GF + MMT = GLF
From Language to Semantics through LF

Michael Kohlhase Jan Frederik Schaefer

Computer Science, FAU Erlangen-Nürnberg

These days, vast amounts of knowledge are available online, most of it in written form. Search engines help us access this knowledge, but aggregating, relating and reasoning with it is still a predominantly human effort. One of the key challenges for automated reasoning based on natural-language texts is the need to extract meaning (semantics) from texts. Natural language understanding (NLU) systems describe the conversion from a set of natural language utterances to terms in a particular logic. Tools for the co-development of grammar and target logic are currently largely missing.

We will describe the **Grammatical Logical Framework** (GLF), a combination of two existing frameworks, in which large parts of a symbolic, rule-based NLU system can be developed and implemented: the Grammatical Framework (GF) and MMT. GF is a tool for syntactic analysis, generation, and translation with complex natural language grammars and MMT can be used to specify logical systems and to represent knowledge in them. Combining these tools is possible, because they are based on compatible logical frameworks: Martin-Löf type theory and LF. The flexibility of logical frameworks is needed, as NLU research has not settled on a particular target logic for meaning representation. Instead, new logics are developed all the time to handle various language phenomena. GLF allows users to develop the logic and the language parsing components in parallel, and to connect them for experimentation with the entire pipeline.

1 Introduction

Natural language semantics studies the meaning of natural language utterances. A fundamental conceptual tool for this are **truth conditions**: the set of conditions under which an NL utterance is true. For example, "*John loves Mary*" is true if and only if John indeed loves Mary. Somewhat less tautologously: two assertions have the same meaning, if they have the same truth conditions. We can therefore identify the meaning of an assertion with its truth conditions [4]. This notion of "meaning" is very general, but also not very constructive. Therefore truth conditions are generally thought of as a minimal axiomatization of the domains of discourse which entails the assertion.

To understand this setup, assume that we use a formal language $\mathcal{FL}$ to express truth conditions. The meaning of "*John loves Mary*" could then be $love(john, mary) \in \mathcal{FL}$. If $\mathcal{FL}$ is the formal language of a logical system, we also have an interpretation function $\mathcal{I}_\varphi$ of $\mathcal{FL}$ expressions into a model and a calculus $\mathcal{C}$ with a derivation relation $\vdash_\mathcal{C}$. If $\mathcal{C}$ is sound and complete, the upper rectangle in Figure 1 commutes. If the calculus $\mathcal{C}$ is an adequate

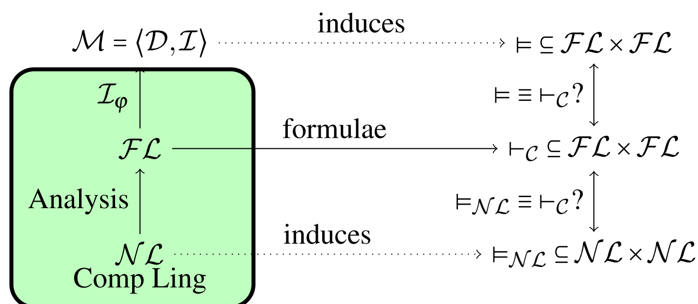

Figure 1: Natural-language inference on different levels.

D. Miller, I. Scagnetto (Eds.)
Logical Frameworks and Meta-Languages:
Theory and Practice 2019 (LFMTP 2019)
EPTCS 307, 2019, pp. 24–39, doi:10.4204/EPTCS.307.4

© Michael Kohlhase & Jan Frederik Schaefer
This work is licensed under the Creative Commons
Attribution-Share Alike License.

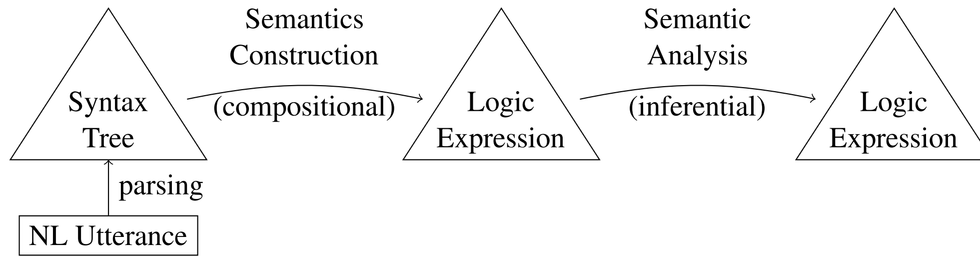

Figure 2: The pipeline of an NLU system.

model for the natural language entailment relation $\models_{\mathcal{NL}}$ – also called "textual entailment" in the linguistics literature – both rectangles in Figure 1 commute and we have a good model for truth conditions and logical entailment for natural language utterances. In this case, it suffices to specify the translation from natural language $\mathcal{NL}$ to the formal language $\mathcal{FL}$ along with a calculus $\mathcal{C}$. And in general, the "NL semantics" literature restricts itself to the box in Figure 1, entrusting the upper square to logicians and the equivalence of $\mathcal{M}$-entailment and textual entailment $\models_{\mathcal{NL}}$ to the logic developers. At the same time, NL semanticists continually need extensions to $\mathcal{FL}$ and $\mathcal{C}$ to model new NL phenomena.

In particular, it is still unachievable to describe a translation from the entirety of natural language into some formal language. Instead, researchers rather focus on particular phenomena in natural language by describing a small subset of natural language utterances (a **fragment**) along with the meaning of these utterances. This **method of fragments** was established by Richard Montague [14]. It typically results in the description of three components:

1. a **grammar** that fixes the language fragment and generates syntax trees
2. a **formal system** in which the semantics of utterances can be expressed
3. a way to transform syntax trees to expressions in the formal system, which is often referred to as **semantics construction**

The semantics construction is based on the **compositionality principle**: the idea that the meaning of a complex utterance is determined by the meaning of its constituents. Thereby, the semantics construction boils down to mapping grammar rules to corresponding semantic operations. Consider, for example, the grammar rule `<sentence> ::= <sentence> "and" <sentence>`. It corresponds to the semantic operation $[\![A]\!] \wedge [\![B]\!]$, where $[\![A]\!]$ and $[\![B]\!]$ are the meanings of the constituent sentences. The semantics construction may be followed by a **semantic analysis**[1], which comprises various non-compositional operations such as inference, anaphora resolution, or contextual anchoring.

