Prototyping Controlled Mathematical Languages in Jupyter Notebooks

Jan Frederik Schaefer, Kai Amann, and Michael Kohlhase

Computer Science, FAU Erlangen-Nürnberg

Abstract. The Grammatical Logical Framework (GLF) is a framework for prototyping the translation of natural language sentences into logic. The motivation behind GLF was to apply it to mathematical language, as the classical compositional approach to semantics construction seemed most suitable for a domain where high precision was mandatory — even at the price of limited coverage. In particular, software for formal mathematics (such as proof checkers) require formal input languages. These are typically difficult to understand and learn, raising the entry barrier for potential users. A solution is to design input languages that closely resemble natural language. Early results indicate that GLF can be a useful tool for quickly prototyping such languages. In this paper, we will explore how GLF can be used to prototype such languages and present a new Jupyter kernel that adds visual support for the development of GLF-based syntax/semantics interfaces.

1 Introduction

The work of mathematicians is increasingly supported by computer software ranging from computer algebra systems to proof checkers and automated theorem provers. Such software typically requires a specialized input language, which users have to learn to use the software themselves and in order to understand how it was used by other people. The latter is of particular interest when it comes to computer supported theorem proving. In mathematics, proofs are much more than mere correctness certificates: they give insights into *why a theorem is true*. A computer proof cannot fulfil this duty if the reader cannot understand it in the first place. The obvious consequence is that input languages should be designed to be as intuitive as possible. In some cases, this could be in the form of a **controlled natural language** — a formal language with well-defined semantics that closely resembles natural language. There are some general-purpose controlled natural languages, most notably Attempto Controlled English (ACE) [FSS98]. But for input languages for mathematical software, we need **controlled mathematical languages**.

State of the Art A **controlled natural language** (CNL) consists of *i*) a natural language fragment e.g. defined by a grammar, *ii*) a formal target language, and *iii*) a program that translates from *i*) to *ii*). Different controlled mathematical languages (CML) have been developed in the past, especially for automatic

proof checkers. An example for this is ForTheL [Pas07], the language of the System for Automated Deduction (SAD). It appears that ForTheL has reached a sweet spot between expressivity and parseability. Implemented with hand-crafted parser combinators in Haskell, however, it is hard to maintain and even harder to extend. More recently, SAD was extended by some of the people behind the Naproche system [Cra13], which has a controlled mathematical language that supports a "controlled" LaTeX input. This resulted in the Naproche-SAD project [FK19]. Over the last few years, research into controlled mathematical languages has gained momentum with Thomas Hales' Formal Abstracts project (e.g. [Hal19]). Its goal is the creation of a controlled mathematical language that translates into the language of the lean theorem prover – a type theory based on the calculus of inductive constructions.

Overview In this paper, we present a setup for prototyping controlled mathematical language. Section 2 describes the underlying technology: the Grammatical Logical Framework (GLF) [KS19]. In Section 3 we introduce a new Jupyter kernel for GLF that makes GLF much more accessible and supports the development and testing of controlled mathematical languages with a variety of features. All listings in this paper are screenshots of Jupyter notebooks. In Section 4, we will discuss some of our insights from our attempts to re-implement ForTheL with GLF. Section 5 concludes the paper.

2 Grammatical Logical Framework

The **Grammatical Logical Framework** (GLF) [KS19] is a tool for prototyping translation pipelines from natural language to logic. As a running example, we will develop a pipeline that translates sentences like *"the derivative of any holomorphic function is holomorphic"* into expressions in first-order logic: $\forall f(\text{holomorphic}(f) \Rightarrow \text{holomorphic}(\text{derivative}(f)))$. This translation pipeline consists of two steps: parsing and semantics construction.

```
abstract Grammar = {
  cat
    Stmt; Term; Notion; Prop;
  fun
    state : Term->Prop->Stmt;
    every : Notion -> Term;
    -- ...
    integer : Notion;
    even : Prop;
    derivative : Term -> Term;
}
```

```
concrete GrammarEng of Grammar = {
  lincat Stmt=Str;Term=Str; -- ...
  lin
    state t p = t ++ "is" ++ p;
    every n = ("every"|"any")++n;
    -- ...
    integer = "integer";
    even = "even";
    derivative t =
        "the derivative of" ++ t;
}
```

Listing 1.1: Sketch of a very simple GF grammar to talk about mathematics.

