Proceedings of the 6th Workshop on Formal and Cognitive Reasoning

GLIF: A Declarative Framework for Symbolic Natural Language Understanding

Jan Frederik Schaefer and Michael Kohlhase[0000-0002-9859-6337]

Computer Science, FAU Erlangen-Nürnberg, Erlangen, Germany

Abstract. With the Grammatical Logical Inference Framework (GLIF), a user can implement the core of symbolic language understanding systems by describing three components, each of which is based on a declarative framework: parsing (with the Grammatical Framework GF), semantics construction (with MMT), and inference (with ELPI). The logical frameworks underlying these tools are all based on LF, which makes the connection very natural. Example applications are the prototyping of controlled natural languages or experiments with new approaches to natural-language semantics. We use Jupyter notebooks for a unified interface that allows quick development of small ideas as well as testing on example sentences.

1 Introduction

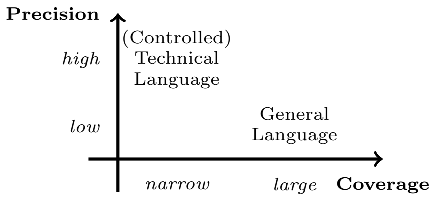


Fig. 1. Tractable NLP problems.

In recent years, the field of natural-language processing has seen a lot of progress through the use of deep learning tools, resulting in many useful applications such as automated text translation. Yet, we want to focus on symbolic approaches. While they cannot compete with deep learning in wide-coverage tasks, they offer high precision processing in restricted domains. A prime example of this is technical language – scientific articles, legal documents, software specification, etc. Using machine learning for such documents poses a number of challenges: little training data exists and high precision (or even verifiability) is mandatory. In some cases, the need for reliable processing means that natural language is abandoned altogether and replaced by a formal language. This, of course, entails a steep learning curve for potential contributors. A compromise are **controlled natural languages** (CNL): formal languages with well-defined semantics that imitate or form a fragment of natural language. Probably the most well-known controlled natural language is the general-purpose language Attempto Controlled English (ACE) [FSS98].

An alternative is Montague's "method of fragments" [Mon70], which aims to exhaust natural languages by a series of ever-increasing "natural language fragments". The main difference to CNLs – fragments are formal languages as well – is that the meaning construction needs to be unambiguous and can be accompanied by context-sensitive semantic/pragmatic analysis phase.

To support the design of such languages, we introduce GLIF, the Grammatical Logical Inference Framework. GLIF is intended as a general framework for the prototyping and implementation of natural-language understanding systems. It allows users to describe a pipeline consisting of three steps: *i*) **parsing**, *ii*) **semantics construction**: mapping abstract syntax trees to (possibly underspecified) expressions, and *iii*) **semantic/pragmatic analysis**: computing fully specified logical expressions and reconciling them with the utterance context – usually an inference-based process. Each step in the pipeline is based on a different framework: Parsing and grammar development are based on the Grammatical Framework (GF) [Ran11], semantics construction and logic development are based on MMT [MMT], and inference is based on ELPI [SCT15], an extension of λProlog. GLIF is an extension of the Grammatical Logical Framework (GLF) [KS19], which doesn't have an inference component.

The third (inference) step is essentially the "understanding part" in the pipeline. Depending on the application, it can have a variety of functions. It may simply modify the results of the semantics construction, which by design is bound to be compositional, with more complex operations, such as simplification or semantic pruning. The inference step can also be used for ambiguity resolution (e.g. by discarding contradictory readings by theorem proving) or the maintenance of a symbolic discourse or dialogue model.

Historically, symbolic natural-language understanding systems have been implemented in declarative programming languages like Prolog or Haskell. We believe that a dedicated framework like GLIF can simplify and speed up the implementation and make the result more maintainable. We are not aware of any other frameworks like GLIF – the closest might be the Grammatical Framework GF, which is one of the components of GLIF.

As a small running example for this paper, we will implement a fragment of English for specifying physical properties of different objects with the example sentence

"*the ball has a mass of 5 kg and a kinetic energy of 12 mN*",

where we use the inference step to disambiguate whether "*12 mN*" stands for "*12 meter Newton*" or "*12 milli Newton*".