Symbolic **natural-language understanding (*NLU*) systems** describe the entire pipeline from strings to semantic representations (Figure 2). They have been used to describe the semantics of a variety of natural-language phenomena. In the process, many different logics have been developed. However, the experiments were mostly done with pen and paper, and have rarely been implemented in software. This can lead to researchers focusing either on the linguistic side or on the logical side of the problem, while the actual semantics construction remains vague.

When someone actually implemented such an NLU system, it was usually done in a programming language like Prolog or Haskell – see e.g. [1, 5]. In both cases, the authors claim that the programming language is an NLU framework – in the first case since Prolog is a declarative programming language and in the second since Haskell is very high-level. In any case, the NLU system requires a considerable – potentially prohibitive – amount of programming work. As far as we can tell, there is no fully declarative

[1] In Anglo-Saxon literature this is sometimes called **pragmatics**.

framework that could be used to do both, the grammar development as well as the logic development and keep them in sync.

In this paper we describe our efforts to create the **Grammatical Logical Framework** (GLF). It combines an existing framework for natural language grammars with an existing framework for logic development. Concretely, we combine the **Grammatical Framework** (GF) [22], with the logic development tool MMT [19]. This is possible, because the logical frameworks underlying these tools are compatible. GF handles the natural language parsing and generates terms (parse trees) in a logical framework (Martin-Löf type theory [21]). MMT, which supports LF and various extensions, maps these terms to expressions in the desired target logic – see Figure 5.

In GLF, an NLU researcher can specify a fragment of a language in GF and, in parallel, develop a logic in MMT, along with a domain theory and the semantics construction. Our framework supports this in various ways, such as:

- it allows the researcher to try out the entire pipeline from an utterance to its logical representation
- it checks the totality of the semantics construction
- the grammar and the logic are type-checked as usual in GF and MMT

We admit that symbolic natural language understanding has dropped from the limelight of computational linguistic research in the last two decades in favor of machine-learning-based approaches. But the success of these has only shadowed the question of semantic analysis and natural language inference. We see a cautious revival of symbolic/logic-based methods in computational linguistics, and we hope that GLF can serve as a tool to facilitate this.

The symbolic approach to NLU needs extensive resources (e.g. grammars and ontologies). Aarne Ranta, the creator of GF, distinguish two areas of NL applications: **consumer tasks** and **producer tasks** [20]. Consumer tasks require large coverage – often achieved through machine learning – and are therefore typically limited in their precision. An example of this would be machine learning-based text translation a la Google Translate. Producer tasks, on the other hand, require high precision, but are restricted in their coverage to a few thousand concepts. An example are technical manuals for complex machinery in dozens of languages, where the consequences of mistranslation may be catastrophic. Beyond translation, producer tasks – the natural hunting grounds of GLF– include understanding of mathematical papers, laws or contracts.

Overview First, we will describe GF and MMT (Sections 2 and 3). After an overview of the GLF system (Section 4), we will describe the semantics construction and semantic analysis (Sections 5 and 6) using a running example. Section 7 contains more examples of how this framework can be used. Section 8 concludes the paper and discusses future work.

Acknowledgements We are grateful for the discussions with and insights from Aarne Ranta, Florian Rabe, and finally Dennis Müller, who has also prototyped an early version of GLF. The work reported here was supported by the German Research Foundation (DFG) under grant KO 2428/18.

2 GF: The Grammatical Framework

The **Grammatical Framework** (GF) [22, 7] can be used to create multilingual grammar applications. GF grammars are divided into two parts: **abstract syntax** and **concrete syntaxes**. The abstract syntax describes the **ASTs** (**abstract syntax trees** or **abstract syntax terms**) covered by the grammar. The concrete syntaxes are AST linearization rules in a specific natural language.

Let us consider a small example: Listing 1 shows an abstract syntax for representing some statements about everyday life such as "*Joan runs and Mary loves Joan*". First, three basic types are introduced (`Stmt`, `Person`, `Action`) with the keyword **`cat`** (in GF they are called **categories**). Afterwards, several function constants are introduced with the keyword **`fun`**. The example utterance "*Mary loves Joan*" would correspond to the AST `act mary (love joan)`. Formally, GF is based on a version of constructive type theory [21]. It supports dependent types, but, in our experience, these are not very useful for most natural-language grammar applications.

```
abstract Life = {
  cat
    Stmt ; Person ; Action ;
  fun
    act : Person -> Action -> Stmt ;
    and : Stmt -> Stmt -> Stmt ;

    joan        : Person ;
    mary        : Person ;
    run         : Action ;
    loveOneself : Action ;
    love        : Person -> Action ;
}
```

```
concrete LifeEng of Life = {
  lincat
    Stmt, Person, Action = Str ;
  lin
    act pers action = pers ++ action ;
    and a b         = a ++ "and" ++ b ;

    joan        = "Joan" ;
    mary        = "Mary" ;
    run         = "runs" ;
    loveOneself = "loves" ++ "herself" ;
    love person = "loves" ++ person ;
}
```

Listing 1: Abstract syntax to talk about ever-day life along with English linearization rules.

GF's concrete syntaxes describe how the ASTs are linearized in a particular natural language. Listing 1 shows a concrete syntax for the English language. First, the linearization types of the categories are defined (**`lincat`**). GF provides a powerful type system for the concrete syntax with record types and more. For this example the basic type `Str` suffices; this would change if a male person is added, since we would have to distinguish between "*loves herself*" and "*loves himself*". The keyword **`lin`** is used to describe the linearizations of function constants.

With the concrete syntax, ASTs can be linearized into English strings. GF also generates a parser from the linearization rules, which allows us to parse English strings into ASTs. By creating another concrete syntax, e.g. for German, we can parse strings in one language and linearize the ASTs into a different language. This allows us to translate between languages. For instance the invocation

```
parse -lang=Eng -cat=Stmt "Mary loves herself" | linearize -lang=Ger
```

in the GF shell yields the result

```
Maria liebt sich
```

The example above is of course a very simple one and does not contain any of the challenges one would typically face in a natural-language grammar. Any more complex natural-language application will have to implement a large number of rules for handling the word order and word forms. In English, for example, verb endings depend on the plurality of the subject. These rules can be implemented in GF's concrete syntaxes using records, tables, etc. Since these language-specific rules are needed in most projects, the GF community provides the **Resource Grammar Library** [9]. It contains rules for currently 36 languages and is an invaluable resource for creating natural-language applications.

3 MMT: Meta-Meta Theories/Tool

OMDoc/MMT (Meta-Meta Theories) is a modular, foundation-independent framework for representing knowledge [19, 16]. The MMT system (Meta-Meta Tool [18, 13]) acts as an OMDoc/MMT API and implements various knowledge management services including type/proof checking.

