Proceedings of the 12th Conference on Language Resources and Evaluation (LREC 2020), pages 5257–5266
Marseille, 11–16 May 2020

ÆTHEL: Automatically Extracted Typelogical Derivations for Dutch

Konstantinos Kogkalidis, Michael Moortgat, Richard Moot
Utrecht Institute of Linguistics OTS, Utrecht University, LIRMM, Université de Montpellier, CNRS
{k.kogkalidis, m.j.moortgat}@uu.nl, richard.moot@lirmm.fr

Abstract

We present ÆTHEL, a semantic compositionality dataset for written Dutch. ÆTHEL consists of two parts. First, it contains a lexicon of supertags for about 900 000 words in context. The supertags correspond to types of the simply typed linear lambda-calculus, enhanced with dependency decorations that capture grammatical roles supplementary to function-argument structures. On the basis of these types, ÆTHEL further provides 72 192 validated derivations, presented in four equivalent formats: natural-deduction and sequent-style proofs, linear logic proofnets and the associated programs (lambda terms) for meaning composition. ÆTHEL's types and derivations are obtained by means of an extraction algorithm applied to the syntactic analyses of Lassy Small, the gold standard corpus of written Dutch. We discuss the extraction algorithm and show how 'virtual elements' in the original Lassy annotation of unbounded dependencies and coordination phenomena give rise to higher-order types. We suggest some example usecases highlighting the benefits of a type-driven approach at the syntax semantics interface. The following resources are open-sourced with ÆTHEL: the lexical mappings between words and types, a subset of the dataset consisting of 7 924 semantic parses, and the Python code that implements the extraction algorithm.

Keywords: Categorial Grammar, Grammar Extraction, Dutch Treebank

1. Introduction

Typelogical categorial grammars offer an attractive view on the syntax-semantics interface. The key idea is that the derivation establishing the well-formedness of a phrase systematically corresponds to a procedure that computes the meaning of that phrase, given the meanings of its constituent words.

This connection between logical derivations and programs, known as the Curry-Howard correspondence (Sørensen and Urzyczyn, 2006), plays a central role in current typelogical grammars. In extended versions of the Lambek calculus (Lambek, 1958) such as (Morrill, 1994; Moortgat, 1996; Morrill et al., 2011; Kubota and Levine, to appear), the Curry-Howard correspondence applies to a language of *semantic* types, where meaning composition is addressed. The semantic type language is obtained by a structure-preserving translation from the *syntactic* type system that handles surface word order, prosodic structure and the like. Alternatively, the type system of Abstract Categorial Grammar (De Groote, 2001) and Lambda Grammars (Muskens, 2001) is designed to capture a 'tectogrammatical' (Curry, 1961) view of language structure that abstracts away from the surface realization. From the abstract syntax now both the meaning representations and the surface forms are obtained by compositional translations, where for the latter one relies on the standard encoding of strings, trees, etc in lambda calculus (Barendregt et al., 2013).

A common feature of these approaches is the use of *resource-sensitive* type logics, with Linear Logic (Girard, 1987) being the key representative. Curry's original formulation of the logic-computation correspondence was for Intuitionistic Logic, the logic of choice for the formalization of mathematical reasoning. For many real-world application areas, Linear Logic offers a more attractive alternative. In Linear Logic, formulas by default stand for non-renewable resources that are used up in the course of a derivation; free copying or deletion is not available.

The resource-sensitive view of logic and computation fits the nature of grammatical composition particularly well. Despite this, linear logic is underrepresented when it comes to linguistic resources that exhibit its potential at the syntax-semantics interface. Motivated by this lack of resources, we introduce ÆTHEL[1], a dataset of typelogical derivations for Dutch. The type logic ÆTHEL employs rests on the non-directional types of simply typed intuitionistic linear logic, enhanced with modal operators that allow the inclusion of dependency information on top of simple function-argument relations. ÆTHEL consists of 72 192 sentences, every word of which is assigned a linear type that captures its functional role within the phrase-local structure. Under the parsing-as-deduction view, these types give rise to proofs, *theorems* of the logic showing how the sentence type is derived from the types of the words it consists of. These derivations are presented in four formats which, albeit being morally equivalent, can be varyingly suitable for different tasks, viz. proofnets, natural deduction and sequent-style proofs, and the corresponding λ-terms. ÆTHEL is generated by an automated extraction algorithm applied to Lassy Small (van Noord et al., 2013), the gold standard corpus of written Dutch. The validity of the extracted derivations is verified by LinearOne (Moot, 2017), a mechanized theorem prover for first-order linear logic.

The structure of the paper is as follows: we begin in Section 2 by providing an overview of the type logic used. We proceed by detailing the proof extraction algorithm in Section 3, before describing the dataset and its accompanying resources in Section 4. We give some conclusive remarks in Section 5.

Related work The work reported on here shares many of the goals of the CCGBank project (Hockenmaier and Steedman, 2007).

A difference is that CCGBank uses syntactic types aiming at wide-coverage parsing, whereas our linear types are

[1]ÆTHEL: Automatically Extracted Theorems from Lassy

geared towards meaning composition. Also different is the treatment of long-range dependencies, where CCG establishes the connection with the 'gap' by means of type-raising and composition rules. From a logical point of view, these CCG derivations are suboptimal: they contain detours that disappear under proof normalization. In our approach, long-range gaps are hypotheses that are withdrawn in a single inference step of conditional reasoning.

2. Type Logic

2.1. ILL$_{\multimap}$

The core of the type system used in this paper is ILL$_{\multimap}$, the simply-typed fragment of Intuitionistic Linear Logic. ILL$_{\multimap}$ types are defined by the following inductive scheme:

$$\mathrm{T} ::= \mathrm{A} \mid \mathrm{T}_1 \multimap \mathrm{T}_2$$

where $\mathrm{A} \in \mathcal{A}$ is a set of *atomic* types and linear implication ($\multimap$) is the only type-forming operation for building complex types. Atomic types are assigned to phrases that can be considered complete in themselves. A complex type $\mathrm{T}_1 \multimap \mathrm{T}_2$ denotes a function that takes an argument of type T_1 to yield a value of type T_2. Such function types are *linear* in the sense that they actually consume their T_1 argument in the process of producing a T_2 result.

