Parsing and Disambiguation of Symbolic Mathematics in the Naproche System

Marcos Cramer, Peter Koepke, and Bernhard Schröder

University of Bonn and University of Duisburg-Essen
{cramer,koepke}@math.uni-bonn.de, bernhard.schroeder@uni-due.de
http://www.naproche.net

Abstract. The Naproche system is a system for linguistically analysing and proof-checking mathematical texts written in a controlled natural language. The aim is to have an input language that is as close as possible to the language that mathematicians actually use when writing textbooks or papers.

Mathematical texts consist of a combination of natural language and symbolic mathematics, with symbolic mathematics obeying its own syntactic rules. We discuss the difficulties that a program for parsing and disambiguating symbolic mathematics must face and present how these difficulties have been tackled in the Naproche system. One of these difficulties is the fact that information provided in the preceding context – including information provided in natural language – can influence the way a symbolic expression has to be disambiguated.

Keywords: Symbolic mathematics, mathematical formulae, Naproche, formula parsing.

1 Introduction

In recent years, formal mathematics has seen remarkable progress. Proofs of significant mathematical theorems like the Jordan Curve Theorem or the Prime Number Theorem have been formalized and formally checked in powerful systems like HOL light [10][1]. Some large scale formalization projects related to current research mathematics are under way [11].

This creates a demand for presenting developments in formal mathematics within the ordinary language used by mathematics. De Bruijn [3], one of the pioneers of formal mathematics, formulated :

> [...] the Automath project tries to bring communication with machines in harmony with the usual communication between people.

He describes his approach as follows:

> So I got to studying the structure of mathematics by starting from the existing mathematical language and from the need to make such language understandable for machines. I think we might call that approach "natural". "Natural deduction" is a part of it. [...] the word "natural" [...] refers to the reasoning habits of many centuries, [...]

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 180–195, 2011.

The Naproche project[1] (NAtural language PROof CHEcking) tries to take up some of the challenges of natural formal mathematics. Due to the tremendous difficulties in in the deep semantic analysis of natural language and common (mathematical) arguments, known from linguistics and artificial intelligence, at the moment, success can only be limited: the general problem appears to be "AI-hard". Naproche therefore restricts itself to a kind of existence proof: to show that one can formulate substantial mathematical texts, so that they are acceptable texts for ordinary mathematicians, but simultaneously computer readable and checkable for linguistic and mathematical correctness. This involves several subtasks, in particular the development of a controlled natural language for mathematics with corresponding parsing mechanisms, parsing of (LaTeX-style) mathematical formulae, translations into first-order formats, and the connection with strong automatic provers to supply missing "trivial" proof elements. Currently we are in the process of reformulating parts of E. Landau's *Grundlagen der Analysis* [16] in the Naproche controlled language and simultaneously developing the Naproche formal mathematics system.

In this paper we address the problem of parsing mathematical formulae embedded in some mathematical text. Despite the widespread assumption that mathematical formulae are exact, they are often very ambiguous in a way that (standard) typing does not sufficiently resolve. We study situations in which further information, mathematical and linguistic, from the ambient text has to be taken into account.

We demonstrate with a number of representative examples, that fairly complex formulae, written in "simple LaTeX", can be correctly parsed, and we expect to be able to parse nearly all formulae that will be coming up in the formalization of Landau's *Grundlagen*. Unfortunately, a further evaluation of our methods appears to be problematic at this moment. Due to the many styles of writing mathematics and coding it in LaTeX we cannot hope to be able to parse arbitrary formulae from large repositories of mathematical material. Like with the controlled input language we are dependent on adequate reformulations, where adequacy has to be judged by experts in the subject. Note that reformulations and reformalizations are ubiquitous in formal mathematics anyway, to get proofs to work. To determine "degrees of naturality" is notoriously problematic, as is well-known from experimental linguistics, and has to be left to the reader's appreciation.

After presenting the Naproche System in section 2, we exhibit the flexibility of symbolic mathematics in section 3, explaining why this flexibility makes symbolic mathematics so difficult to parse and disambiguate. In section 4, we proceed to discuss possible approaches to tackling these difficulties. Our solution to these problems is presented in sections 5 to 8, which are followed by a section on related work and a conclusion.

[1] Naproche is a joint initiative of Peter Koepke (Mathematics, University of Bonn) and Bernhard Schröder (Linguistics, University of Duisburg-Essen). The Naproche system is technically supported by Gregor Büchel from the University of Applied Sciences in Cologne.

