The Grail Theorem Prover

Richard Moot

Chapter 9
The Grail theorem prover: Type theory for syntax and semantics

Richard Moot

Abstract Type-logical grammars use a foundation of logic and type theory to model natural language. These grammars have been particularly successful giving an account of several well-known phenomena on the syntax-semantics interface, such as quantifier scope and its interaction with other phenomena. This chapter gives a high-level description of a family of theorem provers designed for grammar development in a variety of modern type-logical grammars. We discuss automated theorem proving for type-logical grammars from the perspective of proof nets, a graph-theoretic way to represent (partial) proofs during proof search.

9.1 Introduction

This chapter describes a series of tools for developing and testing type-logical grammars. The Grail family of theorem provers have been designed to work with a variety of modern type-logical frameworks, including multimodal type-logical grammars (Moortgat, 2011), NL_{cl} (Barker and Shan, 2014), the Displacement calculus (Morrill et al, 2011) and hybrid type-logical grammars (Kubota and Levine, 2012).

The tools give a transparent way of implementing grammars and testing their consequences, providing a natural deduction proof in the specific type-logical grammar for each of the readings of a sentence. None of this replaces careful reflection by the grammar writer, of course, but in many cases, computational testing of hand-written grammars will reveal surprises, showing unintended consequences of our grammar and such unintended proofs (or unintended *absences* of proofs) help us improve the grammar. Computational tools also help us speed up grammar devel-

Richard Moot
CNRS, LaBRI, Bordeaux University, 351 cours de la Libération, 33405 Talence, France

LIRMM, Montpellier University, 161 rue Ada, 34095 Montpellier cedex 5, France e-mail: Richard.Moot@labri.fr

opment, for example by allowing us to compare several alternative solutions to a problem and investigate where they make different predictions.

This chapter describes the underlying formalism of the theorem provers, as it is visible during an interactive proof trace, and present the general strategy followed by the theorem provers. The presentation in this chapter is somewhat informal, referring the reader elsewhere for full proofs.

The rest of this chapter is structured as follows. Section 9.2 presents a general introduction to type-logical grammars and illustrates its basic concepts using the Lambek calculus, ending the section with some problems at the syntax-semantics interface for the Lambek calculus. Section 9.3 looks at recent developments in type-logical grammars and how they solve some of the problems at the syntax-semantics interface. Section 9.4 looks at two general frameworks for automated theorem proving for type-logical grammars, describing the internal representation of partial proofs and giving a high-level overview of the proof search mechanism.

9.2 Type-logical grammars

Type-logical grammars are a family of grammar formalisms built on a foundation of logic and type theory. Type-logical grammars originated when Lambek (1958) introduced his Syntactic Calculus (called the Lambek calculus, L, by later authors). Though Lambek built on the work of Ajdukiewicz (1935), Bar-Hillel (1953) and others, Lambek's main innovation was to cast the calculus as a logic, giving a sequent calculus and showing decidability by means of cut elimination. This combination of linguistic and computational applications has proved very influential.

In its general form, a type-logical grammar consists of following components:

1. a *logic*, which fulfils the role of "Universal Grammar" mainstream linguistics[1],
2. a *homomorphism* from this logic to intuitionistic (linear) logic, this mapping is the syntax-semantics interface, with intuitionistic linear logic — also called the Lambek-van Benthem calculus, LP (van Benthem, 1995) — fulfilling the role of "deep structure".
3. a *lexicon*, which is a mapping from words of natural language to sets of formulas in our logic,
4. a set of *goal formulas*, which specifies the formulas corresponding to different types of sentences in our grammar.[2]

[1] This is rather different from Montague's use of the term "Universal Grammar" (Montague, 1970). In Montague's sense, the different components of a type-logical grammar together would be an *instantiation* of Universal Grammar.

[2] Many authors use a single designated goal formula, typically *s*, as is standard in formal language theory. I prefer this slightly more general setup, since it allows us to distinguish between, for example, declarative sentences, imperatives, yes/no questions, *wh* questions, etc., both syntactically and semantically.

A sentence $w_1, \ldots, w_n$ is grammatical iff the statement $A_1, \ldots, A_n \vdash C$ is provable in our logic, for some $A_i \in lex(w_i)$ and for some goal formula C. In other words, we use the lexicon to map words to formulas and then ask the logic whether the resulting sequence of formulas is a theorem. Parsing in a type-logical grammar is quite literally a form of theorem proving, a very pure realisation of the slogan "parsing as deduction".

One of the attractive aspects of type-logical grammars is their simple and transparent syntax-semantics interface. Though there is a variety of logics used for the syntax of type-logical grammars (I will discuss the Lambek calculus in Section 9.2.1 and two generalisations of it in Sections 9.3.1 and 9.3.2), there is a large consensus over the syntax-semantics interface. Figure 9.1 gives a picture of the standard architecture of type-logical grammars.

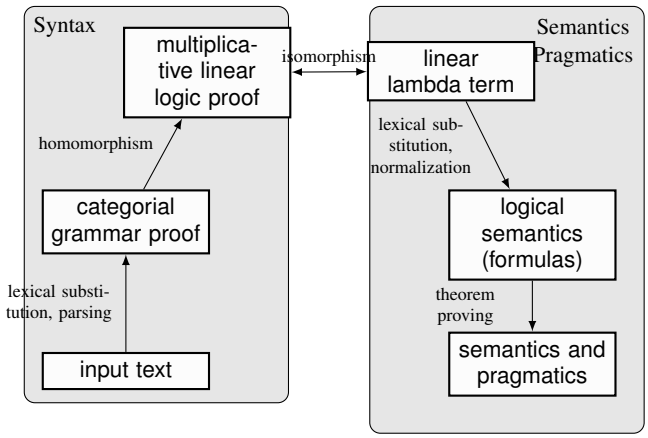


Fig. 9.1 The standard architecture of type-logical grammars

The "bridge" between syntax and semantics in the figure is the Curry-Howard isomorphism between linear lambda terms and proofs in multiplicative intuitionistic linear logic.

Theorem proving occurs in two places of the picture: first when parsing a sentence in a given type-logical grammar and also at the end when we use the resulting semantics for inferences. I will have little to say about this second type of theorem proving (Chatzikyriakidis, 2015; Mineshima et al, 2015, provide some investigations into this question, in a way which seems compatible with the syntax-semantics interface pursued here, though developing a full integrated system combining these systems with the current work would be an interesting research project); theorem proving for parsing will be discussed in Section 9.4.

The lexicon plays the role of translating words to syntactic formulas but also specifies the semantic term which is used to compute the semantics later. The lexicon of a categorial grammar is "semantically informed". The desired semantics of a

sentence allows us to reverse-engineer the formula and lexical lambda-term which produce it.

Many current semantic theories do not provide a semantic formula directly, but first provide a proto-semantics on which further computations are performed to produce the final semantics (eg. for anaphora resolution, presuppositions projection etc.). In the current context this means at least some inference is necessary to determine semantic and pragmatic wellformedness.

9.2.1 The Lambek calculus

To make things more concrete, I will start by presenting the Lambek calculus (Lambek, 1958). Lambek introduced his calculus as a way to "obtain an effective rule (or algorithm) for distinguishing sentences from nonsentences", which would be applicable both to formal and to (at least fragments of) natural languages (Lambek, 1958, p. 154). The simplest formulas used in the Lambek calculus are atomic formulas, which normally include s for sentence, n for common noun, np for noun phrase. We then inductively define the set of formulas of the Lambek calculus by saying that, they include the atomic formulas, and that, if A and B are formulas (atomic or not), then A/B, $A \bullet B$ and $B\backslash A$ are also formulas.

The intended meaning of a formula A/B — called A over B — is that it is looking for an expression of syntactic type B to its right to produce an expression of syntactic type A. An example would be a word like "the" which is assigned the formula np/n in the lexicon, indicating that it is looking for a common noun (like "student") to its right to form a noun phrase, meaning "the student" would be assigned syntactic type np. Similarly, the intended meaning of a formula $B\backslash A$ — called B under A — is that it is looking for an expression of syntactic type B to its left to produce an expression of type A. This means an intransitive verb like "slept", when assigned the formula $np\backslash s$ in the lexicon, combines with a noun phrase to its left to form a sentence s. We therefore predict that "the student slept" is a sentence, given the earlier assignment of np to "the student". Finally, a formula $A \bullet B$ denotes the concatenation of an expression of type A to an expression of type B.

$$\frac{\Delta \vdash A \bullet B \quad \Gamma, A, B, \Gamma' \vdash C}{\Gamma, \Delta, \Gamma' \vdash C}\ [\bullet E] \qquad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \bullet B}\ [\bullet I]$$

$$\frac{\Gamma \vdash A/B \quad \Delta \vdash B}{\Gamma, \Delta \vdash A}\ [/E] \qquad \frac{\Gamma, B \vdash A}{\Gamma \vdash A/B}\ [/I]$$

$$\frac{\Gamma \vdash B \quad \Delta \vdash B\backslash A}{\Gamma, \Delta \vdash A}\ [\backslash E] \qquad \frac{B, \Gamma \vdash A}{\Gamma \vdash B\backslash A}\ [\backslash I]$$

Table 9.1 Natural deduction for **L**

Basic statements of the Lambek calculus are of the form $A_1,\ldots,A_n \vdash C$ (with $n \geq 1$), indicating a claim that the sequence of formulas $A_1,\ldots,A_n$ is of type C; the sequent comma ',' is implicitly associative and non-commutative. Table 9.1 shows the natural deduction rules for the Lambek calculus. Γ, Δ, etc. denote non-empty sequences of formulas.

A simple Lambek calculus lexicon is shown in Table 9.2. I have adopted the standard convention in type-logical grammars of not using set notation for the lexicon, but instead listing multiple lexical entries for a word separately. This corresponds to treating *lex* as a non-deterministic function rather than as a set-valued function.