Given Curry's "proofs as programs" approach, one can view the inference rules that determine whether a conclusion B can be derived from a multiset of assumptions $\mathrm{A}_1, \ldots, \mathrm{A}_n$ as the *typing rules* of a functional programming language. The program syntax is given by the following grammar:

$$M, N ::= x \mid MN \mid \lambda x.M$$

Typing judgements take the form of sequents

$$x_1 : \mathrm{A}_1, \ldots, x_n : \mathrm{A}_n \vdash M : \mathrm{B}$$

stating that a program M of type B can be constructed with parameters x_i of type A_i. Such a program, in the linguistic application envisaged here, is a procedure to compute the meaning of a phrase of type B, given meanings for the constituting words w_i as bindings for the parameters x_i. The typing rules then comprise the Axiom $x : \mathrm{A} \vdash x : \mathrm{A}$, together with inference rules for the elimination and introduction of the linear implication, where ($\multimap E$) corresponds to function application, and ($\multimap I$) to function abstraction.

$$\frac{\Gamma \vdash M : \mathrm{A} \multimap \mathrm{B} \quad \Delta \vdash N : \mathrm{A}}{\Gamma, \Delta \vdash (M\ N) : \mathrm{B}} \multimap E \tag{1}$$

$$\frac{\Gamma, x : \mathrm{A} \vdash M : \mathrm{B}}{\Gamma \vdash \lambda x.M : \mathrm{A} \multimap \mathrm{B}} \multimap I \tag{2}$$

The abstraction rule ($\multimap I$) comes into play in reasoning about higher-order types. We say that atomic types are of order zero. For function types, $\mathit{order}(\mathrm{T}_1 \multimap \mathrm{T}_2)$ is defined as $\max(\mathit{order}(\mathrm{T}_1) + 1, \mathit{order}(\mathrm{T}_2))$. A type of order $n > 1$ is a function that has a function as one of its arguments. A relative pronoun, for example, could be typed schematically as $(\mathrm{NP} \multimap \mathrm{S}) \multimap \textsc{Rel}$. The import of this type is that the relative clause body is an S composed with the aid of an NP hypothesis (the 'gap'); this hypothesis is withdrawn by the ($\multimap I$) rule.

2.2. Dependency enhancement

As it stands, our type system doesn't have the expressivity to capture *grammatical role* information. For example, a ditransitive verb such as 'give' would be typed as $\mathrm{NP} \multimap \mathrm{NP} \multimap \mathrm{NP} \multimap \mathrm{S}$[2]; we would like to be able to distinguish the subject, direct object and indirect object among the three NP arguments.

To address this limitation, we use the modalities of Multimodal Categorial Grammars (Moortgat, 1996). Modalities are unary type-forming operations, commonly used for syntactic control, licensing restricted forms of restructuring or reordering, or blocking overgenerating applications of such operations. Here instead, we use them as a means of injecting dependency information directly into the type system in the form of feature decorations[3].

We augment the type system with unary operators $\diamond^d$ and $\Box^d$, where the d labels are taken from the set of dependency roles (subject, object, modifier, etc) in the corpus. The operators $\diamond^d$ and $\Box^d$ come with their own Elimination and Introduction rules. We assume that the dependency annotations do not affect the Curry-Howard program terms associated with a proof, so we can formulate these inference rules purely on the type level. For our purposes, $\diamond^d$ Introduction and $\Box^d$ Elimination play a key role.

$$\frac{\Gamma \vdash \mathrm{A}}{\langle\Gamma\rangle^d \vdash \diamond^d\mathrm{A}} \diamond I \qquad \frac{\Gamma \vdash \Box^d\mathrm{A}}{\langle\Gamma\rangle^d \vdash \mathrm{A}} \Box E \tag{3}$$

The ($\diamond^d I$) rule says that if from resources Γ one can derive a phrase of syntactic type A, then one can obtain an A phrase with dependency role d by grouping together the Γ resources with the delimiter $\langle\cdot\rangle^d$, indicating that they consitute a *dependency domain*. The ($\Box^d E$) rule, similarly, encloses the Γ resources with the dependency delimiter $\langle\cdot\rangle^d$, this time by unpacking the $\Box^d A$ type of the premise, so that reasoning can now proceed with the A subtype.

The $\diamond^d$ and $\Box^d$ modalities allow us to reconcile the demands of the logical function-argument structure of a phrase with those of its dependency structure (Moortgat and Morrill, 1991).

(i) We use $\diamond^d\mathrm{A} \multimap \mathrm{B}$ for a function acting as a *head* that selects for an A *complement* and assigns it the dependency role d by means of ($\diamond^d I$). For example, intransitive verbs are typed $\diamond^{\mathrm{su}}\mathrm{NP} \multimap \mathrm{S}$.

(ii) We use $\Box^d(\mathrm{A} \multimap \mathrm{B})$ for *non-head* functions, i.e. functions that, in the dependency structure, are dependents with respect to their argument. For example, $\Box^{\mathrm{mod}}(\mathrm{NP} \multimap \mathrm{NP})$ for a noun-phrase modifier, or $\Box^{\mathrm{det}}(\mathrm{N} \multimap \mathrm{NP})$ for a determiner.

To complete the tour of the type system, the *nested* implications of higher-order types come with a structural modality ! on their argument: $!\,\mathrm{A} \multimap \mathrm{B}$. The purpose of the ! modality is to allow for the A hypothesis that is withdrawn in the ($\multimap I$) step to originate from an embedded dependency domain.

[2]Type brackets can be ommitted reading the $\multimap$ operation as right-associative.

[3]For earlier applications of modalities-as-features, see (Heylen, 1999; Johnson, 1999).

In practice, the dependency refinement serves three main purposes. From a semantic perspective, the added operators can be meaningful in the interpretation of the type system, allowing distinct composition recipes for types which would otherwise be equated. Further, through subsuming dependency label information, they allow for a backwards conversion into dependency-based syntactic frameworks. Finally, from a parsing perspective, $\mathrm{ILL}_{\multimap}$ by itself is word-order agnostic, meaning it admits more proofs than linguistically desired. The dependency decorations in this respect act as a balancing counter-weight, which on the one hand increases lexical type ambiguity, but on the other hand provides valuable information to constrain proof search.

For the time being, we specify the dependency-decorated types at the lexical level of our provided proofs, but refrain from incorporating the structural rules imposed for reasons of brevity and simplicity.

3. Extraction

The formulation of the extraction process takes its inspiration from (Moortgat and Moot, 2002; Moot, 2015), modulo some adaptations to account for the discrepancies in the type-logic and the source corpus. Algorithmically, it may be perceived as a pipeline of three distinct components. The first one concerns the transformation of the syntactically annotated input sentences into a *directed acyclic graph* (DAG) format that satisfies the input requirements of the remainder of the algorithm. The intermediate one is responsible for assigning types to the DAG's nodes and asserting their validity within the phrase-local context. Finally, the third component accepts the type-decorated DAG and identifies the interactions between its constituent types, thereby transforming it into a typelogical derivation.

