

The Glue Semantics Workbench: A modular toolkit for exploring Linear Logic and Glue Semantics

Moritz Meßmer
Across Systems GmbH

Mark-Matthias Zymla
University of Konstanz

Proceedings of the LFG'18 Conference

University of Vienna

Miriam Butt, Tracy Holloway King (Editors)


2018

CSLI Publications

pages 268–282

<http://csli-publications.stanford.edu/LFG/2018>

Keywords: Glue semantics, linear logic, implementation, Java, workbench

Meßmer, Moritz, & Zymla, Mark-Matthias. 2018. The Glue Semantics Workbench: A modular toolkit for exploring Linear Logic and Glue Semantics. In Butt, Miriam, & King, Tracy Holloway (Eds.), *Proceedings of the LFG'18 Conference, University of Vienna*, 268–282. Stanford, CA: CSLI Publications. 

Abstract

In this paper we present an easy to use, modular Glue semantic prover building on the work by Crouch & van Genabith (2000) and implemented in Java. We take inspiration from a Glue semantics parser written in Prolog as well as other existing tools such as the NLTK Glue semantics system. The architecture of our semantic parser allows us to explore the computational viability of linear logic as a mechanism for modeling compositional semantics within LFG. Furthermore, it allows researchers interested in linear logic (for computational linguistics) to research its usefulness, when applied to different syntactic models and various formal semantic frameworks. The goal of this resource is to provide an accessible entry point for both beginners and adepts in computational semantics. It thus also has prospective uses as a teaching tool for computational semantics and linear logic.

1 Introduction

In this paper we present an easy to use, modular Glue semantic prover and parser called *Glue semantics workbench* building on the work by Crouch & van Genabith (2000).¹ Thereby, we revive a Glue semantics parser written in Prolog, since this first implementation is not readily accessible anymore, due to the commercialization of the programming language.² Our goal is to translate the system into a more modern implementation within the Java programming language.

Glue semantics is the formalism of choice for formal semantics within the LFG framework (Dalrymple, 2001), but has since attracted interest from different venues, e.g. Asudeh & Crouch (2002); Gotham (2018); Garrette & Klein (2009); Gotham & Haug (to appear). The composition process in this framework is based on linear logic which guides semantic composition comparable to types in Montague semantics (Montague, 1970). Linear logic lends itself well to modelling compositionality due to its resource-sensitivity (Dalrymple, 2001).

The Glue prover presented in this paper is a rejuvenation of existing theoretical and practical approaches to modeling Glue semantics. More specifically, the sys-

[†]We thank the participants of the 2018 LFG conference for valuable feedback. Many thanks also for the very helpful comments by the internal and external reviewers. We are particularly grateful to Richard Crouch and Valeria de Paiva for their support in developing the Glue semantics workbench. Furthermore, we thank the researchers at the CSLI, Stanford for their assistance.

¹The Glue semantics workbench is publicly available on <https://github.com/mmessmer/GlueSemWorkbench>. It is free software and distributed under the conditions of the GNU General Public License.

²The Prolog Glue prover has been designed as part of the Xerox Linguistic Environment (XLE). In its older iterations, this system relied on SICStus Prolog, a commercial strain of the Prolog family of programming languages. More recent iterations of XLE do not rely on SICStus Prolog anymore, however, this is at the cost of certain features of XLE, in particular the transfer system (Crouch et al., 2017) and other systems that rely on a Prolog interface, such as AKR semantics (Bobrow et al., 2007) and the mentioned Prolog Glue prover.

tem is based on a chart parser devised by Hepple (1996). The main element taken from Hepple’s work is the compilation process, which is used to deconstruct Glue premises from higher-order premises into first-order premises. This simplifies the combinatory process of Glue semantic resources. The Glue prover has been further refined with ideas from Gupta & Lamping (1998) that improve efficiency by reducing unnecessary steps in the computation.

Employing these two strategies allows us to present a reasonably efficient algorithm for conducting Glue semantics computations. The modularity of our semantic parser not only allows us to continue the exploration of the computational viability of linear logic as a mechanism for modeling compositional semantics within LFG but it also allows us to explore the interoperability of linear logic with respect to other syntactic theories as well as different semantic formalisms. For this purpose, we illustrate the use of the Glue prover in interaction with both LFG and UD (universal dependency) grammars on the syntactic side and its interaction with Montague-style lambda calculus on the semantic side. To this end, we have implemented a light-weight Montague-style semantics that can be combined with the linear logic prover; however, other semantic formalisms can also be plugged-in without the need to change the overall system.

