CHAPTER 8

AUTOMATED DEDUCTION

One of the attractive aspects of proof nets as discussed in the previous chapter is that they lend themselves well to automated proof search. First of all, in Section 7.6 we saw that we could eliminate cut formulas from proof nets, making it unnecessary to consider cut formulas in our proof search. Secondly, we can restrict ourselves to proof nets where all our axiomatic formulas are atomic, as indicated by the following lemma.

Lemma 8.1 *Given a proof structure $\mathcal{S}$ we can construct a proof structure $\mathcal{S}'$ with the same hypotheses and conclusions where all axiomatic formulas are atomic and where $\widehat{\mathcal{S}'} \twoheadrightarrow_\emptyset \widehat{\mathcal{S}}$. We will call such a proof structure* eta expanded.

Proof By induction on the total complexity of the axiomatic formulas.

If there are no complex axiomatic formulas in the proof structure, we take $\mathcal{S}' = \mathcal{S}$ and an empty conversion sequence.

If we have a proof structure $\mathcal{S}_0$ where the axiomatic formulas have $n+1$ total connectives, we can expand a complex axiomatic $A \bullet_i B$ formula as shown in Figure 8.1. The other connectives are treated similarly. The resulting proof structure $\mathcal{S}_1$ will have two new axiomatic formulas and the total number of connectives of axiomatic formulas will be n.

By induction hypothesis we know that $\widehat{\mathcal{S}'_1} \twoheadrightarrow_\emptyset \widehat{\mathcal{S}_1}$, so we can suffix a $[L\bullet_i]$ contraction producing the following conversion sequence.

$$\widehat{\mathcal{S}'_1} \twoheadrightarrow_\emptyset \widehat{\mathcal{S}_1} \overset{[L\bullet_i]}{\rightarrow} \widehat{\mathcal{S}_0}$$

As we use only contractions, the theorem holds regardless of the structural rules. □

The following corollary is an immediate consequence of Theorem 7.20 and Lemma 8.1.

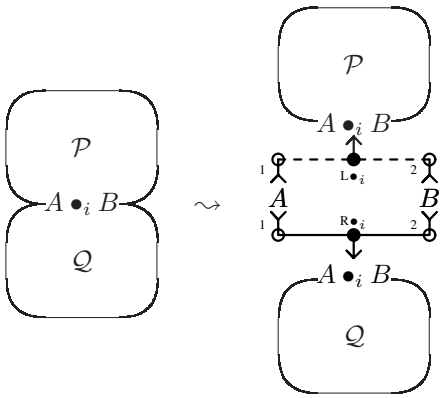

Figure 8.1: Eta expansion step for a $A \bullet_i B$ formula

Corollary 8.2 *For every proof net $\mathcal{P}$ of $\Gamma \vdash C$ there exists a proof net $\mathcal{P}'$, also of $\Gamma \vdash C$, which is cut free and eta expanded.*

So we can, without loss of generality, restrict ourselves to proof structures where all complex formulas are neither axiomatic nor cut formulas. A simple algorithm for the enumeration of cut free, eta expanded proof nets is shown in Table 8.1 on the next page.

We assume computation is nondeterministic, i.e. the steps of our algorithm can produce a number of solutions: the lexicon can produce different formulas for each word, there can be many different ways of identifying the atomic formulas and we might be able to convert our abstract proof structure to many different hypothesis trees. When one step in our algorithm fails to produce a solution, we backtrack to a previous step and try the next solution there until we have found all solutions.

The set of parameters on the final hypothesis tree can restrict the output of the algorithm in any of the following ways.

(i) left to right traversal of the hypothesis tree yields the formulas in the order indicated by the input sequence.

(ii) only binary modes from $I' \subseteq I$ and unary modes $J' \subseteq J$ can occur in the hypothesis tree.

(iii) return only the shortest conversion sequence(s).

When we use our algorithm for parsing a sentence, we typically want to satisfy condition (i). However, it can be useful to see *all* different hypothesis trees for the current proof structure because this might reveal ungrammatical sentences which are derivable with the current lexicon and structural rules.

Sometimes it makes sense to disallow certain modes from appearing in the final hypothesis tree, as indicated by condition (ii). We call a mode $i \in I'$ or $j \in J'$ *external* and a mode $i \in I \setminus I'$ or $j \in J \setminus J'$ an *internal* mode.

Input	– sequence $w_1, \ldots, w_n$ of words – lexicon l, which assigns formulas to words – set of goal formulas Q – set $\mathcal{R}$ of structural rules – set P of parameters restricting the shape of the final hypothesis tree
Output	set of cut free, eta expanded proof nets with hypotheses $l(w_1), \ldots, l(w_n)$ and conclusion $q \in Q$

(1) For each of the words w_i in the input sequence, select one of the formulas assigned to this word from the lexicon and select a $q \in Q$ as the conclusion.

(2) Decompose the formulas according to the links of Table 7.8 on page 108 until we reach the atomic subformulas. The disjoint union of these proof structures is itself a proof structure, though it will have several hypotheses in addition to those from the lexicon and several conclusions in addition to the goal formula.

(3) Identify each atomic premiss with an atomic conclusion to produce a proof structure with hypotheses $l(w_1), \ldots, l(w_n)$ and conclusion q.

(4) Convert the abstract proof structure corresponding to this proof structure to a hypothesis tree using only the structural conversions of $\mathcal{R}$ and the contractions.

(5) Check if this hypothesis tree conforms to our parameters P.

Table 8.1: Proof search algorithm for NL$\Diamond_{\mathcal{R}}$

Finally, parameter (iii) states that we are sometimes only interested in the shortest conversion sequence to a hypothesis tree. This should not be taken as a constraint on the derivability relation in the sense of 'shortest move' constraints proposed in minimalist frameworks (Chomsky 1995), but as a way of preferring conversion sequences without *redundant* structural conversions.