2 The Naproche System

A central goal of Naproche is to develop and implement a controlled natural language (CNL) for mathematical texts which can be transformed automatically into equivalent formulae of first-order logic using methods of computational linguistics [7]. We have developed a prototypical *Naproche system*, which can automatically check texts written in the Naproche CNL for logical correctness [6]. We test this system by reformulating parts of mathematical textbooks and the basics of mathematical theories in the Naproche CNL and having the resulting texts automatically checked.

The Naproche system transforms a given input text into a logical representation of its content, called a *Proof Representation Structure* (PRS). The PRS is checked for logical correctness with the aid of *automated theorem provers*. More precisely, the PRS creation and checking process is performed sentence by sentence: For every sentence in the text, the system first parses the sentence and updates the PRS accordingly; afterwards, it checks the logical correctness of the additions that have been made to the PRS. The checking algorithm keeps a list of first order formulae it considers to be true, called premises, which gets continuously updated during the checking process, and which represents the mathematical knowledge that a reader of the text has collected up to a given point in the text.

3 Symbolic Mathematics

One of the conspicuous features of the language of mathematics is the way it integrates mathematical symbols into natural language material. The mathematical symbols are combined to *mathematical expressions*, which are often referred to as *mathematical formulae* or *mathematical terms* depending on whether they express propositions or whether they refer to a mathematical object. Already at first sight, a whole variety of syntactic rules are encountered for forming complex terms and formulae out of simpler ones; a basic classification of these was provided by Ranta [18]:

- There are infix operators that are used to combine two terms to one complex term, e.g. the $+$ symbol in $m + n$ or $\frac{1}{x} + \frac{x}{1+x}$.
- There are suffix operators that are added after a term to form another term, e.g. the ! symbol in $n!$.
- There are prefix operators that are added in front of a term to form another term, e.g. sin in $\sin x$.
- There are infix relation symbols used to construct a formula out of two terms, e.g. the $<$ symbol in $x < 2$.

As noted by Ganesalingam [9], "this simple classification is adequate for the fragment Ranta is considering, but does not come close to capturing the breadth of symbolic material in mathematics as a whole." It does not include notations

like $[K : k]$ for the degree of a field extension, it does not allow infix operators to have an internal structure, like the $*_G$ in $a *_G b$ for denoting multiplication in a group G, nor does it account for the common way of expressing multiplication by concatenation, as in "$a(b + c)$".

Another kind of prefix operator not mentioned by Ranta is the one that requires its argument(s) to be bracketed, e.g. f in $f(x)$. (Of course, the argument of a prefix operator like sin might also be bracketed, but generally this is done only if the argument is complex and the brackets are needed for making sure the term is disambiguated correctly.) This is even the standard syntax for applying functions to their arguments, in the sense that a newly defined function would be used in this way unless its definition already specifies that it should be used in another way.

The expression $a(x + y)$ can be understood in two completely different ways, depending on what kind of meaning is given to a: If a is a function symbol and $x + y$ denotes a legitimate argument for it, then $a(x + y)$ would be understood to be the result of applying the function a to $x + y$. If on the other hand a, x and y are – for example – all real numbers, then $a(x + y)$ would be understood as the product of a and $x+y$. Now whether a is a function or a real number should have been specified (whether explicitly or implicitly) in the preceding text. This is why we say that the disambiguation of symbolic expressions requires information from the preceding text, and this information might have been provided in natural language rather than in a symbolic way.

As one can already see from these sketches of symbolic mathematics, the task of parsing and disambiguating symbolic expressions has a lot of aspects.

One of the issues that has to be surmounted in order to treat mathematical symbolism directly in a computer program is its two-dimensionality. Mathematicians extensively use superscripts and subscripts and put terms above other terms as in the fraction notation. Naproche has already for some time adopted LaTeX for its input, so that in this paper we restrict ourselves to parsing and disambiguating the LaTeX code that is used for generating mathematical formulae[2]. The reversion of a pictorial symbolic input into a LaTeX input or another linearisation of it is certainly an interesting undertaking, but outside the scope of this paper.

In order to cope efficiently with the diversity of possible LaTeX codes for a given symbolic output – e.g. `a^b` and `a^{b}` both producing a^b – we normalise the LaTeX input before the actual parsing process, in this case to `a^{b}`. For the rest of this paper, we use this normalised LaTeX code whenever it is necessary for the explanation; when the LaTeX code is not necessary for the explanation, we use the typographic notation that depicts the mathematical symbols as they are commonly drawn and printed.

[2] We restrict ourselves to standard LaTeX, i.e. without any user-defined macros. Additionally, we in some respects require the author to use *neat* LaTeX, e.g. to write the sine function using `\sin` rather than `sin` in order to distinguish it from the concatenation of the three variables s, i and n.

