Grail

An Automated Proof Assistant for Categorial Grammar Logics

Richard Moot

Utrecht institute of Linguistics OTS
Trans 10
3512 JK Utrecht
Richard.Moot@let.ruu.nl

Abstract

This paper gives an overview of the Grail system and its use as a tool for the development and prototyping of grammar fragments for categorial logics.

Grail is an automated theorem prover based on proof nets, a graph-based representation of proofs, and labeled deduction. The theorem prover is implemented in SICStus Prolog, the user interface in TclTk.

Though the underlying logic is decidable, and the theorem prover can operate automatically, user guidance is often desirable during the proof search. It can increase the performance of the algorithm and, more importantly, help the user visualize the status of the proof attempt thereby showing *why* a given statement is provable or not.

The Grail user interface is based on the Prolog debugger. At each proof step the user can take one of the following actions: *select* allows the user to select an inference step, *leap* performs automatic proof search until a proof is found, *fail* marks the current branch of the search tree as unsuccessful and *abort* abandons the entire proof attempt.

We have found the interface gives users better insight in the operation of the theorem prover and greatly enhances its facilities for prototyping and debugging of categorial grammars.

1 Introduction

Since 1958 [Lambek 58], linguists and logicians have been trying to specify grammars as logical theories (see e.g. [Morrill 94] or [Moortgat 97] for an overview). An advantage of this approach over more traditional approaches to linguistics is that we can prove our grammars have abstract properties like soundness, completeness and consistency.

In the logical framework we are proposing, the grammaticality of sentence should correspond to the derivability of a logical statement. More specifically a string of words $w_0, \ldots, w_n$ is a sentence of our language if and only if the lexicon l maps these words to formulas such that $l(w_0), \ldots, l(w_n) \vdash s$ is a theorem of our logic.

Grail is a tool which allows you to specify such logical theories together with a lexicon and see what kinds of sentences are derivable for the given logic.

2 Logical Background

Much of the inspiration for the Grail theorem prover comes from recent advances in linear logic [Girard e.a. 95] and labeled deduction [Gabbay 94], so I will briefly touch on these subjects.

2.1 Linear Logic

Linear logic, introduced in [Girard 87], was developed as a logic where the use of the structural rules of contraction and weakening, which apply freely in classical logic, is restricted. Without these structural rules the meaning of our connectives changes. Linear implication, written as '$\multimap$', is perhaps the clearest illustration of this. A formula $A \multimap B$ consumes or destroys an A formula in order to produce a B formula.

The sequent rules for (intuitionistic) linear implication are essentially the same as those of intuitionistic logic. Commutativity of the sequent comma is implicit.

Sequent Rules

$$\frac{}{A \vdash A}\ [Ax] \qquad \frac{\Delta \vdash A \quad \Gamma, A \vdash C}{\Gamma, \Delta \vdash C}\ [Cut]$$

$$\frac{\Delta \vdash A \quad \Gamma, B \vdash C}{\Gamma, \Delta, A \multimap B \vdash C}\ [L\multimap] \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\ [R\multimap]$$

Example 1 *If we assign the formula $np \multimap (np \multimap s)$ to a transitive verb like 'likes', we say it combines with two np formulas like those assigned to 'Nixon' or 'cigars' to produce an s formula.*

$$\frac{\dfrac{}{np \vdash np}\ [Ax] \quad \dfrac{\dfrac{}{np \vdash np}\ [Ax] \quad \dfrac{}{s \vdash s}\ [Ax]}{np, np \multimap s \vdash s}\ [L\multimap]}{np, np \multimap (np \multimap s), np \vdash s}\ [L\multimap]$$

This allows us to derive 'Nixon likes cigars' as a sentence, but not, in the absence of weakening, 'Nixon likes cigars special relativity', nor, in the absence of contraction, 'Nixon likes'. Unfortunately, as commutativity still applies, the current logic makes the incorrect linguistic prediction that any permutation of a sentence is also grammatical. We will remedy this in section 2.2

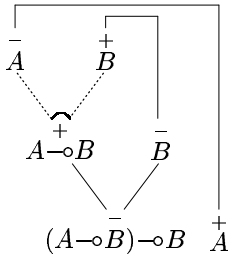
2.1.1 Proof Nets

A clear advantage of the (minimal) fragment of linear logic introduced above is that it has an exceptionally elegant proof theory, called proof nets.

For a given logical statement we can obtain a graph by taking the formula decomposition trees and linking atomic formulas to their negations using axiom links, which correspond to the sequent axiom rule. We call these graphs *proof structures*. Proof structures are a superset of proof nets and not all proof structures will correspond to sequent proofs. However, it turns out that we can distinguish proof nets from other proof structures by looking only at graph theoretic properties of the proof structure.