Parsing is done with the **Grammatical Framework** (GF) [Ran11], which is a powerful tool for the development of natural-language grammars. A GF grammar consists of an **abstract syntax** that describes the parse trees and (possibly multiple) **concrete syntaxes** that describe how these parse trees correspond to strings in a particular language. Listing 1.1 sketches an example GF grammar that can parse sentences like *"every integer is even"*[1]. The abstract syntax introduces categories (node types) and function constants that describe how nodes can be combined. E.g. `state` combines a term and a property into a statement. The sentence *"every integer is even"* thus corresponds to the expression `state (every integer) even`. For our simple example, the concrete syntax is very straight-forward, but in general the concrete syntax has to handle the complex morphology and syntax of natural language. GF supports this with a powerful type system and various mechanisms for modularity and reusability. GF also supplies the *Resource Grammar Library*, which provides re-usable implementations of the morphology and basic syntax for many (≥ 35) languages.

```
theory FOL : ur:?LF =
  propositions : type | # o |
  individuals : type | # ι |
  not : o → o | # ¬ 1 |
  and : o → o → o | # 1 ∧ 2 |
  // ... /
  forall : (ι→o) → o | # ∀ 1 |
  exists : (ι→o) → o | # ∃ 1 |
|

theory DomainTheory : ?FOL =
  integer : ι → o |
  even : ι → o |
  derivative : ι → ι |
  // ... /
|
```

```
view GrammarSemantics :
    ?Grammar -> ?DomainTheory =
  Stmt = o |
  Term = (ι → o) → o |
  Notion = ι → o |
  Prop = ι → o |

  state = [term,pr] term pr |
  every = [notion] [p]
       ∀ [x] notion x ⇒ p x |
  // ... /
  integer = integer |
  even = even |
  derivative = [term] [p]
     term ([x]p(derivative x))|
|
```

Listing 1.2: Example logic, domain theory, and semantics construction in MMT.

The **semantics construction** describes how the parse trees are translated into logical expressions. GLF uses the **Meta Meta Tool** (MMT) for the logic development and semantics construction. MMT is a foundation-independent framework for knowledge representation [MMT]. In MMT, knowledge is represented as **theories**, which contain sequences of constant declarations of the form

```
CONSTANT [: TYPE] [|= DEFINITION] [|# NOTATION] |
```

[1] Note that neither parsing nor the semantics construction are concerned with the validity of a statement.

While MMT itself is foundation independent, MMT theories are typically based on the Edinburgh Logical Framework (LF) [HHP93] and extensions of that in practice. Listing 1.2 contains a theory `FOL` that defines the syntax of first-order logic. First, we need types for propositions and individuals, denoted by $\circ$ and ι respectively. Afterwards, we can declare logical connectives as binary/ternary operators on propositions with the expected prefix/infix notations. For quantifiers, we use higher-order abstract syntax. Together with a domain theory, this allows us to express the meaning of our example sentence as $\forall x(\text{integer}(x) \Rightarrow \text{even}(x))$.

Before we can define the semantics construction, we need to be able to represent GF parse trees as MMT terms. For this, GLF creates a **language theory** from the abstract syntax, i.e. an MMT theory that contains the GF categories as type constants and the GF functions as function constants. Then, we can define the semantics construction as an MMT **view** from the language theory into the domain theory (see Listing 1.2). A view maps every constant in the source theory to a term in the target theory – e.g. statements are mapped to propositions ($\circ$) and properties to unary predicates ($\iota \to \circ$). A term like *"every integer"* should have the meaning $\lambda p.\forall x(\text{integer}(x) \Rightarrow p(x))$, i.e. we apply properties to terms, not the other way around. Note that $\lambda x.M$ is denoted in MMT by `[x] M`.

With all this in place, we can parse the sentence *"every integer is even"* to obtain the parse tree `state (every integer) even`, and then apply the semantics construction to obtain the MMT expression

`([term,prop] term prop) (([n] [p] ∀ [x] n x ⇒ p x) integer) even`,

which β-reduces to the desired `∀ [x] integer x ⇒ even x`.

