Trustworthy Formal Natural Language Specifications

Colin S. Gordon
csgordon@drexel.edu
Drexel University
USA

Sergey Matskevich
sm3372@drexel.edu
Drexel University
USA

Abstract

Interactive proof assistants are computer programs carefully constructed to check a human-designed proof of a mathematical claim with high confidence in the implementation. However, this only validates truth of a formal claim, which may have been mistranslated from a claim made in natural language. This is especially problematic when using proof assistants to formally verify the correctness of software with respect to a natural language specification. The translation from informal to formal remains a challenging, time-consuming process that is difficult to audit for correctness.

This paper shows that it is possible to build support for specifications written in expressive subsets of natural language, within existing proof assistants, consistent with the principles used to establish trust and auditability in proof assistants themselves. We implement a means to provide specifications in a modularly extensible formal subset of English, and have them automatically translated into formal claims, entirely within the Lean proof assistant. Our approach is extensible (placing no permanent restrictions on grammatical structure), modular (allowing information about new words to be distributed alongside libraries), and produces proof certificates explaining how each word was interpreted and how the sentence's structure was used to compute the meaning.

We apply our prototype to the translation of various English descriptions of formal specifications from a popular textbook into Lean formalizations; all can be translated correctly with a modest lexicon with only minor modifications related to lexicon size.

CCS Concepts: • **Software and its engineering → Specification languages**; • **Human-centered computing → Natural language interfaces**.

Onward! '23, October 25–27, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0388-1/23/10…$15.00
https://doi.org/10.1145/3622758.3622890

Keywords: Natural language, formal specification, categorial grammar, computational linguistics

ACM Reference Format:
Colin S. Gordon and Sergey Matskevich. 2023. Trustworthy Formal Natural Language Specifications. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '23), October 25–27, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/3622758.3622890

1 Introduction

Proof assistants can establish very high confidence in the correctness of formal proofs, due to both their rigorous checking and attention to producing independently auditable evidence that the arument is correct [94, 98]. But one of the unavoidable points of trust for even a carefully-implemented proof assistant is the specifications themselves: proving the wrong theorem is of limited use. And only those who can read both formal and informal specifications can even consider whether this has occurred. This is particularly crucial for software verification: software specifications typically originate in natural language, and any accompanying formal specification comes afterwards — which increasingly occurs for compilers [70], operating systems [61], and other high-value software. Currently, the *only* bridge between the formal and informal specifications is the humans who perform the translation. There is no independently checkable record of this translation aside from the possibility of comments or notes by the translators — themselves largely in informal (though likely meticulous) natural language. Simply being familiar with both the specification language and the intended specification is insufficient by itself to bridge this gap [37, 38]: relating the two is a separate skill that is independently challenging to develop.

Ideally, it would be possible to give natural language specifications directly to the proof assistant, for example:

```
theorem thm : "addone is monotone" := ...
```

Robust support for such specifications could enable significant improvements in requirements tracing (machine *checked* mappings from natural language to formal results), including for artifact evaluation; education, where it could help students check their understanding of how either mathematics or program specifications are formalized logically; and even communication with non-technical clients who might wish to have some confidence that a formalization they do not themselves fully understand is correct. On the

last point, Wing's classic paper introducing formal methods [134] posits that customers may read the formal specifications produced from informal requirements, but this is only possible if the client can make sense of the formal specification itself. Machine-checked relationships between natural-language and formalized expressions of software properties can help bridge this knowledge gap by connecting formal properties to natural language a reader with less background in a specific formal logic could understand.

This paper develops the core of an *extensible* approach to allow specifications like:

```
theorem thm : pspec [| addone is monotone |] := ...
```

(where `pspec` returns the logical form of the natural language utterance in quotes). We envision such a system can eventually be used for the purposes above, to generate formal claims about mathematics or programs verified in a proof assistant, whether specified in a proof assistant's own logic, or indirectly through a foundational program logic [5] built inside a proof assistant; we primarily describe an implementation in an early version of Lean 4, with some discussion of a parallel prototype in Coq. Our goal is to enable specification of formal claims using modularly-extensible fragments of natural language, within the proof assistant, which are automatically translated to forms logically equivalent (and in practice often near-identical) to those a verification expert may give while accounting for what is easier or more difficult to prove; proofs (often trivial) relating the expert claim to the formalization extracted from natural language then extend the formal proof with a formalized connection to a description of functionality that may be more easily understood by non-experts in verification, additionally ensuring that the mathematical formalization is consistent with a less formal statement.

This is not a job for machine-learning-centric natural language processing, which despite the recent trend in work on "autoformalization" [135] is incompatible with the goals of using a proof assistant for formal verification. There is no guarantee a learned translation is sensible, and if a translation from natural to formal language ends up being surprising, few machine learning approaches produce an auditable trail of evidence for *why* that translation was produced by the trained model, and no way to precisely fix misunderstandings of specific words. The current state-of-the-art in explaining such translations are techniques such as *salience maps*, which essentially highlight which input words were most influential to the system output without providing a linguistically-grounded reason for the behavior [115]. Meanwhile, *proof certificates* play a central role in the design of trustworthy proof assistants [46, 98] and foundational program verification [5]. Moreover, as proof assistants are often used to formalize properties of new mathematics or new programs, often using new terminology, there will often be a lack of training data for mapping natural language to a formal property. Later, we point out additional ways that the needs of trusted formalization of natural language specifications run afoul of many of machine learning's known limitations, while requiring few of its advantages. (We also point out limited ways machine learning can play a role in optimizing the techniques we employ.)

Fortunately the field of linguistics predates machine learning. Formalizing *categorial grammar* [2, 11, 68, 117] carefully in a proof assistant offers a path to a natural, auditable way to bridge the gap between formal and natural-language specification. In this paper we show a prototype demonstrating that it is possible to parse a string containing a natural language specification into a semantic representation that can be used directly in proofs within a proof assistant (i.e., a proposition in the Calculus of Inductive Constructions), in a linguistically principled way, using typeclasses [114, 116]. We argue that this approach is modular and can extend to sophisticated and substantial subsets of natural language. We also analyze how the trusted computing base is affected when considering trust of a formal verification up to the natural language specification. Unlike many previous attempts at natural language interfaces (Section 7.2), our goal is to supplement, not supplant, traditional proof assistant interfaces.

This paper establishes that the theoretical core is within reach, that these techniques can capture the grammar and semantics of a small sample of natural language specifications from a textbook on verified functional programs, and that it is feasible to implement translation algorithms inside current proof assistants, as a library (though leading to suggestions for improvements to those features).

2 Background & Motivation

This section provides a condensed (and therefore somewhat biased) background in natural language processing and categorial grammar, from a programming languages perspective.

2.1 Categorial Grammars

Categorial grammars are a family of related techniques [89] applying ideas from logic to relate the syntax of natural language with formal representations of what that syntax denotes — often called *logical form* — growing out of Lambek's work [68] in the 1950s (which coined the term). The core idea is to build a sort of substructural type theory where base types correspond to grammatical *categories* (hence *categorial*), from which more complex grammatical categories can be defined. A set of inference rules is then used to define, simultaneously, how grammatical categories combine into larger sentence fragments and how those smaller fragments' meanings (logical forms) are combined into larger meanings. This process bottoms-out at a *lexicon*, giving for each word its grammatical roles (types) and associated denotations. Thus categorial grammar is a system of simulatenously parsing natural language from strings and assigning denotational

$$\backslash\text{-Elim}\ \frac{\Gamma \vdash A \Rightarrow a \quad \Delta \vdash A\backslash B \Rightarrow f}{\Gamma, \Delta \vdash B \Rightarrow (f\ a)} \qquad /\text{-Elim}\ \frac{\Gamma \vdash A/B \Rightarrow f \quad \Delta \vdash B \Rightarrow a}{\Gamma, \Delta \vdash A \Rightarrow (f\ a)}$$

$$/\text{-Comp}\ \frac{\Gamma \vdash A/B \Rightarrow e \quad \Delta \vdash B/C \Rightarrow f}{\Gamma, \Delta \vdash A/C \Rightarrow (e \circ f)} \qquad \backslash\text{-Comp}\ \frac{\Gamma \vdash A\backslash B \Rightarrow e \quad \Delta \vdash B\backslash C \Rightarrow f}{\Gamma, \Delta \vdash A\backslash C \Rightarrow (f \circ e)}$$

Figure 1. A selection of rules used in this paper, all derivable in CTLs, CCG, and other categorial grammars.

semantics — a process traditionally referred to as *semantic parsing* in the computational linguistics literature.

Most prominent in the linguistics community are *combinatory categorial grammars* (CCGs) [117], though also relevant are the *categorial type logics* (CTLs [20][1]). Work on CCGs epmhasizes appropriate constructs for linguistic ends, while CTLs hew close to Lambek's view [68] of categorial grammars as substructural logics for linguistics. While these reflect very different philosophical and practical aims, for our present purposes the distinction is immaterial: it is widely held, and in some cases formalized [42, 63], that rules used in CCGs (including the variant with the most sophisticated linguistic treatments [9]) correspond to theorems in particular CTLs [80]. In this work we use only principles common across all categorial grammars, including CCGs and CTLs.

All categorial grammars parse by combining sentence fragments based on their grammatical types. These types include both atomic primitives (such as noun phrases) as well as more complex types, called *slash-types* that indicate a predicate argument structure (which are used to model, for example, most classes of verbs). Oversimplifying slightly, categorial grammars treat parsing as logical deduction in a residuated non-commutative linear logic.[2] This is essentially a family of linear logics without the structural rule for freely commuting the order of assumptions, thus modeling sensitivity to word order, and picking up as a consequence *two* forms of implication corresponding to whether an implication expects its argument to the left or to the right.[3] The model for the logic is a sequence of words, and types correspond to the grammatical role of a sentence fragment.

A/B is the grammatical type for a fragment that, when given a B to its right, forms an A. $A\backslash B$ is the grammatical type for a fragment that, when given an A to its left, forms a B. In both cases, the argument is "under" the slash, and the result is "above" it.[4]) These are called *slash types*. The grammars include rules to combine adjacent parts of a sentence. The elimination rules for slash types are the first two in Figure 1. The judgment $\Gamma \vdash C \Rightarrow e$ is read as claiming the sequence of words Γ can be combined to form a sentence fragment of grammatical type C, whose underlying semantic form — logical form — is given by e. e is a term drawn from the logical language being used to represent sentence meaning, typically a simply-typed lambda calculus in keeping with Montague [76–78, 91], though in our work we follow the alternative [15, 24, 103, 119] of targeting a dependently-typed calculus. Figure 1 also includes forward and backward composition; rules tend to come in pairs for each direction. The rest of the paper can be followed with only the four rules of this figure, though our implementation includes additional rules from the formal linguistics literature.

A *lexicon* gives the grammatical role and semantics for individual words, providing the starting point for combining fragments. Categorial grammars push all knowledge specific to a particular human language into the lexicon, in categorizing how individual words are used. This allows the core principles to be reused across languages, as evidenced by wide-coverage lexicons for a variety of natural languages including English [53], German [51], Hindi [4], Japanese [75], Arabic [18], French [82], and Dutch and Italian [1].

Together, these allow filling in choices for the metavariables in the rules above, permitting derivations like that in Figure 2. Each grammatical type C corresponds to a particular type in the underlying lambda calculus and the underlying *semantic* type is determined by a systematic translation from the syntactic grammatical type. Borrowing more notation from logic (where this idea is known as a Tarski-style universe [73]), we write $\lfloor C \rfloor$ for C's semantic type. Both slash types correspond to function types in the lambda calculus: $\lfloor A/B \rfloor = \lfloor B\backslash A \rfloor = B \to A$. An invariant of the judgment $\Gamma \vdash C \Rightarrow e$ is that in the underlying logic, e always has type $\lfloor C \rfloor$. This invariant explains why it is correct for "four" to have semantics 4 while "is even" has a function as its semantics. In proof assistants based on type theory, like Coq and Lean, the set of grammatical types can be given as a datatype declaration, and the interpretation function as a function from grammatical types to proof assistant types.

These few rules are enough to formalize a small fragment of English in ways consistent with linguistics research, and demonstrate the possibility of interpreting natural language specifications *within* a proof assistant, in a well-founded and extensible way. Our initial choice of rules is limited, but not fundamentally so. CCGs recognize *mildly context-sensitive languages* [57, 64], which are believed to cover the full range of grammatical constructions in any natural language [118]. All rules of CCGs are encodable in the way we describe, as

[1] Occasionally also called *type-logical grammars* (TLGs).

[2] Technically only CTLs [85] take this as an epistemilogical commitment, while CCGs [117]) are agnostic, inheriting such a relation via Baldridge and Kruijff's work [9].

