

We distinguish two types of restrictions: *Conditions* and *types*. The distinction is not always clear.

Let us first take a look at conditions. In

Let $x \in \mathbb{R}$

"x" is declared with the condition "$x \in \mathbb{R}$". The declaration could be considered as an abbreviation for "*Let x such that $x \in \mathbb{R}$*". So conditions are expressions containing the declared identifier that have to be true. Let us take a look at type restrictions. In

Let x be a real number.

I consider "*a real number*" a type restriction. Sometimes, type restrictions rather denote the function of an identifier. For example in

Let $f(x)$ be a polynomial of degree n.

"n" has the type restriction "*degree*". Obviously, this means that "n" is a natural number. The type restriction of "$f(x)$" would be "*a polynomial of degree n*".

Often, type restrictions and conditions are combined in a declaration. They can be redundant - and in some cases even equivalent:

For every integer $p \in \mathbb{Z}$, we have the following... [13]

So far, our example restrictions reduced the number of objects an identifier could stand for. An extreme case are restrictions that uniquely define the object. We will call these restrictions *definite*. Both, type restrictions and conditions can be definite. A definite condition would for example be "*Let $f'(x) = \frac{d\,f(x)}{dx}$*". Definite type restrictions often start with the definite article "*the*" (e.g. "*Let $f'(x)$ be the derivative of $f(x)$*"). It is not uncommon to combine definite restrictions with non-definite ones, such as e.g. in "*Let $n \in \mathbb{N}$ be the order of G*". The condition "$n \in \mathbb{N}$" is redundant as the restrictions "*the order of G*" is already defining. A trickier example is the following:

Let n be the order of a non-abelian group.

In my opinion, this declaration does *not* contain a definite restriction, as it only tells us that $n \in \{6, 8, 10, \ldots\}$, while

Let n be the order of a non-abelian group G.

contains a definite restriction, setting $n = \|G\|$. We could rewrite it as "*let G be a non-abelian group and let n be the order of G*".

3.2 Identifiers

At this point, we should take a closer look at our concept of an identifier. The most simple type of identifier is just a character, like e.g. in "*Let p be a prime number*", where "p" is

clearly the identifier. We can also have indexed identifiers, e.g. in a declaration like "*Let* $x^1, x^2, x^3 \in \mathbb{R}^3$ *be linearly independent vectors*", where the three identifiers "x^1", "x^2", and "x^3" are declared. Indices can obviously be variables too, like in "*Let* $a_i \in \mathbb{R}^+$ *for* $1 \leq i \leq n$". In this case, the identifier would be "a_i", and "i" is declared in another declaration. There are also elliptic sequences of identifiers such as in "*Let* $a_1, a_2, \ldots, a_n \in \mathbb{R}^+$", where the reader has to see an obvious pattern to understand the entire sequence. At last, we should consider cases like the following:

Let $\langle G, \circ \rangle$ *be a group.*

Here, strictly speaking, "G" is just a set, not a "*group*" - the entire pair "$\langle G, \circ \rangle$" is a group, i.e. we do not have an identifier for the group itself. However, mathematicians are not always strict with their terminology in this respect and refer to the set as the group. We could think of such declarations as a short form for

Let $\mathcal{G} = \langle G, \circ \rangle$ *be a group.*

From now on, we will refer to such identifiers as *structured identifiers*. The use of structured identifiers is not restricted to tuples, vectors, and matrices, as the following example demonstrates:

Let $\{S_i | i \in I\}$ *be a collection of sets. [5]*

For our spotter, we will try to keep things simple. In this case, our goal would be to simply extract the information that the structured identifier "$\{S_i | i \in I\}$" is declared as "*a collection of sets*".

A particularly interesting case of structured identifiers is

Let $\langle R, +, 0, - \rangle$ *be a commutative group, and* $\langle R, *, 1 \rangle$ *a monoid, such that* $\langle R, +, * \rangle$ *is a ringoid, then we call* $\langle R, +, 0, -, *, 1 \rangle$ *a* ***ring****.*

where "R" is used in both structured identifiers in this declaration. This example is discussed in more detail in [Koh16].

3.3 Universal/Existential/Definite Declarations

Apart from distinguishing different types of restrictions, we also distinguish different types of declarations.

We call declarations that contain at least one definite restriction *definite declarations*. Definite restrictions were explained in section 3.1, so we will not describe them here again.

Non-definite declarations can be divided into two categories: *Universal* declarations and *existential* declarations, introducing universally and existentially quantified variables respectively. If an identifier x is universally quantified, statements containing x hold for every value x that matches the restrictions of x. If an identifier is existentially quantified, on the

other hand, the statements containing it hold only for at least one of the values matching the restrictions.

A typical universal declaration would be

> *For any* $a, b \geq 0$ *we have* $\frac{a+b}{2} \leq \sqrt{\frac{a^2+b^2}{2}}$.

In this case the quantification was explicitly stated ("*for any*"). I believe that all (non-definite) declarations without explicitly stated quantification are universal. In the following example, "S" and "$\sim$" are declared. Clearly, it is a universal declaration.

> *Let* S *be a nonempty set and let* $\sim$ *be an equivalence relation on* S*. Then* $\sim$ *yields a partion on* S*, where... [5]*

Sometimes, universal declarations are very similar to implications, like in the following lemma

> *If* $\langle A, R\rangle$ *and* $\langle B, S\rangle$ *are isomorphic well-orderings, then the isomorphism between them is unique. [9]*

You can argue whether it is a declaration or a simple implication. In this particular case the identifiers were declared before. On the other hand, it is not untypical to re-declare identifiers in lemmas and theorems. Either way, the first part of this lemma clearly has the function to restrict the identifiers (consider also examples 3 and 4).