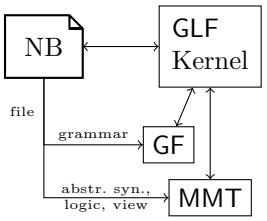
Given the declarative treatment of semantics construction and target logic, GLF can serve as the basis for a rapid prototyping system for the development and implementation of controlled (mathematical) languages. To complete that, we need a good user interface (the extended GF shell that GLF comes with does not qualify).

3 Jupyter Integration

Jupyter [Jup] provides a user environment for working with notebooks, which can contain code cells, explanatory text and interactive widgets. The code cells can be executed in-document, resulting in a very interactive experience. We developed a new Jupyter kernel to bring these features to GLF.

The code cells in a GLF notebook either enrich the language context or contain executable commands. The **language context** consists of GF grammars and MMT theories and views. A user can explore and test the language context with **commands** for e.g. parsing a sentence with the specified grammar and applying the semantics construction.

When a code cell is executed, the first step is to identify its content type. We use simple pattern matching for this. If the code cell extends the language context, we write its content to a file. The file name is simply the name of the grammar/theory/view, which is also extracted during the pattern matching. Afterwards, grammars are imported into GF and MMT (for the language theory) and theories and views are imported into MMT.



For this, GF and MMT are running as subprocesses in the background and the GLF kernel communicates with them via pipes and HTTP respectively. The user gets feedback whether the imports succeeded along with possible error messages.

If a code cell contains commands, on the other hand, they are executed and the output is returned to the user. As GF is an integral part of the system, the GLF kernel supports all of the GF shell commands by simply passing them on to the GF shell. On top of that, we have added a number of kernel commands peculiar to GLF. Some of them are stand-alone commands such as for specifying where the GF and MMT files should be stored. Other commands are intended to be used in combination with GF commands. In the GF shell, commands can be combined with the pipe operator `|`. For example, one might want to use the `parse` command to obtain the parse tree of an English sentence and then use the `linearize` command to transform the parse tree into e.g. an Italian sentence. The Jupyter kernel imitates this behaviour, i.e. if the user enters a command of the form `a | b`, the output of command `a` is used as input for command `b`. By imitating this behaviour, rather than letting the GF shell handle the piping, we can add kernel commands that can also be used in combination with pipes, which lets them blend in more naturally. The **`construct`** command takes a parse tree as argument and sends a semantics construction request to MMT. Therefore, it is commonly used in combination with the `parse` command. The **`show`** command can be used for in-notebook visualization of parse trees (see Figure 1). It is usually used in combination with certain GF commands that generate graph descriptions in the `.dot` format. The `show` command then uses GraphViz to generate images that are displayed in a widget. If there are multiple parse trees (e.g. due to ambiguity), a drop-down menu is created, where the user can select which parse tree to display.

The GLF kernel provides a number of convenience features. Syntax highlighting is based on CodeMirror. In JupyterLab, which has been used for the screenshots in this paper, syntax highlighting is provided by an extension. The syntax highlighting also depends on the content type of the cell, i.e. different rules are used for GF content, MMT content and commands.

Other features are built around tab-completion. MMT theories often use unicode characters for notations. The Jupyter kernel has a list of (currently 426) character sequences that can be tab-completed into unicode characters This is based on a similar list used in other MMT services. The character sequences are inspired by LaTeX macros, which most users should be familiar with. For example, `\subseteq` is completed to ⊆.

Tab-completion is also used for stub generation. A common workflow when writing a GF grammar is to first write the abstract syntax and then implement (possibly multiple) concrete syntaxes. A concrete syntax simply defines a linearization for all symbols introduced in the abstract syntax. The GLF kernel comes with a small script that can parse GF's abstract syntaxes — as long as they only contain commonly used features — and then generate a stub for the