4 Possible Approaches to Disambiguation

If $a(x + y)$ is to be read as the value of a function a at $x + y$, then a has to be a function. This requirement can be understood in two different ways, which are nevertheless related and combinable: Either it is considered to be a *presupposition* of the symbolic expression $a(x + y)$; in this case, the linguistic theory of presuppositions with all its elaborations might be considered to be applicable to this case [13][8]. Or it is considered to be a *type judgement* about a; in this case, it should be possible to formulate a type system for symbolic mathematics and reuse existing ideas from type theory to describe and work with this type system.

In the context of a proof checking system like Naproche, presuppositions have to be checked for their correctness, i.e. the presuppositions of an expression have to be checked to logically follow from the premises that are available at the point where the expression is used [8]. One possible approach that we took into consideration for disambiguating symbolic expressions was to check their presuppositions already during the parsing process, so that readings which lead to wrong presuppositions would already be blocked during the parsing process. This approach, however, has turned out to be far too inefficient: It would involve constantly calling automatic theorem provers during the parsing process and waiting for their output before continuing the parsing.

Another approach is to rely on a type system rather than on presupposition fulfillment for disambiguating symbolic mathematics. In that case, one needs a very rich and flexible type system for symbolic mathematics. Such a type system has been developed ingeniously by Ganesalingam [9]. However, to attain the richness of the type system required for handling all kinds of ambiguities that can arise, he was obliged to require the author of a text that is to be parsed by his system to write sentences whose sole function is to create types that are needed for certain disambiguations. Given that the goal of Naproche is to stay as close as possible to the language that mathematicians naturally use, this aspect of Ganesalingam's approach made it less attractive for us.

So we decided to take up a combined approach, in which there is a relatively simple type system capable of blocking most unwanted readings during the parsing process, with the remaining readings being filtered by checking their presuppositions.

5 A Type System for Symbolic Mathematics

In the type system that Naproche uses for handling symbolic mathematics, there are two basic types: i for individuals and o for formulae expressing propositions. Apart from these, there are function types of the form $[t_1, ...t_n] \to t$, where $t_1, \ldots, t_n$ are the types of the arguments the function takes and t is the type of the term that we get when we apply this function to legitimate arguments. So unlike in the Simple Theory of Types (STT) [5], we have an inherent way of handling multi-argument functions. In STT, multi-argument functions must

be simulated by functions whose codomain type is again a function type, e.g. + would be considered a function from natural numbers to functions from natural numbers to natural numbers. We, however, want to use types to describe how mathematical formulae are structured in actual mathematical texts, and for this purpose it is better to have multi-argument functions inherently in the type system.

Note that formulae are also considered terms (namely terms of type o), and that the logical connectors are considered functions of type $[o, o] \to o$ or $[o] \to o$. Even quantifiers are considered to be functions, namely two-place functions whose first argument has to be a variable and whose second argument is a term of type o that may depend on the variable. We formalise this by writing the type of quantifiers as $[var(\_, X), X - o] \to o]$, where $var(\_, X)$ means that the first argument is a variable X of type _ (i.e. of any type), and $X - o$ means that the second argument is a term of type o possibly depending on X[3].

5.1 Syntactic Types

As already discussed in section 3, functions can behave in syntactically different ways. For example, + is generally used as an infix function symbol ("$a + b$"), whereas the notation $f(x)$ uses a function symbol f in prefix position with its argument in brackets. In Naproche, we distinguish six basic ways in which function symbols behave syntactically, and call these the *syntactic types* of the corresponding function symbols:

1. *infix*: Two-argument function symbol placed between its arguments (e.g. + in $n + m$).
2. *suffix*: One-argument function symbol placed after its argument (e.g. ! in $n!$).
3. *prefix*: One-argument function symbol placed before its argument (e.g. sin in $\sin x$).
4. *classical*: Function symbol with one or more arguments preceding its arguments, which are bracketed and separated by commas (e.g. f in $f(x)$ or $f(x, y)$).
5. *quantifier*: Two-place function symbol placed before its two arguments, where the arguments have to have types of the form $var(t_1, X)$ and $X - t_2$, and where the first argument position may be filled with a variable list rather than a single variable (e.g. $\forall x, y\ R(x, y)$).
6. *circumfix*: Expression for a function with one or more arguments, which are embedded into a predefined string of symbols, with at least one symbol at the beginning, at the end and between any two successive arguments (e.g. the degree of a field extension, $[K : k]$, considered as a two-place function depending on K and k). The *name* of a circumfix function is this predefined string with `[arg]` denoting the positions of its arguments. For example, the name of the field extension function is `[[arg]:[arg]]`.

