# The Glue Semantics Work Bench: A Modular Toolkit for Exploring Linear Logic and Glue Semantics

Moritz Meßmer and Mark-Matthias Zymla
University of Konstanz
moritz.messmer@uni-konstanz.de
mark-matthias.zymla@uni-konstanz.de

In this paper we present an easy to use, modular glue semantic prover building on the work by Crouch and van Genabith (2000) and others to bring more attention to the development of semantic resources for XLE/LFG. 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. 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 wrt. other syntactic theories as well as different semantic formalisms.

Concretely, we present a glue semantics prover written in Java. We chose Java for our implementation, because it is an accessible and widely known state-of-the-art programming language that allows interfacing several other NLP resources such as the Stanford CoreNLP tools. Through this we hope to make glue semantics and linear logic more accessible for modern NLP applications and CL in general. The code consists of a parser for reading linear logic formulas, a lexicon interface for deriving meaning representations from syntactic structures, and the glue prover itself. Our prover (as of now) handles the implicational fragment of linear logic used in glue semantics. It also deals with variables on the glue side which are implicitly bound by a universal quantifier. Linear logic variables are used in lexical entries of quantificational expressions (Dalrymple, 2001).

Even the smallest fragment of linear logic is NP-complete (Kanovich, 1992) and therefore computationally intractable. The algorithm for our prover implements the methods described in Hepple (1996) and Gupta and Lamping (1998) and is inspired by a previous implementation by Richard Crouch. In order to make linear logic computationally feasible three methods are employed which will be briefly outlined here.

(1) $$\frac{f : A \multimap B \ [0] \qquad [x : A]^i \ [1]}{f(x) : B \ [0,1]} \multimap_E$$

(2) $(A \multimap B) \multimap C : x$
$\Rightarrow_{compile} B[A] \multimap C : \lambda v.x(\lambda u.v), \{A\} : u$

In the spirit of chart-parsing algorithms, the Hepple prover employs an indexing system to keep track of resources that are already used and to avoid recomputing steps that have already been made. Each premise is assigned a set of indexes. The initial premises of a proof are each assigned a single unique index. When two premises are combined, a new premise is created whose index set is the joined set of indexes of the initial premises. By making sure that the two index sets of the potentially combinable premises are disjoint, the algorithm avoids redundant recombination steps. (1) shows a combination of two resources with a glue side (linear logic) and a meaning side ($\lambda$-calculus). The Curry-Howard isomorphism describes the correspondence between formulas in a formal logic system and types in a computational system, such as $\lambda$-calculus. In the case of glue semantics this means that operations on the glue side of a term ($\multimap$-introduction and -elimination) have corresponding rules on the meaning side ($\lambda$-abstraction and -application).

Higher-order formulas, that is, implication formulas with complex antecedents, are compiled into sets of first-order formulas which can then be combined using the method described above. Compiling a formula means cutting out the antecedent(s) of a higher-order formula and adding them to the set of premises as assumptions, as shown in (2). This allows us to make deductions with premises of any complexity, however, the algorithm needs to ensure that assumptions are only used to prove the consequent they belonged to in the original formula. For this purpose, extracted assumptions are marked on the original formula as a *discharge*. A formula with a discharge $A$ can only be combined with the respective assumption $A$ or with a formula that was combined with that assumption at some point. Following Gupta and Lamping (1998) we keep track of assumptions and discharges directly. This is facilitated by the object-oriented nature of Java which allows us to simply check whether a discharge and assumption resource refer to the same Java object. In our system, the compilation preserves the Curry-Howard isomorphism on the meaning side by introducing a fresh variable on the extracted assumption and a fresh lambda term which later rebinds the variable of the assumption on the original term, as (2) shows. Again, object-orientated programming helps with this, because atomic parts of formulas can be

referenced from different formulas. By modifying for instance the Java object that signifies a lambda-bound variable, all instances of that variable are modified as well, since they are all references to the same object.