[3] It is the presence of the ability to commute assumptions arbitrarily that allows a single implication to suffice in standard logics.

[4] We follow CTL notation rather than CCG notation (which always puts results to the left) as users of proof assistants tend to be familiar with a range of logics, so the CCG syntax would likely confuse users already familiar with the Lambek calculus and related systems. This notational choice is orthogonal to the choice of which rules to employ.

$$\dfrac{\dfrac{}{\text{“four”} \vdash NP \Rightarrow 4} \quad \dfrac{\dfrac{}{\text{“is”} \vdash (NP\backslash S)/ADJ \Rightarrow \lambda p.\, \lambda x.\, p\ x} \quad \dfrac{}{\text{“even”} \vdash ADJ \Rightarrow \text{even}}\ /\text{-Elim}}{\text{“is even”} \vdash NP\backslash S \Rightarrow (\lambda x.\, \text{even}\ x)}}{\text{“four is even”} \vdash S \Rightarrow \text{even}\ 4}\ \backslash\text{-Elim}$$

Figure 2. A simple semantic parse using categorial grammar.

are the rules of Turing-complete CTLs [21]. Ultimately the question of which rules are required is an empirical one, considering both linguistic constructions and the complexity of recognizing these grammars For this paper we focus on a small set of rules including those above, which are common to most categorial grammars. The rules we use in this paper are intended as a foundation: as the next section points out, categorial grammars are highly modular, allowing not only simple additions of new lexical entries, but simple modular addition of support for new grammatical constructions, often by direct implementation of treatments that have been extensively validated in the linguistics literature.

2.2 Formal Grammars for Natural Languages: Why Categorial Grammars?

Producing formal grammars (categorial or otherwise) for natural language is technically a slight misnomer. Natural language is inherently open-ended, allowing new grammatical constructions to become accepted over time, and new words to be proposed, among other changes; we are ignoring, for present purposes, matters of pronunciation (phonology and phonetics) and changes in written form (orthography), not to mention signed languages. However, as soon as a set of acceptable words and their uses are fixed into a formal grammar, that grammar now describes merely a subset of the actual natural language it models, and by its nature excludes innovations in the lexicon and grammatical structure. This has far-reaching effects if our interest is in writing formal specifications in something approximating natural language: individual programming projects often invent their own terminology! Thus if we use categorial grammars (or any other formal theory of grammar) to parse specifications into a logical form, we are necessarily writing formal specifications in a *formalized subset* of natural language,[5] rather than open-ended natural language.

This is an inherent limitation to any approach to mechanized manipulation of language meaning.[6] The result is that novel grammatical constructions (or simply some not accounted for in the design of the grammar), or previously-unknown words will not be handled *at all* by the formal grammar. This is why the approach we are suggesting is sometimes called *controlled natural language* [40, 65].

However, choice of grammar formalism can moderate these limitations. Rather than attempting to treat all possible current and future natural language sentences out of the box (which is impossible), we can choose foundations that are modularly extensible to new words *and new grammatical constructions*, and can grow incrementally over time as terminology or grammar evolves or changes. Categorial gramamrs are examples of *lexicalized* grammar formalisms [109]. In contrast to the context-free grammars that are well-known in programming languages and software engineering, all knowledge of how an individual word is used is contained in the lexicon alone. Whereas a CFG treatment of English might have separate non-terminals for verb phrases, transitive verbs, and intransitive verbs, and individual words would simply be terminals, in categorial grammars all verbs, including ditransitive verbs (those with direct and indirect objects) are simply built from noun phrases and slash types. A good example of a non-trivial modular addition in our current prototype is in-situ quantification: when a use of the indefinite article “a” in the *middle* of a sentence introduces a top-level existential quantifier in the semantics, as in “addone given 3 returns a natural” — this translates to $\exists x : \mathbb{N}.\, \text{addone}(3) = x$, but knowledge of the existential handling is isolated to the entry for “a.” Adding this support to other systems (Section 7.2) is non-trivial.

Because categorial grammars focus on a set of base categories plus categories that correspond to functions (slash types), the modularity is improved even beyond other lexicalized formalisms, which require a finite set of grammatical categories. Additional slash-based categories can be introduced freely. And the menagerie of categorial grammar variants are largely completely compatible with each other: the *rules* for categorial grammar are modular. This is a suitable basis for tackling a problem whose true empirical needs may not be fully established for years to come, allowing the system to evolve in a flexible way.

Categorial grammars have been extensively validated for modeling a wide array of complex grammatical phenomena across a diverse array of language families, and the linguistics literature contains decades' worth of work studying how to capture subtle linguistic semantic phenomena in these languages via categorial grammars [1, 13, 20, 55, 83, 85]. This means that building on (for now) basic categorial grammars is forward-compatible with decades of thoroughly-validated

[5] Hence the double meaning of our title.

[6] As opposed to mechanized manipulation of language *form* (text) alone, as is done in current large language models, where open-endedness is well-supported (e.g., via the well-known *attention* [127] mechanism, which essentially heuristically identifies chunks of input text to copy into the output), but meaning is not (which is why large language models consistently struggle with basic boolean reasoning [36, 90, 123, 124]).

work on detailed linguistic theories accounting for both syntax *and* semantics of significant natural language fragments. If it is discovered that an additional grammatical structure is necessary or useful, we can essentially consult the linguistics literature and add just the required rules; it is long-standing practice for the different branches to borrow each others' innovations; with rare exception, these combinations are straightforward and require almost no adaptation (and many have already been explored and validated). This stands in contrast to most prior work on natural language inspired programming and specification (Section 7.2), most of which has broadly either ignored linguistic treatments of natural language semantics, or built on foundations that had not yet been thoroughly validated over a period of decades (and ultimately did not stand up to such scrutiny).

This is especially important considering the three major ways a formalized, or controlled, fragment of natural language can fall short. Any formal subset, as noted, omits some natural language out of necessity; the linguistics literature calls this *syntactic under-generation*: the formalized subset is a formal language, and undergeneration occurs when syntactically valid text is not accepted as syntactically valid. There are two related issues, however. One is *syntactic over-generation*: accepting text as syntactically valid that would not generally be accepted as valid by humans. If such text can be written by mistake, and accepted, it is akin to an overly-permissive programming language parser: any interpretation is wrong, as the text should have been rejected. The final major issue is misinterpretation: interpreting syntactically valid text in the formal subset in a way that disagrees with reasonable human interpretations would result in a formal (controlled) natural language subset that is at best confusing, but more generally significantly error-prone if used as a tool frontend. While undergeneration is inherent to formalized natural language, work in linguistics on formal semantics and the interface of syntax and semantics is evaluated largely on the absence of overgeneration and misinterpretation. This is a key source in confidence of the accuracy of our formalization: both the core treatment and some extensions described in Section 4.2 were read directly out of authoritative accounts in the linguistics literature.

3 Categorial Grammars for CIC Specifications

This section describes a model of a very small fragment of English for describing simple mathematics. Our goal is not to present a polished and complete natural language fragment suitable for a wide range of nearly-arbitrary specifications; such a goal is admirable, but such large-scale projects take many years to complete [3, 71, 72]. Our goals are instead to demonstrate the feasibility of adapting extant linguistic characterizations to natural language specifications; show that existing proof assistants (e.g., Lean) include enough machinery to do so now, internally to the tools; and that the selected approach is modular enough to permit growing the scope of a system piecewise over an extended period of time.

We describe how to use the typeclass support of modern proof assistants [114, 116] to perform semantic parsing from natural language[7] to the Calculus of Inductive Constructions (CIC) [93], or alternatively (see Section 4.3) embedded logics. This approach naturally supports an open-ended lexicon, which is essential to modularly expanding the set of words handled by semantic parsing (Section 4). This ensures the extended lexicon can grow in tandem with a formal development, with organization chosen by developers rather than dictated by an external tool, and with the proof assistant automatically checking validity of the lexicon extensions at the same time it checks validity of the rest of the formalization.

While in principle the translation could be carried out via an external tool which may offer other benefits, this approach works in proof assistants today, without additional toolchain issues or concerns about disagreements between an external tool and a proof assistant over what is valid.

Our goal is to allow verifiers to either (1) directly specify in English; or (2) carry out proof developments mostly as they already would, but to additionally state a specification in a formalized subset of English, and prove a lemma showing the traditional specification implies (or is equivalent to) the translated specification. Both cases result in a validataed, auditable connection between English and the proofs.

We initially describe a general overview and notation we use throughout the paper. Section 3.4 describes further changes which compromise readability but drastically improve parsing performance.

The Basics. Most work on categorial grammars lumps all entities — frogs, people, books — into a single semantic type E for "entities", with different varieties distinguished by predicates: e.g., $\mathsf{three} : E$ and $\mathsf{Mary} : E$, but $\mathsf{number}(\mathsf{Mary}) = \mathsf{false}$. This approach is rooted in the assumption that first order logic is an appropriate semantic model for sentence meaning. That assumption is plausible for many general circumstances, popular in formal linguistics, and particularly useful for modeling figurative speech, but stands at odds with making natural language claims about mathematical objects defined in intuitionistic type theory. For logical representations to talk about entities in CIC — which distinguishes natural numbers, rings, monoids, and so on with different types — adjustments must be made. Noun phrases and related syntactic categories must be parameterized by the semantic (CIC) type of entity they concern.

We follow linguistically-motivated work using intuitionistic type theories like CIC for exploring possible logical forms [14, 15, 22, 102–104, 119], in using an alternative model

[7] In this paper, English, but in principle any other natural language with thorough treatments in categorial grammar [1, 4, 51–53, 75, 82].

```
inductive Cat : Type where
  | S -- Sentence/proposition
  | NP : forall {x:Type}, Cat
  | rSlash : Cat -> Cat -> Cat -- A/B
  | lSlash : Cat -> Cat -> Cat -- A\B
  | ADJ : forall {x:Type}, Cat
  | CN : forall {A:Type}, Cat.

@[simp]
def interp (c:Cat) : Type :=
  match c with
  | S => Prop
  | @NP x => x
  | @ADJ x => x -> Prop
  | rslash a b => interp b -> interp a
  | lslash a b => interp a -> interp b
  | @CN x => x -> Prop
```

Figure 3. Core grammatical categories as an inductive type Cat and the mapping $\lfloor - \rfloor$ from grammatical types to semantic types as interp.

where many grammatical categories are indexed by the underlying semantic type to which they refer.

CIC is expressive enough to give the set of grammatical types as a datatype `Cat`, and to give the interpretation of those types into semantic types as a recursive function `interp` within Lean, as shown in Figure 3. Grammatical types (called *categories* in the linguistics literature) Cat include the aforementioned slash types, sentences (S); noun phrases ($NP\ A$) denoting objects of type A in CIC; adjectives ($ADJ\ A$) denoting predicates over such objects[8]; and common nouns ($CN\ A$) denoting predicates on A, both refining the domain of discourse and imposing constraints on the semantic type indices of other phrases in the context of a sentence, used in cases where a sentence must refer to a common class of objects (i.e., a type), such as "natural numbers" or "rings." As mentioned previously, the slash types correspond to function types (the direction is relevant only in the grammar, not the semantics), sentences are modeled by propositions (the type of logical claims), noun phrases of Lean type `t` correspond to elements of `t`, and similarly adjectives correspond to predicates on such types (i.e., elements of `t -> Prop`). These are modeled by the function `interp`, which maps grammatical types to other Lean types. Common nouns are interpreted as predicates as well — this permits constructions such as "even natural" to be parsed as common nouns, with semantics (`λx:Nat, even x`). The `@[simp]` definition marks the definition for unfolding during typeclass resolution.

[8]Not all grammatical treatments treat adjectives directly as primitive (some approaches derive them from other primitive categories), but all frameworks include some for of adjective which is semantically a predicate.

3.1 Combination Rules

Parsing natural language specifications requires automatically applying rules like /-Elim to combine sentence fragments. Rather than modifying a proof assistant, we can use existing trusted[9] functionality to do this for us: typeclasses [114, 116]. These are a mechanism for parameterizing function definitions by a set of (often derivable) operations. Proof assistants such as Coq and Lean permit declaring a typeclass (roughly, an interface), and declaring implementations associated with certain types. The implementations may be parameterized by implementations for other types (such as defining an ordering on pairs in terms of orderings for each component of the pair). When a function is called that relies on a set of operations, the proof assistant attempts to use a form of higher-order unification to construct an appropriate implementation. It is possible to encode categorial grammar rules into typeclasses. Each judgment form corresponds to a typeclass, and each rule corresponds to an instance (implementation) of the typeclass.

