Natural language compositionality in Coq*

Erkki Luuk

Institute of Computer Science, University of Tartu
erkkil@gmail.com

Abstract. The paper presents a new formalism (a minimalist type system) for modeling natural language compositionality over the three levels of compositional semantics, syntax and morphology. The formalism reduces compositionality to a single level by using a subclass of compound types (syntactic compounds of multiple types or their terms) and types for morphemes. The paper includes a detailed comparison with earlier similar approaches and an overview of the implementation of the formalism on a structurally diverse fragment of English in the Coq proof assistant.

Keywords: natural language · compositionality · type theory · Coq.

1 Introduction

The last 70 years or so have seen many concentrated efforts of natural language (NL) formalization, many of which have involved the use of types and functions, either on paper [16,20] or, starting from this century, also as computer code [3,5,26]. In this paper, we introduce a new formalism, a minimalist type system for modeling NL compositionality. We will also present an implementation of the formalism in the Coq proof assistant. The paper is organized as follows. Section 2 gives a motivating example, followed by an overview of the formalism and its novel contributions in section 3. Section 4 introduces the Coq development of NLC on a fragment of English. An extensive comparison of NLC with existing formalisms is in section 5. The paper concludes with a short discussion in the eponymous section.

2 The quest for a parsimonious representation

Traditionally, NL semantics has been modeled in nth order logic, for $n \leq 4$. To understand why this should be so, one should recall the model-theoretic definition of the meaning of a declarative sentence as its truth conditions (conditions on which the sentence would be true) [8]. The main problem with the definition is that it cannot be readily extended to other syntactic categories, such as

* Accepted to LOPSTR 2019 (http://www.cs.unibo.it/projects/lopstr19). This work has been supported by IUT20-56 and European Regional Development Fund through CEES.

e.g. morphemes, words, phrases and interrogative sentences, all of which have meaning. For example, the notion of truth-conditions of a morpheme, word or phrase can be scarcely upheld, unless we arbitrarily conflate the meanings of e.g. *cat*, *a cat* and *there is a cat* as $\exists x.\,\mathrm{cat}(x) := \mathrm{True}$. Clearly, the latter signals a failure of model-theoretic semantics to distinguish between the meanings of the constructions. This aside, the notation is awkward, which becomes clear if we compare the following:

(1) $\exists x.\,\texttt{red}(x) \wedge \texttt{ball}(x)$
(2) `a (red ball)`
(3) $\exists x.\,\texttt{veryred}(x) \wedge \texttt{ball}(x)$
(4) $\exists x.\,\texttt{very}(\texttt{red}(x)) \wedge \texttt{ball}(x)$
(5) `a ((very red) ball)`

First, as mentioned above, (1) fails to capture the semantic differences between *red ball*, *a red ball* and *there is a red ball*. (For (3)–(4), the argument is similar.) Second, as soon as some complexity is added, we need a higher-order logic (4) or must deal with a lossy compression in the first-order logic (3) (the example is lossy as it does not even distinguish between different words)[1]. Third, nothing in *red ball* justifies the existential quantification (at least not in the way it seems justified in *a red ball*). By contrast, the formulas (2) and (5) are free from these problems **and** have a near-optimal correspondence to NL.

Another staple of a (not only) type-theoretical NL modeling has been a **separate** representation of all three levels of a modality-independent NL description (i.e. morphology[2], syntax and semantics). In most (all?) cases, this either means omitting some of the levels or deploying an impressive array of theories, formalisms and interfaces. Thus, in one extreme, we have works dealing only with e.g. semantics [1,17,2], while in the other there is a variety of approaches ranging from the categorial tradition [16,27,21,22] to an entire high-level programming language with dependent types [26]. As compared to these alternatives, the key novelty of the present approach is an integrated and parsimonious modeling of morphology, syntax and semantics. The modeling features types for morphemes, polymorphism and a particular subclass of compound types (syntactic compounds of multiple types or their terms), resulting in a representation with a close correspondence to NL ((2) and (5) are examples). A detailed comparison with related approaches is in section 5.

3 NLC

Let us call our formalism NLC (for natural language code, compositionality or C). NLC is a *type system for modeling semantic, syntactic and morphological compositionality* of NL. Hence, its scope is quite wide as compared to dedicated

[1] The possibilities are not mutually exclusive (see e.g. [20]).
[2] Linguistic morphology studies (the combination of) morphemes, the smallest signs (form-meaning units) in language.

systems (type-driven or otherwise) for modeling one of those three levels, while being narrower than that of "maximal" formalisms such as (e.g.) Head-Driven Phrase Structure Grammar (HPSG) that model also word and morpheme orders and lexical semantics. Thus, NLC handles *all* compositionality a NL formalism is expected to, but does not cover syntax, semantics and morphology in their entirety, as constituent orders and lexicon are omitted. The omission is neither accidental nor strictly a deficiency, as it makes sense to model the compositionality together across all levels, since the compositionality is, in a weaker sense, cross-linguistically universal, while lexicon and constitutent orders are language-specific. By the "weaker sense" I mean "universal in all languages that have the (compositional) categories", such as noun, verb, adjective, nominative, accusative, etc., as I do not claim that all languages have all these categories. However, in all languages that have them, adjectives are functions over nouns and not vice versa, adverbs are functions over adjectives and not vice versa, etc., i.e. the general logic of compositionality is universal.

There are ways to test this. First, I suggest the very general principle

(6) A formula must derive a well-typed standalone NL expression at every stage of derivation,

where by a "stage of derivation" is meant a composition. Thus, from the hypothetical parses of *a very red ball*:

(7) `a (very (red ball))`,
(8) `a (very red) ball`,
(9) `a ((very red) ball)`,

(8) is ruled out by (6) already, as *a very red* is not a well-typed standalone NL expression. Second, a type-theoretic test. One observes that *a very red ball*, *a red ball* and *a ball* are all well-typed standalone NL expressions. Thus, `a` : `X` $\rightarrow$ `XP`, and *very red ball*, *red ball*, *ball* : `X`, where `X` and `XP` are at this point arbitrary labels. Then by (7), `very` : `X` $\rightarrow$ `X`, which does not compute, since *very ball* is ill-typed. Given the lack of sensible alternatives, (9) must be correct. The third test is that of modification or "semantic contribution": e.g. *very* modifies (or contributes to) the semantics of *red*, not e.g. vice versa, and adopting the (very sensible) convention of taking the modifier to be a function over that what it modifies, we get `very red` instead of `red very`, `a ball` instead of `ball a`, etc. Notice that the circumstance of English word order aligning with the function application syntax is somewhat accidental, as all possible orders of {subject, object, verb} are present in the world's languages (although not all are equally common) [11].