An example for an existential declaration would be

> *For all* x *and* y *there is a set* z *such that* $x \in z$ *and* $y \in z$.

where z is existentially quantified.

Difficulties

Now let us look at some examples that do not fit as well into our simple universal/existential distinction. For example:

> *There is no set* z *such that* $\forall x.x \in z$.

Here we have non-existence. For now, we will consider it a universal declaration (as it is equivalent to "*For all* z *it does not hold that* $\forall x.x \in z$"). More problematic is

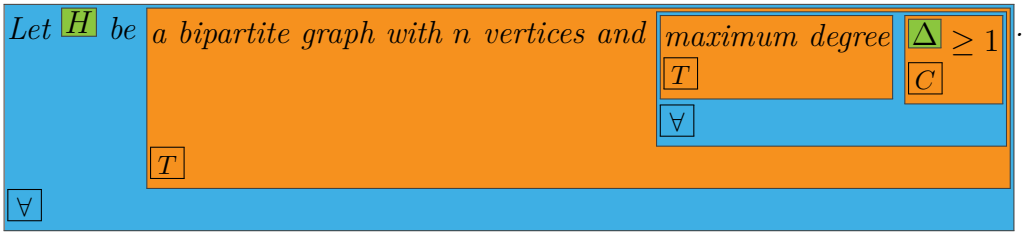
> *For every* $x \in X_n$*, there exist* at most *one pair* $(\nu_1, u) \in N \times U$ *[3]*

Theoretically, we could express "*there exists at most one* $(\nu_1, u) \in N \times U$ *such that* x" as "*there exists no* $(\nu_1, u) \in N \times U$ *such that* x or *there exists a unique* $(\nu_1, u) \in N \times U$ *such that* x" and express unique existence by stating that there do not exist "$(\nu_1', u') \neq (\nu_1, u)$" matching the requirements. However, this is probably not what we want. Maybe we need a more sophisticated representation of quantification. Fortunately, such cases are rare.

3.4 Requirements

There are three different objects our spotter should find: *Declarations*, *restrictions*, and *identifiers*. A declaration should contain a reference to the declarational phrase and the set of identifiers it declares. In addition, it should be marked as universal (marked with $\boxed{\forall}$), existential (marked with $\boxed{\exists}$), or definite (marked with $\boxed{=}$). A restriction should contain a reference to the restricting phrase/formula. It should be marked as either a type restriction (marked with $\boxed{T}$) or a condition (marked with $\boxed{C}$). Optionally, restrictions can be marked to be defining (with $\boxed{=}$). Identifiers need to have a reference to the original identifier in the document and the set of restrictions on it. It can optionally be marked as structured (marked with $\boxed{S}$) or elliptic (marked with $\boxed{E}$), i.e. the identifier is actually an elliptic sequence of identifiers.

Let's demonstrate this on a concrete example:



. [8]

The boxes correspond to different annotated ranges, where identifiers are marked green, restrictions orange, and declarations blue. The visualization does not contain the information which identifiers are declared in each declaration and which restrictions restrict them. It should be obvious that "H" is declared in the outer declaration and restricted by the type restriction "*a bipartite graph with n vertices and maximum degree* $\Delta \geq 1$", while Δ is declared in the nested declaration and restricted by the type "*maximum degree*" and the condition "$\Delta \geq 1$".

3.5 Problems

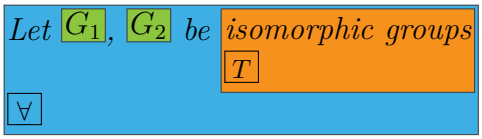
Unfortunately, our spotter requirements have limitations in what they can represent.

Plurals

Consider for example the following declaration:

> *Let* G_1, G_2 *be isomorphic groups.*

Our spotter could find the following results:



Since our spotter is supposed to be simple and should not modify the restrictions in any way, we keep the plural "*groups*". The problem is that our representation does not convey that "*isomorphic*" describes the relation between the two identifiers and is not a property of the groups themselves. It would difficult to extract this information in the first place, as the declaration

> *Let G_1, G_2 be abelian groups.*

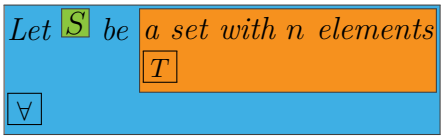
is structurally the same. Having different readings of plurals is a well-known problem in linguistics (see e.g. [Uss08]).

Implicit Declarations

Sometimes, identifiers are declared implicitly. In

> *Let S be a set with n elements*

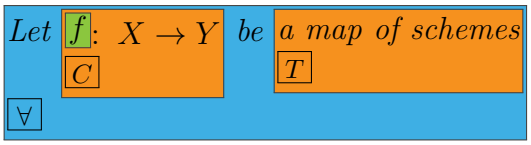
our spotter would find the result



This misses the implied information that "n" has to be a natural number. Another example would be

> *Let $f : X \to Y$ be a map of schemes [13]*

In this case, our spotter might find



which does not capture the fact that "X" and "Y" are implicitly declared as schemes.

Structured Identifiers

In the following declaration

> *Let $\mathcal{G} = \langle G, \circ \rangle$ be a group.*

we obviously declare something as "*a group*". There are several ways of looking at it. We could say that "$\mathcal{G}$" is universally declared as "*a group*", and that "$\langle G, \circ \rangle$" is declared definitely by the condition "$\mathcal{G} = \langle G, \circ \rangle$" - or the other way around. Either way, this violates our

assumption that a declaration is either universal or existential or definite, as we would declare one identifier universally and one definitely. Another way of looking at it is to say that both "$\mathcal{G}$" and "$\langle G, \circ \rangle$" are declared universally as "*a group*" with the condition "$\mathcal{G} = \langle G, \circ \rangle$". Similar situations are shown in examples 10 and 14 in the phenomenology.