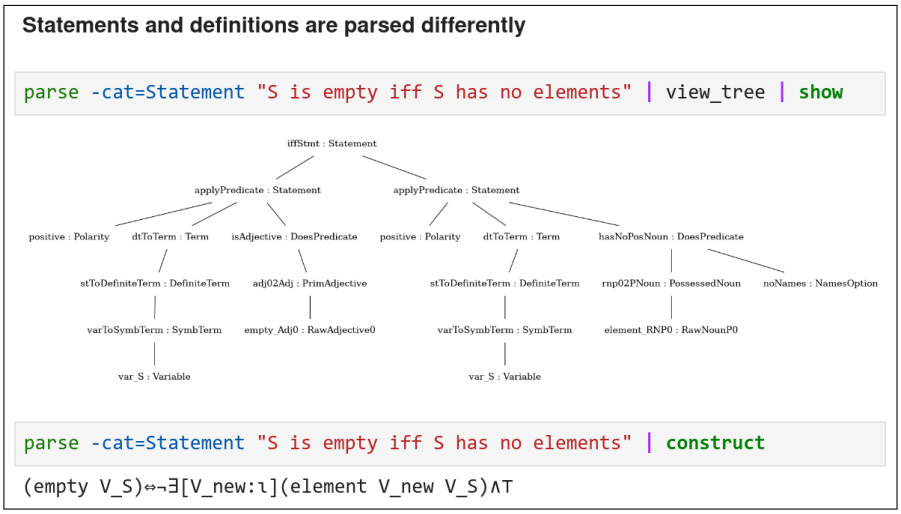


Fig. 1: Fragment of a GLForTheL notebook.

concrete syntaxes. This allows the user to fill out the concrete syntax without having to repeatedly scroll back to the abstract syntax to copy all the symbol names. Similarly, views for the semantics construction have to map every symbol in the abstract syntax to a logical expression, so stubs are generated for that as well (Figure 2). Experience has shown that it is beneficial to add the types of function constants as comments in the generated stubs.

Smaller GLF pipelines like the example in Section 2 can be conveniently implemented and tested in Jupyter notebooks. Of course, larger projects (like the GLForTheL project discussed in the next section) are usually not implemented inside notebooks, but rather in text editors and IDEs. GLF pipelines inherit the modularity of GF grammars and MMT pipelines, and the GLF kernel can use grammars/theories/views defined elsewhere. This way, notebooks can be used for testing and documenting the pipeline, as well as for interactively exploring specific problems during the course of the project.

```
view GrammarSemantics : http://mathhu
    Stmt = _ |
    Term = _ |
    Notion = _ |
    Prop = _ |

    // state : Term → Prop → Stmt |
    state = _ |
    // every : Notion → Stmt |
    every = _ |
    // integer : Notion |
    integer = _ |
    // even : Prop |
```

Fig. 2: Generated stub.

4 Case Study: GLForTheL

ForTheL [Pas07] is the controlled mathematical language of the System for Automated Deduction. We have recently started to experiment with re-implementing

ForTheL in GLF (we call the result GLForTheL). This can serve as a case study that highlights both the capabilities of GLF and the role of Jupyter notebooks during development. In the following paragraphs, we will discuss some of the challenges we encountered during the implementation of GLForTheL.

Binding Variables Our running example already covers quantification in natural language (*"every integer"*). However, in mathematical language this is further complicated by the use of variables, as exemplified in this ForTheL statement:

"there is an integer N that is even"

Here, N has to be bound to a quantifier. This is a problem in GLF, because N is treated as a constant during the semantics construction. Our current workaround in GLForTheL is a special λ binder that turns the bound constant into a variable. For example, the semantics construction might map the statement above to $\exists(\lambda'\ N.\ (\text{int}(N) \wedge \text{even}(N)))$, where λ' and N are simple (function) constants. A post-processing step transforms the "bound constant" N into a variable (including all its occurrences in the body) and replaces λ' by a real λ.

Redundancy in Logical Expressions The handling of variable sequences in GLForTheL results in some artefacts in the logical expressions. An example are trailing $\wedge\, true$ in some statements. While simple artefacts like this could be removed by MMT, there are also more complicated redundancies: GLForTheL translates the statement *"there are sets X, Y such that every element of X is an element of Y"* into the expression

$$\exists Y(\exists X(\text{set}(Y) \wedge \forall n(n \in X \Rightarrow n \in Y) \quad \wedge \quad \text{set}(X) \wedge \forall n(n \in X \Rightarrow n \in Y)))$$

because both X and Y are *"sets such that every..."*. We are currently working on an extension of GLF that adds an inference step to the pipeline (see [SK20]), which could — among other things — be used to implement advanced simplification algorithms.