[3] We use Prolog-like notation, i.e. capital letters for variables and _ for an anonymous variable, when describing the type system.

Now consider an example from real analysis, namely the differentiation function, which is a function from differentiable real functions to real functions, sending any f to its derivative f'. When written in this $'$-notation, this function clearly has syntactic type *suffix*. But when we write $f'(x)$, we use the complex function name f' as a function with syntactic type *classical*. Now this does not seem to depend on the syntactic type of f: Suppose we have defined an extension of the factorial function ! to the reals (e.g. by $x! := \Gamma(x+1)$ using the Gamma function [12]). If we then apply its derivative $!'$ to some real x, we would write $!'(x)$ and not $x!'$[4]. So it seems to be inherent in the way the differentiation function symbol $'$ is used that the complex function name it produces is of syntactic type *classical*. We formalise this by saying that $'$ is of syntactic type *[suffix,classical]*. This means that its basic syntactic type is *suffix*, and the syntactic type of any function name whose head is $'$ is *classical*.

This machinery makes it possible to correctly handle many complicated notations: For example, exponentiation is treated as a function of syntactic type *[circumfix,suffix]* and of type $[i] => ([i] => i))$ (so in this case the notation we use makes us treat this multiple-argument function in the way such functions are treated in STT rather than using an inherent multiple-function type), where the name of the circumfix function is `^{[arg]}`. In the case of `x^{y}`, this function is first applied to `y`, yielding `^{y}`, which is considered a suffix function, so that applying it to x yields `x^{y}`.

In Naproche we distinguish two different kinds of math modes: The first is used for formulae (like $x = y^2$) and terms that serve as definite noun phrases (like $2x - 1$). The second is used for quantified terms, like the first two symbolic expressions in "For every x there is some $f(x)$ such that $R(x, f(x))$." Terms of the first kind are parsed by what we call the *normal formula grammar*, and terms of the second kind are parsed by what we call the *quantterm grammar*.

6 Normal Formula Grammar

Below we describe the normal formula grammar semi-formally by first listing (in a formal DCG-notation with Prolog-like syntax) a list of simplified grammar rules that any term must obey and then describing informally additional constraints that any term must satisfy in order to be actually parsed by the grammar. The constituent "term" used in the DCG rules below has an argument specifying the syntactic type of the term (i.e. a list of basic syntactic types). We use the variable name ST for a variable ranging over syntactic types.

Simplified normal formula grammar

term(ST) → term([classical|ST]), ['('], term_list, [')'].
term(ST) → term(_), term([suffix|ST]).

[4] Since this is a made-up example, we should add that our intuition as to what notation would be appropriate here has been confirmed by a number of mathematicians from the University of Bonn.

term(ST) → term([prefix|ST]), term(_).
term(ST) → term([quantifier|ST]), variable_list, term(_).
term(ST) → term(_), term([infix|ST]), term(_).
term(ST) → circumfix_term(ST).
term(ST) → ['('], term(ST), [')'].
term(ST) → variable(ST).

term_list → term(_), [','], term_list.
term_list → term(_).

variable_list → quantified_variable, [','], variable_list.
variable_list → quantified_variable.
quantified_variable → [_].
variable → [_].

For every predefined or accessible[5] *variable V of syntactic type ST, add a rule of the following form to the grammar:*
variable(ST) → V.

For every accessible circumfix function of syntactic type ST and name $S_1^1 \dots S_1^{n_1}[arg]S_2^1 \dots S_2^{n_2}[arg] \dots [arg]S_m^1 \dots S_m^{n_m}$*, add a rule of the following form to the grammar:*
circumfix_term(ST) →
$[S_1^1], \dots, [S_1^{n_1}], term(\_), [S_2^1], \dots, [S_2^{n_2}], term(\_), \dots, term(\_), [S_m^1], \dots, [S_m^{n_m}]$.

6.1 Operator Priorities

Syntactic disambiguation principles like the precedence of multiplication and division operators over addition and subtraction operators are encoded into the grammar using predefined operator priorities. We use the following operator priorities (in the order of decreasing precedence):

- +, −, → and ↔
- Prefix functions
- Suffix functions
- Other infix functions

Additionally, there is a principle which overrides the above operator priorities, namely that the operators used to form atomic formulae always have a higher precedence than the operators used to combine atomic formulae into complex formulae.