4 Pattern Matcher

The most obvious approach to finding declarations is pattern matching. I tried several existing tools, which all turned out to be unsuitable.

XPath/XQuery [URL10, BCF+07] is simply unfit for our purposes. Just consider the complexity of the following XPath expression matching the `<math>` node in declarations of the form `(L|l)et` **`<math>`** `be (a|an|any|some):`

```
//xhtml:span[@class='ltx_word' and (text()='Let' or text()='let')]/following−sibling::∗[.=//mathml:math and
    position()<3 and .=//xhtml:span[@class='ltx_word' and text()='be']/preceding−sibling::∗[position() >
    last()−3 and .=//xhtml:span[@class='ltx_word' and (text()='a' or text()='an' or text()='some' or text()
    ='any')]/preceding−sibling::∗[position() > last()−3]]]
```

The *mwetoolkit* [Ram15] allows to consider POS tags in pattern definitions, but it does support marking of different parts of a pattern, i.e. we could only extract the information that a sequence of words is a declaration, without any additional information about the restrictions and identifiers.

The last option I considered was using conventional *regular expressions*. With groups it is to some extent possible to mark different parts of a declaration (restrictions and identifiers). Also, the POS tags could be encoded in the string, so that we could consider them as well. Unfortunately, it is impossible to analyze tree structures with regular expressions. However, we will need to analyze the parse trees, because the type restrictions tend to be very long noun phrases.

Therefore, I created my own pattern language as a part of the LLaMaPUn library.

4.1 Requirements

Let us consider a simple example to illustrate our objectives:

> *Let p be a prime number.*

For such declarations we would like the following kind of pattern: "*let*", followed by a `<math>` node, which will contain the identifiers, followed by "*be*", followed by a noun phrase starting with "*a*"/"*an*"/"*any*"/"*some*"/..., which is a type restriction. So in our pattern language we would like to have sequences of words and phrases, which we can mark. In addition, it should be allowed to put further restrictions on phrases, such as requiring it to start with a certain pattern.

4.2 Formal Specification

Let $\mathcal{S}$ denote the set of strings, $\mathcal{POS}$ the set of part-of-speech tags, and $\mathcal{PHR}$ the set of phrases. Also, let $\mathcal{M}$ be a set of markers and $\mathcal{N}$ be a set of notes. We will now inductively define the set of patterns $\mathscr{P}$:

- $\texttt{AnyWord} \in \mathscr{P}$ matches any word
- $\texttt{Word}(s) \in \mathscr{P}$ for $s \in \mathcal{S}$, matching words with the string representation s
- $\texttt{Pos}(p) \in \mathscr{P}$ for $p \subseteq \mathcal{POS}$, matching words with one of the POS tags in p
- $\texttt{WordPos}(s, p) \in \mathscr{P}$ for $s \in \mathcal{S}$ and $p \subseteq \mathcal{POS}$, matching words that match both $\texttt{Word}(s)$ and $\texttt{Pos}(p)$.
- $\texttt{Phrase}(p, d, \mathbf{p}_{\text{start}}, \mathbf{p}_{\text{end}}) \in \mathscr{P}$ for $p \in \mathcal{PHR}$, $d \in \{\texttt{up}, \texttt{down}\}$, and $\mathbf{p}_{\text{start}}, \mathbf{p}_{\text{end}} \in \mathscr{P}$. p denotes the kind of phrase we are looking for (e.g. a noun phrase). If $d = \texttt{up}$ we match the shortest phrase fulfilling all the conditions (bottom-up in tree perspective), and if $d = \texttt{down}$ we match the longest phrase. $\mathbf{p}_{\text{start}}$ and $\mathbf{p}_{\text{end}}$ need to match the beginning and end of the phrase.
- $\texttt{Seq}(\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n) \in \mathscr{P}$, where $\mathbf{p}_i \in \mathscr{P}$ for $i \in \{1, 2, \ldots, n\}$. It matches a sequence of words, where the first words match $\mathbf{p}_1$, the following words match $\mathbf{p}_2$, etc.
- $\texttt{Or}(\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n) \in \mathscr{P}$, where $\mathbf{p}_i \in \mathscr{P}$ for $i \in \{1, 2, \ldots, n\}$. It matches sequences of words that match $\mathbf{p}_i$ for some $i \in \{1, 2, \ldots, n\}$. If a sequence matches multiple patterns, only the first one is matched (important for markers).
- $\texttt{Marked}(m, N, \mathbf{p}, e_{\text{start}}, e_{\text{end}}) \in \mathscr{P}$ where $m \in \mathcal{M}$, $N \subseteq \mathcal{N}$, $\mathbf{p} \in \mathscr{P}$, and $e_{\text{start}}, e_{\text{end}} \in \mathbb{N}$. It matches any sequence matching $\mathbf{p}$, and returns the marker m, along with a set of notes N and the range matched by $\mathbf{p}$, excluding the first e_{start} words and the last e_{end} words.

A pattern matcher takes a sequence of words and a pattern and returns a set of markers with their associated notes and ranges.

Potential Improvements

The pattern language was extended whenever a new feature was required and never really got beyond its experimental stage. At this point, it probably makes sense to re-design the pattern language based on the insights gained so far. In particular, I believe that it would make sense to treat patterns matching a single word differently, which would allow us to do things like matching a word that does *not* match a particular word pattern. Maybe it is even possible to introduce patterns for the presentation MathML in the `<math>` nodes. I have not really thought about it yet.