For now, assume we model the order of composition of sentence fragments as a binary tree of words stored in leaf nodes, explicitly modeling the word groupings as bunched contexts in a substructural logic. Our actual implementation employs some optimizations (Section 3.4) which hide this structure and perform all work modulo associativity rules ($\Gamma, (\Delta, \Upsilon) \equiv (\Gamma, \Delta), \Upsilon$), but this version (consistent with an initial prototype) is a useful learning step. We will write `#` to join two sequences of words. The constructor `one` packages a word into a leaf node.

We define the judgement form $\Gamma \vdash T \Rightarrow e$ as:

```
class Synth (ws:tree String) (c:Cat) where
  denotation : interp c
attribute [simp] Synth.denotation
```

If an instance of `Synth ws C` exists, it comes with an operation `denote` that produces a Lean value of type `interp C` — $\lfloor C \rfloor$. Because e is viewed as an output we would like to query, it is defined as a member of the typeclass, rather than as an additional index. When deriving a formal specification for sentence s, we will arrange for the typeclass machinery to locate an instance of `Synth s S` — checking that the sentence s is a grammatically valid sentence — and request its term denotation when necessary. Translating a specification given by the list of words `ws` corresponds to parsing `w` as a sentence: finding an instance of `Synth ws S`. We define an instance for each rule to encode, such as this one corresponding to \-Elim, which applies the logical form of the functional to the logical form of the argument:

[9]Officially, typeclasses are not part of a trusted computing base, as they elaborate to record operations before being passed to the core proof checking apparatus. In practice, they mediate *which* terms are passed to the core, so calling them *untrusted* would be a misnomer [98].

```
instance SynthLApp {s1 s2 c1 c2}[L:Synth s1 c1]
    [R:Synth s2 (c1 \ c2)] : Synth (s1#s2) c2 where
  denotation := R.denotation L.denotation
```

and for leftward composition (\-Comp):

```
instance LComp {s s' c1 c2 c3}
  [L:Synth s (c1 \ c2)] [R:Synth s' (c2 \ c3)]
  : Synth (s#s') (c1 \ c3) where
  denotation x := R.denotation (L.denotation x)
```

The remaining rules of Figure 1 can also be encoded in this way. The `[simp]` attribute on `denotation` ensures it will automatically be simplified by tactics like `simp` (roughly a combination of Coq's `simpl` and parts of `auto`).

In addition to exploiting the proof assistant's built-in search for parsing, the use of typeclasses means the set of rules is extensible, meaning additional rules could be added for additional grammatical coverage or to speed up the parsing process. More critically, however, it allows easy modular extension of the lexicon with additional words.

3.2 Lexicon

The lexicon is encoded via a typeclass `lexicon`, which assigns grammatical types and semantics to individual words rather than series of words. This is then tied to the `Synth` typeclass:

```
class lexicon (w:String) (c:Cat) where
  denotation : interp c
attribute [simp] Synth.denotation
instance SynthLex {w:String}{C:Cat}[lexicon w C] :
    Synth (one w) C where
  denotation := lexicon.denotation w
```

Lean permits declaring multiple instances for the same word (e.g., if a word has multiple meanings of different grammatical types), giving essentially a free variant of intersection types [81] without the coherence issues described by Carpenter [20] (only one definition will be chosen per appearance of the word). Thus, a dictionary for our approach consists of a set of instance declarations for `lexicon`:

```
instance fourlex : lexicon "four" (@NP Nat) where
  denotation := 4
instance evenlex : lexicon "even" (@ADJ Nat) where
  denotation := fun x => even x = true
instance n_is_adj {T}: lexicon "is"
  (((@NP T) \ S) / (@ADJ T)) where
  denotation := fun a n => a n
instance n_is_n_lex {T}: lexicon "is"
  (((@NP T) \ S) / (@NP T)) where
  denotation := fun a n => a = n
instance given_lex {A B}: lexicon "given"
  ((@NP (A -> B)) \ ((@NP B) / (@NP A))) where
  denotation := fun f arg => f arg
```

Here we have defined two different meanings for "is" allowing it to be used to apply an adjective (e.g., as in "four is even"), or to denote equality (as in "four is four"). The difference between the two, beyond their denotation is the grammatical types: both expect a noun phrase to the left, and some other word to the right: an adjective in the first case, or another noun in the second. Note that in both cases, the adjective or noun phrase must match the type of underlying Lean object the the left-side noun phrase refers to: the argument `n` is in both cases a variable of type `interp (@NP A)=A` because that is the argument of the outermost slash type, while the second argument in each entry corresponds to the interpretation of the second slash type's argument (a predicate or an additional term, respectively). Coupled with a development-specific bit of lexicon to name a particular Lean object of interest:

```
instance addonelex : lexicon Prop "addone"
  (@NP (Nat -> Nat)) where
  denotation := addone -- λx. x + 1
```

this approach permits giving correct denotations to both:

$$[\![\text{addone is monotone}]\!] \equiv \text{monotone}(\text{addone})$$
$$[\![\text{addone given 3 is 4}]\!] \equiv (\text{addone } 3) = 4$$

3.2.1 Quantifiers.

Quantifiers over A can be given grammatical type

$$\text{Quant } A \equiv (S/(NP_A\backslash S)/CN_A)$$

abbreviated with a macro. Thus, a quantifier looks to its right first for a common noun (corresponding to the word identifying the Lean type to quantify over), and after that is combined, the result looks further to the right for a sentence fragment expecting such a thing to its left. (After binding with the common noun, the remainder is in fact a continuation [13].) Then adding a lexicon entry for "every":

```
instance forall_lex {A}: lexicon "every" (quant A)
    where
  denotation := fun N P => forall x, N x -> P x
```

and another for the common noun "natural" (number) allows correctly parsing sentences like

$$[\![\text{every natural is even}]\!] \equiv \forall(n : nat).\ (\text{even } n)$$

(Recall, we must still be able to state claims that are false.) The common noun constrains the quantifier to work with noun phrases referring to natural numbers, also using the predicate semantics to constrain the claim to those elements of the quantified type matching the predicate, as in[10]

$$[\![\text{every odd natural is even}]\!] \equiv \forall(n : nat).\ \text{odd } n \rightarrow (\text{even } n)$$

[10] An additional grammar rule lifts adjectives to noun modifiers CN/CN, allowing "odd natural" to be parsed as a noun.

3.2.2 Coordination. One aspect of natural language which is the source of some interest is that the words "and" and "or" (or their equivalents in other languages) can often be used to combine sentences fragments of widely varying grammatical types. For example, in "four is even and positive" the word "and" conjoins two adjectives: "even" and "positive." Yet in "four is even and is positive" it conjoins two phrases of grammatical type $NP_{\text{nat}} \backslash S$ ("is even" and "is positive").

We can directly adopt a solution from the computational linguistics literature [20], and formalize that "and" and "or" apply to any semantic type that is a function into (a function into…) the type `Prop` of propositions. We define an additional typeclass to recognize such "Prop-like" grammatical types inductively, starting with the grammatical types S and ADJ, and inductively including slash types whose result type is also "Prop-like", which define an operation to lift boolean semantics through repeated functions. We then add a polymorphic lexicon entry for each of "and" and "or" which assigns them any "Prop-like" type.

Thus in a sentence like "four is even and is positive" the two conjuncts are recognized as Prop-like (their underlying semantic type is Nat $\to$ Prop), and the operations of the typeclass recognizing this automatically lift a binary operation on `Prop` to a binary operation on predicates — the classic pointwise lifting of the underlying Heyting algebra [69]. For "and" this lifts logical conjunction to $\lambda P.\, \lambda Q.\, \lambda x.\, P\ x \wedge Q\ x$, which is exactly what is needed — the grammar rules will apply this function to the semantics of the even and positive predicates, and finally 4. Disjunction is handled similarly, and this generalizes to arbitrarily complex slash types whose final semantic result is `Prop`.

3.3 Using Specifications

Defining a function from sentence representations to the denotations of the words in order is then relatively simple:

```
@[simp]
def pspec (ws:tree String) [sem:Synth ws S] : Prop :=
  sem.denotation
```

When invoked with a tree of strings `s`, Lean will search for an instance of `Synth s S` — a parse of the string tree as a complete sentence. The semantics of a sentence has Lean type `Prop` (a proposition, or logical claim, to be proven). To complete the readable surface syntax from the introduction, rather than hand-constructing data structures, we define a macro `[|...|]` that takes a sequence of words (technically, identifiers) and constructs the appropriate representation.

Thus we may translate a range of specifications given an appropriate lexicon, including those below (sugared into math notation for space and readability):

$$[\![\text{addone is monotone}]\!] \equiv \forall x, y : \mathbf{N}.\, x \le y \Rightarrow \text{addone } x \le \text{addone } y$$

$$[\![\text{every natural is non-negative}]\!] \equiv \forall n : \mathbf{N}.\, n \ge 0$$

$$[\![\text{every natural is non-negative and some natural is even}]\!] \equiv (\forall n : \mathbf{N}.\, n \ge 0) \wedge (\exists n : \mathbf{N}.\, \text{even } n)$$

In particular, we can observe how these specifications manifest during interactive proof:

```
def addone_mono : pspec [|addone is monotone|] :=
  by simp
     --⊢ ∀ (x y : Nat), x ≤ y → addone x ≤ addone y
     intro x y h ...
```

i.e., after simplification, a proof may continue either directly, or by appeal to a separate lemma stated purely formally in the case the proof's only goal is to bridge formal and natural language specifications.

If Lean cannot find a `Synth` instance for a specification, the user sees one of two error messages from Lean itself. One possibility is that Lean has exhaustively explored the possibilities and no parse exists (this is typical of a specification using a word not in the lexicon). The other possibility is that Lean has reached its timeout for typeclass instance search. This is linear "fuel"-type timeout parameter that may adjusted per-file, meaning that if a file using these natural language specifications requires longer search times, this can be done locally without forcing an entire development to use the longer search. Anecdotally, use of this style of specification typically does require increasing this parameter, but on a relatively old 4-core machine from 2015, most of the specifications discussed in this paper are parsed within the first author's reaction time, with Lean never exceeding 25% of a single core during parsing, with 2.6GB of memory consumed. Sections 3.4 and 6 address performance in more detail.

3.4 Performance

The performance of semantic parsing depends on both the underlying typeclass resolution procedure, as well as the space of derivations that must be explored during parsing (itself dependent on the data manipulated within rules, as well as which grammatical rules are included).

The structural rules encoded in the `Synth` typeclass instances are the primary drivers of search costs, along with the `lexicon` instances. Since most words have only one or a very small number of grammatical roles (in general, not just in our small prototype [51–53]), we expect that lexicon ambiguity will *not* be a major driver of search costs. Instead, most costs should arise from exploring the space of derivations.

The direct implementation approach described above becomes quite slow for sentences over a few words, so we apply a number of optimizations, all well-established in the computational linguistics literature.

We exploit several classic optimizations from work on parsing natural langauges with Prolog. The first issue is that given a sentence of length n, there are $(2n!)/((n+1)!n!)$ ways to associate segments of that sentence into a binary

tree like that used in the earlier explanation (the *n*th Catalan number). The naïve approach above requires exploring potentially all of these trees in order to yield a complete search procedure, which is prohibitively expensive for even modest sentences. The solution is to represent the sentence parenthesization without explicitly manifesting unique structures. There are established techniques for this in the literature on parsing via logic programming [32, 95], specifically the technique of *difference lists*. The idea is that rather than representing the parenthesizations as a tree, to represent it as a *span*. A difference list is a pair of lists X and Y where Y is a *suffix* of X, and the difference list then represents the list segment from the start of X to the suffix covered by Y — `[3,4,5]` and `[5]` represents the prefix `[3,4]`, but without explicitly constructing a fresh set of cons cells: the spine of the original list of words is reused.

Lean's use of tabled resolution [114] (roughly, memoization) works nicely with this change in representation, but just as in Prolog, memoizing based on large structures (specifically, lists of UTF-8 strings) is expensive. So we also apply the other standard parsing-via-logic-programming optimization of representing lists not as explicit lists, but as natural numbers representing the starting index of the sublist (roughly, as how many elements to drop from the list of words being parsed). So for a locally-fixed list `["three","is","even"]`, rather than reprenting the full span as the list, or the explicit difference list pair of `["three","is","even"]` and `[]`, the representation of the span is the pair of 0 and 3 (the list segment starting at index 0 and dropping index 3 and beyond). Even in unary form this makes the table lookups faster (comparing unary naturals of maximum size/length n is faster than comparing lists of maximum size/length n containing strings which must be compared), but Lean additionally represents naturals internally via the GMP library, making the comparisons even faster: naturals within range of a machine word are represented as a machine word (plus tag).

