XLE+Glue – A new tool for integrating semantic analysis in XLE

Mary Dalrymple
University of Oxford

Agnieszka Patejuk
University of Oxford/Institute of Computer Science, Polish Academy of Sciences

Mark-Matthias Zymla
University of Konstanz

Proceedings of the LFG'20 Conference

On-Line

Miriam Butt, Ida Toivonen (Editors)

2020

CSLI Publications

pages 89–108

http://csli-publications.stanford.edu/LFG/2020

Keywords: XLE, Glue, GSWB, semantics, codescription

Dalrymple, Mary, Patejuk, Agnieszka, & Zymla, Mark-Matthias. 2020. XLE+Glue – A new tool for integrating semantic analysis in XLE. In Butt, Miriam, & Toivonen, Ida (Eds.), *Proceedings of the LFG'20 Conference, On-Line*, 89–108. Stanford, CA: CSLI Publications.

Abstract

This paper presents the XLE+Glue system which provides an interface between XLE and the Glue Semantics Workbench, a tool for computational Glue semantics. It describes how to develop grammars encoding glue premises and how to calculate meanings based on these premises.

1 Overview

In this paper, we present XLE+Glue,[1] a resource for grammar developers that integrates semantic capabilities into the Xerox Linguistics Environment (XLE; Crouch et al. (2017)). Although XLE is the main computational implementation of LFG in general, it mainly focuses on the syntactic components of the grammar theory. While there exist notable approaches to semantic analysis paired with the system (see Crouch and King (2006), Crouch (2005)), resources for the theoretically founded formalism of Glue semantics remain sparse. To address this shortcoming, we developed an interface for XLE which integrates a glue prover – the Glue Semantics Workbench (GSWB; Meßmer and Zymla (2018)) – making it possible to derive semantic representations via linear logic (Dalrymple 1999).

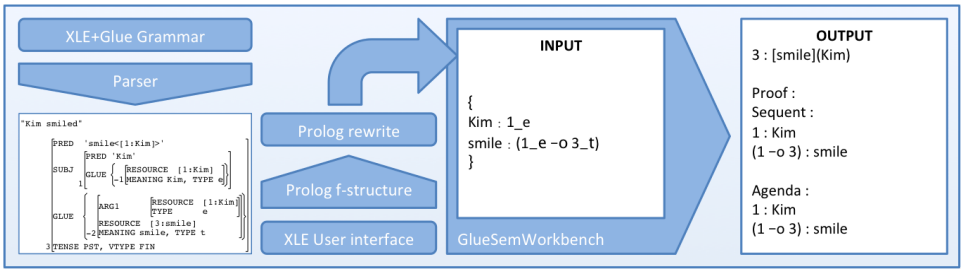

Figure 1: The XLE+Glue pipeline includes a glue prover (GSWB)

Figure 1 illustrates the overall structure of the XLE+Glue system. The system requires an XLE+Glue grammar, i.e., a grammar which encodes glue premises (meaning constructors) in its output (see the f-structure in Figure 1). The system processes the output of the parser using a Prolog rewrite script: on the basis of the Prolog representation of a given XLE parse, it creates an input file to the GSWB which is used to derive a glue proof as described in Meßmer and Zymla (2018).

Glue meaning constructors consist of a semantic side (any semantic formalism) and a glue side (a linear logic expression of linguistic resources of a given type). The up and down metavariables in lexical entries are instantiated to indices representing particular f-structures (or s-structures; Dalrymple (1999)), for instance:

(1) a. $Kim : \uparrow_e \quad \Rightarrow \quad Kim : k_e$

†Agnieszka Patejuk gratefully acknowledges the Mobilność Plus mobility grant awarded by the Polish Ministry of Science and Higher Education.

[1] The system is available from https://github.com/Mmaz1988/xle-glueworkbench-interface. For information on installing and running XLE+Glue, see the README or the manual in the repository.

b. *smile* : $(\uparrow \text{SUBJ})_e \multimap \uparrow_t \quad \Rightarrow \quad$ *smile* : $k_e \multimap s_t$

In the first part of this paper, we explain the encoding of meaning constructors in XLE+Glue grammars; for ease of presentation, we make several assumptions. First, we assume simple untyped meaning representations such as *Kim* and *smile* which serve as placeholders for various potential meaning languages.[2] Second, we assume that the glue side of meaning constructors refers to f-structures (as in our sample grammar `glue-basic.lfg`) rather than their semantic projections or other linguistic levels. However, our system can handle meaning constructors referring to other linguistic levels (as demonstrated in our sample grammar `glue-basic-semstr.lfg`, which includes a semantic projection).

Our system provides two ways of encoding meaning constructors. Both methods rely on the presence of a special GLUE attribute whose value is a set of meaning constructors in AVM format. The left-hand side of Figure 1 illustrates the embedded encoding: meaning constructors are encoded in an attribute-value matrix (AVM) format in which embedding in the AVM mirrors the structure of the glue side of the meaning constructor. We describe this method in Section 2. In the string-based method, the meaning constructor is represented as a sequence of substrings which are values of an ordered set of attributes, as described in Section 5.2.

In the second part of this paper, we explain how XLE interacts with the GSWB to provide Glue semantics derivations. First, we explain the requirements imposed by the GSWB in Section 3. Based on this knowledge, we describe in Section 4 the Prolog rewrite script which serves as the bridge between XLE and the GSWB.

Finally, in Section 5 we describe various ways in which the system can be adapted to particular use cases. As mentioned above, there are two alternative methods of encoding meaning constructors in the XLE output. Furthermore, the system allows for variation in terms of meaning constructors: we focus on the possibility of using different semantic formalisms on the meaning side of XLE+Glue meaning constructors. Section 6 concludes the paper.