As an example for the functioning of these syntactic disambiguation principles,

[5] Given that Naproche's Proof Representation Structures are a variant of Discourse Representation Structures [14], "accessible" is to be understood as in Discourse Representation Theory. Basically, an accessible variable is a variable that was introduced in the preceding text that we can refer to by using the same variable name.

(1) $x + yz = \sin an! \wedge x = y \rightarrow z - y + z = 0$

is disambiguated as

(2) $(((x + (yz)) = \sin(a(n!))) \wedge (x = y)) \rightarrow (((z - y) + z) = 0)$.

In all cases that we are aware of, these syntactic disambiguation principles lead to an intuitive reading of the symbolic expression.

6.2 Defaultness of the Syntactic Type *Classical*

As already alluded in section 3, the syntactic type *classical* is the default syntactic type for newly introduced functions. This principle is implemented into the grammar by an additional constraint that in the second to fifth DCG rule specified above, as well as in the rule "variable → [_].", the syntactic type of a term may not be instantiated to *infix*, *prefix*, *quantifier*, *suffix* or *circumfix*. For example, the requirement of the final term to have "suffix" as syntactic type in the second rule means that this syntactic type must already be associated with the term when parsing it and may not be attached to the term afterwards. There is a limited list of predefined infix function symbol $(\cdot, +, -, *, ., \circ, /)$ for which this constraint does not apply.

In practice, this constraint means that when you are quantifying over a function, this function may be used with classical syntactic type or, if a preferred infix function symbol is used, with infix syntactic type, but not with prefix, suffix or quantifier syntactic type. So (3) and (4) are allowed, but (5), (6) and (7) (with z read as an infix, f as a prefix and g as a suffix function symbol) are not allowed.

(3) $\exists f\ f(a) = 0$

(4) $\exists *\ x * x = x$

(5) $\exists z\ xzx = x$

(6) $\exists f\ fa = 0$

(7) $\exists g\ ag = 0$

The defaultness of the syntactic type *classical* also explains why we don't formalise functions used in this syntactic way as circumfix functions. This would certainly be possible: A one-argument classical function f could also be considered a circumfix function with name `f([arg])`. However, this way we would not be able to account for the fact that a function that was introduced without fixing its syntactic type can be used with syntactic type *classical*.

6.3 Predefined Variables

It should be noted that we do not make the distinction between variables and constants that is usually made in the syntax of first-order logic and many other logical systems. In the semi-formal language of mathematics, there is a continuum between variable-like and constant-like expressions; this continuum is captured in Naproche through the use of dynamic quantification inherent in Discourse Representation Theory [14], so that the bivalent distinction used in first-order logic is not needed.

However, logical constants are still treated in a special way, namely as "predefined variables". These are also given a predefined type and syntactic type as follows:

Predefined variable	Type	Syntactic type
$\rightarrow$, $\leftrightarrow$, $\wedge$ and $\vee$	$[o, o] \rightarrow o$	*infix*
$\neg$	$[o] \rightarrow o$	*prefix*
$\forall$ and $\exists$	$[var(\_, X), X - o] \rightarrow o$	*quantifier*
$=$	$[T, T] \rightarrow o$[6]	*infix*
$\neq$	$[\_, \_] \rightarrow o$[7]	*infix*

6.4 Kinds of Variables

In the parsing process we distinguish different kinds of variables:

- Predefined variables (logical constants)
- Bound variables
- Variables that were implicitly introduced earlier on in the symbolic expression and are now reused
- Accessible variables whose antecedent is in the same sentence
- Accessible variables whose antecedent is in a preceding sentence
- Implicitly introduced variables

When trying to parse a variable, we always first try to parse it according to a variable kind higher up in the above list before trying the kinds lower down in the list. Once a variable has been parsed in one way, it may no longer be parsed in such a way as to be of a kind that is mentioned later in the above list than the kind that it has already been assigned. This means, for example, that if x is accessible and we parse $\exists x\ x + x = x$, then all instances of x in this formula are bound by the existential quantifier; none of the instances of x refers to the accessible variable.

6.5 Coverage of the Formula Grammar

The formula grammar can cope with almost all terms that serve as definite noun phrases and formulae found in mathematical texts. Here is a list of formulae that can be correctly parsed and disambiguated by it:

[6] i.e. the two arguments must be of the same type.
[7] i.e. the two arguments may be of distinct types.

$$x(y+z) = 0$$

$$x = y < z$$

$$x *_G x = x$$

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

$$x_0 \lim_{x \to x_0} f(x^2) = 2f(\frac{x_0^{x_0}}{2}) \neq f'(N!)$$

$$T = m_0 \frac{l^2}{2}((cos\varphi_0\varphi_0')^2 + (-sin\varphi_0\varphi_0')^2)$$

Of course, these formulae can only be parsed if the types and syntactic types of the function symbols appearing in them are known in advance. This information is created by the *quantterm* grammar described in section 7 when the functions are introduced.