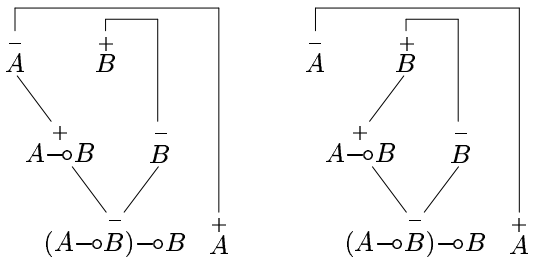
Linear implication is decomposed as follows, depending on whether it occurs in a positive (non-negated or succedent) or negative (negated or antecedent) position.

Example 2 *A proof structure for the sequent* $(A \multimap B) \multimap B \vdash A$.



From a proof structure we can obtain a *correction graph* by 'switching' each of the dotted links either to the left or to the right. A proof structure is a *proof net* if and only if all its correction graphs are acyclic and connected.

Example 3 *The proof structure of example 2 is* not *a proof net, as shown by the correction graph on the right.*



Proof search for a given logical statement consists merely of trying all linkings of atomic formulas. A practical downside is that for $2n$ atomic formulas there can be $O(n!)$ of these linkings, so while computing them all is possible, it is unlikely that it can always been done efficiently.

2.2 Categorial Grammars and Labeled Deduction

Although the proof net approach described above gives us a very elegant proof theory, it does so only for associative, commutative logics. For serious linguistics, however, we will want the structural rules of associativity and commutativity, like weakening and contraction, to be *optional*.

In categorial grammars we drop commutativity and associativity as global options. Without commutativity the connective $\multimap$ will split into two versions, depending on whether the implication looks for its argument to the left or right. We will write A/B (resp. $B\backslash A$) for the formula which yields an A when it finds a B to its right (resp. left).

Because different linguistic constructions may require access to different sets of structural rules, we will also use a multimodal logic. We will indicate the modes of connectives by using an index as subscript (e.g. $A/_iB$, $A\backslash_iB$).

In order to build upon the proof theoretic advances of linear logic, we embed the categorial logic into linear logic by means of labeling. Instead of using formulas A as our basic declarative unit we use labeled formulas $X : A$, where the label X represents structural information. The label assigned to the (single) succedent formula of the proof represents the way the formulas in the antecedent are organized. Labeled sequent rules for '/' are the following (the rules for '\' are symmetric).

$$\frac{\Delta \vdash Y : B \quad \Gamma, X \bullet_i Y : A \vdash Z : C}{\Gamma, \Delta, X : A/_iB \vdash Z : C}\ [L/] \qquad \frac{\Gamma, \mathbf{x} : B \vdash X \bullet_i \mathbf{x} : A}{\Gamma \vdash X : A/_iB}\ [R/]$$

Compared to the sequent rules for linear implication, we note that the only difference is in the labels. For the $[L/]$ rule, the label $X \bullet_i Y$ assigned to the result category A is a structured term indicating that the label Y computed for the formula B occurs to the direct right of structure label X assigned to formula $A/_iB$. The $[R/]$ rule says that in order to prove a formula $A/_iB$ we assign B a new label $\mathbf{x}$ and demand it occurs on the direct right of the structure label assigned to formula A.

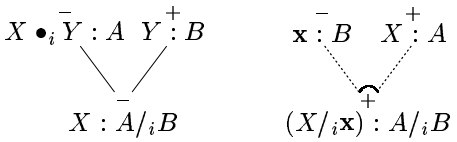
Structural rules operate on the labels only. For example, we can declare a mode c to be commutative by means of a structural rule which allows us to replace sublabels $X \bullet_c Y$ by $Y \bullet_c X$.

$$\frac{\Gamma \vdash Z[Y \bullet_c X] : C}{\Gamma \vdash Z[X \bullet_c Y] : C}\ [Com]$$

Example 4 *When we have a commutativity rule for mode c,* $\mathbf{x} : A/_c B \vdash \mathbf{x} : B\backslash_c A$ *is a theorem.*

$$\dfrac{\dfrac{}{\mathbf{y} : B \vdash \mathbf{y} : B}\ [Ax] \quad \dfrac{\dfrac{}{\mathbf{x} \bullet_c \mathbf{y} : A \vdash \mathbf{x} \bullet_c \mathbf{y} : A}\ [Ax]}{\mathbf{x} \bullet_c \mathbf{y} : A \vdash \mathbf{y} \bullet_c \mathbf{x} : A}\ [Com]}{\dfrac{\mathbf{x} : A/_c B, \mathbf{y} : B \vdash \mathbf{y} \bullet_c \mathbf{x} : A}{\mathbf{x} : A/_c B \vdash \mathbf{x} : A\backslash_c B}\ [R\backslash]}\ [L/]$$

Moortgat [Moortgat 97, section 7] proposes the following way to add labeling to proof nets (the cases for '\' are again symmetric).