3.1. Lassy

Lassy (van Noord et al., 2013) is a dataset of syntactically annotated written Dutch. It is subdivided in two parts, Lassy Small and Lassy Large, both of which have been automatically annotated by the Alpino parser (Bouma et al., 2001). For the purposes of this work we focus our attention at the gold-standard Lassy Small, which (unlike its sibling) has been manually verified and corrected. Lassy Small enumerates about 65 000 sentences (or a million words), originating from various sources such as e-magazines, legal texts, manuals, Wikipedia content and newspaper articles among others. The Lassy annotations are essentially DAGs, where nodes correspond to words and phrases labeled with lexical and phrasal categories, connected with edges that capture the syntactic functions between them[4]. DAGs allow for reentrancy, whereby a node has multiple incoming dependency edges. In the Lassy annotation, cases of reentrancy are handled via phantom syntactic nodes coindexed with nodes that carry real content, so as to allow the annotation DAG to be visually represented as a tree.

[4]A concise description of the syntactic category tags and dependency labels of Lassy can be found at http://nederbooms.ccl.kuleuven.be/eng/tags

3.2. Preprocessing

The extraction algorithm is formulated in terms of operations on DAGs. The first step mandated is therefore to collapse all coindexed duplicate nodes of each word or phrase into one, which inherits all the incoming edges of the original tree.

Next, we wish to treat annotation instances that are problematic for our type logic, stemming from schemes that under-specify the phrase structure. The two usual culprits are discourse-level annotations, which do not exhibit a consistent function-argument articulation, and multi-word units, the children of which are syntactically indiscernible. In the first case, we erase branches related to discourse structure (in many cases, these consist simply of structures annotated as "discourse unit" with substructures each marked as "discourse part"). To minimize the amount of annotations lost, we reconstruct an independent sample from each disconnected subgraph positioned under the cut-off point. To resolve multi-word phrases without resorting to ad-hoc typing schemes, we merge participating nodes, essentially treating the entire phrase as a single word. Punctuation symbols are dropped, as they are left untreated by the original annotation.

Beyond ensuring compatibility, we apply a number of transformations to Lassy's annotations designed to homogenize the types extracted. First, we remove the phantom syntactic nodes used to express the 'understood' subject or object of non-finite verb forms, since this is semantic information that does not belong in a proper syntactic annotation. Second, we replace the generic *conj* and *mwu* tags (for conjunctions and multi-word units, respectively), by recursively performing a majority vote over their daughters' tags. Finally, we refine the generic *body* label by specifying the particular kind of clause it applies to, thus splitting it into *rhd body*, *whd body* and *cmp body* for body of a relative, wh- and comparative clause respectively.

Edge erasures performed by the above procedures might lead to artifacts, which we take measures against. We redeem DAGs with multiple sources by creating a distinct graph for each source, constituted by (a replica of) the source-reachable subset of the original graph. We remove redundant intermediate nodes with a single incoming and a single outgoing edge, and redirect the incoming edge to its corresponding descendant. Conjunctions left with no more than one conjunct are truncated and replaced by their sole daughter.

3.3. Type Assignment

The output of the preprocessing procedure is a number of rooted DAGs per Lassy sample. The goal now is to assign types to the nodes of these DAGs, according to the dependency-decorated scheme of Section 2. We begin this endeavour by specifying two look-up tables that the type assignment algorithm is parametric to; one from part-of-speech tags and phrasal categories into atomic types, and one from dependency labels into modal operators (refer to Appendix 1 for details). Further, we specify the edge labels that correspond to phrasal heads and the ones that correspond to modifiers. We then formulate the type assignment process as a conditional iteration over a DAG, with each it-

eration progressively inferring and assigning the types of a subset of the DAG's nodes, and the termination condition being the absence of any untyped nodes.
In abstract terms, the algorithm looks as follows:

```
TYPEDAG :: DAG $d$ → DAG
  RETURN LAST (UNFOLD STEP $d$)
STEP :: DAG $d$ → OPTION DAG
  $d'$ ← LAST (UNFOLD TYPESIMPLE $d$)
  $d'$ ← LAST (UNFOLD TYPENONLOCAL $d'$)
  $d'$ ← LAST (UNFOLD TYPECONJUNCTIONS $d'$)
  RETURN $d'$ IF $d' \neq d$ else NOTHING
```

The iteration loop consists of three steps, each being the unfold of a function that takes a DAG, selects its fringe of typeable nodes, assigns a type to each node within, and returns the new (partially) typed DAG. Each of these three functions differs in how it selects for its fringe and how it manufactures type assignments. We detail their functionality in the next paragraphs; an illustrative example is also given in Figure 1a.

Simple Clauses The first step internalizes the typing of sub-graphs within the DAG that exhibit simple syntactic clauses.
First off, we select *context-independent* nodes, i.e. the root, plus those that are either leaves or have all of their daughters typed (possibly excluding modifier- and head-labeled daughters) and are not targets of exclusively modifying or head-labeled edges themselves; we type them by simply translating their part-of-speech or phrasal category tags into the corresponding atomic type.
The next step is to assign a type to nodes acting as *phrasal heads* and *modifiers*. These may only be typed insofar as both their parents and all of their siblings are typed, imposing the analogous fringe conditions. We assign phrasal heads the complex type:

$$\diamond^{d_1}\mathrm{A}_1 \multimap \ldots \multimap \diamond^{d_n}\mathrm{A}_n \multimap \mathrm{R},$$

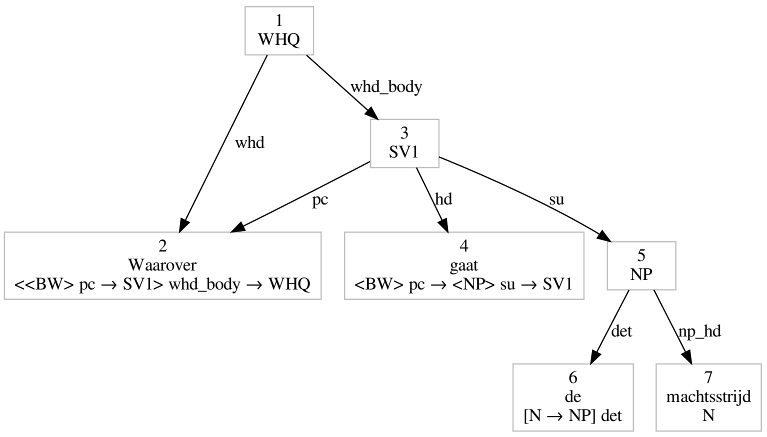
with $\diamond^{d_1}\mathrm{A}_1, \ldots, \diamond^{d_n}\mathrm{A}_n$ the list of dependency-decorated types for the *complements* and R the phrasal type for the *result*. In order to obtain a consistent currying of multi-argument function types, we appeal to an obliqueness ordering of the dependency labels (see e.g. (Dowty, 1982)), and have the curried n-ary function type consume its n arguments from most oblique $\diamond^{d_1}\mathrm{A}_1$ to least oblique $\diamond^{d_n}\mathrm{A}_n$. We refer to Appendix 2 for details of the obliqueness order assumed here.
Modifiers are treated as non-head functors; their typing is based on the polymorphic scheme $\Box^d(\mathrm{R} \multimap \mathrm{R})$, i.e. they are typed as endomorphisms of the phrasal type they are modifying, fixed to determine the dependency role d they realize.
Similarly, *determiners* are also treated as non-head functors: nouns are recognized as the syntactic heads of a noun phrase, in which determiners still assume the functor role. As such, they are assigned a boxed type $\Box^{\mathrm{det}}(\mathrm{N} \multimap \mathrm{NP})$.
In order to obtain proper higher-order type assignments (with functors deriving functors), head (and determiner) nodes are typed in a bottom-up fashion, whereas modifier nodes are typed top-down.