Lexicon Management Adding another word to the grammar usually requires new entries to the abstract syntax, concrete syntax, domain theory, and semantics construction. To simplify this, we have developed a tool that automatically creates these entries from a custom lexicon file. Especially in mathematics, though, there is another problem: new words and notations are introduced whenever needed, so the lexicon is growing while a document is processed. This problem is currently unsolved in GLF, which relies on a pre-defined lexicon. In the context of controlled languages, it may be argued that a document should have a preamble defining the necessary lexicon. Another solution could be a two-pass process, where a document-specific lexicon could be generated in a pre-processing step that harvests the definienda.

Better Target Logic: DRT Discourse Representation Theory (DRT) [KR93] solves various problems that arise from using first-order logic as the target representation in compositional natural-language semantics by introducing discourse representation structures as an intermediate representation, which can be compiled

into first-order logic. Variants of DRT have been repeatedly used in the context of mathematical language (e.g. [Cra+10]).

Neither ForTheL nor GLForTheL use DRT. One of the consequences is that statements like

"if the square of some integer N is even then N is even"

get translated into

$$(\exists v(\text{int}(v) \wedge \text{even}(\text{square}(v)))) \Rightarrow \text{even}(N),$$

because *"some"* usually means that the expression is existentially quantified (think e.g. of *"H is contained in some ball $B \subseteq U$"*). Of course, one would expect to get

$$\forall v((\text{int}(v) \wedge \text{even}(\text{square}(v))) \Rightarrow \text{even}(v)).$$

To remedy this in the future, we are currently looking into different ways of representing DRT in MMT.

From Sentences to Discourse GLF operates on the sentence level. This becomes a problem when variables are introduced in one sentence (*"let G be a group"*) and then used in another sentence. Our implementation extracts the restrictions ($G : group$) and keeps G as a free variable in the following sentences. In general, our goal is to extend GLF with an inference step (as mentioned above), which could be used to combine this information. It would also allow experimentation with other discourse-level challenges such as anaphor resolution.

5 Conclusion and Evaluation

We have introduced a new Jupyter front-end for GLF; together they form a rapid prototyping system for controlled mathematical languages.

So far, it has been mostly used in a one-semester course on logic-based natural language processing [LBS20] at FAU Erlangen-Nürnberg. In previous years, we connected GF and MMT via some rather fragile Scala code, which was so inconvenient that we mostly used GF and MMT independently. Last semester we had the first iteration of the lecture using GLF and Jupyter. In the lab sessions (half of the course), we explored different natural-language semantics phenomena by implementing the language-to-logic pipeline in GLF. Since the class was rather small, we basically asked the students to tell us what to enter into the notebook and could immediately test the ideas. After some cleanup, the notebooks could be shared with students — much more easily than the messy file collections we had had in previous years. The modularity of GLF also allowed us to try different semantics constructions for the same grammar. We also used Jupyter notebooks for homework assignments, providing partial implementations where the students had to implement the critical parts. The students also had the option to implement the homework without Jupyter notebooks (using command line tools for testing), but almost all students chose to use Jupyter notebooks. One of the

biggest challenges was the installation on students' computers. This was further complicated by the need for updates throughout the semester as we improved the implementation. Going forward, we are planning to provide Docker images and online notebooks that can be used instead.

The newly reached maturity of GLF allowed us to return to our original motivation to apply it to mathematical language. As a larger case study, we have discussed our attempts to implement GLForTheL, a variant of ForTheL, which was our first larger project. Since the goal of GLForTheL is to imitate ForTheL, our experimentation was primarily focused on the ways technical challenges can be handled. While Jupyter notebooks were the go-to tool for these experiments, they were less useful for implementing the actual GLForTheL project, since it was much larger (currently 39 different node types and over 50 production rules). GLForTheL imposes tighter restrictions on the input language than ForTheL, rejecting ungrammatical statements like *"S are a sets"*, which are accepted by ForTheL. For a long time we supported both a German and an English concrete syntax. All this was possible without much effort, due to GF's powerful grammar mechanisms.

At its current state, our GLForTheL re-implementation can translate two example files from the SAD repository (excluding proofs). Since GLForTheL requires well-formed English sentences, it will never have the same coverage as ForTheL, but this was not our goal after all. One of the more complex sentences GLForTheL can currently handle is the definition *"a subset of S is a set T such that every element of T belongs to S"*, which results (after some α-renaming for readability) in

$$\forall T.(subsetof\ T\ S) \Leftrightarrow (set\ T) \wedge \forall x.(elementof\ x\ T) \wedge \top \Rightarrow (belongto\ x\ S) \wedge \top.$$

Expanding the coverage to more examples mostly boils down to extending the lexicon and adding the occasional grammatical rule. However, more work is needed to handle binary relations imposed on variable sequences as in *"let x, y, z be pairwise linearly independent vectors"*.