OMDoc/MMT knowledge is represented in **theories**, which contain (among other things) declarations of constants. A constant declaration `c[:τ][=δ][#ν]` can have various components such as a type τ, a definiens δ or a notation ν.

OMDoc/MMT is a modular framework: domain ontologies and logics are represented as graphs of theory presentations inter-linked by **theory morphisms** that model inheritance and interpretation. In practice OMDoc/MMT development follows a "little theories" paradigm, where each theory presentation only contributes a small number of declarations to maximize re-use of content.

Due to the foundation-independence, various logical frameworks can be implemented in MMT, but in practice LF and extensions are usually sufficient to represent a wide range of formal systems in MMT. See [3, 12] for the LATIN Logic Atlas and Integrator, a large modular theory graph of LF-encoded logics.

As an example, let us create an MMT theory `PropLogicSyntax` for propositional logic, which we can base on LF. Note that LF is just another theory in MMT, the **meta theory** of `PropLogicSyntax`. The theory LF provides λ for functions, Π for dependent types, the kind `type` , and the function type constructor →. Apart from a meta theory, `PropLogicSyntax` has four constant declarations (Listing 2). First, the constant `prop` is declared as a type with the notation `o` (the symbol `#` introduces the notation). The components of a constant declaration are separated by the delimiter `|`, declarations with ‖, and theories/views with ⟦. We can declare conjunction (`and`) and negation (`neg`) as binary/unary operations on propositions using the notation `o` introduced for `prop`. Applying De Morgan's law, disjunction (`or`) can be defined in terms of conjunction and negation. Note that in MMT lambda expressions like $\lambda x, y.M$ are written as `[x,y] M`.

```
theory PropLogicSyntax : ur:?LF =
    prop : type        | # o ‖
    and  : o → o → o | # 1 ∧ 2 ‖
    neg  : o → o       | # ¬ 1 ‖
    or   : o → o → o | # 1 ∨ 2 |
         = [x,y] ¬ ((¬ x) ∧ (¬ y)) ‖
⟦
```

```
theory PropLogicND : ur:?LF =
    include ?PropLogicSyntax ‖
    ded : o → type | # ⊢ 1 ‖

    andI  : {A,B} ⊢ A → ⊢ B → ⊢ (A ∧ B) ‖
    andEr : {A,B} ⊢ (A ∧ B) → ⊢ B ‖
    ...
```

Listing 2: Propositional logic in MMT: syntax (left) and proof theory (right).

The right side of Listing 2 shows the usual representation of the natural deduction calculus for propositional logic via the Curry-Howard isomorphism. We will use it for the verification of truth conditions in domain theories below.

To model the semantics of propositional logic in set theory, we create a new theory `PropLogicModel` (Listing 3) that is based on set theory (`sets:?AllSets`). In OMDoc/MMT terms, set theory is the meta theory of `PropLogicModel`. The meta theory relation is a theory morphism, very similar to inclusion/inheritance.

We want to interpret propositions as sets of satisfying variable assignments and interpret falsity as the empty set, therefore `PropLogicModel` has a single constant declaration, which introduces a new type constant with the notation `A` for variable assignments.

The meaning of `PropLogicSyntax` can now be established with another kind of theory morphism: a

```
theory PropLogicModel :
            sets:?AllSets =
    assignment : type | # A ‖
❚
```

```
view PropLogicSemantics :
        ?PropLogicSyntax -> ?PropLogicModel =
    prop = set A ‖
    and = [φ, ψ] φ ∩ ψ ‖
    neg = [φ] (fullset A) \ φ ‖
❚
```

Listing 3: Semantics description as view (right) into `PropLogicModel` (left).

view maps undefined constants from the source theory to objects in the target theory. In this case, we will describe the meaning of `PropLogicSyntax` with a view from `PropLogicSyntax` to `PropLogicModel` (Listing 3). A proposition can be represented as the set of variable assignments that make it true. Then, the conjunction of two propositions corresponds to the intersection of two sets and negation corresponds to taking the complement set. As disjunction (`or`) is defined in terms of conjunction and negation, we do not need to map it to anything here. Figure 3 provides an overview of the different theories and morphisms we have used.

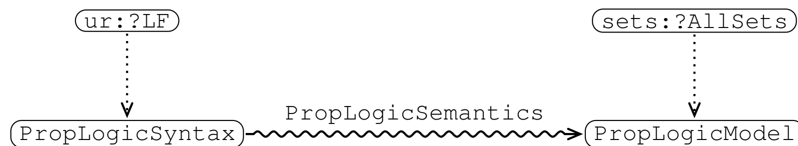

Figure 3: Propositional Logic as an OMDoc/MMT Theory Graph.

4 GLF: The Grammatical Logical Framework

GLF combines GF and MMT by exploiting the compatibility of the underlying logical frameworks. In the next sections we will explain how this leads to a framework for natural-language semantics. We start with the information and software architecture of the system.

4.1 GF vs MMT – Two Sister Formats

GLF's combination of GF and MMT is based on the fact that GF's abstract syntaxes can be trivially represented as MMT/LF theories. We will refer to an MMT theory that corresponds to a GF abstract syntax module as a **language theory**.

We can illustrate this using the `Life` grammar from Listing 1. For instructive reasons, we will split it into two parts: a `LifeGrammar` that deals with the "grammatical" structures of a language and a `LifeLex` that contains the lexical entries. Listing 4 shows the `LifeGrammar` in GF, along with its language theory in MMT. GF categories correspond to type constants and GF functions to function constants in MMT. Listing 4 also shows the `LifeLex` abstract syntax in GF with the corresponding language theory. Note that `LifeLex` extends `LifeGrammar` in GF, and correspondingly in MMT `LifeLex` includes `LifeGrammar`. Generally, the MMT module system subsumes the one of GF, thus we can build the information architecture in parallel.