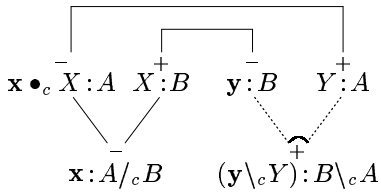
We can see the similarity between the labeled sequents and the labeled proof nets. Only the case for positive occurrences is somewhat different from the $[R/]$ rule: the label $X/_i\mathbf{x}$ should be seen as a constraint specifying that $\mathbf{x}$ should be a sublabel occurring on the immediate right of X. We check this constraint by means of a conversion:

$$(X \bullet_i Y)/_i Y \rightarrow_{R/} X$$

Similarly, we will have a label conversion for each of the structural rules.

Using this labeling we can generate an acyclic, connected proof structure, compute the label of the succedent formula and check if we can satisfy all constraints on this label by means of the conversion rules.

Example 5 *A proof net corresponding to the sequent proof above would look like*



The unifications $X := \mathbf{y}$ *and* $Y := \mathbf{x} \bullet_c X$ *will produce the label* $\mathbf{y}\backslash_c(\mathbf{x} \bullet_c \mathbf{y})$ *for the conclusion. We can then apply the conversions*

$$\mathbf{y}\backslash_c(\mathbf{x} \bullet_c \mathbf{y}) \rightarrow_{Com} \mathbf{y}\backslash_c(\mathbf{y} \bullet_c \mathbf{x}) \rightarrow_{R\backslash} \mathbf{x}$$

showing $\mathbf{x} : A/_c B \vdash \mathbf{x} : B\backslash_c A$ *is a theorem.*

3 The Grail Theorem Prover

3.1 Brief history

In the end of '95, the first incarnation of Grail was a piece of Prolog code of some 250 lines. You could enter a logical statement and wait until it produced an answer in the form of `yes` or `no`, or until you got bored (which happened a lot those days). In spite of a number of improvements to the efficiency of the original code, several grammar fragments designed in it could not handle longer sentences in a reasonable amount of time.

I therefore tried to give a user-friendly representation of the computation state with which the user can inspect and guide the computation. The benefits of this are twofold: firstly, the user can select a promising state from the possible states and abandon hopeless subgoals, and secondly, it gives the user insight into *why* specific statements are underivable, without having to use the Prolog debugger where tracing the execution is difficult even for the programmer.

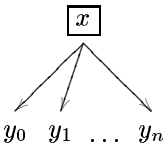
The current version is some 8000 lines of mixed Prolog and TclTk code, using the TclTk library included with SICStus Prolog. It can be used without knowledge about Prolog and produces output in human-friendly natural deduction format.

Grail is used as a research tool and as courseware for introductory to advanced level courses in categorial grammar.

3.2 Execution Model

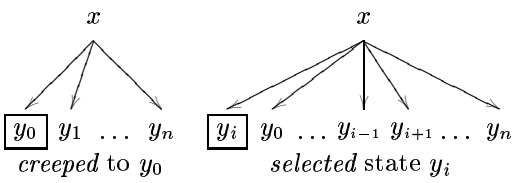
Though we work in a hybrid system of proof nets and labeling, the basic execution model of the interactive mode is the same in both cases. It is based on a simplified version of the Prolog debugger.

From the current state x we can apply a finite number of steps to get in a next state y_i. If no rules are applicable or the user selects *fail*, we backtrack to the first parent of the current state which still has unvisited daughter states and proceed from there. When the current state is a successful state (i.e. we have found a proof) we can either *abort* the computation or continue to search for more proofs.



When we select *creep* the proof continues with y_0 as its next state, as shown below. This step is nondeterministic; when no proof can be found for descendants of state y_0, computation will continue with state y_1.

The *select* step, of which creep is a special case, enables the user to reorder the goals, such that a state y_i of his choice will be the next state. When no proof can be found for descendants of this state, control is passed to the user who is then allowed to select a next state.



There are two other safe commands we can use: *leap*, after which Grail will compute until it has found the first solution and *nonstop* which will cause Grail to grind on until all proofs have been found.