2 Encoding meaning constructors as AVMs

(2-a) illustrates the meaning constructor for the proper name *Kim* in the standard format, and (2-b) illustrates the corresponding AVM encoding, where the meaning constructor appears as a member of the GLUE set. In this section, we follow the usual notational convention of referring to f-structures by means of letters like k and s, but in later sections we will use numbers instead, since the Prolog rewrite script relies on the numeric indices assigned to f-structures (or other relevant structures) in the Prolog output format of XLE. We start with examples where each GLUE set contains only one meaning constructor, but in other cases several meaning constructors appear as members of the GLUE set, as we show in Section 2.2.

[2] See Section 3.2 for discussion of the ways in which meanings can be represented in the system, including as terms of the typed lambda calculus.

(2) a. *Kim*: k_e

b. k : $\left[\begin{array}{ll} \text{PRED} & \text{`KIM'} \\ \text{GLUE} & \left\{\left[\begin{array}{ll} \text{MEANING} & \text{KIM} \\ \text{RESOURCE} & k \\ \text{TYPE} & e \end{array}\right]\right\} \end{array}\right]$

In the AVM encoding,[3] each glue meaning constructor minimally consists of three attributes. The value of MEANING is the semantic (left-hand) side of the meaning constructor, while RESOURCE and TYPE specify the glue (right-hand) side: RESOURCE points to the relevant linguistic resource, while TYPE specifies RESOURCE's type. In XLE notation, the f-structure constraints contributed by a proper name like "Kim" are:

(3)
```
(^ PRED) = 'Kim'
(%mc MEANING) = Kim
(%mc RESOURCE) = ^
(%mc TYPE) = e
%mc $ (^ GLUE)
```

`%mc` is a local name (see Section 2.3.4) used to construct an attribute-value structure containing attributes specifying the glue meaning constructor (`MEANING`, `RESOURCE`, `TYPE`). This f-structure is added to the `GLUE` set by specification of the constraint `%mc $ (^ GLUE)`.

A glue meaning constructor may also involve implication, as for a verb like *smile* in (4), where a resource is consumed in order to produce another resource. The standard meaning constructor for the verb *smile* is given in (4-a), and the AVM translation is given in (4-b). In the AVM encoding, ARG1 is the first resource to be consumed, ARG2 the second, etc. The resources to be consumed are specified using the RESOURCE and TYPE attributes. See (5) for constraints contributed by *smile*:

(4) a. *smile*: $k_e \multimap s_t$

b. s : $\left[\begin{array}{ll} \text{PRED} & \text{`SMILE<SUBJ>'} \\ \text{SUBJ} & k\text{:[ ]} \\ \text{GLUE} & \left\{\left[\begin{array}{ll} \text{MEANING} & \text{SMILE} \\ \text{ARG1} & \left[\begin{array}{ll} \text{RESOURCE} & k \\ \text{TYPE} & e \end{array}\right] \\ \text{RESOURCE} & s \\ \text{TYPE} & t \end{array}\right]\right\} \end{array}\right]$

(5)
```
(^ PRED) = 'smile<(^ SUBJ)>'
(%mc MEANING) = smile
(%mc RESOURCE) = ^
(%mc TYPE) = t
```

[3]Here and in the rest of this section, glue premises are presented in the embedded encoding format. We discuss the alternative flat encoding format in Section 5.2.

```
(%mc ARG1 RESOURCE) = (^ SUBJ)
(%mc ARG1 TYPE) = e
%mc $ (^ GLUE)
```

The f-structure for the sentence "Kim smiles" is shown in (6). The Prolog rewrite component (described in Section 4) collects up all of the premises in the GLUE sets, rewrites each premise into a format suitable for input to the prover (as shown in (7)), and passes the complete set of premises to the prover.

(6)
$$
s:\begin{bmatrix}
\text{PRED} & \text{`SMILE<SUBJ>'} \\
\text{SUBJ} & k:\begin{bmatrix}
\text{PRED} & \text{`KIM'} \\
\text{GLUE} & \left\{\begin{bmatrix}\text{MEANING} & \text{KIM} \\ \text{RESOURCE} & k \\ \text{TYPE} & e\end{bmatrix}\right\}
\end{bmatrix} \\
\text{GLUE} & \left\{\begin{bmatrix}
\text{MEANING} & \text{SMILE} \\
\text{ARG1} & \begin{bmatrix}\text{RESOURCE} & k \\ \text{TYPE} & e\end{bmatrix} \\
\text{RESOURCE} & s \\
\text{TYPE} & t
\end{bmatrix}\right\}
\end{bmatrix}
$$

(7)
```
{
Kim : k_e
smile : k_e -o s_t
}
```

2.1 Universal quantification over meaning constructors

In (8-b), the f-structure labeled e is the attribute-value encoding of the meaning constructor for the generalized quantifier "every" given in (8-a). It has two arguments: ARG1 represents the restriction of the quantifier, and ARG2 represents its scope. The value of the ARG1 attribute encodes the implication $(p_e \multimap p_t)$ (where p is the value of the PRED attribute of the noun phrase, as shown in (8-b)), which corresponds to a common noun meaning.[4] The value of the ARG2 attribute encodes an implication from m_e to F_t, where F is a variable bound by a universal quantifier, representing the scope of the quantifier, which is freely chosen: the universal quantifier $\forall$ allows for a choice among various scope possibilities. At the top level we have a new attribute FORALL,[5] which encodes the universal quantifier $\forall$ in (8-a).

[4]Our sample grammars make the non-standard assumption that the meaning of a common noun is a function from its PRED value of type e to its PRED value of type t; that is, a common noun like "person" has a lexical entry of the following form:

person: $(\uparrow \text{PRED})_e \multimap (\uparrow \text{PRED})_t$

This is done for simplicity, to avoid the introduction of attributes encoding VAR and RESTR as in standard treatments, and is not a necessary feature of the implementation.

[5]To display the attribute FORALL in XLE, select "constraints" in "Views" menu (or press "c") in the window containing the glue premises in AVM format.