There are some limitations of the current implementation of the formula grammar that we are aware of: Firstly our formula grammar can only handle variable binding if the occurrence of the variable that binds the other occurrences precedes the bound occurences. Hence the formula grammar cannot handle the integral notation of the form $\int f(x)dx$, where the first occurence of x is bound by the final occurence of x. Furthermore, the formula grammar can currently not cope with formula fragments like "$= 0$" nor with formulas containing triple dots like "$n \in \{1, \ldots, N\}$". However, we believe that the approach presented in this paper constitutes a framework for tackling even these harder cases, i.e. that the current limitations are not due to principle limitations of our approach, but rather due to the prototypical character of the implementation.

As already mentioned in the introduction, a quantitative evaluation of the coverage of the formula grammar is a highly nontrivial task. It involves reformulating the natural language context of the formulae in a controlled natural language, so that a full semantic analysis of the context can be achieved. This has so far only be accomplished for the first chapter of Landau's *Grundlagen der Analysis*, where the formula grammar parsed and correctly disambiguated all formulae [4].

7 Quantterm Grammar

Consider the following example text from [15]:

(8) Suppose that, for each vertex v of K, there is a vertex $g(v)$ of L such that $f(st_K(v)) \subset st_L(g(v))$. Then g is a simplicial map $V(K) \to V(L)$, and $|g| \backsimeq f$.

Here the natural language quantification "there is a vertex $g(v)$" locally introduces a new vertex to the discourse; but since the choice of the vertex depends on v and we are universally quantifying over v, it globally introduces a function

g to the discourse. In the next sentence there is an explicit reference to this implicitly introduced function.

Quantterms are symbolic expressions that appear in the scope of a natural language quantification, and are either just simple variables (in which case we call them *simple quantterms*), or, like in the above example, complex expressions that implicitly introduce a function to the discourse.

In order to discuss the functioning of quantterms, consider the following three example sentences:

(9) There is some y such that $R(y)$.

(10) For every x there is some y such that $R(x, y)$.

(11) For every x there is some $g(x)$ such that $R(x, g(x))$.

As described in [8], the premise added to the premise list for representing the information from (9) would not be $\exists y\ R(y)$, but $R(c_y)$ for a new constant symbol c_y. The reason for this replacement of existentially quantified variables by constant symbols is that the first-order quantifier $\exists$ does not have the dynamic properties of the natural language quantification with "there is": After stating sentence (9), we can later use the symbol y to refer to the same object that was introduced by this sentence. If we represented the content of the sentence by $\exists y\ R(y)$, then the scope of the y would only be this formula and could thus not include later uses of y. By using $R(c_y)$ for the content of (9) and replacing later uses of y by c_y, we do get the wanted coreference between the y in (9) and the later y.

This replacement of an existentially quantified variable by a constant is a special case of *skolemization* [2][8]. In the representation of sentence (10) we make use of the more general kind of skolemization, which involves introducing new function symbols rather than new constant symbols. Its representation becomes $\forall x\ R(x, f_y(x))$, where f_y is a newly introduced function symbol; $f_y(x)$ replaces all occurrences of y in the scope of $\forall x$, where the argument x makes explicit that the choice of y depends on the value of x.

In the case of sentence (11), $g(x)$ can at first be considered to just be a complex variable name, usable in this very form later on in the sentence. Just as in the case of sentence (10), we skolemize this variable and make its dependencies explicit; in this case $g(x)$ depends on x. All this is the same as for sentence (10). But now we have to take into account that the author made this dependency explicit by writing $g(x)$ instead of y. This makes it possible to identify g with the skolem function that skolemization gives rise to, and to use this g as a function outside the universally quantified sentence in which $g(x)$ was introduced.

Now let us look at a somewhat more complex example:

(12) For all x, y there is some $g_x(y)$ such that $R(x, y, g_x(y))$.

After this sentence, we want to be able to use a function of syntactic type *[circumfix,classical]* named `g_{[arg]}`. So already when parsing the quantterm, we want to identify this syntactic type and name of the head function. This is

done by recursively allowing the head function of a quantterm to be again a quantterm. So in the case of $g(x)$ in (11), g may again be a quantterm, and is actually a simple quantterm. Now in the case of (12), g_x is first identified as head function of syntactic type *classical*, and is further analysed as quantterm. This further analysis recognises g_x as circumfix function `g_{[arg]}`.

