SPINDLE:
Spinning Raw Text into Lambda Terms with Graph Attention

Konstantinos Kogkalidis[◇] and **Michael Moortgat**[◇] and **Richard Moot**[□]

◇ Institute for Language Sciences, Utrecht University

□ LIRMM, Université de Montpellier, CNRS

{k.kogkalidis,m.j.moortgat}@uu.nl, richard.moot@lirmm.fr

Abstract

This paper describes SPINDLE[1] – an open source Python module implementing an efficient and accurate parser for written Dutch that transforms raw text input to programs for meaning composition, expressed as λ terms. The parser integrates a number of breakthrough advances made in recent years. Its output consists of hi-res derivations of a multimodal type-logical grammar, capturing two orthogonal axes of syntax, namely deep function-argument structures and dependency relations. These are produced by three interdependent systems: a static type-checker asserting the well-formedness of grammatical analyses, a state-of-the-art, structurally-aware supertagger based on heterogeneous graph convolutions, and a massively parallel proof search component based on Sinkhorn iterations. Packed in the software are also handy utilities and extras for proof visualization and inference, intended to facilitate end-user utilization.

1 Introduction

The transparency and formal well-behavedness of lambda calculi make them the ideal format for expressing compositional structures, a fact that has been duly emphasized by parsers and tools with a predominant focus on semantics. Lambda calculi form a key ingredient of type-logical grammars, where they find use as the computational counterpart of a so-called *grammar logic*, a substructural logic of the intuitionistic linear variety that is designed to capture (aspects of) natural language syntax and semantics (Moortgat, 1997). For type-logical grammars, the Curry-Howard isomorphism guarantees a straightforward passage between logical rules, type constructors and term-forming operators; put simply, Parse ≡ Proof ≡ Program, and Category ≡ Proposition ≡ Type. The modus operandi

is straightforward: a lexicon associates words with logical formulas, and the logic's rules of inference decide how formulas may interact with one another.

By extension, words may only combine in a strict, well-typed manner, forming larger phrases in the process. Parsing becomes a process of logical deduction, at the end of which the result (a proof) gives rise to a recipe for meaning assembly (a program). This program is turned into executable code as soon as one plugs in appropriate interpretations for the lexical constants (words) and for the term operations (composition instructions). The set-up is general-purpose in that it readily accommodates different choices for these interpretations; valid targets can for instance be found in (truth-conditional) formal semantics, distributional-compositional models (Sadrzadeh and Muskens, 2018), or tableau-based theorem provers (Abzianidze, 2017).

In this work, we are interested in what happens prior to semantic execution; that is, we abstract away from lexical semantics and seek to reveal the compositional recipe underlying a natural language utterance. To that end, we employ a type grammar aimed at capturing two different *syntactic* axes, only rarely observed together in the wild: function-argument structures and dependency relations. To procure a derivation from an input phrase, we design and implement a system combining three distinct but communicating components. Component number one is the implementation of the grammar's type system — it comes packed with a number of useful facilities, most important being a static type checker that verifies the syntactic well-formedness of the analyses construed. Component number two is a supertagger responsible for assigning a type to each input word — the tagger is formulated on the basis of a hyper-efficient heterogeneous graph convolution kernel that boasts state-of-the-art accuracy among categorial grammar datasets. The third and last component is a neural permutation module that

---

[1]Stylized spind²λe and standing for **s**pindle **p**arses **in**to **d**ependency-**d**ecorated λ **e**xpressions. Source code and user instructions can be found at https://github.com/konstantinosKokos/spindle.

exploits the linearity constraint of the target logic to simplify proof search as optimal transport learning (Peyré et al., 2019) — this reformulation allows for a massively parallel and easily optimizable implementation. The three components alternate roles through the processing pipeline, switching between phases of low level linear algebra routines and high level logical reasoning (GPU and CPU intensive, respectively). Their integration yields a lightspeed-fast and highly accurate neurosymbolic parser, neatly packaged and made publicly available.

2 System Decomposition

2.1 Type Grammar

The system's theoretical backbone is its type logic – a uniquely flavoured, semantics-first type-logical grammar that strays from the categorial norm in two major ways. First, it focuses on deep syntactic structure (or tectogrammar, in Curry's terms) rather than surface form; its functional types are therefore oblivious to directional or positional constraints, abiding only to the linearity condition: every occurrence of an atomic proposition must be used once and exactly once. Second, it dresses functional types up, so as to have them encode grammatical functions, making a three-way distinction between complements, heads and adjuncts.