(8) a. *every*: $\forall F.(p_e \multimap p_t) \multimap (m_e \multimap F_t) \multimap F_t$

b. m :
$$\left[\begin{array}{ll}\text{PRED} & \boxed{1}\, p \\ \text{GLUE} & \left\{\begin{array}{l} e : \left[\begin{array}{ll}\text{MEANING} & \textit{every} \\ \text{FORALL} & F \\ \text{ARG1} & \left[\begin{array}{ll}\text{ARG1} & \left[\begin{array}{ll}\text{RESOURCE} & \boxed{1} \\ \text{TYPE} & e\end{array}\right] \\ \text{RESOURCE} & \boxed{1} \\ \text{TYPE} & t\end{array}\right] \\ \text{ARG2} & \left[\begin{array}{ll}\text{ARG1} & \left[\begin{array}{ll}\text{RESOURCE} & m \\ \text{TYPE} & e\end{array}\right] \\ \text{RESOURCE} & F \\ \text{TYPE} & t\end{array}\right] \\ \text{RESOURCE} & F \\ \text{TYPE} & t\end{array}\right] \end{array}\right\}\end{array}\right]$$

2.2 Multiple meaning constructors contributed by a single word

(9) a. *every*: $\forall F.(p_e \multimap p_t) \multimap (m_e \multimap F_t) \multimap F_t$
person: $p_e \multimap p_t$

b. m :
$$\left[\begin{array}{ll}\text{PRED} & \boxed{1}\, \textit{everyone} \\ \text{GLUE} & \left\{\begin{array}{l} e : \left[\begin{array}{ll}\text{MEANING} & \textit{every} \\ \text{FORALL} & F \\ \text{ARG1} & \left[\begin{array}{ll}\text{ARG1} & \left[\begin{array}{ll}\text{RESOURCE} & \boxed{1} \\ \text{TYPE} & e\end{array}\right] \\ \text{RESOURCE} & \boxed{1} \\ \text{TYPE} & t\end{array}\right] \\ \text{ARG2} & \left[\begin{array}{ll}\text{ARG1} & \left[\begin{array}{ll}\text{RESOURCE} & m \\ \text{TYPE} & e\end{array}\right] \\ \text{RESOURCE} & F \\ \text{TYPE} & t\end{array}\right] \\ \text{RESOURCE} & F \\ \text{TYPE} & t\end{array}\right] \\ n : \left[\begin{array}{ll}\text{MEANING} & \textit{person} \\ \text{ARG1} & \left[\begin{array}{ll}\text{RESOURCE} & \boxed{1} \\ \text{TYPE} & e\end{array}\right] \\ \text{RESOURCE} & \boxed{1} \\ \text{TYPE} & t\end{array}\right] \end{array}\right\}\end{array}\right]$$

Example (9) illustrates the encoding of the meaning constructor for the quantifier *everyone*, decomposed into a meaning constructor contributing the "every" part

of the meaning and another meaning constructor contributing the "person" part. This allows for the modification of the restriction of the quantifier in examples like *everyone who smiled*, and illustrates the possibility for a single word to contribute more than one meaning constructor to the GLUE set. In (9-b), the f-structure labeled e is the same as the attribute-value encoding of the meaning constructor for *every* given in (8-b) in the previous section. The second member of the set, labeled n, is the same as the contribution we would expect for the common noun *person*.

2.3 Templates for meaning constructors

The XLE+Glue system features a number of sample grammars that we refer to in this paper. These grammars provide a set of templates for encoding meaning constructors which may be generally useful, though it is of course possible for grammar writers to develop their own set of templates or to modify the sample templates as needed. We describe the basic templates here; more discussion and a detailed description of the sample grammars can be found in the XLE+Glue manual available in the GitHub repository (see footnote 1) as well as in the comments in the grammar files.

2.3.1 The basic definitions

All of the templates which are used in defining meaning constructors in the sample grammars using the AVM encoding call the two basic templates `GLUE-RESOURCE` and `GLUE-MEANING`. The template `GLUE-RESOURCE` specifies an attribute-value structure `TypedRES` as having the value `R` for the attribute `RESOURCE` and the value `TY` for the attribute `TYPE`. The attributes `RESOURCE` and `TYPE` and their values must appear in all meaning constructors in the embedded encoding format[6] and argument specifications, to identify the relevant linguistic resource and its type.

(10)
```
GLUE-RESOURCE(R TypedRES TY) =  (TypedRES RESOURCE) = R
                                (TypedRES TYPE) = TY.
```

The template `GLUE-MEANING` specifies an attribute-value structure `TypedRES` as having the value `M` for the attribute `MEANING`. This attribute corresponds to the left-hand (meaning) side of the meaning constructor, and must appear once, at the top level of all AVM meaning constructors.

(11)
```
GLUE-MEANING(TypedRES M) =  (TypedRES MEANING) = M.
```

2.3.2 Non-implicational meaning constructors: Proper names

In the sample grammar `glue-basic.lfg`, the lexical entry for the proper name *Kim* is as in (12):

[6]This requirement does not apply to grammars which, instead of the embedded encoding, use the alternative string-based encoding of glue premises described in Section 5.2.

(12) `Kim N * @(PROPERNOUN Kim).`

The sample grammar `glue-basic.lfg` provides the template in (13) for proper names. It defines the f-structure `PRED` value, and calls the template `GLUE-PROPERNOUN` to define the meaning constructor in AVM format, passing in the argument `P`.

(13)
```
PROPERNOUN(P) = (^ PRED) = 'P'
                @(GLUE-PROPERNOUN P).
```

In `glue-basic.lfg`, the argument `P` of `PROPERNOUN` is used to construct the f-structure semantic form as well as appearing as the value of the `MEANING` attribute in the AVM meaning constructor. If it is desirable for the f-structure `PRED` value to be different from the `MEANING` value of the AVM meaning constructor, the `PROPERNOUN` template would have to be defined to take two arguments, one providing the `PRED` value and the other providing the `MEANING` value.