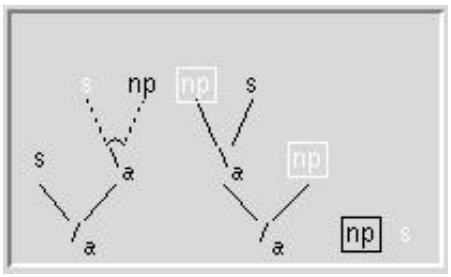
The final two commands are unsafe in that they can prevent proofs from being found by pruning the search space. *fail* will mark the current state as unsuccessful and continue computation on the next unvisited state, and *abort* will completely abandon the current proof attempt even if in leap or nonstop mode.

3.3 The Proof Net Window

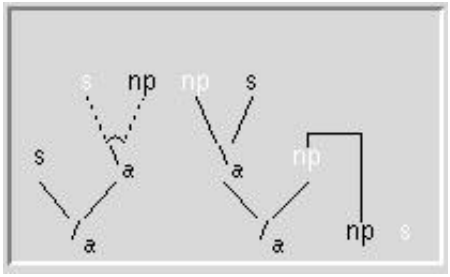
In the proof net window we see the current partial proof structure, with positive atomic formulas drawn in white and negative atomic formulas drawn in black. Here atomic formulas of opposite polarity are linked until we find a proof structure which is both acyclic and connected, after which we can try to satisfy the label constraints.

As an example we give a proof of the theorem $s/_a(np\backslash_a s), (np\backslash_a s)/_a np, np \vdash s$ which the lexicon would produce for 'Someone killed JFK'.

We first select an atom. This is a committed choice step; after we select the atom we are forced to link it before we can select another atom. After this selection all possible ways to link it by an axiom link will appear. As shown below, the selected *np* will appear in a black box and the atoms of opposite polarity which have not been tried before will appear in a white box. Each of these atoms will represent a possible next computation state and they will normally be searched from left to right. For the moment however, we use some user guidance and select the rightmost atom, knowing we can always try the other possibility later.

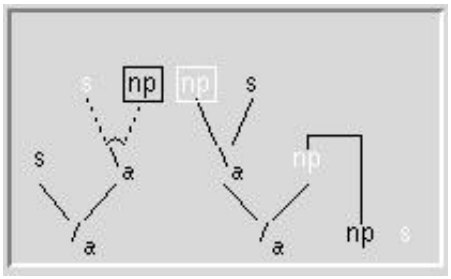


1. Selecting *np*

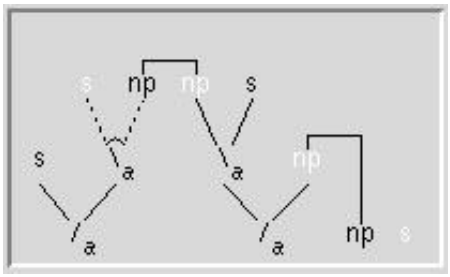


2. Linking

Next, we select the second negative *np*, which we are forced to link to the final *np*.

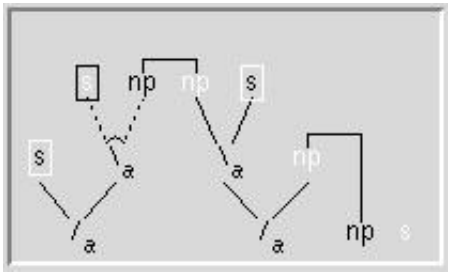


3. Selecting other *np*

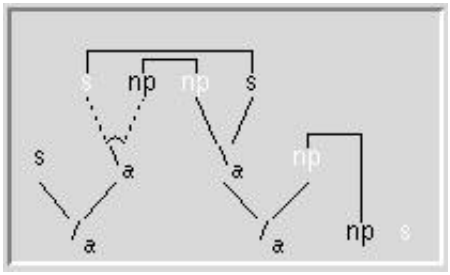


4. Linking

We select one of the *s* formulas. Linking it to the rightmost *s* would create a cycle with the switch set to the left and cause that branch of the search space to fail immediately, so we pick the other *s*.

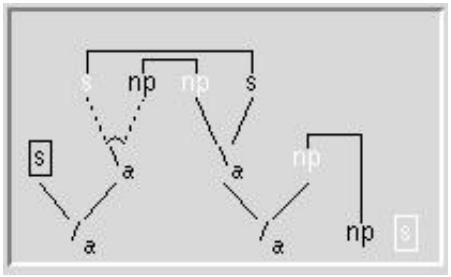


5. Selecting *s*

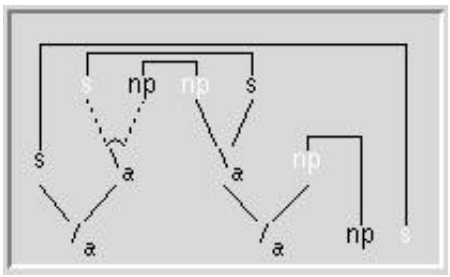


6. Performing noncyclic link

Finally, we connect the last two formulas.