Returning to the topic of universality, it is easy to observe that the well-typedness is semantically motivated. This is significant, as NL semantics is generally assumed to be universal, owing to the possibility of translation from any one language to another. Thus, with a slightly more general (or abstract) notation, one could even approximate a Universal Grammar with the formulas (a possibility not pursued here).

Finally, since we are dealing with a (weakly) universal, semantically motivated phenomenon, using types to model it is not a long shot, given the common definition of 'type' as a category of semantic value. A remarkable feature of NLC is modeling syntactic and morphological compositionality without specialized notations like trees, phrase structure rules or (as in HPSG) attribute value matrices. In fact, the "look and feel" of NLC is very similar to NL.

As a type system, NLC requires only four rules:

$$\frac{a \in \mathcal{M}^k}{a : T^k}\ \text{AT-Intro},$$

$$\frac{e_0 : T_0^k,\ ...,\ e_m : T_m^k \quad a : T_0^k \to ... \to T_m^k \to T_{m+1}^k}{a(e_0, ..., e_m) : T_{m+1}^k}\ \text{FT-Elim},$$

$$\frac{e_0 : T_0^k,\ ...,\ e_m : T_m^k \quad \vdash \quad a(e_0, ..., e_m) : T_{m+1}^k}{((e_0, ..., e_m) \mapsto a(e_0, ..., e_m)) : T_1^k \to ... \to T_m^k \to T_{m+1}^k}\ \text{FT-Intro},$$

$$\frac{B : T^k \qquad c_0 : C_0^k,\ ...,\ c_{n+1} : C_{n+1}^k \qquad B \mapsto c_0, ..., B \mapsto c_{n+1}}{B : C_0^k..C_{n+1}^k}\ \text{LT-Intro}.$$

AT-Intro (atomic type introduction) generates atomic terms of NLC and introduces type variables for them. The rule says that all morphemes have types in NLC (technically: "if a is a morpheme of language k then a has type T^k", where T^k is a proper type variable[3]). FT-Elim (function type elimination) is a computation rule, with $a(e_0, ..., e_m)$ an application and $T_0^k \to ... \to T_{m+1}^k$ the right-associative function type. FT-Intro (function type introduction) is standard and derived from FT-Elim. LT-Intro (lump type introduction) is, together with AT-Intro, NLC-specific. B is a term constant and $C_0^k, ..., C_{n+1}^k$ type constants in NLC, $x \mapsto y$ is a function interpreting x as y, and $C_0^k..C_{n+1}^k$ the notation for a lump type (comprising types C_0^k through C_{n+1}^k, i.e. there must be at least two). The interpretations must not preclude each other[4]. All indices are natural numbers.

Lump types are compound types that satisfy LT-Intro. Compound type is defined as a syntactic compound of multiple types or their terms. For lump types, the exact principle of compounding is irrelevant, as long as it supports function types over the compound types defined by the principle. For example, Appendix B models a tiny NL fragment with lump types implemented as application, record, product or Π-types[5]. Since intersection types are, by definition, a small subclass of lump types, it is semantically natural to encode lump

[3] A proper type is a type that is not a universe.

[4] The interpretations that preclude each other (e.g. the interpretations of *run* as a noun and verb) are dealt with polymorphism instead.

[5] The file is online at `https://gitlab.com/jaam00/nlc/blob/master/compound.v`

types as intersection types in the relatively few languages that have them (e.g. TypeScript, Flow...).

In NLC, functions must be derived manually from NL expressions and the abovementioned logic of composition. In general, if there is a function $f : A \to B$ in NLC, there is no function $g : B \to A$. This follows from observing NL expressions, from the functions of linguistic categories like noun, adjective, etc., and from the general principle of asymmetry in NL composition[6]. But whence the principle? A necessary explanation for it is the fact that function composition is highly non-commutative. Another one is the ease of interpretation brought about by the avoidance of functional symmetry (e.g. while flexibles[7] are ambiguous, I am not aware of one "flexing" between f and g).

3.1 Lump types

The use of lump types is linguistically motivated by most stems belonging to several types, e.g. *john* is a proper name, in nominative case, male, a physical, sentient, limbed, etc. entity, while *run* is a flexible, a noun in nominative or a verb taking a limbed argument. Flexibles are tricky, as they exhibit a true polymorphism, with different readings (e.g. the nominal and verbal readings of *run*) precluding one another. When different typings always coincide, as with e.g. *john* or *nut*, it is far more convenient to pack them together into a lump type than to keep track of n distinct typings. Similarly, it may be preferable to index the case (e.g. nominative) on the lump type rather than to make it a modifer over an argument, as in `NOM john`. Thus, it is desirable to have both lump types and polymorphism. Below are examples of *john* typed as a lump type[8]:

```
Check john: XP0 (hu _) NOM SG (Pn Ml).
Check john: XP0 Phy NOM SG (Pn Ml).
Check limbed john: XP0 Lim NOM SG PN+.
Fail Check limbed john: XP0 Phy NOM SG (Pn Ml).
Fail Check limbed (john: XP0 Phy NOM SG (Pn Ml)).
```

This tells that *john* is of type `XP`[9], human, in nominative, singular, proper name, and male. `XP`s are arguments of verbs and flexibles, and proper names are `XP0`s because there are different ways of deriving an `XP` (e.g. zero-derivation in proper names or prepending an article as in *the man*). Humans have specific selectional restrictions[10] such as physical, sentient, bilogical, animate, limbed, etc. The argument of `hu` tells which one is active in the context. For example, the adjective

[6] There is a lot of dedicated research on this, see e.g. the contents of volumes [9]–[10]

[7] Flexibles are word classes that "flex" between different word classes (e.g. *walk* is a noun and a verb) [19].

[8] The full Coq code for the fragment is at `https://gitlab.com/jaam00/nlc/blob/master/frag.v`

[9] Frequently also reffered to as NP or DP.

[10] Selectional restrictions are the ontological restrictions imposed by a relation on its arguments' types.

limbed and the flexible *throw* require a limbed entity for their first (and in case of the adjective, only) argument. `(hu H_Phy)` reduces to `Phy`. `PN+` is the notation for a proper name with additional (gender) information.

The encoding of lump types as application types has the benefit of (at least visual) simplicity. In Coq, one can usually concoct a simpler notation for a type, but it would be difficult to get a simpler one than this.