The similarity between GF abstract syntaxes and their language theories in MMT enables us to leave GF after parsing and continue with the semantics construction in MMT (Section 5). Specifically, an AST

```
abstract LifeGrammar = {
  cat
    Stmt ;
    Person ;
    Action ;
  fun
    act : Person -> Action -> Stmt ;
    and : Stmt -> Stmt -> Stmt ;
}

abstract LifeLex = LifeGrammar ** {
  fun
    joan         : Person ;
    mary         : Person ;
    run          : Action ;
    loveOneself  : Action ;
    love         : Person -> Action ;
}
```

```
theory LifeGrammar : ur:?LF =

    Stmt   : type ‖
    Person : type ‖
    Action : type ‖

    act : Person → Action → Stmt ‖
    and : Stmt → Stmt → Stmt ‖
❚

theory LifeLex : ur:?LF =
    include ?LifeGrammar ‖
    joan        : Person ‖
    mary        : Person ‖
    run         : Action ‖
    loveOneself : Action ‖
    love        : Person → Action ‖
❚
```

Listing 4: The `LifeGrammar` and its extension `LifeLex` in GF (left) and MMT (right).

in GF like `act joan (love mary)` can trivially be mapped to the MMT term `act joan (love mary)`.

4.2 The GLF System

GLF is a relatively thin wrapper around the unchanged GF and MMT systems: GF is started in server mode and GLF communicates with it via HTTP requests – this turned out to be easier to set up than using the GF Java bindings. The bridge between GF and MMT consists of

1. a small script that given an abstract GF grammar $\mathcal{G}$ generates the language theory $\widehat{\mathcal{G}}$ and a stub for the semantics construction view $\widetilde{\mathcal{G}}$ (cf. Section 5) and
2. a translator that translates $\mathcal{G}$-ASTs in GF into $\widehat{\mathcal{G}}$-terms in MMT (cf. Section 5).

Figure 4 shows the pipeline: first, an utterance is parsed by GF into an AST, which GLF translates into an equivalent OMDoc/MMT term, which MMT can use for the semantics construction and analysis and pass on to an application. We can see this as a refinement/implementation of Figure 2.

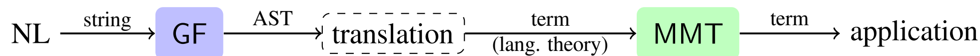

Figure 4: The general GLF pipeline.

4.3 GLF Fragments

Together, the GF grammars and MMT theories constitute a full formal basis for parsing and inference, therefore we call it a **GLF Fragment**. Even though this example is very basic and didactically motivated, the general structure of a GLF fragment is the same. It consists of:

1. an abstract GF/MMT syntax, such as the one in Listing 4,
2. a set of concrete GF syntaxes for them, as in Listing 1 on the right,
3. an MMT logic, such as the one in Listings 2 (syntax) and 3 (semantics), and

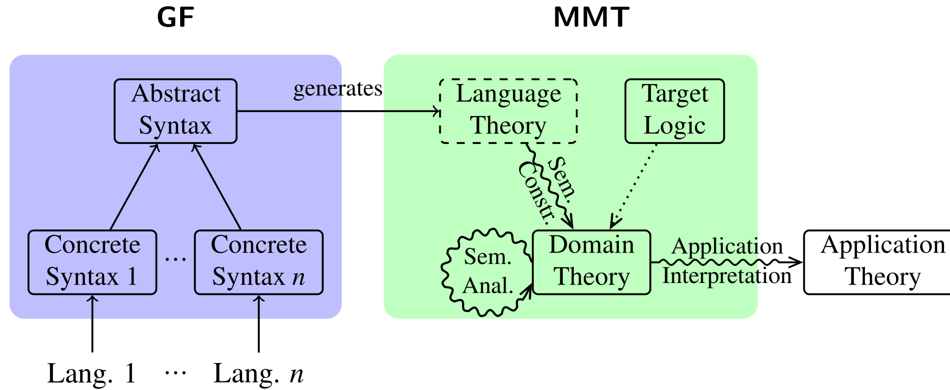

Figure 5: The GLF($\mathcal{F}$) Pipeline.

4. an MMT view for the semantics construction, such as the one in Listing 6.

We write GLF($\mathcal{F}$) for GLF with fragment $\mathcal{F}$ loaded. As we can directly experiment with a GLF package $\mathcal{F}$, GLF($\mathcal{F}$) can be used as a fragment development system.

Figure 5 refines the pipeline from Figure 4 taking the components of the GLF fragment $\mathcal{F}$ into account: an utterance is parsed by GF with one of the concrete syntaxes to obtain an AST (specified by the abstract syntax). Different concrete syntaxes correspond to different languages. Afterwards, GLF translates the AST into an equivalent MMT term in the language theory. Semantics construction, via MMT (cf. Section 5), and semantic analysis in MMT (Section 6) follow.

4.4 GLF Applications

We provide a small GLF distribution at [10] that comes with MMT and the grammars and semantics constructions described in this paper. Concretely, this allows the reader to try out the examples in the command-line by entering sentences and receiving the result of the semantics construction. Here is an abbreviated example interaction based on an example extension of the `Life` fragment described in Section 7.1:

```
Please enter a sentence: John and Mary run
I got the following interpretations:
(run' john')∧(run' mary')
Please enter a sentence: John loves everyone
I got the following interpretations:
∀[x:ι]love' john' x
```

Note that the example above doesn't require any implementation work at all – only a GLF fragment, which gets passed to GLF via command-line arguments.

In Section 6 we will motivate the use of tableaux for semantic analysis. There is also a small demo for that, which allows the user to enter sentences and outputs the updated belief state. Here is an abbreviated example interaction:

```
Please enter a sentence: if John doesn't love Mary then Mary doesn't love John
Here is my belief state:
  Option 1: love' john' mary'
  Option 2: NOT love' mary' john'
Please enter a sentence: John doesn't love Mary
Here is my belief state:
```

```
Option 1: NOT love’ mary’ john’ ;  NOT love’ john’ mary’
```

5 Semantics Construction

The key observation of this paper is that semantics construction in Montagovian settings can be formalized in OMDoc/MMT as a view from the language theory of a fragment $\mathcal{F}$ to the corresponding domain theory. GLF generates the language theory in $\mathcal{F}$ from the abstract grammar in $\mathcal{F}$ and translates ASTs to MMT, so that we can execute the semantics construction step by MMT functionality: view application and simplification.

Let us turn back to our example to fortify our intuition. We need to define a suitable target logic and a domain theory. For this small example we don’t need a powerful logic: it suffices to extend the propositional logic defined in Listing 2 by a type constant for individuals (Listing 5). In this logic, we can define the domain theory (also Listing 5), which simply consists of constants such as `joan_DT` or `love_DT`. As notation, we introduce `joan′` etc., following the convention in NL semantics that the meaning of “*Joan*” is `joan′`.

```
theory LogicSyntax : ur:?LF =
    include ?PropLogicSyntax ‖
    individual : type | # ι ‖
⟧
```

```
theory LifeDT : ?LogicSyntax =
    joan_DT : ι           | # joan’ ‖
    mary_DT : ι           | # mary’ ‖
    run_DT  : ι → o       | # run’ ‖
    love_DT : ι → ι → o | # love’ ‖
⟧
```

Listing 5: Logic and domain theory for the `Life` grammar.

