Reaping the Benefits of Modularization in Flexiformal Mathematics by GF-based AST Transformations

Josefin Kelber[1], Michael Kohlhase[2], Jan Frederik Schaefer[2], and Marcel Schütz[2]

[1] Technical University of Munich (TUM), Germany
[2] Computer Science, FAU Erlangen-Nürnberg, Germany

Abstract. Flexiformal documents – i.e. documents with embedded semantic annotations that make some aspects of their content machine-actionable – can be instrumented to make interaction with the underlying knowledge more efficient and effective. Fostering such interactions via semantic services has proven very successful in university education, but the practical applicability is limited by the cost of flexiformalization. A method for lowering (flexi)-formalization costs is to use modular representations to profit from enhanced source sharing and induced (generated) content. In fully formal environments this is well-understood and implemented in many systems.
In this paper we show that many of the formal techniques carry over to the informal setting if we parse (rigorous) natural language with a semantically optimized grammar and work on abstract syntax trees instead of formulae.
We present *i)* a set of use cases for generating learning material to be used in an educational setting – concretely in the field of theoretical computer science, *ii)* a GF grammar that allows to syntactically analyze the underlying language fragment, *iii)* a set of AST-to-AST simplifications that can be used to fine-tune the wording and formulae of the generated content and adapt it to the scientific jargon, and *iv)* a prototypical implementation that shows the technique in action.

Keywords: Flexiformal Mathematics · FTML · Mathematical Language · Theory Morphisms · GF · NLP

1 Introduction

Scientific knowledge is highly inter-related. Relations range from the mappings that underlie application of theoretical results – objects, concepts and relations in the theory are mapped to objects in the application domain which obey the theory assumptions – to inter-theory interpretations and refinements, which allow to import insights, constructions, methods, and arguments from the source theory into the target theory, called **recontextualization** in [KS24].

V. de Paiva and P. Koepke (Eds.): CICM 2025, LNAI 16136, pp. 191–207, 2025.
https://doi.org/10.1007/978-3-032-07021-0_11

In modular representation formats, such relations can be made explicit as first-class citizens – called **theory morphisms**, views, realizations, or just interpretations – which can be inspected, composed, applied, and reasoned with. In fully formal representation formats, this is a very well-understood process, and most formal mathematical libraries utilize the synergies induced by theory morphisms. In type theoretic contexts, these are often implemented as functions between record types. Here we follow a heterogeneous notion of theories (see [MRK18]).

Reaping the synergies is much more difficult in **flexiformal** representation formats where the semantics of informal content is (partially) elucidated by semantic annotations (see [Koh13]): instead of definition expansion and possibly simplification in formulae, we need to manipulate informal technical language. In [KS24] we have shown that transporting quiz problems given in mathematical vernacular is possible in principle using a simple template-based approach: given sufficient (but still natural) semantic annotations of the source quiz problems, flexiformal theory morphisms have the necessary information (and language snippets) to assemble relatively natural-looking quiz problems in the context of the target theory. Of course, the template-based approach only allows very limited simplifications of the generated language, which can also contain grammatical problems like failure of agreement or inflection, but more generally loses much of the conciseness we are used to in mathematical language.

Contribution In this paper we want to lift some of these limitations: instead of composing semantically annotated LaTeX snippets via templates, we parse the natural language with the Grammatical Framework (GF [Ran04; GF]), do the language composition at the level of GF abstract syntax trees, and leave the linearization of these to the GF system to guarantee grammatical correctness of the result. This had been anticipated in [KS24], but needed a semantic grammar for mathematics, which we have developed for this paper.

A side effect of this move from LaTeX strings to GF abstract syntax trees (essentially λ-terms in LF [Pfe01]) is that we can do more complex replacement and simplification operations and thus model some of the more advanced fine-tuning of mathematical notions when processing mathematical language.

As a running example, we will recontextualize the definition of a path in graph theory to the context of automata theory. With sufficient training, a human reader can easily recognize how automata correspond to graphs, but students in theoretical computer science courses often struggle with this and may benefit from an explicit recontextualization or generated fine-grained explanations of the relationship.

Context & Generality Our work is developed in the context of ALeA [Ber+23], a symbolic learning support system. While ALeA provides the main motivation for the work reported here, the particular setup is secondary. The main assumptions are that the underlying knowledge representation admits theories and theory morphisms and that the presentation of the knowledge is flexiformal.

For instance, the Isabelle/Isar [Wen07] universe would be a likely candidate where the methods presented here would be applicable.

Overview In Section 2, we discuss flexiformal recontextualization and explore relevant phenomena using a running example. In Section 3, we discuss an implementation that shows the feasibility of our approach. Section 4 concludes the paper and discusses future work. An extended version of this paper with an appendix of flexiformal rewriting operations is available at [Kel+25].

Acknowledgments. The work reported in this article was conducted as part of the VoLL-KI project (see `https://voll-ki.de`) funded by the German Ministry of Research, Technology and Space under grant 16DHBKI089. We gratefully acknowledge the input of Dennis Müller in the development of suitable representations flexiformal theory morphisms in SᴛᴇX and FLAMS.

2 Flexiformal Recontextualization

Our goal with flexiformal recontextualization is to automatically translate learning materials about concepts from one scientific theory into the context of another theory to support students learning those concepts.