Applying this latter optimization unfortunately means we must also compute word-list indices using typeclasses, in order to tie lexicon entries stated in terms of words to spans in terms of natural numbers. It also means that because the string is no longer an explicit part of the context of the `Synth` judgment/typeclass, we must use Lean's module boundaries to isolate specification searches from each other. For each specification we open a new module, declare the string to parse to the typeclass machinery with a module-local instance, and then use a variant of `spec` adapted for the changes above. This adds a bit of verbosity, but could in principle be alleviated via macros or additional facilities for exposing control over clearing the memoization tables; currently the only such mechanism for this in Lean is that tables are cleared of anything module-local when that module's checking is complete.

Finally, we apply standard techniques [35] to further prune the search space of instances of spurious ambiguity (searching only over normal form derivations), and impose two static bounds on the search: limiting how many times coordinators may undergo pointwise lifting, and using Lean's existing heartbeat timeout counter, which gives roughly linear control over typeclass search time.

Worst Cases. The particular grammar rules we are currently working with have only context-free recognition power, known to require cubic time in the length of the sentence to parse. The mildly-context-sensitive classes of categorial grammars favored by linguists have $O(n^6)$ worst-case parsing cost [57], though in practice the common case can be made quite fast [26–28].

4 Modularity and Extension: Growing a Lexicon, Handling More Logics

The previous section described only a small fragment of English suitable for formalizing mathematical claims. Because categorial grammars are *lexicalized* grammars (recall Section 2.2) which use a small number of special-purpose rules (like those in Figure 1) and otherwise leave knowledge of a language to per-word entries, they naturally support modular extension. In particular, the availability of slash types (directed function types) affords significant flexibility to define new grammatical roles without disrupting the core rules, and extensions to attach modalities to the slashes [9, 80] allow further constraints capturing the subtleties of natural language to be captured solely by giving precise grammatical types (and semantics) to individual words.

4.1 Managing Words

Adding new words to a categorial grammar lexicon is conceptually as simple as adding the word, particular grammatical type, and associated denotation to the database. This makes it easy to extend a system with new concepts (e.g., new algebraic structures); lexicon entries to deal with concepts defined in a proof assistant library can be distributed as a part of that library. Conversely, if a word or particular usage of a word is found to be confusing to humans, leading to ambiguity, or otherwise problematic, it can be removed from the lexicon while affecting only inputs that use that word in that way (i.e., the problematic ones).

In practice the situation will be more complex, but we expect most extension to require little, if any, special linguistic knowledge. Assuming a robust core lexicon, it is likely that most extensions will be additions of words with simpler categories. Experiments on a large English lexicon showed [52] that when training on most of lexicon, the unseen words in a held-out test set were primarily nouns (35.1%) or transformations of nouns (e.g., adjectives, at 29.1%). These are

the simplest categories to provide semantics for (types, objects, and predicates), strongly suggesting that proof assistant users with no special linguistics background could make most extensions themselves. Similar experiments for a wide-coverage lexicon of German [51] show over half of unknown words to be nouns, suggesting this feasibility extends beyond just English.

Careful readers or prior students of linguistics may have wondered when matters of verb tense, noun case and number, grammatical gender,[11] etc. would arise. In full linguistic treatments, these are reflected in additional parameters to some grammatical categories. So for example, in our setting a noun phrase would be parameterized not only by the underlying referent type, but also by the case, number and so on; lexicon entries would then carry these through appropriately (making it possible to for example, require the direct object of a verb to be in the accusative case rather than nominative). We have omitted such a treatment here partly because it would obscure the key ideas while adding little value, partly because many of these distinctions are less important for our examples in English (which has fewer syntactic case distinctions than other languages), and partly because some aspects (like tense) may make sense only for specific embedded specification logics. We leave general-purpose treatments of these issues to future work.

4.2 Supporting Additional Grammatical Constructions

Formalization of significant fragments of language much deal with more subtle constructions that what we have described so far can handle. However, what we have described thus far is essentially read directly out of the literature on linguistic semantics. Linguists have spent many decades building out knowledge of how to handle more sophisticated uses of quantification [79, 118] ("every," "some," "most"), resolving pronoun references [54], discontinuity [84] (where a word is far from a word it modifies), and much more [20, 85]. Critically, because categorial grammars are lexicalized, *most grammatical constructions require no special handling* given an appropriate base categorial grammar.

Our experience thus far has borne out this claim of modularity. As we have developed our prototype, we have only needed to modify or extend the core grammatical types of Figure 3 for two reasons:

Prepositional Phrases. We added a category for prepositional phrases, indexed by the variety of English preposition (*of*, *into*, etc.). The set of preposition indices is incomplete, but with the exception of constructions like "*of naturals*" (which implies a need for a common noun, as opposed to "*of 3*"), these simply take a noun phrase to their right and have the identity function as semantics (i.e., "*of 3*" simply denotes 3). This is the standard treatment of prepositional phrases in compositional linguistic semantics.

Anaphora / References. Some specifications will use indirect references, called anaphora — pronouns ("it"), articles ("the"), and other words that have no self-contained meaning but instead refer to concepts used earlier in a sentence. We have prototyped a refinement of Jacobson-style [54] treatment of anaphora (references to things mentioned earlier in the sentence, such as pronouns or some uses of "*the*"); this involves the addition of another slash type $A \mid B$ for expressions of category A if some kind of missing B (i.e., what is being referred to) is resolved, and isolated the additional rules (which increase the size of the search space) in a separate module (so those rules are not always on), including a variant for named variable references. While there exist many categorial grammar solutions to the problem of anaphora [42, 54, 79, 84], all of them rely on such an additional construction for sentence fragments with missing referents. Jacobson's approach is the basis for most other treatments; unfortunately we are not aware of extensions of the normal form search pruning we use [35] to this feature, and consider such optimization future work.

Beyond these changes, which are backwards-compatible with the exposition in Section 3, we have added additional logical rules, such as lifting adjectives into noun modifiers (e.g.,. "*even*" can be lifted for use as a modifier of "*natural*"). Because these are presented as `Synth` instances are modular additions to the core (they rely on some rules which slightly increase parsing time, so are in an optional module that need not be imported unless pronouns are used). Beyond that, all extensions are simply lexicon entries, notably all quantifiers, including uses of "*a*" and "*any*" embedded mid-sentence.

4.3 Beyond CIC

While our framing so far has focused on generating specifications which in Lean have type `Prop`, this is not required. Categorial grammars require only that their top-level semantic truth type have the structure of a Heyting Algebra [69]: a type with binary operators for standard logical operators.

Our Lean formalizations in fact makes this generalization: the core machinery is polymorphic over an arbitrary choice of Heyting Algebra, with a lexicon split between entries polymorphic over the Heyting Algebra being targeted (e.g., "or" and "and") and words specific to a given Heyting Algebra (e.g., an adjective given as a Lean predicate must target Lean's `Prop`).

This means the core idea applies not only to specs of type Prop, but that this machinery can be readily retargeted to any logic formalized within the proof assistant, such as LTL [97] or CTL [29]. This is not itself novel (Section 7 discusses some prior approaches to this) but working directly within a formalized proof assistant brings accuracy benefits to such

[11] Which does not exist in English, but does in German, French, and other languages

efforts. A partial formalization of Dzifcak et al.'s work [34] in our original Coq prototype [44] revealed that they made use of invalid lexicon entries: in this paper's notation, entries for words with grammatical category C whose specified semantics were not of type `interp C`.

5 Trust and Auditing

One of the essential criteria for an LCF-style proof assistant is the production of an independently-checkable proof certificate [98]. While we have proposed using typeclass machinery to automatically parse and denote, and the typeclass resolution itself is typically not viewed as part of the trusted computing base (TCB), it does effectively produce a form of proof certificate. The typeclass machinery explicitly constructs an instance of the typeclass — an element of the corresponding record type — and passes it to `pspec`. So the proof assistant's kernel sees (effectively) a categorial grammar proof, constructed via typeclass instances rather than constructors of an inductive data type (with `Synth` instance names in lieu of constructors). This explicit term persists into the compiled forms Lean already produces, and could be identified by an independent proof checker that wished to also validate the natural language interpretation.

We can think of several ways a user might accidentally or maliciously risk confusing an independent checker. All but one can easily be detected by a checker aware of the categorial grammar specification typeclasses. The final possibility amounts to changing the specification in the proof certificate.

First, a user may redefine or extend our core instances (for `Synth`) to produce a different denotation. A certificate checker would already ensure these are type-correct. A natural-language-specification-aware extension could check that the `Synth` instances correspond to the desired rules. Or to better support some of the extensibility arguments made earlier, the `Synth` typeclass could be modified to also carry a justification of its conclusions in a more general substructural logic [42, 63], which would amount to requiring extensions to carry conservativity proofs over a trusted linguistic base system.

Second, a user may extend the lexicon with additional words or additional grammatical roles for a given word, introducing ambiguity into the parsing. Checking for ambiguity is relatively straightforward: setting aside indexing by CIC types, equivalence of grammatical types is decidable, and a checker could conservatively require that any lexicon entries with the same index-erased grammatical types have clearly-distinct indices (in which case they would not unify during typeclass search under any circumstances). An independent checker could verify the absence of ambiguity in the lexicon, or alternatively surface the use of any ambiguity in a parsing derivation for human inspection.

Finally, a user could also manipulate the lexicon, for example *redefining* (or *mis*defining) "monotone" to denote λf, True. This is arguably a form of modifying the specification by changing definitions, rather than sneaking a broken proof past a certificate checker. It is analagous to changing a definition of a property verified by a proof — a working proof with the wrong definition is wrong, but this leaves behind evidence of the incorrect definition, by leaving evidence of how "monotone" was interpreted. This would however require human intervention to detect.

These possible forms of attack highlight the main sources of trust added when considering natural language specifications in the approach we describe: the grammatical rules for combining phrases, well-formedness of the lexicon, and the definitions of words in the lexicon.

Beyond these, there is the general issue of *ambiguity* in parsing. Semantic parsing gives rise to two forms of ambiguity: *spurious* ambiguity (where there are multiple parsing derivations, but they yield equivalent semantics), and *true* ambiguity (where the different derivations yield truly different semantics). Spurious ambiguity is typically tackled by searching only for normal-form derivations [35], as we do. Actual ambiguity can arise from multiple lexical entries with the same grammatical types but different meanings as in the "attack" described above (for which we described mitigations), but can also sometimes arise from matters such as quantifier scoping: in "every child ate a pizza," was there a pizza for each child (the most common reading), or was there a single pizza shared by all (less common, but acceptable). There are two approaches to mitigating these. First, there is linguistic evidence for claiming that there truly is a most common resolution, and the grammar can be tailored to prefer that linguistically more common result. For example, there is evidence that in English, quantifiers earlier in a sentence (to the left) tend to outscope those to the right [13], as in the example above. Moreover, because of how quantifier nesting works formally, this is likely to be even further emphasized in English for formal specifications. Second, it is possible in general to perform an exhaustive search to identify ambiguity (then allowing a rewrite to remove the ambiguity); Coq's typeclasses include options to perform exhaustive search to ensure no ambiguity exists.

6 Case Study: Sorting and Multisets in VFA

Appel's *Verified Functional Algorithms* [6] is a textbook on verification of functional algorithms in Coq. To evaluate our prototype on non-hand-picked specifications, we translated formal lemma statements (but not proofs) of specifications from Chapters 2 and 3 of the book into Lean. These chapters deal with verification of insertion sort on lists of natural numbers, initially in terms of list properties (Chapter 2) and then in terms of an extensional view of lists as an ordered multiset (Chapter 3). Chapter 2 asks students to prove 5 specific lemmas about insertion sort and a helper function for insertion into a sorted list. (It also includes lemmas proving

Table 1. Translation experiments with *Verified Functional Algorithms* specifications.