Now that we have a target logic and a domain theory, we can define the semantics construction as a modular view in MMT. Listing 6 represents it in two stages according to the grammar/lexicon partition of the source theory: The `LifeGrammarSemantics` view maps all constants from `LifeGrammar` to objects in `LogicSyntax`. The type constants `Stmt`, `Person` and `Action` are mapped to propositions, individuals and unary predicates respectively. `act` is mapped to a simple function application (with the argument order reversed) and `and` to the conjunction of `LogicSyntax` (recall that `[a,b]a∧b` stands for $\lambda a.\lambda b.a \wedge b$). Since `LifeLex` includes `LifeGrammar`, all included constants must be mapped somewhere as well. We can do this by simply including the `LifeGrammarSemantics` view described above. In this small example, most new constants introduced in `LifeLex` have a corresponding element in the domain theory. The only exception is `loveOneself`, which can be described in terms of `love′`. Figure 6 provides an overview of the theories and views we just defined. MMT verifies the completeness of the semantics construction (i.e. that every element of the grammar gets mapped to something) by checking the totality of the views.

```
view LifeGrammarSemantics :
        ?LifeGrammar -> ?LogicSyntax =
    Stmt   = o ‖
    Person = ι ‖
    Action = ι → o ‖

    act = [pers, action] action pers ‖
    and = [a,b] a ∧ b ‖
⟧
```

```
view LifeLexSemantics :
        ?LifeLex -> ?LifeDT =
    include ?LifeGrammarSemantics ‖
    joan        = joan’ ‖
    mary        = mary’ ‖
    run         = run’ ‖
    loveOneself = [x] love’ x x ‖
    love        = love’ ‖
⟧
```

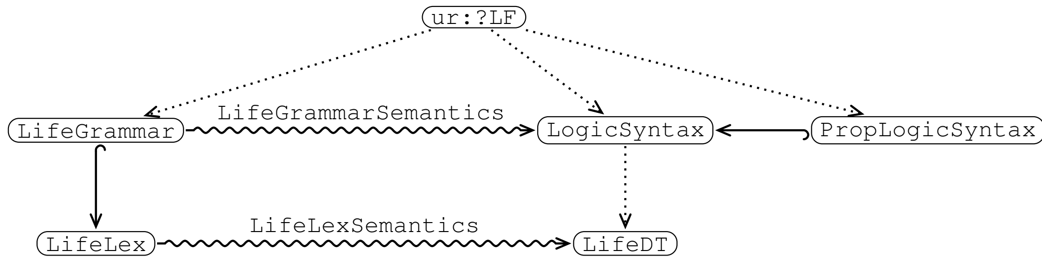

Figure 6: Semantics construction for the `Life` example using MMT's views.

Listing 6: Semantics construction for the `Life` example using views into the domain theory.

Now we have everything we need to do the semantics construction for natural language utterances of our small `Life` fragment of the English language. For example, we can parse the sentence "*Joan loves herself*" with GF, which results in the AST `act joan loveOneself`. Semantics construction just applies the view `LifeLexSemantics` from Listing 6 to the OMDoc/MMT term `act joan loveOneself`. MMT computes `([pers,action]action pers)joan′ [x]love′ x x` and simplifies it to `love′ joan′ joan′`.

We end this section with a remark on truth conditions: In MMT the truth conditions of a statement can be represented as an MMT theory. For example, the truth conditions of "*Mary runs and Joan runs*" (or `run′ mary′ ∧ run′ joan′`) could be represented as a OMDoc/MMT theory with the axioms `a1 : ⊢ run′ mary′` and `a2 : ⊢ run′ joan′`. Note that these axioms directly induce a Herbrand model, which is useful for many NLU systems. It then remains to show that the statement indeed follows from the truth conditions, which we can verify by giving a natural deduction proof – i.e. a term in the theory `PropLogicND` on the right of Listing 2 – which can be verified by the MMT type-checker.

6 Semantic Analysis

At the end of the previous section we have described how truth conditions can be represented as theories containing some axioms that form a Herbrand model. For an NLU system, these Herbrand models should of course be generated automatically. This is possible with tableau calculi which provide automated theorem proving and (Herbrand) model generation.

Following [11] let us consider the following mini-discourse: "*Mary is married to John. Her husband is not in town.*" After the semantic analysis it should be clear that John is not in town (indeed this is one of the truth conditions). The result of the semantics construction could be the following expression in sorted first-order logic: $married'(mary', john') \wedge \exists X_{\mathbb{M}}, Y_{\mathbb{F}}.(husband'(X,Y) \wedge \neg inTown'(X))$ where $\mathbb{M}$ is a sort for male people and $\mathbb{F}$ a sort for female people. These sorts are useful, because they correspond to English pronouns making it clear that e.g. "*her*" cannot refer to "*John*". To infer that "*her husband*" is "*John*", we need world knowledge that describes married men as husbands and imposes monogamy: $\forall X_{\mathbb{F}}, Y_{\mathbb{M}}.married'(X,Y) \Rightarrow husband'(Y,X) \wedge (\forall Z_{\mathbb{M}}.husband'(Z,X) \Rightarrow Z = Y)$. Model generation with a tableau calculus would then result in the following facts: $married'(mary', john')$, $husband'(john', mary')$, $\neg inTown'(john')$ along with various negative facts in different branches (e.g. $\neg married'(mary', mary')$).

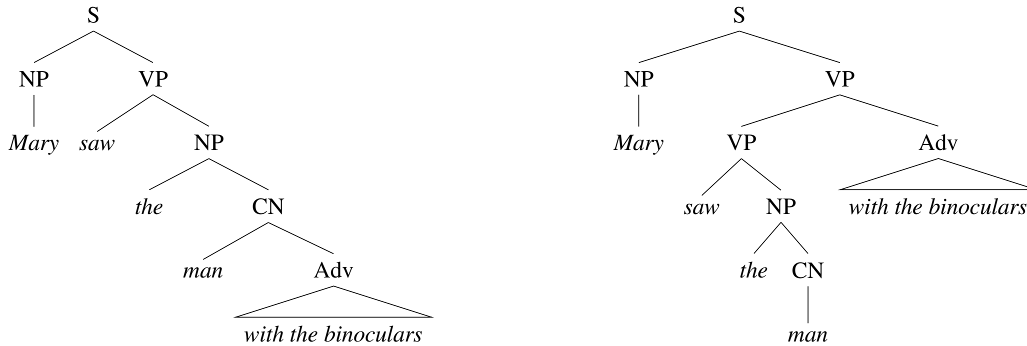

Figure 7: Two different (simplified) parse trees for the same sentence.