A full exposition of the grammar is beyond the scope of this paper, but a superficial and simplified rundown should help shed light on what is to follow. Its first aspect, **function-argument structures**, is modeled using linear logic's implication arrow, $\multimap$, which gives us access to resource-conscious versions of function application and variable abstraction (Girard, 1987; Abramsky, 1993). In their linguistic usecase, functional types of the form $A\multimap B$ denote predicates that *consume* a single occurrence of some object of type A, the result being a composite phrase of type B. Reasoning about gaps, ellipses and the like is accomplished with the aid of higher-order types, i.e. instances of the previous scheme where A is itself a function — these higher-order types launch a process of **hypothetical reasoning**, whereby we may temporarily assume the existence of a resource to produce a derivation locally, only to later withdraw the hypothesis, creating a new function in the process. The second aspect, **dependency relations**, are modeled using a labeled assortment of residuated pairs of unary operators lent from temporal logic. Atomic types without any

$$\frac{}{x : A \vdash x : A}\ \text{id} \qquad \frac{(c \mapsto A) \in \mathcal{L}}{c : A \vdash c : A}\ \text{lex}$$

$$\frac{\Gamma \vdash s : A\multimap B \quad \Delta \vdash t : A}{\Gamma, \Delta \vdash s\,t : B}\ \multimap E \qquad \frac{\Gamma, x : A \vdash s : B}{\Gamma \vdash \lambda x.s : A\multimap B}\ \multimap I$$

$$\frac{\Gamma \vdash s : \Box_\delta A}{\langle \Gamma \rangle^\delta \vdash \blacktriangledown_\delta s : A}\ \Box_\delta E \qquad \frac{\Gamma \vdash s : A}{\langle \Gamma \rangle^\delta \vdash \vartriangle_\delta s : \Diamond_\delta A}\ \Diamond_\delta I$$

Figure 1: Logical rules of inference used by the type grammar (subset). The id rule instantiates a fresh variable of arbitrary type A. The lex rule provides declares a constant c as being of type A, given type assignment $c \mapsto A$ pulled from the lexicon $\mathcal{L}$ (or, in the post-neural era, the supertagger). Introduction rules are complex types constructors, elimination rules are destructors. The $\multimap E$ rule says a term s of type $A\multimap B$ derived from premises Γ can apply to a term t of type A derived from premises Δ, producing a complex term $s\,t$ of type B derived from the merger of Γ and Δ. The $\multimap I$ rule says that if the premises of some term s of type B include a variable x of type A, we can abstract over the latter, producing a term $\lambda x.s$ of type $A\multimap B$. The $\Box_\delta E$ rule removes the box from a term s of type $\Box_\delta A$, producing term $\blacktriangledown_\delta s$ of type A and enclosing the premises under brackets $\langle \_ \rangle^\delta$. Dually, the $\Diamond_\delta I$ rule puts a term s of type A under the scope of a diamond, producing term $\vartriangle_\delta s$ of type $\Diamond_\delta A$ and again enclosing the premises under brackets $\langle \_ \rangle^\delta$.

dependency decorations are assigned to linguistically autonomous units and phrases, e.g. NP for a noun phrase. Functional types denoting heads impose a diamond $\Diamond_c$ on the complements they select for, label c being the dependency slot the complement is to occupy, e.g. $\Diamond_{su}NP\multimap S_{main}$ for an intransitive verb looking for a subject-marked noun phrase to produce a matrix clause. Dually, functional types denoting adjuncts are themselves decorated with a box $\Box_a$, label a now being the dependency role projected by the adjunct prior to application, e.g. $\Box_{mod}(NP\multimap NP)$ for an adjective, promising to provide a function over noun phrases if one is to remove its box. Introducing a diamond or eliminating a box leaves a *structural* imprint that encloses complete phrases under brackets, and a *computational* imprint that calls for a special treatment of the wrapped term – both labeled by the grammatical function of the diamond (resp. box) that was introduced (resp. eliminated). The key logical rules of the type grammar and their isomorphic term operations are presented in Figure 1.