While LT-Intro has not been defined before, it is still possible that lump types have been used implicitly. The closest examples I have found are from Coq [3,4] and Grammatical Framework (GF) [26]. GF's set of compound types contains e.g. function, record and table types. The use of function types is computational, i.e. they are not used for storing values (as would be the case with lump types), while records and tables engage concrete (i.e. linearized) syntax and morphology. At least superficially, there is no good match with NLC, which has only abstract morphosyntax. Below is a relevant example of a GF's record and table[11]:

(10)
```
noun "vino" "vini" Masc : {table {Sg => vino ; Pl => vini} :
Number => Str ; _ : Gender}
```

A table is a finite function[12], in this case from number to string, while the record type has two fields (for the table and gender, respectively). Thus, an Italian noun's type is indexed by number, two strings and gender. Below are relevant examples of records in the Coq implementations:

```
Record Irishdelegate: CN := mkIrishdelegate { c :> delegate; _ : Irish c }.
Record Book: Set := mkBook {Arg:> PhyInfo; Qualia: BookQualia Arg}.
```

The first is a rendering of *Irish delegate* as a record type [3], the second of *book* as another record type [4], a physical/informational object with Pustejovsky's [25] qualia structure. The structure specifies `Qualia` as a human-made physical/informational object (`PhyInfo` and `BookQualia` are also record types).

Of the various indices to the right of `:`, the record `noun "vino" "vini" Masc` could be said to have only type `Gender`, i.e. it does not qualify as a lump type by LT-Intro. In particular, as it represents both `Sg` and `Pl`, it lacks number; similarly, it represents both strings (and a record itself is not a string of the object language). The Coq examples fare better in this respect, as an Irish delegate could be sensibly typed as a delegate and as something Irish (and likewise for a book).

The conclusion is that, differently from `{table {...} ...}`, `Book` and `Irishdelegate` are indeed lump types, although very different from `XP0 _ _ _ _`. Perhaps the main difference is that `Book` and `Irishdelegate` model only semantics[13], while `XP0 _ _ _ _` models syntax, semantics and morphology. Another difference is in their implementations. An obvious advantage of record types is that they occur

[11] Taken from `http://www.grammaticalframework.org/doc/tutorial/gf-tutorial.html#toc64`

[12] A function with a finite enumeration of all argument-value pairs.

[13] I assume that `Irish c` represent something Irish rather than a phrase *Irish ....*

in many programming languages, but there are several alternatives, such as Σ-, Π-, list, application or Cartesian product types (some of which — but not all — can be used for defining record types in some languages [6]).

4 Implementation

NLC has been implemented for an English fragment containing stems, nouns, verbs, flexibles, proper names, pronouns, XPs, adjectives, sentential, adjectival and generic adverbs, determiners, demonstratives, quantifiers, tense-aspect-mood, gender, number and nonfinite markers, cases, adpositions, sentences (both simple and complex), connectives, connective phrases (for substantives, adjectives, adverbs and sentences), copulas, complementizers and selectional restrictions (for physical, informational, limbed, animate, biological and sentient entities). The linguistic categories not formalized in the fragment are gerunds, participles, auxiliary verbs, interrogatives, numerals, negation, mass/count distinction and unspecified selectional restrictions (and possibly others). These are left as a future work.

In general, the implementation has few representatives of each category, as the goal was to model a structurally rich rather than quantitatively extensive fragment. For a structurally rich fragment, a diverse set of correctly composing types must be defined manually. A quantitative extension of this set should be generated semi-automatically, combining resources (tagged corpora or dictionaries) with standalone scripts or machine learning.

4.1 Base types

The implementation was done in Coq (ver. 8.9). We start by defining some categories[14]:

```
Inductive NUM := SG | PLR. (*number: singular, plural*)
Inductive GN := Ml | Fm. (*gender*)
Inductive ST := F | N | PN | PR | Nn (x:GN) | Pn (x:GN) | Pr (x:GN).
(*argument stems: flexible, noun, proper name, pronoun, also w/ gender*)
Parameter NF S: Prop. (*nonfinite, sentence*)
Inductive CA := NOM | ACC | GEN | LAT | LOC | DIR.
(*case/adposition: nominative, accusative, genitive, lative, locative...*)
```

Coq has inductive types, and there are some reasons to prefer them over variables (defined with e.g. `Parameter` in Coq). First, Gallina's (Coq's specification language's) pattern matching works only over inductive types, so we should use them whenever we want it (Coq's official tactic language Ltac[15] has a general syntactic pattern matching). Second, if we want to define proof schemes (e.g. the

[14] The code is taken from Supplement I: `https://gitlab.com/jaam00/nlc/blob/master/frag.v`

[15] `https://coq.inria.fr/distrib/current/refman/proof-engine/ltac.html`

Boolean equality) over a type, the type must also be inductive. Finally, differently from inductive types, variables have no term resolution, so inductives are preferable for a better type inference. To give an example, `Inductive NU := SF | PF (x:DD)` declares an inductive type `NU: Type` with two constructors (i.e. canonical terms) `SF: NU` and `PF: DD → NU` (`Type` is the universe of types).

While not much implementationally, the following line is a central one, as it defines our lump types **and** (for exposition purposes, a minimal) phrase architecture:

```
Parameter STM PLU XP: SR -> CA -> NUM -> ST -> Type. (*stem, plural, XP*)
```

Thus, `STM _ _ _ _`, `PLU _ _ _ _`, etc., are our lump types (in Coq, `_` is a placeholder for any admissible type or term). The comments `(*...*)` explain what types of phrases they are: the first the stem phrase, the second plural phrase, etc. Thus we get e.g. `car: STM _ _ _ _` and `PL car: PLU _ _ _ _`, where `PL` is a plural marker (such as *-s* in English).

NL relations (adjectives, determiners, etc.) frequently take different types of phrases as arguments. So we need a device to generalize over n types. For this, we will use canonical structures[16] (canonical instances of record types):

```
(*(in|out)put for ADJ, input to PL...*)
Structure CSTM := cs {gs: SR -> CA -> NUM -> ST -> Type}.
Canonical Structure cs_s: CSTM := cs STM.
Canonical Structure cs_p: CSTM := cs PLU.
```

Now, whenever we need to refer to e.g. the general input/output type for `ADJ`, we write `gs _`. This is known as notation overloading (and will, in this case, reduce to `STM` or `PLU`).

4.2 Selectional restrictions

Selectional restrictions are semantic (or ontological) restrictions imposed by NL relations on their arguments; e.g. *throw* selects for the first argument that is a limbed and the second that is a physical entity. It is straightforward to type something which has an elementary selectional restriction (e.g. *hut*, a physical entity). But what about words like *he* or *john*, which are (at least) physical, sentient, informational, animate, biological and limbed entities? One way to do this is to rely on notation overloading; this is the approach taken in Supplement I. In the more recent Supplement II[17] we take a shorter way, and define selectional restrictions (SRs) as

[16] `https://coq.inria.fr/distrib/current/refman/addendum/canonical-structures.html`. We could also use type classes but canonicals are more minimalist.