The paper is structured as follows: In Section 2 we briefly introduce the formalities of linear logic and how it can be used in the domain of compositional semantics. Readers who are familiar with the subject and are interested in the concrete implementation may jump directly to Section 3. The architecture of the system is described in Section 4, which involves the assembly of the linear logic prover with syntactic parsers and formal semantic models. This is of particular interest for readers who intend to work with the Glue semantics workbench. Section 5 concludes.

2 Glue Semantics

Glue semantics is a framework that continues to attract interest not only in the LFG community. Its elegance in terms of aligning the structure of logic proofs with the structure of semantic meaning compositions through the Curry-Howard-Isomorphism has motivated researchers to adapt it for other frameworks such as HPSG (Asudeh & Crouch, 2002), LTAG (Frank & van Genabith, 2001) and Minimalism (Gotham, 2015, 2018). Instead of relying on rules that map the syntactic structure to semantic composition rules, Glue semantics uses a fragment of linear logic to constrain the composition of meaning representations.

In this scenario, a semantic representation is a pair consisting of a linear logic side and a meaning side (in this paper: Montague-style lambda calculus). Thereby, the logic side constrains the possible combination of semantic elements. I.e. the linear logic side of a lexical entry constrains the compositional possibilities of its

meaning side. This is reflected in the Curry-Howard-Isomorphism. The isomorphism describes the correspondence between natural deduction proofs, i.e. the logic side, and computational models like lambda calculus, i.e. the meaning side. It is the foundation for the pairing of logics used in the Glue approach. More concretely, the Curry-Howard-Isomorphism states that lambda abstraction on the meaning side corresponds to \multimap introduction and functional application corresponds to \multimap elimination. This is illustrated in the following figure. On the left side, it is shown how the introduction of a linear implication affects the meaning side: A lambda function is generated. On the right side, the correspondence between a functional application and the combination of a linear implication with its corresponding resource is depicted. Due to this system, Glue semantics formulas can be composed and decomposed on the logic side and the meaning side in concord.

$$\frac{\begin{array}{c} [x : A]^i \\ \vdots \\ f(x) : B \end{array}}{\lambda x.f(x) : A \multimap B} \multimap_{I,i} \qquad \frac{f : A \multimap B \quad a : A}{f(a) : B} \multimap_E$$

Figure 1: Implication introduction and elimination

In Glue semantics proofs, the \multimap_E rule is applied when *combining* two meaning constructors, while the \multimap_I rule is used for introducing assumptions. In linear logic proofs, assumptions are a deduction tool for deriving a proof whose premises are not immediately compatible. For a proof to be valid, all assumptions that have been made during the deduction have to be reintroduced via implication introduction. This follows from the general principle of linear logic, which states that a valid proof needs to consume all available resources. In other words, assumptions are simply treated as additional resources that emerge during the computation.

- (1) **John** $j : g$
Mary $m : h$
loves $\lambda x.\lambda y.loves(x, y) : g \multimap (h \multimap f)$

$$\frac{\frac{\lambda x.\lambda y.loves(x, y) : g \multimap (h \multimap f) \quad j : g}{\lambda y.loves(j, y) : h \multimap f} \quad m : h}{loves(j, m) : f}$$

Figure 2: Derivation of *John loves Mary*.

The propositional implicational fragment of linear logic paired with lambda-calculus is already capable of deriving simple meaning structures as shown in Figure 2. However, as soon as scope-taking expressions enter the stage and potentially

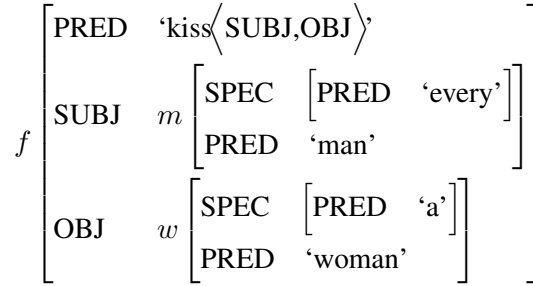


Figure 3: F-structure of *Every man loves a woman*.

introduce ambiguities, this fragment of linear logic does not suffice anymore. A quantifier expression can take different constituents as its scope, therefore the labels on the linear logic side cannot be fixed to constants. Instead, the scope of a quantifier is encoded with linear logic variables. These variables are introduced by a universal quantifier which binds a variable that is instantiated to a linear logic constant in the derivation process. Consequently, we move from a propositional linear logic fragment to a higher-order predicate logic fragment of linear logic involving universal quantification over f-structure labels.³ Consider the following sentence and its f-structure.

(2) Every man kisses a woman.

