Universal Dependencies to Logical Forms with Negation Scope

Federico Fancellu **Siva Reddy** **Adam Lopez** **Bonnie Webber**
ILCC, School of Informatics, University of Edinburgh
f.fancellu@sms.ed.ac.uk, siva.reddy@ed.ac.uk, {alopez, bonnie}@inf.ed.ac.uk

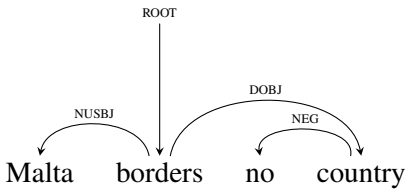
Abstract

Many language technology applications would benefit from the ability to represent negation and its scope on top of widely-used linguistic resources. In this paper, we investigate the possibility of obtaining a first-order logic representation with negation scope marked using *Universal Dependencies*. To do so, we enhance *UDepLambda*, a framework that converts dependency graphs to logical forms. The resulting *UDepLambda¬* is able to handle phenomena related to scope by means of an higher-order type theory, relevant not only to negation but also to universal quantification and other complex semantic phenomena. The initial conversion we did for English is promising, in that one can represent the scope of negation also in the presence of more complex phenomena such as universal quantifiers.

1 Introduction

Amongst the different challenges around the topic of negation, detecting and representing its scope is one that has been extensively researched in different sub-fields of NLP (e.g. Information Extraction (Velldal et al., 2012; Fancellu et al., 2016)). In particular, recent work have acknowledged the value of representing the scope of negation on top of existing linguistic resources (e.g. AMR – Bos (2016)). Manually annotating the scope of negation is however a time-consuming process, requiring annotators to have some expertise of formal semantics.

Our solution to this problem is to automatically convert an available representation that captures negation into a framework that allows a rich variety of semantic phenomena to be represented, including scope. That is, given an input sentence, we show how its *universal dependency* (UD) parse can be converted into a representation in first-order logic (FOL) with lambda terms that captures both predicate–argument relations and scope.



(a) UD Dependency Tree

$$\lambda e.\exists x \exists y.borders(e) \wedge country(x) \wedge no(x) \wedge Malta(y) \wedge arg1(e,y) \wedge arg2(e,x)$$

(b) UDepLambda Logical Form

$$\forall x.country(x) \rightarrow \neg \exists e \exists y.borders(e) \wedge Malta(y) \wedge arg1(e,y) \wedge arg2(e,x)$$

(c) Desired Logical Form

Figure 1: The dependency tree for *'Malta borders no country'* and its logical forms

Our approach is based on *UDepLambda* (Reddy et al., 2017; Reddy et al., 2016), a constraint framework that converts dependency graphs into logical forms, by reducing the lambda expressions assigned to the dependency edges using the lambda expressions of the connected head and child nodes. The edge labels in the input UD graph are only edited minimally so to yield a more fine-grained description on the phenomena they describe, while lexical information is used only for a very restricted class of lexical items, such as negation cues. A FOL representation of the entire input graph can be then obtained by traversing the edges in a given order and combining their semantics.

However, in its original formulation, UDe-

Proceedings of the Workshop Computational Semantics Beyond Events and Roles (SemBEaR), pages 22–32,
Valencia, Spain, April 4, 2017.

pLambda does not handle either universal quantifiers or other scope phenomena. For example, the sentence 'Malta borders no country' has the UD graph shown in Figure 1(a). When compared to the correct representation given in Figure 1(c), the UDepLambda output shown in Figure 1(b) shows the absence of universal quantification, which in turn leads negation scope to be misrepresented.

For this reason, we set the foundation of **UDepLambda¬** (UDepLambda-*not*), an enhanced version of the original framework, whose type theory allows us to jointly handle negation and universal quantification. Moreover, unlike its predecessor, the logical forms are based on the one used in the 'Groeningen Meaning Bank' (GMB; (Basile et al., 2012a)), so to allow future comparison to a manually annotated semantic bank.

Although the present work shows the conversion process for English, given that the edge labels are *universal*, our framework could be used to explore the problem of representing the scope of negation in the other 40+ languages universal dependencies are available in. This could also address the problem that all existing resources to represent negation scope as a logical form are limited to English (e.g. GMB and 'DeepBank' (Flickinger et al., 2012)) or only to a few other languages (e.g. 'The Spanish Resource Grammar' (Marimon, 2010)).

In the reminder of this paper, after introducing the formalism we will be working in (§3), we will work the theory behind some of the conversion rules, from basic verbal negation to some of the more complex phenomena related to negation scope, such as the determiner 'no'(§4.1), the interaction between the negation operator and the universal classifier (§4.2) and non-adverbial or lexicalized negation cues such as 'nobody', 'nothing' and 'nowhere' (§4.3). Limitations, where present, will be highlighted.

Contribution. The main contribution of the paper is *UDepLambda¬*, a UD-to-FOL conversion framework, whose type theory is able to handle scope related phenomena, which we show here in the case of negation.

Future work. *UDepLambda¬* can serve as a basis for further extensions that could apply to other complex semantic phenomena and be learned automatically, given the link to a manually annotated semantic bank.

2 Related work

Available resources that contain a representation of negation scope can be divided in two types: 1) those that represent negation as a FOL (or FOL-translatable) representation (e.g. GMB, 'DeepBank'), where systems built using these resources are concerned with correctly *representing* FOL variables and predicates in the scope of negation; and 2) those that try to ground negation at a string-level, where both the negation operator and scope are defined as spans of text (Vincze et al., 2008; Morante and Daelemans, 2012). Systems trained on these resources are then concerned with *detecting* these spans of text.

Resources in 1) are limited in that they are only available in English or for a small number of languages. Moreover no attempt has been made to connect them to more widely-used, cross-linguistic frameworks.

On the other hand, grounding a semantic phenomenon to a string-level leads to inevitable simplification. For instance, the interaction between the negation operator and the universal quantifier (e.g. 'Not every staff member is British' vs. 'None of the staff members are British'), along a formal representation that would allow for inference operations is lost. Furthermore, each corpus is tailored to different applications, making annotation styles across corpora incompatible. Nonetheless these resources have been widely used in the field of Information Extraction and in particular in the Bio-Medical domain.