2 The GLIF System

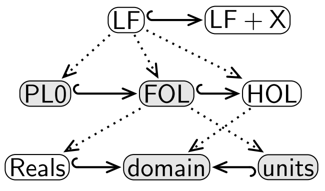


Fig. 2. Meta-Theories in MMT

Before diving into the details of the GLIF pipeline, we need to briefly introduce MMT, the centerpiece of GLIF. MMT is a modular, foundation-independent knowledge representation framework [MMT]. Knowledge is represented in the form of **theories**, which contain a sequence of **declarations** for symbols, axioms, definitions, and theorems. Theories can be linked via **theory morphisms**: truth-preserving mappings which assign expressions in the target theory to symbols in the source theory. **Meta-theories** – the ones imported via the dotted arrows in Figure 2 – furnish the languages for specifying properties and relations. Theories are used at various levels: the **domain theories** modularly formalize properties of the domain; units and quantities in our running example. Their meta-theories are logics (here propositional, first-order, and higher-order logics), which are specified in e.g. the Edinburgh Logical Framework LF [HHP93] or extensions (LF + X). Meaning trickles down the meta relation from **urtheories** like LF via the meta-theory morphisms all the way to the domain theories.

GLIF exploits the similarity of LF with the logical frameworks underlying GF and ELPI, which results in very intuitive transitions between the three systems involved. Figure 3 illustrates the GLIF pipeline. In the following sections we will take a closer look at each of the three processing steps.

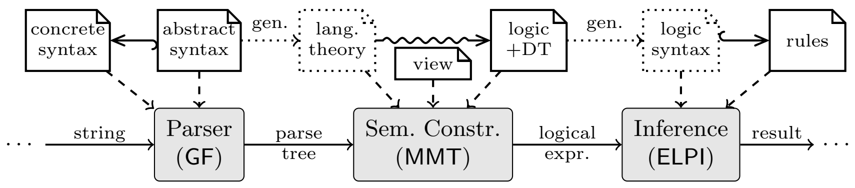


Fig. 3. The GLIF Pipeline. Dashed arrows (⇢) indicate what specifications (🗋) are needed for each processing step. Some specifications (⬚) can be generated automatically from another specification ($\overset{\text{gen.}}{\dashrightarrow}$). ↪ indicates imports and ⇝ illustrates the semantics construction view.

2.1 Parsing

For the first step (parsing), we use the Grammatical Framework (GF) [Ran11], which provides powerful mechanisms for the development of natural language grammars and comes with a library that implements the basic morphology and syntax of ≥ 38 languages. GF grammars come in two parts: abstract syntax and concrete syntax. The **abstract syntax** specifies the abstract syntax trees (ASTs) supported by the grammar in a type-theoretical fashion, while the **concrete syntax** describes how these ASTs correspond to strings in a language. For our example sentences, we have e.g. the following rules in the abstract syntax:

```
measure : Measurable -> Int -> Unit -> Measurement;
combine : Measurement -> Measurement -> Measurement;
hasProp : Object -> Measurement -> S;         -- S = sentence
```

The measure rule combines something measurable (like "*kinetic energy*"), with an integer and a unit into a Measurement (e.g. "*a kinetic energy of 12 mN*"). combine simply combines the measurements of two different properties ("*a mass* *of 5 kg and a kinetic energy of 12 mN*"). In the GF concrete syntax we can describe how these rules correspond to strings:

```
measure m int unit = "a" ++ m ++ "of" ++ int.s ++ unit;
combine a b = a ++ "and" ++ b;
```

In this very simple example, we only combine token (sequences) with the ++ operator. While the rules intuitively describe the linearization (mapping ASTs to strings), GF can also generate a parser from such a specification. For more complex language phenomena, GF offers powerful mechanisms like records and parameter types. Let's say that we want to support plurals (e.g. "*the ball and the train have a mass of 5 kg*"). Then we have to pick the right verb form of "*have*" depending on the number of the noun. For this we turn objects into records with a field s for the string representation and n for the number:

```
hasProp obj m = obj.s ++ have ! obj.n ++ m;
```

In general, developers can avoid dealing with such low-level problems by using GF's Resource Grammar Library, which covers the basic syntax and morphology of many languages.

With the abstract and concrete syntax in place, we can start parsing sentences. If a sentence is ambiguous according to the grammar, GF generates multiple ASTs. For the example sentence "*the ball has a mass of 5 kg and a kinetic energy of 12 mN*", the two trees are shown in Figure 4.

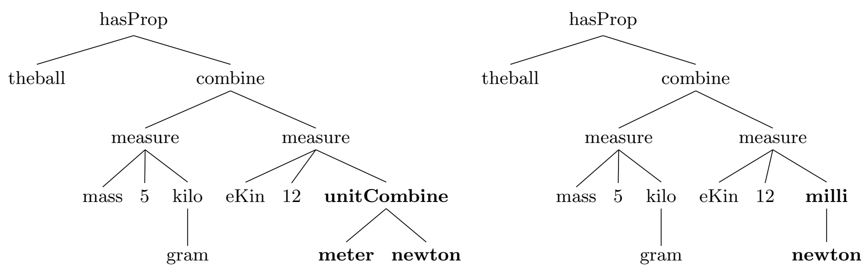


Fig. 4. The ambiguity of *mN* results in two different ASTs.