2.1 Example: Paths in Graphs and Automata

As a motivating example, we consider the notion of a path in a finite automaton, which is often used in the literature on automata theory, for instance:

- "state 6 is reached by the path $1 \to 2 \to 3 \to 6$" [HMU07, p. 91],
- "show that if there exists a path in $\mathcal{A}$ that leads from a state of $\mathcal{C}_1$ to a state of $\mathcal{C}_2$, ..." [ST09, p. 178],
- "the path starts or stops at state $k+1$ but doesn't go through any state numbered higher than k" [Mar03, p. 115].

These three examples have in common that the notion of a path in an automaton is not defined in the text preceding those quotes. Instead, the notion of a path in a *graph* – or, more precisely, in an *edge-labeled multidigraph*[3]. For a human reader who is sufficiently well trained in mathematics, it does not impose any difficulties to recognize (the transition system underlying) an automaton as a graph and apply the notion of a path in a graph to the setting of automata. However, for some students this conceptual leap is not obvious, and they would benefit from further explanation. Active documents allow us to show supplementary information on demand, e.g. by hovering over the term *"path"* in the above sentences. In this hover, we want to provide

[3] For the sake of brevity we use "graph" as a synonym for edge-labeled multidigraphs (or quivers) in the following.

1) the **recontextualized definition**, i.e. the definition of a *"path in an automaton"*,
2) the **original definition** of a *"path in a graph"*,
3) the **recontextualization morphism**, i.e. an explanation how an automaton induces a graph.

Figure 1 shows what such a pop-up note could look like for non-deterministic finite automata (NFAs).

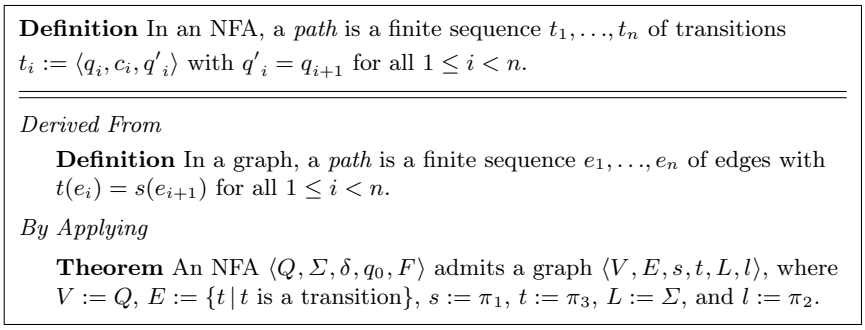


Fig. 1. Example hover for the word *"path"* in the context of NFAs, containing the recontextualized definition along with an (optional) explanation how it was derived. Here, $s(e)$ and $t(e)$ denote the source and target, resp., of an edge e, and π_i denotes projection to the i-th component of a tuple (in an active document, the reader can find this out by hovering over *"s"*, *"t"* and *"π"*).

Manually authoring the recontextualizations would be tedious and error-prone, as a simple back-of-the-envelope calculation illustrates: if we have 10 graph concepts (e.g. *path*, *cycle*, *loop*, *connectivity*, ...) that have to be recontextualized to 15 different structures (e.g. NFAs, DFAs, RDF triple collections, implication graphs, ...), we would need i) 10 original definitions, ii) 15 recontextualization morphisms, and iii) 150 recontextualized definitions. And, of course, graphs are just one of many theories that can be recontextualized. Other examples from our introductory course for symbolic AI alone include, e.g., logical systems, constraint satisfaction problems and optimization problems.

2.2 Recontextualization as a Pushout

In [KS24] we discussed that recontextualization corresponds to a **pushout** in a theory graph, which, in principle, allows us to generate recontextualizations automatically: Suppose we have i) a theory of graphs extended by a definition of the concept of a path, ii) a theory of automata and iii) a theory morphism between them. Then we can obtain the definition of a path in an automaton by computing the pushout of the span given by i), ii) and iii), as illustrated in Figure 2. Computing such pushouts is well-understood in a fully formal setting, where we essentially apply the substitution specified by the morphism. However,

our flexiformal setting is more challenging because sometimes the substitutions have to be applied to (semantically annotated) natural language.

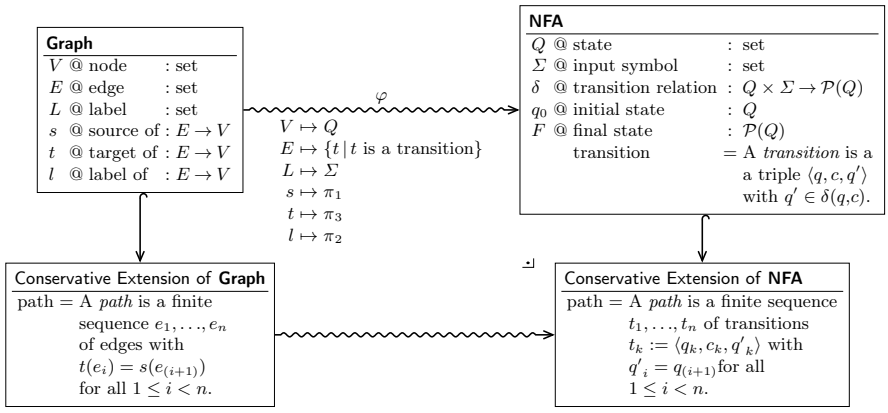


Fig. 2. Pushout for paths in graphs and finite automata. An entry of the form [⟨notation⟩ "@"] ⟨name⟩ (":" ⟨type⟩ | "=" ⟨definition⟩) is a declaration of a symbol with name ⟨name⟩, optional symbolic notation ⟨notation⟩ and either type ⟨type⟩ or definitional expression ⟨definition⟩. If the symbol denotes a set, we follow the convention to name the symbol after the elements of that set.