Throughout the next sections we will present some improvements over the initial, naive algorithm of Table 8.1. These improvements can be categorized as follows.

[Compilation] This is a standard programming technique where predictable computation steps are done in advance and the results stored or where we collapse several simple steps into a single derived step. In the context of declarative programming languages compilation is sometimes called *partial execution* (Pereira & Shieber 1987). We will apply partial execution to the current problem in Section 8.2, where we will store abstract proof structures in the lexicon, eliminating step 2 from the algorithm and in Section 8.6 where we compile multiple par

contractions into a single, derived contraction.

[Divide and Conquer] This refers to the basic technique of solving a problem by dividing it in several simpler problems in such a way that a solution to all these simple problems is a solution to the complete problem as well. We will use this strategy in Sections 8.5 and 8.9 where we will use components as a natural way of restricting structural rule applications.

[Early Failure] In constraint programming (Dechter 2000), it is often possible to get good performance on computationally intractable problems. This is done by strategies for realizing, as early as possible, that the current choices we have made will never lead us to a solution. While early failure may take the form of simple, deterministic tests, it can sometimes also consist of doing computations as early as possible. We will see examples of this in Sections 8.1, 8.3, 8.4 and 8.7.

[Parallel Computation] We will develop a way of performing the structural conversions in parallel in Section 8.8.

A general trade-off we will see is that we can sacrifice generality or completeness for efficiency. Some of the most powerful heuristics mentioned in this chapter function only for restricted fragments of NL$\Diamond_{\mathcal{R}}$ and in the next chapter we will see that only quite restricted fragments of NL$\Diamond_{\mathcal{R}}$ are decidable in polynomial time.

8.1 Invariants

As connecting the atomic formulas and the structural conversions are computationally expensive, it is desirable to do some static tests on the set of proof structures we get from the lexical formulas after the unfolding stage of the algorithm to make sure we at least have a chance of ultimately converting to a hypothesis tree. The following are two simple tests to reject proof structures which can never satisfy our correctness criterion.

First, by our definition of hypotheses and conclusions of proof structures, all atomic formulas other than lexical formulas or the conclusion must be both a premiss and a conclusion of some link in a proof structure with hypotheses $l(w_1), \ldots, l(w_n)$ and conclusion q. So we can count if each of these atomic formulas occurs as many times as a conclusion as it occurs as a premiss. This is sometimes called the *count check* (van Benthem 1986).

Secondly, the following lemma, suggested to me by Quintijn Puite, gives us a condition on the number of binary links occurring in a proof net.

Lemma 8.3 *Suppose we have a proof structure $\mathcal{S}$ with h hypotheses, t binary tensor links, p binary par links and a single conclusion. Then the following holds if $\mathcal{S}$ is a proof net.*

$$t + 1 = p + h$$

Proof Reasoning backwards from the hypothesis tree to the initial hypothesis structure we see that it holds for the hypothesis tree (with $p = 0$), that the structural conversions and the unary contractions preserve t, p and h and that the contractions for the binary links increase t and p simultaneously. □

We cannot use the same reasoning for unary connectives; though the contractions for the unary connectives remove one unary tensor and one unary par link, even in the case without structural rules we can state only that $p_1 \leq t_1$, where p_1 is the number of unary par links and t_1 the number of unary tensor links. This is because there can be an arbitrary number of unary tensor links in the final hypothesis tree. In the presence of structural rules, which possibly increase the number of unary tensor links, there is little we can tell simply from counting the unary links.

8.2 Compiling the Lexicon

Instead of have a lexicon which consists of formulas, we can compile the formulas of the lexicon to proof structures, which we can further compile to abstract proof structures, where we keep track of the output formulas of every abstract proof structure. We can denote this by using square brackets, for example. A formula between square brackets is then a 'true' hypothesis or conclusion of the abstract proof structure, we will call it a *bound* formula. The other formulas are atoms which will disappear after they are used for axiom connections, so they are 'temporary' hypotheses and conclusions, we will call them *free* formulas.

This is only necessary so we can distinguish between an atomic formula a used a hypothesis and an atomic formula a used as a conclusion, the former looking like shown below on the left, the latter looking like shown below on the right.

$$\begin{array}{c}[a]\\ \vdots \\ a\end{array} \qquad\qquad \begin{array}{c}a\\ \vdots \\ [a]\end{array}$$

Alternatively, we can say that the hypotheses and conclusions of a proof structure or an abstract proof structure are the conclusion of 'hypothesis' links or the premiss of 'conclusion' links respectively.

In the depiction of the proof structures, this will have the advantage that all axiomatic formulas are now the active formula of two links, by symmetry with the cut formulas, which are the main formula of two links. It is often of mnemonic value to use the word w to which this lexical proof structure is

assigned instead of 'Hyp' as the label of of a hypothesis link. We'll give an example of this in Section 8.4.

Identifying two vertices by means of an axiom connection is possible in an abstract proof structure if one has the formula F as a free premiss and the other has the formula F as a free conclusion. Both formulas will disappear after the axiom connection.