2.2 Semantics Construction

The semantics construction is implemented in MMT. We connect GF to MMT by reinterpreting the abstract syntax as an MMT theory (the **language theory**). This lets us interpret the ASTs as terms in that theory. The target of the semantics construction is an MMT theory that describes the logic syntax and a domain theory. For our example, we need a type for propositions, which we will denote by o, and logical conjunction, which we will denote with the infix operator ∧. We will also need some information about units.

```
theory PL0 =
  proposition : type ❘ # o ❙
  and : o → o → o ❘ # 1 ∧ 2 ❙
  . . . ❚
```

```
theory units =
  unit : type ❘ # u ❙
  mult : u → u → u ❘ # 1 · 2 ❙
  gram : u ❘ # gram ❙ ❚
```

At the heart of the semantics construction is now a **view** – a particular type of theory morphism – that maps every symbol in the language theory to an object in the target logic/domain theory. The translation of ASTs to logical expressions thus boils down to applying a view to an MMT term. The compositionality of this process typically means that some subtrees have to be translated to λ-functions (a well-established approach in natural language semantics). In our case, for example, "*a mass of 5 kg*" gets translated to $\lambda x.mass\ x\ (quant\ 5\ kilo\ gram)$. The combine node, which combines measurements M and N, becomes $\lambda x.Mx \wedge Nx$. In MMT syntax we write this as

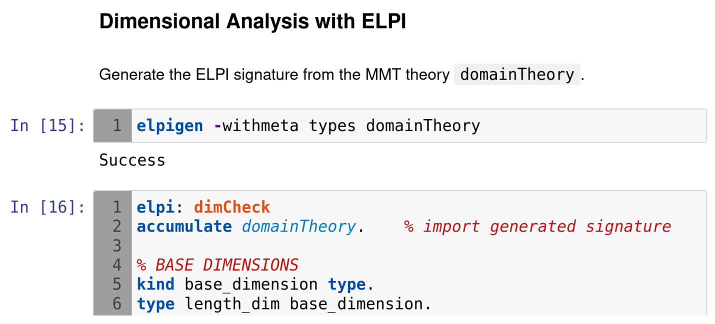
```
combine = [M,N] [x] (M x) ∧ (N x) ❙
```

where $[\cdot]$ is MMT's notation for λ-abstraction. We also map the syntactic categories to types in the logic:

```
Measurement = ι → o ❙    // unary predicates ❙
```

This enables MMT to rigorously type-check the semantics construction. After the semantics construction is applied to an AST, the λ-functions are eliminated through β-reduction and we get the following two logical expressions:

$$\text{(mass theball (quant 5 kilo gram))} \wedge \text{(ekin theball (quant 12 milli Newton))}$$
$$\text{(mass theball (quant 5 kilo gram))} \wedge \text{(ekin theball (quant 12 meter}\cdot\text{Newton))}$$



Dimensional Analysis with ELPI

Generate the ELPI signature from the MMT theory `domainTheory`.

```
In [15]:  elpigen -withmeta types domainTheory

Success

In [16]:  elpi: dimCheck
          accumulate domainTheory.    % import generated signature

          % BASE DIMENSIONS
          kind base_dimension type.
          type length_dim base_dimension.
```

Fig. 5. ELPI code in Jupyter.

2.3 Inference

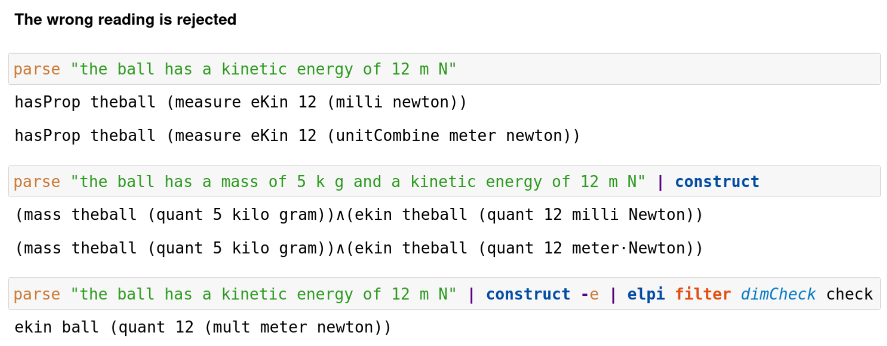
For the inference step, we use ELPI [SCT15], an extension of λProlog. The advantage of choosing λProlog over classical Prolog variants is that variable binding can be naturally represented through λ-expressions (*higher-order abstract syntax*), which is needed for many logics, including first-order logic. MMT supports the transition to ELPI by generating the signature of the logic and domain theory, and by exporting the generated logical expressions in ELPI syntax. Here are the first lines of the signature generated for our example:

```
kind proposition type.
type and proposition -> proposition -> proposition.
```

MMT can also generate ELPI provers from calculi specified in MMT [Koh+20]. For our example, we use hand-written rules to perform a dimensional analysis, which checks whether the units match the expected quantity. The GLIF interface (next section) provides different commands for using ELPI predicates in a pipeline to e.g. transform or filter logical expressions. In our example, of course, we want to filter the results of the semantics construction.