6.1 Handling Natural Language Ambiguity

Natural language is very prone to ambiguity. One cause of ambiguity is **lexical ambiguity**: one word can have several meanings (or, the other way around: different words can be spelt the same). For example, in the sentence "*Mary works at a bank*", the word "*bank*" could either refer to a financial institution or to a geographical feature (river bank). In GF, this can be handled by providing two different entries in the abstract syntax – `bank_institute` and `bank_river`, which are both linearized to the string "*bank*" in the concrete grammar for English. Parsing "*Mary works at a bank*" then results two different ASTs – one with each meaning of the word "*bank*". Semantic analysis with world knowledge about legal working conditions might then indicate a preference for the "financial institution reading".

Another type of ambiguity is **structural ambiguity**, which means that the sentence structure is ambiguous. For example, in the sentence "*Mary saw the man with the binoculars*", the phrase "*with the binoculars*" could either refer to the verb "*saw*" or to the object "*the person*". GF handles this ambiguity also by creating one AST for each reading (Figure 7).

GLF propagates ambiguity by applying semantics construction to each AST and passing the results on to semantic analysis. The tableau machine from [11], for example, would be given a disjunction of all terms, which creates one branch for each reading. It then saturates the tableau inferentially with the available world knowledge until a resource criterion is met, and continues with all remaining open branches. [11] considers the open branches of a tableau, after the machine has been fed with a discourse sentence-wise, as the **semantic ambiguity** of the discourse – after all, they induce Herbrand models of the discourse. Tableaux machines can be implemented easily on top of the MMT data structures. We have experimented with a simplified tableau machine that can handle the `LogicSyntax` described in Listing 5. An example output is shown at the end of Section 4.4.

Finally, note that the syntactic ambiguity described above does not necessarily lead to semantic ambiguity. For example, the sentence "*A and B and C*" may be structurally ambiguous and result in the terms $(\llbracket A \rrbracket \wedge \llbracket B \rrbracket) \wedge \llbracket C \rrbracket$ and $\llbracket A \rrbracket \wedge (\llbracket B \rrbracket \wedge \llbracket C \rrbracket)$. However, the terms are equal given the associativity of $\wedge$, i.e. there is no semantic ambiguity.

7 Examples

The running example based on the `Life` fragment was very trivial and does not really justify the efforts of combining MMT and GF. Therefore, we will showcase two more fragments that have a more interesting target logic and semantics construction. In Section 7.1 we will show a Montague-style variation of the

```
abstract Quantified = {
  cat
    NP; VP; V2; S;
  fun
    applyObject  : V2 -> NP -> VP ;
    makeSentence : NP -> VP -> S ;
    and_NP       : NP -> NP -> NP ;
    everyone     : NP ;
    someone      : NP ;
}
```

```
concrete QuantifiedEng of Quantified =
  open SyntaxEng,ParadigmsEng,DictEng in {
  lincat  -- map to RGL categories
    NP = NP; VP = VP; V2 = V2; S = S;
  lin
    applyObject v2 np = mkVP v2 np ;
    makeSentence n v = mkS (mkCl n v) ;
    and_NP a b = mkNP and_Conj a b ;
    everyone = everyone_NP ;
    someone = someone_NP ;
}
```

Listing 7: Variation of the `Life` grammar that supports quantifiers (“*everyone*”, “*someone*”).

`Life` fragment which contains quantifiers. In Section 7.2, we will use modal logic to model propositional attitudes.

7.1 Montague-Style Quantifiers

Noun phrases are natural-language phrases that can serve as the subject or object of a verb. In the `Life` grammar, we represented them with the category `Person`. In this example, we will modify the grammar to support more complex noun phrases. In particular, we will be able to talk about several people (“*John and Mary*”) and to quantify over the set of all people (“*someone*” or “*everyone*”). This makes the semantics construction more challenging. The new abstract syntax is shown in Listing 7 on the left. Note that we switched to the more syntactic (linguistic) category names `NP` (noun phrase), `VP` (verb phrase), `V2` (transitive verb) and `S` (sentence). We can use GF’s resource grammar library [9] – using an **`open`** clause – to create the English concrete syntax as demonstrated in Listing 7 on the right. It provides all the language-specific rules we need, which means that we do not need to deal with the word order and endings ourselves. While it may not be very useful for our small example fragments, it can simplify the implementation of larger fragments significantly.

We will use first-order logic for the target representation. The sentence “*John loves everyone*” could then be represented as $\forall x.love'(john',x)$. `FOL_Syntax` (Listing 8) is an extension of the `LogicSyntax` (defined in Listing 5) with quantifiers using higher-order abstract syntax. An extension for first-order natural deduction calculus is straightforward and well-understood.

```
theory FOL_Syntax : ur:?LF =
    include ?LogicSyntax ‖
    forall : (ι → o) → o | # ∀ 1 ‖
    exists : (ι → o) → o | # ∃ 1 | = [p] ¬ (∀ [x] (¬ p x)) ‖
⟧
```

Listing 8: First-order logic as an extension of `LogicSyntax`.