Finally, it is also worth mentioning that there has been some attempts to use formal semantic representations to detect scope at a string level. Packard et al. (2014) used hand-crafted heuristics to traverse the MRS (Minimal Recursion Semantics) structures of negative sentences to then detect which words were in the scope of negation and which were not. Basile et al. (2012b) tried instead to first transform a DRS (Discourse Representation Structure) into graph form and then align this to strings. Whilst the MRS-based system outperformed previous work, mainly due to the fact that MRS structures are closely related to the surface realization, the DRT-based approach performed worse than most systems, mostly given to the fact that the formalism is not easily translatable into a theory-neutral surface-oriented representation.

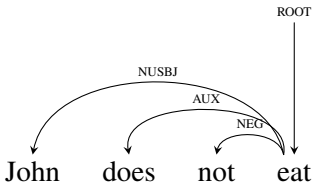


Figure 2: Dependency graph for the sentence 'John does not eat'

3 UDepLambda¬

We introduce here the foundations of *UDepLambda¬*, an enhancement to the *UDepLambda* framework to convert a UD graph into its correspondent logical form. As its predecessor, the conversion takes place in four different steps: *enhancement*, *binarization*, *substitution* and *composition*. Whereas binarization and composition are the same as *UDepLambda*, substitution differs in:

- using a higher order type-theory to deal with universal quantification, which can interact with other scope operator such as negation;
- using FOL expressions based on those used in the Groeningen Meaning Bank (GMB), so as to link to a manually–annotated semantic bank which can be leveraged for future work.[1]

The details of the four steps are as follows:

Enhancement. In this step, we first convert a dependency tree to a dependency graph using existing enhancements in UDepLambda. The enhanced dependency labels are represented in red color. In future, we will replace this step with existing enhancements (Schuster and Manning, 2016).

Binarization. The dependency graph is mapped to a LISP-style s-expression, where the order of the edge traversal is specified. For instance, the expression (*nsubj* (*aux* (*neg* eat not) does) John) indicates that the semantic representation of the sentence in Figure (2) is derived by composing the semantics of the edge *nsubj* with the logic form of 'John' and of the phrase 'does not eat'. The semantics of the phrase 'does not eat' is in turn derived by composing the edge *aux* with the phrase 'not eat' and the auxiliary 'does'. Finally 'not' and 'eat' are composed along the edge *neg*.

The order of traversal follows an *obliqueness hierarchy* which defines a strict ordering of the modifiers of a given head traversed during composition. This hierarchy is reminiscent of bottom-up traversal in a binarized constituency tree (where for instance the direct object is always visited before the subject). Furthermore, for a head to be further composed, all its modifiers needs to be composed first. In the sentence in Figure (2), this hierarchy is defined as *neg* $>$ *aux* $>$ *nusbj*, where the semantics of the subject can be applied only when the other modifiers to the verb-head have been already composed.

Substitution. The substitution step assigns a lambda expression to each edge and vertex (i.e. word) in the graph. *The lambda expressions of the edges are manually crafted to match the semantics of the edge labels while no assumption is made on the semantics of the word-vertices which are always introduced as existentially bound variables*. This allows us*not* to rely for most part on any language-specific lexical information. These expressions follows recent work on semantic compositionality of complex phenomena in event semantics (Champollion, 2011). In doing this, we generalize our type theory as follows:

- Each word-vertex is assigned a semantic type $\langle\langle v, t\rangle, t\rangle$ or $\langle\langle v, t\rangle, t\rangle$ (here shortened in $\langle vt, t\rangle$), where v stands for either a paired variable of type $\mathrm{Event} \times \mathrm{Individual}$. This is in contrast with the type assigned to words in the original UDepLambda $\langle v, t\rangle$. The result of this type-raising operation is clear when we compare the following lambda expressions:

  UDepLambda: $\lambda x.man(x_a)$
  UDepLambda¬: $\lambda f.\exists x.man(x_a) \wedge f(x)$

  where the 'handle' f allows for complex types to be added inside another lambda expression.
  Following the GMB, proper nouns are treated like indefinite nouns, being linked to a existentially-bound variable (e.g. John := $\lambda f.\exists x.named(x_a, John, PER) \wedge f(x)$).

[1] The current study ignores certain aspects of Discourse Representation Theory (Kamp et al., 2011) on which the GMB is based, which are secondary to the issues we are focussed on.

- Each edge is assigned the semantic type $\langle\langle vt,t\rangle,\langle\langle vt,t\rangle,\langle vt,t\rangle\rangle\rangle$ where we combine a generalized quantifier over the parent word (P) with the one over the child word (Q) to return another generalized quantifier (f). For instance, when reducing the sub-expression (*nsubj* eat John), we first reduce the parent vertex 'eat' (P) and then the child vertex 'John'(Q) using the semantics of the subject ('Actor' in the GMB).

  nsubj:= $\lambda P.\lambda Q.\lambda f.P(\lambda x.f(x) \wedge Q(\lambda y.$ $Actor(x_e,y_a)))$

  When compared to the original UDepLambda expression (of type $\langle\langle v,t\rangle,\langle\langle v,t\rangle,\langle v,t\rangle\rangle\rangle$):

  $\lambda f.\lambda g.\lambda x.\exists y.f(x_e) \wedge g(y_a) \wedge arg1(x_e,y_a)$

  unlike its predecessor, UDepLambda¬ allows for nested dependencies between parent and child node which is necessary to model scope phenomena.

- In cases such as the sub-expression (*neg* 'John does eat' not), the edge label *neg* and the word 'not' carry the exact same semantics (i.e. the negation operator ¬). For these *functional words* we try to define semantics on the dependency edges only rather than on the word. As shown below, reducing Q does not impact the semantic composition of the edge *neg*:

  neg:= $\lambda P.\lambda Q.\lambda f.\neg P(\lambda x.f(x))$
  not:= $\lambda f.TRUE$

Composition. The lambda expressions are reduced by following the traversal order decided during the *binarization* step. Let's exemplify the composition step by showing at the same time how **simple verbal negation** composes semantically, where the input s-expression is (*neg* (*aux* (*nsubj* eat John) does) not). The substitution step assigns vertices and edges the following semantics:

'eat' := $\lambda f.\exists x.eat(x_e) \wedge f(x)$
'not' := $\lambda f.TRUE$
'John' := $\lambda f.\exists x.named(x_a,John,PER) \wedge f(x)$
'does' := $\lambda f.TRUE$
nsubj:= $\lambda P.\lambda Q.\lambda f.P(\lambda x.f(x) \wedge Q(\lambda y.Actor(x_e,y_a)))$
aux := $\lambda P.\lambda Q.\lambda f.P(\lambda x.f(x))$
neg:= $\lambda P.\lambda Q.\lambda f.\neg P(\lambda x.f(x))$
ex-closure:= $\lambda x.TRUE$

where the subscripts e and a stands for the event-type and the individual-type existential variable respectively. As for the edge *neg*, the child of a *aux* edge is ignored because not contributing to the overall semantics of the sentence.[2] We start by reducing (*neg* eat not), where P is the parent vertex 'eat' and Q the child vertex 'not'. This yields the expression:[3]

$$\lambda f.\neg\exists x.eat(x_e) \wedge f(x)$$

We then use this logic form to first reduce the lambda expression on the edge *aux*, which outputs the same input representation, and then compose this with the semantics of the edge *nsubj*. The final representation of the sentence (after we apply existential closure) is as follows:

$$\neg\exists x.\exists y.eat(x_e) \wedge named(y_a,John,PER) \wedge Actor(x_e,y_a)$$

Given the resulting logical form we consider as part of negation scope all the material under the negation operator ¬.

4 Analysis of negative constructions

4.1 The quantifier 'no'

Let's consider the sentence 'No man came' along with its dependency trees and logical form, shown in Figure 3.

As shown in Figure 3(b), one shortcoming of the original UDepLambda is that it doesn't cover universal quantification. However, even if we were to assign any of the following lambda expressions containing material implication to the *neg* edge connecting parent-λf ('man') and child-λg ('no'):

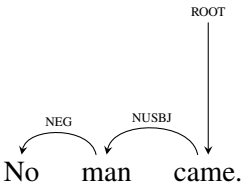
$$?\lambda f.\lambda g.\lambda x.f(x) \rightarrow \neg f(x)$$
$$?\lambda f.\lambda g.\lambda x.f(x) \rightarrow g(x)$$

the resulting expressions would have no means of later accommodating the event 'came' in the consequent of the material implication:

$$*\lambda x.man(x) \rightarrow \neg man(x)$$
$$*\lambda x.man(x) \rightarrow no(x)$$

[2] The present work does not consider the semantics of time the word 'does' might contribute to.

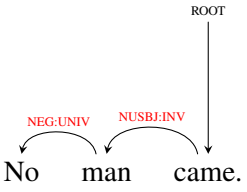
[3] Step-by-step derivations are shown in Appendix A.



(a) Original UD Dependency Tree

$$\exists x.came(x_e) \wedge \neg\exists y.man(y_a) \wedge arg1(x_e, y_a)$$

(b) UDepLambda Logical Form



(c) Enhanced UD Dependency Tree

$$\forall y.man(y_a) \rightarrow \neg\exists x.came(x_e) \wedge Actor(x_e, y_a)$$

(d) Desired Logical Form

Figure 3: The dependency trees for *'No man came'* (top: original UD tree; bottom: enhanced UD tree) and its logical forms

The higher-order type theory of UDepLambda¬ solves this problem by ensuring that a) there is a universal quantified variable along with material implication and b) the entity bound to it ($man(x)$) is introduced only in the antecedent, whereas the negated event (along with other arguments) only in the consequent. The lambda expression assigned to the *neg* edge is the following

$$\lambda P.\lambda Q.\lambda f.\forall x.(P(\lambda y.EQ(x, y)) \rightarrow \neg f(x))$$

where f allows to leave a 'handle' for the event 'came' to be further composed in the subsequent only, whereas the two-place function EQ(x,y) as argument of P binds the word in the parent node with the universally quantified variable x.

It is worth mentioning at this point that although the universal quantifier 'no' is parsed as depending from an edge *neg*, it possesses a semantics that distinguishes it from other negative adverbs such as 'not' or 'never', in the fact that they bind their head to a universally quantifiable variable. In these cases we also *enhance* the label on the dependency edge to reflect this more fine-grained distinction. In the presence of 'no' the *neg* edge becomes **neg:univ** if its child vertex is a universal quantifier. This edit operation relies on having a list of lexical items for both universal quantifiers and negation cues in a language, which is easily obtainable given that these items form a small, closed class.

A further edit operation is needed to make sure that the quantifier always outscopes the negation operator; to do so, we modify the semantics of the edge that connects the head of the edge *neg:univ* ('man') with its parent ('came'), *nsubj*, by inverting the order of the Q and P, so that the former outscopes the latter. We call this enhanced edge an '*edge-name:*inv.' edge. Compared to *nsubj*, the semantics of **nsubj:inv** would be as follows:

nsubj :=
$$\lambda P.\lambda Q.\lambda f.P(\lambda x.f(x) \wedge Q(\lambda y.Actor(x_e, y_a)))$$
nsubj-inv :=
$$\lambda P.\lambda Q.\lambda f.Q(\lambda y.P(\lambda x.Actor(x_e, y_a) \wedge f(x)))$$

Using the edited input UD graph, the hierarchy we follow during composition is *neg:univ* > *nsubj:inv* to yield the s-expression (*nsubj:inv* (*neg:univ* no man) came). Given the following input semantics:

man:= $\lambda f.\exists x.man(x_a) \wedge f(x)$
came:= $\lambda f.\exists x.came(x_e) \wedge f(x)$
neg:univ:=
$$\lambda P.\lambda Q.\lambda f.\forall x.(P(\lambda y.EQ(x, y)) \rightarrow \neg f(x))$$
nsubj:inv :=
$$\lambda P.\lambda Q.\lambda f.Q(\lambda y.P(\lambda x.Actor(x_e, y_a) \wedge f(x)))$$

we first reduce the lambda expression on the edge *neg:univ.* to yield the expression $\lambda f.\forall x.(man(x_a) \rightarrow \neg f(x))$ and then combine it along the edge *nsubj:inv* to yield the following representation:

$$\forall y.(man(y_a) \rightarrow \neg\exists x.came(x_e) \wedge Actor(x_e, y_a))$$

, where the scope of negation is correctly converted as inside the universal quantifier.