In [KS24], we described a template-based approach for flexiformal recontextualization that would simply substitute all components of the theory of graphs in the definiens of the *"path"* definition

> A *path* is a finite sequence $e_1, \ldots, e_n$ of edges with $t(e_i) = s(e_{i+1})$ for all $1 \leq i < n$.

with their respective values under the morphism φ, which is relatively straightforward if the relevant components (like the word *"edges"*) are semantically annotated. We obtain the following definition of a path in an automaton by applying that mechanism:

> A *path* is a finite sequence $e_1, \ldots, e_n$ of elements of $\{t \mid t \text{ is a transition}\}$ with $\pi_3(e_i) = \pi_1(e_{i+1})$ for all $1 \leq i < n$.

Semantically, we conflate nouns – like *"edge"* – with the set of all elements the noun stands for – in this case *"E"*. Linguistically, this requires us to insert *"element(s) of"* when substituting a noun with a set. Getting the correct form of the noun (*"element"* vs. *"elements"*) in a template-based approach is difficult, and support in our previous work [KS24] was limited to very specific cases.

2.3 Polishing the Result

While the result of the translation is technically correct, it is not as concise and clear as a human author would write it. Although statistical methods, like delegating the task to an LLM, might succeed on a wider range of sentences than the approach we are presenting, those methods cannot guarantee that meaning will be preserved – which would be problematic e.g. in an educational setting. Instead, we model the simplification process as a sequence of **meaning-preserving rewriting operations**.

To gain a better intuition, let us simplify the (recontextualized) definition from the previous section step by step:

1. **Comprehension term reduction:** First we reduce the comprehension term expression *"elements of* $\{t \mid t$ *is a transition}*" to *"transitions"*:

   > A *path* is a finite sequence $e_1, \ldots, e_n$ of transitions with $\pi_3(e_i) = \pi_1(e_{i+1})$ for all $1 \leq i < n$.

2. **Structure expansion:** Then we expand the noun phrase *"finite sequence* $e_1, \ldots, e_n$ *of transitions"* by appending a variable definition *"*$e_k := \langle q_k, c_k, {q'}_k \rangle$*"*:

   > A *path* is a finite sequence $e_1, \ldots, e_n$ of transitions $e_k := \langle q_k, c_k, {q'}_k \rangle$ with $\pi_3(e_i) = \pi_1(e_{i+1})$ for all $1 \leq i < n$.

3. **Variable expansion:** This allows us to expand later occurrences of *"*e*"* (*"*e_i*"* and *"*e_{i+1}*"*):

   > A *path* is a finite sequence $e_1, \ldots, e_n$ of transitions $e_k := \langle q_k, c_k, {q'}_k \rangle$ with $\pi_3(\langle q_i, c_i, {q'}_i \rangle) = \pi_1(\langle q_{i+1}, c_{i+1}, {q'}_{i+1} \rangle)$ for all $1 \leq i < n$.

4. **Projection reduction:** Now that the arguments of the projections are triples we can evaluate the projections:

   > A *path* is a finite sequence $e_1, \ldots, e_n$ of transitions $e_k := \langle q_k, c_k, {q'}_k \rangle$ with ${q'}_i = q_{i+1}$ for all $1 \leq i < n$.

5. **Variable renaming (optional):** As a last step we rename the variables e_j to t_j:

   > A *path* is a finite sequence $t_1, \ldots, t_n$ of transitions $t_k := \langle q_k, c_k, {q'}_k \rangle$ with ${q'}_i = q_{i+1}$ for all $1 \leq i < n$.

Note that steps 2–3 are motivated by the desire to eliminate the projections in step 4 – otherwise, they would make the result less concise.

During our research, we have collected a growing set of basic rewriting rules in [Rew] and in the long version of this paper [Kel+25, Appendix A] that modify natural language expressions in a way that preserves their meaning with the ultimate goal of making them more concise and natural. Collecting further rules and implementing them is ongoing work.

For instance, to perform the first rewriting step from our example above, we can apply rule *Element-NP CTR* from [Kel+25, Appendix A.6], namely

$$\frac{\text{“}\textit{element } [x] \textit{ of } \{y \mid y \textit{ is a } \langle\textit{noun phrase}\rangle\}\text{”}}{\text{“}\langle\textit{noun phrase}\rangle\ [x]\text{”}}\ ,$$

to the expression *“elements of $\{t \mid t$ is a transition$\}$”* which yields the expression *“transitions”*. Note that we have to respect the grammatical number of the noun *“element”* in the input expression which determines the grammatical number of the noun phrase (*“transition”*) in the output expression. In our implementation, these rules act on GF abstract syntax trees (see subsection 3.2 and subsection 3.3).