[17] `https://gitlab.com/jaam00/nlc/blob/master/cop.v`. Primarily a testbed for new design choices, Supplement II is the more experimental and smaller fragment of the two. Both Supplements have some unique features while being fully compatible otherwise.

```
(*selectional restrictions*)
Inductive SRH := Phy | Sen | Inf | Ani | Bio. (*humans*)
Inductive SRV := Phy_v | Ani_v. (*vehicles*)
Inductive SR := srh (x:SRH) | hum_sr (x:SRH):> SR | veh_sr (x:SRV):> SR.
Notation "' x" := (srh x) (at level 8). (*general*)
Notation "> x" := (hum_sr x) (at level 8). (*humans (coerced to general)*)
Notation "'>v' x" := (veh_sr x) (at level 8). (*vehicles (coerced)*)
```

Humans have a distinct type of SRs, and the general restrictions are obtained by parametrizing on this type (the circumstance that general restrictions are limited to those of humans is coincidental). Due to the coercion (`:>`), `SRH` is a subtype of `SR`, while a term of `SR` is not a term of `SRH`. The notations are added for convenience. Vehicles are dealt with likewise. Now we may declare

```
Definition ADJ {x y z w u} := gs x y z w u -> gs x y z w u.
Parameter red: forall {x y z w}, @ADJ x Phy y z w.
Parameter ill: forall {x y z w}, @ADJ x Bio y z w.
```

and get the desired result: `ill` takes and returns stem phrases of biological and `red` of physical entities[18].

We have also implemented an on/off switch for SRs, enabling us to respect and ignore them with minimal changes to the code (commenting out an import of one of the two modules specifying differentiated and uniform SRs, respectively). The modules themselves are implemented differently in Supplements I and II; in the latter, the implementation is simpler, lacking some of the features of Supplement I[19]. When selectional restrictions are switched off, the code is checked only for morphosyntactic well-typedness.

4.3 Sentences

There are at least two ways to define sentences, taken in Supplements I and II, respectively. Since the latter is more recent, shorter, and seemingly also more parsimonious, we will review the latter approach only. After the base types and selectional restrictions have been defined (in sections 4.1-4.2), we give the following minimal (and almost standalone) fragment for exposition purposes:

```
Structure CPX := cpx {gp: SR -> CA -> NUM -> ST -> Type}.
Canonical Structure cp_p := cpx PLU.
Canonical Structure cp_x := cpx XP.
```

[18] Arguments in braces are implicit (i.e. implicitly applied), and `@ADJ` makes all arguments of `ADJ` explicit.

[19] If SRs are switched on, then e.g. `red (limbed man)` does not type check, since the phrase `limbed man` has restriction `Lim` while `red` expects `Phy`. To circumvent the limitation, Supplement I employs a special notation `[...]`. In particular, `red [limbed ...]` type checks there if SRs are off; if SRs are on, e.g. `red [limbed john]` is well- and `red [limbed book]` ill-typed. A more elegant, notationless solution is being developed for Supplement II.

```
Parameter hut: STM 'Phy (acc acc_n) SG N.
Parameter car: forall {x:SRV}, STM x (acc acc_n) SG N.
Parameter man: forall {x:SRH}, STM x (acc acc_n) SG N.
Parameter a: forall {x y z w}, gs x y z SG w -> gp cp_x y z SG w.
Parameter he: forall {x:SRH}, XP x NOM SG (Pr Ml).
Parameter him: forall {x:SRH}, XP x ACC SG (Pr Ml).

Parameter TAM: forall {x:Type}, (NF -> x) -> x.  (*tense-aspect-mood*)
Structure ANI := ani0 {ani:SR}.  (*animate entities*)
Canonical Structure ani_a := ani0 'Ani. (*default*)
Canonical Structure ani_b {x} := ani0 > x.  (*human*)
Canonical Structure ani_c {x} := ani0 >v x.  (*vehicle*)
Parameter hit: forall {w u k d m n f}, NF ->
gp cp_x (ani f) NOM w u -> gp cp_x k (acc d) m n -> S.

Check TAM hit (a man) (a hut). (*tests*)
Fail Check TAM hit (a hut) (a man). (*wrong selectional restriction*)
Fail Check TAM hit man (a man). (*"man" is not XP*)
Check TAM hit (a man) (a man).
Fail Check TAM hit (a man) he. (*wrong case*)
Check TAM hit (a man) him.
Check TAM hit he him.
Fail Check TAM hit him him. (*wrong case*)
Fail Check TAM hit (red he) him. (*"he" is not STM*)
Fail Check TAM hit (red man) him. (*"red man" is not XP*)
Check TAM hit (a (red man)) him.
Check TAM hit (a car) him.
```

We define the canonical structure `CPX` for accessing the elements of the set `{PLU, XP}` via notation overloading, and declare the variables `a`, `he`, etc. The variables correspond to morphemes, and all the argument variables (`he`, `man`, etc.) have lump types, implemented as function applications. `acc _` reduces to accusative or false accusative, the latter being defined as nominative. We need this for words such as `hut`, `man` and `car`, where nominative and accusative are indistinguishable. The definition of false accusative is omitted for space considerations.

To use a relation like *hit* on its arguments, it must be finitized with a tense-aspect-mood marker (e.g. `TAM` or `PAST`). A convenient way to do this is to declare an infinitive marker `NF` and prepend `NF →` to the relation's function type. As `NF` is an empty type, a function of type `NF → _` behaves like an infinitive, i.e. cannot be applied to anything. However, we can declare a function of type `∀ {x:Type}, (NF → x) → x`, and apply this to the infinitive to make it finite.

We are using the notation overloading trick again for `hit`, for it to accept various kinds of animate entities (in the fragment, humans and vehicles, even if they happen to have e.g. selectional restrictions `> Inf` and `>v Phy_v`).

4.4 Appendices and tests

For space considerations, longer (but still interesting) examples are at the end of the paper in the Appendices (the full code is in the online Supplements). Below is a short overview of the contents of the Appendices.