#	Ch	Original (formal) and Generated (if different)
1	Sort	Original: "insertion maintains sortedness" (`forall a l, sorted l -> sorted (insert a l)`) Translated (6s): "insertion of any natural maintains sortedness" (α-equivalent)
2		Original: "insertion sort makes a list sorted" (`forall l, sorted (sort l)`) Translated (5.32s): "sort sorts any list of naturals" (α-equivalent)
3		[No Original English] (`forall x l, Permutation (x::l) (insert x l)`) Proposed & Translated (7.4s): "insert is a permutation of cons" (α-equivalent)
4		Original & Translated (1.84s): "sort is a permutation" (`forall l, Permutation l (sort l)`)
5		[No original English] (`forall l, sorted (a l) /\ Permutation l (a l)`) Proposed & Translated (5.72s): "sort is a sorting permuting algorithm" (`exists a, (forall l, sorted (a l)) /\ (forall l, Permutation l (a l)) /\ a=sort`)
6	Multiset	Original & Translated (1.27s): "union is associative" (`forall a b c, union a (union b c)=union (union a b) c`)
7		Original & Translated (1.24s): "union is commutative" (`forall a b, union a b = union b a`)
8		Original: "insert produces the same contents as merely prepending the inserted element to the front of the list" (`forall x l, contents (insert x l) = contents (x :: l)`) Translated*: "insertion and cons of any value yield equal contents" (α-equivalent)
9		Original & Translated (1.27s): "sort preserves contents" (`forall l, contents l = contents sort l`)
10		[No original English] (`forall l, contents l = contents (sort l) /\ sorted (sort l)`) Proposed & Translated (7.18s): "sort preserves contents and sorts" (α-equivalent)

equivalence of two definitions of a sorted predicate, but we omit these because internals of inductive definitions are not in the intended scope of the prototype.) 3 of these lemmas come with explicit English descriptions of the corresponding lemma statement, while 2 have no direct English correspondence given. Chapter 3 asks students to prove a range of specific lemmas about multisets, as well as additional specifications of insertion sort in terms of multisets (where Chapter 2 uses the notion of list permutations). We have translated 5 specifications from Chapter 2, and 5 specifications from Chapter 3, using our Lean implementation [45]. For each of these we describe the original text (if given), a proposed adjustment with justification (if given), the original formalization, and the result of parsing the English into Lean with our prototype after adding appropriate lexicon entries. In cases where we changed the original text, we explain why. The purpose of this case study is to put pressure on the system regarding (1) its ability to express precise formal claims chosen without this system in mind, (2) its ability to (approximately) relate textbook-level English prose to formal specifications, and (3) to surface some linguistic issues at play in formal specification. Table 1 summarizes our experiments.

In general, we reworded a number of examples which had explicit English translations in *VFA*, most commonly to be more explicit about quantification, as in the first example. Even in this small sample, English specifications tend to ellide details of what they quantify over. While it would be possible (due to our categories being indexed by CIC types) to maintain this ellision, we opted to keep the explicit quantification in part to minimize how much of English grammar our prototype required. Our prototype does not contain a full formalization of the English language (per the discussion of Section 2.2), but none of the specifications we consider is beyond grammatical formalization: consider CCGBank [53], which gives CCG categories and desired parses for 48,934 English sentences from the Wall Street Journal, and is the basis of CCG grammars that have been used to parse even more complex texts, such as all of *Alice in Wonderland* [136]. Formalizing a more comprehensive English grammar is a long-term undertaking. Our goal with this experiment was to explore both grammatical feasibility and implementation feasibility for smaller examples.

For Chapter 2, we have translated all five specifications from English into formal specifications that are logically equivalent (for 4/5, α-equivalent) to the hand-written specifications from the textbook. We also proposed what we feel are reasonable English equivalents to specifications that were formalized but not explicity described in English. Example 5 is illustrative: it is clearly logically equivalent to the original, but because the English uses the indefinite article "a" in a way that formal linguistic semantics typically treats as introducing an existential quantifier (e.g., as in "I have a duck"), the generated formal specification has an existential quantifier.

For Chapter 3, we have translated the specifications from the main portion of the chapter, omitting an extension treating an alternative proof approach, where some specifications are complex strengthened inductive steps which are typically not the sort one would specify in natural language.[12]

The one specification we do not translate from that main section is the specification described in English by "multisets in a nested union can be swapped." This text is both

[12] Consider `forall l x n, S n = contents l x -> exists l1 l2, l = l1 ++ x :: l2 /\ contents (l1 ++ l2) x = n`

linguistically interesting, and under-specified. It is ambiguous because, we believe, most readers would not understand what this meant from the text alone. (Consider if you can understand it before checking the formalization in a footnote.[13]) It is also linguistically interesting as it refers to the syntax used to write the property, rather than directly describing a property of the union operation. We could imagine a number of ways of making the exact English work in this case, or rewriting it slightly more generally without recourse to syntax (e.g., defining "un-nestable" as a lexical entry and handling it similarly to examples 6 and 7), but we believe the proper way to address it is with a more thorough study of the linguistics of how mathematical text refers to syntax — work which is a separate research agenda unto itself, building on what we have done here.

One specification, 8, can be formalized, and the parse structure is not particularly complex, but Lean curently does not synthesize it because at present it does not pull in the lexical entries for "yield" or "equal" during typeclass resolution, for unknown reasons; the lexical categories are not more complex than other entries, and we can explicitly provide the entries and obtain semantics α-equivalent to the original, but Lean fails to produce a `Synth` instance for either of those 1-word spans of the sentence at a grammatical category matching the lexicon entry declaration. We are actively investigating the cause of this.

Timing. We are limited in the precision of our measurements for timing, as Lean exposes no direct way to measure the search time for a particular typeclass resolution problem. The times reported in Table 1 are the time to run `lake build` (Lean's project build command) in the root of the project, after building the full project, then removing all intermediate and final build artifacts for the particular specification, and running `lake build` again to compile *only* that file. These time measurements then include process start-up time, time to parse the project file and identify the file with the missing build, and to parse, typecheck (including the typeclass search), and compile that Lean file. Measurements were taken on a 2020 1.4GHz MacBook Pro with 16 GB of RAM. This is enough to show the general range of times for parsing modest specifications; in a large proof development, we would not expect parsing of natural language style specifications for top-level specifications (thus likely avoiding the complexities of finding language for examples like that in footnote 12) to be a dominant cost.

Beyond Chapters 2 and 3. We have also looked at later chapters of *VFA* to anticipate other challenges, grammatical, semantic, and implementation-related, which must be resolved in the long term. The "advanced" portion of Chapter 3, which we did not formalize above, requires pronouns and textual variable names. We have implemented grammatical support for named variables in text based on classic grammatical treatments of pronoun binding, but extending the idea of normal form parsing to these models is unresolved in the computational linguistics literature. Multiple later chapters also shift from the monomorphic specifications of early chapters, to polymorphic specifications. These introduce interesting open technical challenges on the categorial grammar side. There, phrases such as "every tree" (as in binary search tree) is actually referring to *three* quantifications, not one: a quantification over key and value types, as well as a specific tree with those arbitrary key and value types. This poses challenges for the decoding function `interp`, as well as for the grammatical constraints across the rest of a sentence. Naïve extension of `interp` runs into universe size issues: quantification of a type in `Type n` lives in `Type (n+1)`. So straightforward approaches to extending `interp` run into problems where different cases should be returning types in different levels of the universe hierarchy. This does not occur in traditional linguistics where all entities are of a single type e, or even in prior work on type-theoretical semantics [24, 103], where all types are explicitly assumed to live in a single universe and polymorphism is not addressed (as the primary interest there generally remains modeling general linguistic issues, not mathematical issues). If that problem were resolved, there is then the problem that the indices of categories later in the sentence, which would want to refer to trees with particular key-value types, cannot be directly indexed by types quantified within the *denotation* of another word. Lean has some support for declaring typeclass instances with certain parameters held abstract, allowing `Synth` instances to have arguments of the form `[forall T, Synth ... ((@NP T) \\ S)]`, which could have the choice of `T` supplied by another entry when computing denotations.

7 Related Work

7.1 Categorial Grammars and Type Theory

Categorial grammars and dependent type theories for natural language semantics have long histories [68, 119, 126]. Our proposal differs from that work in our focus on building a system to describe dependent types directly in a system where they are required, as opposed to most prior work's focus on using dependent types for linguistic outcomes.

Others have used type theories like Lean's for linguistic semantics [15, 22, 102, 103, 119], broadly making the argument that variants of dependent type theory offer a range of appealing options for modeling natural language semantics, and fix some percieved deficiencies in the use of a lambda calculus over first-order logic formulas. This work consistently focused on using this as a means to study linguistics. The notion of indexing some grammatical categories by the type of a referent in such an underlying type theory comes

[13] It is formalized as `forall a b c, union a (union b c) = union b (union a c)`.

from Ranta's work [104] on studying the linguistics of mathematical statements, though this was focused purely on the study of mathematical language, and not on interfacing with mathematics work carried out in type theory. Ranta [103], Kokke [62] and Kiselyov [60] have formalized variants of categorial grammar with semantics in proof assistants, but only as object logics of study in order to prove properties of those systems, rather than as working parsers integrated with other uses of proof assistants. This leaves much to explore in integrating categorial grammar with various forms of type-theoretical language semantics [24], some of which coincide with common specification patterns.

On the linguistic side, our particular choice to represent common nouns as *typed predicates* appears to be new. Traditional linguistic semantics going back to Montague [76] assume a single universe of discourse represented by a type `e`, with nouns denoting predicates on `e` , so the noun *even natural* would denote a predicate (`λx, even x ∧ natural x`)). Traditional type-theoretic semantics treats common nouns exclusively as types [23, 103] (an even natural there would be a subset type). Retoré [108] treats common nouns as *either* types or predicates on a single universe of discourse, depending on the circumstance. We treat them as both *simultaneously*, allowing them to play a role in unification during parsing while also contributing further refining behaviors. We believe that this is likely to work better for a wide variety of formalizations, as subset types remain relatively less-frequently-used in proof assistants; we leave validating this conjecture to future work.

7.2 Natural Language for (Semi)Formal Specifications

We are hardly the first to argue for narrowing the gap between natural language and formal specifications, Nor are we the first to attempt this via formal grammars that also model semantics.

Seki et al. [112, 113] is the earliest approach we are aware of, using an alternative lexicalized grammar formalism (HPSG [100]) to translate natural langauge to first-order logic. Their prototype was divorced from any particular use of the resulting formal specifications.

Dzifcak et al. [34] used CCGs to translate natural language specifications to CTL*, though as mentioned earlier their semantics contain semantic type errors which are caught by working within a proof assistant that *enforces* consistency between grammatical and semantic types (i.e., caught by the dependent type of `Synth.denotation`). They also translate to PDL, reusing some entries by not type-checking their semantics, enabling the mistakes above.

One recurring theme in formalized natural language interfaces to logics is intentionally (by design) reflecting aspects of the syntax of a target logical language back into their handling of natural language syntax, in a way that superficially *appears* to be consistent with the natural language, but in fact leads to subtly incorrect semantics. For example, PENG [111] and its derivatives reflect the operator precedence of first-order logic into its interpretation of English, which is wrong, leading to misinterpreting phrases like "not yellow or blue" as in "the signal is not yellow or blue" as $(\neg \textit{yellow}) \vee \textit{blue}$ because negation binds more tightly than disjunction *even in PENG's English surface syntax*, when most English speakers would interpret the phrase as $\neg(\text{yellow} \vee \textit{blue})$. Focusing on actual linguistic treatments of syntax and semantics as we do leaves such phrases ambiguous; while we have not implemented ambiguity detection, ambiguities cannot be caught if they are intentionally not modeled. While we have set aside our Coq prototype [44], Coq's typeclass implementation has a flag `Typeclasses Unique Solutions` which verifies there is exactly one instance (e.g., of `Synth`) solving a problem, and fails otherwise; such a feature could be added to Lean and would then implement ambiguity detection for our encoding (in addition to its other uses for typeclasses in general).[14] PENG also restricts use of adjectives to a single adjective immediately before a noun, while our grammar is more general without much active effort (beyond choosing an established, known-generalizable grammatical and semantic basis): the rule lifting an adjective to a common noun modifier (CN/CN) applies, and then handling of phrases like "positive even prime natural" simply fall out of /-Comp and /-Elim.

The most successful and long-lived prior effort in this space (in active development for over 20 years), by most metrics, is *Attempto Controlled English* (ACE) [40], which is also emblematic of the philosophy behind most controlled natural language [39, 65]. ACE is a highly-regimented formal fragment of English [122] which aims to be an *editorialized* version of English grammar meant specifically for first-order logic specifications. Its primary implementation uses unification-based parsing, as in our work and most categorial grammar work, via Prolog's definite clause grammars [95, 96]. ACE is widely credited as one of the earliest attempts to specify software in English [40, 41], by translating specifications into Prolog clauses, which can then be used as a knowledge base for queries. This kind of controlled natural langue interface is a classic application of Prolog, predating ACE [129]. However, this results in a system that is disconnected from toolsets that people have ended up commonly using to specify software implementations: using an ACE-translated specification in another tool would require additional engineering work to export the Prolog representation of the specification to a form used by another tool (e.g., Z3). While not a fundamental issue, to the best of our knowledge, this engineering work has never occurred. ACE and similar systems also attempt to reduce the burden of growing a lexicon by treating most words as essentially uninterpreted functions or relations, rather than tying into existing formal definitions.

[14] This relies on our use of normalized parse trees.