7.1 Disambiguating Quantterms

Now one problem is that the quantterm grammar finds a number of possible readings for any input. For example, $f(x, y)$ can be interpreted in four ways:

1. as two-place classical function f (depending on x and y)
2. as two-place circumfix function `f([arg],[arg])` (depending on x and y)
3. as one-place circumfix function `f([arg],y)` (depending on x)
4. as one-place circumfix function `f(x,[arg])` (depending on y).

Here we want to choose the first reading as the preferred reading to be used by the program. This is done by a special algorithm for selecting the preferred reading, which works as follows:

- Non-*circumfix* readings are always preferred over *circumfix* readings.
- Between two *circumfix* readings, one is preferred over the other if its *circumfix* name has an `[arg]` at a place, where the other has a symbol.
- A reading that has *classical* in the second position of the syntactic type list is preferred over one that does not. (This principle is needed, for example, to ensure that in $f'(x)$, $'$ is interpreted as a suffix function making f' classical rather than as a classical function making $'(x)$ a suffix function.)
- When none of the above rules decides which reading is better, we recursively check which head function is preferred by those rules.

8 Disambiguation after Parsing

As mentioned in section 4, the type system is not capable of blocking all unwanted readings. This is due to the fact that our type system is not fine-grained enough. All objects that are not functions are of the same type, namely i. So, for example, both natural numbers and sets would be of the type i. If one has defined that for sets A, B, the expression A^B denotes the set of functions from A to B, and one has furthermore defined that for natural numbers m, n, the expression m^n denotes the nth power of m, then one has defined two functions of syntactic type *[circumfix,suffix]* and type $[i] \to ([i] \to i)$, both named `^{[arg]}`. Since their name, type and syntactic type are identical, they are indistinguishable during the parsing process. Thus, the ambiguity arising from this notational clash has to be resolved after the parsing process.

After updating the Proof Representation Structure with the representation of a parsed sentence, the Naproche system checks this added representation for logical correctness. This checking process involves the checking of presuppositions

[8]. The two just mentioned functions of equal name, type and syntactic type would trigger different presuppositions: The first would trigger the presupposition that both of its arguments are sets, whereas the second would trigger the presupposition that both of its arguments are numbers. Since it is not possible for both of these presuppositions to be fulfilled for a given pair of arguments, the ambiguity can certainly be removed in the process of checking the presuppositions[8].

It is also possible that the type information needed for disambiguating a symbolic expression is only available after the completion of the parsing process for that expression. Suppose, for example, that a user has defined a relation "$>$" on both natural numbers and functions of natural numbers, and uses the symbol 1 not only for the natural number 1, but also for the identity function. Now consider the following sentence:

(13) For all $x > 1$ such that $x^2 + 1$ is prime we have $R(x)$.

If the exponential notation x^2 is only defined for numbers and not for functions, then this sentence can be disambiguated using type information: x has to be of type i in "$x^2 + 1$" and therefore also in "$x > 1$", and so the "$>$" in "$x > 1$" refers to the relation on numbers and not the one on functions. But this type-based disambiguation of "$x > 1$" was not possible during the process of parsing "$x > 1$", because at that point "$x^2 + 1$" had not yet been parsed. In order to handle such type-based disambiguations that occur after that parsing of an expression, we use *type-dependency graphs*, which specify which reading of an expression depends on which type judgements. A detailed description of type-dependency graphs would, however, go beyond the scope of this paper.

9 Related Work

For understandable reasons, most formal mathematics systems simplify symbolic mathematics to a purely formal language, thus avoiding the issues that our paper is intended to tackle. Even languages of systems that clearly aim at a higher degree of naturality, like Mizar [17] and SAD [19], still largely treat the symbolic parts of mathematical texts like a formal language. The only work outside Naproche we are aware of that recognises the problem of parsing and disambiguating symbolic mathematics as intertwined with the natural language component of mathematical texts and as of a completely different kind than parsing formal languages is Ganesalingam [9]. He has analysed the language of mathematics – including symbolic mathematics – in much detail and developed a very ingenious theory for "a computer language which closely resembles the

[8] It is of course also possible that a user defines two clashing notations whose presuppositions may be fulfilled by the same argument(s); this, however, is almost certainly bad style, so that the user should get a warning from the system when this happens; nevertheless, the system does always choose one reading as the preferred one, using other heuristics, for example preferring notations that were defined later over ones that were defined earlier.

language used by human mathematicians in publications"[9]. We owe him a lot, since his work has enhanced our understanding of the language of mathematics and has helped us to develop the ideas presented in this paper. There are, however, two main differences between Ganesalingam's approach and ours:

Firstly, he has a methodological principle that no mathematical content is encoded directly into his theory, and he considers such syntactic disambiguation principles as the precedence of multiplication over addition as part of mathematical content[10] Thus he does not encode such principles into his theory, but requires the author to write sentences of the following form in order to get the desired disambiguation of arithmetic expressions:

(14) If m, n and k are natural numbers, then "$m + nk$" means "$m + (nk)$".