Inverting the order of the parent and child nodes in the semantics of the *:inv.* edge always allows to represent the universally quantified element as outscoping the event it depends on. At the same time, all other arguments and modifiers of the parent event will always compose inside the consequent. This applies to our initial example in Figure 1, where composing the s-expression (*dobj:inv.* borders 'no country') to yield the expression:

$$\lambda f.\forall y.(country(y_a) \rightarrow \neg\exists x.borders(x_e) \wedge Theme(x_e, y_a) \wedge f(x))$$

, makes sure that further material can only be added in place of $f(e)$, which is inside the scope of $\neg$, in turn in the scope of $\forall$. So when composing the semantics of the subject 'Malta' (:= $\lambda f.\exists x.named(x_a, Malta, ORG) \wedge f(x)$), the universal will still have wide-scope, as shown below:

$$\forall y.(country(y_a) \rightarrow \neg\exists x.\exists z.named(z_a, Malta, PER) \wedge borders(x_e z) \wedge Theme(x_e, y_a) \wedge Actor(x_e, z_a))$$

4.2 Negation and universal quantifier

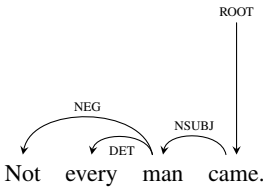
Alongside quantifiers inherently expressing negation, as the one shown in the previous section, another challenging scope representation arises during the interaction between a negation cue and a non-negative universal quantifier, such 'every'. Let's take as example the sentences 'Not every man came', shown in Figure 4 alongside its FOL representation.

If compared to the representation of the sentence 'No man came', where the universal quantifier outscopes the negation operator, the construction 'not every' yields the opposite interaction where the quantifier is in the scope of $\neg$ (correspondent to the meaning 'there exists some man who came').

As shown in the previous section and here in Figure 4(b), UDepLambda cannot deal with such constructions, yielding a meaning where there exists and event but there doesn't exists the entity that performs it. On the other hand, UDepLambda$\neg$ can easily derive the correct representation by applying the same edits to the UD graph shown in the previous section. First, we enhance the *det* edge to become a more fine-grained *det:univ* in the presence of the child node 'every'. Second, we change *nsubj* into *nsubj-inv.*, since a universal quantifier is in its yield. The lambda expression assigned to the edge *det:univ* is as follows:

det:univ:=$\lambda P.\lambda Q.\lambda f.\forall x.(P(\lambda y.EQ(x, y)) \rightarrow f(x))$

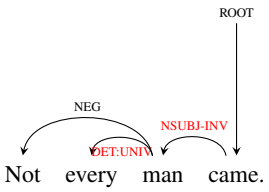
Once again, we deploy the usual bottom-up binarization hierarchy where all modifiers of a head need to be composed before the head itself can be used for further composition. In the case of 'not every...', we start from the modifiers 'every'



(a) UD tree

$$\lambda f.\lambda g.\lambda x.came(x_e) \wedge arg1(x_e, y_a) \wedge \neg\exists y.man(y_a) \wedge every(y_a)$$

(b) UDepLambda Logical Form



(c) Enhanced UD tree

$$\neg\forall y.(man(y_a) \rightarrow \exists x.came(x_e) \wedge Actor(x_e, y_a))$$

(d) Desidered Logical Form

(*nsubj-inv.* came (*neg* (*univ* man every) not))

(e) S-Expression

Figure 4: The sentence 'Not every man came' along with its dependency trees and logical forms

and 'not' and compose the edges following the order *det:univ* $\rangle$ *neg* so to make sure that negation operator $\neg$ outscopes the universal quantifier $\forall$. After the modifiers of the head 'man' are composed, we can then move on to compose the head itself with its governor node, the event 'came'. The *nsubj:inv.* edge ensures that the subject scopes over the event and not the other way around. Following this, we are able to obtain the final representation:

$$\neg\forall y.(man(y_a) \rightarrow \exists x.came(x_e) \wedge Actor(x_e, y_a))$$

4.3 Nobody/nothing/nowhere

As shown in Table 1, 'nobody', 'nothing' and 'nowhere' belong to that class of negation cues whose parent edge do not mark them as inherently expressing negation. However using an hand-crafted list of negation cues for English, we can detect and assign them the semantic representation $\lambda f.\neg\exists x.thing/person/location(x_a) \wedge f(x)$, where the negation operator scopes over an existentially bound entity.

Binarization and composition vary according to

	nobody	nothing	nowhere
nsubj	7	18	-
dobj	-	34	-
conj	-	8	-
nsubjpass	1	6	-
root	-	8	-
advmod	-	-	3
nmod	-	4	-
other	-	8	-
tot.	8	86	3

Table 1: Distribution of nobody, nothing and nowhere with their related dependency tags as they appear in the English UD corpus (McDonald et al., 2013)

whether these elements are arguments or adjuncts. If an argument, the scope of negation includes also the event, otherwise the latter is excluded. To this end, let's compare the sentences 'Nobody came' and 'John came with nothing', along with their dependency graphs and logic forms (Figure 5).

The argument 'nobody' in 'Nobody came' yields a scope reading where the negation operator scopes over the existential. To achieve such reading we once again convert the *nsubj* (or any argument edge for that matter) into a *nsubj:inv.* edge. This is reminiscent of how we handled universal quantification when we introduced the quantifier 'no', which is in fact integral part of such lexical elements (the semantics of '*no*-body came' can be in fact read as 'for all x such that x is a person that x did not come'). Also, the fact that the semantics of these elements is represented through an existential and not a universal bound variable is no problem since we are working under the equivalence $\forall x.P(x) \rightarrow \neg Q(x) \equiv \neg\exists x.P(x) \wedge Q(x)$.

Given the s-expression (*nsubj:inv.* came nobody) the composition is then as follows:

$$\neg\exists x.\exists y.person(y_a) \wedge f(x) \wedge Actor(x_e, y_a) \wedge came(x_e)$$

On the other hand, when the negated lexical element is embedded in an adjunct, as in 'with nothing', no enhancement of the original dependency edges takes place since we want to preserve negation scope inside the phrase (so to yield a reading where the event 'John came' did indeed take place). By substituting and combining the semantics of the s-expression (*nmod:with* came nothing), where the edge *nmod:with* is assigned the lambda expression $\lambda P.\lambda Q.\lambda f.P(\lambda x.f(x) \wedge Q(\lambda y.with(x_e, y_a)))$, we obtain the following logic form:

$$\lambda f.\exists x.came(x_e) \wedge f(x) \wedge \neg\exists y.(thing(y_a) \wedge with(x_e, y_a))$$

, where we can the scope of negation is limited to the propositional phrase. Given that the f is outside the scope of negation, further compositions (in the case along the edge *nsubj.*) will also compose outside it, yielding the correct form in Figure (5).