A significant philosophical difference between most controled natural language systems and our goals is the designer expectations for how restrictive the grammar of a system might be. ACE, PENG, and other systems generally assume the set of grammatical categories is fixed at system design time, forever. Per our discussion in Section 4.1, as most lexical entries have obvious basic categories such as nouns and adjectives, adding additional lexical entries in those categories, as in our work, is straightforward. However, adding new grammatical constructions in these earlier frameworks requires adding new non-terminals to their grammars. Because most linguistics research of the past several decades does not use standard phrase structure grammars (CFGs), this leaves such extensions in a difficult position: implementors must either roll their own extensions in a way that has never been validated by linguists, or are stuck with grammatical treatments that are decades out of date. In contrast, our experience in Section 4.2 shows that the linguistics literature on categorial grammars offers modular grammatical extensions to new features which have been extensively-vetted by linguists, and can often be directly transcribed from the linguistics literature. This is in addition to the fact that in many cases these extensions do not require modifying core rules, but instead follow from additions of individual lexicon entries (because categorial grammars are lexicalized).

We believe there are two primary reasons that the work above has not lead to widely-used controlled natural language specifications, both of which we avoid. The first is that prior implementations have been designed to prioritize parsing and translation over use of the resulting specifications; most prior work produces Prolog representations of specifications. This is primarily an engineering problem, but of the sort that frequently hampers adoption of ideas. Our implementation is a library for an actively developed proof assistant that is actively used for specification and verification of mathematical claims about both software and mathematics, using features that already exist for purposes unrelated to natural language interfaces.

The second is that prior work made what we believe are suboptimal choices regarding which influences from linguistics to incorporate. These include conscious choices like PENG's aforementioned reflection of FOL operator precedence into English (similar confusion between natural English interpretation and how words are repurposed for specific logics also lead to confusions with LTL [47]). It also includes cases like Vadera and Meziane's work [125], where not only were similar unnatural heuristics reflected back into English (so users had to remember, and be able to consistently apply, the implementation's heuristics for quantification in order to understand specifications), but time has since revealed the work they were built on to be far less complete treatments of the linguistic phenomena than were claimed (in good faith) at the time. Vadera and Meziane adopted Hess's [50] approach to resolving ambiguities in quantifier scoping (an approach motivated by dismissing as mathematically impossible approaches that not only worked mathematically, but are now accepted as standard, thoroughly empirically validated treatments [120, 121]). These types of issues are why we have taken pains to work with an approach to controlled natural language that borrows as heavily as possible from widely-vetted linguistic theories: while it is relatively straightforward (though significant work) to choose a syntactic subset of a natural language, without specific efforts to draw on linguistic theories of meaning with significant empirical validation (and in particular validation of combined syntax and semantics) it is easy to accidentally (or intentionally) formalize a controlled natural language in such a way that the natural and formal interpretations of text silently diverge in critical ways, or unnecessarily restrict the language to avoid those issues.

This is the key limitation of what may be the most closely-related work to what we propose here: two systems that focus on controlled natural language in proof assistants. The earlier is work on GF-Alfa [48], which used Ranta's Grammatical Framework (GF) [105] to add translation from English to a proof assistant based on Martin-Löf type theory. GF is a toolkit for building bidirectional relationships between logical form and text: for both parsing, and generation (discussed in the next subsection). The paper describes the implementation of writing natural language specifications, but does not discuss actual use of the feature. The other system is current, developed contemporaneously with our earlier Coq prototype: Naproche [33] is an integration into the Isabelle proof assistant of the ForTheL controlled natural language [92]. ForTheL is a substantial template-based controlled natural language. While significant amounts of undergraduate algebra and set theory have been specified in ForTheL,[15] the language is centered around describing new definitions (which then act as noun phrases), and *notions*, which are predicates usable as common nouns or adjectives. It is not possible to add additional verbs, so half of our case study sentences (1, 2, 8, 9, and 10) cannot be specified in ForTheL without significant rewriting to avoid non-standard verbs; this applies to many other VFA specifications as well. In any case, because both systems use ad hoc grammars that are not based on established linguistic theories, extensions could easily go awry by over-generating or misinterpreting. This is slightly less likely with ForTheL specifications than with GF-based specifications: ForTheL's grammar is quite restrictive, while GF grammar definitions encourage regular expression style descriptions of arbitrary text fragments, which can easily lead to unexpected parses. These systems, unlike ours, do permit associating definitions and proofs with controlled English text (both involve changes to the proof assistant interfaces specifically for controlled natural

[15] https://github.com/naproche/FLib

language interfaces), with the same caveats about grammatical expressivity. We suspect we could also support definitions by additionally using Lean's macro facilities (which remain usable by libraries, without modifying Lean) to register the definitions, but believe this is actually counter-productive for type-theory-based systems, as the exact form of an inductive type and computational behavior of function definitions can have major impacts on the difficulty of a proof; as earlier, we recommend instead proving relationships between manual formal definitions and descriptions translated from English. Neither of these systems produce anything akin to a proof certificate recording how English text was translated.

Our use of categorial grammar is in some ways also a hedge against the possibility that the relevant linguistic theories are further revised or refined; categorial grammars are naturally open-ended and modular, allowing further refinements to be overridden. Picking a basis with extensive linguistic validation also opens the door, in the future, to exploiting linguistically-grounded practices in grammar engineering [16, 106] and *semantics-aware* grammar induction [66, 67] to ease growing the formalized coverage of the natural language over time — a path not available to attempts not rooted in extensive linguistic coverage.

Seki et al.'s early work using HPSG [112, 113], turns out to be prescient: HPSG is also a lexicalized grammar formalism, and is now roughly as thoroughly investigated as categorial grammars [86]; it would be reasonable to pursue our agenda based on HPSG rather than categorial grammar, though we believe categorial grammar's connections to substructural logic make it a slightly better basis for proof assistant users. However, at the time, HPSG was nascent: it was first proposed only 3 years before Seki's initial attempt [113], and only widely publicized in the year prior to Seki's work [99]. So at the time, HPSG did not offer the extensive library of formal treatments of linguistic phenomena that are now available for both categorial grammars and HPSG.

More recent closely related is work on using natural language to describe *tests*. Most recently, in parallel with this work, Gordon [43] proposed using categorial grammars to generate property-based tests [25] from English, using off-the-shelf CCG tools to generate an abstraction of JavaScript tests. While using CCGs for parsing in principle makes Gordon's approach extensible like ours, that work's lexicon is sufficiently limited that the English accepted is fairly formulaic and template-like. Harris and Harris [49] used a variant of context-free grammars to generate hardware tests expressed in CTL, realizing an idea first proposed (but never evaluated) by Nelken and Francez [88].

From Formal Specifications to Natural Language. More distantly related is work translating the reverse direction from formal specifications to English [19, 56], and work on expressing proofs themselves in (semi-)natural language [31, 130], though these all emphasize highly-restricted fragments of natural language, rather than using foundations that capture language structure more generally. In general, categorial grammars can be used for generation of text from a logical representation, intuitively by running the parsing unification process "in reverse," searching for a parse tree whose semantics are equivalent to the logical form being described [59]. This approach has the advantage of being able to use the same grammar (combination rules and lexicon) that are used for parsing, so in principle a grammar grown for parsing as we have proposed in this paper can be repurposed. This has been seriously-explored for CCGs in particular [87, 131–133]. But in practice, this approach to text generation is more difficult than semantic parsing, because the search space (all derivations whose semantics match a target) is much larger, and these approaches do not currently handle quantifiers, which are essential for formal specifications. Additional challenges arise in taking human-written formal specifications as input, as the grammar rules yield unreduced function applications (so at least $\beta\eta$-equivalence would need to be considered), and some semantic treatments related to quantifiers would require considering full logical equivalence in the target logic (consider the existential in Sentence 7 of Table 1).

Existing tools for translating from formal specifications to English avoid the difficulties just highlighted by either restricting the specifications they handle (e.g., specification languages without quantifiers [19, 56]), or using more ad hoc treatments of grammar (as in Ranta's GF, which essentially translates first-order logic to English via a recursive function from formula syntax trees to strings, filling in a template for each node type [107]). GFT was previously used in the aforementioned GF-Alfa system [48] to translate formal specifications into English. The case study reported there suggests that the workflow involved heavy specification-specific customization of the grammar, in a way supported by user interface extensions in GF-Alfa, but seemingly requiring significant additional manual effort for each specification translated to English.

7.3 Controlled Natural Language Beyond Specifications

Also related is the long history of work on *programming* in (controlled) natural language, including both program text itself [10, 30] and interactions with IDEs [101] manipulating programs written in standard programming languages (e.g., Java). Most prominently, COBOL was intended to make code "readable by managers or other non-programmers" [110]. Most attempts at programming languages in a natural language style were targeted at non-technical or less-technical users, and at specific domains (such as Applescript [30]), even if the targeted language was actually quite general (e.g., COBOL [110]). None of these systems were originally intended to have user-extensible grammars, though some, including COBOL, were later supplemented with token-based macro preprocessors [128], which have modularity problems

now well-known from experience with the C preprocessor. Some of these, notably COBOL and AppleScript, have seen significant use for extended periods of time by many users in their target domains. Attempts at fully-general natural language programming [10] have been less successful, largely because they lead to cases where the natural language is interpreted unexpectedly [17, 74], *and there is no way within the system to correct this misinterpretation.* In particular, we can find no prior system for general-purpose natural language programming which includes a fall-back to a more traditional (and less ambiguous) programming language (most efforts were aimed at non-programmers). Our proposal avoids this problem: we are proposing to *supplement* a deeply formal system for specification and proof with controlled natural language, not supplant the formal approaches. The intended users of our approach are experts in formal specification who wish to record the relationship between (controlled) natural language and formal specifications. Incompleteness or unexpected behaviors of our approach (which we have already argued can be modularly adjusted in general) will never prevent a Lean user from completing their proof, but only prevent them from recording the formal-natural relationship in a rigorous way.

7.4 Autoformalization

What we do in this paper falls under a broad interpretation of what is now known as *autoformalization* in the machine learning community. Current techniques in this space have promising results, with one system [135] successfully formalizing *and proving* (in Isabelle) 35.2% of a collection of English Math Olympiad-style problems. While still impressive (there is no explicit lexicon), this statistic ellides important limitations, including those fundamental to large language models (mentioned earlier), and specifics of the datasets used (which extended to pre-university math problems, but also includes significant primary-school-level math). Most critically, machine learning approaches currently used for this are built on techniques which consistently struggle to correctly handle even slightly-nontrivial boolean reasoning [36, 90, 123, 124] (let alone more complex logical operations like universal quantification [8]).

8 Looking Forward

We have presented evidence that it is plausible to support natural language specifications in current proof assistants by exploiting existing typeclass machinery, with no additional tooling required. Carried further, this could be useful in many ways. It can reduce the gap between informal and formal specifications, reducing (though not eliminating) trust in the manual formalization of requirements. Potentially non-experts in verification could understand some theorem statements, gaining confidence that a verification result matched their understanding of desired properties. And this could be used in educational contexts to help students learn or check informal-to-formal translations.

Of course, the details matter as well, and it will take time to realize a prototype that is broadly useful. First and foremost, a rich lexicon is required. As explained earlier, at least the initial lexicon will need to be manually constructed (borrowing grammatical categories from existing lexicons [1, 53], and filling in the semantics) before it would be fruitful to adapt techniques for learning lexicons [7, 58, 66] to extend the manually-crafted base. Guiding this effort would require a substantial collection of examples of natural-language descriptions of formal claims, both for prioritizing lexicon growth and for validation that the approach is growing to encompass real direct descriptions of claims.

Our performance results, while modest in scope, are informative, highlighting that many specifications are likely to be parseable in times that are tolerable in interactive settings. External implementations of parsers could potentially be used to speed up parsing, and they could emit proof certificates in the form of explicit construction of our `Synth` instances. However this requires additional layers of separate tooling from the typeclass-based approach explored here, and also risks problems with incompatibilities between tools and specific proof assistants; working within the proof assistant guarantees all generated semantics are well-typed (even in the presence of features like implicit arguments and rich macro support in defining semantic terms), a problem which has arisen in other settings (as in our discovery that some of Dzifcak's lexical entries have type-incompatible semantics [34], or as in Gordon's report [43] that NLTK's combination rules sometimes introduce type errors into semantics).

It is possible that small differences will be required between standard natural language grammars and those used by this approach, arising from distinctions important to proof assistants but irrelevant to colloquial language. This is already the case, as mentioned, with the indexing of some grammatical categories with the semantic types of referents, following Ranta's early work on formalizing mathematical prose [104]. This direction offers opportunities to collaborate with linguists working in syntax and compositional semantics [12, 55, 118]. Such collaborations could both help with possible novel linguistic features of "semi-formal" natural language, and offers a setting for applying classical linguistic techniques in a domain where they provide unique value.