4.3 Implementation

The pattern matcher has been implemented as part of the LLaMaPUn library (section 2.2) using the Rust programming language [MK14]. It may be possible to compile the patterns into automata, similarly to the way regular expressions are compiled, however, pattern matching was not the main concern of this thesis and a naive pattern matcher is sufficiently efficient for our purposes. For now, the matcher tries for every word, whether it is the beginning of a match by naively recursing through the pattern. Afterwards, it returns the matches and markers.

4.4 Example Pattern

Let's take a look at an example pattern, which could be used for a declaration spotter. My actual spotter uses a more complex pattern. In the spotter, the pattern is encoded using Rust's data structures. Since it is not that readable, I will use a more readable syntax here:

```
# matches an identifier, marked with an IDENTIFIER marker
identifier := Marked(IDENTIFIER, {}, Word("MathFormula"), 0, 0)

# indefinite article
indef_article := Or(Word("a"), Word("an"), Word("any"), Word("every"), Word("each"), Word("all"))

# longest noun phrase starting with an indefinite article, with a RESTRICTION marker
long_np := Marked(RESTRICTION, {"type", "long"},
                  Phrase(NP, down, indef_article, AnyWord), # no restriction on last word
                  0, 0) # don't exclude anything from marker range

# matches e.g. "let p be a prime number"
let_pattern := Seq(Word("let"), identifier, Word("be"), long_np)

# matches e.g. "a prime number p"
np_with_id := Marked(RESTRICTION, {"type", "short"},
                     Phrase(NP, up, indef_article, identifier), # us non-greedy up instead of down
                     0, 1) # exclude identifier from marker range

# matches e.g. "there exists a prime number p"
exists_pattern := Seq(Word("there"), Or(Word("is"), Word("exists")), np_with_id)

# matches e.g. "for every prime number p"
for_pattern := Seq(Word("for"), np_with_id)

# matches universal declarations
universal := Marked(DECLARATION, {"universal"}, Or(let_pattern, for_pattern), 0, 0)

# matches existential declarations
existential := Marked(DECLARATION, {"existential"}, exists_pattern, 0, 0)

# and finally our declaration pattern
declaration := Or(universal, existential)
```

5 Analysis

The pattern matcher provides us with information about the declarations, their restrictions, and the <math> nodes which should contain the identifiers. Now we need to look into the <math> nodes to find potential identifiers and additional restrictions. This can be done reasonably well using simple heuristics. Afterwards, we can take the context into consideration to improve the results.

5.1 Identifier Extraction

I use simple heuristics for the extraction of identifiers, because they turned out to work reasonably well, and anything more thorough would be out of the scope of this thesis. Let us first take a look at an example:

Let $x_1, \ldots, x_n \in \mathbb{R}$.

From our pattern matcher we know, that this is potentially a universal declaration whose identifiers are contained in the <math> node corresponding to "$x_1, \ldots, x_n \in \mathbb{R}$". In order to extract the identifiers, we look into the presentation MathML:

```
<mrow xmlns="http://www.w3.org/1998/Math/MathML" id="S0.Ex1.m1.1.13" xref="S0.Ex1.m1.1.13.cmml
    ">
  <mrow id="S0.Ex1.m1.1.13.1" xref="S0.Ex1.m1.1.13.1.1.cmml">
    <msub id="S0.Ex1.m1.1.13.1.2" xref="S0.Ex1.m1.1.13.1.1.cmml">
      <mi id="S0.Ex1.m1.1.1" xref="S0.Ex1.m1.1.1.cmml">x</mi>
      <mn id="S0.Ex1.m1.1.2.1" xref="S0.Ex1.m1.1.2.1.cmml">1</mn>
    </msub>
    <mo id="S0.Ex1.m1.1.6">,</mo>
    <mi mathvariant="normal" id="S0.Ex1.m1.1.7" xref="S0.Ex1.m1.1.7.cmml">&amp;#x2026;</mi>
    <mo id="S0.Ex1.m1.1.8">,</mo>
    <msub id="S0.Ex1.m1.1.13.1.4" xref="S0.Ex1.m1.1.13.1.1.cmml">
      <mi id="S0.Ex1.m1.1.9" xref="S0.Ex1.m1.1.9.cmml">x</mi>
      <mi id="S0.Ex1.m1.1.10.1" xref="S0.Ex1.m1.1.10.1.cmml">n</mi>
    </msub>
  </mrow>
  <mo id="S0.Ex1.m1.1.11" xref="S0.Ex1.m1.1.11.cmml">&amp;#x2208;</mo>
  <mi id="S0.Ex1.m1.1.12" xref="S0.Ex1.m1.1.12.cmml">&amp;#x1D411;</mi>
</mrow>
```

We will now go over a few heuristics that turned out to be useful in our spotter. Identifiers are typically <mi> nodes or <msub>, <msup>, or <msubsup> nodes, whose first child is an <mi> node.

We are not only interested in isolated identifiers, but also in *sequences* of identifiers. With *sequences* we mean a sequence of identifier nodes, separated by <mo> nodes containing a separator symbol. The separator symbol can be any symbol, but it should be the same for every <mo> node in the sequence. A particularly obvious case is the comma as a separator. However, there are many relational symbols that are used as well, such as "$<$", or "$\subset$". A

special case of sequences are elliptic sequences, which can easily identified by the three periods ("...") . An example is provided in the phenomenology in example 5. LaTeXML conveniently translates them to the unicode character `U+2026` (`HORIZONTAL ELLIPSIS`), no matter whether `\ldots` or simply `...` was used.