Other rules are more complicated and involve information that can not be obtained from the syntactical structure of the expression alone. For instance the *structure expansion* step in our example above requires to query the definiens of *“transition”* from a knowledge base. In even more extreme cases, a rewriting rule can demand a logical inference to be carried out before it can be applied. For instance, rule $\iota_\in$ *Conversion* from [Kel+25, Appendix A.7], namely

$$\frac{\vdash |A| = 1 \qquad \vdash a \in A \qquad \text{“}b \in A\text{”}}{\text{“}b = a\text{”}}\ ,$$

allows to convert an expression of the form *“$b \in A$”* to *“$b = a$”* under the constraint that it can be inferred that A consists of exactly one element a. While implementing this rule in its full generality may not be feasible, we can surely cover special cases that frequently occur in practice.

2.4 A Flexiformal Recontextualization Pipeline

In summary, we can think of flexiformal recontextualization as a two-step process: substitution application and simplification. The substitution is based on the recontextualization morphism, but we can use the same process to apply other substitutions. For example, in this paper, we will sometimes apply substitutions derived from definitions to perform definition expansion. Simplification is based on rewriting rules that maintain the meaning of the expression, but the application of those rules has to be carefully controlled.

While substitution and simplification are well-understood in a fully formal setting, dealing with the natural language in a flexiformal setting is more challenging. Semantic annotations can link parts of the natural language to the symbols they refer to, which helps identify where to apply the substitutions.

3 Implementation

We have implemented – on a prototype level – flexiformal substitution and simplification in the context of ALeA, using a rule-based approach to ensure correctness and trustworthiness of the generated content. The relevant documents are parsed with a natural language grammar into GF abstract syntax trees (ASTs). The ASTs are then processed and transformed with a set of hand-written rules, and the results are linearized into strings.

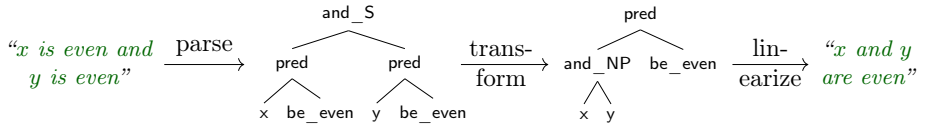
3.1 Context: The ALeA Ecosystem

Our work is implemented in the sTEX/rusTEX/FTML/FLAMS/ALeA ecosystem, where technical knowledge is represented in sTEX [MK22], a variant of LaTeX that supports the annotation of the semantic structures underlying the text content. The rusTEX system transforms sTEX sources into FTML, an extension of HTML5 that preserves all the semantic annotations from sTEX [Mül23]. FTML in turn can be processed by the FLAMS system [Mü] that serves the HTML5 content and integrates knowledge management services based on the annotations into it – turning the content into active documents. The ALeA system [Ber+23] in turn pairs up the active document facilities with learner competency modeling in an adaptive learning assistant that has been used for helping more than 500 students/semester master Math, AI, and CS courses at FAU over the last two years.

Our prototype is based on FTML because processing sTEX documents, which can contain arbitrary LaTeX, is very difficult. To be precise, we use an FTML-inspired HTML extension because FTML was still in flux during development. We will adapt our implementation to real FTML once the FTML format is stable.

3.2 A Grammar for Flexiformal Mathematics

For parsing and linearization, we use GF, the Grammatical Framework [Ran04], which was designed for (multi-lingual) natural language processing. GF separates between the **abstract syntax** of the language and the **concrete syntax** that describes how abstract syntax trees (ASTs) are represented as strings. The concrete syntax has powerful mechanisms like record types and tables to deal with the peculiarities of natural language such as inflection and agreement. The abstract grammar usually abstracts away from these details, which lets us manipulate the ASTs without worrying about such "low-level" aspects of natural language. To illustrate this, consider the following minimal example where we parse a sentence, transform the AST by an aggregation rule, and then linearize the result into a new sentence string:



Here, the information whether we should use *"is even"* or *"are even"* in the final sentence is not explicitly present in the AST. Instead, the concrete syntax rule for pred must specify how the correct verb form is chosen depending on the number of the subject.

The GF community has developed the **Resource Grammar Library** (**RGL**) [GFR], which implements the basic syntax and morphology of many languages. While the RGL is intended as a library for developing application-specific grammars, we have also experimented with using it directly for parsing (see [Kel24] for more details). With some modifications to the RGL, e.g. to support mathematical language and the semantic markup of sTEX, we were able to parse roughly

14 % of the English definitions in SMGloM (a flexiformal glossary that acts as a domain model for semantic annotations in STEX/ALEA, see [SMG]). Coverage was limited, among other reasons, because of missing words and abbreviations in the lexicon, problems with pre-processing the STEX files, and missing support for some sentence structures. While many of these issues could be improved, there are more fundamental problems with directly using the RGL: The grammar has a high degree of syntactic ambiguity, which results in many possible ASTs (often in the thousands, but sometimes many more because ambiguities multiply out). This makes the proper treatment of ambiguities (see subsection 3.4) impractical. More importantly, implementing substitution and simplification for RGL ASTs is cumbersome and error-prone.