(3) $m \left[\begin{array}{l} \text{SPEC} \quad \left[\text{PRED} \quad \text{'every'} \right] \\ \text{PRED} \quad \text{'man'} \end{array} \right] \quad m_\sigma \left[\begin{array}{l} \text{VAR} \quad v \left[\begin{array}{l} \left[\right] \end{array} \right] \\ \text{RESTR} \quad r \left[\begin{array}{l} \left[\right] \end{array} \right] \end{array} \right]$

(4) **every**: $[((\text{SPEC } \uparrow)_\sigma \text{ VAR}) \multimap ((\text{SPEC } \uparrow)_\sigma \text{ RESTR})]$
 $\multimap \forall X. [((\text{SPEC } \uparrow)_\sigma \multimap X) \multimap X]$
man: $(\uparrow_\sigma \text{ VAR}) \multimap (\uparrow_\sigma \text{ RESTR})$

(4) shows the Glue side of the meaning constructor of the quantifying expression *every*. The scope part of the quantifier universally quantifies over variables of type t . This can be seen as a quantifier which expresses that any f-structure can be inserted as the scope of *every*. Deriving the two scope constellations now requires us to restructure the proof with the tools of natural deduction: eliminating implications by combining resources and introducing implications by making (temporary) assumptions.

Given the following meaning constructors with instantiated labels, we can make two logically equivalent derivations, which result in two different readings. For reasons of brevity, the quantified NPs are already combined with their restrictors.

³This is a fairly standard system in the Glue literature. However, it has been argued that a first-order linear logic fragment is sufficient to model natural language semantics. For discussion on this topic, see Kokkonidis (2008). Thanks to the external reviewer for bringing this to our attention.

every man	$\lambda P.\forall x[\text{man}(x) \rightarrow P(x)]$:	$(m_\sigma \multimap f_\sigma) \multimap f_\sigma$
a woman	$\lambda Q.\exists y[\text{woman}(y) \wedge Q(y)]$:	$(w_\sigma \multimap f_\sigma) \multimap f_\sigma$
kiss	$\lambda x.\lambda y.\text{kiss}(x,y)$:	$m_\sigma \multimap (w_\sigma \multimap f_\sigma)$

In their original form given by the lexical entries, the quantifiers cannot combine with the verb. One (surface scope) or both (inverse scope) lambda slots of the verb have to be temporally saturated with assumptions. Assumptions need to be reintroduced into the computation later in the proof in accordance with Figure 1. As shown in Figure 4a and 4b, the assumed resources are marked with square brackets and indices. They introduce a temporary unbound variable that is bound later via lambda abstraction when the linear implication is reintroduced via \multimap introduction.

The Glue semantics fragment introduced above is the foundation for the Glue semantics workbench. In the next section we show how this system is translated into a computationally viable Glue semantics parser.

3 The linear logic prover algorithm

While this system of using assumptions in the deduction process to temporarily saturate lambda binders on the meaning side is very elegant for dealing with scope ambiguities since it is independent of syntactic assumptions about the phenomenon, it is a very complex system for any automatic proving algorithm. In practice, even the implicational fragment of linear logic used here is NP-complete and may be computationally intractable once the formulas reach a certain complexity. An algorithm to circumvent some of the computational complexity was proposed by Hepple (1996) and is used for our proving algorithm as well. Additionally, our proving algorithm uses a system, based on Gupta & Lamping (1998), that distinguishes between symmetric modifier resources and asymmetric skeleton resources to further increase its efficiency. The algorithm is based on three principles to transform linear logic proofs into computationally tractable algorithms: (I) indexation of premises (II) compilation of nested implications (III) separation of modifier-type premises. In the following, the three principles will be explained and the algorithm for calculating proofs is outlined.

3.1 Basic first-order chart prover

As has been pointed out by Crouch & van Genabith (2000), Glue semantics strongly resembles categorial grammar systems and a Glue semantics proof resembles the principles of syntactic chart parsing techniques. In both systems individual items (premises in linear logic and words or constituents for chart parsers) are taken from the agenda and combined to obtain intermediate results which in turn are combined again until the agenda is empty. The proving system by Hepple (1996) makes use of this resemblance by adapting the chart parsing technique for linear logic proofs in the form of **indexation**. The Glue premises are each assigned a set of indices in Hepple's prover algorithm. Initial premises are added to an

$$\begin{array}{c}
\frac{[X : m_\sigma]^1 \quad \lambda x.\lambda y.\text{kiss}(x, y) : m_\sigma \multimap (w_\sigma \multimap f_\sigma)}{\lambda y.\text{kiss}(X, y) : w_\sigma \multimap f_\sigma \quad \lambda Q.\exists y[\text{woman}(y) \wedge Q(y)] : (w_\sigma \multimap f_\sigma) \multimap f_\sigma} \multimap E \\
\frac{\exists y[\text{woman}(y) \wedge \text{kiss}(X, y)] : f_\sigma}{\lambda P.\forall x[\text{man}(x) \rightarrow P(x)] : (m_\sigma \multimap f_\sigma) \multimap f_\sigma \quad \lambda x.\exists y[\text{woman}(y) \wedge \text{kiss}(x, y)] : m_\sigma \multimap f_\sigma} \multimap I,1 \\
\frac{\forall x[\text{man}(x) \rightarrow \exists y[\text{woman}(y) \wedge \text{kiss}(x, y)]] : f_\sigma}{\lambda P.\forall x[\text{man}(x) \rightarrow \exists y[\text{woman}(y) \wedge \text{kiss}(x, y)]] : f_\sigma} \multimap E
\end{array}$$