8.3 Acyclicity and Connectedness

If we translate formulas and antecedent terms of NL$\Diamond_{\mathcal{R}}$ to formulas and antecedent terms of multiplicative intuitionistic linear logic as follows

$$\begin{aligned}
\|a\| &= a \\
\|\Diamond_j A\| &= \|A\| \\
\|\Box_j^{\downarrow} A\| &= \|A\| \\
\|A/_i B\| &= \|B\| \multimap \|A\| \\
\|B\backslash_i A\| &= \|B\| \multimap \|A\| \\
\|A \bullet_i B\| &= \|A\| \otimes \|B\|
\end{aligned}$$

$$\begin{aligned}
\|\langle \Gamma \rangle^j\| &= \|\Gamma\| \\
\|\Gamma \circ_i \Delta\| &= \|\Gamma\|, \|\Delta\|
\end{aligned}$$

then every derivable sequent of NL$\Diamond_{\mathcal{R}}$ corresponds to a derivable sequent of MILL.

We define switchings and correction graphs for abstract proof structures of NL$\Diamond_{\mathcal{R}}$ analogous to the way we did for L-proof structures in Section 7.8. As we already noted in Section 4.5, we have linear time algorithms for checking whether all correction graphs of a proof structure are acyclic and connected. Because of this, it seems prudent to make an acyclicity and connectedness test before trying to convert the abstract proof structure to a tree.

In many cases, we can already see during the stage where we are connecting the axioms that, no matter how we continue, we will never produce an acyclic and connected abstract proof structure.

If an abstract proof structure $\mathcal{A}$ has a substructure with a cyclic correction graph, then $\mathcal{A}$ will have a cyclic correction graph too.

Similarly, if an abstract proof structure $\mathcal{A}$ has a correction graph with disconnected substructures $\mathcal{A}_1, \ldots, \mathcal{A}_n$ then every disconnected substructure must have a free formula at at least one of its vertices, otherwise it will be impossible to produce a connected correction graph for $\mathcal{A}$ even after we perform further axiomatic connections.

8.4 Axiomatic Connections

The algorithm, as shown in Table 8.1 does not specify anything about the order in which we perform the axiomatic connections in step 3. From a logical point of view the order in which we connect the axioms does not matter, but

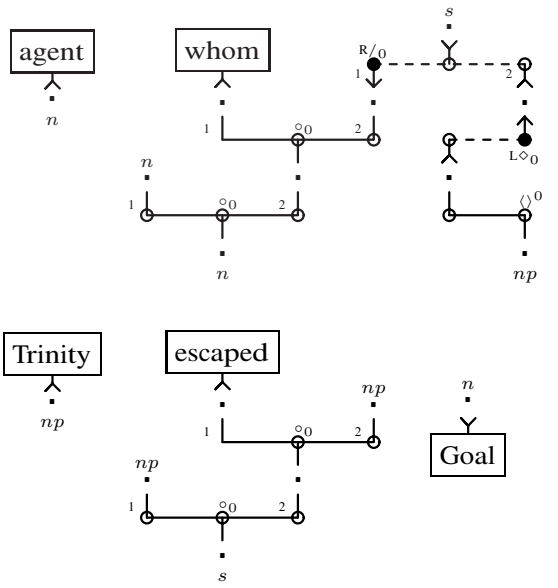

Figure 8.2: Lexical aps's for 'agent whom Trinity escaped'

from a computational point of view it is useful to keep the principles of early failure in mind and always connect the axiom which is the most restricted in its possibilities. This will make the information in the proof structure more explicit, which in turn can trigger other early failure mechanisms.

Let's look at an example. Figure 8.2 shows the lexical abstract proof structures for 'agent whom Trinity escaped' according to the lexicon of Table 7.7 on page 106.

We see one s conclusion and one s premiss, two np conclusions and two np premisses, and two n conclusions and two n premisses. In this case, there is only one possible way of connecting the s formulas, so this is the preferred connection, resulting in the abstract proof structure shown in Figure 8.3.

After this connection, some information which was implicit in Figure 8.2 has become explicit, for example that, unless some structural conversion operates on this abstract proof structure, the word 'whom' will precede the word 'escaped' in the final hypothesis tree. We'll see in Section 8.7 how to exploit this kind of word order information.

At the current stage, superficially, it doesn't matter if we decide to link the np's or the n's, because in both case we have to consider two possibilities. However, should we choose to link the n conclusion attached to 'agent' to the n premiss of the goal formula, we would produce a disconnected abstract proof structure. So, if we take the acyclicity and connectedness criterion discussed in Section 8.3 into account, we have only one way of connecting the four n formulas, namely as shown in Figure 8.4.

Finally, we connect the np formulas. We have two possibilities here, depending on where we connect the np conclusion corresponding to 'Trinity'