$lex(Alyssa) = np$	$lex(ran) = np\backslash s$
$lex(Emory) = np$	$lex(slept) = np\backslash s$
$lex(logic) = np$	$lex(loves) = (np\backslash s)/np$
$lex(the) = np/n$	$lex(aced) = (np\backslash s)/np$
$lex(difficult) = n/n$	$lex(passionately) = (np\backslash s)\backslash(np\backslash s)$
$lex(erratic) = n/n$	$lex(during) = ((np\backslash s)\backslash(np\backslash s))/np$
$lex(student) = n$	$lex(everyone) = s/(np\backslash s)$
$lex(exam) = n$	$lex(someone) = (s/np)\backslash s$
$lex(who) = (n\backslash n)/(np\backslash s)$	$lex(every) = (s/(np\backslash s))/n$
$lex(whom) = (n\backslash n)/(s/np)$	$lex(some) = ((s/np)\backslash s)/n$

Table 9.2 Lambek calculus lexicon

Proper names, such as "Alyssa" and "Emory" are assigned the category np. Common nouns, such as "student" and "exam" are assigned the category n. Adjectives, such as "difficult" or "erratic" are not assigned a basic syntactic category but rather the category n/n, indicating they are looking for a common noun to their right to form a new common noun, so we predict that both "difficult exam" and "exam" can be assigned category n. For more complex entries, "someone" is looking to its right for a verb phrase to produce a sentence, where $np\backslash s$ is the Lambek calculus equivalent of verb phrase, whereas "whom" is first looking to its right for a sentence which is itself missing a noun phrase to its right and then to its left for a noun.

Given the lexicon of Table 9.2, we can already derive some fairly complex sentences, such as the following, and, as we will see in the next section, obtain the correct semantics.

(1) Every student aced some exam.

(2) The student who slept during the exam loves Alyssa.

One of the two derivations of Sentence (1) is shown in Figure 9.2. To improve readability, the figure uses a "sugared" notation: instead of writing the lexical hypothesis corresponding to "exam" as $n \vdash n$, we have written it as $exam \vdash n$. The

withdrawn np's corresponding to the object and the subject are given a labels p_0 and q_0 respectively; the introduction rules are coindexed with the withdrawn hypotheses, even though this information can be inferred from the rule instantiation.

We can always uniquely reconstruct the antecedent from the labels. For example, the sugared statement "p_0 aced $q_0 \vdash s$" in the proof corresponds to $np, (np\backslash s)/np, np \vdash s$.

$$\dfrac{\dfrac{\text{every} \vdash (s/(np\backslash s))/n \quad \text{student} \vdash n}{\text{every student} \vdash s/(np\backslash s)}\ /E \qquad \dfrac{\dfrac{\dfrac{[p_0 \vdash np]^2 \qquad \dfrac{\text{aced} \vdash (np\backslash s)/np \quad [q_0 \vdash np]^1}{\text{aced } q_0 \vdash np\backslash s}\ /E}{p_0 \text{ aced } q_0 \vdash s}\ \backslash E}{p_0 \text{ aced} \vdash s/np}\ /I_1 \qquad \dfrac{\text{some} \vdash ((s/np)\backslash s)/n \quad \text{exam} \vdash n}{\text{some exam} \vdash (s/np)\backslash s}\ /E}{p_0 \text{ aced some exam} \vdash s}\ \backslash E \qquad \dfrac{p_0 \text{ aced some exam} \vdash s}{\text{aced some exam} \vdash np\backslash s}\ \backslash I_2}{\text{every student aced some exam} \vdash s}\ /E$$

Fig. 9.2 "Every student aced some exam" with the subject wide scope reading.

Although it is easy to verify that the proof of Figure 9.2 has correctly applied the rules of the Lambek calculus, finding such a proof from scratch may look a bit complicated (the key steps at the beginning of the proof involve introducing two np hypotheses and then deriving s/np to allow the object quantifier to take narrow scope). We will defer the question "given a statement $\Gamma \vdash C$, how do we decide whether or not it is derivable?" to Section 9.4 but will first discuss how this proof corresponds to the following logical formula.

$$\forall x.[\mathit{student}(x) \Rightarrow \exists y.[\mathit{exam}(y) \wedge \mathit{ace}(x,y)]]$$

9.2.2 The syntax-semantics interface

For the Lambek calculus, specifying the homomorphism to multiplicative intuitionistic linear logic is easy: we replace the two implications '$\backslash$' and '$/$' by the linear implication '$\multimap$' and the product '$\bullet$' by the tensor '$\otimes$'. In a statement $\Gamma \vdash C$, Γ is now a multiset of formulas instead of a sequence. In other words, the sequent comma ',' is now associative, commutative instead of associative, non-commutative. For the proof of Figure 9.2 of the previous section, this mapping gives the proof shown in Figure 9.3.

We have kept the order of the premisses of the rules as they were in Figure 9.2 to allow for an easier comparison. This deep structure still uses the same atomic formulas as the Lambek calculus, it just forgets about the order of the formulas and therefore can no longer distinguish between the leftward looking implication '$\backslash$' and the rightward looking implication '$/$'.

To obtain a semantics in the tradition of Montague (1974), we use the following mapping from syntactic types to semantic types, using Montague's atomic types e (for *entity*) and t (for *truth value*).

$$\dfrac{\dfrac{n \multimap ((np \multimap s) \multimap s) \quad n}{(np \multimap s) \multimap s}\ \multimap E \qquad \dfrac{\dfrac{[np]^2 \qquad \dfrac{np \multimap (np \multimap s) \quad [np]^1}{np \multimap s}\ \multimap E}{s}\ \multimap E}{np \multimap s}\ \multimap I_1 \qquad \dfrac{n \multimap ((np \multimap s) \multimap s) \quad n}{(np \multimap s) \multimap s}\ \multimap E}{\dfrac{s}{np \multimap s}\ \multimap I_2}\ \multimap E$$

Fig. 9.3 Deep structure of the derivation of Figure 9.2.

$$\begin{aligned} np^* &= e \\ n^* &= e \to t \\ s^* &= t \\ (A \multimap B)^* &= A^* \to B^* \end{aligned}$$

Applying this mapping to the deep structure proof of Figure 9.3 produces the intuitionistic proof and the corresponding (linear) lambda term as shown in Figure 9.4

$$\dfrac{\dfrac{z_0^{(e\to t)\to(e\to t)\to t} \quad z_1^{e\to t}}{(z_0\, z_1)^{(e\to t)\to t}}\ \to E \qquad \dfrac{\dfrac{\dfrac{[x^e]^2 \qquad \dfrac{z_2^{e\to(e\to t)} \quad [y^e]^1}{(z_2\, y)^{e\to t}}\ \to E}{((z_2\, y)\, x)^t}\ \to E}{\lambda y.((z_2\, y)\, x)^{e\to t}}\ \to I_1 \qquad \dfrac{z_3^{(e\to t)\to(e\to t)\to t} \quad z_4^{e\to t}}{(z_3\, z_4)^{(e\to t)\to t}}\ \to E}{((z_3\, z_4)\, \lambda y.((z_2\, y)\, x))^t}\ \to E}{\lambda x.((z_3\, z_4)\, \lambda y.((z_2\, y)\, x))^{e\to t}}\ \to I_2}{((z_0\, z_1)\, (\lambda x.((z_3\, z_4)\, \lambda y.((z_2\, y)\, x))))^t}\ \to E$$

Fig. 9.4 Intuitionistic proof and lambda term corresponding to the deep structure of Figure 9.3.

The computed term corresponds to the derivational semantics of the proof. To obtain the complete meaning, we need to substitute, for each of $z_0, \ldots, z_4$, the meaning assigned in the lexicon.

For example, "every" has syntactic type $(s/(np\backslash s))/n$ and its semantic type is $(e \to t) \to (e \to t) \to t$. The corresponding lexical lambda term of this type is $\lambda P^{e\to t}.\lambda Q^{e\to t}.(\forall(\lambda x^e.((\Rightarrow (Px))(Qx))))$, with '$\forall$' a constant of type $(e \to t) \to t$ and '$\Rightarrow$' a constant of type $t \to (t \to t)$. In the more familiar Montague formulation, this lexical term corresponds to $\lambda P^{e\to t}.\lambda Q^{e\to t}.\forall x.[(Px) \Rightarrow (Qx)]$, where we can see the formula in higher-order logic we are constructing more clearly. Although the derivational semantics is a linear lambda term, the lexical term assigned to "every" is not, since the variable x has two bound occurrences.

The formula assigned to "some" has the same semantic type but a different term $\lambda P^{e\to t}.\lambda Q^{e\to t}.(\exists(\lambda x^e.((\wedge(Px))(Qx))))$.

The other words are simple, "exam" is assigned $exam^{e\to t}$, "student" is assigned $student^{e\to t}$, and "aced" is assigned $ace^{e\to(e\to t)}$.

So to compute the meaning, we start with the derivational semantics, repeated below.

$$((z_0\, z_1)\, (\lambda x.((z_3\, z_4)\, \lambda y.((z_2\, y)\, x))))$$

Then we substitute the lexical meanings, for $z_0, \ldots, z_4$.

$$\begin{aligned}
z_0 &:= \lambda P^{e\to t}.\lambda Q^{e\to t}.(\forall(\lambda x^e.((\Rightarrow (Px))(Qx))))\\
z_1 &:= student^{e\to t}\\
z_2 &:= ace^{e\to(e\to t)}\\
z_3 &:= \lambda P^{e\to t}.\lambda Q^{e\to t}.(\exists(\lambda x^e.((\wedge(Px))(Qx))))\\
z_4 &:= exam^{e\to t}
\end{aligned}$$

This produces the following lambda term.

$$\begin{aligned}
&((\lambda P^{e\to t}.\lambda Q^{e\to t}.(\forall(\lambda x^e.((\Rightarrow (Px))(Qx))))\, student^{e\to t})\\
&(\lambda x.((\lambda P^{e\to t}.\lambda Q^{e\to t}.(\exists(\lambda x^e.((\wedge(Px))(Qx))))\, exam^{e\to t})\\
&\qquad \lambda y.((ace^{e\to(e\to t)}\, y)\, x))))
\end{aligned}$$

Finally, when we normalise this lambda term, we obtain the following semantics for this sentence.

$$(\forall(\lambda x^e.((\Rightarrow (student^{e\to t})\, x))(\exists(\lambda y^e.((\wedge(exam^{e\to t}\, y))(((ace^{e\to(e\to t)}\, y)\, x)))))$$

This lambda term represents the more readable higher-order logic formula[3].

$$\forall x.[student(x) \Rightarrow \exists y.[exam(y) \wedge ace(x,y)]]$$

Proofs in the Lambek calculus, and in type-logical grammars are subsets of the proofs in intuitionistic (linear) logic and these proofs are compatible with formal semantics in the tradition initiated by Montague (1974).