The only limitation we have observed so far concerns 'nowhere' (:= $\lambda f.\exists x.location(x_a) \wedge f(x)$) and the fact it is always associated with a dependency tag *advmod*. The tag *advmod* describes however the manner an action is carried out and has the logical form $\lambda P.\lambda Q.\lambda f.P(\lambda x.f(x) \wedge Q(\lambda y.Manner(x_e, y_a)))$. This is however different from how 'nowhere' is treated in the Groeningen Meaning Bank, where it is described as *where* and not *how* the event takes place. That is, our framework would assign a sentence like 'They got nowhere near the money' the logical form $\exists x.got(x_e) \wedge \neg\exists y.(location(y_a) \wedge Manner(x_e, y_a))$, whereas the one contained in the GMB is: $\exists x.got(x_e) \wedge \neg\exists y_a.(location(y_a) \wedge in(x_e, y_a))$

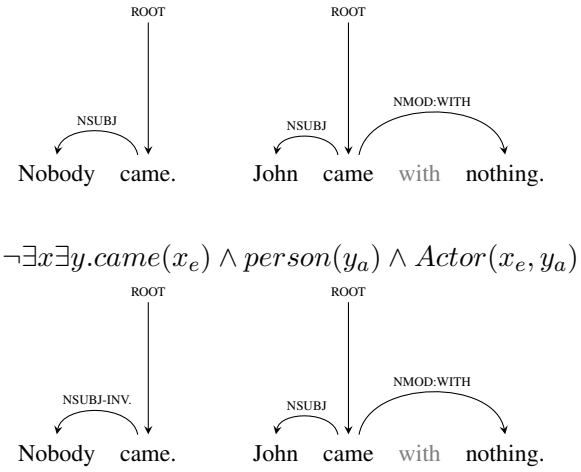


Figure 5: Dependency graphs and FOL representations for the sentences 'Nobody came' (above) vs. 'John came with nothing' (below).

4.4 Further remarks

Building on the principle of relying on dependency label information as much as possible while

using minimal lexical information, we showed that UDepLambda¬ is able to deal with complex phenomena involving negation scope. Although the theory is not shown in full here, these phenomena also include **discontinuous scope** and **affixal negation**, which are part of more recent corpora used to train systems for detecting negation scope at a *string level* (Morante and Daelemans, 2012).

The ability of dealing with discontinuous scope spans, such as in sentences like 'John screamed and did not laugh', where the subject 'John' and the predicate 'did not laugh' are part of the negation scope but 'screamed' is not, comes from the dependency graphs themselves, where we can recover the shared material by means of simple transformation heuristics.[4]

As for affixal negation (e.g. 'John is *im*patient'), one could use a similar heuristics as in the case of 'nobody/nothing/nowhere' where the lexical element is enhanced with the negation operator ('patient':= $\lambda f.\exists x.patient(x_e) \wedge f(x) \rightarrow \lambda f.\neg\exists x.patient(x_e) \wedge f(x)$). This relies again on bespoke list of words containing an affixal negation cue.

Given that UDepLambda¬ is based on dependency graphs, the primary limitations of our system are how certain phenomena are handled (or better *not* handled) by a dependency parse. This includes multi-word cues such as 'no way' and 'by no means' and the construction 'neither ... nor'.

5 Conclusion and future work

This paper addressed the problem of representing negation scope from universal dependencies by setting the foundations of *UDepLambda¬*, a conversion framework whose high-order type theory is able to deal with complex semantic phenomena related to scope. The conversion processes we presented show that it is possible to rely on dependency edges and additionally to minimal language-dependent lexical information to compose the semantics of negation scope. The fact that this formalism is able to correctly compose the scope for many complex phenomena related to negation scope is promising.

We are currently working on extending this work in two directions:

1. *Automatic framework evaluation*: given the conversion rules presented in this paper, we are planning to automatically convert the UD graphs for the sentences in the GMB so to compare the graph we automatically generate with a gold-standard representation. This would also to identify and quantify the errors of our framework.
2. *Automatic semantic parsing*: given the connection between this framework and the GMB, we would like to explore the possibility of learning the conversion automatically, so not to rely on an hand-crafted hierarchy to decide the order of edge traversal.

[4]We are considering substituting such heuristics with the *Enhanced++* version of the Stanford Dependencies (Schuster and Manning, 2016) where implicit relations between content words are made explicit by adding relations and augmenting relation names.

References

Valerio Basile, Johan Bos, Kilian Evang, and Noortje Venhuizen. 2012a. Developing a large semantically annotated corpus. In *LREC 2012, Eighth International Conference on Language Resources and Evaluation*.

Valerio Basile, Johan Bos, Kilian Evang, and Noortje Venhuizen. 2012b. Ugroningen: Negation detection with discourse representation structures. In *Proceedings of the First Joint Conference on Lexical and Computational Semantics-Volume 1: Proceedings of the main conference and the shared task, and Volume 2: Proceedings of the Sixth International Workshop on Semantic Evaluation*, pages 301–309. Association for Computational Linguistics.

Johan Bos. 2016. Expressive power of abstract meaning representations. *Computational Linguistics*.

Lucas Champollion. 2011. Quantification and negation in event semantics.

Federico Fancellu, Adam Lopez, and Bonnie Webber. 2016. Neural networks for negation scope detection. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, volume 1, pages 495–504.

Dan Flickinger, Yi Zhang, and Valia Kordoni. 2012. Deepbank: A dynamically annotated treebank of the wall street journal. In *Proceedings of the 11th International Workshop on Treebanks and Linguistic Theories*, pages 85–96.

Hans Kamp, Josef Van Genabith, and Uwe Reyle. 2011. Discourse representation theory. In *Handbook of philosophical logic*, pages 125–394. Springer.

Montserrat Marimon. 2010. The spanish resource grammar. In *LREC*.

Ryan T McDonald, Joakim Nivre, Yvonne Quirmbach-Brundage, Yoav Goldberg, Dipanjan Das, Kuzman Ganchev, Keith B Hall, Slav Petrov, Hao Zhang,

Oscar Täckström, et al. 2013. Universal dependency annotation for multilingual parsing. In *ACL (2)*, pages 92–97.

Roser Morante and Walter Daelemans. 2012. Conandoyle-neg: Annotation of negation in conan doyle stories. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation, Istanbul*. Citeseer.

Woodley Packard, Emily M Bender, Jonathon Read, Stephan Oepen, and Rebecca Dridan. 2014. Simple negation scope resolution through deep parsing: A semantic solution to a semantic problem. In *ACL (1)*, pages 69–78.