$$
\cfrac{
  \cfrac{
    \cfrac{\text{is}}{c_1 \vdash \Diamond_{predc}\text{PRON} \multimap \Diamond_{su}\text{NP} \multimap \text{SVI}}\ \text{lex}
    \quad
    \cfrac{}{x \vdash \Diamond_{predc}\text{PRON}}\ \text{id}
    \Big/ \multimap E
  }{c_1, x \vdash \Diamond_{su}\text{NP} \multimap \text{SVI}}
  \quad
  \cfrac{
    \cfrac{
      \cfrac{\text{die}}{c_2 \vdash \Box_{det}(N \multimap NP)}\ \text{lex}
      \ \Big/\ \Box_{det}E \ \to\ \langle c_2 \rangle^{det} \vdash N \multimap NP
      \quad
      \cfrac{
        \cfrac{\text{rare}}{c_3 \vdash \Box_{mod}(N \multimap N)}\ \text{lex}
        \Big/ \Box_{mod}E \to \langle c_3 \rangle^{mod} \vdash N \multimap N
        \quad \cfrac{\text{tekening}}{c_4 \vdash N}\ \text{lex}
      }{\langle c_3 \rangle^{mod}, c_4 \vdash N}\ \multimap E
    }{\langle c_2 \rangle^{det}, \langle c_3 \rangle^{mod}, c_4 \vdash NP}\ \multimap E
    \Big/ \Diamond_{su}I
    \to \langle\langle c_2 \rangle^{det}, \langle c_3 \rangle^{mod}, c_4 \rangle^{su} \vdash \Diamond_{su}NP
  }{\cdots}\ \multimap E
}{
\begin{array}{l}
c_1, x, \langle\langle c_2 \rangle^{det}, \langle c_3 \rangle^{mod}, c_4 \rangle^{su} \vdash \text{SVI} \\
c_1, \langle\langle c_2 \rangle^{det}, \langle c_3 \rangle^{mod}, c_4 \rangle^{su} \vdash \Diamond_{predc}\text{PRON} \multimap \text{SVI} \quad \multimap I\\
\langle c_1, \langle\langle c_2 \rangle^{det}, \langle c_3 \rangle^{mod}, c_4 \rangle^{su} \rangle^{whbody} \vdash \Diamond_{whbody}(\Diamond_{predc}\text{PRON} \multimap \text{SVI}) \quad \Diamond_{whbody}I
\end{array}
}
$$

$$
\cfrac{\text{Wat}}{c_0 \vdash \Diamond_{whbody}(\Diamond_{predc}\text{PRON} \multimap \text{SVI}) \multimap \text{WHQ}}\ \text{lex}
$$

$$
c_0, \langle c_1, \langle\langle c_2 \rangle^{det}, \langle c_3 \rangle^{mod}, c_4 \rangle^{su} \rangle^{whbody} \vdash c_0\, \triangle_{whbody}(\lambda x.c_1\ x\ \triangle_{su}(\blacktriangledown_{det}(c_2)\,(\blacktriangledown_{mod}(c_3)\ c_4))) : \text{WHQ} \qquad \multimap E
$$

Dependency arcs: whbody, su, det, mod over the words: Wat — is — die — rare — tekening

Figure 2: Natural deduction proof for the sentence *Wat is die rare tekening?* 'What is that strange drawing?'. For space economy, compositional λ term is only explicitly written in the endsequent (bottom of the proof). From the antecedent structure of the endsequent, we may also recover a dependency tree. Color coding serves to informally differentiate between complement (red) vs. adjunct (green) structural brackets/dependency arcs.

2.1.1 Proof Representation

Proofs in the type logic are traditionally served in the tree-like **natural deduction** format. Proofs in natural deduction benefit from an easy translation to (i) λ expressions, by following the rules of Figure 1, and (ii) dependency trees, by simply casting structural brackets to dependency arcs, going from the head of each phrase to (the heads of) its dependents. Figure 2 presents a visual example. An alternative representation is in the far less verbose format of a **proof net**, a geometric construction that abstracts away from the bureaucratic book-keeping of hypothetical reasoning and tree-structured rule ordering. Figure 3 presents the proof net equivalent of the running example. Proof nets are easier to reason about in a neural setup by allowing us to treat parsing as the vastly simplified problem of matching each occurrence of an atomic proposition in *negative* position, i.e. a prerequisite of a conditional implication, with an occurrence in *positive* position, i.e. a (conditionally) proven statement. The parallel nature of proof nets allows the matching to occur simultaneously across the entire proof; that is, all decisions are done in a single instant, without the bottleneck of having to wait for conditionals to be satisfied in a bottom-up fashion. On the other hand, proof nets are slightly under-specified compared to natural deduction proofs, being explicit only with respect to function-argument structures – translating from one format to another requires establishing some conventions on what constitutes a canonical proof.