The tricky part is now the semantics construction. Consider the sentence “*John and Mary run*”, which should result in the expression $run'(john') \wedge run'(mary')$. Since the semantics construction has to be compositional, we need to construct this expression from the meaning of “*John and Mary*” and

the meaning of "*run*". The standard solution in a Montagovian framework [15] is **type raising**[2]: by defining the meaning of "*John and Mary*" as $\lambda p.p(john') \wedge p(mary')$, we can compositionally define the meaning of "*John and Mary run*" as $(\lambda p.p(john') \wedge p(mary'))run'$, which can be β-reduced to the desired expression $run'(john') \wedge run'(mary')$. This means that the type of noun phrases is now $(\iota \to o) \to o$. Another way to look at this is that the meaning of a noun phrase is the set of its properties. This approach results in mappings for the semantics construction like these:

```
mary = [p : ι → o] p mary' ∥
everyone = [p : ι → o] ∀ ([x : ι] p x) ∥
and_NP = [np1, np2] [p : ι → o] (np1 p) ∧ (np2 p) ∥
```

In order to handle transitive verbs like "*love*", it is necessary to apply type raising to the verbs as well: by providing verb phrases with noun phrases as arguments rather than individuals, their type becomes $((\iota \to o) \to o) \to o$ where $(\iota \to o) \to o$ is the type of a noun phrase. Similarly, transitive verbs get two noun phrases as arguments, resulting in the type $((\iota \to o) \to o) \to ((\iota \to o) \to o) \to o$. This allows us to define the semantics of verbs in the following way:

```
run = [np] np run' ∥
loveOneself = [np] np ([x : ι] (love' x) x) ∥
love = [np1, np2] np1 ([x : ι] np2 ([y : ι] love' y x)) ∥
```

Note that `run' :` $\iota \to o$ and `love' :` $\iota \to \iota \to o$ are constants from the domain theory (we can reuse the domain theory of the `Life` example in Listing 5).

It is now straightforward to define the semantics construction for `applyObject` and `makeSentence`:

```
applyObject = [v2, np] v2 np ∥
makeSentence = [np, vp] vp np ∥
```

With all this, we can parse the example sentence "*John and Mary love everyone*" to obtain the AST `makeSentence (and_NP john mary)(applyObject love everyone)`. Applying the semantics construction, we obtain the expression

```
([np,vp]vp np) (([np1,np2][p:ι→o](np1 p)∧(np2 p)) ([p:ι→o]p john') [p:ι→o]p mary')
(([v2,np]v2 np) ([np1,np2]np1 [x:ι]np2 [y:ι]love' y x) [p:ι→o]∀[x:ι]p x)
```

which β-reduces to `∀[x:ι](love' john' x)∧(love' mary' x)` as desired.

7.2 (Multi) Modal Logic

In this example, we will use multi modal logic with the following modalities:

- **deontic** modality, expressing that something is obligatory or permitted
- **epistemic** modality, expressing that someone believes something to be true or possibly true

For an introductory discussion of propositional attitudes and modal logics in natural language semantics we refer the reader to [6]. This text also discusses a plethora of other phenomena and logics, which we could have used as examples. Indeed, all these logics – and their combinations – are a good validation of the necessity of a tool like GLF.

We can describe this logic in MMT by extending the `LogicSyntax` from Listing 5. Listing 9 shows the syntax for multi modal logic in general and for our specific case with deontic and epistemic modalities. As a notation we introduce ⟦m⟧ for the box operator with modality `m` and ⟪m⟫ for the diamond operator with modality `m`.

[2] The reader may be familiar with **continuation-passing style**, which is a similar concept in programming languages.

```
theory MultiModalLogic : ur:?LF =
    include ?LogicSyntax ‖
    modality : type      | # μ ‖
    box : μ → o → o      | # ⟦ 1 ⟧ 2 ‖
    diamond : μ → o → o | # ⟪ 1 ⟫ 2
        | = [mod] [φ] ¬ ⟦ mod ⟧ ¬ φ ‖
⟧
```

```
theory DEModalLogic : ur:?LF =
    include ?MultiModalLogic ‖
    deontic   : μ      | # d ‖
    epistemic : ι → μ | # e 1 ‖
⟧
```

Listing 9: Multi modal logic syntax and an extension for deontic-epistemic modal logic.

This way, we can express the meaning of e.g. "*John is not allowed to run*" as `¬⟪d⟫(run′ john′)` where `d` is the deontic modality. Similarly, we can express the meaning of "*Mary believes that John is happy*" with epistemic modality: `⟦e mary′⟧(happy′ john′)`.

In GF, we can describe "*be allowed to*" and "*have to*" as verb phrase modifiers (`VpModifier`), while "*Mary believes that*" can be described as a sentence modifier (`SModifier`). To handle negations, we can introduce the category `Pol` (for polarity), indicating whether a sentence is negated. Listing 10 shows the abstract syntax in GF. For example, "*John doesn't run*" would be parsed into the AST `makeS neg john run` and "*John has to run*" would result in `makeS pos john (modifyVP pos have_to run)`.

```
abstract Modal = {
  cat
    S ; VP ; Person;
    VpModifier ; SModifier;
    Pol; -- negative/positive polarity
  fun
    pos, neg : Pol;
    makeS : Pol->Person->VP->S;
    modifyVP : Pol->VpModifier->VP->VP;
    modifyS : Pol->SModifier->S->S;
    be_allowed_to : VpModifier;
    have_to : VpModifier;
    believe : Person->SModifier;
}
```

```
view ModalSemantics : ?Modal -> ?ModalDT =
  S = o ‖ VP = ι → o ‖ Person = ι ‖
  VpModifier = o → o ‖ SModifier = o → o ‖
  Pol = o → o ‖

  pos = [φ] φ ‖
  neg = [φ] ¬ φ ‖
  makeS = [pol,pers,vp] pol (vp pers) ‖
  modifyVP = [pol,m,vp] [x] m(pol(vp x)) ‖
  modifyS = [pol,m,s] pol (m s) ‖
  be_allowed_to = [φ] ⟪ d ⟫ φ ‖
  have_to = [φ] ⟦ d ⟧ φ ‖
  believe = [pers] [s] ⟦ e pers ⟧ s ‖
⟧
```

Listing 10: The GF abstract syntax along with the semantics construction.

The semantics construction is now rather straightforward (Listing 10): sentences are propositions, verb phrases are unary predicates and persons are individuals. The modifiers `VpModifier` and `SModifier` have the type $o \to o$, which allows us to define, e.g., `be_allowed_to` as $\lambda\varphi.\langle\!\langle d\rangle\!\rangle\varphi$, where $\langle\!\langle d\rangle\!\rangle$ is the diamond operator with deontic modality. The polarities can also be expressed as functions on propositions: positive polarity is the identity function while negative polarity is negation. These different components are combined with the functions `makeS`, `modifyVP` and `modifyS`, which simply apply the different components to each other. For example, `makeS` is mapped to `[pol,pers,vp] pol (vp pers)`. With this, the semantics construction of the example sentence "*John doesn't run*" would result in the term `¬(run′ john′)`. An example sentence using modalities would be "*John doesn't believe that Mary has to run*", which results in the term `¬⟦(e john′)⟧⟦d⟧(run′ mary′)`.