The examples we have seen are not truth-functional; however, sometimes a truth-functional semantics may be preferred. Appendix A introduces an optional truth-functionality module from Supplement I. Appendix B illustrates some ways of defining lump types (namely, as application, record, Cartesian product and Π-types). Appendix C shows some tests from the Supplements. A handful of the most revealing ones are below:

```
Check PRES know (several (PL man)) (a (few (PL man))).
(* several men know a few men *)
(* "john threw madly blue stones at the hut
and red limbed boys" has 2 parses: *)
Check madly (PAST throw) john (blue (-s stone))
(at (and (the hut) (red [limbed (-s boy)]))): S.
Check PAST throw john (madly blue (-s stone))
(at (and (the hut) (red [limbed (-s boy)]))): S.
(* Supplement I uses notation "[...]" to make
e.g. a limbed entity into a physical one *)

Check PRES throw (all (the (-s boy))) (every ball) (to him): S.
(* "all the boys throw every ball to him" is a sentence *)
Fail Check PAST throw john (all (-s stone)) (at he).
(* "john threw all stones at he": wrong case *)
Fail Check PRES throw (a walk) john. (* "a walk throws john": wrong SR *)
Check PRES throw john (-s stone) (at (all (madly (madly red)
(red (-s hut))))): S.
(* "john throws stones at all madly, madly red, red huts" is a sentence *)
Check PRES throw john (-s stone) (at (all (madly (madly red)
(red [limbed [-s hut]])))): S.
(* "john throws stones at all madly, madly red, red, limbed huts"
checks only if SRs are off *)
Check all ((and madly madly) red
(red (red [and blue limbed [-s john]]))): QU2 _ _ _ _.
(* "all madly and madly red, red, red, blue
and limbed johns" is a quantifier phrase *)
```

5 Related work

The tradition of using type theory for modeling NL started with Lambek calculus [16] in the pre-existing tradition of categorial grammar, invented by Ajdukiewicz in 1935 and developed (unbeknowst to Lambek) in the beginning of 1950s by Bar-Hillel [23]. The categorial tradition of NL modeling that followed [16] is rich but discontinuous (see [23] for an overview). Combinatory Categorial Grammar

(CCG) [27] and multimodal type-logical grammars [21] are some representatives of the tradition, one of the later additions to which is the Grail parser/automated theorem prover for multimodal categorial grammars [22].

It is convenient to compare all these (and other) categorial formalisms (CFs) to NLC together. A major similarity is that both NLC and the CFs model syntax and semantics together, and since both are type-driven, syntax may be even dismissed as a separate level of description[20]. Another major similarity is that both model NL compositionality with functions. There are more similarities, but with this the major ones seem to end.

A few major differences are as follows. The formalisms are categorial, NLC not. The CFs have diverse (and sometimes quite complex) underpinnings, while NLC is just a minimalist type system, i.e. very simple in comparison. The third difference is that NLC's types are syntactic, morphological and/or semantic, while CFs have only syntactic (and some, e.g. ACG [12] and its generalization AACG [15], also semantic) types. The fourth is that CFs allow to model constituent order, NLC not. The fifth is that (some?) CFs (e.g. CCG) can yield multiple formulas for a single NL expression independently of interpretation ambiguity, which is not the case in NLC.

Besides the categorial tradition, there are other typed approaches to NL to consider. The historical starting point for them is Montague Grammar of the 1970s, e.g. [20] has the primitive types of truth-values and entities, and diverse function types. More relevant for NLC — and especially its implementation — is the work that has followed in a richly typed setting, i.e. using dependent and/or polymorphic types. Methodologically, the closest approaches to NLC are Grammatical Framework (GF) [26] and a series of typed formalisations on different levels of abstraction [7,18,3,17], some of them implemented in Coq [4,5,18].

We start with GF, which is richly typed, uses record types and models NL, so is somewhat similar to NLC. However, NLC's difference from GF is profound. GF is a *formal language for writing NL grammars*, while NLC is a *type system for modeling semantic, syntactic and morphological compositionality* of NL. In particular, the "grammar" in the name of GF implies syntax and morphology but not semantics. It is only recently that parts of Abstract Meaning Representation and FrameNet libraries for GF have been implemented[13,14]. And the use of record types in GF, although pervasive, is very different from their possible use in NLC and does not follow LT-Intro (see section 3.1).

The rest of the above-cited works model only semantics [1,4,17,2] or semantics with some syntax [3,5], and while some of them use record types [7,18,4,3], and some even (implicit) lump types [4,3] (cf. section 3.1), none does so in the way suggested by NLC. Another novelty, both in this group and for Coq, is the scope of NLC's implementation, which is so far the broadest (both in terms of the number of linguistic categories and levels of description involved).

[20] E.g. "syntactic structure is merely the characterization of the process of constructing a logical form, rather than a representational level..." [27], p. xi.

In contrast to all other formalisms, NLC integrates syntactic, morphological, and compositional semantic information in a single level of type. For example, a modern typed formalism like AACG [15] requires (besides using Categorial Grammar and category theory in addition to type theory) simply-typed lambda-calculus, two separately typed syntaxes (abstract and concrete), typed semantics, syntax-semantics and abstract-concrete syntax interfaces, and does not even cover compositional semantics, i.e. selectional restrictions. An advantage of (A)ACG [15,12], HPSG [24], GF [26] and many other formalisms is that they support language-specific constituent orders; in addition, ACG and AACG have truth-functional semantics. The first can be implemented with a language-specific function from formulas to strings, the second has been implemented (with an Ltac tactic, which is only one possibility for this) in Appendix A.

6 Discussion

We presented a new formalism, a type system NLC, as well as its implementation on a structurally diverse fragment of English. As compared to the usual three-level representation of compositional semantics, syntax and morphology, NLC reduces NL compositionality to a single level by using lump types and types for morphemes. This main novelty of the formalism is captured in rules AT-Intro and LT-Intro.

The implementation was done in Coq and, in hindsight, the choice of the programming language seems somewhat accidental. Quite possibly, it would have been easier to do this in some other language. The prime suspects are languages with type systems of roughly the same level of complexity, e.g. OCaml, Agda and Haskell. However, perhaps NLC could be also encoded in a qualitatively simpler type system. As we can have e.g. intersection types without dependent and polymorphic types, it is clear that the latter are not required for lump types. Also, subtyping polymorphism is attainable in several languages without dependent and polymorphic types. However, without dependent or polymorphic types, getting useful function types will be more tricky (if not altogether impossible).