2.1.2 Implementation

The syntax of the type system is implemented as a tiny DSL written in Python.[2] It is used as the representation format of Æthel (Kogkalidis et al., 2020a), a dataset of some 70 000 analyses of written Dutch, which also constitutes the system's training data. The implementation was originally designed to assert the type-safety of the dataset, to facilitate the conversion between natural deduction trees, λ terms and proof nets, and to ease third-party corpus analysis by providing niceties such as search and pretty printing utilities, cross compilation to LaTeX for visualization purposes, interfaces for proof transformations, etc. All these functionalities are imported unchanged. The conversion routines allow us to conduct neural proof search in the favorable regime of proof nets, and convert the result to natural deduction format only at the very end, just for the sake of presentation and/or sanity testing. Importantly, the type-checker is re-purposed as a tool for verifying the correctness of analyses constructed – an analysis that does not amount to a valid proof will fail to pass the checker, throwing a type error and alerting us to the fact. In other words, we can blindly trust anything the parser gives us as correct, at least in the sense of (proof-theoretic) syntactic validity.

---

[2] Source code can be found at https://github.com/konstantinosKokos/aethel.
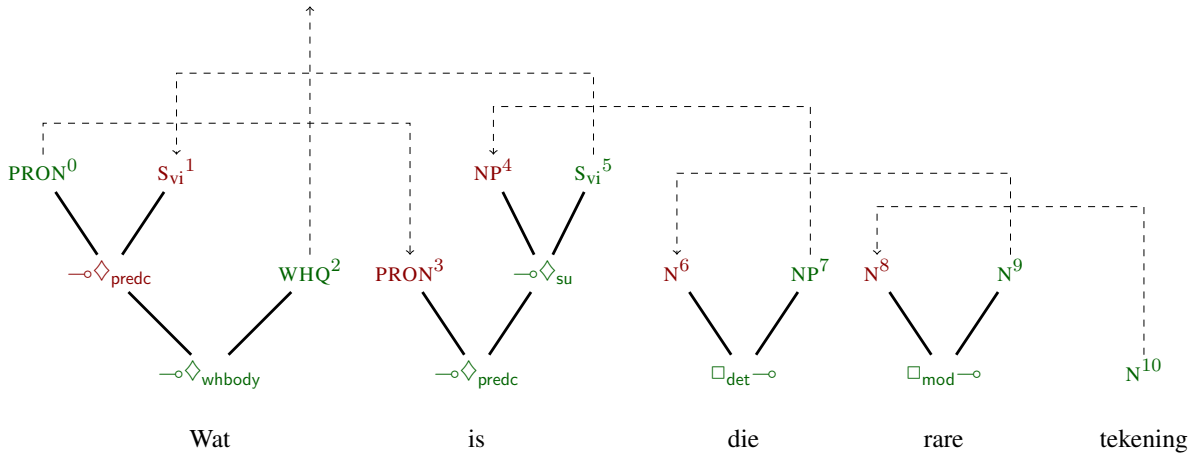
Figure 3: Proof net equivalent of the proof of Figure 2, with unary diamonds (resp. boxes) fused with the implication dominating (resp. dominated by) them for depth compression. Atomic propositions are indexed by enumeration for identification purposes. Color coding here serves to differentiate between resources we have (green) and resources we need (red) – the rule is start green from the bottom, change (resp. keep) color for the left (resp. right) daughter of an implication. Bold edges denote the tree structure underlying type assignments. Dashed edges denote the correct matching between resources of opposite polarity.