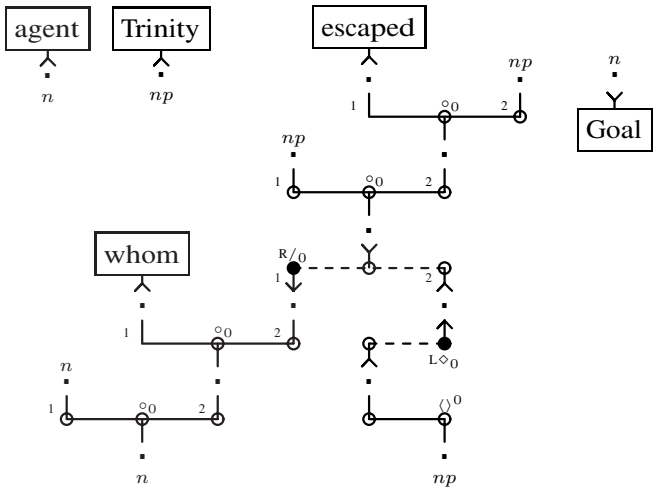

Figure 8.3: Abstract proof structure after the s connection

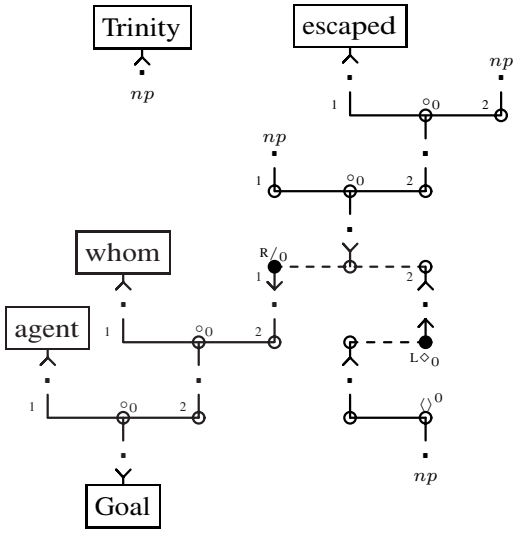

Figure 8.4: Abstract proof structure after the n connections

to the left or right premiss. In this case, only the first possibility allows the resulting abstract proof structure to convert to a hypothesis tree, as shown in Example 7.13

In the worst case, with n premisses and n conclusions, we might have to consider all $n!$ different connections, but the strategy of first connecting the atomic formula with the smallest number of possible candidates for connection appears to be quite powerful.

Another strategy to perform the axiom links would be to perform them

incrementally from left to right, starting with the first word of the sentence and trying to make as many connections as possible after each new word. This has been independently proposed by Johnson (1998) and by Morrill (1998) (2000). Both authors present evidence that the number of unconnected atomic formulas in a proof structure corresponds to the relative difficulty a human would have when processing the sentence, giving some surprising psycholinguistic support for the use of proof nets in linguistic analysis. I want to avoid making any claims about the psychological reality of the current, opportunistic literal selection strategy. Connecting the operation of an automated theorem prover to psychological processes is, in my opinion, neither necessary nor desirable.

Finally, the axiom connections are related to strategies for resolution-based theorem provers. Eisinger & Ohlbach (1993) give a good overview of different literal selection strategies.

8.5 Components

Components, which we introduced in Section 7.6 to prove cut elimination, have some good properties we can use for automated deduction.

Recall from Definition 7.17 that a component of an abstract proof structure $\mathcal{A}$ is a maximally connected substructure with respect to the tensor links of $\mathcal{A}$. Structural conversions operate on one component only, and leave all other components unchanged. Contractions operate on one component, in which they erase a tensor link, then merge this component with another.

Definition 8.4 *If a component $\mathcal{C}$ of an abstract proof structure $\mathcal{A}$ does not contain vertices which are the output vertex of a par link, but*

(i) *either $\mathcal{C} = \mathcal{A}$, that is, there are no par links in $\mathcal{A}$*

(ii) *or both input vertices of a par link are in $\mathcal{C}$*

we call the component active. *Otherwise, we will call it* waiting,

Similarly, if all input vertices of a par link are in the same component, we will call this link active. *Otherwise, we will call it* waiting.

Example 8.5 *The components in the abstract proof structure of Example 7.10 are drawn in black in Figure 8.5 on the following page. Note that the top component consists of only a single vertex.*

In this abstract proof structure there is only one active component, the middle one, and only the $[L\Diamond_0]$ link is active.

Lemma 8.6 *We can restrict ourselves to conversion sequences of the following form.*

(1) Apply a number of structural conversions in an active component.

(2) If the abstract proof structure still has par links, contract an active par link of which the inputs are in the current component, reassess the active components and continue from 1.

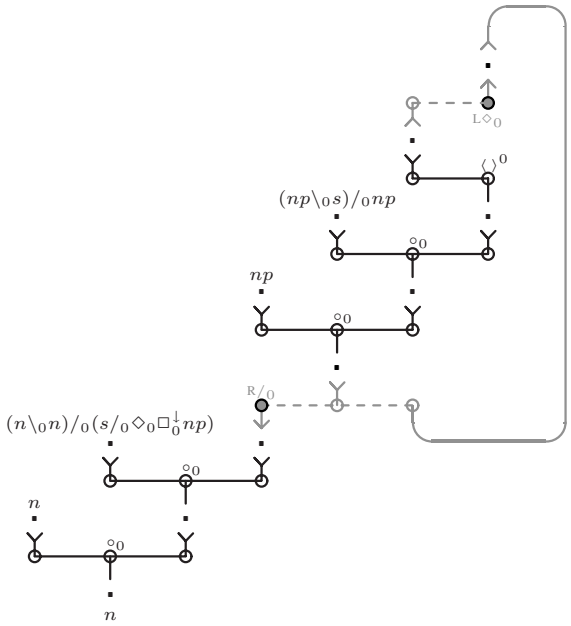

Figure 8.5: The components of Figure 7.9

Proof As noted before, the different components of an abstract proof structure are independent until they are united into one component by a contraction. As the contractions are defined to operate by contracting a par link of which all active vertices are connected to the same tensor link, this means the first contraction in any abstract proof structure must be a par link of which the active vertices are already in the same component.

Because we also prohibit the active component from containing the main vertex of a par link, no contractions other than the contraction of one of the active par links from this component will affect the current component. □

Some remarks need to be made. When an abstract proof structure has multiple active components, we can apply structural rules to them independently; we can start with any active component we want or we can even operate on all active components in parallel.

However, when a component has multiple active par links, it is possible that we are in the following situation: if we contract one before the other, we can convert the abstract proof structure to a hypothesis tree, but if we contract them in the other order, conversion to a hypothesis tree may not be possible. Figure 8.6 on the next page presents two examples of such situations, which occur in the base logic NL$\diamond$.

On the left of the figure, both the $[L\diamond_0]$ and the $[R\square_0^{\downarrow}]$ link are active, but if we contract the $[L\diamond_0]$ link first, we will be unable to contract the $[R\square_0^{\downarrow}]$ link. On the other hand, if we contract the $[R\square_0^{\downarrow}]$ link first, we produce a redex for the $[L\diamond_0]$ contraction immediately.