As a representative fragment of NL morphology and syntax has not been directly formalized in a general-purpose programming language before[21], at the time of starting the project (in 2016), I was unaware of both the goal's feasibility and the eventual form it might take. Generally speaking, choosing a proof assistant for such work makes sense — if it is (theoretically) possible to formalize all mathematics in it, it must be also possible to formalize a substantial fragment of NL (provided there is a mathematical representation of it). There were some intuitions and ideas about the suitable representation to start with but, in the end, Coq "corrected" them profoundly. From the very beginning, as a methodological guideline, there was the idea to produce code that would not only represent but *look like* NL. The motivation behind this was an elementary

[21] It has been directly formalized in a specialized programming language (GF), and indirectly in a general-purpose programming language (Haskell, in which GF is written).

theory of NL expression as function (or perhaps relation) application. In the end, to get that kind of code, my problems mostly reduced to getting Coq's type inference to work. Coq has several (and sometimes quite involved) devices for this — custom notations, Ltac, type classes, canonical structures and more — but it may well be that the implementation-specific struggle would have been alleviated in some other language. On the other hand, there is no dedicated language for this kind of work[22] and Coq, even if an overkill, certainly qualifies as something that should get the work done. However, after giving all credit to Coq's complexity and relatively advanced type inference, I encourage the interested reader to experiment along these lines not only in Coq but a programming language of their choice.

References

1. Asher, N.: Selectional restrictions, types and categories. Journal of Applied Logic **12**(1), 75–87 (2014). https://doi.org/10.1016/j.jal.2013.08.002
2. Bekki, D., Asher, N.: Logical polysemy and subtyping. In: Motomura, Y., Butler, A., Bekki, D. (eds.) New Frontiers in Artificial Intelligence, pp. 17–24. Springer, Berlin, Heidelberg (2013)
3. Chatzikyriakidis, S., Luo, Z.: Natural language inference in Coq. Journal of Logic, Language and Information **23**(4), 441–480 (Dec 2014). https://doi.org/10.1007/s10849-014-9208-x
4. Chatzikyriakidis, S., Luo, Z.: Individuation criteria, dot-types and copredication: A view from modern type theories. In: Proceedings of the 14th Meeting on the Mathematics of Language (MoL 2015). pp. 39–50. Association for Computational Linguistics (2015). https://doi.org/10.3115/v1/W15-2304, `http://www.aclweb.org/anthology/W15-2304`
5. Chatzikyriakidis, S., Luo, Z.: Proof assistants for natural language semantics. In: Amblard, M., de Groote, P., Pogodalla, S., Retoré, C. (eds.) Logical Aspects of Computational Linguistics. Celebrating 20 Years of LACL (1996–2016). pp. 85–98. Springer, Berlin, Heidelberg (2016), `http://www.cs.rhul.ac.uk/~zhaohui/LACL16PA.pdf`
6. Constable, R.L.: Recent results in type theory and their relationship to Automath. In: Kamareddine, F.D. (ed.) Thirty Five Years of Automating Mathematics, pp. 37–48. Springer Netherlands, Dordrecht (2003)
7. Cooper, R.: Records and record types in semantic theory. Journal of Logic and Computation **15**(2), 99–112 (2005). https://doi.org/10.1093/logcom/exi004
8. Davidson, D.: Truth and meaning. Synthese **17**(3), 304–323 (1967), `http://www.jstor.org/stable/20114563`
9. Di Sciullo, A.M. (ed.): Asymmetry in Grammar: Volume 1: Syntax and semantics. John Benjamins (2003), `https://www.jbe-platform.com/content/books/9789027296801`
10. Di Sciullo, A.M. (ed.): Asymmetry in Grammar: Volume 2: Morphology, phonology, acquisition. John Benjamins (2003), `https://www.jbe-platform.com/content/books/9789027296795`

[22] GF does not count, as it does not allow to choose between mathematical formalisms.

11. Dryer, M.S.: Order of Subject, Object and Verb. In: Dryer, M.S., Haspelmath, M. (eds.) The World Atlas of Language Structures Online. Max Planck Institute for Evolutionary Anthropology, Leipzig (2013), http://wals.info/chapter/81
12. de Groote, P.: Towards abstract categorial grammars. In: Proceedings of 39th Annual Meeting of the Association for Computational Linguistics. pp. 252–259. Association for Computational Linguistics, Toulouse, France (Jul 2001). https://doi.org/10.3115/1073012.1073045, https://www.aclweb.org/anthology/P01-1033
13. Gruzitis, N., Dannélls, D.: A multilingual FrameNet-based grammar and lexicon for controlled natural language. Lang Resources & Evaluation **51**(1), 37–66 (2017). https://doi.org/10.1007/s10579-015-9321-8
14. Gruzitis, N.: Abstract meaning representation (amr) (2019), https://github.com/GrammaticalFramework/gf-contrib/tree/master/AMR
15. Kiselyov, O.: Applicative abstract categorial grammar. In: Kanazawa, M., Moss, L.S., de Paiva, V. (eds.) NLCS'15. Third Workshop on Natural Language and Computer Science. EPiC Series in Computing, vol. 32, pp. 29–38. EasyChair (2015). https://doi.org/10.29007/s2m4, https://easychair.org/publications/paper/RPN
16. Lambek, J.: The mathematics of sentence structure. The American Mathematical Monthly **65**(3), 154–170 (1958)
17. Luo, Z.: Type-theoretical semantics with coercive subtyping. In: Semantics and Linguistic Theory. vol. 20, pp. 38–56. Vancouver (2010)
18. Luo, Z.: Contextual analysis of word meanings in type-theoretical semantics. In: Pogodalla, S., Prost, J.P. (eds.) Logical Aspects of Computational Linguistics. pp. 159–174. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
19. Luuk, E.: Nouns, verbs and flexibles: implications for typologies of word classes. Language Sciences **32**(3), 349–365 (2010). https://doi.org/10.1016/j.langsci.2009.02.001
20. Montague, R.: The proper treatment of quantification in ordinary English. In: Portner, P., Partee, B.H. (eds.) Formal Semantics: The Essential Readings, pp. 17–34. Blackwell, Oxford (2002)
21. Moortgat, M.: Categorial type logics. In: van Benthem, J., ter Meulen, A. (eds.) Logic and Language, pp. 95–179. Elsevier, Amsterdam, North-Holland (2011)
22. Moot, R.: The Grail Theorem Prover: Type Theory for Syntax and Semantics. In: Luo, Z., Chatzikyriakidis, S. (eds.) Modern Perspectives in Type-Theoretical Semantics, Part III, vol. Studies in Linguistics and Philosophy, pp. 247–277. Springer (2017). https://doi.org/10.1007/978-3-319-50422-3_10, https://hal-lirmm.ccsd.cnrs.fr/lirmm-01471644
23. Morrill, G.: A chronicle of type logical grammar: 1935–1994. Research on Language and Computation **5**, 359–386 (09 2007). https://doi.org/10.1007/s11168-007-9034-2
24. Pollard, C., Sag, I.A.: Head-Driven Phrase Structure Grammar. University of Chicago Press, Chicago (1994)
25. Pustejovsky, J.: The Generative Lexicon. MIT Press, Cambridge, MA. (1995)
26. Ranta, A.: Grammatical Framework: a type-theoretical grammar formalism. The Journal of Functional Programming **14**(2), 145–189 (2004). https://doi.org/10.1017/S0956796803004738
27. Steedman, M.: The Syntactic Process. MIT Press, Cambridge, MA, USA (2000)

A Appendix

A.1 Truth-functionality

The fragment we have introduced so far is not truth-functional, i.e. we cannot compute truth values on it. In a sense, this does not matter, as we clearly do not need truth-functionality for compositional semantics, syntax and morphology. However, truth-functionality is certainly something that comes in handy in natural language inference (not to mention model-theoretic semantics, of what it is the foundation). For the latter reason, truth-functionality is a quality of NL formalisms that philosophers of language and students of Montague, in particular, have always insisted on. Fortunately, it is relatively easy to turn our fragment truth-functional.