2.2 Supertagging Module

Lexical type ambiguity and lexical type sparsity are common and pervasive problems for any categorial grammar. The de facto approach rests on a supertagger, a neural module replacing the fixed lexicon, traditionally formulated as a sequence classifier and trained to produce the most plausible type assignments for each word in the context of an input sentence. Here, these problems are exacerbated by the highly elaborated type system. Some 80% of Æthel's approx. 6 000 types are rare (i.e. have less than 10 occurrences in the corpus), and some 10% of the total sentences contain at least 1 such rare type. This necessitates a more ambitious treatment than the standard "set-and-forget" approach of completely discarding rare type assignments as inconsequential. The solution comes in the form of a **constructive supertagger**, an auto-regressive neural decoder that is trained to construct types on the fly according to their algebraic decomposition, rather than treat them as singular, opaque blocks (Kogkalidis et al., 2019). This configuration enables the construction of valid types regardless of whether they have been seen before or not, extending coverage beyond the training data. The supertagger employed here follows a geometrically informed, task-specific decoding order, whereby types are represented as the structural unfolding of binary trees. Following Prange et al. (2021), trees are decoded in parallel across the entire batch of input sequences, establishing an upper temporal

bound on decoding that scales with the maximal tree depth – in practice, a constant. To circumvent the locality of a standard tree decoder, the target output being not a batch of trees but a batch of **sequences of trees** (see Figure 3), the supertagger is formulated as a a graph neural network, utilizing message-passing connections to transfer feedback from tree nodes to their lexical roots and from lexical roots to their neighbours, ensuring that decisions made at each decoding step are influenced by prior decisions across the entire output (Kogkalidis and Moortgat, 2022). As a result, it strikes the perfect balance between the speed and memory efficiency of a tree-shaped architecture, allowing for more training iterations and faster inference, and the stronger autoregressive properties of a seq2seq model, improving performance. Further, being inherently constrained to trees, its output is structurally correct-by-construction – under no circumstance can any of the types produced be ill-formed. Used in isolation, the architecture currently sits at the top of the accuracy leaderboard for categorial grammar supertagging across different formalisms and languages – the performance is marginally inferior in the multi-task training setup adopted here.

2.3 Permutation Module

Conducting search over proof nets is typically ill-advised. The problem traditionally involves examining all possible bijections between positive and negative atomic propositions. The number of

such bijections scales factorial to the number of atomic propositions, quickly becoming prohibitive. To navigate this combinatorially explosive landscape, **neural proof nets** relax proof search into a continuous, differentiable problem, where finding the correct bijection is translated to a transportation problem (Kogkalidis et al., 2020b) learned by yet another graph neural network. Concretely, the representations of all occurrences of atomic propositions are extracted from the decoder and binned according to their sentential index, sign and polarity (e.g. a single bin would be all occurrences of a positive NP in sentence #13 of the input batch). Each bin is contrasted with its inverse polarity counterpart using some similarity metric (here a weighted inner product). The result is a collection of square matrices, each matrix containing attention weights (or similarity scores) in the cartesian product of positive and negative items of the same sign and sentence. These matrices are grouped by their cardinality, and the Sinkhorn operator (Sinkhorn and Knopp, 1967) is used to push them towards binarity and bistochacity, yielding approximations of permutation matrices representing the goal bijections (Mena et al., 2018).

To make things concrete using the running example of Figure 3, each of the atomic types S_{vi}, NP and PRON has a single negative and a single positive occurrence, therefore their bijections are trivial (a testament to supertagging being almost parsing). The single positive occurrence of WHQ stands for the goal type of the phrase, and remains unmatched. Only N requires a decision, having two possible bijections. The correct candidate is encoded by the permutation table below, where rows enumerate positive and columns negative items:

$$\Pi_N := \begin{array}{c|c|c|} & 6 & 8 \\ \hline 9 & \checkmark & \\ \hline 10 & & \checkmark \\ \hline \end{array}$$

This reformulation entails a tremendous speedup: the painstakingly slow problem of symbolic proof search is cast into simple, well-optimized and batchable matrix operations. The current parser builds on the insight that the permutation module is invariant to the source of atomic symbol representations, and in fact greatly benefits from the faster and more accurate task-adapted supertagger.