For the example in this section, we have calculated the semantics of a simple example in "slow motion": many authors assign a lambda term directly to a proof in their type-logical grammar, leaving the translation to intuitionistic linear logic implicit.

[3] We have used the standard convention in Montague grammar of writing $(p\,x)$ as $p(x)$ and $((p\,y)\,x)$ as $p(x,y)$, for a predicate symbol p.

Given a semantic analysis without a corresponding syntactic proof, we can try to reverse engineer the syntactic proof. For example, suppose we want to assign the reflexive "himself" the lambda term $\lambda R^{(e\to e\to t)}\lambda x^{e}.((Rx)x)$, that is, a term of type $(e\to e\to t)\to e\to t$. Then, using some syntactic reasoning to eliminate implausible candidates like $(np \multimap n) \multimap n$, the only reasonable deep structure formula is $(np \multimap np \multimap s) \multimap (np \multimap s)$ and, reasoning a bit further about which of the implications is left and right, we quickly end up with the quite reasonable (though far from perfect) Lambek calculus formula $((np\backslash s)/np)\backslash(np\backslash s)$.

9.2.3 Going further

Though the Lambek calculus is a beautiful and simple logic and though it gives a reasonable account of many interesting phenomena on the syntax-semantics interface, the Lambek calculus has a number of problems, which I will discuss briefly below. The driving force of research in type-logical grammars since the eighties has been to find solutions to these problems and some of these solutions will be the main theme of the next section.

Formal language theory

The Lambek calculus generates only context-free languages (Pentus, 1997). There is a rather large consensus that natural languages are best described by a class of languages at least slightly larger than the context-free languages. Classical examples of phenomena better analysed using so-called mildly context-sensitive language include verb clusters in Dutch and in Swiss German (Huijbregts, 1984; Shieber, 1985).

The Syntax-Semantics Interface

Though our example grammar correctly predicted two readings for Sentence (1) above, our treatment of quantifiers doesn't scale well. For example, if we want to predict two readings for the following sentence (which is just Sentence (1) where "some" and "every" have exchanged position)

(3) Some student aced every exam.

then we need to add an additional lexical entry both for "some" and for "every"; this is easily done, but we end up with two lexical formulas for both words. However, this would still not be enough. For example, the following sentence is also grammatical.

(4) Alyssa gave every student a difficult exam.

(5) Alyssa believes a student committed perjury.

In Sentence (4), "every student" does not occur in a peripheral position, and though it is possible to add a more complex formula with the correct behaviour, we would need yet another formula for Sentence (5). Sentence (5) is generally considered to have two readings: a *de dicto* reading, where Alyssa doesn't have a specific student in mind (she could conclude this, for example, when two students make contradictory statements under oath, this reading can be felicitously followed by "but she doesn't know which"), and a *de re* reading where Alyssa believes a specific student perjured. The Lambek calculus cannot generate this second reading without adding yet another formula for "a".

It seems we are on the wrong track when we need to add a new lexical entry for each different context in which a quantifier phrase occurs. Ideally, we would like a *single* formula for "every", "some" and "a" which applied in all these different cases.

Another way to see this is that we want to keep the deep structure formula $n \multimap ((np \multimap s) \multimap s)$ and that we need to replace the Lambek calculus by another logic such that the correct deep structures for the desired readings of sentences like (4) and (5) are produced.

Lexical Semantics

The grammar above also overgenerates in several ways.

1. "ace" implies a (very positive) form of evaluation with respect to the object. "aced the exam" is good, whereas "aced Emory", outside of the context of a tennis match is bad. "aced logic" can only mean something like "aced the exam for the logic course".
2. "during" and similar temporal adverbs imply its argument is a temporal interval: "during the exam" is good, but "during the student" is bad, and "during logic" can only mean something like "during the contextually understood logic lecture"

In the literature on semantics, there has been an influential movement towards a richer ontology of types (compared to the "flat" Montagovian picture presented above) but also towards a richer set of operations for *combining* terms of specific types, notably allowing type coercions (Pustejovsky, 1995; Asher, 2011). So an "exam" can be "difficult" (it subject matter, or informational content) but also "take a long time" (the event of taking the exam). The theory of semantics outlined in the previous section needs to be extended if we want to take these and other observations into account.

9.3 Modern type-logical grammars

We ended the last section with some problems with using the Lambek calculus as a theory of the syntax-semantics interface. The problems are of two different kinds.

1. The problems of the syntax-semantic interface, and, in a sense, also those of formal language theory are problems where the deep structure is correct but our syntactic calculus cannot produce an analysis mapping to the desired deep structure. We will present two solutions to these problems in Sections 9.3.1 and 9.3.2.
2. The problems of lexical semantics, on the other hand require a more sophisticated type system than Montague's simply typed type-logical with basic types e and t and mechanisms like coercions which allow us to conditionally "repair" certain type mismatches in this system. We will discuss a solution to this problem in Section 9.3.3.

9.3.1 Multimodal grammars

Multimodal type-logical grammars (Moortgat, 2011) take the non-associative Lambek calculus as its base, but allow multiple families of connectives.

For the basic statements $\Gamma \vdash C$ of the Lambek calculus, we ask the question whether we can derive formula C, the succedent, from a sequence of formulas Γ, the antecedent. In the multimodal Lambek calculus, the basic objects are labeled binary trees[4]. The labels come from a separate set of indices or modes I. Multimodal formulas are then of the form $A/_i B$, $A \bullet_i B$ and $A\backslash_i B$, and antecedent terms are of the form $\Gamma \circ_i \Delta$, with i an index from I (we have omitted the outer brackets for the rules, but the operator $\circ_i$ is non-associative). Sequents are still written as $\Gamma \vdash C$, but Γ is now a binary branching, labeled tree with formulas as its leaves.

Given a set of words $w_1, \ldots, w_n$ and a goal formula C, the question is now: is there a labeled tree Γ with formulas $A_1, \ldots, A_n$ as its yield, such that $\Gamma \vdash C$ is derivable and $A_i \in lex(w_i)$ for all i (the implementation of Section 9.4.1 will automatically compute such a Γ).

The rules of multimodal type-logical grammars are shown in Table 9.3. In the rules, $\Gamma[\Delta]$ denotes an antecedent tree Γ with distinguished subtree Δ — the subtree notation is a non-associative version of the Lambek calculus antecedent Γ, Δ, Γ', where Δ is a subsequence instead of a subtree as it is in $\Gamma[\Delta]$.

Each logical connective with mode i uses a structural connective $\circ_i$ in its rule. For the $/E$, $\bullet I$ and $\backslash E$ rules, reading from premisses to conclusions, we *build* structure. For the $/I$, $\bullet E$ and $\backslash I$ rules we *remove* a structural connective with the same mode as the logical connective. The natural deduction rules use explicit antecedents, although, for convenience, we will again use coindexation between the introduction rules for the implications '$/$' and '$\backslash$' and its withdrawn premiss (and similarly for the $\bullet E$ rule and its two premisses).

[4] We can also allow unary branches (and, more generally n-ary branches) and the corresponding logical connectives. The unary connectives $\Diamond$ and $\Box$ are widely used, but, since they will only play a marginal role in what follows, I will not present them to keep the current presentation simple. However, they form an essential part of the analysis of many phenomena and are consequently available in the implementation.

Logical Rules

$$\frac{\Delta \vdash A \bullet_i B \quad \Gamma[A \circ_i B] \vdash C}{\Gamma[\Delta] \vdash C}\ [\bullet_i E] \qquad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma \circ_i \Delta \vdash A \bullet_i B}\ [\bullet_i I]$$

$$\frac{\Gamma \vdash A/_i B \quad \Delta \vdash B}{\Gamma \circ_i \Delta \vdash A}\ [/_i E] \qquad \frac{\Gamma \circ_i B \vdash A}{\Gamma \vdash A/_i B}\ [/_i I]$$

$$\frac{\Gamma \vdash B \quad \Delta \vdash B \backslash_i A}{\Gamma \circ_i \Delta \vdash A}\ [\backslash_i E] \qquad \frac{B \circ_i \Gamma \vdash A}{\Gamma \vdash B \backslash_i A}\ [\backslash_i I]$$

Structural Rules

$$\frac{\Gamma[\Xi'[\Delta_1, \ldots, \Delta_n]] \vdash C}{\Gamma[\Xi[\Delta_{\pi_1}, \ldots, \Delta_{\pi_n}]] \vdash C}\ [SR]$$

Table 9.3 Natural deduction for $\mathbf{NL}_{\mathscr{R}}$

The main advantage of adding modes to the logic is that modes allow us to control the application of structural rules lexically. This gives us fine-grained control over the structural rules in our logic.

For example, the base logic is non-associative. Without structural rules, the sequent $a/b, b/c \vdash a/c$, which is derivable in the Lambek calculus is *not* derivable in its multimodal incarnation $a/_a b, b/_a c \vdash a/_a c$. The proof attempt below, with the failed rule application marked by the '$\lightning$' symbol, shows us that the elimination rules and the introduction rule for this sequent do not match up correctly.

$$\dfrac{\dfrac{a/_a b \vdash a/_a b \qquad \dfrac{b/_a c \vdash b/_a c \quad c \vdash c}{b/_a c \circ_a c \vdash b}\ [/E]}{a/_a b \circ_a (b/_a c \circ_a c) \vdash a}\ [/E]}{\dfrac{(a/_a b \circ_a b/_a c) \circ_a c \vdash a}{a/_a b \circ_a b/_a c \vdash a/_a c}\ [/I]}\ \lightning$$

This is where the structural rules, shown at the bottom of Table 9.3 come in. The general form, read from top to bottom, states that we take a structure Γ containing a distinguished subtree Ξ which itself has n subtrees $\Delta_1, \ldots, \Delta_n$, and we replace this subtree Ξ with a subtree Ξ' which has the same number of subtrees, though not necessarily in the same order (π is a permutation on the leaves). In brief, we replace a subtree Ξ by another subtree Ξ' and possibly rearrange the leaves (subtrees) of Ξ, without deleting or copying any subtrees. Examples of structural rules are the following.

$$\frac{\Gamma[\Delta_1 \circ_a (\Delta_2 \circ_a \Delta_3)] \vdash C}{\Gamma[(\Delta_1 \circ_a \Delta_2) \circ_a \Delta_3] \vdash C}\ Ass \qquad \frac{\Gamma[(\Delta_1 \circ_1 \Delta_3) \circ_0 \Delta_2] \vdash C}{\Gamma[(\Delta_1 \circ_0 \Delta_2) \circ_1 \Delta_3] \vdash C}\ MC$$

The first structural rule is one of the structural rules for associativity. It is the simplest rule which will make the proof attempt above valid (with $\Gamma[]$ the empty

context, $\Delta_1 = a/_a b$, $\Delta_2 = b/_a c$ and $\Delta_3 = c$). This structural rule keeps the order of the Δ_i the same.