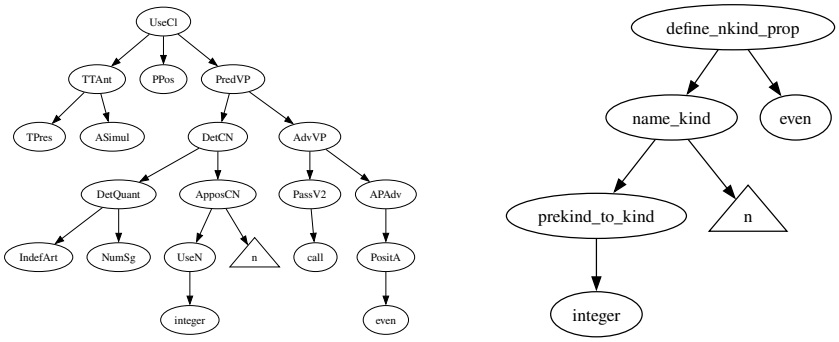


Fig. 3. ASTs for the phrase *"an integer n is called even"*. *Left:* Syntactic AST from an extended RGL grammar. *Right:* A more semantic AST from our application grammar. The formula *"n"* is actually encoded as a MathML tree, which was omitted for conciseness.

The better approach is to develop a new grammar that uses the RGL as a mere library. More concretely, we can implement a completely new abstract syntax that is more suitable for our application, and use the RGL only for the concrete syntax. In particular, we can design the abstract syntax to be more semantic, which makes processing the ASTs much easier. For example, consider the phrase *"an integer n is called even"* (typically followed by *"iff ..."*). In this case, syntactic details (e.g. that *"is called even"* is a verb phrase in passive voice) are not relevant for further processing. The relevant information is that the property of being *"even"* is defined for integers, and that we use *"n"* as a variable. In our grammar, we refer to words like *"integer"* as Kinds. A Kind with a variable is a NamedKind. So for the phrase above, we have a rule define_nkind_prop that combines a NamedKind and a Property into a definition core. Figure 3 visualizes the ASTs for the example phrase.

There are other ways to express the same meaning, e.g. *"an integer n is said to be even"* or *"an integer n is even"*. Syntactically, they are rather different (e.g. the latter one does not use passive voice), but for our processing we basically

want to treat them the same. GF supports variants, which allows us to use the define_nkind_prop rule for all three cases. In that case, the AST would not carry information about the original variant, and the linearization could use a different variant. While this should not result in ungrammatical or semantically incorrect sentences, it may result in less natural output. To avoid this, we use separate rules for each variant and use a consistent naming scheme (e.g. define_nkind_v1 and define_nkind_v2), so that we can ignore but preserve variant information throughout the recontextualization pipeline.

In our pipeline, we cannot ignore the HTML structure of the input document. For example, some semantic annotations, like definienda, are encoded via tags with specific attributes. Furthermore, formulae are encoded in MathML. To handle this, we include HTML tags in the abstract syntax trees. As a consequence, HTML tags cannot occur in arbitrary places, but rather must match the structure of the abstract syntax. For example, *"an even integer"* cannot be parsed, but *"an even integer"* can be parsed. Our experience is that the HTML structure usually matches the grammatical structure in practice – and if it does not, authors could consider that as an annotation error.

3.3 Substitution and Simplification

In a formal setting, substitution replaces a symbol with a term. In our flexiformal setting, this basically corresponds to replacing an AST subtree that corresponds to a specific symbol with a different subtree. For example, in the example in Section 2, we replaced *"edges"* with *"elements of* $\{t \mid t \text{ is a transition}\}$*"*.

In practice, the flexiformal substitution is a bit more complex. Consider, e.g. the sentence

> Let m be a positive integer.

and a substitution given by the definition

> An integer n is called *positive* iff $n > 0$.

The output is

> Let m be a integer such that $m > 0$.

In this case, the definiendum of *"positive"*, *"*$\cdot > 0$*"*, has a hole that must be filled with *"*m*"*. The AST of *"*$m > 0$*"* is then attached via a *"such that"* clause and the *"positive"* subtree is removed from the original AST. Figure 4 visualizes both the input and output ASTs.

Note that substitution in ASTs inherits a form of subject reduction from the underlying logical framework of GF: a (grammatically) well-typed substitution into (grammatically) well-typed ASTs gives (grammatically) well-typed ASTs – in fact they need to be well-typed if we want to linearize them with GF. If the grammatical types of a substitution do not match, like *"positive"* and *"*$\cdot > 0$*"* in