2.4 Integration

Neurosymbolic integration yields an end-to-end pipeline that consists of the following phases. First, the user inputs a list of sentences to be parsed. Contextualized token representations are obtained from a fine-tuned $BERT_{BASE}$ model, which are then aggregated according to the input's word boundaries. The resulting word representations act as initial seeds for decoding to begin on an empty canvas. During decoding, types are progressively constructed, while seeds are updated by exchanging messages with one another on the basis of their sequential proximity. After a small number of decoding steps, the process terminates, yielding a sequence of type assignments for each input sentence. A rudimentary invariance check is then performed, controlling whether each sequence counts an equal number of atomic propositions of each polarity. Sentences failing the invariance check are not eligible for proof search, and their analysis stops early. Passing sentences are symbolically processed to obtain a collection of sparse indexing tensors, used to gather the decoder's representations into the bins described earlier. Bins are batched and contrasted, and a small number of Sinkhorn iterations is employed as a 2-dimensional softmax analogue. The soft Sinkhorn distances are discretized using the Hungarian algorithm in order to enforce bijectivity (Jonker and Volgenant, 1987). Bijections are reassociated with their origin symbols and sentences, using the reverse of the previous indexing operation. Control is then passed to the symbolic component, which attempts to traverse the candidate proof nets, verifying the correctness criteria of acyclicity and connectedness in the process (Danos and Regnier, 1989). The traversal coincides with a translation to a natural deduction format, the construction of which corresponds to static type checking of the output (Lamarche, 2008). Assuming no type mismatches are caught, the output is a proof proper, which by Curry-Howard isomorphism is rewritten as a dependency-decorated λ term. The user is finally presented with an analysis for each input sentence – ideally, a λ term, but occasionally a rejected intermediate result together with an error description.

3 Evaluation

3.1 Performance

The system has been evaluated on the test set of Æthel. Without any pre-filtering or post-processing

training wheels (i.e. no constraints on sentence length, type rarity/depth or cardinality of bijections), the parser produces a proof that satisfies strict syntactic equality with the ground truth in 3 191 of the 5 770 test set samples. This amounts to a significant 55.30% of the test sentences being analyzed without a single error with respect to type assignments, phrasal chunking, function-argument structures and dependency annotations produced[3] In total, 4 901 sentences are assigned a passing analysis, which sets the coverage to a more modest 84.94%. The discrepancy between the high accuracy and low coverage is due to the rigidness of the type system: only 5 010 of the sentences satisfy the invariance check, being thus amenable to any proof. This signals that the performance bottleneck lies on the supertagger rather than the permutation module; a parse is assigned to 97.82% of parsable sentences, and it's also the perfect parse 75.30% of the time. These findings are summarized in Table 1.

parsability	coverage
(some proof obtainable)	(some proof obtained)
86.83	84.94
types correct	**accuracy**
(correct proof obtainable)	(correct proof obtained)
56.88	55.30

Table 1: Sentential-level evaluation of the parser.

To obtain a more refined perspective on performance, we employ an adaptation of the parsing community's favorite F_1-score. Concretely, we gather all samples for which a proof was produced, and decompose both prediction and ground truth into their respective sets of subproofs. We measure tp as the two sets' intersection, fp as the difference between predicted and correct subproofs and fn as the difference between correct and predicted subproofs, from which we may obtain precision as $p = {}^{tp}/_{(tp+fp)}$, recall as $r = {}^{tp}/_{(tp+fn)}$ and their harmonic mean as $F_1 = {}^{2pr}/_{(p+r)}$. On top of the vanilla versions of these metrics, we can also examine relaxations by incorporating a combination of two modulo factors. Relaxation one targets the functional core of the logic, applying a forgetful transformation that strips proofs of their modalities in order to examine typed function-argument

structures in isolation. Relaxation two targets the modal enhancement of the logic, collapsing the set of atomic types into a single point (thus treating all functional types of the same *shape* as equal) in order to examine dependency structures in isolation. Relaxing on both axes at once is essentially replacing type constants by type variables, where all we care about are the type- and dependency-agnostic linear function-argument structures – this is the metric most comparable to external theories.[4] Note that relaxations are performed only *after* inference – the point being that a strict proof must have been produced for its relaxations to be considered (i.e. lax accuracy is still bottlenecked by strict coverage). The results are averaged over covered samples[5] and presented in Table 2.

modulo	local metrics		
	p	r	F_1
–	89.52	89.68	89.39
modalities	90.93	91.13	90.85
functional types	91.09	91.26	90.97
both	92.31	92.52	92.24

Table 2: Decomposition metrics and relaxations.