The third method, which improves the efficiency of the algorithm, is a separation of skeleton and modifier resources, as proposed by Gupta and Lamping (1998). A modifier resource is one whose glue side is of the form $A \multimap A$, where A may be an atom or a complex formula. This means that every every resource on the consumer side of the implication is matched up with one on the producer side. Such resources significantly increase the computational complexity of a proof, because they can be applied in varying orders and therefore significantly increase the number of possible combinations. By separating non-symmetric skeleton resources from modifier resources and combining all skeleton resources first, the algorithm cuts out unnecessary combination steps and thus makes the deduction easier to compute.

Hooking up the glue prover with a syntactic parser is straightforward. There are in principle two possibilities: produce semantic resources from the syntactic output as strings that can be parsed by the glue prover. These strings have an intuitive, easy to type syntax. They can potentially be generated in various ways. One example would be: as output generated by the XLE transfer system, that allows for rewriting of syntactic representations via specific transfer rules (Crouch et al., 2017). This system has already been used extensively for generating semantic representations (Crouch (2005); Crouch and King (2006) among others), however, not specifically for glue semantics. Rather, the transfer system has been used to explore alternative semantic representations – most prominently abstract knowledge representations (AKR) – for NLP applications (Bobrow et al., 2007).

The second way to generate lexical resources that can be fed into the glue prover is to generate them directly within the Java framework available in the code. This is useful for example when working with Java resources such as the Stanford dependency parser that is part of the CoreNLP bundle (Chen and Manning, 2014). In this case the surface form of the glue premises is generated automatically. The system presented here comes with an inbuilt lexicon for the Stanford parser comparable to the system proposed in Garrette and Klein (2009). It generates semantic resources for minimal elements (nodes) of the tree and enriches them with semantic information that is available for the construction of other resources, i.e. it builds up a semantic structure akin to the traditional s(emantic)-structure used in LFG (Bresnan et al., 2015). In terms of the semantic representation, the glue prover comes with a basic implementation of lambda calculus, which on the one hand is necessary for the compilation of linear logic formulas (see above) and on the other hand can be used to hook the prover up with a semantic formalism. Consider the traditional example of quantifier ambiguity illustrated below in (3). The nominal arguments provide a semantic resource for each of their parts, i.e. the noun and the quantifier. The noun resource corresponds to the first (complex) argument of the respective quantifier and forms the restrictor of the quantification. Each quantifier introduces a free scope variable which is picked up by the verbal root to form the appropriate semantic resource. Their respective semantic forms can be derived via the Curry-Howard isomorphism resulting in the traditional first-order logic quantifiers.

All in all, we present a system for glue semantics at the core of which lies a simple but powerful glue prover. The prover is supported by a module allowing to generate typed lambda-calculus formulas and the corresponding lambda-calculus operations (e.g. $\beta$-conversion). Furthermore, the package comes with a module that translates syntactic input into semantic resources, i.e. a system that generates a lexicon. The whole system is generated with extensibility in mind and can find its uses both in active research of the syntax/semantics interface as well as formal semantics and education on LFG and computational semantics.

(3) Every man owns a dog.
   a. $\forall x[man(x) \rightarrow \exists z[dog(z) \wedge owns(x,z)]]$
   b. $\exists z[dog(z) \wedge \forall x[man(x) \rightarrow owns(x,z)]]$

$$
\mathrm{f}\begin{bmatrix}
\text{PRED } \text{'own<man,dog>'} \\
\text{SUBJ} \quad g\begin{bmatrix}\text{PRED 'man'} \\ \text{DET} \quad \begin{bmatrix}\text{PRED 'every'}\end{bmatrix}\end{bmatrix} \\
\text{OBJ} \quad h\begin{bmatrix}\text{PRED 'dog'} \\ \text{DET} \quad \begin{bmatrix}\text{PRED 'a'}\end{bmatrix}\end{bmatrix} \\
\ldots
\end{bmatrix}
$$

Figure 1: Sample f-structure (simplified)

$$
\begin{aligned}
\mathrm{g} \;&:\; ((g \multimap g) \multimap ((\boldsymbol{h} \multimap Y) \multimap Y))\,, \\
&\quad\; (g \multimap g) \\
\mathrm{h} \;&:\; ((i \multimap i) \multimap ((\boldsymbol{j} \multimap X) \multimap X))\,, \\
&\quad\; (i \multimap i) \\
\mathrm{f} \;&:\; \boldsymbol{h} \multimap \boldsymbol{j} \multimap f
\end{aligned}
$$