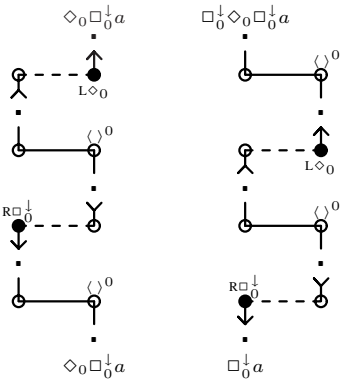

Figure 8.6: Conflicts between two active par links

On the right of the figure, we have the same active component, with the same active par links, only now we are in the opposite situation: contracting the $[L\Diamond_0]$ link is required before contracting the $[R\Box_0^{\downarrow}]$ link if we want to convert to a hypothesis tree.

Definition 8.7 *A component is* completed *if none of its vertices are labeled with free formulas.*

When a component is both completed and active, further axiomatic connections will not be relevant for that component, at least not until one of the active par links bordering it will be contracted, which may cause the new component to be incomplete again. Therefore, when we are performing the axiom connections, we have the possibility of performing conversions on active, completed components whenever we produce them. However, it may not necessarily be the best strategy to contract par links bordering completed components whenever we encounter them. We have to be careful which strategy we prefer: do we choose eager evaluation, that is contract par links as soon as the component it is attached to is completed, or lazy evaluation, that is wait with contracting par links as much as possible.

The advantage of eager evaluation appears to be that we can detect par links which are not contractable at an early stage, thereby triggering early failure mechanisms and possible preventing unnecessary computations.

However, it is possible lazy evaluation gives better performance. Continuing connecting axioms may fail with respect to other early failure mechanisms which are computationally less expensive. Waiting may also produce other active, completed components which are smaller or more likely to fail. Finally, with respect to eager evaluation it is difficult to decide between multiple, active par links and making the wrong choice can lead to a dead end in the search space.

The Grail automated theorem prover discussed in Appendix A gives you the choice of performing eager or lazy evaluation of par links. Section A.3.9

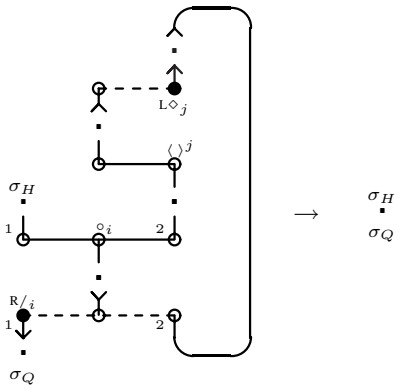

Figure 8.7: A combined $/_i\Diamond_j$ contraction

gives details on how to set up these parameters.

8.6 Focusing

One of the insights of focusing proofs (Andreoli 2000) is that multiple par links which are connected in such a way that the active formula of one par link is the main formula of the other, can be executed at the same time. We will call such par links *consecutive* par links. In Figure 8.5 on page 142 the $[L\Diamond_0]$ and the $[R/_0]$ link are consecutive par links. We can treat these configurations as if they were a single logical operator. In the case of '$/_0\Diamond_0$' its logical rules would be the following.

$$\frac{\Gamma[A] \vdash C \quad \Delta \vdash B}{\Gamma[A/_0\Diamond_0 B \circ_0 \langle\Delta\rangle^0] \vdash C}\ [L/_0\Diamond_0] \qquad \frac{\Gamma \circ_0 \langle B\rangle^0}{\Gamma \vdash A/_0\Diamond_0 B}\ [R/_0\Diamond_0]$$

The contraction corresponding to this combination is shown in Figure 8.7.

A similar strategy is not possible with a tensor/par combination. For example, though we can add the following rules for $\Diamond_0\Box_0^\downarrow$ to our sequent calculus

$$\frac{\Gamma[A] \vdash C}{\Gamma[\Diamond_0\Box_0^\downarrow A] \vdash C}\ [L\Diamond_0\Box_0^\downarrow] \qquad \frac{\langle\Gamma\rangle^0 \vdash A}{\langle\Gamma\rangle^0 \vdash \Diamond_0\Box_0^\downarrow A}\ [R\Diamond_0\Box_0^\downarrow]$$

these rules will be *incomplete*, that is, there are valid derivations with formulas of the form $\Diamond_0\Box_0^\downarrow A$ which you will not find using the combined rules above, but which you will find using a separate $\Diamond$ and $\Box^\downarrow$ rule. In the example above, applying the $[L\Diamond_0]$ rule instead of the combined rule would possibly open up new structural rules. Example 7.4 on page 106 shows such a situation. If we look at the sequent rules of Table 7.5 on page 105, we see that from a forward chaining proof search perspective the tensor rules add

structural information, whereas the par rules remove it under certain conditions.

With respects to the components of an abstract proof structure, consecutive par links will correspond to components consisting of a single vertex, see for example Figure 8.5. Now, it is possible to extend the definition of active par links to active consecutive par links.

Definition 8.8 *Consecutive par links are* active *whenever all the inputs to the tree of par links are in the same component.*

Lemma 8.9 *We can restrict ourselves to conversion sequences of the following form.*

(1) Apply a number of structural conversions in an active component.

(2) If the abstract proof structure still has par links, contract a tree of active consecutive par links of which all leaves are in the current component, reassess the active components and continue from 1.