We haven't considered multi-character identifiers yet, which are usually not correctly represented in the MathML. For example, if a mathematician writes `$sin(x)$` instead of `$\sin(x)$`, the corresponding MathML would be

```
<mrow xmlns="http://www.w3.org/1998/Math/MathML" id="S0.Ex2.m1.1.7" xref="S0.Ex2.m1.1.7.cmml"
    >
  <mi id="S0.Ex2.m1.1.1" xref="S0.Ex2.m1.1.1.cmml">s</mi>
  <mo id="S0.Ex2.m1.1.7.1" xref="S0.Ex2.m1.1.7.1.cmml">&#x2062;</mo>
  <mi id="S0.Ex2.m1.1.2" xref="S0.Ex2.m1.1.2.cmml">i</mi>
  <mo id="S0.Ex2.m1.1.7.1a" xref="S0.Ex2.m1.1.7.1.cmml">&#x2062;</mo>
  <mi id="S0.Ex2.m1.1.3" xref="S0.Ex2.m1.1.3.cmml">n</mi>
  <mo id="S0.Ex2.m1.1.7.1b" xref="S0.Ex2.m1.1.7.1.cmml">&#x2062;</mo>
  <mrow id="S0.Ex2.m1.1.7.2">
    <mo stretchy="false" id="S0.Ex2.m1.1.4">(</mo>
    <mi id="S0.Ex2.m1.1.5" xref="S0.Ex2.m1.1.5.cmml">x</mi>
    <mo stretchy="false" id="S0.Ex2.m1.1.6">)</mo>
  </mrow>
</mrow>
```

Note that the unicode character `U+2062` is `INVISIBLE TIMES`, i.e. the "sin" gets interpreted as a product of "s", "i", and "n". We can work around this using another simple heuristic: Sequences of identifiers interrupted by `<mo>` nodes with invisible multiplication tend to be just one multi-character identifier (in declarations).

A special case of identifiers are *structured identifiers* (see section 3.2). E.g. in algebra papers, they occur very often in the form of tuples, such as in the following simple example:

Let $\langle G, \circ \rangle$ *be a group.*

For "$\langle G, \circ \rangle$" we get the following presentation MathML:

```
<mrow xmlns="http://www.w3.org/1998/Math/MathML" id="S0.Ex4.m1.1.6" xref="S0.Ex4.m1.1.6.1.cmml
    ">
  <mo stretchy="false" id="S0.Ex4.m1.1.1">&#x27E8;</mo>
  <mi id="S0.Ex4.m1.1.2" xref="S0.Ex4.m1.1.2.cmml">G</mi>
  <mo id="S0.Ex4.m1.1.3">,</mo>
  <mo id="S0.Ex4.m1.1.4" xref="S0.Ex4.m1.1.4.cmml">&#x2218;</mo>
  <mo stretchy="false" id="S0.Ex4.m1.1.5">&#x27E9;</mo>
</mrow>
```

So we could identify these structured identifiers easily as a sequence of identifiers enclosed by round or angular brackets and separated by commata wrapped in `<mo>` nodes.

5.2 Selection

After extracting the potentially declared identifiers, we need to decide which ones actually got declared. We can again use heuristics to make this decision. If the `<math>` node starts with an elliptic sequence, we assume that this sequence is being declared. If it starts with a non-elliptic, comma-separated sequence or any sequence accompanied by a plural type restriction, we assume that the identifiers of the sequence are being declared. Otherwise, we take the first identifier in the `<math>` node. If there is no identifier, we assume that our matcher found a wrong match and discard it.

Another idea, which I have not yet implemented, would be to consider a larger context. For example, identifiers that have recently been declared (e.g. in the same sentence or paragraph) are less likely to be declared again. However, it is not particularly easy, because identifiers often get re-declared. In the case of "less important identifiers" such as indices this sometimes happen multiple times in a paragraph (see e.g. example 17). In addition, identifiers are frequently declared after their first usage, as for example in "*We know that* $x^2 \geq 0$ *for all* $x \in \mathbb{R}$" (see also [WG10]). Getting this analysis right would probably require a much more detailed investigation and is in my opinion unlikely to improve the results significantly.

6 Results

Unfortunately, the spotter does not support definite declarations, because I only recently changed my mind about them. There is another, technical restriction: KAT (see section 2.4) does not support the annotation of discontinuous ranges, so e.g. the following example could not be annotated without using something like linked lists of ranges in KAT:

> *A submonoid* H *of* $\mathbb{N}_0$ *[6]*

6.1 Evaluation

For completeness, I will provide a short evaluation. The evaluation does not consider definite restrictions, because they are not yet supported by the spotter as mentioned before. Also, identifiers declared inside `<math>` nodes (as in "*... such that* $\forall x \in \mathbb{R}$*...*") were excluded. It is not always clear whether something is a declaration and *the results should be taken with a grain of salt.*

I used four arXiv documents for the evaluation: [4], [14], [7], and [3]. 20 declarations did not get considered, because they could not be annotated in KAT for technical reasons. This issue will be solved soon, as we are moving away from restricting annotations to node ranges, towards an XPointer based approach with string offsets. Apart from that, there are in total 375 declarations in these documents (according to my counting). 130 of them were annotated correctly, 212 were not found, and 33 were found with errors, such as missing restrictions or wrong identifiers. In addition, 22 of the matches were actually no declarations. This gives

us a precision of $\frac{130}{130+33+22} \approx 0.7$, and a recall of $\frac{130}{375} \approx 0.35$. The F1-score score is roughly 0.46. Again: The results should be taken with caution, as it is not always clear what we should consider a declaration.

Let us now look at some of the problems the spotter had during the evaluation. First, we will consider wrong detections, and then we will take a look at the declarations we missed.

6.2 Wrong Results

One thing that caused many problems were sentences like

> *Then $B(f)$ is a discrete set of points. [4]*

The spotter interprets this as "$B(f)$" being declared as "*a discrete set of points*". We could get rid of many such errors by excluding the case when the pattern is preceded by a "*then*". This is not yet support by our pattern language.