The rule above on the right is slightly more complicated. There, the positions of Δ_2 and Δ_3 are swapped as are the relative positions of modes 0 and 1. Rules like this are called "mixed commutativity", they permit controlled access to permutation. One way to see this rule, seen from top to bottom, is that is "moves out" a Δ_3 constituent which is on the right branch of mode 1. Rules of this kind are part of the solution to phenomena like Dutch verb clusters (Moortgat and Oehrle, 1994).

Many modern type-logical grammars, such as the Displacement calculus and NL_{cl} can be seen as multimodal grammars (Valentín, 2014; Barker and Shan, 2014).

9.3.2 First-order linear logic

We have seen that multimodal type-logical grammars generalise the Lambek calculus by offering the possibility of fine-tuned controlled over the application of structural rules. In this section, I will introduce a second way of extending the Lambek calculus.

Many parsing algorithms use pairs of integers to represent the start and end position of substrings of the input string. For example, we can represent the sentence

(6) Alyssa believes someone committed perjury.

as follows (this is a slightly simplified version of Sentence (5) from Section 9.2.3); we have treated "committed perjury" as a single word.

$$0 \xrightarrow{\text{Alyssa}} 1 \xrightarrow{\text{believes}} 2 \xrightarrow{\text{someone}} 3 \xrightarrow{\text{committed perjury}} 4$$

The basic idea of first-order linear logic as a type-logical grammar is that we can code strings as pairs (or, more generally, tuples) of integers representing string positions. So for deciding the grammaticality of a sequence of words $w_1, \ldots, w_n \vdash C$, with a goal formula C, we now give a parametric translation from $\|A_i\|^{i-1,i}$ for each lexical entry w_i and $\|C\|^{0,n}$ for the conclusion formula.

Given these string positions, we can assign the noun phrase "Alyssa" the formula $np(0,1)$, that is a noun phrase from position 0 to position 1. The verb "believes", which occurs above between position 1 and 2, can then be assigned the complex formula $\forall x_2.[s(2,x_2) \multimap \forall x_1.[np(x_1,1) \multimap s(x_1,x_2)]]$, meaning that it first selects a sentence to its right (that is, starting at its right edge, position 2 and ending anywhere) and then a noun phrase to its left (that is, starting anywhere and ending at its left edge, position 1) to produce a sentence from the left position of the noun phrase argument to the right position of the sentence argument.

We can systematise this translation, following Moot and Piazza (2001), and obtain the following translation from Lambek calculus formulas to first-order linear logic formulas.

$$\|p\|^{x,y} = p(x,y)$$
$$\|A/B\|^{x,y} = \forall z.\|B\|^{y,z} \multimap \|A\|^{x,z}$$
$$\|B\backslash A\|^{y,z} = \forall x.\|B\|^{x,y} \multimap \|A\|^{x,z}$$
$$\|A \bullet B\|^{x,z} = \exists y.\|A\|^{x,y} \otimes \|B\|^{y,z}$$

Given this translation, the lexical entry for "believes" discussed above is simply the translation of the Lambek calculus formula $(np\backslash s)/s$, with position pair $1,2$, to first-order linear logic. Doing the same for "committed perjury" with formula $np\backslash s$ and positions $3,4$ gives $\forall z.[np(z,3) \multimap s(z,4)]$. For "someone" we would simply translate the Lambek calculus formula $s/(np\backslash s)$, but we can do better than that: when we translate "someone" as $\forall y_1.\forall y_2.[(np(2,3) \multimap s(y_1,y_2)) \multimap s(y_1,y_2)]$, we improve upon the Lambek calculus analysis.

As we noted in Section 9.2.3, the Lambek calculus cannot generate the "de re" reading, where the existential quantifier has wide scope. Figure 9.5 shows how the simple first-order linear logic analysis *does* derive this reading.

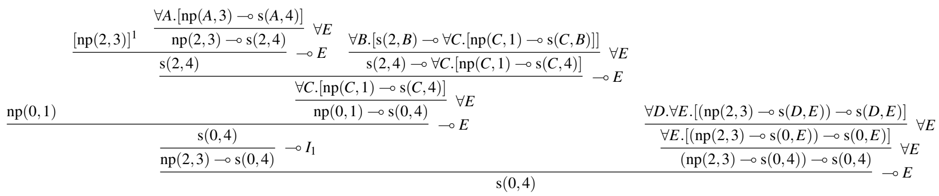


Fig. 9.5 "De re" reading for the sentence "Alyssa believes someone committed perjury".

Besides the Lambek calculus, first-order linear logic has many other modern type-logical grammars as fragments. Examples include lambda grammars (Oehrle, 1994), the Displacement calculus (Morrill et al, 2011) and hybrid type-logical grammars (Kubota and Levine, 2012). We can see first-order linear logic as a sort of "machine language" underlying these different formalisms, with each formalism introducing its own set of abbreviations convenient for the grammar writer. Seeing first-order linear logic as an underlying language allows us to compare the analyses proposed for different formalisms and find, in spite of different starting points, a lot of convergence. In addition, as discussed in Section 9.4.2, we can use first-order linear logic as a uniform proof strategy for these formalisms.

Syntax-Semantics Interface

As usual, we obtain the deep structure of a syntactic derivation by defining a homomorphism from the syntactic proof to a proof in multiplicative intuitionistic linear logic. For first-order linear logic, the natural mapping simply forgets all first-order

$$\frac{\begin{array}{c}[A]^i[B]^i\\ \vdots\\ A\otimes B \quad C\end{array}}{C}\otimes E_i \qquad \frac{A \quad B}{A\otimes B}\otimes I$$

$$\frac{A \quad A\multimap B}{B}\multimap E \qquad \frac{\begin{array}{c}[A]^i\\ \vdots\\ B\end{array}}{A\multimap B}\multimap I$$

$$\frac{\begin{array}{c}[A]^i\\ \vdots\\ \exists x.A \quad C\end{array}}{C}\exists E_i^* \qquad \frac{A[x:=t]}{\exists x.A}\exists I$$

$$\frac{\forall x.A}{A[x:=t]}\forall E \qquad \frac{A}{\forall x.A}\forall I^*$$

* no free occurrences of x in any of the free hypotheses

Table 9.4 Natural deduction rules for MILL1

quantifiers and replaces all atomic predicates $p(x_1,\ldots,x_n)$ by propositions p. Since the first-order variables have, so far, only been used to encode string positions, such a forgetful mapping makes sense.

However, other solutions are possible. When we add semantically meaningful terms to first-order linear logic, the Curry-Howard isomorphism for the first-order quantifiers will give us dependent types and this provides a natural connection to the work using dependent types for formal semantics (Ranta, 1991; Pogodalla and Pompigne, 2012; Luo, 2012b, 2015).

9.3.3 The Montagovian Generative Lexicon

In the previous sections, we have discussed two general solutions to the problems of the syntax-semantics interface of the Lambek calculus. Both solutions proposed a more flexible syntactic logic. In this section, we will discuss a different type of added flexibility, namely in the syntax-semantics interface itself.

The basic motivating examples for a more flexible composition have been amply debated in the literature (Pustejovsky, 1995; Asher, 2011). Our solution is essentially the one proposed by Bassac et al (2010), called the Montagovian Generative Lexicon. I will only give a brief presentation of this framework. More details can be found in Chapter 6.

Like many other solutions, the first step consists of splitting Montague's type e for entities into several types: physical objects, locations, informational objects,

eventualities, etc. Although there are different opinions with respect to the correct granularity of types (Pustejovsky, 1995; Asher, 2011; Luo, 2012a), nothing much hinges on this for the present discussion.

The second key element is the move to the second-order lambda calculus, system F (Girard et al, 1995), which allows abstraction over *types* as well as over terms. In our Lambek calculus, the determiner "the" was assigned the formula np/n and the type of its lexical semantics was therefore $(e \to t) \to e$, which we implement using the ι operators of type $(e \to t) \to e$, which, roughly speaking, selects a contextually salient entity from (a characteristic function of) a set. When we replace the single type e by several different types, we want to avoid listing several separate syntactically identical by semantically different entries for "the" in the lexicon, and therefore assign it a polymorphic term $\Lambda\alpha.\iota^{(\alpha\to t)\to\alpha}$ of type $\Pi\alpha.((\alpha \to t) \to \alpha)$, quantifying over *all* types α. Though this looks problematic, the problem is resolved once we realise that only certain function words (quantifiers, conjunctions like "and") are assigned polymorphic terms and that we simply use universal instantiation to obtain the value of the quantifier variable. So if "student" is a noun of type human, that is of type $h \to t$, then "the student" will be of type h, instantiating α to h. Formally, we use β reduction as follows (this is substitution of types instead of terms, substituting type h for α).

$$((\Lambda\alpha.\iota^{(\alpha\to t)\to\alpha})\{h\}\, student^{h\to t}) =_\beta (\iota\, student)^h$$

The final component of the Montagovian Generative Lexicon is a set of lexically specified, optional transformations. In case of a type mismatch, an optional transformation can "repair" the term.

As an example from Moot and Retoré (2011) and Mery et al (2013), one of the classic puzzles in semantics are plurals and collective and distributive readings. For example, verbs like "meet" have collective readings, they apply to groups of individuals collectively, so we have the following contrast, where collectives like committees and plurals like students can meet, but not singular or distributively quantified noun phrases. The contrast with verbs like "sneeze", which force a distributive reading is clear.

(7) The committee met.

(8) All/the students met

(9) *A/each/the student met.

(10) All/the students sneezed.

(11) A/each/the student sneezed.