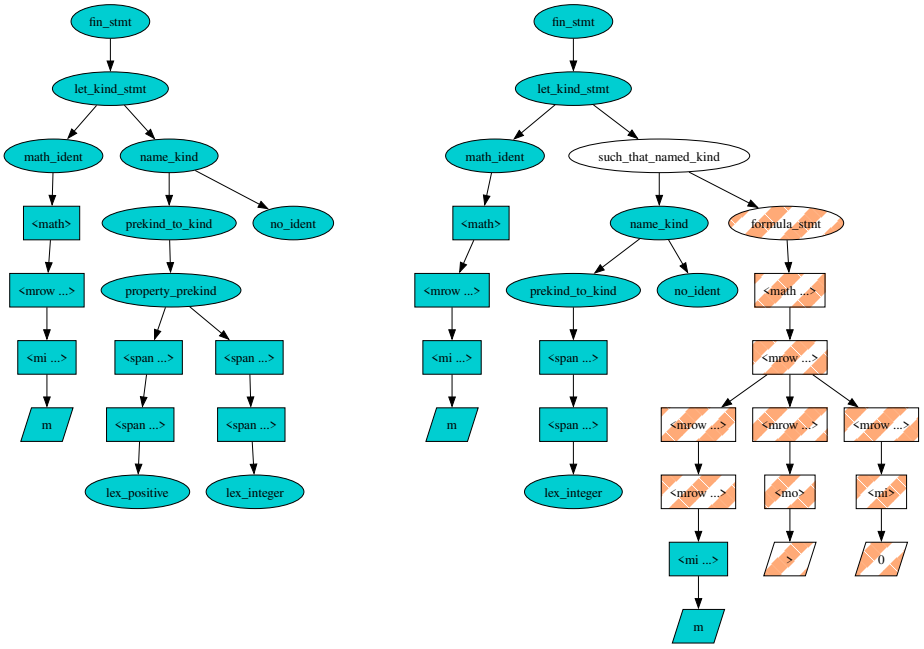


Fig. 4. Example of a substitution. *Left:* Input AST (*"Let m be a positive integer"*). *Right:* Output AST (*"Let m be a integer such that $m > 0$"*). The nodes from the input AST are colored turquoise, while the nodes from the definiendum are striped. The node shapes indicate the type of the node (grammar node/HTML tag/MathML literal).

the example above, we have to insert "glue language" (e.g. *"... such that ..."*) to ensure well-typedness.

After substitution, the ASTs are simplified with custom rules (see subsection 2.3), Currently, we only have implemented a small subset of our collection of rewriting rules. In our implementation, rewriting rules are triggered by the "desire" for a specific simplification. For example, the structure expansion in the running example (which by itself is not a simplification) gets triggered by the "desire" to apply projection reduction.

It appears that getting significant simplification coverage is a non-trivial task. At the moment, implementing new rewriting rules is complicated by the fact that semantic information is encoded in the ASTs in the attributes of HTML nodes, which makes the processing more cumbersome – especially for MathML nodes where the syntactic/presentational structure generally does not match the semantic structure well. We are planning to improve this by modelling the semantic structure more directly in the ASTs.

3.4 Handling Ambiguity

Sometimes, multiple ASTs correspond to the same string. In this case, GF can generate all possible ASTs. This gives us a basic tool for handling ambiguity:

if our grammar is sufficiently broad to cover the correct reading, we know that one of the ASTs will be the correct one. Our grammar/ASTs are designed to be somewhat semantic (see subsection 3.2). As a consequence, this mechanism does not just handle classical structural ambiguity, but also more semantic ambiguity. A simple example for this is lexical ambiguity (which is less relevant for us though, as discussed below). Lexical ambiguity can be handled by introducing multiple grammar rules for an ambiguous word, like the word *"even"*, which can refer to an integer being divisible by two or a function f having the property $f(x) = f(-x)$. By making two rules that linearize to *"even"*, e.g. even_function and even_integer, we would get two different ASTs for the same string – one for each meaning.

Disambiguation With Semantic Annotations The semantic annotations in STEX/FTML documents can help with disambiguation. For example, in a sufficiently annotated STEX/FTML document, technical terms are explicitly disambiguated with the concept they refer to. Assuming a sufficiently annotated document, we therefore do not have to worry about lexical ambiguity – at least for technical terms.

The STEX/FTML annotations can also help with other types of ambiguity. For example, the sentence

> An integer n is even iff n is divisible by 2.

could be interpreted as a theorem or a definition (epistemic ambiguity). In a definition, *"even"* would be annotated as a definiendum, but in a theorem it would be annotated as a symbol reference instead.

Ensuring Correct Results Despite Ambiguity As discussed above, during parsing we can get multiple ASTs for the same string because of ambiguity. Conversely, multiple ASTs can be linearized to the same string. This allows us to have correct results despite unresolved ambiguity. Consider, for example, an input sentence that gets parsed into two different ASTs, A and B. We apply some transformations to get A' and B'. If the linearizations of A' and B' are equal, it is ultimately irrelevant whether A or B were the correct reading. After all, the final document will only contain the linearization.

Of course, the result may be ambiguous – both A' and B' are possible readings –, but this can be considered a feature: natural language, including in mathematics, is ambiguous, and restricting the output to non-ambiguous sentences would be a severe limitation. With flexiformal substitution, we can get additional ambiguity from the substitution value itself, which increases the overall number of readings further: If we have a single substitution with readings $S_1, \ldots, S_n$ and input readings $I_1, \ldots, I_m$, we get $m \cdot n$ resulting ASTs (applying S_1 to I_1, S_1 to I_2, etc.). The final result has to be compatible with all of them. One option to increase the chances of getting a compatible result (which we have not yet implemented) is to optionally ignore some rewriting rules during simplification. For example, if one particular rewriting rule is not applicable in one of the ASTs,

we may still get a linearization that is compatible with all other ASTs if we skip this rewriting rule in all other ASTs as well.

If no compatible linearization can be found, it would be up to the user to select the correct string from a list of options.

In the case of the running example (see subsection 2.3), our grammar gets two different readings (after filtering; see below) – one where the *"with"*-phrase is attached to *"sequence"*, and one where it is attached to *"edge"*. Both readings result in the same linearization after substitution and simplification.