The following is an optional extension of (our implementation of) NLC. For degenerate models (in which all sentences are uniformly true, false or undecidable) one only needs to add a coercion such as

```
Parameter s_prop:> S -> Prop. (* all S-s undecidable *)
```

For nontrivial models, a convenient solution is to define an Ltac match, e.g.

```
Parameter s_prop:> S -> Prop. (*coerce S to Prop*)
Notation ":>> x" := (s_prop (s1s2 (s0s1 x))) (at level 179).
Notation "$ x" := ltac:(let u := constr:(x:Prop) in
lazymatch u with
  | :>> (PAST _ _) => exact True
  | s_prop (s1s2 (however (PAST _ _))) => exact True
  | :>> (PRES sleep john) => exact True
  | :>> (variad_S0 _ _ _ (PRES walk (-s boy))) => exact True
  | :>> (PRES _ _) => exact False
  | :>> (variad_S0 _ _ _ (PRES _ _)) => exact False
  | :>> (and (PAST _ _) (PAST _ _)) => exact True (*sleep..*)
  | :>> (and (PAST _ _) (variad_S0 _ _ _ (PAST _ _)))  => exact True (*walk..*)
  | :>> (and (PAST _ _) (PRES sleep john)) => exact True
  | :>> (and (PAST _ _) (variad_S0 _ _ _ (PRES walk (-s boy))))  => exact True
  | :>> (and ?m ?n) => idtac "unanalyzed:" m n
  | s_prop (s1s2 (however ?n)) => idtac "unanalyzed:" n
  | x => idtac "unanalyzed:" x": Prop"
  (*leave some potentially t-functional constructs unanalyzed*)
  end) (at level 199).
```

to get the axioms we want[23], and then proceed by theorem proving based on them:

```
(*a trivial proof that "boys don't sleep" and "john sleeps"*)
Theorem pres: (~ ($ (PRES sleep (-s boy)))) /\ ($ (PRES sleep john)).
Proof. firstorder. Qed.
```

[23] Notice that we use the `let` construct to get `x:Prop`.

B Appendix

```
(* Compound types for NL modeling: application, record, product and pi-types; written and
best viewed in Coq 8.9 --- https://gitlab.com/jaam00/nlc/blob/master/compound.v *)

Module Types. (*shared types*)

Inductive ST := F | N | PN. (*stem: flexible, noun, proper name*)
Inductive NUM := SG | PLR. (*number: singular, plural*)
Inductive SR := Phy | Sen | Inf | Lim. (*selectional restriction: physical, sentient..*)
Inductive CA := NOM | GEN | LAT | DIR | LOC. (*case/adposition: nominative, genitive..*)

End Types.

Module Application. (*----------------------*)
Export Types.

Parameter STM PLU XP: SR -> CA -> NUM -> ST -> Type.
Parameter hut: STM Phy NOM SG N.
Parameter thought: STM Inf NOM SG N.

Structure CSTM := cx {g: SR -> CA -> NUM -> ST -> Type}.
Canonical Structure cx_x: CSTM := cx STM.
Canonical Structure cx_x': CSTM := cx PLU.

Parameter red: forall {x y z w}, g x Phy y z w -> g x Phy y z w.
Parameter a: forall {x y z w}, g x y z SG w -> XP y z SG w.
Parameter the: forall {x y z u w}, g x y z u w -> XP y z u w.
Parameter PL: forall {x y z w}, g x y z SG w -> PLU y z PLR w.

End Application.

Module Record. (*----------------------*)
Export Types.

Inductive CAT := STM | PLU | XP. (*stem, plural, XP*)
Record POS := mkPOS { cw:CAT; cw0:SR;
cw1:CA; cw2:NUM; cw3:ST }.
Parameter co:> POS -> Set.
Parameter hut: mkPOS STM Phy NOM SG N.
Parameter thought: mkPOS STM Inf NOM SG N.

Structure CSTM := cx {g: CAT}.
Canonical Structure cx_x: CSTM := cx STM.
Canonical Structure cx_x': CSTM := cx PLU.

Parameter red: forall {x y z w}, mkPOS (g x) Phy y z w -> mkPOS (g x) Phy y z w.
Parameter a: forall {x y z w}, mkPOS (g x) y z SG w -> mkPOS XP y z SG w.
Parameter the: forall {x y z u w}, mkPOS (g x) y z u w -> mkPOS XP y z u w.
```

```
Parameter PL: forall {x y z w}, mkPOS (g x) y z SG w -> mkPOS PLU y z PLR w.

End Record.

Module Product. (*----------------------*)
Export Types.

Inductive CAT := STM | PLU | XP.
Open Scope type.
Definition POS := CAT*SR*CA*NUM*ST.
Parameter co:> POS -> Set.
Definition pos x y z w u := (x, y, z, w, u): POS.
Parameter hut: pos STM Phy NOM SG N.
Parameter thought: pos STM Inf NOM SG N.

Structure CSTM := cx {g: CAT}.
Canonical Structure cx_x: CSTM := cx STM.
Canonical Structure cx_x': CSTM := cx PLU.

Parameter red: forall {x y z w}, pos (g x) Phy y z w -> pos (g x) Phy y z w.
Parameter a: forall {x y z w}, pos (g x) y z SG w -> pos XP y z SG w.
Parameter the: forall {x y z u w}, pos (g x) y z u w -> pos XP y z u w.
Parameter PL: forall {x y z w}, pos (g x) y z SG w -> pos PLU y z PLR w.

End Product.

Module Pi. (*----------------------*)
Export Types.

Inductive CAT := STM | PLU | XP.
Parameter POS: forall (cw:CAT)(cw0:SR)
(cw1:CA)(cw2:NUM)(cw3:ST), Type.
Parameter hut: POS STM Phy NOM SG N.
Parameter thought: POS STM Inf NOM SG N.

Structure CSTM := cx {g: CAT}.
Canonical Structure cx_x: CSTM := cx STM.
Canonical Structure cx_x': CSTM := cx PLU.

Parameter red: forall {x y z w}, POS (g x) Phy y z w -> POS (g x) Phy y z w.
Parameter a: forall {x y z w}, POS (g x) y z SG w -> POS XP y z SG w.
Parameter the: forall {x y z u w}, POS (g x) y z u w -> POS XP y z u w.
Parameter PL: forall {x y z w}, POS (g x) y z SG w -> POS PLU y z PLR w.

End Pi.

Module Test. (*comment out imports as required*)
Import (* Application *) (* Record *) Product (* Pi *).

Check red hut.
```

```
Check a thought.
Fail Check red thought.
Check a (red hut).
Check the (red hut).
Check the (red (PL hut)).
Fail Check red (a hut).
Fail Check a (red (PL hut)).
Fail Check a (the (red hut)).
Fail Check the a.

End Test.
```