Another problem were declarations with many `<math>` nodes like the following:

> *Two patterns $p_1, p_2 \in Q^E$ are **mutually erasable** (briefly, m.e.) for $\mathcal{A}$ if $F_{\mathcal{A}}(c_1) = F_{\mathcal{A}}(c_2)$ for any $c_1, c_2 \in \mathcal{C}$ such that $(c_1)_{|E} = p_1$, $(c_2)_{|E} = p_2$, and $(c_1)_{|G\setminus E} = (c_2)_{|G\setminus E}$. [3]*

This would correspond to the following plaintext representation:

> Two patterns **MathFormula** are mutually erasable (briefly, m.e.) for **MathFormula**
> if **MathFormula** for any **MathFormula** such that **MathFormula**, **MathFormula**, and **MathFormula**.

Such sentences challenge Senna and they are not captured well by the spotter patterns, so the identifier is often assumed in the wrong `<math>` node. Maybe it makes sense to create individual patterns for such sequences.

6.3 Missing Results

Several kinds of declarations were missed very often, as I did not consider such cases in the patterns. For example, declarations with adjective restrictions were usually missed, as e.g. in

> *Let $f : \mathbb{C} \to \mathbb{C}$ be quasiregular. [4]*

Another mistake I made was categorically excluding every declaration that has an equal sign after the identifier assuming that they must be declared definitely. This excluded declaration like "*for $k = 1, \ldots, n$*", and in [14] it excluded of lot of declaration which had the form

> *Suppose $x = \{x_n\}_{n=1}^{\infty} \in l^{\infty}(V)$. [14]*

One could argue that "x" is definitely declared (and thus correctly discarded by our spotter), however, "$\{x_n\}_{n=1}^{\infty}$" is universally declared with the restriction "$\{x_n\}_{n=1}^{\infty} \in l^{\infty}(V)$".

Also, patterns for declarations with identifiers in multiple `<math>` nodes are missing. So declarations like the following would be missed:

> *[...] where* $\sigma_1, \sigma_2, \ldots, \sigma_k$ *and* u *are complex parameters. [7]*

6.4 Potential Improvements

The most obvious improvement would be to put more efforts into the patterns and the identifier extraction, as not many efforts were put into the optimization of the spotter. Simply fixing some of the problems in the previous section would probably improve the quality of the results significantly. Once a large fraction of declarations is covered, it will become more and more challenging to further optimize the spotter by manually searching for wrong, too general, or missing patterns. At this point, we will be able to generate statistics about which identifiers are typically declared by what kind of restrictions, e.g. "p" might often be declared as "*a prime number*", sometimes as "*an integer*", and sometimes as "*a probability*" - also depending on the topic of the paper. We can use this information to find declarations which are not covered by the spotter yet. For example, we could search for sentences containing the undeclared identifier "p" and the word "*prime number*". This way, new patterns can be discovered and added to the spotter.

This strategy reminds a bit of bootstrapping (as described in [PLB+06]). However, in this case the relations are very weak, as "p" might be declared in many other ways than just as a "*prime number*". Therefore, unsupervised learning probably won't be a viable option.

7 Conclusion

We have looked at a framework for the implementation of spotters, which works reasonably well in my opinion. There are on-going efforts to improve the way we store ranges, which would probably enable us to do larger and more complex experiments.

In my opinion, the pattern language turned out to be quite useful. It should probably get re-designed and re-implemented in a less experimental manner. Then we could start implementing better and more spotters. The heuristics for the identifier extraction worked surprisingly well, but at some point we will probably need more sophisticated tools to analyze the `<math>` nodes.

Acknowledgements

I would like to thank Prof. Kohlhase for his continuous support and guidance and the incredible number of useful suggestions. Also, I would like to thank Deyan Ginev, who introduced me to this research and provided a lot of useful feedback and without whom there would be no LLaMaPUn library.

References

[ABC+10] Ron Ausbrooks, Stephen Buswell, David Carlisle, Giorgi Chavchanidze, Stéphane Dalmas, Stan Devitt, Angel Diaz, Sam Dooley, Roger Hunter, Patrick Ion, Michael Kohlhase, Azzeddine Lazrek, Paul Libbrecht, Bruce Miller, Robert Miner, Murray Sargent, Bruce Smith, Neil Soiffer, Robert Sutor, and Stephen Watt. Mathematical Markup Language (MathML) version 3.0. W3C Recommendation, World Wide Web Consortium (W3C), 2010.

[ArX] `arxiv.org` e-Print archive.

[BCF+07] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery: An XML Query Language, January 2007.

[Col11] R. Collobert. Deep learning for efficient discriminative parsing. 2011.

[DGK+14] Mircea Alex Dumitru, Deyan Ginev, Michael Kohlhase, Vlad Merticariu, Stefan Mirea, and Tom Wiesing. System description: Kat an annotation tool for stem documents. 2014.

[Gan09] Mohan Ganesalingam. *The Language of Mathematics*. PhD thesis, Cambridge University, 2009.

[GJA+09] Deyan Ginev, Constantin Jucovschi, Stefan Anca, Mihai Grigore, Catalin David, and Michael Kohlhase. An architecture for linguistic and semantic analysis on the arXMLiv corpus. In *Applications of Semantic Technologies (AST) Workshop at Informatik 2009*, 2009.

[Kam81] Hans Kamp. A theory of truth and semantic representation. In J. Groenendijk, Th. Janssen, and M. Stokhof, editors, *Formal Methods in the Study of Language*, pages 277–322. Mathematisch Centrum Tracts, Amsterdam, Netherlands, 1981.

[Koh16] Michael Kohlhase. The role of discourse referents in theory graphs. 2016.