Figure 2: Correspondence between verb, nominal predicates and quantifiers

**Output of a sample derivation for the sentence "Every man owns a dog" (surface scope):**

$(g \multimap g) : \lambda y\_e.man(y)$ [0]
$((g \multimap g) \multimap ((h \multimap Y) \multimap Y)) : \lambda P\_ < e,t > .\lambda Q\_ < e,t > .\forall x[P(x) \rightarrow Q(x)]$ [1]
$(i \multimap i) : \lambda x'\_e.dog(x')$ [2]
$((i \multimap i) \multimap ((j \multimap X) \multimap X)) : \lambda R\_ < e,t > .\lambda S\_ < e,t > .\exists z[R(z) \wedge S(z)]$ [3]
$(h \multimap (j \multimap f)) : \lambda x\_e.\lambda y\_e.owns(x,y)$ [4]

Compiled resources:
$(g \multimap g) : \lambda s\_e.man(s)$[0]
$(Y[\{h\}] \multimap (g \multimap Y)) : \lambda u\_t.\lambda t\_t.\lambda Q\_ < e,t > .\lambda P\_ < e,t > .\forall y[P(y) \rightarrow Q(y)](\lambda x\_e.t)(\lambda x'\_e.u)$[1]
$(i \multimap i) : \lambda v\_e.dog(v)$[2]
$(X'[\{j\}] \multimap (i \multimap X')) : \lambda t'\_t.\lambda s'\_t.\lambda S\_ < e,t > .\lambda R\_ < e,t > .\exists z[R(z) \wedge S(z)](\lambda y\_e.s')(\lambda y'\_e.t')$[3]
$(h \multimap (j \multimap f)) : \lambda u'\_e.\lambda v'\_e.owns(u',v')$[4]
$\{g\} : x$[5]
$\{h\} : x'$[6]
$\{i\} : y$[7]
$\{j\} : y'$[8]

Checking simple prover...
Combining premises $(i \multimap i) : \lambda v\_e.dog(v)$[2] and $\{i\} : y$[7]
$\longrightarrow i\{i\} : dog(y)$[2,7]

Combining premises $\{h\} : x'$[6] and $(h \multimap (j \multimap f)) : \lambda u'\_e.\lambda v'\_e.owns(u',v')$[4]
$\longrightarrow (j \multimap f)\{h\} : \lambda v'\_e.owns(x',v')$[4,6]

Combining premises $(j \multimap f)\{h\} : \lambda v'\_e.owns(x',v')$[4,6] and $\{j\} : y'$[8]
$\longrightarrow f\{h,j\} : owns(x',y')$[4,6,8]

Combining premises $f\{h,j\} : owns(x',y')$[4,6,8]
and $(X'[\{j\}] \multimap (i \multimap X')) : \lambda t'\_t.\lambda s'\_t.\lambda S\_ < e,t > .\lambda R\_ < e,t > .\exists z[R(z) \wedge S(z)](\lambda y\_e.s')(\lambda y'\_e.t')$[3]
$\longrightarrow (i[\{i\}] \multimap f)\{h\} : \lambda s'\_t.\lambda S\_ < e,t > .\lambda R\_ < e,t > .\exists z[R(z) \wedge S(z)](\lambda y\_e.s')(\lambda y'\_e.owns(x',y'))$[3,4,6,8]

Combining premises $(i[\{i\}] \multimap f)\{h\} : \lambda s'\_t.\lambda S\_ < e,t > .\lambda R\_ < e,t > .\exists z[R(z) \wedge S(z)](\lambda y\_e.s')(\lambda y'\_e.owns(x',y'))$[3,4,6,8]
and $i\{i\} : dog(y)$[2,7]
$\longrightarrow f\{h\} : \exists z[dog(z) \wedge owns(x',z)]$[2,3,4,6,7,8]