C Appendix

```
(* The following examples are generously commented and should be self-explanatory *)
(*from https://gitlab.com/jaam00/nlc/blob/master/frag.v*)

Check PAST throw john. (*"John threw" type checks..*)
Fail Check PAST throw john: S. (*..but not as a sentence.*)
Check PAST throw john (-s stone): S.
Check PAST throw john (-s stone) (at (the hut)): S.
(*"At the hut" can be the 3rd argument of "throw" (above),
but not the 2nd or 1st one:*)
Fail Check PAST throw john (at (the hut)).
Fail Check PAST throw (a hut).
Check PAST throw [a hut]. (*..except when in brackets (works only if SRs are off).*)

Fail Check PAST throw john (-s stone) (in (the hut)).
(*"In the hut" cannot be an argument of "throw"..*)
Check in (the hut) (PAST throw john (-s stone)): S.
(*..but it can be a sentence modifier..*)
Check at (every hut) (PAST throw john (-s stone)): S.
(*..and so can "at every hut"..*)
Check however (PAST throw john (-s stone)): S.
(*..and sentential adverbs like "however".*)
Fail Check and (PAST throw john (a stone)) john.
(*Connectives cannot range over a sentential
and nominal argument, but can range over nominal..*)
Check and (every john) (all (the (-s boy))).
Check and (PAST walk (-s boy) (to (all (the (-s hut)))))
(PAST sleep john). (*..or sentential arguments..*)
Check and (PAST walk john) (PAST sleep john).
(*..(also w/ optional arguments omitted).*)
Fail Check in (all (the hut)). (*ungrammatical?*)
Check in (the (entire hut)). (*grammatical*)
Check madly ((in (the (entire hut))) ((PAST walk) (all (the (-s boy))))): S.
(*"All the boys walked madly in the entire hut" is a sentence*)

(*Examples of compound types:*)
```

```
Check john: XP0 (hu H_Phy) NOM SG (Pn M1).
(*"John" is an XP, human, proper name, male,
in nominative, singular, a physical entity..*)
Check john: XP0 (hu H_Lim) NOM SG PN+. (*..and a limbed entity.*)
Check john: XP0 Lim NOM SG (Pn M1). (*The normalized version of the above.*)
Check the john: XP2 _ NOM SG (Pn M1). (*"The John" is another type of XP.*)
Check and (a (good throw)) (entire (-s throw)).
(*"Throw" is a flexible (a function or argument (as above))..*)
Check and (PRES stone john (a boy)) (PAST throw john (-s stone)
((at (-s stone)))). (*..and "stone" likewise.*)
(*So "at stones" is an XP, flexible in lative,
plural and a physical entity:*)
Check at (-s stone): XP2 Phy LAT PLR F.
(*As prescribed by "red", "red John" is a physical entity:*)
Check red john: XP0 Phy NOM SG (Pn M1).
Check [red john]: XP0 Lim NOM SG PN+.
(*..but we can make it into a limbed one w/ our bracket notation (above).
"PN+" means proper name w/ gender info*)
Check hut: STM Phy _ _ _.
(*"Hut" is a stem and always a physical entity..*)
Check [hut]: STM Lim _ _ _.
(*..so we can make it into a limbed one only if SRs are off
(the above line fails if SRs are on).*)
Check and (the (entire hut)) (all (-s john)): XP2 Phy NOM _ _.
(*"The entire hut and all Johns" is an XP
and physical entity in nominative..*)
Check and (the (entire hut)) (all (-s john)): XP2 Phy ACC' _ _.
(*..or pseudo-accusative (by zero-derivation)..*)
Check [and (the (entire hut)) (all (-s john))]: XP2 Sen ACC' _ _.
(*..which can be made into a sentient entity if SRs are off.*)

(*Arguments of verbs and flexibles must be proper XPs:*)
Fail Check PAST throw (the boy) (entire stone) (to (him)).
Check PAST throw (the boy) (the (entire stone)) (to (him)): S.
(*Arguments of V and F must have correct
cases and selectional restrictions:*)
Check PAST throw he (all (-s stone)) (at (john)): S.
Check PAST throw john (all (the (-s stone))) (at (him)): S.
Fail Check PAST throw him (all (-s stone)) (at john). (*wrong case*)
Fail Check PRES throw john (a walk). (*wrong case and SR*)

(*from https://gitlab.com/jaam00/nlc/blob/master/cop.v*)

Fail Check are (PL man) (PL hut). (*"men are huts" won't do*)
Fail Check is i me. (*fails as desired (because gp.. doesn't match XP..)*)
Fail Check PRES know i (who (is ill ill)). (*"i know ill who is ill" won't do*)
Check TAM know i (who (are ill (PL man))).
(*"i know men who are ill" (TAM is tense-aspect-mood marker)*)
Check PRES know i (who (PRES COP ill (the man))). (*i know the man who is ill*)
Check TAM know i (who (are ill (the (PL man)))). (*i know the men who are ill*)
```

```
Fail Check TAM know i (who (are ill (a (PL man)))).
(*"i know a men who are ill" won't do*)
Check PAST know i (who (TAM COP ill (a man))). (*i knew a man who is ill*)
Check TAM know i (who (PAST COP ill (the man))). (*i know the man who is ill*)

Fail Check PRES PRES know i me.
Fail Check PAST (PAST know) i me.
Check PAST know i (the hut).
Check TAM know (a man) (the hut). (*a man knows the hut*)

Fail Check PRES know i the.
Fail Check PRES know i i.
Fail Check know i me. (*TAM required, know is nonfinite*)
Check PAST know i me.
Check PRES know i me.
Check TAM know i (who (is (the man) me)). (*i know the man who is me*)
Fail Check PRES know i (who (is (the man) i)).  (*wrong case*)
```