[KWZ08] Fairouz Kamareddine, J. B. Wells, and Christoph Zengler. Computerising mathematical text with mathlang. *Electron. Notes Theor. Comput. Sci.*, 205:5–30, 2008.

[Mil] Bruce Miller. `LaTeXML`: A LaTeX to XML converter. Web Manual at `http://dlmf.nist.gov/LaTeXML/`. seen September2011.

[MK14] Nicholas D. Matsakis and Felix S. Klock, II. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 103–104, New York, NY, USA, 2014. ACM.

[MSB+14] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, 2014.

[PLB+06] M. Pasca, D. Lin, J. Bigham, A. Lifchits, and A. Jain. Organizing and searching the world wide web of facts-step one: the one-million fact extraction challenge. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21, page 1400. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.

[Ram15] Carlos Ramisch. *Multiword Expressions Acquisition: A Generic and Open Framework*, volume XIV of *Theory and Applications of Natural Language Processing*. Springer, 2015.

[RCK11] L. Bottou M. Karlen K. Kavukcuoglu R. Collobert, J. Weston and P. Kuksa. Natural language processing (almost) from scratch. 2011.

[SS14] Ulf Schoeneberg and Wolfram Sperber. Pos tagging and its applications for mathematics, 2014.

[URL10] XPath Reference. Recommendation, W3C, 2010.

[Uss08] Cherlon Ussery. Processing plural dps: Collective, cumulative, and distributive interpretations, 2008.

[WG10] Magdalena Wolska and Mihai Grigore. Symbol declarations in mathematical writing: A corpus study. In Petr Sojka, editor, *Towards Digital Mathematics Library, DML workshop*, pages 119–127. Masaryk University, Brno, 2010.

[Wol13] Magdalena A. Wolska. *Student's Language in Computer-Assisted Tutoring of Mathematical Proofs*. PhD thesis, ComputerLinguistik, Saarland University, 2013.

[Zin03] Claus Zinn. A computational framework for understanding mathematical discourse. *Logic Journal of the IGPL*, 11(4):457–484, 2003.

Appendices

A Phenomenology

When designing a spotter, one stumbles over many fascinating phenomena. Here, I will present a list of declarations, which is supposed to cover a large variety of phenomena one can encounter. Only few of them are correctly identified by our spotter. The list only contains real-world examples - this restriction makes it much harder to show phenomena in their "pure" form. Therefore, I will also provide a list of common phenomena and reference some examples containing them:

1. Elliptic identifier sequence: 5, 13
2. Declaration after first usage: 2, 6
3. Long restrictions: 1, 5
4. Nested declarations: 1, 5
5. No identifier declared: 7
6. Non-obvious identifiers declared: 6
7. Structured identifiers: 10, 14
8. Multiple identifiers: 3, 6, 14

Here are the examples. Some parts are highlighted to guide the reader's attention towards particular phenomena.

1. Example

   > *Let C be a complete nonsingular irreducible curve over an algebraically closed field k of characteristic 0, which is called a* curve *in this paper. [6]*

   Restrictions can be very long and contain nested declarations, which makes finding the right boundaries rather tricky.
2. Example

   > *By Lie's theorem, there is a common eigenvector, say ν, with $g(\nu) = \chi(g) \cdot \nu$ for all $g \in G$. [1]*

   There are many rare declaration patterns such as this one, which means that sometimes a particular style of an author makes the spotter miss many declarations in a document. Apparently, there can be a "*say*" before the introduced identifier. Also, it is a simple example of an identifier ("g") being declared after its first occurrence.
3. Example

   > *Here we have used that if H and K are Lie subgroups of G with $H \cap K = 1$, and HK is a closed subgroup of G, then HK is homeomorphic to $H \times K$. [1]*

Here, two identifiers, "H" and "K" are declared in the same declaration. Their restrictions are very intertwined.

4. Example

   Let F be a graph with m edges and no isolated vertices. Then, for $k \geq 3$ it holds that

   $$r_k(F) \leq k^{3\cdot 2^{-1/3} k m^{2/3} + k(2m)^{1/3}} 8m.$$

   Further we study the case when F is bipartite and show an upper bound $r_k(F) \leq k^{(1+o(1))2\sqrt{mk}}$. [8]

   In the original document, everything until the first big inequality is part of a theorem. The next sentence (starting with "*Further we study...*") is not part of the theorem anymore, but still uses the declaration of "F" and further restricts it.

5. Example

   Suppose G is a graph with vertex set V_1, and let $V_1 \supset \ldots \supset V_q$ be a family of nested subsets of V_1 such that $|V_q| \geq x \geq 4n$, and for $1 \leq i < q$, all but less than $(2d)^{-d}\binom{x}{d}$ d-sets $U \subset V_{i+1}$ satisfy $|N(U) \cap V_i| \geq x$. [8]

   First of all, this shows how complicated declarations can get. Only an experienced reader can quickly grasp the structure of this declaration. The declaration of "V_1", ..., "V_q" is very interesting. The identifiers are introduced using an ellipsis. However, one of these identifiers ("V_1") has already been introduced before. This still makes sense, as the sequence of identifiers, related by "$\supset$", is declared in its entirety as "*a family of nested subsets*", rather than the identifiers being declared individually.

6. Example

   A system is called banded*, if $a_{ij} = 0$ for all $|i - j| \geq k$ for some $k < n$. [11]*

   This shows very nicely how two identifiers, "i" and "j", can be declared and restricted as part of a larger inequality.

7. Example

   We have an isomorphism $Spec(D^{perf}(X)) \simeq X$ of ringed spaces [13]

   While it looks a lot like a declaration, in fact, no identifier is introduced for the isomorphism.