Proof According to Lemma 8.6 we can restrict ourselves to conversion sequences where we apply structural conversions in an active component, then contract an active par link attached to the current component. Now let $\mathcal{C}$ be an active component and L be an active par link which is part of an active consecutive par link. Performing this par contraction will remove a tensor link from the active component and if the par link was part of a consecutive par link, the component will now be attached to a single vertex. This means any structural conversion which is possible after this contraction was already possible before this contraction as well, since the new component is a proper substructure of the component just before the contraction. □

8.7 Word Order

There are cases where we can see from the abstract proof structure we are constructing that it will never convert to an abstract proof structure where the words of the input sequence occur in the right order.

For example, the abstract L-proof structures we introduced in Section 7.8 have only a structural rule representing associativity, and no structural rules which could change the order of the hypotheses to the abstract proof structure. This means that whenever we make a axiomatic connection which does not respect the order of the words in the input sentence, we will never be able to contract the abstract L-proof structure to a single hypothesis comb. Also, when the second conclusion of $[R/]$ link is connected to anything but the last hypothesis of a comb, contracting it will never be possible. Similar arguments can be made for the other par links.

This property is a reflex in our proof net calculus of the planarity condition we discussed in Section 4.7. We call binary modes for which either no structural rules or one or both of the associativity structural rules apply *continuous*.

A way of enforcing planarity for some modes but not for others is by first order approximation. In Section 5.1.4 we gave embedding results for both L and LP into the first order multiplicative fragment of linear logic. By giving every continuous mode the L translation and every discontinuous mode the LP translation, we have a simple way of enforcing at least some of the word order constraints on derivation. We can also imagine giving certain modes a slightly more sophisticated translation, such as those suggested in Section 5.2.2 for relative pronouns. Morrill (1999) gives similar suggestions for using first order constructs to enforce word order constraints.

In some cases, we can also use an eager evaluation strategy for performing the structural conversions and put the words in the right order with respect to eachother whenever possible; each time we perform an axiom connection between two disjoint abstract proof structures, we merge the words of the two abstract proof structures until the new aps has all words in the right order again. Again, eager evaluation can be dangerous because it can force us to select the wrong alternative or because it can be impossible to put two words in the right order until they are put in a bigger context.

The Grail automated theorem prover of Appendix A gives you the choice of how to evaluate the word order constraints, see Section A.3.9 for details on how to set up these parameters.

8.8 Parallel Computation

Though we have already seen that it is possible to apply structural rules in parallel in different active components, in this section we will see that if we represent a component appropriately, we can apply structural rules in parallel in the *same* component as well.

This is done simply by allowing every vertex to be the conclusion and the premiss of more than one link. We call such a structure a parallel abstract proof structure. This makes it a quite a bit harder to represent the abstract proof structures in an orderly way. I will choose to represent parallel aps's by just listing the links, in what I will call the distributed representation of a parallel aps, even though this will mean vertices can occur in multiple places. I think the alternative, having every vertex fixed and drawing the links between them, will lead to unnecessarily cluttered figures.

Example 8.10 *Let's look at the abstract proof structure corresponding to the sequent* $a/_a a, a/_a a, a/_a a \vdash a/_a a$ *shown in Figure 8.8 on the next page, where we assume mode* a *is associative but not commutative.*

When we present this abstract proof structure as a parallel aps, it will look as shown in Figure 8.9 on the facing page. The links are named L_1 *to* L_4 *for future reference.*

We perform structural conversions on parallel abstract proof structures as follows.

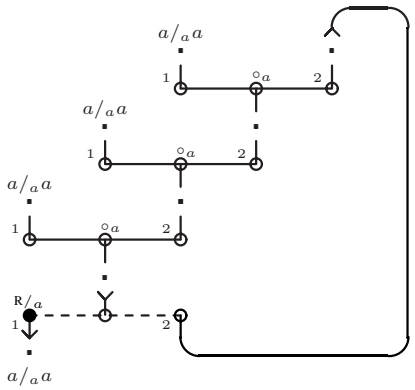

Figure 8.8: Abstract proof structure for $a/_a a, a/_a a, a/_a a \vdash a/_a a$

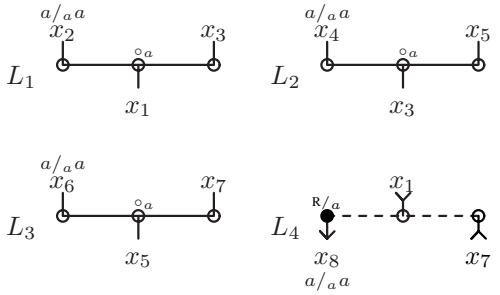

Figure 8.9: The aps of Figure 8.8 in a distributed representation

– find all sets of connected tensor links which are the redex of a structural conversions.

– using fresh internal vertices, add the reduct of the structural conversion to the parallel aps unless the reduct is equivalent, up to renaming of the internal vertices, with links which are already present in the parallel aps.

Example 8.11 *Using one application of the associativity rule, we can expand the parallel aps of Figure 8.9 with the links shown in Figure 8.10 on the next page. From L_1 and L_2 associativity gives us L_5 and L_6, and we can reassociate L_2 and L_3 into L_7 and L_8.*

For the next generation, we know that at least one of the links to which we apply a structural rule must have been introduced in the last generation, because this is the only way we can produce new links.