(a) Glue proof: *Every man kisses a woman surface scope*

$$\begin{array}{c}
\frac{[X : m_\sigma]^1 \quad \lambda x.\lambda y.\text{kiss}(x, y) : m_\sigma \multimap (w_\sigma \multimap f_\sigma)}{\lambda y.\text{kiss}(X, y) : w_\sigma \multimap f_\sigma \quad [Y : w_\sigma]^2} \multimap E \\
\frac{\text{kiss}(X, Y) : f_\sigma}{\lambda P.\forall x[\text{man}(x) \rightarrow P(x)] : (m_\sigma \multimap f_\sigma) \multimap f_\sigma \quad \lambda x.\text{kiss}(x, Y) : m_\sigma \multimap f_\sigma} \multimap I,1 \\
\frac{\forall x[\text{man}(x) \rightarrow \text{kiss}(x, Y)] : f_\sigma}{\lambda y.\forall x[\text{man}(x) \rightarrow \text{kiss}(x, y)] : w_\sigma \multimap f_\sigma \quad \lambda Q.\exists y[\text{woman}(y) \wedge Q(y)] : (w_\sigma \multimap f_\sigma) \multimap f_\sigma} \multimap I,2 \\
\frac{\exists y[\text{woman}(y) \wedge \forall x[\text{man}(x) \rightarrow \text{kiss}(x, y)]] : f_\sigma}{\lambda P.\forall x[\text{man}(x) \rightarrow \exists y[\text{woman}(y) \wedge \text{kiss}(x, y)]] : f_\sigma} \multimap E
\end{array}$$

(b) Glue proof: *Every man kisses a woman inverse scope*

Figure 4: Quantifier ambiguity in Glue

agenda and each is assigned a single index. For each combination of two premises, the index sets of the two premises are also combined and the joined index set is assigned to the newly created premise. If a premise $A : [0]$, for example, is combined with a premise $A \multimap B : [1]$ then the newly created premise is $B : [0, 1]$. In order to ensure that each possible combination is checked, the algorithm works with an agenda, containing all “fresh” premises and a database containing all the premises which have already been taken from the agenda. For each premise taken off the agenda, the algorithm checks combinatory possibilities with all premises in the database. If the current premise combines with one from the database, the newly created premise is also added to the agenda. After all checks have been made, the current premise is moved from the agenda to the database and the algorithm proceeds with the next premise from the agenda. Unnecessary or invalid steps in the computation can thus be avoided by requiring that when combining two premises, their index sets must be disjoint.

3.2 Compilation of higher-order premises

As mentioned above, this simple chart prover algorithm reaches its limits as soon as the proof contains higher-order premises. For our algorithm, higher-order linear logic formulas are nested implications where the antecedent is itself an implication.⁴ In a natural deduction-style proof, these formulas require making assumptions and discharging them at some points of the proof. It would take an algorithm a great deal of computational effort to determine when it is necessary to make an assumption and when to discharge it. Therefore Hepple’s prover implements a computationally feasible solution to that problem: every higher-order premise is compiled by separating its antecedent as an additional premise and adding it to the agenda, marked as an assumption.⁵ The premise from which the auxiliary premise is taken is marked with a dependency on the respective auxiliary premise. This step is repeated until only first-order premises are left. In the notation of our proving algorithm, auxiliary premises are marked with $\{ \}$ and their dependencies (*discharges*) are marked on the original premise with $[]$ ⁶. More concretely, these references are implemented in our code such that each premise has two lists associated with it, one for assumptions and one for discharges.

$$(5) \quad (a \multimap b) \multimap c [0] \Rightarrow_{\text{compile}} b[a] \multimap c [0];$$

⁴Note that this does not correspond to higher-order linear predicate logic formulas as discussed in the previous section.

⁵In Hepple (1996) the term *assumption* is used to describe these auxiliary premises. In this paper, both of these terms refer specifically to premises that have been generated via the compilation process; i.e. premises that have been cut off from a higher-order premise.

⁶In the original algorithm by Hepple, references to auxiliary premises are made via their indices. The premise from which an assumption is compiled out will from now on be called the assumption’s *host* premise. For our system we decided to add references to the Glue resources themselves, as that makes the proofs more readable and is easily implemented due to Java’s object-oriented programming paradigm.