8 Conclusion and Future Work

We have presented GLF, a simple framework for experimenting with natural-language semantics and developing Montagovian fragments. It is – to our knowledge – the first framework that allows to implement the entire pipeline from language parsing to the semantic analysis in a declarative way. The GLF system reduces the creation of a symbolic NLU application to the following three steps:

1. Write a GF grammar (abstract syntax + possibly multiple concrete syntaxes), possibly re-using large pieces of the GF resource grammer.
2. Define the target logic and domain theory, and define the semantics construction view in MMT using the GLF-generated language theory and a view stub.
3. Implement some form of semantic analysis and application logic.

We have tested this workflow with several examples, including the ones described in this paper. The most complex example was the record λ-calculus for complex noun meanings from [17]. We have also used GLF as an educational tool in a course on logic-based natural language semantics at FAU Erlangen-Nürnberg.

To increase GLF's usefulness, we are planning to work on the following points:

1. Providing a unified interface for the grammar and semantics development. Concretely, we started work on a Jupyter kernel for GLF, extending our GF Kernel (see [8]). Our goal is that the GF grammar, the target logic, and the semantics construction view can all be implemented, tested, and documented in a single and coherent notebook.
2. Enabling GLF to check that all λ expressions introduced by type raising for the sake of compositionality get β-reduced away in semantics construction to ensure that the resulting expression is in the target logic.
3. Extending MMT to calling an off-the-shelf theorem prover in the semantic analysis phase. This is particularly useful for pruning out readings in the style of [2] and thus reducing overall ambiguity.
4. Looking into the reverse pipeline (logic-to-language translation), which could be a nice feature for displaying e.g. inferred results.
5. Adding support for regression testing and automated evaluation against a gold standard to facilitate realistic, corpus-driven development of fragments and ontologies.

A small GLF distribution along with the examples in this paper can be found at [10].

All in all, we hope that the GLF system constitutes a tool that facilitates NLU development and experiments, and can thus kick-start a "logic revival" in computational linguistics. Like the systems it combines, it may also act as a bridge between the respective communities.

References

[1] Patrick Blackburn & Johan Bos (2005): *Representation and Inference for Natural Language. A First Course in Computational Semantics*. CSLI.

[2] Patrick Blackburn, Johan Bos, Michael Kohlhase & Hans de Nivelle (2001): *Inference and Computational Semantics*. In Harry Bunt, Leen Kievit, Reinhard Muskens & Margriet Verlinden, editors: *Computing Meaning (Volume 2)*, Kluwer Academic Publishers, pp. 11–28, doi:10.1007/978-94-010-0572-2_2.

[3] Mihai Codescu, Fulya Horozal, Michael Kohlhase, Till Mossakowski & Florian Rabe (2011): *Project Abstract: Logic Atlas and Integrator (LATIN)*. In James Davenport, William Farmer, Florian Rabe & Josef Urban, editors: *Intelligent Computer Mathematics*, *LNAI* 6824, Springer Verlag, pp. 289–291, doi:10.1007/978-3-642-22673-1_24. Available at https://kwarc.info/people/frabe/Research/CHKMR_latinabs_11.pdf.

[4] Donald Davidson (1967): *Truth and Meaning*. *Synthese* 17, doi:10.1007/bf00485035.

[5] Jan van Eijck & Christina Unger (2010): *Computational Semantics with Functional Programming*. Cambridge University Press, doi:10.1017/cbo9780511778377.

[6] L. T. F. Gamut (1991): *Logic, Language and Meaning, Volume II, Intensional Logic and Logical Grammar*. 2, University of Chicago Press, Chicago.

[7] *GF - Grammatical Framework*. http://www.grammaticalframework.org.

[8] *GF Kernel*. https://github.com/kwarc/gf_kernel.

[9] *GF Resource Grammar Library: Synopsis*. https://www.grammaticalframework.org/lib/doc/synopsis/index.html.

[10] *GLF Demo Repository*. https://gl.kwarc.info/COMMA/glf-demo-lfmtp2019.

[11] Michael Kohlhase & Alexander Koller (2000): *Towards A Tableaux Machine for Language Understanding*. In Johan Bos & Michael Kohlhase, editors: *Proceedings of Inference in Computational Semantics ICoS-2*, Computational Linguistics, Saarland University, pp. 57–88.

[12] *The LATIN Logic Atlas*. https://gl.mathhub.info/MMT/LATIN.

[13] *MMT – Language and System for the Uniform Representation of Knowledge*. project web site at https://uniformal.github.io/. Available at https://uniformal.github.io/.

[14] R. Montague (1970): *English as a Formal Language*, chapter Linguaggi nella Societa e nella Tecnica, B. Visentini et al eds, pp. 189–224. Edizioni di Communita, Milan. Reprinted in [23], 188–221.

[15] Richard Montague (1974): *The Proper Treatment of Quantification in Ordinary English*. In R. Thomason, editor: *Formal Philosophy. Selected Papers*, Yale University Press, New Haven.

[16] Dennis Müller & Florian Rabe (2019): *Rapid Prototyping Formal Systems in MMT: Case Studies*. Available at https://kwarc.info/people/frabe/Research/MR_prototyping_19.pdf.

[17] Manfred Pinkal & Michael Kohlhase (2000): *Feature Logic for Dotted Types: A Formalism for Complex Word Meanings*. In: *Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics*, Hongkong, pp. 521–528, doi:10.3115/1075218.1075284. Available at http://kwarc.info/kohlhase/papers/acl00.pdf.

[18] Florian Rabe (2013): *The MMT API: A Generic MKM System*. In Jacques Carette, David Aspinall, Christoph Lange, Petr Sojka & Wolfgang Windsteiger, editors: *Intelligent Computer Mathematics*, *Lecture Notes in Computer Science* 7961, Springer, pp. 339–343, doi:10.1007/978-3-642-39320-4_25.

[19] Florian Rabe & Michael Kohlhase (2013): *A Scalable Module System*. *Information & Computation* 0(230), pp. 1–54, doi:10.1016/j.ic.2013.06.001. Available at http://kwarc.info/frabe/Research/mmt.pdf.

[20] Aarne Ranta: *Grammatical Framework - Formalizing the Grammars of the World*. http://www.grammaticalframework.org/~aarne/gf-google-2016.pdf.

[21] Aarne Ranta (2004): *Grammatical Framework — A Type-Theoretical Grammar Formalism*. *Journal of Functional Programming* 14(2), pp. 145–189, doi:10.1017/S0956796803004738.

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

[23] R. Thomason, editor (1974): *Formal Philosophy: selected Papers of Richard Montague*. Yale University Press, New Haven, CT.