In the Montagovian Generative lexicon, we can models these fact as follows. First, we assign the plural morphology "-s" the semantics $\Lambda\alpha\lambda P^{\alpha\to t}\lambda Q^{\alpha\to t}.|Q| > 1 \wedge \forall x^\alpha.Q(x) \Rightarrow P(x)$, then "students" is assigned the following term $\lambda Q^{h\to t}.|Q| > 1 \wedge \forall x^h.Q(x) \Rightarrow student(x)$, that is the sets of cardinality greater than one such that all its members are students. Unlike "student" which was assigned a term of type $h \to t$, roughly a property of humans, the plural "students" is assigned a term of

type $(h \to t) \to t$, roughly a property of sets of humans. Consequently, the contrast between "the student" and "the students" is that the first is of type h (a human) and the second of type $h \to t$ (a set of humans) as indicated below.

phrase	syntactic type	lambda-term
the student	np	$(\iota \mathit{student})^{h}$
the students	np	$(\iota(\lambda Q^{h\to t}.\lvert Q\rvert > 1 \wedge \forall x^{h} Q(x) \Rightarrow \mathit{student}(x)))^{h\to t}$

Therefore, the meaning of "the students" is the contextually determined set of humans, from the sets of more than one human such that all of them are students.

Then we distinguish the verbs "meet" and "sneeze" as follows, with the simpler verb "sneeze" simply selecting for a human subject and the collective verb "meet" selecting for a set of humans (of cardinality greater than one) as its subject.

word	syntactic type	lambda-term
met	$np\backslash s$	$\lambda P^{h\to t}.\lvert P\rvert > 1 \wedge \mathit{meet}(P)$
#		$\Lambda\alpha\lambda R^{(\alpha\to t)\to t}\lambda S^{(\alpha\to t)\to t}\forall P^{\alpha\to t}.S(P) \Rightarrow R(P)$
$\mathit{met}^{\#}$	$np\backslash s$	$\lambda R^{(h\to t)\to t}\forall P^{h\to t}.R(P) \Rightarrow \lvert P\rvert > 1 \wedge \mathit{meet}(P)$
sneezed	$np\backslash s$	$\lambda x^{h}.\mathit{sneeze}(x)$
*		$\Lambda\alpha\lambda P^{\alpha\to t}\lambda Q^{\alpha\to t}\forall x^{\alpha}.Q(x) \Rightarrow P(x)$
$\mathit{sneezed}^{*}$	$np\backslash s$	$\lambda P^{h\to t}.\forall x^{h}.P(x) \Rightarrow \mathit{sneeze}(x)$

Given these basic lexical entries, we already correctly predict that "the student met" is ill-formed semantically (there is an unresolvable type mismatch) but "the students met" and "the student sneezed" are given the correct semantics.

The interesting case is "the students sneezed" which has as its only reading that each student sneezed individually. Given that "the students" is of type $h \to t$ and that "sneezed" requires an argument of type h, there is a type mismatch when we apply the two terms. However, "sneeze" has the optional distributivity operator '*', which when we apply it to the lexical semantics for "sneeze" produces the term $\lambda P^{h\to t}.\forall x^{h}.P(x) \Rightarrow \mathit{sneeze}(x)$, which combines with "the students" to produce the reading.

$$\forall x^{h}.(\iota(\lambda Q^{h\to t}.\lvert Q\rvert > 1 \wedge \forall y^{h} Q(y) \Rightarrow \mathit{student}(y))\, x) \Rightarrow \mathit{sneeze}(x)$$

In other words, all of the members of the contextually determined set of more than human which are all students, sneeze.

The basic idea for the Montagovian Generative Lexicon is that lexical entries specify optional transformations which can repair certain sorts of type mismatches in the syntax-semantics interface. This adaptability allows the framework to solve many semantic puzzles.

Though a proof-of-concept application of these ideas exists, more robust and scalable applications, as well as efforts incorporate these ideas into wide-coverage semantics, are ongoing research.

9.4 Theorem proving

When looking at the rules and examples for the different logics, the reader may have wondered: how do we actually find proofs for type-logical grammars? This question becomes especially urgent once our grammars become more complex and the consequences of our lexical entries, given our logic, become hard to oversee. Though pen and paper generally suffice to show that a given sentence is derivable for the desired reading, it is generally much more laborious to show that a given sentence is underivable or that it has only the desired readings. This is where automated theorem provers are useful: they allow more extensive and intensive testing of your grammars, producing results more quickly and with less errors (though we should be careful about too naively assuming the implementation we are using is correct: when a proof is found it is generally easy to verify its correctness by hand, but when a proof isn't found because of a programming error this can be hard to detect).

Though the natural deduction calculi we have seen so far can be used for automated theorem proving (Carpenter, 1994; Moot and Retoré, 2012), and though Lambek (1958) already gave a sequent calculus decision procedure, both logics have important drawbacks for proof search.

Natural deduction proofs have a 1-1 correspondence between proofs and readings, though this is somewhat complicated to enforce for a logic with the $\bullet E$ rule (and the related $\Diamond E$ rule). For the sequent calculus, the product rule is just like the other rules, but sequent calculus suffers from the so-called "spurious ambiguity" problem, which means that it generates many more proofs than readings.

Fortunately, there are proof systems which combine the good aspects of natural deduction and sequent calculus, and which eliminate their respective drawbacks. Proof nets are a graphical representation of proofs first introduced for linear logic (Girard, 1987). Proof nets suffer neither from spurious ambiguity nor from complications for the product rules.

Proof nets are usually defined as a subset of a larger class, called *proof structures*. Proof structures are "candidate proofs": part of the search space of a naive proof search procedure which need not correspond to actual proofs. Proof nets are those proof structures which correspond to sequent proofs. Perhaps surprisingly, we can distinguish proof nets from other proof structures by looking only at graph-theoretical properties of these structures.

Proof search for type-logical grammars using proof nets uses the following general procedure.

1. For each of the words in the input sentence, find one of the formulas assigned to it in the lexicon.
2. Unfold the formulas to produce a partial proof structure.
3. Enumerate all proof structures for the given formulas by identifying nodes in the partial proof structure.
4. Check if the resulting proof structure is a proof net according to the correctness condition.

In Sections 9.4.1 and 9.4.2 we will instantiate this general procedure for multimodal type-logical grammar and for first-order linear logic respectively.

9.4.1 Multimodal proof nets

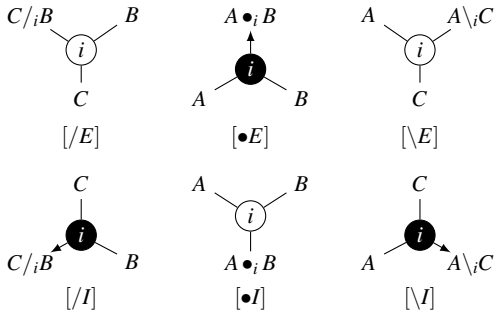


Table 9.5 Links for multimodal proof nets

Table 9.5 presents the links for multimodal proof nets. The top row list the links corresponding to the elimination rules of natural deduction, the bottom row those corresponding to the introduction rules. There are two types of links: tensor links, with an open center, and par links, with a filled center. Par links have a single arrow pointing to the main formula of the link (the complex formula containing the principal connective). The top and bottom row are up-down symmetric with tensor and par reversed. The tensor links correspond to the logical rules which build structure when we read them from top to bottom, the par links to those rules which remove structure.

The formulas written above the central node of a link are its premisses, whereas the formulas written below it are its conclusions. Left-to-right order of the premisses as well as the conclusions is important.

A *proof structure* is a set of formula occurrences and a set of links such that:

1. each formula is at most once the premiss of a link,
2. each formula is at most once the conclusion of a link.

A formula which is not the premiss of any link is a conclusion of the proof structure. A formula which is not the conclusion of any link is a hypothesis of the proof structure. We say a proof structure with hypotheses Γ and conclusions Δ is a proof structure of $\Gamma \vdash \Delta$ (we are overloading of the '$\vdash$' symbol here, though this use should always be clear from the context; note that Δ can contain multiple formulas).

After the first step of lexical lookup we have a sequent $\Gamma \vdash C$, and we can enumerate its proof structures as follows: unfold the formulas in Γ, C, unfolding them

so that the formulas in Γ are hypotheses and the formula C is a conclusion of the resulting structure, until we reach the atomic subformulas (this is step 2 of the general procedure), then identify atomic subformulas (step 3 of the general procedure, we turn to the last step, checking correctness, below). This identification step can, by the conditions on proof structures only identify hypotheses with conclusions and must leave all formulas of Γ, including atomic formulas, as hypotheses and C as a conclusion.

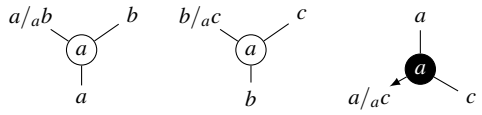


Fig. 9.6 Lexical unfolding of $a/_a b, b/_a c \vdash a/_a c$

Figure 9.6 shows the lexical unfolding of the sequent $a/_a b, b/_a c \vdash a/_a c$. It is already a proof structure, though a proof structure of $a, a/_a b, b, b/_a c, c \vdash a, a/_a c, b, c$ (to the reader familiar with the proof nets of linear logic: some other presentations of proof nets use more restricted definitions of proof structures where a "partial proof structure" such as shown in the figure is called a *module*).

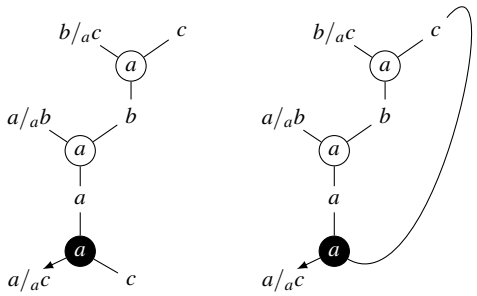


Fig. 9.7 The proof structure of Figure 9.6 after identification of the a and b atoms (left) and after identification of all atoms

To turn this proof structure into a proof structure of $a/_a b, b/_a c \vdash a/_a c$, we identify the atomic formulas. In this case, there is only a single way to do this, since a, b and c all occur once as a hypothesis and once as a conclusion, though in general there may be many possible matchings. Figure 9.7 shows, on the left, the proof structure after identifying the a and b formulas. Since left and right (linear order), up and down (premiss, conclusion) have meaning in the graph, connecting the c formulas is less obvious: c is a conclusion of the $/I$ link and must therefore be below it, but

a premiss of the $/E$ link and must therefore be above it. This is hard to achieve in the figure shown on the left. Though a possible solution would be to draw the figure on a cylinder, where "going up" from the topmost c we arrive at the bottom one, for ease of type-setting and reading the figure, I have chosen the representation shown in Figure 9.7 on the right. The curved line goes up from the c premiss of the $/E$ link and arrives from below at the $/I$ link, as desired. One way so see this strange curved connection is as a graphical representation of the coindexation of a premiss with a rule in the natural deduction rule for the implication.