$\{a\} [1]$

By adding a reference to the extracted assumption to its host premise, the algorithm prevents invalid proofs where the assumption might be used, without later discharging it. This restriction is achieved by adding two rules to the proving algorithms. First, a premise P containing a set of discharges δ may only combine with a premise whose list of assumptions α is a subset of δ . In that case, all matching assumption and discharge pairs are removed from the newly created premise. Second, if two premises contain (or are themselves) assumptions, their lists of assumptions are joined. With these modifications, a proof is now only valid if the resulting premise, besides containing all initial indices, does not have any assumptions or discharges associated with it.

$$(6) \quad \frac{b\{a\}[1, 2] \quad b[a] \multimap C[3]}{c[1, 2, 3]}$$

So far, only the linear logic side has been dealt with, but of course the semantic side of a premise is affected by the compilation process as well. As mentioned before, operations on linear logic proofs and operations on lambda-expressions on the semantic side of premises are aligned via the Curry-Howard isomorphism. Implication elimination on the Glue side of a premise can therefore be seen as a functional application operation on the semantic side, while implication introduction amounts to functional abstraction. This becomes relevant when proofs contain assumptions. Auxiliary premises that are introduced into a Glue semantics proof carry unbound variables on the semantic side. When an assumption is combined with another premise, this variable is then inserted into the semantic representation of that other premise via functional application. Later in the proof, when the assumption is discharged, the assumption variable is bound by a lambda term again.

This elegant system of temporarily saturating λ -slots in semantic computation is one of the reasons why Glue semantics interests formal semanticists. It allows a system of formally resolving ambiguities without having to rely on additional abstract systems such as a logical form or Cooper-storage (Cooper, 1983). The semantic aspect of Glue proofs is covered by Hepple (1996) as well. In his algorithm, auxiliary premises created in the compilation process carry temporary variables as well, but the re-binding of the variables is done via an additional lambda binder that is functionally applied to the semantic representation of the host premise. As soon as the premise containing the unbound assumption variable combines with its host the lambda term binds the variable. The lambda term binding the variable can then be applied to the original meaning representation. In regular lambda calculus such an "accidental" binding an unbound variable by adding a lambda binder is not a legal operation. However, Hepple's algorithm uses this operation in a deliberate and controlled manner. By using different variables for each newly created formula during the compilation process, the algorithm therefore ensures that free variables are accidentally bound by the wrong lambda binder. In our prover algo-

rithm the creation of new variables is handled centrally for all formulas in a proof. This allows full control of which variables are used and inserted into formulas. The compilation of glue formulas with semantic representations is illustrated in Figure 5.

$$(7) \quad \frac{\frac{H[g_2] \multimap H : \lambda u. \lambda P. \forall x[\text{person}(x) \wedge P(x)](\lambda v. u) \quad \frac{g_1 \multimap f : \lambda y. \text{sleep}(y) \quad \{g_2\} : v}{f\{g_2\} : \text{sleep}(v)} \text{[H/f]}}{f : \lambda P. \forall x[\text{person}(x) \wedge P(x)](\lambda v. \text{sleep}(v))} \beta\text{-conversion}}{f : \forall x[\text{person}(x) \wedge \text{sleep}(x)]} \text{[H/f]}}$$

Figure 5: Every person sleeps. – Hepple style

3.3 Treating modifier premises

While this algorithm is already capable of handling linear logic formula of the implicational fragment, it is still rather inefficient if a proof contains *modifier*-type premises. Modifiers as defined by Gupta & Lamping (1998) are premises whose linear logic side has a certain pattern. This pattern can be seen if occurrences of linear logic atoms, or in the case of Glue semantics, type labels, are assigned a polarity depending on whether they occur in the antecedent or the consequent of a linear implication: the consequent has the same polarity as the whole implication and the antecedent has the opposite polarity. Assuming that linear logic formulas as a whole always have positive polarity, the polarity of each atom can thus be assigned:

$$(8) \quad ((f_+ \multimap g_-)_- \multimap (f_- \multimap g_+)_+)_+ \\ (9) \quad ((v_+ \multimap r_-)_- \multimap ((g_+ \multimap X_-)_- \multimap X_+)_+)_+$$

The formula in (8) is considered to be a modifier type because all positive occurrences of Glue labels are matched up with negative occurrences. In Glue semantics, lexical entries for adjuncts are usually modifier types because they modify the meaning of an f-structure node without altering its type. Other lexical entries are mostly purely skeleton-type because they only consist of singular positive or negative occurrences of each label.