8. Example

   Let B be the vertex set of a largest bipartite induced subgraph of G. [12]

   It looks a lot like a definite declaration of "B", but since there is no identifier for the "*largest bipartite induced subgraph of G*", I would consider it a universal declaration.

9. Example

Let h as above and set $L = \max\{K, 1/K\}$. [4]

"h" is declared here, but the restriction is just a reference to a previous declaration.

10. Example

    A sequence $x = \{x_n\}_{n=1}^{\infty} \in l^{\infty}(V)$ is called quasi-almost convergent to $v \in V$ if $\forall L \in \Pi$, $L(x - \widetilde{v}) = 0$. [14]

    The declaration with "$x = \{x_n\}$" is very interesting. This phenomenon is also described in section 3.5. Also note the declaration of "L" completely inside a `<math>` node.

11. Example

    Choose N_2 such that $(\|x_1\|_V + \|x_2\|_V + \cdots + \|x_{N_1}\|_V)/N_2 < \varepsilon/2$. [14]

    This statement clearly introduces "N_2" and puts a restriction on it. Yet, it does not really "feel" like a normal declaration to me. I believe that it is related to the concept of *fixed* variables.

12. Example

    it suffices to find a particular Banach limit functional L_0 such that $L_0(x) = p(x)$. [14]

    This statement introduces "L_0" as "*a particular Banach limit functional*" and restricts it definitely (as I understand it) with "$L_0(x) = p(x)$". However, its existence is only shown later in the proof.

13. Example

    Let $\mathcal{U}$ be a standard tableau with k cells and the q-contents $\sigma_1, \sigma_2, \ldots, \sigma_k$. [7]

    Note how "k" is introduced and implicitly restricted as an integer. The declaration of the "*q-contents*" is much more interesting: Are "$\sigma_1, \sigma_2, \ldots, \sigma_k$" declared definitely as the "*q-contents*" of "$\mathcal{U}$", or are they declared universally at the same time as "$\mathcal{U}$" along the lines "*for all $\mathcal{U}$ and $\sigma_1, \sigma_2, \ldots, \sigma_k$, where $\sigma_1, \sigma_2, \ldots, \sigma_k$ are the q-contents of $\mathcal{U}$*"? This might be related to the problem described in section 3.5. Note: I do not know whether the "*q-contents*" uniquely define a "*standard tableau*".

14. Example

    If $\alpha = (i, j)$ is a cell of λ, then [...] [7]

    See also section 3.5

15. Example

Thus, for all n large enough,

$$\frac{H_{n,S'}(c_1,c_2)}{\gamma_{S'}(n)} \leq \frac{H_{\beta n,S}(c_1,c_2)}{\gamma_S(\beta n)} \cdot \frac{\gamma_S(\beta n)}{\gamma_{S'}(n)} \leq \frac{\alpha_1 \beta^k}{\alpha_2} \cdot \frac{H_{\beta n,S}(c_1,c_2)}{\gamma_S(\beta n)} \ ,$$

and the rightmost term vanishes for $n \to \infty$. [3]

Here, "n" is first vaguely declared as "*large enough*" and later we declare it with "*for $n \to \infty$*". I am not sure whether "$n \to \infty$" is universal or definite.

16. Example

    Then, whatever $\{X_n\}$ is, $(W, \{X_n\})$-surjectivity implies $(B, \{X_n\})$-surjectivity; [3]

"*whatever $\{X_n\}$ is*" is a remarkably vague declaration.

17. Example

    Given a set of points V, a k-simplex is an unordered subset $\{v_0, v_1, \ldots, v_k\}$ where $v_i \in V$ and $v_i \neq v_j$ for all $i \neq j$. The faces of this k-simplex consist of all $(k-1)$-simplices of the form $\{v_0, \ldots, v_{i-1}, v_{i+1}, \ldots, v_k\}$ for some $0 \leq i \leq k$. [2]

Here, "i" is first declared with the condition "$i \neq j$", and in the next sentence, it is declared as "$0 \leq i \leq k$". I am not sure how "i" is quantified in the second example.

B References for Examples

[1] Azad, H., and Biswas, I. A note on real algebraic groups, 2013.

[2] Bubenik, P., Carlsson, G., Kim, P. T., and Luo, Z. Statistical topology via morse theory, persistence and nonparametric estimation.

[3] Capobianco, S. Surjunctivity for cellular automata in besicovitch spaces, 2007.

[4] Fletcher, A., and Goodman, D. Quasiregular mappings of polynomial type in r$\hat{2}$, 2010.

[5] Fraleigh, J., and Katz, V. *A first course in abstract algebra*. Addison-Wesley world student series. Addison-Wesley, 2003.

[6] Harui, T., Komeda, J., and Ohbuchi, A. The weierstrass semigroups on double covers of genus two curves, 2013.

[7] Isaev, A. P., Molev, A. I., and Os'kin, A. F. On the idempotents of hecke algebras.

[8] Johst, K., and Person, Y. On the multicolor ramsey number of a graph with m edges, 2013.

[9] Kunen, K. *Set theory : an introduction to independence proofs*. Studies in logic and the foundations of mathematics. Elsevier science, Amsterdam, Lausanne, New York, 1980.

[10] Lasjaunias, A., and Yao, J.-Y. On certain recurrent and automatic sequences in finite fields, 2016.

[11] Linsen, L. Esm4a - numerical methods. Lecture Slides, 2015.

[12] Puleo, G. J. On a conjecture of erdos, gallai, and tuza, 2014.

[13] Yang, S. K-theoretic chow groups of derived categories of schemes-on a question by green-griffiths, 2013.

[14] You, C. Simplified and equivalent characterizations of banach limit functional and strong almost convergence, 2009.