Figure 9.7 therefore shows, on the right, a proof structure for $a/_a b, b/_a c \vdash a/_a c$. However, is it also a *proof net*, that is, does it correspond to a proof? In a multi-modal logic, the answer depends on the available structural rules. For example, if no structural rules are applicable to mode a then $a/_a b, b/_a c \vdash a/_a c$ is underivable, but if mode a is associative, then it is derivable.

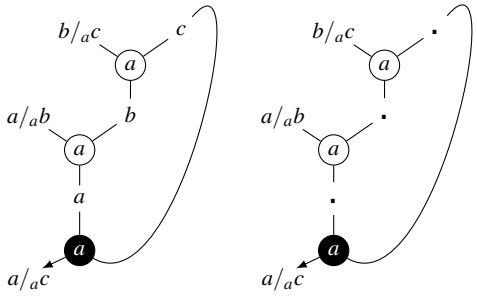


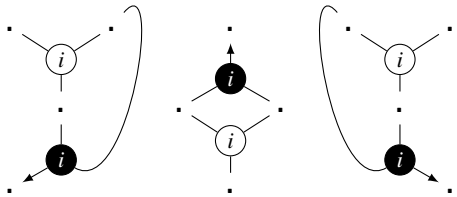
Fig. 9.8 The proof structure of Figure 9.7 (left) and its abstract proof structure (right)

We decide whether a proof structure is a proof net based only on properties of the graph. As a first step, we erase all formula information from the internal nodes of the graph; for administrative reasons, we still need to be able to identify which of the hypotheses and conclusion of the structure correspond to which formula occurrence[5]. All relevant information for correctness is present in this graph, which we call an *abstract proof structure*.

We talked about how the curved line in proof structures (and abstract proof structure) corresponds to the coindexation of discharged hypotheses with rule names for

[5] We make a slight simplification here. A single vertex abstract proof structure can have both a hypothesis and a conclusion without these two formulas necessarily being identical, e.g. for sequents like $(a/b) \bullet b \vdash a$. Such a sequent would correspond to the abstract proof structure $\begin{smallmatrix}(a/b)\bullet b\\ \cdot\\ a\end{smallmatrix}$. So, formally, both the hypotheses and the conclusions of an abstract proof structure are assigned a formula and when a node is both a hypothesis and a conclusion it can be assigned two different formulas. In order not to make the notation of abstract proof structure more complex, we will stay with the simpler notation. Moot and Puite (2002) present the full details.

the implication introduction rules. However, the introduction rules for multimodal type-logical grammars actually do more than just discharge a hypothesis, they also check whether the discharged hypothesis is the immediate left (for $\backslash I$) or right (for $/I$) daughter of the root node, that is, that the withdrawn hypothesis A occurs as $A \circ_i \Gamma$ (for $\backslash I$ and mode i) or $\Gamma \circ_i A$ (for $/I$ and mode i). The par links in the (abstract) proof structure represent a sort of "promise" that will produce the required structure. We check whether it is satisfied by means of contractions on the abstract proof structure.



$[/I]$ $[\bullet E]$ $[\backslash I]$

Table 9.6 Contractions — multimodal binary connectives

The multimodal contractions are shown in Table 9.6. All portrayed configurations contract to a single vertex: we erase the two internal vertices and the paired links and we identify the two external vertices, keeping all connections of the external vertices to the rest of the abstract proof structure as they were: the vertex which is the result of the contraction will be a conclusion of the same link as the top external vertex (or a hypothesis of the abstract proof structure in case it wasn't) and it will be a premiss of the same link as the bottom external vertex (or a conclusion of the abstract proof structure in case it wasn't).

The contraction for $/I$ checks if the withdrawn hypothesis is the right daughter of a tensor link with the same mode information i, and symmetrically for the $\backslash I$ contraction. The $\bullet E$ contraction contracts two hypotheses occurring as sister nodes.

All contractions are instantiations of the same pattern: a tensor link and a par link are connected, respecting left-right and up-down the two vertices of the par link without the arrow.

To get a better feel for the contractions, we will start with its simplest instances. When we do pattern matching on the contraction for $/I$, we see that it corresponds to the following patterns, depending on our choice for the tensor link (the par link is always $/I$).

$$C/_i B \vdash C/_i B$$
$$A \vdash (A \bullet_i B)/_i B$$
$$A \vdash C/_i (A\backslash_i C)$$

A proof structure is a proof net iff it contracts to a tree containing only tensor links using the contractions of Table 9.6 and any structural rewrites, discussed below — Moot and Puite (2002) present full proofs. In other words, we need to contract all par links in the proof structure according to their contraction, each contraction ensuring the correct application of the rule after which it is named. The abstract proof structure on the right of Figure 9.8 does not contract, since there is no substructure corresponding to the $/I$ contraction: for a valid contraction, a par link is connected to both "tentacles" of a single tensor link, and in the figure the two tentacles without arrow are connected to different tensor links. This is correct, since $a/_a b, b/_a c \vdash a/_a c$ is underivable in a logic without structural rules for a.

However, we have seen that this statement becomes derivable once we add associativity of a and it is easily verified to be a theorem of the Lambek calculus. How can we add a modally controlled version of associativity to the proof net calculus? We can add such a rule by adding a rewrite from a tensor tree to another tensor tree with the same set of leaves. The rewrite for associativity is shown in Figure 9.9. To apply a structural rewrite, we replace the tree on the left hand side of the arrow by the one on the right hand side, reattaching the leaves and the root to the rest of the proof net.

Just like the structural rules, a structural rewrite always has the same leaves on both sides of the arrow — neither copying nor deletion is allowed[6], though we can reorder the leaves in any way (the associativity rule doesn't reorder the leaves).

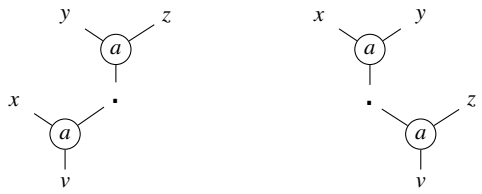


Fig. 9.9 Structural rewrites for associativity of mode a.

Figure 9.10 shows how the contractions and the structural rewrites work together to derive $a/_a b, b/_a c \vdash a/_a c$.

We start with a structural rewrite, which rebrackets the pair of tensor links. The two hypotheses are now the premisses of the same link, and this also produces a contractible structure for the $/I$ link. Hence, we have shown the proof structure to be a proof net.

[6] From the point of view of linear logic, we stay within the purely multiplicative fragment, which is simplest proof-theoretically.

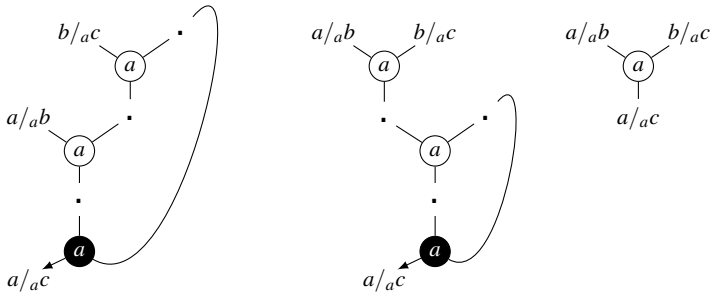


Fig. 9.10 Structural rewrite and contraction for the abstract proof structure of Figure 9.8, showing this is a proof net for $a/_a b \circ_a b/_a c \vdash a/_a c$

In the Grail theorem prover, the representation of abstract proof structures looks as shown in Figure 9.11 (this is an automatically produced subgraph close to the graph on the left of Figure 9.10, though with a non-associative mode *n* and therefore not derivable). This graph is used during user interaction. The graphs are drawn using GraphViz, an external graph drawing program which does not guarantee respecting our desires for left, right and top/bottom, so tentacles are labeled 1, 2 and 3 (for left, right and top/bottom respectively) to allow us to make these distinctions regardless of the visual representation. Vertices are given unique identifiers for user interaction, for example to allow specifying which pair of atoms should be identified or which par link should be contracted.

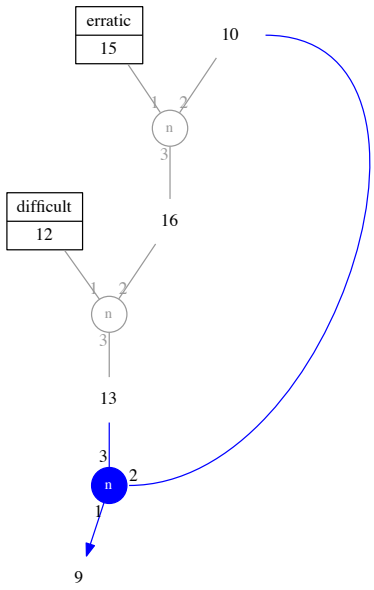


Fig. 9.11 Interactive Grail output

Although the structural rules give the grammar writer a great deal of flexibility, such flexibility complicates proof search. As discussed at the beginning of Section 9.4, theorem proving using proof nets is a four step process, which in the current situation looks as follows: 1) lexical lookup, 2) unfolding, 3) identification of atoms, 4) graph rewriting. In the current

case, both the graph rewriting and the identification of atoms are complicated[7] and since we can interleave the atom connections and the graph rewriting it is not a priori clear which strategy is optimal for which set of structural rules. The current implementation does graph rewriting only once all atoms have been connected.

The Grail theorem prover implements some strategies for early failure. Since all proofs in multimodal type-logical grammars are a subset of the proofs in multiplicative linear logic, we can reject (partial) proof structures which are invalid in multiplicative linear logic, a condition which is both powerful and easy to check.

As a compromise between efficiency and flexibility, Grail allows the grammar writer to specify a first-order approximation of her structural rules. Unlike the test for validity in multiplicative linear logic which is valid for any set of structural rules, specifying such a first-order approximation is valid only when there is a guarantee that all derivable sequents in the multimodal grammar are a subset of their approximations derivable in first-order linear logic. Errors made here can be rather subtle and hard to detect. It is recommended to use such methods to improve parsing speed only when a grammar has been sufficiently tested and where it is possible to verify whether no valid readings are excluded, or, ideally, to prove that the subset relation holds between the multimodal logic and its first-order approximation.