Non Local Dependencies Lassy treats non-local phenomena such as relative clauses and constituent questions by inserting a *secondary* edge pointing from a phrasal node embedded (possibly deeply) within the relative or question clause body on to the relativizing or interrogative pronoun. Such pronouns then serve two roles. In their primary role, they act as head of the relative or interrogative clause they project. In their second role, they contribute to the composition of the dependency domain of the subclause where their secondary edge has its origin. In this respect, they can be distinguished from other nodes due to having multiple incoming edges, the labels of which are distinct from one another. Both of these two roles have already been addressed by the algorithm, but only in isolation.
To reconcile this, we select our fringe as nodes falling under this construction, and which have already been assigned some implicational type $\diamond^d\mathrm{X} \multimap \mathrm{Y}$ depicting the top-level clause functor (conforming to the flow of the first iteration step). We then inspect the secondary dependency edge originating from (a subgraph of) X in order to update the aforementioned type to $\diamond^d(\diamond^e\mathrm{E} \multimap \mathrm{X}) \multimap \mathrm{Y}$.
The updated higher-order type has a nested implication: from the parsing-as-deduction perspective this means that in order to obtain a result of type Y, the relative or question pronoun has to assemble its argument X with the aid of a hypothesis $\diamond^e\mathrm{E}$. Hypothetical reasoning (the $\multimap$ Introduction rule) is a key feature of our typelogical toolkit; it obviates the need for phantom 'gap' categories in unbounded dependency constructions.
The identity of $\diamond^e$ and E depends on what the label of the secondary dependency edge is. In case of an argument edge, it is simply the pair of translations from the dependency label and the node's part of speech or phrasal syntactic tag respectively. In case of a modifying edge, the hypothesis subtype E is itself a box functor $\Box^m(\mathrm{Z} \multimap \mathrm{Z})$, corresponding to the endomorphism on the type Z of the edge's source node. In both cases, if the source node is in fact a sub-graph of X, the hypothesis type $\diamond^e\mathrm{E}$ is prefixed with the structural ! operator, allowing it to traverse intermediate dependency domains as necessary to reach the point where ($\multimap I$) can withdraw it.
For any non-terminal nodes the type of which has been altered, we iteratively update the types of all heads and modifiers lying underneath in the DAG so as to account for the new phrasal type.

Conjunctions Phenomena of coordination and ellipsis pose a challenge for our resource-sensitive type logic: deriving conjunction types for incomplete phrases from the types of the complete conjuncts would require copying in the logic, an operation that our linear type system rules out. Our approach here is to replace syntactic copying in the logical derivations by lexical polymorphism. More precisely, *coordinators* are typed according to the generically polymorphic scheme:

$$\diamond^{\mathrm{cnj}}\mathrm{X} \multimap \ldots \multimap \diamond^{\mathrm{cnj}}\mathrm{X} \multimap \mathrm{X},$$

where the number of arguments is concurring with the number of conjuncts. To determine the value of X, we take the following steps. We begin by setting X_0 as the atomic



(a) The dependency graph, as typed by the extraction component. The algorithm begins by simply translating the context-independent root 1 and leaf nodes 2 and 7 to WHQ, BW and N respectively. Given the absence of untyped children (modulo heads), intermediate nodes 3 and 5 are then typed as SV1 and NP, followed by the typing of functor nodes 4 and 6 as $\diamond^{\text{pc}}\text{BW} \multimap \diamond^{\text{su}}\text{NP} \multimap \text{SV1}$ and $\Box^{\text{det}}(\text{N} \multimap \text{NP})$. As 2 is also a phrasal head, its type is initially updated to $\diamond^{\text{whd body}}\text{SV1} \multimap \text{WHQ}$. Finally, the second iteration step alters the type of 2 to account for the hypothetical gap, giving it its final assignment $\diamond^{\text{whd body}}(\diamond^{\text{pc}}\text{BW} \multimap \text{SV1}) \multimap \text{WHQ}$.

$$\dfrac{\dfrac{\text{Waarover}}{\vdash \diamond^{\text{whd body}}(\diamond^{\text{pc}}\text{BW} \multimap \text{SV1}) \multimap \text{WHQ}}\,\mathcal{L} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{gaat}}{\vdash \diamond^{\text{pc}}\text{BW} \multimap \diamond^{\text{su}}\text{NP} \multimap \text{SV1}}\,\mathcal{L} \quad \dfrac{}{\diamond^{\text{pc}}\text{BW} \vdash \diamond^{\text{pc}}\text{BW}}\,\mathcal{A}}{\diamond^{\text{pc}}\text{BW},\ \text{gaat} \vdash \diamond^{\text{su}}\text{NP} \multimap \text{SV1}}\multimap E \quad \dfrac{\dfrac{\dfrac{\dfrac{\text{de}}{\vdash \Box^{\text{det}}\text{N} \multimap \text{NP}}\,\mathcal{L}}{(\text{d})_{\text{det}} \vdash \text{N} \multimap \text{NP}}\,\Box^{\text{det}}I \quad \dfrac{\text{machtstrijd}}{\vdash \text{N}}\,\mathcal{L}}{(\text{d})_{\text{det}}\ \text{m} \vdash \text{NP}}\multimap E}{\big((\text{d})_{\text{det}}\ \text{m}\big)_{\text{su}} \vdash \diamond^{\text{su}}\text{NP}}\,\diamond^{\text{su}}I}{\diamond^{\text{pc}}\text{BW},\ \text{g}\ \big((d)_{\text{det}}\ \text{m}\big)_{\text{su}} \vdash \text{SV1}}\multimap E}{\text{g}\ \big((\text{d})_{\text{det}}\ \text{m}\big)_{\text{su}} \vdash \diamond^{\text{pc}}\text{BW} \multimap \text{SV1}}\multimap I}{\big(\text{g}\ \big((\text{d})_{\text{det}}\ \text{m}\big)_{\text{su}}\big)_{\text{whd body}} \vdash \diamond^{\text{wh body}}(\diamond^{\text{pc}}\text{BW} \multimap \text{SV1})}\,\diamond^{\text{whd body}}I}{\text{Waarover}\ \big(\text{gaat}\ \big((\text{de})_{\text{det}}\ \text{machtstrijd}\big)_{\text{su}}\big)_{\text{whd body}} \vdash \text{WHQ}}\multimap E$$