Siva Reddy, Oscar Täckström, Michael Collins, Tom Kwiatkowski, Dipanjan Das, Mark Steedman, and Mirella Lapata. 2016. Transforming dependency structures to logical forms for semantic parsing. *Transactions of the Association for Computational Linguistics*, 4:127–140.

Siva Reddy, Oscar Täckström, Slav Petrov, Mark Steedman, and Mirella Lapata. 2017. Universal semantic parsing. *arXiv Preprint*.

Sebastian Schuster and Christopher D Manning. 2016. Enhanced english universal dependencies: An improved representation for natural language understanding tasks. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*.

Erik Velldal, Lilja Øvrelid, Jonathon Read, and Stephan Oepen. 2012. Speculation and negation: Rules, rankers, and the role of syntax. *Computational linguistics*, 38(2):369–410.

Veronika Vincze, György Szarvas, Richárd Farkas, György Móra, and János Csirik. 2008. The bioscope corpus: biomedical texts annotated for uncertainty, negation and their scopes. *BMC bioinformatics*, 9(11):S9.

A Step-by-step λ-reductions

*Throughout the derivations, we are going to use the variable e in place of x_e and z,y or x in place of x_a. Due to space restrictions, we skip reduction for existential closure ($\rightarrow_{ex-clos}$).

A.1 'John does not eat'

$\lambda P.\lambda Q.\lambda f.P(\lambda e.f(e) \wedge Q(\lambda x.Actor(e,x)))(\lambda f.\exists e.eat(e) \wedge f(e))$
$\rightarrow_{\alpha}\ \lambda P.\lambda Q.\lambda f.P(\lambda e.f(e) \wedge Q(\lambda x.Actor(e,x)))(\lambda g.\exists e'.eat(e') \wedge g(e'))$
$\rightarrow_{\beta}\ \lambda Q.\lambda f.\lambda g.[\exists e'.eat(e') \wedge g(e')](\lambda e.f(e) \wedge Q(\lambda x.Actor(e,x)))$
$\rightarrow_{\beta}\ \lambda Q.\lambda f.\exists e'.eat(e') \wedge \lambda e.[f(e) \wedge Q(\lambda x.Actor(e,x))](e')$
$\rightarrow_{\beta}\ \lambda Q.\lambda f.\exists e'.eat(e') \wedge f(e') \wedge Q(\lambda x.Actor(e',x))$
$\rightarrow_{\beta}\ \lambda Q.\lambda f.\exists e'.eat(e') \wedge f(e') \wedge Q[\lambda x.Actor(e',x)](\lambda f.\exists x.named(x,John,PER) \wedge f(x))$
$\rightarrow_{\alpha}\ \lambda Q.\lambda f.\exists e'.eat(e') \wedge f(e') \wedge Q[\lambda x.Actor(e',x)](\lambda g.\exists z.named(z,John,PER) \wedge g(z))$
$\rightarrow_{\beta}\ \lambda f.\exists e'.eat(e') \wedge f(e') \wedge \lambda g.[\exists z.named(z,John,PER) \wedge g(z)](\lambda x.Actor(e',x))$
$\rightarrow_{\beta}\ \lambda f.\exists e'.eat(e') \wedge f(e') \wedge \exists z.named(z,John,PER) \wedge \lambda x.[Actor(e',x)](z)$
$\rightarrow_{\beta}\ \lambda f.\exists e'.eat(e') \wedge f(e') \wedge \exists z.named(z,John,PER) \wedge Actor(e',z)$

$\lambda P.\lambda Q.\lambda f.\neg P(\lambda e.f(e))(\lambda f.\exists e'.\exists z.eat(e') \wedge f(e') \wedge named(z,John,PER) \wedge Actor(e',z))$
$\rightarrow_{\alpha}\ \lambda P.\lambda Q.\lambda f.\neg P(\lambda e.f(e))(\lambda g.\exists e'.\exists z.eat(e') \wedge g(e') \wedge named(z,John,PER) \wedge Actor(e',z))$
$\rightarrow_{\beta}\ \lambda Q.\lambda f.\neg\lambda g.[\exists e'.\exists z.eat(e') \wedge g(e') \wedge named(z,John,PER) \wedge Actor(e',z)](\lambda e.f(e))$
$\rightarrow_{\beta}\ \lambda Q.\lambda f.\neg\exists e'.\exists z.eat(e') \wedge \lambda e.[f(e)](e') \wedge named(z,John,PER) \wedge Actor(e',z)$
$\rightarrow_{\beta}\ \lambda Q.\lambda f.\neg\exists e'.\exists z.eat(e') \wedge f(e') \wedge named(z,John,PER) \wedge Actor(e',z)$
$\rightarrow_{\beta}\ \lambda f.\neg\exists e'.\exists z.eat(e') \wedge f(e') \wedge named(z,John,PER) \wedge Actor(e',z)$
$\rightarrow_{\beta}\ \lambda f.[\neg\exists e'.\exists z.eat(e') \wedge named(z,John,PER) \wedge Actor(e',z) \wedge f(e')](\lambda x.TRUE)$
$\rightarrow_{\beta}\ \neg\exists e'.\exists z.eat(e') \wedge named(z,John,PER) \wedge Actor(e',z) \wedge \lambda x.[TRUE](e')$
$\rightarrow_{\beta}\ \neg\exists \mathbf{e'}.\exists \mathbf{z}.\mathbf{eat}(\mathbf{e'}) \wedge \mathbf{named}(\mathbf{z},\mathbf{John},\mathbf{PER}) \wedge \mathbf{Actor}(\mathbf{e'},\mathbf{z})$

A.2 'No man came'

$\lambda P.\lambda Q.\lambda f.\forall x.(P(\lambda y.EQ(x,y)) \rightarrow \neg f(x))(\lambda f.\exists x.man(x) \wedge f(x))$
$\rightarrow_{\alpha}\ \lambda P.\lambda Q.\lambda f.\forall x.(P(\lambda y.EQ(x,y)) \rightarrow \neg f(x))(\lambda f'.\exists z.man(z) \wedge f'(z))$
$\rightarrow_{\beta}\ \lambda Q.\lambda f.\forall x.(\lambda f'.[\exists z.man(z) \wedge f'(z)](\lambda y.EQ(x,y)) \rightarrow \neg f(x))$
$\rightarrow_{\beta}\ \lambda Q.\lambda f.\forall x.(\exists z.man(z) \wedge \lambda y.[EQ(x,y)](z) \rightarrow \neg f(x))$
$\rightarrow_{\beta}\ \lambda Q.\lambda f.\forall x.(\exists z.man(z) \wedge EQ(x,z) \rightarrow \neg f(x))$
$\rightarrow_{EQ}\ \lambda Q.\lambda f.\forall x.(man(x) \rightarrow \neg f(x))$
$\rightarrow_{\beta}\ \lambda f.\forall x.(man(x) \rightarrow \neg f(x))$