A great deal of work lies ahead, but the potential benefits seem to more than justify further exploration in this direction.

Acknowledgments

This work was supported in part by the US National Science Foundation under Grant No.: CCF-2220991 (https://www.nsf.gov/awardsearch/showAward?AWD_ID=2220991).

References

[1] Lasha Abzianidze, Johannes Bjerva, Kilian Evang, Hessel Haagsma, Rik van Noord, Pierre Ludmann, Duc-Duy Nguyen, and Johan Bos. 2017. The Parallel Meaning Bank: Towards a Multilingual Corpus of Translations Annotated with Compositional Meaning Representations. In *EACL*.

[2] Kazimierz Ajdukiewicz. 1935. Die syntaktische konnexität. Studia Philosophica, 1: 1–27. Reprinted in Storrs McCall, ed., Polish Logic 1920–1939, 207–231.

[3] Hiyan Alshawi. 1992. *The core language engine*. MIT press.

[4] Bharat Ram Ambati, Tejaswini Deoskar, and Mark Steedman. 2018. Hindi CCGbank: A CCG treebank from the Hindi dependency treebank. *Language Resources and Evaluation* 52, 1 (2018), 67–100.

[5] Andrew W Appel. 2001. Foundational proof-carrying code. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS 2001)*. IEEE, 247–256.

[6] Andrew W. Appel. 2022. *Verified Functional Algorithms*. Software Foundations, Vol. 3.

[7] Yoav Artzi. 2016. Cornell SPF: Cornell Semantic Parsing Framework. arXiv:arXiv:1311.3011

[8] Nicholas Asher, Swarnadeep Bhar, Akshay Chaturvedi, Julie Hunter, and Soumya Paul. 2023. Limits for learning with language models. In *Proceedings of the 12th Joint Conference on Lexical and Computational Semantics (*SEM 2023)*. Association for Computational Linguistics, Toronto, Canada, 236–248. https://doi.org/10.18653/v1/2023.starsem-1.22

[9] Jason Baldridge and Geert-Jan M. Kruijff. 2003. Multi-modal Combinatory Categorial Grammar. In *Proceedings of the Tenth Conference on European Chapter of the Association for Computational Linguistics - Volume 1* (Budapest, Hungary) *(EACL '03)*. Association for Computational Linguistics, Stroudsburg, PA, USA, 211–218. https://doi.org/10.3115/1067807.1067836

[10] Bruce W. Ballard and Alan W. Biermann. 1979. Programming in Natural Language: "NLC" as a Prototype. In *Proceedings of the 1979 Annual Conference (ACM '79)*. Association for Computing Machinery, New York, NY, USA, 228–237. https://doi.org/10.1145/800177.810072

[11] Yehoshua Bar-Hillel. 1953. A quasi-arithmetical notation for syntactic description. *Language* 29, 1 (1953), 47–58.

[12] Chris Barker and Pauline Jacobson (Eds.). 2007. *Direct compositionality*. Oxford University Press.

[13] Chris Barker and Chung-chieh Shan. 2014. *Continuations and natural language*. Vol. 53. Oxford Studies in Theoretical.

[14] Daisuke Bekki. 2012. Dependent Type Semantics: An Introduction. In *Logic and Interactive Rationality (LIRA) Yearbook 2012, Volume 1*. 277–300.

[15] Daisuke Bekki. 2014. Representing Anaphora with Dependent Types. In *Logical Aspects of Computational Linguistics - 8th International Conference, LACL 2014, Toulouse, France, June 18-20, 2014. Proceedings*. 14–29. https://doi.org/10.1007/978-3-662-43742-1_2

[16] Emily M. Bender and Guy Emerson. 2021. Computational linguistics and grammar engineering. In *Head-Driven Phrase Structure Grammar: The Handbook [86]*.

[17] Alan W Biermann, Bruce W Ballard, and Anne H Sigmon. 1983. An experimental study of natural language programming. *International journal of man-machine studies* 18, 1 (1983), 71–87.

[18] Stephen A Boxwell and Chris Brew. 2010. A Pilot Arabic CCGbank.. In *LREC*.

[19] David A Burke and Kristofer Johannisson. 2005. Translating formal software specifications to natural language. In *International Conference on Logical Aspects of Computational Linguistics*. Springer, 51–66.

[20] Bob Carpenter. 1997. *Type-logical semantics*. MIT press.

[21] Bob Carpenter. 1999. The Turing-completeness of multimodal categorial grammars. *JFAK: Essays dedicated to Johan van Benthem on the occasion of his 50th birthday. Institute for Logic, Language, and Computation, University of Amsterdam. Available on CD-ROM at http://turing. wins. uva. nl* (1999).

[22] Stergios Chatzikyriakidis and Zhaohui Luo. 2014. Natural Language Inference in Coq. *Journal of Logic, Language, and Information* 23 (2014). Issue 4.

[23] Stergios Chatzikyriakidis and Zhaohui Luo. 2017. On the interpretation of common nouns: Types versus predicates. In *Modern perspectives in type-theoretical semantics*. Springer, 43–70.

[24] Stergios Chatzikyriakidis, Zhaohui Luo, et al. 2017. *Modern perspectives in type-theoretical semantics*. Vol. 98. Springer.

[25] Koen Claessen and John Hughes. 2011. QuickCheck: a lightweight tool for random testing of Haskell programs. In *In Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

[26] Stephen Clark and James R. Curran. 2003. Log-linear Models for Wide-coverage CCG Parsing. In *Proceedings of the 2003 Conference on Empirical Methods in Natural Language Processing (Conference on Empirical Methods on Natural Language Processing '03)*. Association for Computational Linguistics, Stroudsburg, PA, USA, 97–104.

[27] Stephen Clark and James R. Curran. 2007. Wide-Coverage Efficient Statistical Parsing with CCG and Log-Linear Models. *Computational Linguistics* 33, 4 (Dec. 2007), 493–552.

[28] Stephen Clark, Julia Hockenmaier, and Mark Steedman. 2002. Building Deep Dependency Structures with a Wide-coverage CCG Parser. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics* (Philadelphia, Pennsylvania) *(ACL '02)*. Association for Computational Linguistics, Stroudsburg, PA, USA, 327–334. https://doi.org/10.3115/1073083.1073138

[29] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. 1986. Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8, 2 (1986), 244–263.

[30] William R Cook. 2007. Applescript. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. 1–1.

[31] Marcos Cramer, Bernhard Fisseni, Peter Koepke, Daniel Kühlwein, Bernhard Schröder, and Jip Veldman. 2009. The naproche project controlled natural language proof checking of mathematical texts. In *International Workshop on Controlled Natural Language*. Springer, 170–186.

[32] Veronica Dahl. 1994. Natural language processing and logic programming. *The Journal of Logic Programming* 19 (1994), 681–714.

[33] Adrian De Lon, Peter Koepke, Anton Lorenzen, Adrian Marti, Marcel Schütz, and Makarius Wenzel. 2021. The Isabelle/Naproche natural language proof assistant. In *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*. Springer International Publishing, 614–624.

[34] Juraj Dzifcak, Matthias Scheutz, Chitta Baral, and Paul Schermerhorn. 2009. What to do and how to do it: Translating natural language directives into temporal and dynamic logic representation for goal management and action execution. In *2009 IEEE International Conference on Robotics and Automation*. IEEE, 4163–4168.

[35] Jason Eisner. 1996. Efficient Normal-Form Parsing for Combinatory Categorial Grammar. In *34th Annual Meeting of the Association for Computational Linguistics*. 79–86.

[36] Allyson Ettinger. 2020. What BERT is not: Lessons from a new suite of psycholinguistic diagnostics for language models. *Transactions of the Association for Computational Linguistics* 8 (2020), 34–48.

[37] Kate Finney. 1996. Mathematical notation in formal specification: Too difficult for the masses? *IEEE Transactions on Software Engineering* 22, 2 (1996), 158–159.

[38] Kate M Finney and Alex M Fedorec. 1996. An empirical study of specification readability. In *Teaching and Learning Formal Methods*.

Academic Press.
[39] Norbert E Fuchs. 2009. Controlled Natural Language. In *Workshop on Controlled Natural Language, CNL*. Springer.
[40] Norbert E Fuchs, Uta Schwertel, and Sunna Torge. 1999. Controlled natural language can replace first-order logic. In *14th IEEE International Conference on Automated Software Engineering*. IEEE, 295–298.
[41] Norbert E Fuchs and Rolf Schwitter. 1996. Attempto Controlled English (ACE). In *First International Workshop on Controlled Language Applications (CLAW)* (University of Leuven, Belgium). http://attempto.ifi.uzh.ch/site/pubs/papers/CLAW96.ps
[42] Jager Gerhard et al. 2005. *Anaphora and type logical grammar*. Vol. 24. Springer Science & Business Media.
[43] Colin S. Gordon. 2022. Towards Property-Based Tests in Natural Language. In *44th IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER)*. IEEE. https://doi.org/10.1145/3510455.3512781
[44] Colin S. Gordon and Sergey Matskevich. 2022. *Natural Language Specifications in Proof Assistants*. Technical Report arXiv cs.PL 2205.07811. Computing Research Repository (CoRR). https://doi.org/10.48550/arXiv.2205.07811 arXiv:2205.07811
[45] Colin S. Gordon and Sergey Matskevich. 2023. Artifact for Trustworthy Formal Natural Language Specifications. https://doi.org/10.5281/zenodo.8329080
[46] Mike Gordon. 2000. From LCF to HOL: a short history. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. 169–186.
[47] Ben Greenman, Sam Saarinen, Tim Nelson, and Shriram Krishnamurthi. 2022. Little Tricky Logic: Misconceptions in the Understanding of LTL. *The Art, Science, and Engineering of Programming* 7, 2 (2022).
[48] Thomas Hallgren and Aarne Ranta. 2000. An extensible proof text editor. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 70–84.
[49] Christopher B Harris and Ian G Harris. 2015. Generating formal hardware verification properties from Natural Language documentation. In *Proceedings of the 2015 IEEE 9th International Conference on Semantic Computing (IEEE ICSC 2015)*. IEEE, 49–56.
[50] Michael Hess. 1985. How Does Natural Language Quantify?. In *Second Conference of the European Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics, Geneva, Switzerland. https://aclanthology.org/E85-1002
[51] Julia Hockenmaier. 2006. Creating a CCGbank and a wide-coverage CCG lexicon for German. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 505–512.
[52] Julia Hockenmaier and Mark Steedman. 2005. *CCGbank: User's Manual*. Technical Report.
[53] Julia Hockenmaier and Mark Steedman. 2007. CCGbank: a corpus of CCG derivations and dependency structures extracted from the Penn Treebank. *Computational Linguistics* 33, 3 (2007), 355–396.
[54] Pauline Jacobson. 1999. Towards a variable-free semantics. *Linguistics and philosophy* 22, 2 (1999), 117–185.
[55] Pauline I Jacobson. 2014. *Compositional semantics: An introduction to the syntax/semantics interface*. Oxford University Press.
[56] Kristofer Johannisson. 2007. Natural language specifications. In *Verification of Object-Oriented Software. The KeY Approach*. Springer, 317–333.
[57] Aravind K. Joshi, David J. Weir, and K. Vijay-Shanker. 1990. *The Convergence of mildly context-sensitive grammar formalisms*. Technical Report MS-CIS-90-01. University of Pennsylvania (Philadelphia, PA US), Philadelphia. http://opac.inria.fr/record=b1042789
[58] Makoto Kanazawa. 1995. *Learnable classes of categorial grammars*. CSLI Publications, Stanford University.
[59] Martin Kay. 1996. Chart Generation. In *34th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Santa Cruz, California, USA, 200–204. https://doi.org/10.3115/981863.981890
[60] Oleg Kiselyov. 2015. Applicative abstract categorial grammars in full swing. In *JSAI International Symposium on Artificial Intelligence*. Springer, 66–78.
[61] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM Trans. Comput. Syst.* 32, 1, Article 2 (Feb. 2014), 70 pages. https://doi.org/10.1145/2560537
[62] Wen Kokke. 2015. Formalising type-logical grammar in Agda. In *1st Workshop on Type Theory and Lexical Semantics*.
[63] Geert-Jan M. Kruijff and Jason Baldridge. 2000. Relating categorial type logics and CCG through simulation. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.27.4152 Unpublished manuscript..
[64] Marco Kuhlmann, Alexander Koller, and Giorgio Satta. 2015. Lexicalization and Generative Power in CCG. *Computational Linguistics* 41, 2 (2015), 187–219.
[65] Tobias Kuhn. 2014. A survey and classification of controlled natural languages. *Computational linguistics* 40, 1 (2014), 121–170.
[66] Tom Kwiatkowski, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2011. Lexical Generalization in CCG Grammar Induction for Semantic Parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing* (Edinburgh, United Kingdom) *(Conference on Empirical Methods on Natural Language Processing '11)*. Association for Computational Linguistics, Stroudsburg, PA, USA, 1512–1523.
[67] Tom Kwiatkowski, Luke S. Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2010. Inducing Probabilistic CCG Grammars from Logical Form with Higher-Order Unification.. In *Conference on Empirical Methods on Natural Language Processing*. ACL, 1223–1233.
[68] Joachim Lambek. 1958. The mathematics of sentence structure. *The American Mathematical Monthly* 65, 3 (1958), 154–170.
[69] Joachim Lambek. 1988. Categorial and categorical grammars. In *Categorial grammars and natural language structures*. Springer, 297–317.
[70] Xavier Leroy. 2009. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446.
[71] Mitchell Marcus, Grace Kim, Mary Ann Marcinkiewicz, Robert MacIntyre, Ann Bies, Mark Ferguson, Karen Katz, and Britta Schasberger. 1994. The Penn Treebank: annotating predicate argument structure. In *Proceedings of the workshop on Human Language Technology*. Association for Computational Linguistics, 114–119.
[72] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a Large Annotated Corpus of English: The Penn Treebank. *Comput. Linguist.* 19, 2 (June 1993), 313–330. http://dl.acm.org/citation.cfm?id=972470.972475
[73] Per Martin-Löf and Giovanni Sambin. 1984. *Intuitionistic type theory*. Vol. 9. Bibliopolis Naples.
[74] Lance A Miller. 1981. Natural language programming: Styles, strategies, and contrasts. *IBM Systems Journal* 20, 2 (1981), 184–215.
[75] Koji Mineshima, Ribeka Tanaka, Pascual Martínez-Gómez, Yusuke Miyao, and Daisuke Bekki. 2016. Building compositional semantics and higher-order inference system for a wide-coverage Japanese CCG parser. In *EMNLP*.
[76] Richard Montague. 1970. English as a Formal Language. In *Linguaggi nella societa e nella tecnica*, Bruno Visentini (Ed.). Edizioni di Communita, 188–221.
[77] Richard Montague. 1970. Universal grammar. *Theoria* 36, 3 (1970), 373–398.
[78] Richard Montague. 1973. The proper treatment of quantification in ordinary English. In *Approaches to natural language*. Springer,