(b) Structurally decorated linear logic corresponding to the above dependency graph, showcasing the effect of the $\diamond I$ and $\Box E$ rules on recovering the original dependency structure in the form of annotated bracketings. Intermediate steps depict words using their initial letters for space economy. Lexical type assignments are denoted with $\mathcal{L}$ and the identity axiom with $\mathcal{A}$.

Figure 1: Typed DAG and structurally annotated proof for Lassy sample `WS-U-E-A-0000000236.p.11.s.1.xml`, depicting an analysis for the phrase *Waarover gaat de machtstrijd* ("What is the power struggle about?").

type assigned to each of the conjuncts[5]. We then select copied nodes (that is, nodes with multiple incoming edges, the labels of which coincide), which are lying under the inspected conjunction node and not below any other common ancestor and set X to:

$$!\diamond^{d_1} \text{C}_1 \multimap \ldots \multimap ! \diamond^{d_n} \text{C}_n \multimap \text{X}_0,$$

where $\diamond^{d_1}\text{C}_1, \ldots, \diamond^{d_n}\text{C}_n$ is the obliqueness-sorted list of dependency-decorated types of the copied nodes, combined with the structural ! to allow their unhindered relocation within the conjunct's dependency domain. This scheme provides a uniform treatment of arbitrarily nested elliptical conjunctions and allows their logical derivation by means of higher-order types and hypothetical reasoning, without appealing to copying.

[5]We ensure a singular set of conjunct types by standardizing them using a majority-biased conjunct type relabeling.

3.4. Axiom Linking

The algorithm specified above assigns a type to each DAG node; the multiset of types given to terminal nodes should admit the derivation of the root's type (i.e. the type of the sample phrase as a whole should be derivable by the types of its constituent words); it does not, however, specify the derivation itself. To that end we design an additional algorithmic component, which accepts a type-annotated DAG and produces the linear logic proof it prescribes.

The first choice to make is of how to encode proofs; standard choices would be Gentzen style proofs (either in natural deduction or sequent format) or proofnets (Girard, 1987). The proofnet presentation is more appealing, as it combines the pleasant property of natural deduction (one-to-one correspondence with the program terms for meaning composition) with the good computational properties of sequent calculus (decidable proof search). The type system's structural rules are not explicitly specified in the proof rep-

resentations.
Providing the full theory behind proofnets goes beyond the scope of this paper; what follows is a simplified summary. We begin by assigning types a *polarity*. In the context of a logical judgement, types appearing in antecedent position (i.e. assumptions left of the turnstile) are negative, whereas types appearing in succedent position (i.e. conclusions right of the turnstile) are positive. Polarities are then propagated to subformulas as follows. If a type is atomic, its polarity remains unaltered. If it is an implication $T_1 \multimap T_2$, then the polariy of T_2 persists, whereas the polarity of T_1 is reversed.
Atoms nested within complex types are assigned a polarity by recursive application of the above scheme. At its essence, a proofnet is a bijection between positive and negative atoms, i.e. a pairing of each positive atom with an (otherwise equal) negative one[6].
Finding a proof is then equated with constructing the appropriate bijection. To ensure that such a bijection is indeed possible, we first perform a rudimentary correctness check, asserting the branch-wise invariance count of atoms and implications (Van Benthem, 1991). We then project the types of the DAG's terminal nodes into a flattened sequence, sort them based on the corresponding word order, and decorate each atom with an integer, thus distinguishing between unique occurrences of the same atom. Our goal then lies in propagating these indices upwards along the DAG, linking atom pairs as we go. The algorithmic procedure is outlined below.
We first instantiate an empty proofnet in the form of a bijective function from indices to indices. We additionally implement a function, which takes a proofnet and two equal types of inverse polarity and recursively matches their corresponding atoms, updating the proofnet in the process. For simple branchings, we isolate the arguments of the phrasal functor and identify them with the types of its sibling nodes on the basis of the diamond or box decoration of the former and the dependency labels of the latter. The resulting pairs are matched and the proofnet is expanded. The type of the node dominating the branch is then indexed with the functor's result index(es). A bottom-up iteration then gradually indexes the DAG's non-terminal nodes while filling in the proofnet.
For constructions involving hypothetical reasoning, such as elliptic conjunctions and non-local phenomena, the process is a bit more intricate. The higher-order types involved in these do double duty, both providing the functor that composes the outer phrase and supplying the material missing from the inner phrase(s). To resolve such constructions, we simplify the higher-order types by subtracting their embedded arguments and traversing the DAG to find the branch that misses them. Once there, we detach edges that are secondary or point to copied nodes, and replace them by edges (of the same label) that point to placeholder nodes carrying the aforementioned subtracted arguments. This transformation essentially converts the typed DAG back into a typed tree, reducing the problem once more to the simple case.

[6]This bijection must satisfy certain correctness criteria, i.e. not all such bijections constitute valid proofnets.

3.5. Verification

The extraction procedure described in the previous section produces type assignments together with axiom links, i.e. a bijection matching each atomic subformula occurrence to another of opposite polarity. Albeit being legitimate bijections, these axiom links are not necessarily proof nets; a set of correctness criteria needs to be satisfied for a bijection to be translatable into a proof net (a proof net then typically corresponds to many proofs, which differ only in inessential rule permutations). To validate the correctness of the algorithm, we use use LinearOne[7], a linear logic theorem prover to verify that all extracted structures are indeed linear logic proofs. In addition to asserting correctness and providing a sanity check on the extraction algorithm, LinearOne also produces a linear logic proof in sequent calculus and natural deduction presentations as well as the corresponding λ-term for each input proof net candidate.

3.6. Analysis

The end-to-end pipeline yields a number of samples, each corresponding to a mechanically verified typelogical derivation for a sentential or phrasal syntactic analysis. Even though everything on the output end is proven correct (i.e. all extracted proof nets satisfy the correctness criteria), the extraction algorithm fails to produce a derivation in a limited number of instances. These failures are recognized during runtime and their cause is pinpointed; we provide a breakdown of the algorithm's coverage at each step of the process.