$\lambda P.\lambda Q.\lambda f.Q(\lambda x.P(\lambda e.Actor(e,x) \wedge f(e)))(\lambda f.\exists e.came(e) \wedge f(e))$
$\rightarrow_{\alpha}\ \lambda P.\lambda Q.\lambda f.Q(\lambda x.P(\lambda e.Actor(e,x) \wedge f(e)))(\lambda g.\exists e'.came(e') \wedge g(e'))$
$\rightarrow_{\beta}\ \lambda Q.\lambda f.Q(\lambda x.\lambda g.[\exists e'.came(e') \wedge g(e'))](\lambda e.Actor(e,x) \wedge f(e)))$
$\rightarrow_{\beta}\ \lambda Q.\lambda f.Q(\lambda x.\exists e'.came(e') \wedge \lambda e.[Actor(e,x) \wedge f(e)](e'))$
$\rightarrow_{\beta}\ \lambda Q.\lambda f.Q(\lambda x.\exists e'.came(e') \wedge Actor(e',x) \wedge f(e'))$
$\rightarrow_{\alpha}\ \lambda Q.\lambda f.Q(\lambda x.\exists e'.came(e') \wedge Actor(e',x) \wedge f(e'))(\lambda f'.\forall x'.(man(x') \rightarrow \neg f'(x'))$
$\rightarrow_{\beta}\ \lambda f.\lambda f'.[\forall x'.(man(x') \rightarrow \neg f'(x'))](\lambda x.\exists e'.came(e') \wedge Actor(e',x) \wedge f(e'))$
$\rightarrow_{\beta}\ \lambda f.\forall x'.(man(x') \rightarrow \neg\lambda x.[\exists e'.came(e') \wedge Actor(e',x) \wedge f(e')](x'))$
$\rightarrow_{\beta}\ \lambda f.\forall x'.(man(x') \rightarrow \neg\exists e'.came(e') \wedge Actor(e',x') \wedge f(e'))$
$\rightarrow_{ex-clos.}\ \forall \mathbf{x'}.(\mathbf{man}(\mathbf{x'}) \rightarrow \neg\exists \mathbf{e'}.\mathbf{came}(\mathbf{e'}) \wedge \mathbf{Actor}(\mathbf{e'},\mathbf{x'}))$

A.3 'Not every man came'

$\rightarrow_{\forall}\ \lambda f.\forall x.(man(x) \rightarrow f(x))$
$\rightarrow_{\neg}\ \lambda f.\neg\forall z.(man(z) \rightarrow f(z))$

$\lambda P.\lambda Q.\lambda f.Q(\lambda x.P(\lambda e.Actor(e,x) \wedge f(e)))(\lambda f.\exists e.came(e) \wedge f(e))$

$\rightarrow_{\alpha}\ \lambda P.\lambda Q.\lambda f.Q(\lambda x.P(\lambda e.Actor(e,x) \wedge f(e)))(\lambda g.\exists e'.came(e') \wedge g(e'))$
$\rightarrow_{\beta}\ \lambda Q.\lambda f.Q(\lambda x.\lambda g.[\exists e'.came(e') \wedge g(e'))](\lambda e.Actor(e,x) \wedge f(e)))$
$\rightarrow_{\beta}\ \lambda Q.\lambda f.Q(\lambda x.\exists e'.came(e') \wedge \lambda e.[Actor(e,x) \wedge f(e)](e'))$
$\rightarrow_{\beta}\ \lambda Q.\lambda f.Q(\lambda x.\exists e'.came(e') \wedge Actor(e',x) \wedge f(e'))$

$\lambda Q.\lambda f.Q(\lambda x.\exists e'.came(e') \wedge Actor(e',x) \wedge f(e'))(\lambda f.\neg\forall z.(man(z) \rightarrow f(z))$
$\rightarrow_{\alpha}\ \lambda Q.\lambda f.Q(\lambda x.\exists e'.came(e') \wedge Actor(e',x) \wedge f(e'))(\lambda f'.\neg\forall z.(man(z) \rightarrow f'(z))$
$\rightarrow_{\beta}\ \lambda f.\lambda f'.[\neg\forall z.(man(z) \rightarrow f'(z))](\lambda x.\exists e'.came(e') \wedge Actor(e',x) \wedge f(e'))$
$\rightarrow_{\beta}\ \lambda f.\neg\forall z.(man(z) \rightarrow \lambda x.[\exists e'.came(e') \wedge Actor(e',x) \wedge f(e')](z))$
$\rightarrow_{\beta}\ \lambda f.\neg\forall z.(man(z) \rightarrow \exists e'.came(e') \wedge Actor(e',z) \wedge f(e'))$
$\rightarrow_{ex-clos.}\ \neg\forall \mathbf{z}.(\mathbf{man}(\mathbf{z}) \rightarrow \exists \mathbf{e'}.\mathbf{came}(\mathbf{e'}) \wedge \mathbf{Actor}(\mathbf{e'},\mathbf{z}))$

A.4 'Nobody came'

$\lambda P.\lambda Q.\lambda f.Q(\lambda x.P(\lambda e.f(e) \wedge Actor(e,x)))(\lambda f.\exists e.f(e) \wedge came(e))$
$\rightarrow_{\alpha}\ \lambda P.\lambda Q.\lambda f.Q(\lambda x.P(\lambda e.f(e) \wedge Actor(e,x)))(\lambda g.\exists e'.g(e') \wedge came(e'))$
$\rightarrow_{\beta}\ \lambda Q.\lambda f.Q(\lambda x.\lambda g.[\exists e'.g(e') \wedge came(e')](\lambda e.f(e) \wedge Actor(e,x)))$
$\rightarrow_{\beta}\ \lambda Q.\lambda f.Q(\lambda x.\exists e'.\lambda e.[f(e) \wedge Actor(e,x)](e') \wedge came(e'))$
$\rightarrow_{\beta}\ \lambda Q.\lambda f.Q(\lambda x.\exists e'.f(e') \wedge Actor(e',x) \wedge came(e'))$