221–242.
[79] Michael Moortgat. 1996. Generalized quantifiers and discontinuous type constructors. In *Discontinuous Constituency*. NATURAL LANGUAGE PROCESSING, Vol. 6. Mouton de Gruyter, 181–208.
[80] Michael Moortgat. 1996. Multimodal linguistic inference. *Journal of Logic, Language and Information* 5, 3 (01 Oct 1996), 349–385. https://doi.org/10.1007/BF00159344
[81] Michael Moortgat. 1999. Constants of grammatical reasoning. In *Constraints and resources in natural language syntax and semantics*. 195–219.
[82] Richard Moot. 2015. A type-logical treebank for French. *Journal of Language Modelling* 3, 1 (2015), 229–264.
[83] Richard Moot and Christian Retoré. 2012. *The logic of categorial grammars: a deductive account of natural language syntax and semantics*. Vol. 6850. Springer.
[84] Glyn Morrill. 1995. Discontinuity in categorial grammar. *Linguistics and Philosophy* 18, 2 (1995), 175–219.
[85] Glyn V Morrill. 2012. *Type logical grammar: Categorial logic of signs*. Springer Science & Business Media.
[86] Stefan Müller, Anne Abeillé, Robert D Borsley, and Jean-Pierre Koenig. 2021. *Head-Driven Phrase Structure Grammar: The Handbook*. Language Science Press.
[87] Crystal Nakatsu and Michael White. 2010. Generating with Discourse Combinatory Categorial Grammar. *Linguistic Issues in Language Technology* 4 (Sep. 2010). https://doi.org/10.33011/lilt.v4i.1221
[88] Rani Nelken and Nissim Francez. 1996. Automatic translation of natural language system specifications into temporal logic. In *International Conference on Computer Aided Verification*. Springer, 360–371.
[89] Richard T. Oehrle, Emmon Bach, and Deirdre Wheeler. 1988. *Categorial Grammars and Natural Language Structures*. Springer.
[90] Lalchand Pandia and Allyson Ettinger. 2021. Sorting through the noise: Testing robustness of information processing in pre-trained language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 1583–1596. https://aclanthology.org/2021.emnlp-main.119
[91] Barbara H Partee and Herman LW Hendriks. 1997. Montague grammar. In *Handbook of logic and language*. Elsevier, 5–91.
[92] Andrei Paskevich. 2007. The syntax and semantics of the ForTheL language. http://nevidal.org/download/forthel.pdf
[93] Christine Paulin-Mohring. 1993. Inductive definitions in the system coq rules and properties. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 328–345.
[94] Lawrence C Paulson. 1990. *Logic and computation: interactive proof with Cambridge LCF*. Vol. 2. Cambridge University Press.
[95] Fernando CN Pereira and Stuart M Shieber. 1987. PROLOG and Natural Language Analysis.
[96] Fernando CN Pereira and David HD Warren. 1980. Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial intelligence* 13, 3 (1980), 231–278.
[97] Amir Pnueli. 1977. The Temporal Logic of Programs. In *FOCS*. IEEE.
[98] Robert Pollack. 1998. How to believe a machine-checked proof. In *Twenty Five Years of Constructive Type Theory*. Oxford University Press, 205–220.
[99] Carl Pollard and Ivan A Sag. 1987. *Information-based syntax and semantics: Vol. 1: fundamentals*. Center for the Study of Language and Information.
[100] Carl Pollard and Ivan A Sag. 1994. *Head-driven phrase structure grammar*. University of Chicago Press.
[101] David Price, Ellen Rilofff, Joseph Zachary, and Brandon Harvey. 2000. NaturalJava: A Natural Language Interface for Programming in Java. In *Proceedings of the 5th International Conference on Intelligent User Interfaces* (New Orleans, Louisiana, USA) *(IUI '00)*. Association for Computing Machinery, New York, NY, USA, 207–211. https://doi.org/10.1145/325737.325845
[102] Aarne Ranta. 1991. Intuitionistic categorial grammar. *Linguistics and Philosophy* 14, 2 (1991), 203–239.
[103] Aarne Ranta. 1994. *Type-theoretical Grammar*. Oxford University Press, Inc., New York, NY, USA.
[104] Aarne Ranta. 1995. Context-relative syntactic categories and the formalization of mathematical text. In *International Workshop on Types for Proofs and Programs*. Springer, 231–248.
[105] Aarne Ranta. 2004. Grammatical framework. *Journal of Functional Programming* 14, 2 (2004), 145–189.
[106] Aarne Ranta. 2011. *Grammatical framework: Programming with multilingual grammars*. Vol. 173. CSLI Publications, Center for the Study of Language and Information Stanford.
[107] Aarne Ranta. 2011. Translating between language and logic: what is easy and what is difficult. In *Automated Deduction–CADE-23: 23rd International Conference on Automated Deduction, Wrocław, Poland, July 31-August 5, 2011. Proceedings 23*. Springer, 5–25.
[108] Christian Retoré. 2013. The montagovian generative lexicon: a type theoretical framework for natural language semantics. In *19th international conference on types for proofs and programs (TYPES 2013)*.
[109] Yves Schabes. 1990. *Mathematical and computational aspects of lexicalized grammars*. Ph. D. Dissertation. Copyright - Copyright UMI - Dissertations Publishing 1990; Last updated - 2015-08-28.
[110] Ben Schneiderman. 1985. The relationship between COBOL and computer science. *Annals of the History of Computing* 7, 4 (1985), 348–352.
[111] Rolf Schwitter. 2002. English as a formal specification language. In *Proceedings. 13th International Workshop on Database and Expert Systems Applications*. IEEE, 228–232.
[112] Hiroyuki Seki, Tadao Kasami, Eiji Nabika, and Takashi Matsumura. 1992. A method for translating natural language program specifications into algebraic specifications. *Systems and computers in Japan* 23, 11 (1992), 1–16.
[113] Hiroyuki Seki, Eiji Nabika, Takashi Matsumura, Yujii Sugiyama, Mamoru Fujii, Koji Torii, and Tadao Kasami. 1988. A processing system for programming specifications in a natural language. In *[1988] Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences. Volume II: Software track*, Vol. 2. IEEE, 754–763.
[114] Daniel Selsam, Sebastian Ullrich, and Leonardo de Moura. 2020. Tabled typeclass resolution. *arXiv preprint arXiv:2001.04301* (2020).
[115] Anders Søgaard. 2021. Explainable Natural Language Processing. *Synthesis Lectures on Human Language Technologies* 14, 3 (2021), 1–123.
[116] Matthieu Sozeau and Nicolas Oury. 2008. First-class type classes. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 278–293.
[117] Mark Steedman. 2001. *The Syntactic Process*. The MIT Press.
[118] Mark Steedman. 2012. *Taking scope: The natural semantics of quantifiers*. Mit Press.
[119] Göran Sundholm. 1986. Proof theory and meaning. In *Handbook of philosophical logic*. Springer, 471–506.
[120] A Szabolcsi. 1997. *Ways of Scope Taking*. Vol. 65. Springer Science & Business Media.
[121] Anna Szabolcsi. 2010. *Quantification*. Cambridge University Press.
[122] Attempto Controlled English Team. [n. d.]. ACE 6.7 Syntax Report. http://attempto.ifi.uzh.ch/site/docs/syntax_report.html
[123] Aaron Traylor, Roman Feiman, and Ellie Pavlick. 2021. AND does not mean OR: Using Formal Languages to Study Language Models' Representations. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics*.
[124] Thinh Hung Truong, Timothy Baldwin, Karin Verspoor, and Trevor Cohn. 2023. Language models are not naysayers: an analysis of

language models on negation benchmarks. In *Proceedings of the 12th Joint Conference on Lexical and Computational Semantics (*SEM 2023)*. Association for Computational Linguistics, Toronto, Canada, 101–114. https://doi.org/10.18653/v1/2023.starsem-1.10

[125] Sunil Vadera and Farid Meziane. 1994. From English to formal specifications. *Comput. J.* 37, 9 (1994), 753–763.

[126] Johan van Benthem. 1990. Categorial Grammar and Type Theory. *Journal of Philosophical Logic* 19, 2 (1990), 115–168. http://www.jstor.org/stable/30226424

[127] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[128] Dennis M Volpano and Hubert E Dunsmore. 1984. Empirical investigation of COBOL features. *Information Processing & Management* 20, 1-2 (1984), 277–291.

[129] David H.D. Warren and Fernando C.N. Pereira. 1982. An Efficient Easily Adaptable System for Interpreting Natural Language Queries. *American Journal of Computational Linguistics* 8, 3-4 (1982), 110–122. https://aclanthology.org/J82-3002

[130] Markus Wenzel. 1999. Isar—a generic interpretative approach to readable formal proof documents. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 167–183.

[131] Michael White. 2006. CCG Chart Realization from Disjunctive Inputs. In *Proceedings of the Fourth International Natural Language Generation Conference*. Association for Computational Linguistics, Sydney, Australia, 12–19. https://aclanthology.org/W06-1403

[132] Michael White and Jason Baldridge. 2003. Adapting Chart Realization to CCG. In *Proceedings of the 9th European Workshop on Natural Language Generation (ENLG-2003) at EACL 2003*. Association for Computational Linguistics, Budapest, Hungary. https://aclanthology.org/W03-2316

[133] Michael White, Rajakrishnan Rajkumar, and Scott Martin. 2007. Towards broad coverage surface realization with CCG. In *Proceedings of the Workshop on Using corpora for natural language generation*. Copenhagen, Denmark. https://aclanthology.org/2007.mtsummit-ucnlg.4

[134] Jeannette M Wing. 1990. A specifier's introduction to formal methods. *Computer* 23, 9 (1990), 8–22.

[135] Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. 2022. Autoformalization with large language models. *Advances in Neural Information Processing Systems* 35 (2022), 32353–32368.

[136] Richie Yeung and Dimitri Kartsaklis. 2021. A CCG-Based Version of the DisCoCat Framework. In *Proceedings of the 2021 Workshop on Semantic Spaces at the Intersection of NLP, Physics, and Cognitive Science (SemSpace)*. Association for Computational Linguistics, Groningen, The Netherlands, 20–31. https://aclanthology.org/2021.semspace-1.3

Received 2023-04-28; accepted 2023-08-11