2.4 User Interface

GLIF can be used through Jupyter notebooks via a custom kernel. ELPI, GF and MMT content can be implemented directly in the notebooks. Figure 5 shows how the ELPI signature can be generated (**elpigen**) and afterwards used with λProlog's **accumulate**. For larger projects, however, it is generally preferable to develop the content outside of notebooks. Notebooks can then still be used for testing and demonstrating the developed pipelines. Figure 6 demonstrates the entire pipeline for our example sentence: **parse** parses the input, **construct** applies the semantics construction, and **elpi filter** filters out any results rejected by the dimensional analysis. In the example, the reading milli Newton for mN is discarded. Other features of the notebook interface include the (visual) display of parse trees and stub generation e.g. for the semantics construction. The Jupyter interface of GLF – the predecessor of GLIF – is described in more detail at [SAK20].



The wrong reading is rejected

```
parse "the ball has a kinetic energy of 12 m N"
hasProp theball (measure eKin 12 (milli newton))
hasProp theball (measure eKin 12 (unitCombine meter newton))

parse "the ball has a mass of 5 k g and a kinetic energy of 12 m N" | construct
(mass theball (quant 5 kilo gram))∧(ekin theball (quant 12 milli Newton))
(mass theball (quant 5 kilo gram))∧(ekin theball (quant 12 meter·Newton))

parse "the ball has a kinetic energy of 12 m N" | construct -e | elpi filter dimCheck check
ekin ball (quant 12 (mult meter newton))
```

Fig. 6. Parsing, semantics construction and filtering in Jupyter. The | operator pipes the output of the previous command into the next command.

3 Conclusion

We have presented GLIF, a declarative framework in which natural-language understanding systems can be implemented by specifying *i*) a grammar, *ii*) a target logic and domain theory, *iii*) the semantics construction, *iv*) and inference rules.

We have used GLIF in a one-semester course on logic-based natural-language semantics at FAU Erlangen-Nürnberg [LBS20], implementing a sequence of Montague-style fragments of English and tableau-based semantic/pragmatic analysis processes.

As a larger case study, [SAK20] presents a description of our attempt to re-implement an existing controlled natural language for mathematics using a predecessor of GLIF. The resulting pipeline can parse sentences like "*a subset of S is a set T such that every element of T belongs to S*", and translates them into first-order logic:

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

GLIF can be used through Jupyter notebooks, which increases the accessibility significantly. More details on a previous version of the Jupyter kernel (that doesn't support inference), can be found at [SAK20]. The Jupyter kernel itself, along with a link to an online demo, is at [GLIF].

References

[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.

[GLIF] *GLIF Kernel*. URL: https://github.com/KWARC/GLIF (visited on 03/22/2020).

[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.

[Koh+20] Michael Kohlhase, Florian Rabe, Claudio Sacerdoti Coen, and Jan Frederik Schaefer. "Logic-Independent Proof Search in Logical Frameworks (short paper)". In: *10th International Joint Conference on Automated Reasoning (IJCAR 2020)*. Ed. by Nicolas Peltier and Viorica Sofronie-Stokkermans. Springer Verlag, 2020.

[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*. 2020. URL: https://kwarc.info/teaching/LBS/notes.pdf (visited on 05/30/2020).

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

[Mon70] R. Montague. “English as a Formal Language”. In: Reprinted in [Tho74], 188–221. Edizioni di Communita, Milan, 1970, pp. 189–224.

[Ran11] Aarne Ranta. *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, 2011.

[SAK20] Jan Frederik Schaefer, Kai Amann, and Michael Kohlhase. “Prototyping Controlled Mathematical Languages in Jupyter Notebooks”. In: *Mathematical Software – ICMS 2020. 7th international conference*. Ed. by Anna Maria Bigatti, Jacques Carette, James H. Davenport, Michael Joswig, and Timo de Wolff. Vol. 12097. LNCS. Springer, 2020, pp. 406–415. URL: https://kwarc.info/kohlhase/papers/icms20-glf-jupyter.pdf.

[SCT15] Claudio Sacerdoti Coen and Enrico Tassi. *The ELPI system*. 2015. URL: https://github.com/LPCIC/elpi.

[Tho74] R. Thomason, ed. *Formal Philosophy: selected Papers of Richard Montague*. Yale University Press, New Haven, CT, 1974.