Preprocessing Lassy-Small contains 65 200 analyses. We drop 156 of these due to being a single punctuation mark. Out of the remaining 65 044, we obtain 75 060 independent DAGs (1.15 DAGs per Lassy sample on average).

Typing The type assignment algorithm produces a correct output for 72 198 samples (96.2% input coverage). The majority of failed cases relates to conjunction schemata; in particular, 1 492 samples contain conjunction branches that lack a coordinating word and 443 samples contain asymmetric conjunctions. Both cases could be trivially solved, for instance by promoting the first conjunct to the branch head or expanding the polymorphic coordinator scheme to account for unequal constituents; the ad-hoc nature of such solutions would however degrade the quality and consistency of the lexical types, and we therefore abstain from implementing them. The remaining 494 cases (0.6% of the input) are generic failures arising from copying outside the context of a conjunction, annotation errors and discrepancies, and preprocessing artifacts.

Linking and Verifying Out of the 72 198 typed DAGs, the axiom linking algorithm fails to produce a sane bijection in 6 (of which 4 are edge cases and 1 is an annotation error). All 72 192 outputs of the axiom linking algorithm are validated by the theorem prover, leading to an end-to-end coverage of 96.2%.

[7]Available at `https://github.com/RichardMoot/LinearOne`.

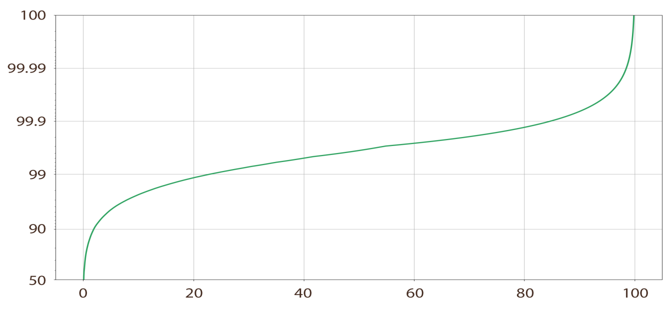


Figure 2: Fraction of words covered as a function of types included. Y axis depicts the percentage of words that can be analyzed (*logit*-transformed). X axis illustrates the percentage of unique types considered. A point (x, y) then represents the $\%$ of words y in the corpus that could be type-assigned if all but the $x\%$ most common types were discarded.

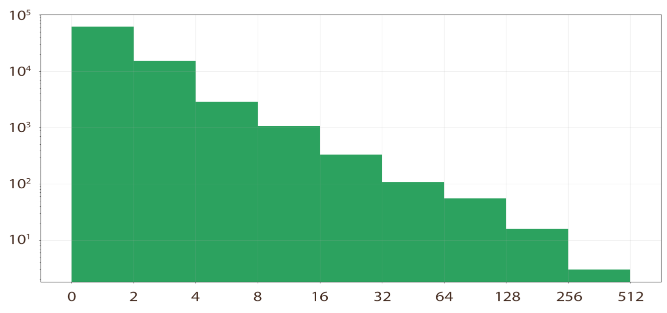


Figure 3: Lexical ambiguity histogram. Y axis depicts the number of words (*log*-scaled), X axis depicts the number of types (*log*$_2$-scaled). A bar spanning the horizontal range (x_1, x_2) with height y indicates that a total of y words are associated with x_1 to x_2 number of types.

4. Resources

4.1. Code

We make the Python code implementing the extraction algorithm publicly available[8]. It is parametric and tunable with respect to the part of speech and phrasal category translation tables (both in terms of domain and codomain, allowing either a refinement or a consolidation of the atomic type set considered), but also the dependency labels (similarly allowing an adjustment of the number of diamond operators). The algorithm is, to a certain degree, agnostic about the underlying type system specification; in other words, it is easy to adapt to different grammars, and can be fine-tuned according to the user's needs and purposes. Further, the algorithm is immediately applicable to Lassy-Large, which boasts a total of 700 million words, significantly outnumbering Lassy-Small. Its massive size, together with the extraction algorithm, grants easy access to a large amount of cross-domain silver standard type-theoretic analyses, as well as a potential corpus for unsupervised language modeling enhanced with lexicalized structural biases. Last but not least, the extraction algorithm is compatible with the Alpino parser's (Bouma et al., 2001) output format; combining the two would then essentially account to an "off the shelf" typelogical theorem prover for written Dutch that produces computational terms along with the standard parses.

4.2. Lexicon

Dissociating leaf nodes from their DAG, we obtain a weighted lexicon mapping words to type occurrences. Our lexicon enumerates a total of 77 283 distinct words and 5 747 unique types made out of 31 atomic types and 26 dependency decorations. On average, each word is associated with 1.79 types and each type with 24 words. Figure 3 presents a histogram of lexical type ambiguity; most words are unambiguous, being always assigned a single type, whereas 20 words (mostly coordinators and auxiliary verbs) are highly ambiguous, being associated with more than 128 unique types each. Figure 2 presents the relation between lexical coverage and types considered; evidently, the 1 150 most common types (20%) suffice to cover 99% of the corpus (on a per-word basis). The lexicon can be utilized as a stand-alone resource, useful for studying grammatical relations and syntactic variation at the lexical level.

4.3. Theorems

The core resource is a collection of 72 192 typelogical derivations[9].
The primitive component behind each derivation is a type-annotated sentence or phrase. On average, samples consist of 12 lexical items (a 25% drop compared to the unprocessed source corpus due to multi-word merges and detached branches); figure 4 presents a histogram of sample lengths. As already hinted by Figure 2, the fine-grained nature of the type system has the side-effect of enlarging the lexicon's co-domain, and therefore the type sparsity, in comparison to other lexicalized grammar formalisms. Figure 5 presents the relation between corpus coverage and types considered; no less than the 5 000 most common types (85%) are required to parse 99% of the corpus. This suggests that, in themselves, the annotated sentences constitute a challenging supertagging task as well as a potential benchmark for open-world classification (Kogkalidis et al., 2019).

[8]The code can be found at `https://github.com/konstantinosKokos/Lassy-TLG-Extraction`.