Combining premises $f\{h\} : \exists z[dog(z) \wedge owns(x',z)]$[2,3,4,6,7,8]
and $(Y[\{h\}] \multimap (g \multimap Y)) : \lambda u\_t.\lambda t\_t.\lambda Q\_ < e,t > .\lambda P\_ < e,t > .\forall y[P(y) \rightarrow Q(y)](\lambda x\_e.t)(\lambda x'\_e.u)$[1]
$\longrightarrow (g[\{g\}] \multimap f) : \lambda t\_t.\lambda Q\_ < e,t > .\lambda P\_ < e,t > .\forall y[P(y) \rightarrow Q(y)](\lambda x\_e.t)(\lambda x'\_e.\exists z[dog(z) \wedge owns(x',z)])$[1,2,3,4,6,7,8]

Combining premises $(g \multimap g) : \lambda s\_e.man(s)$[0] and $\{g\} : x$[5]
$\longrightarrow g\{g\} : man(x)$[0,5]

Combining premises $g\{g\} : man(x)$[0,5]
and $(g[\{g\}] \multimap f) : \lambda t\_t.\lambda Q\_ < e,t > .\lambda P\_ < e,t > .\forall y[P(y) \rightarrow Q(y)](\lambda x\_e.t)(\lambda x'\_e.\exists z[dog(z) \wedge owns(x',z)])$[1,2,3,4,6,7,8]
$\longrightarrow f : \forall y[man(y) \rightarrow \exists z[dog(z) \wedge owns(y,z)]]$[0,1,2,3,4,5,6,7,8]
...
Found valid deduction(s):
$f : \forall y[man(y) \rightarrow \exists z[dog(z) \wedge owns(y,z)]]$[0,1,2,3,4,5,6,7,8]
$f : \exists z[dog(z) \wedge \forall y[man(y) \rightarrow owns(y,z)]]$[0,1,2,3,4,5,6,7,8]

# References


Bobrow, Daniel G., Bob Cheslow, Cleo Condoravdi, Lauri Karttunen, Tracy Holloway King, Rowan Nairn, Valeria de Paiva, Charlotte Price, and Annie Zaenen. 2007. PARC's Bridge and Question Answering System. In *Proceedings of the GEAF 2007 Workshop*, Pages 1–22.

Bresnan, Joan, Ash Asudeh, Ida Toivonen, and Stephen Wechsler. 2015. *Lexical-functional syntax*, volume 16. John Wiley & Sons.

Chen, Danqi and Christopher Manning. 2014. A Fast and Accurate Dependency Parser Using Neural Networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Pages 740–750.

Crouch, Dick, Mary Dalrymple, Ronald M. Kaplan, Tracy Holloway King, John T. Maxwell III, and Paula Newman. 2017. *XLE Documentation*. Palo Alto Research Center.

Crouch, Richard. 2005. Packed Rewriting for Mapping Semantics to KR. In *Proceedings of the Sixth International Workshop on Computational Semantics (IWCS-6)*, Pages 103–114. Tilburg.

Crouch, Richard and Tracy Holloway King. 2006. Semantics via F-Structure Rewriting. In M. Butt and T. H. King, editors., *Proceedings of the LFG06 Conference*, Pages 145–165. Stanford, CA: CSLI Publications.

Crouch, Richard and Josef van Genabith. 2000. Linear logic for linguists. *URL: http://www2. parc. com/istl/members/crouch* .

Dalrymple, Mary. 2001. *Lexical Functional Grammar*, volume 34 of *Syntax and Semantics*. New York: Academic Press.

Garrette, Dan and Ewan Klein. 2009. An Extensible Toolkit for Computational Semantics. In *Proceedings of the Eighth International Conference on Computational Semantics*, Pages 116–127. Association for Computational Linguistics.

Gupta, Vineet and John Lamping. 1998. Efficient linear logic meaning assembly. In *Proceedings of the 17th international conference on Computational linguistics-Volume 1*, Pages 464–470. Association for Computational Linguistics.

Hepple, Mark. 1996. A compilation-chart method for linear categorial deduction. In *Proceedings of the 16th conference on Computational linguistics-Volume 1*, Pages 537–542. Association for Computational Linguistics.

Kanovich, M. I. 1992. Horn programming in linear logic is np-complete. In *[1992] Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, Pages 200–210.