Our GLForTheL case study also indicated the need for a processing step after the semantics construction. [SK20] describes an extension of GLF that adds an inference component, which can be used for e.g. simplification, ambiguity resolution or theorem proving. Additional work is needed for lexicon management and regression testing.

Overall, we believe our experiences with GLForTheL confirm our hypothesis that GLF+Jupyter provide a flexible framework for the quick prototyping of controlled mathematical languages. The Jupyter kernel along with a link to an online version can be found at [GLFa] and the GLForTheL code at [GLFb].

References

[Cra+10] Marcos Cramer et al. "The Naproche Project Controlled Natural Language Proof Checking of Mathematical Texts". In: *Controlled Natural Language, Workshop on Controlled Natural Language, CNL 2009. Revised Papers*. Ed.

by Norbert E. Fuchs. LNCS 5972. Springer, 2010, pp. 170–186. DOI: 10.1007/978-3-642-14418-9_11.

[Cra13] Marcos Cramer. "Proof-checking mathematical texts in controlled natural language". eng. PhD thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, 2013. URL: http://hss.ulb.uni-bonn.de/2013/3390/3390.pdf.

[FK19] Steffen Frerix and Peter Koepke. *Making Set Theory Great Again*. Talk at the Artificial Intelligence and Theorem Proving Conference. 2019. URL: http://aitp-conference.org/2019/slides/PK.pdf.

[FSS98] Norbert E. Fuchs, Uta Schwertel, and Rolf Schwitter. "Attempto Controlled English - Not Just Another Logic Specification Language". In: *Proceedings of the 8th International Workshop on Logic Programming Synthesis and Transformation*. LOPSTR '98. Springer-Verlag, 1998, 1–20.

[GLFa] *GLF Kernel*. URL: https://github.com/kaiamann/glf_kernel (visited on 03/27/2020).

[GLFb] *GLForTheL – implementing ForTheL in GLF*. URL: https://gl.mathhub.info/comma/glforthel (visited on 03/27/2020).

[Hal19] Thomas Hales. *An Argument for Controlled Natural Languages in Mathematics*. 2019. URL: https://jiggerwit.wordpress.com/2019/06/20/an-argument-for-controlled-natural-languages-in-mathematics/ (visited on 05/09/2020).

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. "A framework for defining logics". In: *Journal of the Association for Computing Machinery* 40.1 (1993), pp. 143–184.

[Jup] *Project Jupyter*. URL: http://www.jupyter.org (visited on 08/22/2017).

[KR93] Hans Kamp and Uwe Reyle. *From Discourse to Logic: Introduction to Model-Theoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory*. Dordrecht: Kluwer, 1993.

[KS19] Michael Kohlhase and Jan Frederik Schaefer. "GF + MMT = GLF – From Language to Semantics through LF". In: *Proceedings of the Fourteenth Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP 2019*. Ed. by Dale Miller and Ivan Scagnetto. Vol. 307. Electronic Proceedings in Theoretical Computer Science (EPTCS), 2019, pp. 24–39. DOI: 10.4204/EPTCS.307.4.

[LBS20] Michael Kohlhase. *Logic-Based Natural Language Processing: Lecture Notes*. 2020. URL: http://kwarc.info/teaching/LBS/notes.pdf (visited on 05/07/2020).

[MMT] *MMT – Language and System for the Uniform Representation of Knowledge*. URL: https://uniformal.github.io/.

[Pas07] Andrei Paskevich. *The syntax and semantics of the ForTheL language*. 2007.

[Ran11] Aarne Ranta. *Grammatical Framework: Programming with Multilingual Grammars*. ISBN-10: 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth). Stanford: CSLI Publications, 2011.

[SK20] Jan Frederik Schaefer and Michael Kohlhase. "The GLIF System: A Framework for Inference-Based Natural-Language Understanding". submitted. 2020. URL: http://kwarc.info/kohlhase/submit/cicm20-glif.pdf.