[9]The public subset of the dataset consists of 7 924 sentences and is available online `https://github.com/konstantinosKokos/aethel-public`

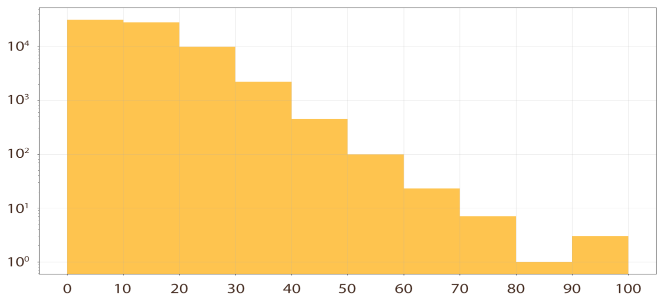


Figure 4: Sentence length histogram. Y axis depicts the number of sentences (*log*-transformed), X axis depicts the number of words. A bar spanning the horizontal range (x_1, x_2) with height y indicates that a total of y sentences have a length of x_1 to x_2 words (with punctuations removed and multi-word phrases counted as a single lexical item).

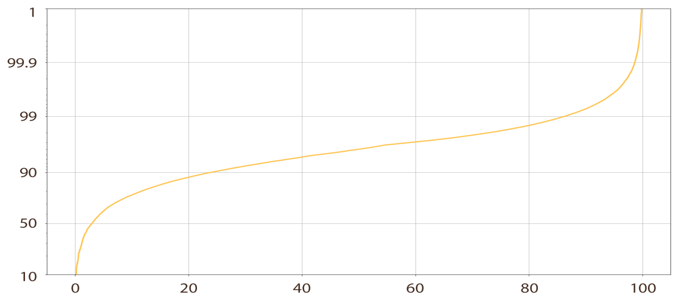


Figure 5: Fraction of sentences covered as a function of types included. Y axis depicts the percentage of sentences that can be analyzed (*logit*-transformed). X axis depicts the percentage of unique types considered. A point (x, y) then represents the $\%$ of sentences y in the corpus, all words of which could be type assigned if all but the $x\%$ most common types were discarded.

Of higher importance are, however, the derivations themselves. As stated earlier, they are provided in four distinct formats; as natural deduction and sequent calculus proofs, bijections between atomic formulas (proofnets) as well as λ-terms. The type system, being based on ILL, is agnostic about word order and thus inherently ambiguous. Out of the (possibly many) potential proofs, the dataset specifies the one that is linguistically acceptable, determining the correct flow of information within the sentence's constituents (an example of such a derivation is presented in Figure 1b). This has multiple ramifications and usecases.

First and foremost, it opens a path towards integrated neuro-symbolic approaches to parsing. On one hand, lexical interactions are constrained to just those that are respectful of the typing information. On the other hand, neural approaches can be applied to narrow down the resulting search space, simultaneously utilizing semantic and syntactic information, with the dependency enrichment providing an additional heuristic. In this context, selecting a parsing methodology and the appropriate proof format is up to the end-user; for instance, shift-reduce parsing would be easier accomplished on the binary branching natural deduction structures (Shieber et al., 1995; Ambati et al., 2016), λ-terms would be more accommodating for generalized translation architectures like sequence transducers (Zettlemoyer and Collins, 2012), proofnet bijections could be obtained via neural permutation learning (Mena et al., 2018) etc; supertagging and type representations can also be jointly optimized (Lewis et al., 2016; Kasai et al., 2017).

The dataset can also emerge as a useful resource for "pure" parsing as deduction. The type grammar and its derivations can be utilized as a stepping stone towards stricter type systems. Noting that our abstract types are in alignment with Lambek types (modulo directionality), linear implications can be gradually replaced with directed divisions based on their aggregated corpus-wide behavior, easing the transition towards an (either hybrid or multi-modal) typelogical dataset.

5. Conclusion

We have described a linear type system that captures abstract syntactic function-argument relations, but is also able to distinguish between arguments on the basis of their dependency roles. We have presented a methodology for extracting these linear types as well as their interactions out of dependency parsed treebanks. Our approach is modular, allowing a large degree of parameterization, and general enough to accommodate alternative type systems and source corpora. We implemented and applied a concrete algorithmic instantiation, which we ran on the Lassy treebank, generating a large dataset of type-theoretic syntactic derivations for written Dutch. Utilizing a theorem prover, we verified the correctness of the algorithm's output, and transformed it into a number of different realizations to facilitate its use in different contexts. Taking advantage of the Curry-Howard correspondence between linear logic and the simply-typed λ-calculus, we naturally equated our derivations to computational terms which characterize the flow of information within a sentence. Disconnecting types from their derivations, we are left with a pairing of sentences to type sequences; disconnecting them from their surrounding context, we obtain a highly refined lexicon mapping words to type occurrences. We make a significant portion of the above resources publicly available. Our hope is that our resources will find meaningful applications at the intersection of formal and data-driven methods, in turn giving rise to practically applicable insights on the syntax-semantics interface.

6. Acknowledgements

Konstantinos and Michael are supported by the Dutch Research Council (NWO) under the scope of the project "A composition calculus for vector-based semantic modelling with a localization for Dutch" (360-89-070).

7. Bibliographical References

Ambati, B. R., Deoskar, T., and Steedman, M. (2016). Shift-reduce CCG parsing using neural network models. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 447–453.

Barendregt, H., Dekkers, W., and Statman, R. (2013). *Lambda Calculus with Types*. Perspectives in Logic. Cambridge University Press.

Bouma, G., Van Noord, G., and Malouf, R. (2001). Alpino: Wide-coverage computational analysis of Dutch. *Language and Computers*, 37:45–59.

Curry, H. B. (1961). Some logical aspects of grammatical structure. *Structure of language and its mathematical aspects*, 12:56–68.

De Groote, P. (2001). Towards abstract categorial grammars. In *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics*, pages 252–259.

Dowty, D. (1982). Grammatical relations and Montague grammar. In P. Jacobson et al., editors, *The nature of syntactic representation*, pages 79–130. Reidel.

Girard, J.-Y. (1987). Linear logic. *Theoretical computer science*, 50(1):1–101.

Heylen, D. (1999). Underspecification in type-logical grammars. In Alain Lecomte, et al., editors, *Logical Aspects of Computational Linguistics*, pages 180–199, Berlin, Heidelberg. Springer Berlin Heidelberg.

Hockenmaier, J. and Steedman, M. (2007). CCGbank: A corpus of CCG derivations and dependency structures extracted from the Penn treebank. *Computational Linguistics*, 33(3):355–396.

Johnson, M. (1999). A resource sensitive interpretation of Lexical Functional Grammar. *Journal of Logic, Language and Information*, 8(1):45–81.

Kasai, J., Frank, R., McCoy, R. T., Rambow, O., and Nasr, A. (2017). TAG parsing with neural networks and vector representations of supertags.

Kogkalidis, K., Moortgat, M., and Deoskar, T. (2019). Constructive type-logical supertagging with self-attention networks. In *Proceedings of the 4th Workshop on Representation Learning for NLP (RepL4NLP-2019)*.