7. Selecting final *s*



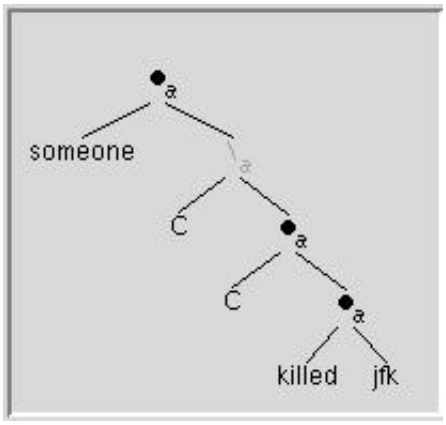
8. Done

3.4 The Rewrite Window

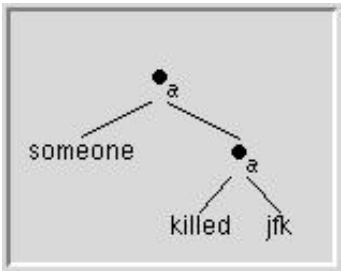
We check the label constraints by means of a very simple term rewrite system. Because a typical grammar fragment will have a number of rules (like e.g. commutativity) for which a naive algorithm would loop, Grail keeps track of visited labels in a closed set. Again, we have the option to either creep through Grail's built-in rewrite system, which uses breadth first search with local heuristic ordering, or to select the conversion we want to apply ourselves.

Grail will draw the current label as a tree. Next states will be all labels which can be obtained by a single rewrite step from the current label. Clicking on a node in the tree will cause a pop-up menu to appear with the possible conversion steps rooted at this node. Here we can select the conversion we want to apply, and Grail will keep track of the other possibilities.

For the example we used in the previous section, we will start with the label below.



The constructors denoting unsatisfied constraints, like the '$\backslash_a$' above, will be drawn in dark grey. In this case we can immediately convert this label to



which satisfies all constraints.

Typically there will be a large number of language specific structural rules in a grammar fragment and checking the label constraints will be a lot more complicated than the one step reduction above, making user interaction a valuable asset.

4 Conclusions

We have given an overview of the Grail interactive theorem prover and its underlying logical theory. Grail displays an intuitive representation of the state of the computation and allows the user to guide the computation by interacting with this representation.

On the proof net level, an advantage over sequent or natural deduction systems is that linking atomic formulas is a relatively trivial way to generate all proofs for a given statement. User guidance allows more experienced users to perform the axiom links they are interested in immediately, thereby sidestepping the $O(n!)$ complexity.

On the label rewrite level, it is often enlightening to see Grail (ab)use your carefully chosen structural rules in unintended ways, showing linguistically incorrect predictions of your logical theory, or to see it fail to satisfy a critical constraint, pointing to a missing or not sufficiently general structural rule. User interaction can considerably improve the performance by allowing the user to perform the intended label conversions himself.

Finally, though proof nets are in many ways an optimal proof theory for proof *search*, natural deduction is generally a better theory to *display* them. Therefore, source code which transforms the completed proof net into LaTeX natural deduction output is included with the release.

References

[BtM 97] Benthem, J. van, and A. ter Meulen (eds.), *Handbook of Logic and Language*, Elsevier, 1997.

[Gabbay 94] Gabbay, D. *Labeled Deductive Systems*, Report MPI-I-94-223, Max-Planck-Institut für Informatik, Saarbrücken, 1994.

[Girard 87] Girard, J.Y. *Linear Logic*, Theoretical Computer Science 50, 1987, pp. 1-102.

[Girard e.a. 95] Girard, J.Y., Y. Lafont and L. Regnier (eds.), *Advances in Linear Logic*, London Mathematical Society Lecture Notes, Cambridge University Press, 1995

[Lambek 58] Lambek, J. *The Mathematics of Sentence Structure*, American Mathematical Monthly 65, 1958, pp. 154-170.

[Moortgat 97] Moortgat, M. *Categorial Type Logics*, chapter 2 of [BtM 97].

[Morrill 94] Morrill, G. *Type Logical Grammar. Categorial Logic of Signs*, Kluwer, Dordrecht, 1994.