`GLUE-PROPERNOUN` simply calls `GLUE-REL0-MC` (mnemonic for "meaning constructor for relation of arity 0": in other words, a meaning constructor that requires no arguments). It specifies the first and second arguments of the template as `^` and `e` for all proper nouns, and passes in the value of `P` as the third argument.

(14)
```
GLUE-PROPERNOUN(P) = @(GLUE-REL0-MC ^ e P).
```

In the `glue-basic.lfg` grammar, it would also have been possible for the `PROPERNOUN` template to call `GLUE-REL0-MC` directly, providing the arguments `^` and `e`. The intermediate template `GLUE-PROPERNOUN` allows for the possibility that in scaling up to a more complete grammar, additional specifications may be associated with the `GLUE-PROPERNOUN` template, besides the definition of the meaning constructor.

The definition of `GLUE-REL0-MC` is:

(15)
```
GLUE-REL0-MC(R TY M) = @(GLUE-RESOURCE R %mc TY)
                       @(GLUE-MEANING %mc M)
                       %mc $ (R GLUE).
```

This template calls two basic templates: `GLUE-RESOURCE` and `GLUE-MEANING`. The call to `GLUE-RESOURCE` specifies properties of the AVM meaning constructor `%mc`: it has an attribute `RESOURCE` whose value is `R`, and it has an attribute `TYPE` whose value is `TY`. The call to `GLUE-MEANING` provides the value `M` for the attribute `MEANING` in `%mc`. The final line requires `%mc` to appear as a member of the `GLUE` set in the f-structure `R`.

When the template `GLUE-REL0-MC` is called with arguments `^`, `e`, and `Kim`, an AVM `%mc` is created which corresponds to the simple meaning constructor *Kim:*$\uparrow_e$. This AVM has three attributes: `RESOURCE`, whose value is `^`; `TYPE`, whose value is `e`; and `MEANING`, whose value is `Kim`. The final line of this template specifies that `%mc` is a member of the `GLUE` set in the f-structure `R`. Thus, the template call `@(PROPERNOUN Kim)` produces the f-description given in (3).

2.3.3 Meaning constructors requiring arguments: Intransitive verbs

In `glue-basic.lfg`, the lexical entry for the intransitive verb *smiled* is:

(16)
```
smiled   V   *  @(VERB-SUBJ smile)
                @VPAST.
```

The template `VPAST` specifies a past tense feature in the f-structure; we do not discuss this template here. The template `VERB-SUBJ` is defined as:

(17)
```
VERB-SUBJ(P) =  (^ PRED) = 'P<(^ SUBJ)>'
                @(GLUE-VERB-SUBJ P).
```

As with the proper name template described in the previous section, the template argument `P` is used to define both the f-structure semantic form and the `MEANING` value of the AVM meaning constructor. If this is not desirable, the template `VERB-SUBJ` should be defined to take two arguments, one specifying the semantic form and the other the value of the `MEANING` feature in the AVM meaning constructor. The template `GLUE-VERB-SUBJ` is defined as:

(18)
```
GLUE-VERB-SUBJ(P) =  @(GLUE-REL1-MC (^ SUBJ) e ^ t P).
```

As with the `GLUE-PROPERNOUN` template, the `GLUE-VERB-SUBJ` template simply calls `GLUE-REL1-MC` (mnemonic for "meaning constructor for relation of arity 1": in other words, a meaning constructor that requires one argument). In scaling up to a more complete grammar, there may be additional semantic specifications associated with `GLUE-VERB-SUBJ`. The template `GLUE-REL1-MC` is defined as:

(19)
```
GLUE-REL1-MC(A1 A1TY R TY M) =
            @(GLUE-RESOURCE R %mc TY)
            @(GLUE-RESOURCE A1 (%mc ARG1) A1TY)
            @(GLUE-MEANING %mc M)
            %mc $ (R GLUE).
```

The first, third, and fourth lines of this template are the same as for the template `GLUE-REL0-MC`: they specify that the meaning constructor in the `GLUE` set of this verb is called `%mc`, that it has an attribute `RESOURCE` with value `R`, and that it has an attribute `TYPE` with value `TY`. The additional specification in the second line adds an attribute `ARG1` to the structure, whose value for the `RESOURCE` feature is `A1`, and whose value for the `TYPE` feature is `A1TY`. When the template `GLUE-REL1-MC` is called with arguments `(^ SUBJ)`, `e`, `^`, `t`, and `smile`, the resulting f-description is as in (5).

2.3.4 Scope of local names

When writing an XLE+Glue grammar, it is important to be aware of the scope of local names (variables prefixed with `%`) in XLE. The scope is limited to the c-

structure category in which the variable is used.[7] This means that every time a given local name (for example: `%test`) is used within the same c-structure category, it refers to the same object. For instance, if there are two template calls using the same local name (`%test`) within one lexical entry, these template calls will impose constraints on the same object (one that corresponds to `%test`).

Depending on the intended effect, such behaviour of local names with respect to their scope may be a feature (when it is the intention to constrain the same object by separate template calls) or it may be undesired (when the intention is to impose constraints on two distinct objects by separate template calls) – this is why it is crucial to be aware of this when using local names.

This practical issue arises in sample grammars when a given c-structure category contributes more than one glue premise (see Section 2.2). For instance, as explained in the sample grammar `glue-basic.lfg`, the template `GLUE-NOUN0-MC` providing the meaning constructor for (common) nouns uses the local name `%mcn`, because it must be different from the local name `%mc` used by the template `GLUE-QUANT-MC` – both templates are called by the template `QUANT` which is called in the lexical entry of the quantifier "everyone". Another example can be found in the template `GLUE-ADJ0` which provides two meaning constructors for prenominal adjectives – the call to the template `GLUE-REL1-MC` uses the local name `%mc` to provide the meaning constructor for the basic meaning of the adjective, while the call to the template `GLUE-ADJ-MODIFIER` provides the meaning constructor combining the adjective with the noun by calling the template `GLUE-MODIFIER1` which uses the local name `%mcm` to build this meaning constructor.