3.2 Efficiency

Regarding efficiency, the architecture contains a non-negligible total of 117M parameters, 94% of which are inherited from the underlying BERT model. The memory footprint of the network's forward pass does not exceed 3.5GB on the test set with a batch size of 64, making it reasonably lightweight for use at home. Using a middle range laptop GPU, the network takes about 15 seconds to tag the full test set (i.e. 370 sents/sec or 6 000 tokens/sec), and 80 seconds to tag and parse it (i.e. 70 sents/sec), including tokenization and post-processing. Note, however, that sentence-level batching is not yet implemented for inference mode proof search, i.e. sentences are tagged in parallel but proven sequentially. Cross-sentential padding and batching of Sinkhorn inputs is in the works – benchmarking shows that the asymptotic behavior of a forward pass over batches of 64 matrices only starts becoming apparent when they exceed

---

[3]This is comparable to the state-of-the-art for the similar (in fact simpler) problem of CCG parsing; see Clark (2021) for an up-to-date overview.

[4]Proofs are in β and η normal, so no free points from abstractions. Variables are only equal if they match in both name and type, so no free points from variable instantiations either.

[5]Averaging over the full test set would artificially inflate p and deflate r scores, since no partial proofs are returned from failing samples.

```python
>>> from inference import InferenceWrapper as IW
>>> from aethel.utils.tex import compile_tex, sample_to_tex
>>> parser = IW(weight_path='./data/model_weights.pt')
>>> analysis = parser.analyze(['Wat is die rare tekening?'])[0]
>>> analysis
Analysis(
    lexical_phrases=(
        LexicalPhrase(string=Wat, type=(◇whbody(◇predc(VNW)→SV1))→WHQ, len=1),
        LexicalPhrase(string=is, type=◇predc(VNW)→◇su(NP)→SV1, len=1),
        LexicalPhrase(string=die, type=□det(N→NP), len=1),
        LexicalPhrase(string=rare, type=□mod(N→N), len=1),
        LexicalPhrase(string=tekening, type=N, len=1),
        LexicalPhrase(string=?, type=PUNCT, len=1)),
    proof=c0, ⟨c1, ⟨⟨c2⟩det, ⟨c3⟩mod, c4⟩su⟩whbody ⊢ c0 △whbody((λx0.c1 x0 △su(▼det(c2) (▼mod(c3) c4)))) : WHQ)
>>> tex_proof = compile_tex(sample_to_tex(analysis))  # see Figure 2 for output :)
```

Figure 4: User interaction example in python console.

the order of 2^7, being locked at an insubstantial 1ms prior to that.[6]

4 User Interface

The user interface is bare-bones, but simple and easy to use. The parser is currently packaged as a code repository, which, once downloaded, can be used as a python module. A front-end class wraps around the scary inner workings of the parser and provides easy access to an inference routine. Structure checking is handled internally and error handling is graceful: the user is guaranteed an output even in the case of a partial failure. The output implements the same protocols as samples of the Æthel corpus, and is thus compatible with all of the latter's bells and whistles. Proofs can be pretty-printed, interactively processed and transformed (e.g. for semantic applications), or visualized using LaTeX as a middlewoman. For the more ambitious, training and evaluation utilities are also available.

5 Conclusion & Future Work

Thus concludes the demonstration tour of spind$^2\lambda$e: a unique neurosymbolic parser that can accurately and efficiently convert raw text into λ expressions. Unlike cheaper alternatives, these λ expressions are not structureless ad-hoc imitations produced from arbitrary decoding, but executable, type-safe and 100% **guaranteed correct** programs. The software focuses on Dutch, but the universality of the intuitionistic linear core allows easy cross-lingual adaptation that essentially boils down to retraining with a new type lexicon; a French implementation is currently in the works (De Pourtales et al., 2023).

As to what the future holds, the intention is to keep the module synchronized and up-to-date with Æthel: any upcoming major release of the latter will be reflected in an update of the former (be it soft patching or retraining). Compatibility aside, planned features include deploying the module as a web service, compiling it as a stand-alone package and documenting the annotations (so as to be more inclusive towards the type-uninitiated). Any performance, stability or efficiency improvements stemming from related research or moments of engineering inspiration are also likely to find their way to the user-facing front. Contributions and feedback are always welcome.