There are some cases where skeleton and modifier-type occurrences are mixed inside a formula. Quantifiers, like the one in (9), for example, are mostly skeleton-types except for the matching positive and negative occurrence of the Glue variable (X in the example above) which denotes the scope. These mixed-type quantifiers are also compiled. In general, all premises that are not pure modifiers are compiled until they are either pure skeletons or have the form $a \multimap M$, where a is an atomic type and M is a pure modifier type. The latter sort of premise is treated like a skeleton during the deduction process (until the atomic antecedent is consumed and the premise becomes a pure modifier). Such cases occur for certain kinds of modifiers. One such example would be recursive modification as described in Dalrymple (2001). In order to obtain the correct meaning for the phrase *apparently*

Swedish man, Dalrymple (2001) proposes an internally structured meaning of adjectival and adverbial modifiers. This in principle means that the meaning of these modifiers is deconstructed into two separate modifiers. One constructor contributes the lexical information of the respective modifier and the second constructor contributes the structure for semantic composition, i.e. it guarantees that *apparently* modifies the complete noun phrase *swedish man* and not just *man*. This leads to the following lexical entries:

Swedish1	$\lambda x, Swedish(x)$	$: (g_v \multimap g_\sigma)$
Swedish2	$\lambda Q. \lambda P. \lambda x. Q(x) \wedge P(x)$	$: (g_v \multimap g_\sigma) \multimap ((v \multimap r) \multimap (v \multimap r))$
apparently1	$\lambda P, apparently(P)$	$: (h_v \multimap h_\sigma)$
apparently2	$\lambda Q. \lambda R. \lambda x. Q(R(x))$	$: (h_v \multimap h_\sigma) \multimap ((g_v \multimap g_\sigma) \multimap (g_v \multimap g_\sigma))$
man	$\lambda y. man(y)$	$: (v \multimap r)$
ap. sw. man	$\lambda x. apparently(swedish(x) \wedge man(x))$	$: (v \multimap r)$

In the above case the meaning constructors of *Swedish2* and *apparently2* would each be compiled once, so they have the appropriate form. The adjective *Swedish1* may then combine either directly with *Swedish2* so it can be applied to the noun; or it may first combine with the two lexical entries for *apparently* to yield the fully modified phrase. The former combination would be possible and one of the partial solutions, but not a valid one, as it does not contain all initial premises. Only the second option would be recognized as a valid solution by the prover.

4 Structure of the workbench

This section presents the overall structure of the system. Thereby, we want to give the reader a brief overview of the packages and how they are organized to allow for extension with one's own work.

When implementing this program, the intention was not only to provide an easily-accessible Glue prover, but also to create a tool that is interesting for formal linguists, especially formal semanticists and those working at the syntax-semantics interface, and also for users who are interested in NLP applications. In order to make the workbench extendable for any of these purposes, the code was modularized. Figure 6 shows the structure of the program, with arrows indicating the flow of data. Blue boxes represent packages and green boxes represent Java classes. The prover itself, as the core component of the workbench, has its own module. It takes a list of lexical entries as input and then searches for all valid solutions using the algorithm outlined above. The deduction process, as well as all valid solutions are printed and displayed to the user. Input for the *prover* module can be generated in two ways: by directly entering all lexical entries or by using an interface to XLE or to the Stanford CoreNLP dependency parser (Manning et al., 2014).⁷

⁷As of the publication of this paper, there exists no interface with non-Java libraries in the workbench. This means that the dependency structure is generated at run-time upon entering a sentence since it can use the Stanford CoreNLP Java library. On the other hand, XLE parses still need to be generated externally. We hope to integrate this functionality for future iterations of the workbench.