2.4 Interim summary

So far, we have explained how to encode meaning constructors as AVMs, including how to make use of templates which are prevalent in grammar development with XLE. The main benefit of the AVM encoding is that it makes use of XLE's capabilities to ascertain well-formedness of the underlying structures. Meaning constructors are stored in the Prolog output of XLE, so it is important to provide a principled way of encoding them that does not clutter the output.

In the next part of the paper, we explain some technical details related to the encoding of meaning constructors as required by the GSWB. One of the main contributions of the XLE+Glue system is the translation between the meaning constructors encoded in XLE output and the input format for meaning constructors required by the GSWB. This is crucial for the system, since the output of XLE is based on Prolog, while the GSWB uses a more general format aimed at mimicking the way in which meaning constructors are encoded in Glue semantics theory.

[7] https://ling.sprachwiss.uni-konstanz.de/pages/xle/doc/notations.html#N4.1.6: "A local name can be used as a variable whose scope is limited to the schemata associated with a particular category or lexical item."

3 Semantic representations and the prover

The GSWB takes a set of premises, i.e., instantiated meaning constructors encoded in a specific string format, as input to calculate the semantics of an utterance. In this section, we describe the required input for the GSWB: Section 3.1 explains the encoding of linear logic formulas on the glue side, while Section 3.2 presents different ways of encoding the meaning side of a meaning constructor.

The GSWB uses a parser for semantic types which is shared between the semantic parser and the linear logic parser. See (20) for the available atomic types.[8] Complex types consist of atomic types, commas and angular brackets: see (21).

(20) Atomic types: e, s, v, t

(21) Complex types:

a. $< e, < e, < s, t >>>$
b. $<< s, t >, < s, t >>$
c. $<< e, t >, t >$
d. $<< e, t >, << e, t >, t >>$
e. $<< e, < s, t >>, < s, t >>$

3.1 Encoding the glue side

3.1.1 Parsing linear logic formulas

The GSWB encodes linear logic formulas in a simple string format that is visually similar to the actual symbols used in linear logic. For example, the $\multimap$ symbol is replaced by `-o`. This is illustrated in (22), where some sample formulas from the fragment of linear logic that is covered by the parser are shown. As shown in (23), formulas can also be stated without type declarations.[9]

(22) a. `g_e`
b. `(g_e -o (d_e -o (i_s -o h_t)))`
c. `((i_s -o h_t) -o (t_s -o f_t))`
d. `AX_t.((d_e -o X_t) -o X_t)`
e. `((g_e -o g_t) -o AX_t.((d_e -o X_t) -o X_t))`
f. `AX_t.AY_s.((d_e -o (Y_s -o X_t)) -o(Y_s -o X_t))`

(23) `((g -o g) -o AX.((d -o X) -o X))`

Examples (24) and (25) describe the formation of linear logic formulas that are well-formed from the perspective of the parser implemented in the prover. Most importantly, the parentheses around linear logic formulas are obligatory. Type declarations should either be applied to all constants and variables, or to none. Type

[8] Types need to be specifically declared in the code of the GSWB, so it is not straightforward to introduce new types. We intend to address this issue in future iterations of the GSWB. In the meantime, you can contact the authors to get help with the implementation of new types.

[9] An element without a type declaration is treated as an element of type *t*.

declarations are indicated by an underscore, e.g., `_t`. Universal quantification over linear logic formulas is encoded using an upper-case `A` followed by some variable (e.g. `X`) and a dot. As of now, the scope of a linear logic quantifier is the rest of the formula and does not need to be indicated, and in fact should not be indicated via parentheses or brackets.[10] This means that linear logic quantifiers behave differently from the first-order logic quantifiers introduced in Section 3.2.2.

(24) Atomic elements:
   a. *Constants:* String of lower-case alphanumeric characters; optionally with type declaration
   b. *Variables:* String of upper-case alphanumeric characters; optionally with type declaration

(25) Linear logic formulas:
   a. *Linear implication:* (ϕ -o ψ), where ϕ, ψ are well-formed formulas
   b. *Linear quantification:* AX. ϕ, where ϕ is a well-formed formula

3.2 Encoding the meaning side

The GSWB currently supports three modes for encoding meaning representations. Each mode needs to specify a procedure for encoding functional application and abstraction. The default mode of the GSWB is a simple concatenation mode.

3.2.1 Concatenation mode

In this mode, any (string-based) format of semantic representation is compatible with XLE+Glue. In this simple format, functional application is expressed in the output by wrapping the argument in parentheses and concatenating functor and argument as in (26-a). Abstraction is handled by introducing a corresponding lambda binder as in (26-b).

(26) a. Combining (1 $\multimap$ 0) : smile and 1 : Kim
   to: 0 : smile(Kim)
   b. smile(x) to λx.smile(x)

3.2.2 Semantic parser mode

This is the second input mode for the GSWB. The semantic parser provided by the GSWB can parse lambda expressions in accordance with a GSWB-internal semantic fragment, supporting alpha- and beta-conversion. This section describes how to use this semantic parser and provides guidelines for writing lambda expressions that can be parsed by it.

To activate the semantic parser, change the value of the variable `semParser` to `1` (instead of `0`) in the `xlerc` file. When this mode is active, unparsable input on

[10] This behaviour is currently investigated and might change in the future.

the meaning side will result in a parsing error. The current version of the semantic parser is completely independent of the glue side, which means that type restrictions need to be manually added. Furthermore, eta-conversion is not possible. This may change in the future.

The semantic parser parses lambda expressions and first-order logic terms. First-order predicates are encoded in the classic prefix notation. There is no convention with respect to the casing of predicates or constants, thus, the FOL terms in (27) all express a *liking* relation between two constants.