The next section will discuss first-order proof nets in their own right. Though these proof nets have been used as an underlying mechanism in Grail for a long time, we have seen in Section 9.3.2 that many modern type-logical grammars are formulated in a way which permits a direct implementation without an explicit set of structural rules.

As to the proof search strategy used by Grail, it is an instance of the "dancing links" algorithm (Knuth, 2000): when connecting atomic formulas, we always link a formula which has the least possibilities and we rewrite the abstract proof structures only once a fully linked proof structure has been produced. Though the parser is not extremely fast, evaluation both on randomly generated statements and on multimodal statements extracted from corpora show that the resulting algorithm performs more than well enough (Moot, 2008).

9.4.2 First-order proof nets

Proof nets for first-order linear logic (Girard, 1991) are a simple extension of the proof nets for standard, multiplicative linear logic (Danos and Regnier, 1989). Compared to the multimodal proof nets of the previous section, all logical links have the main formula of the link as their conclusion but there is now a notion of *polarity*, corresponding to whether or not the formula occurs on the left hand side of the turnstile (negative polarity) or on the right hand side (positive polarity).

[7] Lexical ambiguity is a major problem for automatically extracted wide-coverage grammars as well, though standard statistical methods can help alleviate this problem (Moot, 2010).

We unfold a sequent $A_1, \ldots, A_n \vdash C$ by using the negative unfolding for each of the A_i and the positive unfolding for C. The links for first-order proof nets are shown in Table 9.7.

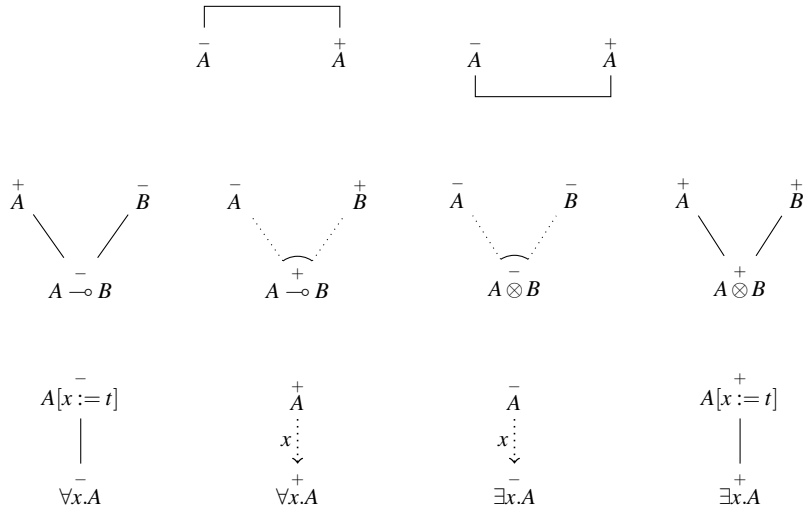


Table 9.7 Logical links for MILL1 proof structures

Contrary to multimodal proof nets, where a tensor link was drawn with an open central node and a par link with a filled central node, here par links are drawn as a connected pair of dotted lines and tensor links as a pair of solid lines.

As before, premisses are drawn above the link and conclusions are drawn below it. With the exception of the cut and axiom links, the order of the premisses and the conclusions is important. We assume without loss of generality that every quantifier link uses a distinct eigenvariable.

A set of formula occurrences connected by links is a proof structure if every formula is at most once the premiss of a link and if every formula is exactly once the conclusion of a link. Those formulas which are not the premiss of any link are the conclusions of the proof structure — note the difference with multimodal proof nets: a proof structure has conclusions but no hypotheses and, as a consequence, each formula in the proof net must be the conclusion of exactly one (instead of at most one) link.

For polarised proof nets, unfolding the formulas according to the links of Table 9.7 no longer produces a proof structure, since the atomic formulas after unfolding are not the conclusions of any link. Such "partial proof structures" are called a modules. To turn a module into a proof structure, we connect atomic formulas of opposite polarity by axiom links until we obtain a complete matching of the atomic formulas, that is until every atomic formula is the conclusion of an axiom link.

The negative $\forall$ and the positive $\exists$ link, are defined using substitution of an arbitrary term t for the eigenvariable of the link. In actual proof search, we use unification of these variables when the axiom links are performed.

As usual, not all proof structures are proof nets. However, since the logical rules for the quantifiers make essential use of the notion of "free occurrence of a variable", this should be reflected in out correctness condition. Girard (1991) uses a notion of switching for proof structures which extends the switchings of Danos and Regnier (1989).

A *switching* is, for each of the binary par links a choice of its left or right premiss and for each of the unary par links with eigenvariable x a choice of one of the formulas in the structure with a free occurrence of x *or* of the premiss of the rule.

Given a switching, a *correction graph* replaces a binary par link by a connection from the conclusion of the link to the premiss chosen by the switching, and it replace a unary par link by a link from the conclusion to the formula chosen by the switching.

Finally, a proof structure is a *proof net* when all its correction graphs are both acyclic and connected (Girard, 1991).

As an example, look at the proof structure of $a \multimap \exists x.b(x) \vdash \exists y.[a \multimap b(y)]$ shown in Figure 9.12. This statement is *not* derivable in first-order linear logic (nor in intuitionistic logic). Consider therefore the switching connecting the binary par link to its left premiss a and the link for x to the formula $a \multimap b(x)$ (it has a free occurrence of x, so this like is a valid switching).

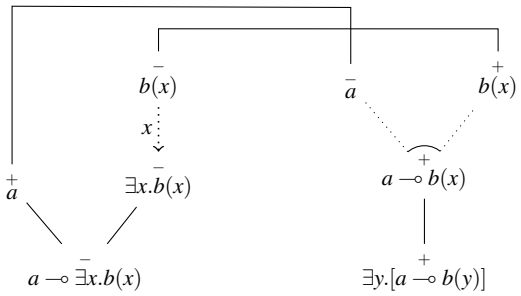


Fig. 9.12 Proof structure for $a \multimap \exists x.b(x) \vdash \exists y.[a \multimap b(y)]$.

This switching produces the correction graph shown in Figure 9.13. It contains a cycle, drawn with bold edges, and is therefore not a proof structure (in addition, the b axiom is disconnected from the rest of the structure, giving a second reason for rejecting the proof structure).

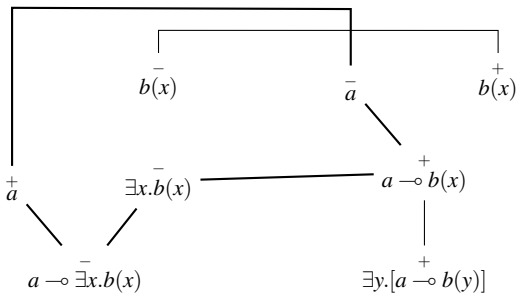


Fig. 9.13 Correction graph for the proof structure of Figure 9.12 with the cycle indicated, showing $a \multimap \exists x.b(x) \vdash \exists y.[a \multimap b(y)]$ is underivable

Contractions

Though switching conditions for proof nets are simple and elegant, they don't lend themselves to naive application: already for the example proof structure of Figure 9.12 there are six possible switchings to consider and, as the reader can verify, only the switching shown in Figure 9.13 is cyclic (and disconnected). In general, it is often the case that all switchings but one are acyclic and connected, as it is here.

Though there are efficient ways of testing acyclicity and connectedness for multiplicative proof nets (Guerrini, 1999; Murawski and Ong, 2000) and it seems these can be adapted to the first-order case (though some care needs to be taken when we allow complex terms), the theorem prover for first-order linear logic uses a extension of the contraction criterion of Danos (1990).

Given a proof structure we erase all formulas from the vertices and keep only a set of the free variables at this vertex. We then use the contractions of Table 9.8 to contract the edges of the graph. The resulting vertex of each contraction has the union of the free variables of the two vertices of the redex (we remove the eigenvariable x of a $\forall$ contraction, " $\Rightarrow_u$"). A proof structure is a proof net iff it contracts to a single vertex using the contractions of Table 9.8.

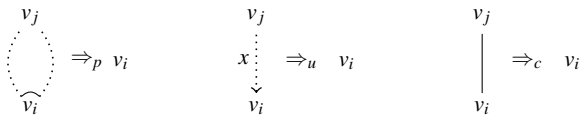


Table 9.8 Contractions for first-order linear logic. Conditions: $v_i \neq v_j$ and, for the u contraction, all free occurrences of x are at v_j.

To give an example of the contractions, Figure 9.14 shows the contractions for the underivable proof structure of Figure 9.12. The initial structure, which simply takes the proof structure of Figure 9.12 and replaces the formulas by the corresponding

set of free variables, is shown on the left. Contracting the five solid edges using the *c* contraction produces the structure shown in the figure on the right.

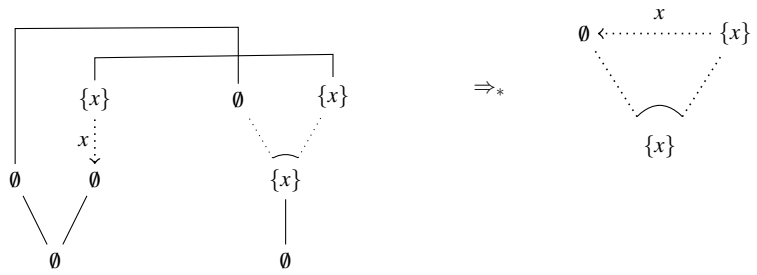


Fig. 9.14 Contractions for the underivable $a \multimap \exists x.b(x) \vdash \exists y.[a \multimap b(y)]$.

No further contractions apply: the two connected dotted links from the binary par link do not end in the same vertex, so the par contraction *p* cannot apply. In addition, the universal contraction *u* cannot apply either, since it requires all vertices with its eigenvariable *x* to occur at the node from which the arrow is leaving and there is another occurrence of *x* at the bottom node of the structure. We have therefore shown that this is not a proof net.