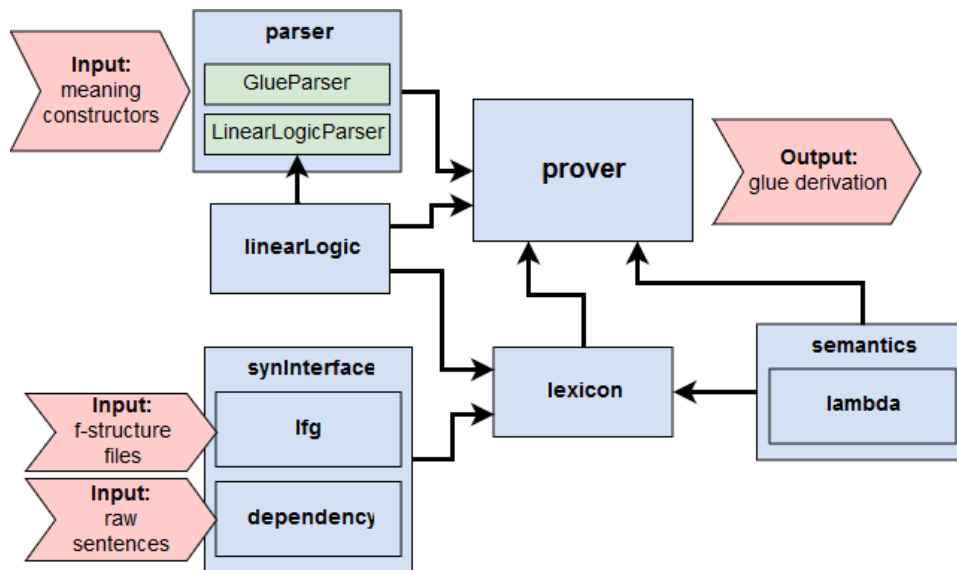


Figure 6: Module diagram

The Workbench also has a built-in parsing system by using the programming interface of the Stanford CoreNLP tools to create dependency structures. This allows users to parse single sentences and create the appropriate lexical entries from a small toy lexicon. This is discussed in more detail in section 4.1.

Either way, the user input is converted to a list of premises as input for the prover. The components for parsing lexical entries directly are situated inside the *parser* module, while the LFG and dependency structure input is handled in separate packages inside the *synInterface* module. All Java resources that are related to linear logic and the underlying proof system are part of the *linearLogic* module. Classes used for representing semantic formalisms can be found inside the *semantics* module. It contains an implementation of Montague-style lambda calculus that is used as a default semantic framework for the prover. In the *lexicon* module all classes for creating lexical entries from syntactically parsed input can be found. The distribution of the workbench contains a toy lexicon, but it can be extended, or even completely replaced, as desired by the user. In the remainder of this section, the two methods of providing input for the prover will be outlined.

4.1 Generating lexical entries

The systems for creating lexical entries are very similar for LFG and dependency structures. therefore only the generation of lexical entries based on LFG structures will be described here. The f-structure parser in the *synInterface* module reads f-structure files in Prolog syntax and generates lexical entries from the syntactic information extracted from the input file. The XLE interface is thus compatible with stand-alone XLE distributions and also with Prolog output generated by the

INESS XLE-Web service (Rosén et al., 2012). It also has a small toy lexicon integrated that can be extended and modified. In the original release of the Workbench, it contains classes for verbs (intransitive, transitive and ditransitive), common and proper nouns, determiners (including quantifiers) and adjectival modifiers.

Both syntactic frameworks access the *lexicon* module for generating lexical entries. In this module, the information given by the syntactic analysis is used to generate a semantic representation for the semantic side and a linear logic formula for the Glue side of the lexical entry. These lexical entries are then converted into premises that can be used by the prover. The generation of lexical entry objects from the input data follows the same principle in both systems. First, the root predicate and its arguments are determined. The arguments are resolved first so the appropriate template for the verb can be chosen, based on the subcategorization frame. The head of an argument f-structure is resolved first and afterwards its subordinate items, such as modifiers and determiners. Such dependents of an f-structure head include structural information about their head in their Glue meaning constructors.

$$(10) \quad \left[\begin{array}{l} \text{PRED 'man'} \\ \text{ADJ} \left\{ \left[\text{PRED 'swedish'} \right] \right\} \end{array} \right]$$

$$(11) \quad \mathbf{man} \lambda x.man(x) : v \multimap r \\ \mathbf{Swedish} \lambda P.\lambda y.Swedish(y) \wedge P(y) : (v \multimap r) \multimap (v \multimap r)$$

Consider the lexical entries for the NP 'Swedish man' in (11) (Dalrymple, 2001). The adjectival modifier 'Swedish' is part of the f-structure of the NP and therefore uses the same Glue labels as its head. The noun itself has the semantic type $\langle e, t \rangle$ and as the adjective modifies this meaning, its type is $\langle \langle e, t \rangle, \langle e, t \rangle \rangle$. These semantic types are reflected in the Glue labels. The lexical resource we provide uses a top-down algorithm for generating meaning constructors. In the example above, ((10)), this would be the constructor for *man*. The modifier *swedish* which is subordinated from an f-structure perspective can thus access the relevant Glue elements which have been generated for the governing structure. Thus, in ((11)) the entries share the Glue constants v and r . In other words, the head of each dependent is always resolved first and therefore the necessary information of a given (partial) f-structure can be passed down to its modifiers. In the toy lexicon that is provided with the Glue Workbench modifiers only take the Glue labels of their heads as arguments, but as all lexical and functional information of the parent f-structures is available during the instantiation of the lexical entry for the modifier, other restrictions, such as semantic types could be passed as well.