(27) a. `like(mary,semantics)`
b. `LIKE(mary,semantics)`
c. `like(MARY,SEMANTICS)`

Lambda expressions are introduced via a scope defining bracket and a slash, followed by the variable that the lambda operator binds. Variables require a type declaration to be distinguished from constants. This is done by using an underscore and a type as specified at the beginning of Section 3. Bound occurrences of a variable should not be typed again. The scope of the lambda function is separated from the binder via a dot. It can be any kind of well-formed (lambda) expression. See (28) for some examples of lambda expressions.

(28) a. `[/x_e.sleep(x)]`
b. `[/x_e.[/w_s.sleep(x,w)]]`
c. `[/P_<e,t>.[/Q_<e,t>.[/x_e.(P(x) & Q(x))]]]`

The basic **logic operators** $\wedge$, $\vee$ and $\rightarrow$ can be used as infix operators (see (29-a)–(29-c)), although their scope has to be defined via brackets or parentheses. Brackets may indicate operator and quantifier scope simultaneously (see (29-d)). Other operators must be encoded as FOL predicates in prefix notation (see (29-e)).

(29) a. Logical 'and' ($\wedge$): `(P(x) & Q(x))`
b. Logical 'or' ($\vee$): `(P(x) v Q(x))`
c. Logical 'implication' ($\rightarrow$): `(P(x) -> Q(x))`
d. Variant with brackets: `Ex_e[P(x) & Q(x)]`
e. Prefix notation: `equals(x,y)`

Quantifiers are introduced via the upper-case letters `A` and `E`, and the typed variable that they bind. The scope is defined via brackets as shown in (30).[11]

(30) a. `Ex_e[dog(x) & bark(x)]`
b. `Ax_e[cat(x) -> sleep(x)]`

Functional application steps such as in the semantic terms for quantifiers are determined contextually. Consider `P(x)` and `Q(x)` in example (31). The P and Q

[11] Since `A` and `E` are reserved for quantifiers, these letters should not be used to encode other terms, e.g., variables, or constants.

variables over predicates followed by the x variable as an argument are translated as functional application steps (apply P/Q to x), rather than as a one-place predicate with a bound variable ($P(x)$).

(31) `[/P_<e,t>.[/Q_<e,t>.Ex_e[P(x) & Q(x)]]]`

Abstraction is handled in the same way as in the concatenation mode: by adding a lambda binder to semantic formula, that is represented in terms of the λ symbol.

3.2.3 Prolog mode for external semantic representations

The third mode supported by the GSWB is the Prolog mode, which shows how the GSWB can be made compatible with other semantic resources. It can be activated by setting the `semParser` value to `2` in the `xlerc` file. This mode implements an alternative string encoding of semantic objects based on the system presented in Blackburn and Bos (2006). Using this system means that all constants are expressed in terms of lower-case letters and all variables are encoded as (starting with) upper-case letters. Functional application is expressed in terms of the two-place predicate `app/2`, where the first argument is the functor and the second argument is the argument. Lambda expressions, and, thus, lambda abstraction, are introduced by wrapping a term in the two-place predicate `lam/2`. The first argument denotes the variable that is bound by the lambda and the second argument of `lam/2` denotes the body of the function. (32-a) is an example of a lambda expression in Prolog notation. This corresponds to the functional application shown in (32-b). In the complex argument of this formula, *x* is combined with $\lambda v.bone(v)$, which is visually indicated as a function in terms of the square brackets. The variable *x* is then abstracted over by adding λx to combine with the quantifier.

(32) a. `app(lam(R,lam(S,every(Y,imp(app(R,Y),app(S,Y))))),`
`lam(X,app(lam(V,bone(V)),X)))`

b. $\lambda R.\lambda S.\forall y[R(y) \rightarrow S(y)](\lambda x.[\lambda v.bone(v)](x))$

4 Prolog rewrite component

The Prolog rewrite component takes the Prolog output of an XLE parse[12] as input and translates it into a set of premises based on the specifications introduced in the previous section. It does not rely on any particular assumptions about where the GLUE attributes must appear; GLUE attributes and their values are a part of f-structure in our sample grammar `glue-basic.lfg`, while our sample grammar `glue-basic-semstr.lfg` places them at s-structure. Indeed, the system works even if some GLUE attributes appear at f-structure, and others appear in other structures. It is also not necessary for the meaning constructors to be distributed in any particular way in the structure in which they appear; the system simply gathers

[12] https://ling.sprachwiss.uni-konstanz.de/pages/xle/doc/xle.html#Prolog_Output

up all members of every GLUE set in the input representation, rewrites them into the standard format, and passes the resulting set of standard-format meaning constructors to the prover. Thus, decisions about which structure hosts the GLUE attribute and its values should be made on the basis of linguistic considerations, and are not determined by properties of the implementation. Attributes other than the GLUE attributes and their values are ignored and discarded by the rewrite component.

Each element of the GLUE set provides one premise: as explained in Section 2 for the embedded encoding format of premises, the value of MEANING provides the semantic side, while RESOURCE, TYPE, and the ARG1…N attributes provide the glue side.[13]

The values of the RESOURCE attributes are instantiated to the numeric labels provided by XLE. Because the numeric indexing for semantic forms in the Prolog output format is independent of the numeric indexing for other structures (for example, there may be an f-structure with numeric index 1 and also a semantic form with numeric index 1 in the same f-structure), the numeric index of a semantic form is additionally prefixed with an S, e.g., S1, to ensure uniqueness of indices. As described in Section 2.1, the FORALL attribute is used to encode linear quantification. Different quantified variables are distinguished by combining the label F with the unique f-structure index.