Since there are no structural rewrites, the contractions for first-order linear logic are easier to apply than those for multimodal type-logical grammars: it is rather easy to show confluence for the contractions (the presence of structural rules, but also the unary versions of the multimodal contractions, means confluence is not guaranteed for multimodal proof nets). We already implicitly used confluence when we argued that the proof structure in Figure 9.14 was not a proof net. The theorem prover uses a maximally contracted representation of the proof structure to represent the current state of proof search and this means less overhead and more opportunities for early failure during proof search.

Like before, the theorem proving uses four steps, which look as follows in the first-order case: 1) lexical lookup, 2) unfolding, 3) axiom links with unification, 4) graph contraction. Unlike the multimodal proof nets of the previous section, the graph contractions are now confluent and can be performed efficiently (the linear time solutions for the multiplicative case may be adaptable, but a naive implementation already has an $O(n^2)$ worst-case performance). After lexical lookup, theorem proving for first-order linear logic unfolds the formulas as before, but uses a greedy contraction strategy. This maximally contracted partial proof net constrains further axiom links: for example, a vertex containing a free variable *x* cannot be linked to the conclusion of the edge of its eigenvariable (the vertex to which the arrow of the edge with variable *x* points) or to one of its descendants, since such a structure would fail to satisfy the condition that the two vertices of a $\forall$ link for the *u* contraction of Figure 9.8 are distinct. Another easily verified constraint is that two atomic

formulas can only be connected by an axiom link if these formulas unify[8]. Like for multimodal proof nets, the first-order linear logic theorem prover chooses an axiom link for one of the atoms with the fewest possibilities.

9.4.3 Tools

Table 9.9 lists the different theorem provers which are available. Grail 0 (Moot et al, 2015) and Grail 3 (Moot, 2015a) use the multimodal proof net calculus of Section 9.4.1, whereas LinearOne (Moot, 2015c) uses the first-order proof nets of Section 9.4.2. GrailLight (Moot, 2015b) is a special-purpose chart parser, intended for use with an automatically extracted French grammar for wide-coverage parsing and semantics (Moot, 2010, 2012). All provers are provided under the GNU Lesser General Public License — this means, notably, there is no warranty, though I am committed to making all software as useful as possible; so contact me for any comments, feature requests or bug reports. All theorem provers can be downloaded from the author's GitHub site.

`https://github.com/RichardMoot/`

The columns of table Table 9.9 indicate whether the theorem provers provide natural deduction output, graph output (of the partial proof nets), whether there is an interactive mode for proof search, whether the implementation is complete and whether the grammar can specify its own set of structural rules; "NA" means the question doesn't apply to the given system (GrailLight doesn't use a graphs to represent proofs and first-order linear logic does not have a grammar-specific set of structural rules). The table should help you select the most adequate tool for your purposes.

LinearOne provides natural deduction output not only for first-order linear logic, but also for the Displacement calculus, hybrid type-logical grammars and lambda grammars. That is, the grammar writer can write a grammar in any of these formalisms, LinearOne will do proof search of the translation of this grammar in first-order linear logic and then translate any resulting proofs back to the source language.

Prover	ND	Graph	Interactive	Complete	User-defined SR
Grail 0	+	–	–	+	+
Grail 3	–	+	+	+	+
GrailLight	+	NA	+	–	–
LinearOne	+	+	–	+	NA

Table 9.9 The different theorem provers

[8] As discussed in Section 9.4.1, the multimodal theorem prover allows the grammar writer to specify first-order approximations of specific formulas. So underneath the surface of Grail there is some first-order reasoning going on as well.

The syntactic example proofs in this chapter have been automatically generated using these tools and the corresponding grammars files, as well as many other example grammars, are included in the repository.

References

Ajdukiewicz K (1935) Die syntaktische Konnexität. Studies in Philosophy 1:1–27
Asher N (2011) Lexical Meaning in Context: A Web of Words. Cambridge University Press
Bar-Hillel Y (1953) A quasi-arithmetical notation for syntactic description. Language 29(1):47–58
Barker C, Shan C (2014) Continuations and Natural Language. Oxford Studies in Theoretical Linguistics, Oxford University Press
Bassac C, Mery B, Retoré C (2010) Towards a type-theoretical account of lexical semantics. Journal of Logic, Language and Information 19(2):229–245, DOI 10.1007/s10849-009-9113-x, URL http://dx.doi.org/10.1007/s10849-009-9113-x
van Benthem J (1995) Language in Action: Categories, Lambdas and Dynamic Logic. MIT Press, Cambridge, Massachusetts
Carpenter B (1994) A natural deduction theorem prover for type-theoretic categorial grammars. Tech. rep., Carnegie Mellon Laboratory for Computational Linguistics, Pittsburgh, Pennsylvania
Chatzikyriakidis S (2015) Natural language reasoning using Coq: Interaction and automation. In: Proceedings of Traitement Automatique des Langues Naturelles (TALN 2015)
Danos V (1990) La logique linéaire appliquée à l'étude de divers processus de normalisation (principalement du λ-calcul). PhD thesis, University of Paris VII
Danos V, Regnier L (1989) The structure of multiplicatives. Archive for Mathematical Logic 28:181–203
Girard JY (1987) Linear logic. Theoretical Computer Science 50:1–102
Girard JY (1991) Quantifiers in linear logic II. In: Corsi G, Sambin G (eds) Nuovi problemi della logica e della filosofia della scienza, CLUEB, Bologna, Italy, vol II, proceedings of the conference with the same name, Viareggio, Italy, January 1990
Girard JY, Lafont Y, Regnier L (eds) (1995) Advances in Linear Logic. London Mathematical Society Lecture Notes, Cambridge University Press
Guerrini S (1999) Correctness of multiplicative proof nets is linear. In: Fourteenth Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Science Society, pp 454–263
Huijbregts R (1984) The weak inadequacy of context-free phrase structure grammars. In: de Haan G, Trommelen M, Zonneveld W (eds) Van Periferie naar Kern, Foris, Dordrecht
Knuth DE (2000) Dancing links. arXiv preprint cs/0011047
Kubota Y, Levine R (2012) Gapping as like-category coordination. In: Béchet D, Dikovsky A (eds) Logical Aspects of Computational Linguistics, Springer, Nantes, Lecture Notes in Computer Science, vol 7351, pp 135–150
Lambek J (1958) The mathematics of sentence structure. American Mathematical Monthly 65:154–170
Luo Z (2012a) Common nouns as types. In: Logical aspects of computational linguistics (LACL2012), Springer, Lecture Notes in Artificial Intelligence, vol 7351
Luo Z (2012b) Formal semantics in modern type theories with coercive subtyping. Linguistics and Philosophy 35(6):491–513
Luo Z (2015) A Lambek calculus with dependent types. In: Types for Proofs and Programs (TYPES 2015), Tallinn
Mery B, Moot R, Retoré C (2013) Plurals: individuals and sets in a richly typed semantics. In: The Tenth International Workshop of Logic and Engineering of Natural Language Semantics 10 (LENLS10)

Mineshima K, Martínez-Gómez P, Miyao Y, Bekki D (2015) Higher-order logical inference with compositional semantics. In: Proceedings of Empirical Method for Natural Language Processing (EMNLP 2015)

Montague R (1970) Universal grammar. Theoria 36(3):373–398

Montague R (1974) The proper treatment of quantification in ordinary English. In: Thomason R (ed) Formal Philosophy. Selected Papers of Richard Montague, Yale University Press, New Haven

Moortgat M (2011) Categorial type logics. In: van Benthem J, ter Meulen A (eds) Handbook of Logic and Language, North-Holland Elsevier, Amsterdam, chap 2, pp 95–179

Moortgat M, Oehrle RT (1994) Adjacency, dependency and order. In: Proceedings 9th Amsterdam Colloquium, pp 447–466

Moot R (2008) Filtering axiom links for proof nets. Tech. rep., CNRS and Bordeaux University

Moot R (2010) Wide-coverage French syntax and semantics using Grail. In: Proceedings of Traitement Automatique des Langues Naturelles (TALN), Montreal, system Demo

Moot R (2012) Wide-coverage semantics for spatio-temporal reasoning. Traitement Automatique des Languages 53(2):115–142

Moot R (2015a) Grail. `http://www.labri.fr/perso/moot/grail3.html`, mature and flexible parser for multimodal grammars

Moot R (2015b) Grail light. `https://github.com/RichardMoot/GrailLight`, fast, lightweight version of the Grail parser

Moot R (2015c) Linear one: A theorem prover for first-order linear logic. `https://github.com/RichardMoot/LinearOne`

Moot R, Piazza M (2001) Linguistic applications of first order multiplicative linear logic. Journal of Logic, Language and Information 10(2):211–232

Moot R, Puite Q (2002) Proof nets for the multimodal Lambek calculus. Studia Logica 71(3):415–442

Moot R, Retoré C (2011) Second order lambda calculus for meaning assembly: on the logical syntax of plurals. In: Computing Natural Reasoning (COCONAT), Tilburg

Moot R, Retoré C (2012) The Logic of Categorial Grammars: A Deductive Account of Natural Language Syntax and Semantics. No. 6850 in Lecture Notes in Artificial Intelligence, Springer

Moot R, Schrijen X, Verhoog GJ, Moortgat M (2015) Grail0: A theorem prover for multimodal categorial grammars. `https://github.com/RichardMoot/Grail0`

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

Murawski AS, Ong CHL (2000) Dominator trees and fast verification of proof nets. In: Logic in Computer Science, pp 181–191

Oehrle RT (1994) Term-labeled categorial type systems. Linguistics & Philosophy 17(6):633–678

Pentus M (1997) Product-free Lambek calculus and context-free grammars. Journal of Symbolic Logic 62:648–660

Pogodalla S, Pompigne F (2012) Controlling extraction in abstract categorial grammars. In: de Groote P, Nederhof MJ (eds) Proceedings of Formal Grammar 2010–2011, Springer, LNCS, vol 7395, pp 162–177

Pustejovsky J (1995) The generative lexicon. M.I.T. Press

Ranta A (1991) Intuitionistic categorial grammar. Linguistics and Philosophy 14(2):203–239

Shieber S (1985) Evidence against the context-freeness of natural language. Linguistics & Philosophy 8:333–343

Valentín O (2014) The hidden structural rules of the discontinuous Lambek calculus. In: Casadio C, Coecke B, Moortgat M, Scott P (eds) Categories and Types in Logic, Language, and Physics: Essays dedicated to Jim Lambek on the Occasion of this 90th Birthday, no. 8222 in Lecture Notes in Artificial Intelligence, Springer, pp 402–420