Limitations

The implementation described capitalizes on a disentanglement between neural and symbolic operations to improve efficiency. But doing so comes at the heavy price of a unidirectional data flow that lacks feedback. The symbolic component has the singular role of testing and verifying the neural output, but emits back no messages of its own. Failures may be caught, but they are nonetheless irrecoverable – a partial output that fails some structural constraint signifies an abrupt and non-negotiable end to the processing pipeline, significantly reducing coverage. A better operationalization would be to use the symbolic core to continuously ask for neural output as long as the structural constraints are not met (or the user is not satisfied with the parse provided). However, this would only be feasible if the neural components were to be extended with some notion of backtracking. In that sense, the parallel nature of both the supertagger and the parser becomes now a double-edged sword, hindering the potential applicability of standard heuristic algorithms like beam search.

More generally, the software carries the standard

---

[6]To comprehend how extremely unrealistic 2^7 is, consider that this would amount to finding the correct bijection out of $2^7! = 3.9 \times 10^{215}$ possibilities across 64 pairs of sets in parallel.

risks of any NLP architecture reliant on machine learning, namely linguistic biases inherited from the unsupervised pretraining of the incorporated language model and annotation biases derived from the supervised training over human-labeled data.

Acknowledgements

References

Samson Abramsky. 1993. Computational interpretations of linear logic. *Theoretical computer science*, 111(1-2):3–57.

Lasha Abzianidze. 2017. LangPro: Natural language theorem prover. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 115–120, Copenhagen, Denmark. Association for Computational Linguistics.

Stephen Clark. 2021. Something old, something new: Grammar-based CCG parsing with transformer models. *CoRR*, abs/2109.10044.

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

Vincent Danos and Laurent Regnier. 1989. The structure of multiplicatives. *Archive for Mathematical logic*, 28(3):181–203.

Caroline De Pourtales, Julien Rabault, Konstantinos Kogkalidis, and Richard Moot. 2023. DeepGrail: Neural proof nets for French. Technical report, LIRMM. Forthcoming.

Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*.

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

Roy Jonker and Anton Volgenant. 1987. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38(4):325–340.

Konstantinos Kogkalidis and Michael Moortgat. 2022. Geometry-aware supertagging with heterogeneous dynamic convolutions.

Konstantinos Kogkalidis, Michael Moortgat, and Tejaswini Deoskar. 2019. Constructive type-logical supertagging with self-attention networks. In *Proceedings of the 4th Workshop on Representation Learning for NLP (RepL4NLP-2019)*, pages 113–123, Florence, Italy. Association for Computational Linguistics.

Konstantinos Kogkalidis, Michael Moortgat, and Richard Moot. 2020a. ÆTHEL: Automatically extracted typelogical derivations for Dutch. In *Proceedings of the 12th Language Resources and Evaluation Conference*, pages 5257–5266, Marseille, France. European Language Resources Association.

Konstantinos Kogkalidis, Michael Moortgat, and Richard Moot. 2020b. Neural proof nets. In *Proceedings of the 24th Conference on Computational Natural Language Learning*, pages 26–40, Online. Association for Computational Linguistics.

François Lamarche. 2008. Proof nets for intuitionistic linear logic: Essential nets.

Gonzalo Mena, David Belanger, Scott Linderman, and Jasper Snoek. 2018. Learning latent permutations with gumbel-sinkhorn networks. In *International Conference on Learning Representations*.

Michael Moortgat. 1997. Categorial type logics. In *Handbook of logic and language*, pages 93–177. Elsevier.

Gabriel Peyré, Marco Cuturi, et al. 2019. Computational optimal transport: With applications to data science. *Foundations and Trends® in Machine Learning*, 11(5-6):355–607.

Jakob Prange, Nathan Schneider, and Vivek Srikumar. 2021. Supertagging the long tail with tree-structured decoding of complex categories. *Transactions of the Association for Computational Linguistics*, 9:243–260.

Mehrnoosh Sadrzadeh and Reinhard Muskens. 2018. Static and dynamic vector semantics for lambda calculus models of natural language. *J. Lang. Model.*, 6(2):319–351.

Richard Sinkhorn and Paul Knopp. 1967. Concerning nonnegative matrices and doubly stochastic matrices. *Pacific Journal of Mathematics*, 21(2):343–348.