(33) a. `AF11_t.((s1_e -o s1_t) -o ((4_e -o F11_t) -o F11_t))`

b. $\forall F11_t.((s1_e \multimap s1_t) \multimap ((4_e \multimap F11_t) \multimap F11_t))$

c. 4:
$$\left[\begin{array}{ll}\text{PRED} & 1\\ \text{GLUE} & \left\{\left[\begin{array}{ll}\text{MEANING} & \textit{every}\\ \text{FORALL} & \texttt{F11}\\ \text{ARG1} & \left[\begin{array}{ll}\text{ARG1} & \left[\begin{array}{ll}\text{RESOURCE} & 1\\ \text{TYPE} & e\end{array}\right]\\ \text{RESOURCE} & 1\\ \text{TYPE} & t\end{array}\right]\\ \text{ARG2} & \left[\begin{array}{ll}\text{ARG1} & \left[\begin{array}{ll}\text{RESOURCE} & 4\\ \text{TYPE} & e\end{array}\right]\\ \text{RESOURCE} & \texttt{F11}\\ \text{TYPE} & t\end{array}\right]\\ \text{RESOURCE} & \texttt{F11}\\ \text{TYPE} & t\end{array}\right]\right\}\end{array}\right]$$

Example (33-a) shows the output produced by the rewrite component for a gener-

[13] In fact, only the attributes MEANING, RESOURCE, TYPE, and FORALL have a special status in the embedded encoding format. All other attributes are assumed to represent arguments, which are consumed according to alphabetical order. It would also be possible to use A, B, C; A1, A2, A3; or any other alphabetically ordered series of attributes for arguments. It is not possible to substitute other names for the special attributes MEANING, RESOURCE, TYPE, FORALL when the embedded encoding format is used.

alized quantifier encoded in AVM format as (33-c) (as discussed in example (8)), corresponding to the standard format meaning constructor in (33-b). All of the conventions discussed above are illustrated in (33-a): `F11` is bound by a universal quantifier, `s1` refers to the semantic form whose index is 1, and `4` refers to the f-structure whose index is 4.

5 Illustrating the flexibility of the system

In this section, we present a variety of different modifications of the XLE+Glue system, including the possibility to use different semantic formalisms, as well as alternative encodings of glue premises in XLE. We also show how additional (semantic) resources can be added to the pipeline. Through this, we demonstrate how to enhance the functionality and coverage of the system.

5.1 Different semantic representations

5.1.1 Event semantics (with semantic parser)

While `glue-basic-semparser.lfg` is the sample grammar using the semantic parser (see Section 3.2.2), `glue-basic-semparser_ND.lfg` is its modified version using Neo-Davidsonian event semantics (Parsons 1990).

Rather than using predicates with variable arity (depending on the number of arguments of the predicate), in event semantics the predicate has only one argument, the event variable, while the dependents of the predicate are related to it using separate predicates whose names correspond to the semantic role of the given dependent (such as *agent*, *theme*, etc.).

The examples below provide semantic representations of the running example "Kim smiled" produced by the grammars `glue-basic-semparser.lfg` and `glue-basic-semparser_ND.lfg`, respectively: in (34-a) the predicate $smile$ has one argument (Kim), while in (34-b) the only argument of $smile$ is the event variable (here: z), while Kim is related to the event z using the $agent$ predicate (Kim is the $agent$ of z).

(34) a. `smile(Kim)`
b. `exists([λz_v.and(smile(z),agent(z,Kim))])`

5.1.2 DRT semantics (with Prolog mode)

In Section 3.2.3, we demonstrated that the GSWB supports Prolog-style encoding of semantic formulas as output. Using this mode, we provide a DRT-mode in XLE+Glue to illustrate the possibility to interact with different semantic resources. To activate the DRT-mode, set the following values of variables in the `xlerc` file: `processDRT` to `1`, `semParser` to `2` (Prolog mode on).

The DRT-mode makes use of the Boxer DRT system (Bos 2008, 2015, Blackburn and Bos 2006) optimized to interpret λ-DRT formulas as described in Gotham

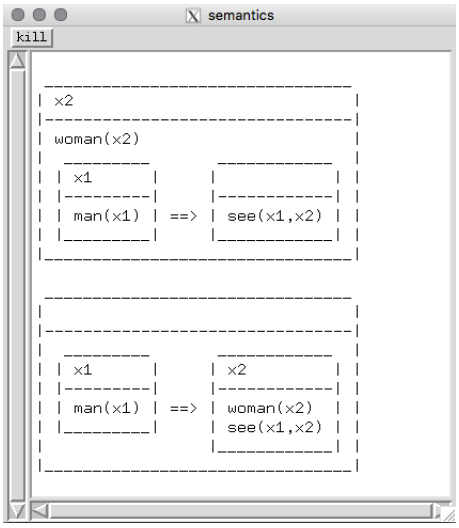

Figure 2: Boxer-style DRT representation: *Every man saw a woman*

and Haug (2018).[14] To include those two components, we extended the λ-DRT system with some simple wrapper code to execute it from within XLE+Glue. Thanks to the Prolog-style encoding option of the GSWB, the lambda DRT component can directly process the GSWB's output and produce a graphical Boxer-style representation of the Prolog term that is presented in the output window of XLE+Glue. This is shown in Figure 2.

5.2 Meaning constructors encoded as strings

Section 2 describes the f-structure encoding of meaning constructors as AVMs, using the attributes RESOURCE, TYPE, MEANING, and ARG1…ARGN. In this encoding, the embedding in the AVM representation reflects the structure of the linear logic expression that is encoded, since material on the left-hand side of a linear implication is represented as the f-structure value of an attribute such as ARG1.

This section describes an alternative encoding: the string-based, flat encoding. In this encoding, the substrings of the meaning constructor are encoded as values of the attributes in a single AVM, which are concatenated together to produce the input to the prover. This is in some ways a simpler encoding, since it does not require the construction of a complex AVM to reflect the structure of the linear logic term. However, it requires a detailed understanding of the input format required by the GSWB prover, and it is also easier to make mistakes in the encoding, which can make it harder to use.

The flat encoding is illustrated in (35), where 1 is the label assigned by XLE to the outermost f-structure:

[14] We thank Johan Bos and Matthew Gotham for making their λ-DRT tools available to us.