Reducing the Number of Readings by Filtering Even though our pipeline can handle ambiguities to some extent, reducing the number of readings is still desirable. This can significantly improve performance by dampening the combinatorial explosion and, as a side effect, simplify the substitution and rewriting rules as fewer cases have to be considered. There are two ways to reduce the number of readings: grammar adjustments and filtering after parsing, which come with different trade-offs.

Let us illustrate this with the example string *"finite subset of* $\mathbb{N}$*"*, which in our grammar is called a Kind. There are two ways to read it (see Figure 5): either *"finite"* modifies *"subset of* $\mathbb{N}$*"*, or *"of* $\mathbb{N}$*"* is an argument of *"finite subset"*. We can adjust the grammar to enforce the latter reading by introducing a separate syntactic category, PreKind, that can only be modified by adjectives, and then turned into a Kind that can only be modified with prepositional arguments.

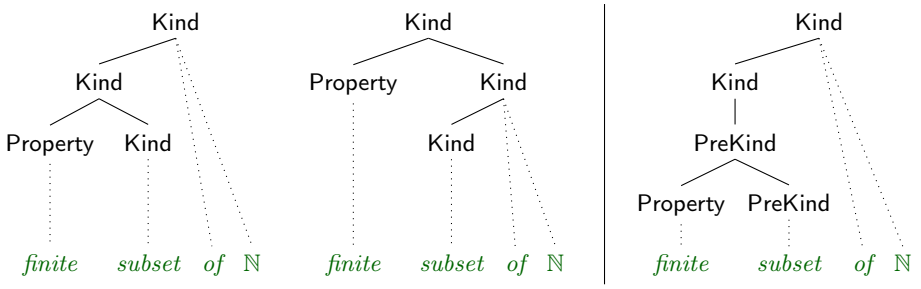


Fig. 5. Readings for an example phrase. For conciseness, syntactic categories are displayed instead of rule names. With a simple grammar, we get two different readings (*left*), but by introducing a new syntactic category, PreKind, we can enforce a single reading (*right*) at the cost of added complexity.

As a consequence of this, it can now be ambiguous where exactly an HTML tag is in the AST. For example, in *"finite subset of* $\mathbb{N}$*"*, the tag can now be wrapped around *"finite subset"* either before or after the PreKind is turned into a Kind. To remedy this, we can simply filter out all ASTs where an HTML tag is wrapped around the AST node that turns a PreKind into a Kind.

In general, grammar adjustments lead to a more complex grammar that may be harder to understand and maintain. Filtering, on the other hand, keeps the grammar simple, but may lead to performance problems if too many readings are first generated and then filtered out. Furthermore, filtering must be done carefully to ensure that only redundant readings are removed. Finding the right balance between the two approaches remains work in progress.

3.5 State of the Implementation

The implementation described here is our fourth prototype for exploring the feasibility of rule-based flexiformal recontextualization of learning objects. Its design was informed by the previous prototype implementations. The first prototype was presented in [KS24], and used simple templates for recontextualization of sTeX content. The next prototype was developed in the context of a Bachelor's thesis of one of the authors (see [Kel24]), and explored flexiformal definition expansion with a grammar based on GF's Resource Grammar Library (see subsection 3.2). To overcome some of the challenges from processing LaTeX/sTeX sources directly, a third prototype was developed based on the FTML format. Ultimately, it became clear that a more semantic grammar as described in subsection 3.2 was necessary to develop robust rules for substitution and simplification, which led to the implementation described in this paper.

We have implemented all rules necessary for the running example from Section 2. We can handle the whole pipeline of AST-based relocalization by substitution and polishing via rewriting with the exception of variable renaming. In constrast to the deterministic/semantic rewriting rules, variable renaming has a strong cognitive flavor, which calls for different methods and quality measures; so we leave it to future work.

At this point, every new example requires work on the grammar and the substitution/rewriting rules. While the grammar and, to an extent, the substitution rules give the impression of converging, getting a reasonable coverage for the rewriting rules appears to be a much longer-term enterprise.

This shows that our overall architecture works in principle; but also highlights that we need extensions in the grammar and rewriting rules to scale to realistic applications in ALeA.

We are currently refactoring the prototype into a robust foundation for further development, which will be hosted at `https://github.com/FlexiFormal/relocalization`. The repository also contains links to the prototype implementations.

4 Conclusion & Future Work

We have shown how we can extend content generation – recontextualization; technically (theory) morphism application – from the formal to the flexiformal setting by parsing the mathematical vernacular in the source into ASTs, applying AST substitutions, and then fine-tuning the conciseness of the generated

language with rewriting rules. We have shown the practical feasibility of the method with a GF grammar and a set of domain-specific AST rewriting rules.

While the implementation is still at a prototype stage – it can handle our running example and a few more – we feel confident that we can scale coverage to useful levels with more development. Most importantly, we feel that the overall information architecture has stabilized, and that this is a sufficient milestone that needs to be communicated to the CICM community, so that others may become involved.

While the GF grammar has proven easy to extend for new sentences, our "rational reconstruction" of recontextualization at the AST level has revealed a surprising extent and variety of simplifications needed at this level to account for the concise and rigorous mathematical language we see in mathematical texts written by humans. During our explorative work, we have collected a set of – mostly foundational – rewriting rules that cover cases like our running example (see [Kel+25, Appendix A]). We believe that this set is by no means complete outside the set-theoretic foundation, and we will have to extend it considerably to scale the AST-to-AST transformation. It is still an open task to systematically evaluate the scalability and coverage of those rules. To increase coverage we plan to continue collecting rules, involving all AI methods (both symbolic as well as statistical) at our disposal. However, in practice the process of finding new transformation rules is "co-recursive" with domain formalization, theory graph development, and grammar design, which makes the overall process hard to automate. We conjecture that the AST rewriting can be very useful for polishing and adapting all kinds of flexiformal transformation tasks.