The third generation tensor links are shown in Figure 8.11 on the following page. L_9 and L_{10} have been obtained from L_3 and L_5, whereas L_{11} and L_{12} have been obtained from L_1 and L_7. The redexes formed by L_5 and L_6 and by L_7 and L_8 have not been triggered, because their reducts would be equivalent up to renaming of the

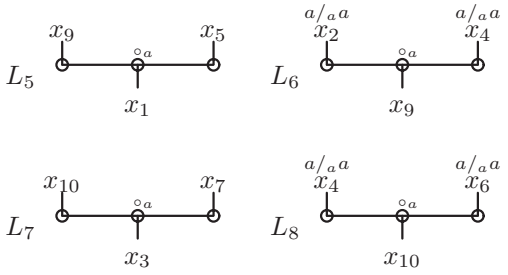

Figure 8.10: Second generation tensor links

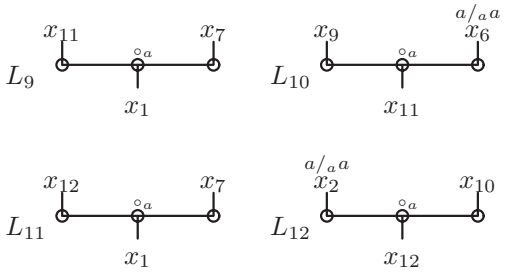

Figure 8.11: Third generation tensor links

internal nodes to L_1 and L_2 and to L_2 and L_3 respectively.

We have completed the structural rule applications: no structural conversion from the current state will produce new links. We are now in a position to apply the $[R/_a]$ contraction, which will identify vertices x_8, x_{11} and x_{12} and which will erase all links which have become unreachable from the root vertex x_8. The resulting parallel aps is shown in Figure 8.12.

For the current example, the contraction system described in Section 7.8 is, of course, much more efficient. However, the setup described here is very general and works for any set of structural rules

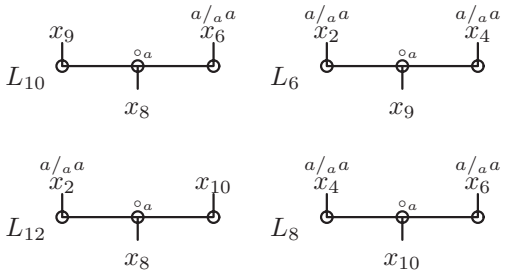

Figure 8.12: Parallel aps after the $[L/_a]$ contraction

The methodology outlined in this section is very close to finding a canonical model according to the Kripke semantics discussed in Section 3.6. The two frame constraints for an associative mode a would be the following.

$$\forall x_1.x_2.x_3.x_4.x_5.\ x_1R_ax_2x_3 \wedge x_3R_ax_4x_5 \rightarrow \exists x_6.\ x_1R_ax_6x_5 \wedge x_6R_ax_2x_4$$
$$\forall x_1.x_2.x_3.x_4.x_5.\ x_1R_ax_2x_5 \wedge x_2R_ax_3x_4 \rightarrow \exists x_6.\ x_1R_ax_3x_6 \wedge x_6R_ax_4x_5$$

When we interpret xR_iyz as there is a tensor link of mode i with premisses y and z and conclusion z, all we do when applying the structural rules to a parallel aps is to add tensor links and vertices which must exists according to the frame constraints.

An interesting possibility to investigate would be to extend parallel computation beyond single components, possibly even extending it to compute different lexical assignments to words of the input sentence in parallel and compare the time and space complexity with the sequential version of Table 8.1.

8.9 Rule Filtering

In many cases we can see from the shape of the structural conversions that some structural conversions can never produce a redex for the contraction of the active par link we are looking at.

Example 8.12 *If we want to add features for person, number, gender and case to a simple English grammar, in the style of Heylen (1999), we might do this as shown in Table 8.2. The features on the lexical entry for 'he' show it is a 3rd person singular masculine nominative pronoun. For every feature we have a 'top' and and a 'bottom' element. For example the gender feature has, in addition to values m for masculine, f for feminine and n for neuter a value g which includes any gender feature and a value G which is included by any gender feature, as stated by the following structural rules, where $i \in \{m, f, n\}$.*

$$\frac{\Gamma[\langle\Delta\rangle^G] \vdash C}{\Gamma[\langle\Delta\rangle^i] \vdash C}\ [i,G] \qquad \frac{\Gamma[\langle\Delta\rangle^i] \vdash C}{\Gamma[\langle\Delta\rangle^g] \vdash C}\ [g,i]$$

In the lexicon of Table 8.2, 'they' is assigned gender feature G which means it can satisfy any gender requirement from the verb. On the other hand, 'smiles' selects for a subject with gender feature g which means any marking for gender will satisfy it. An example derivation of 'Marla smiles' is shown in Figure 8.13.

In the example above, we only have structural rules for inclusion and none for interaction. The abstract proof structure for this derivation, where, for the sake of simplicity, we abstract over the $\Box_3^{\downarrow}$ and $\Box_{sg}^{\downarrow}$ connectives, is shown in Figure 8.14.

$$
\begin{array}{ll}
l(\textit{he}) & = \Box_3^{\downarrow}\Box_{sg}^{\downarrow}\Box_m^{\downarrow}\Box_{nom}^{\downarrow}np \\
l(\textit{she}) & = \Box_3^{\downarrow}\Box_{sg}^{\downarrow}\Box_f^{\downarrow}\Box_{nom}^{\downarrow}np \\
l(\textit{it}) & = \Box_3^{\downarrow}\Box_{sg}^{\downarrow}\Box_n^{\downarrow}\Box_C^{\downarrow}np \\
l(\textit{him}) & = \Box_3^{\downarrow}\Box_{sg}^{\downarrow}\Box_m^{\downarrow}\Box_{acc}^{\downarrow}np \\
l(\textit{her}) & = \Box_3^{\downarrow}\Box_{sg}^{\downarrow}\Box_f^{\downarrow}\Box_{acc}^{\downarrow}np \\
l(\textit{they}) & = \Box_3^{\downarrow}\Box_{pl}^{\downarrow}\Box_G^{\downarrow}\Box_{nom}^{\downarrow}np \\
l(\textit{them}) & = \Box_3^{\downarrow}\Box_{pl}^{\downarrow}\Box_G^{\downarrow}\Box_{acc}^{\downarrow}np \\
l(\textit{Marla}) & = \Box_3^{\downarrow}\Box_{sg}^{\downarrow}\Box_f^{\downarrow}\Box_C^{\downarrow}np \\
l(\textit{smiles}) & = \Box_3^{\downarrow}\Box_{sg}^{\downarrow}\Box_g^{\downarrow}\Box_{nom}^{\downarrow}np\backslash s \\
l(\textit{hates}) & = (\Box_3^{\downarrow}\Box_{sg}^{\downarrow}\Box_g^{\downarrow}\Box_{nom}^{\downarrow}np\backslash s)/\Box_p^{\downarrow}\Box_n^{\downarrow}\Box_g^{\downarrow}\Box_{acc}^{\downarrow}np
\end{array}
$$