(35) a. `Kim:1_e`

b.
$$1:\begin{bmatrix} \text{PRED} & \text{`KIM'} \\ \text{GLUE} & \left\{ \begin{bmatrix} \text{A1} & \text{KIM} \\ \text{A2} & : \\ \text{A3} & 1 \\ \text{A4} & \_ \\ \text{A5} & \text{e} \end{bmatrix} \right\} \end{bmatrix}$$

c.
Kim	:	1	_	e
A1	A2	A3	A4	A5

In this encoding, the substrings of the meaning constructor input to the GSWB prover are encoded directly. By convention, this encoding uses the attributes A1,A2,..., but in fact any attributes can be used except for the special attribute MEANING, which indicates to the transfer component that the embedded encoding is being used. It is possible to have meaning constructors encoded in the string-based format and the standard embedded format coexisting in the same f-structure, or even in the same GLUE set.

In the transfer component using the flat encoding, the attributes are sorted into alphabetical order, and the substrings of the meaning constructor are concatenated according to that ordering. (35-c) shows the correspondence between the attributes of the f-structure in (35-b) and the resulting meaning constructor. A cautionary note: if there are 10 or more meaning constructor substrings encoded via attributes `A1...A10...` in an AVM, the attribute `A10` will sort alphabetically between the attributes `A1` and `A2`; in that case, therefore, the single-digit attributes should be prefixed with 0 (`A01,A02,...A10,A11,...`) to ensure that the values of the attributes are concatenated in the correct order.

In the sample grammar `glue-basic-flat-encoding.lfg`, the lexical entry for *Kim* shown in (12) calls the `PROPERNOUN` template in (13), which in turn calls the template `GLUE-REL0-MC` defined in (14). In the string-based, flat encoding, `GLUE-REL0-MC` is defined as follows:

(36)
```
GLUE-REL0-MC(R TY M) =  (%mc A1) = M
                        (%mc A2) = `:
                        (%mc A3) = R
                        (%mc A4) = _
                        (%mc A5) = TY
                        %mc $ (R GLUE).
```

The value of the attribute `A1` is the meaning term, which is the first component of the meaning constructor for *Kim*. The value of `A2` is the colon separating the meaning side of the meaning constructor from the glue side, which must be quoted with a backquote. The value of `A3` is the f-structure for *Kim*, the value of `A4` is the underscore separating the f-structure from its type, and the value of `A5` is its type, as specified by `TY` in the template call to `GLUE-REL0-MC`. Further examples can be found in the sample grammar `glue-basic-flat-encoding.lfg`.

6 Conclusion

In this paper we presented the XLE+Glue system, which provides an interface between XLE and the GSWB with the goal to contribute to the ongoing reinvigoration of computational Glue semantics.

We paid particular attention to the encoding of glue formulas within XLE grammars for which we have presented several alternatives to show that the system can deal flexibly with different ways of tackling the issue. In particular, we presented a novel encoding of glue premises in terms of AVMs, where linear logic terms are encoded in a hierarchical structure. However, we also demonstrated that an alternative flat encoding is possible. Furthermore, we showed how the XLE+Glue system can be made compatible with different semantic formalisms as well as additional semantic resources.

Although this paper presents the XLE+Glue resource in terms of a co-descriptive approach to Glue semantics in the sample grammars, the encodings presented in this paper are in fact agnostic with respect to ideas about the syntax/semantics interface and the choice of semantic formalism. This means that we provide a flexible system for computational Glue semantics that can be optimized to cater for the needs of an individual grammar developer and the needs of the given grammar theory that the developer wants to implement.

This paper provides an introduction to the XLE+Glue system and describes some of the more important technical details. However, we strongly encourage the reader to consult the manual provided in the XLE+Glue GitHub repository (see footnote 1) before starting to work with the system, as it describes its technical underpinnings in more detail. For a quick start guide to experiment with the system, see the README file in the repository which provides a minimal description for setting up the system. Both of these resources will be continuously updated as new features are introduced to XLE+Glue to make the system a long-lasting resource for computational linguists.

References

Blackburn, Patrick and Bos, Johan. 2006. Working with discourse representation theory. *An Advanced Course in Computational Semantics* http://www.let.rug.nl/bos/comsem/book2.html.

Bos, Johan. 2008. Wide-Coverage Semantic Analysis with Boxer. In *Semantics in Text Processing 2008 conference proceedings*, pages 277–286.

Bos, Johan. 2015. Open-Domain Semantic Parsing with Boxer. In Beáta Megyesi (ed.), *Proceedings of the 20th Nordic Conference of Computational Linguistics (NODALIDA 2015)*, pages 301–304.

Crouch, Dick. 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, Dick, Dalrymple, Mary, Kaplan, Ron, King, Tracy, Maxwell, John and Newman, Paula. 2017. *XLE Documentation*. Palo Alto Research Center, https://ling.sprachwiss.uni-konstanz.de/pages/xle/doc/xle_toc.html.

Crouch, Dick and King, Tracy Holloway. 2006. Semantics via F-Structure Rewriting. In Miriam Butt and Tracy Holloway King (eds.), *Proceedings of the LFG06 Conference*, pages 145–165, Stanford, CA: CSLI Publications.

Dalrymple, Mary. 1999. *Semantics and Syntax in Lexical Functional Grammar: The Resource Logic Approach*. MIT Press.

Gotham, Matthew and Haug, Dag Trygve Truslew. 2018. Glue Semantics for Universal Dependencies. In Miriam Butt and Tracy Holloway King (eds.), *Proceedings of the LFG'18 Conference, University of Vienna*, pages 208–226, Stanford, CA: CSLI Publications.

Meßmer, Moritz and Zymla, Mark-Matthias. 2018. The Glue Semantics Workbench: A Modular Toolkit for Exploring Linear Logic and Glue Semantics. In Miriam Butt and Tracy Holloway King (eds.), *Proceedings of the LFG'18 Conference, University of Vienna*, pages 249–263, Stanford, CA: CSLI Publications.

Parsons, Terence. 1990. *Events in the Semantics of English: A Study in Subatomic Semantics*. Cambridge, MA: The MIT Press.