Note that already the current rule-set is non-confluent. Thus their application is a non-trivial search/optimization problem where the quality measure is conciseness and readability of the generated language. How to manage/implement this scalably and independently of the concrete rule set – and that has to be our goal – is an open research question.

References

[Ber+23] M. Berges, J. Betzendahl, A. Chugh, M. Kohlhase, D. Lohr, and D. Müller. "Learning Support Systems based on Mathematical Knowledge Managment". In: *Intelligent Computer Mathematics (CICM) 2023*. Ed. by C. Dubois and M. Kerber. Vol. 14101. LNAI. Springer, 2023. DOI: 978-3-031-42753-4. URL: https://url.mathhub.info/CICM23ALEA.

[GF] *GF - Grammatical Framework*. URL: http://www.grammaticalframework.org (visited on 09/27/2017).

[GFR] B. Bringert, T. Hallgren, and A. Ranta. *GF Resource Grammar Library: Synopsis*. URL: https://www.grammaticalframework.org/lib/doc/synopsis/ (visited on 03/11/2020).

[HMU07] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 3rd ed. Pearson/Addison Wesley, 2007.

[Kel24] J. Kelber. "Flexiformal Definition Expansion and Parsing Mathematical Natural Language". Bachelor's thesis. FAU Erlangen-Nürnberg, 2024.

[Kel+25] J. Kelber, M. Kohlhase, J. F. Schaefer, and M. Schütz. "Reaping the Benefits of Modularization in Flexiformal Mathematics by GF-based AST Transformations (Extended Version)". 2025. URL: https://kwarc.info/people/jfschaefer/pubs/2025-recontextualization-cicm-extended.pdf.

[Koh13] M. Kohlhase. "The Flexiformalist Manifesto". In: *14th International Workshop on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2012)*. Ed. by A. Voronkov, V. Negru, T. Ida, T. Jebelean, D. Petcu, S. M. Watt, and D. Zaharie. Timisoara, Romania: IEEE Press, 2013, pp. 30–36. URL: https://kwarc.info/kohlhase/papers/synasc13.pdf.

[KS24] M. Kohlhase and M. Schütz. "Reusing Learning Objects via Theory Morphisms". In: *Intelligent Computer Mathematics (CICM) 2024*. Ed. by A. Kohlhase and L. Kovacz. Vol. 14960. LNAI. Springer, 2024, pp. 165–182. DOI: 10.1007/978-3-031-66997-2_10. URL: https://url.mathhub.info/relocalization/.

[Mar03] J. Martin. *Introduction to Languages and the Theory of Computation*. McGraw-Hill higher education. McGraw-Hill, 2003.

[MK22] D. Müller and M. Kohlhase. "sTeX3 – A LaTeX-based Ecosystem for Semantic/Active Mathematical Documents". In: *TUGboat; TUG 2022 Conference Proceedings* 43.2 (2022). Ed. by K. Berry, pp. 197–201. URL: https://kwarc.info/people/dmueller/pubs/tug22.pdf.

[MRK18] D. Müller, F. Rabe, and M. Kohlhase. "Theories as Types". In: *9th International Joint Conference on Automated Reasoning*. Ed. by D. Galmiche, S. Schulz, and R. Sebastiani. Springer Verlag, 2018. URL: https://kwarc.info/kohlhase/papers/ijcar18-records.pdf.

[Mül23] D. Müller. "An HTML/CSS schema for TeX primitives – generating high-quality responsive HTML from generic TeX". In: *TUGboat; TUG 2023 Conference Proceedings* 44.2 (2023), pp. 275–286. URL: https://kwarc.info/people/dmueller/pubs/tug23.pdf.

[Mü] D. Müller. FLAMS – *Flexiformal Annotation Management System*. URL: https://github.com/flexiformal/flams (visited on 07/24/2025).

[Pfe01] F. Pfenning. "Logical Frameworks". In: *Handbook of Automated Reasoning*. Ed. by A. Robinson and A. Voronkov. Vol. I and II. Elsevier Science and MIT Press, 2001.

[Ran04] A. Ranta. "Grammatical Framework — A Type-Theoretical Grammar Formalism". In: *Journal of Functional Programming* 14.2 (2004), pp. 145–189.

[Rew] *Collection of flexiformal rewriting rules*. URL: https://github.com/flexiformal/rewriting-rules (visited on 07/25/2025).

[SMG] *SMGloM: A Semantic Multilingual Glossary Resource for Mathematics*. URL: http://gl.mathhub.info/smglom (visited on 07/24/2025).

[ST09] J. Sakarovitch and R. Thomas. *Elements of Automata Theory*. Cambridge University Press, 2009.

[Wen07] M. Wenzel. "Isabelle/Isar – a generic framework for human-readable proof documents". In: *From Insight to Proof – Festschrift in Honour of Andrzej Trybulec* (2007). Ed. by R. Matuszewski and A. Zalewska.