Table 8.2: A lexicon with feature information

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{np\vdash np}\,[Ax]}{\langle\Box_C^{\downarrow}np\rangle^C\vdash np}\,[L\Box_C^{\downarrow}]}{\langle\Box_C^{\downarrow}np\rangle^{nom}\vdash np}\,[\textit{nom,C}]}{\Box_C^{\downarrow}np\vdash\Box_{nom}^{\downarrow}np}\,[R\Box_{nom}^{\downarrow}]}{\langle\Box_f^{\downarrow}\Box_C^{\downarrow}np\rangle^f\vdash\Box_{nom}^{\downarrow}np}\,[L\Box_f^{\downarrow}]}{\langle\Box_f^{\downarrow}\Box_C^{\downarrow}np\rangle^g\vdash\Box_{nom}^{\downarrow}np}\,[\textit{g,f}]}{\Box_f^{\downarrow}\Box_C^{\downarrow}np\vdash\Box_g^{\downarrow}\Box_{nom}^{\downarrow}np}\,[R\Box_g^{\downarrow}]}{\langle\Box_{sg}^{\downarrow}\Box_f^{\downarrow}\Box_C^{\downarrow}\rangle^{sg}np\vdash\Box_g^{\downarrow}\Box_{nom}^{\downarrow}np}\,[L\Box_{sg}^{\downarrow}]}{\Box_{sg}^{\downarrow}\Box_f^{\downarrow}\Box_C^{\downarrow}np\vdash\Box_{sg}^{\downarrow}\Box_g^{\downarrow}\Box_{nom}^{\downarrow}np}\,[R\Box_{sg}^{\downarrow}]}{\langle\Box_3^{\downarrow}\Box_{sg}^{\downarrow}\Box_f^{\downarrow}\Box_C^{\downarrow}np\rangle^3\vdash\Box_{sg}^{\downarrow}\Box_g^{\downarrow}\Box_{nom}^{\downarrow}np}\,[L\Box_3^{\downarrow}]}{\Box_3^{\downarrow}\Box_{sg}^{\downarrow}\Box_f^{\downarrow}\Box_C^{\downarrow}np\vdash\Box_3^{\downarrow}\Box_{sg}^{\downarrow}\Box_g^{\downarrow}\Box_{nom}^{\downarrow}np}\,[R\Box_3^{\downarrow}]\qquad\dfrac{}{s\vdash s}\,[Ax]}{\Box_3^{\downarrow}\Box_{sg}^{\downarrow}\Box_f^{\downarrow}\Box_C^{\downarrow}np\circ\Box_3^{\downarrow}\Box_{sg}^{\downarrow}\Box_g^{\downarrow}\Box_{nom}^{\downarrow}np\backslash s\vdash s}\,[L\backslash]
$$

Figure 8.13: Derivation of 'Marla smiles' with feature information

Now it is clear that the $[R\Box_{nom}^{\downarrow}]$ contraction does not depend in any way on what happens to the $[\langle\rangle^f]$ link; only the $[\langle\rangle^C]$ link is relevant to this contraction and there is only one rule which can produce a $[\langle\rangle^{nom}]$ link, namely the [*nom,C*] conversion. Similarly, after the $[R\Box_{nom}^{\downarrow}]$ contraction we only the [*g,f*] rule can produce a redex for the $[R\Box_g^{\downarrow}]$ contraction.

The general idea here is: given an active par link which is not a redex for the corresponding contraction, we look at which structural conversions could be the last conversion just before the contraction and then, recursively, we look at which conversions could have produced the redex for the previous conversion. This strategy is particularly effective when modes have only

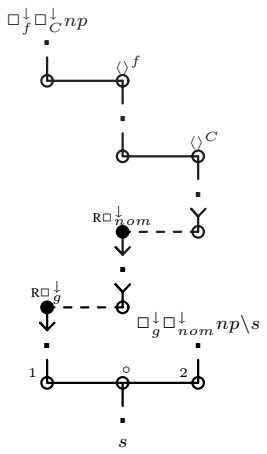

Figure 8.14: Abstract proof structure for 'Marla smiles'

structural rules for inclusion, as in the example above, but other grammars can benefit from this strategy as well. If we look at Figure 8.8 on page 147 again, it is obvious that in order to produce a redex for a $[R/_a]$ contraction we only need to use the $[Ass1]$ structural rule, whereas we would only need to use the $[Ass2]$ structural rule to produce a $[R\backslash_a]$ redex. Either structural rule could be the last conversion before a $[L\bullet_a]$ contraction, however.

8.10 Conclusions

We have seen several ways of improving the efficiency of the initial, naive algorithm by giving heuristics which are applicable in many cases. Though the complexity results from the next chapter make it unlikely that we will find an efficient algorithm for the general problem, it is possible to find algorithms which work reasonably well for large subclasses of the problem.