4.2 Parsing Glue premises

As Glue semantics is a framework with growing interest from different semantic and syntactic backgrounds, the Workbench tries to honor that diversity by provid-

ing the possibility of directly entering and parsing Glue meaning constructors that can then be fed into the Glue prover. The “native” system of the Workbench is a Montague-style lambda calculus, but it is possible to use other semantic frameworks, such as DRT. However, by default, the compilation algorithm employed by the prover uses lambda abstraction and lambda application operations to modify the meanings accordingly. The compilation algorithm was implemented in such a way that it is possible to add a different semantic formalism (via Java interfaces).

When entering lexical entries manually, the prover will use the “default” classes for generating the meaning side. This means that all semantic representations will be treated as an atomic string of characters that is not modified. During the derivation process lambda abstractions may be added and modified, but the core meaning provided in the original lexical entry will remain untouched. That way, the Workbench allows using any kind of semantic framework as input and the semantic representations that are derived by the prover can be evaluated manually or with a beta reduction tool.

5 Conclusion

In this paper the Glue Semantics Workbench was outlined as a tool for research at the syntax/semantics interface. The Workbench is centered around a Glue proving algorithm which is able to process Glue semantic expressions that are part of the implicational subset of linear logic, commonly used in the newer style of Glue semantics. Our implementation resolves issues with the computational tractability and implements some improvements in efficiency, using the algorithms outlined by Hepple (1996) and Gupta & Lamping (1998). Due to its modularized implementation via Java packages, the Workbench allows some flexibility for the user. It offers three modes for providing lexical entries as input for the parser: entering and parsing them directly or letting the *lexicon* and *synInterface* modules derive them either from LFG f-structures or dependency parses. The modular structure allows relatively easy modification and extension of its modules.

References

- Asudeh, Ash & Richard Crouch. 2002. Glue Semantics for HPSG. In *Proceedings of the 8th International HPSG Conference*, 1–19.
- Bobrow, Daniel G., Bob Cheslow, Cleo Condoravdi, Lauri Karttunen, Tracy Holloway King, Rowan Nairn, Valeria de Paiva, Charlotte Price & Annie Zaenen. 2007. PARC’s Bridge and Question Answering System. In *Proceedings of the GEAF 2007 Workshop*, 1–22.
- Cooper, Robin. 1983. *Quantification and Syntactic Theory*, vol. 21 Synthese Language Library, Texts and Studies in Linguistics and Philosophy. Dordrecht: Springer.

- Crouch, Dick, Mary Dalrymple, Ronald M. Kaplan, Tracy Holloway King, John T. Maxwell III & Paula Newman. 2017. *XLE Documentation*. Palo Alto Research Center.
- Crouch, Richard & Josef van Genabith. 2000. Linear Logic for Linguists. <http://www2.parc.com/istl/members/crouch>.
- Dalrymple, Mary. 2001. *Lexical Functional Grammar*, vol. 34. Academic Press.
- Frank, Anette & Josef van Genabith. 2001. GlueTag-Linear Logic Based Semantics for LTAG – and What It Teaches Us About LFG and LTAG. In *Proceedings of the LFG01 Conference*, CSLI Publication.
- Garrette, Dan & Ewan Klein. 2009. An Extensible Toolkit for Computational Semantics. In *Proceedings of the eighth international conference on computational semantics*, 116–127. Association for Computational Linguistics.
- Gotham, Matthew. 2015. Towards Glue Semantics for Minimalist Syntax. *Cambridge Occasional Papers in Linguistics* 8. 5683.
- Gotham, Matthew. 2018. Making Logical Form Type-Logical: Glue Semantics for Minimalist Syntax. *Linguistics and Philosophy* 1–46.
- Gotham, Matthew & Dag Haug. to appear. Glue semantics for Universal Dependencies. In *Proceedings of the 23rd lexical functional grammar conference*, tbd.
- Gupta, Vineet & John Lamping. 1998. Efficient Linear Logic Meaning Assembly. In *Proceedings of the 17th international conference on computational linguistics-volume 1*, 464–470. Association for Computational Linguistics.
- Hepple, Mark. 1996. A Compilation-Chart Method for Linear Categorical Deduction. In *Proceedings of the 16th conference on computational linguistics-volume 1*, 537–542. Association for Computational Linguistics.
- Kokkonidis, Miltiadis. 2008. First-Order Glue. *Journal of Logic, Language and Information* 17(1). 43–68.
- Manning, Christopher, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard & David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, 55–60.
- Montague, Richard. 1970. English as a formal language. In Bruno Visentini (ed.), *Linguaggi nella societa e nella tecnica*, 188–221. Edizioni di Comunita.
- Rosén, Victoria, Koenraad De Smedt, Paul Meurer & Helge Dyvik. 2012. An Open Infrastructure for Advanced Treebanking. In *Meta-research workshop on advanced treebanking at Irec2012*, 22–29. Hajič, Jan.