We on the other hand do not want to require the author to write things that mathematicians do not normally write, and so decided to encode some basic syntactic disambiguation principles directly into our theory.

Secondly, as already alluded in section 4, he relies much more heavily on a type system than we do for disambiguating symbolic mathematics. This is due to the fact that he does not include presuppositions into the disambiguation machinery. By making use of presuppositions for disambiguation, we were able to attain similar goals as Ganesalingam with a much more coarse type system. One of the benefits of the coarseness of the type system is that we do not require the author to make statements whose only goal is to influence the typing of symbolic material.

10 Conclusion

We have presented the difficulties that a computer program for analysing mathematical texts faces with respect to symbolic mathematics, given that the input language is to be as similar as possible to the language that mathematicians commonly use in journals and textbooks. We have described how these difficulties are solved in the Naproche system, and compared this solution to Ganesalingam's solution.

References

1. Avigad, J., Donnelly, K., Gray, D., Raff, P.: A formally verified proof of the prime number theorem. ACM Transactions on Computational Logic 9(1:2) (2007)
2. Brachman, R., Levesque, H.: Knowledge Representation and Reasoning. Morgan Kaufmann Publishers, Massachusetts (2004)
3. de Bruijn, R.G.: Reflections on Automath. In: Nederpelt, R.P., et al. (eds.) Selected Papers on Automath. Studies in Logic, vol. 133, pp. 201–228, 215. Elsevier, Amsterdam (1994)

[9] Page 9 in [9]
[10] Page 105 in [9]).

4. Carl, M., Cramer, M., Kühlwein, D.: Chapter 1 from Landau in Naproche 0.5 (2011), http://naproche.net/downloads/2011/landauChapter1.pdf
5. Church, A.: A Formulation of the Simple Theory of Types. The Journal of Symbolic Logic 5(2), 56–68 (1940)
6. Cramer, M., Koepke, P., Kühlwein, D., Schröder, B.: The Naproche System. Calculemus, Emerging Trend Paper (2009)
7. Cramer, M., Fisseni, B., Koepke, P., Kühlwein, D., Schröder, B., Veldman, J.: The Naproche Project – Controlled Natural Language Proof Checking of Mathematical Texts. In: Fuchs, N.E. (ed.) CNL 2009. LNCS, vol. 5972, pp. 170–186. Springer, Heidelberg (2010)
8. Cramer, M., Kühlwein, D., Schröder, B.: Presupposition Projection and Accommodation in Mathematical Texts. In: Semantic Approaches in Natural Language Processing: Proceedings of the Conference on Natural Language Processing 2010 (KONVENS), pp. 29–36. Universaar (2010)
9. Ganesalingam, M.: The Language of Mathematics, PhD thesis, University of Cambridge (2009)
10. Hales, T.: Jordan's proof of the Jordan Curve theorem. Studies in Logic, Grammar and Rhetoric 10(23) (2007)
11. Hales, T.: Introduction to the Flyspeck Project. Dagstuhl Seminar Proceedings (2006)
12. Heuser, H.: Lehrbuch der Analysis. In: Teil 2, 6th edn., B.G. Teubner, Stuttgart (1991)
13. Kadmon, N.: Formal Pragmatics. Wiley-Blackwell, Oxford, UK (2001)
14. Kamp, H., Reyle, U.: From Discourse to Logic: Introduction to Model-theoretic Semantics of Natural Language. Kluwer Academic Publishers, Dordrecht (1993)
15. Lackenby, M.: Topology and Groups. Lecture Notes (2008), http://people.maths.ox.ac.uk/lackenby/tg050908.pdf
16. Landau, E.: Grundlagen der Analysis, 3rd edn (1960)
17. Matuszewski, R., Rudnicki, P.: Mizar: The first 30 years. Mechanized Mathematics and Its Applications 4 (2005)
18. Ranta, A.: Structure grammaticales dans le français mathématique II (suite et fin). Mathématiques, Informatique et Sciences Humaines 139, 5–36 (1997)
19. Verchinine, K., Lyaletski, A., Paskevich, A.: System for Automated Deduction (SAD): a tool for proof verification. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 398–403. Springer, Heidelberg (2007)