Kubota, Y. and Levine, R. (to appear). *Type-Logical Syntax*. MIT Press.

Lambek, J. (1958). The mathematics of sentence structure. *The American Mathematical Monthly*, 65(3):154–170.

Lewis, M., Lee, K., and Zettlemoyer, L. (2016). LSTM CCG parsing. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 221–231.

Mena, G., Belanger, D., Linderman, S., and Snoek, J. (2018). Learning latent permutations with gumbel-sinkhorn networks. *arXiv preprint arXiv:1802.08665*.

Moortgat, M. and Moot, R. (2002). Using the Spoken Dutch Corpus for type-logical grammar induction. In *Proceedings of the Third International Conference on Language Resources and Evaluation, LREC 2002, Las Palmas*. European Language Resources Association.

Moortgat, M. and Morrill, G. (1991). Heads and phrases. type calculus for dependency and constituent structure. Ms, Utrecht University.

Moortgat, M. (1996). Multimodal linguistic inference. *Journal of Logic, Language and Information*, 5(3/4):349–385.

Moot, R. (2015). A type-logical treebank for French. *Journal of Language Modelling*, 3(1):229–264.

Moot, R. (2017). The Grail theorem prover: Type theory for syntax and semantics. In Stergios Chatzikyriakidis et al., editors, *Modern Perspectives in Type Theoretical Semantics*, pages 247–277. Springer.

Morrill, G., Valentín, O., and Fadda, M. (2011). The displacement calculus. *Journal of Logic, Language and Information*, 20(1):1–48.

Morrill, G. (1994). *Type logical grammar - categorial logic of signs*. Kluwer.

Muskens, R. (2001). Lambda grammars and the syntax-semantics interface. In Robert Van Rooij et al., editors, *Proceedings of the Thirteenth Amsterdam Colloquium*, pages 150–155. Amsterdam: ILLC.

Shieber, S. M., Schabes, Y., and Pereira, F. C. (1995). Principles and implementation of deductive parsing. *The Journal of logic programming*, 24(1-2):3–36.

Sørensen, M. H. and Urzyczyn, P. (2006). *Lectures on the Curry-Howard Isomorphism*, volume 149. Elsevier.

Van Benthem, J. (1991). Language in action. *Journal of philosophical logic*, 20(3):225–263.

van Noord, G., Bouma, G., Van Eynde, F., de Kok, D., van der Linde, J., Schuurman, I., Sang, E. T. K., and Vandeghinste, V., (2013). *Large Scale Syntactic Annotation of Written Dutch: Lassy*, pages 147–164. Springer Berlin Heidelberg, Berlin, Heidelberg.

Zettlemoyer, L. S. and Collins, M. (2012). Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. *arXiv preprint arXiv:1207.1420*.

Appendix

1. Translation Tables

Atomic Types Table 1 presents the set of atomic types and their origins (part-of-speech and phrasal category tags). The current translation domain utilizes the Lassy part of speech tagset (*pt*); other options could be either the alpino tagset (*pos*) or even the detailed tagset (*postag*). Note also that in our usecase there is a one-to-one correspondence between tags and types. This does not necessarily need to be the case; one could as well choose to collapse one or more tags onto the same type (e.g. translate *vnw* to NP or all sentential tags to S). The extraction algorithm is parametric to all above possible variations.

Tag	Description	Assigned Type
	Short POS Tags	
adj	Adjective	ADJ
bw	Adverb	BW
let	Punctuation	LET
lid	Article	LID
n	Noun	N
spec	Special Token	SPEC
tsw	Interjection	TSW
tw	Numeral	TW
vg	Conjunction	VG
vnw	Pronoun	VNW
vz	Preposition	VZ
ww	Verb	WW
	Phrasal Category Tags	
advp	Adverbial Phrase	ADV
ahi	Aan-Het Infinitive	AHI
ap	Adjectival Phrase	AP
cp	Complementizer Phrase	CP
detp	Determiner Phrase	DETP
inf	Bare Infinitival Phrase	INF
np	Noun Phrase	NP
oti	Om-Te Infinitive	OTI
pp	Prepositional Phrase	PP
ppart	Past Participial Phrase	PPART
ppres	Present Participial Phrase	PPRES
rel	Relative Clause	REL
smain	SVO Clause	SMAIN
ssub	SOV Clause	SSUB
sv1	VSO Clause	SV1
svan	Van Clause	SVAN
ti	Te Infinitive	TI
whq	Main WH-Q	WHQ
whrel	Free Relative	WHREL
whsub	Subordinate WH-Q	WHSUB

Table 1: Part-of-speech tags and phrasal categories, and their corresponding type translations.

Dependency Decorations Table 2 presents the set of dependency decorations (i.e. modal operators) and their descriptions. Most decorations coincide with a Lassy dependency label or a derivative thereof.

Decoration	Description	Precedence
app	Apposition	20
cmp_body	Complementizer body	18
cnj	Conjunct	0
det	Noun-phrase determiner	1
hdf	Final part of circumposition	7
ld	Locative Complement	6
me	Measure Complement	5
mod	Modifier	21
obcomp	Comparison Complement	3
obj1	Direct Object	12
obj2	Secondary Object	10
pc	Prepositional Complement	8
pobj	Preliminary Object	13
predc	Predicative Complement	11
predm	Predicative Modifier	19
rhd-body	Relative clause body	17
se	Obligatory Reflexive Object	9
su	Subject	14
sup	Preliminary Subject	15
svp	Separable Verbal Participle	2
vc	Verbal Complement	4
whd-body	WH-question body	16

Table 2: Dependency relations and their corresponding implication labels.

2. Obliqueness Hierarchy

Phrasal heads are assigned functor types; in the multi-argument case, these would be of the form:

$$(\mathrm{A}_1 \otimes \mathrm{A}_2 \otimes \ldots \mathrm{A}_n) \multimap \mathrm{R},$$

or their curried equivalent:

$$\mathrm{A}_1 \multimap \mathrm{A}_2 \multimap \ldots \mathrm{A}_n \multimap \mathrm{R}$$

To avoid the inconvenience of (in this case, superficial) distinction between different argument type permutations, we impose a strict full order on argument sequences, loosely based on the obliqueness hierarchy of their syntactic roles (apparent through their modal decorations). The ordering is presented in the third column of Table 2; the lower a label's number, the less prominent (i.e. more oblique) the argument it marks, causing it to be consumed first. This scheme, recursively applied, provides a unique implicational type for each functor's argument-permutation class. Functors carrying *cnj*-decorated arguments (i.e. coordinators and their derivatives) are the only kind of functor which permits two distinct argument of the same decoration; we sort those based on the conjuncts' order within the sentence.