$\lambda Q.\lambda f.Q(\lambda x.\exists e'.f(e') \wedge Actor(e',x) \wedge came(e'))(\lambda f.\neg\exists x.person(x) \wedge f(x))$
$\rightarrow_{\alpha}\ \lambda Q.\lambda f.Q(\lambda x.\exists e'.f(e') \wedge Actor(e',x) \wedge came(e'))(\lambda g.\neg\exists z.person(z) \wedge g(z))$
$\rightarrow_{\beta}\ \lambda f.\lambda g.[\neg\exists z.person(z) \wedge g(z)](\lambda x.\exists e'.f(e') \wedge Actor(e',x) \wedge came(e'))$
$\rightarrow_{\beta}\ \lambda f.\neg\exists z.person(z) \wedge \lambda x.[\exists e'.f(e') \wedge Actor(e',x) \wedge came(e')](z)$
$\rightarrow_{\beta}\ \lambda f.\neg\exists z.\exists e'.person(z) \wedge f(e') \wedge Actor(e',z) \wedge came(e')$
$\rightarrow_{ex-clos.}\ \neg\exists \mathbf{z}.\exists \mathbf{e'}.\mathbf{person}(\mathbf{z}) \wedge \mathbf{Actor}(\mathbf{e'},\mathbf{z}) \wedge \mathbf{came}(\mathbf{e'})$

A.5 'John came with nothing'

$\lambda P.\lambda Q.\lambda f.P(\lambda e.f(e) \wedge Q(\lambda x.with(e,x)))(\lambda f.\exists e.came(e) \wedge f(e))$
$\rightarrow_{\alpha}\ \lambda P.\lambda Q.\lambda f.P(\lambda e.f(e) \wedge Q(\lambda x.with(e,x)))(\lambda g.\exists e'.came(e') \wedge g(e'))$
$\rightarrow_{\beta}\ \lambda Q.\lambda f.\lambda g.[\exists e'.came(e') \wedge g(e')](\lambda e.f(e) \wedge Q(\lambda x.with(e,x)))$
$\rightarrow_{\beta}\ \lambda Q.\lambda f.\exists e'.came(e') \wedge f(e') \wedge \lambda e.[Q(\lambda x.with(e,x))](e')$
$\rightarrow_{\beta}\ \lambda Q.\lambda f.\exists e'.came(e') \wedge f(e') \wedge Q(\lambda x.with(e',x))$
$\rightarrow_{\beta}\ \lambda Q.\lambda f.\exists e'.came(e') \wedge f(e') \wedge Q[\lambda x.with(e',x)](\lambda f.\neg\exists x.thing(x) \wedge f(x))$
$\rightarrow_{\alpha}\ \lambda Q.\lambda f.\exists e'.came(e') \wedge f(e') \wedge Q[\lambda x.with(e',x)](\lambda g.\neg\exists z.thing(z) \wedge g(z))$
$\rightarrow_{\beta}\ \lambda f.\exists e'.came(e') \wedge f(e') \wedge \lambda g.[\neg\exists z.thing(z) \wedge g(z)](\lambda x.with(e',x))$
$\rightarrow_{\beta}\ \lambda f.\exists e'.came(e') \wedge f(e') \wedge \neg\exists z.thing(z) \wedge \lambda x.[with(e',x)](z)$
$\rightarrow_{\beta}\ \lambda f.\exists e'.came(e') \wedge f(e') \wedge \neg\exists z.thing(z) \wedge with(e',z)$

$\lambda P.\lambda Q.\lambda f.P(\lambda e.f(e) \wedge Q(\lambda x.Actor(e,x)))(\lambda f.\exists e'.came(e') \wedge f(e') \wedge \neg\exists z.thing(z) \wedge with(e',z))$
$\rightarrow_{\alpha}\ \lambda P.\lambda Q.\lambda f.P(\lambda e.f(e) \wedge Q(\lambda x.Actor(e,x)))(\lambda g.\exists e'.came(e') \wedge g(e') \wedge \neg\exists z.thing(z) \wedge with(e',z))$
$\rightarrow_{\beta}\ \lambda Q.\lambda f.\lambda g.[\exists e'.came(e') \wedge g(e') \wedge \neg\exists z.thing(z) \wedge with(e',z)](\lambda e.f(e) \wedge Q(\lambda x.Actor(e,x)))$
$\rightarrow_{\beta}\ \lambda Q.\lambda f.\exists e'.came(e') \wedge \lambda e.[f(e) \wedge Q(\lambda x.Actor(e,x))](e') \wedge \neg\exists z.thing(z) \wedge with(e',z)$
$\rightarrow_{\beta}\ \lambda Q.[\lambda f.\exists e'.came(e') \wedge f(e') \wedge Q(\lambda x.Actor(e',x)) \wedge \neg\exists z.thing(z) \wedge with(e',z)]$
$(\lambda g.\exists y.named(y,John,PER) \wedge g(y))$
$\rightarrow_{\beta}\ \lambda f.\exists e'.came(e') \wedge f(e') \wedge \lambda g.[\exists y.named(y,John,PER) \wedge g(y)](\lambda x.Actor(e',x)) \wedge \neg\exists z.thing(z) \wedge with(e',z)]$
$\rightarrow_{\beta}\ \lambda f.\exists e'.came(e') \wedge f(e') \wedge \exists y.named(y,John,PER) \wedge \lambda x.[Actor(e',x)](y) \wedge \neg\exists z.thing(z) \wedge with(e',z)$
$\rightarrow_{\beta}\ \lambda f.\exists e'.came(e') \wedge \exists y.named(y,John,PER) \wedge f(e') \wedge Actor(e',y) \wedge \neg\exists z.thing(z) \wedge with(e',z)$
$\rightarrow_{ex-clos.}\ \exists \mathbf{e'}.\exists \mathbf{y}.\mathbf{came}(\mathbf{e'}) \wedge \mathbf{named}(\mathbf{y},\mathbf{John},\mathbf{PER}) \wedge \mathbf{Actor}(\mathbf{e'},\mathbf{y}) \wedge \neg\exists \mathbf{z}.\mathbf{thing}(\mathbf{z}) \wedge \mathbf{with}(\mathbf{e'},\mathbf{z})$