In: *Proceedings of CoNLL-2000 and LLL-2000*, pages 184–193, Lisbon, Portugal, 2000.

Incorporating Linguistics Constraints into Inductive Logic Programming

James Cussens
Dept. of Computer Science
University of York
Heslington, York, YO10 5DD, UK
jc@cs.york.ac.uk

Stephen Pulman
University of Cambridge Computer Laboratory
New Museums Site, Pembroke Street
Cambridge CB2 3QG, UK
Stephen.Pulman@cl.cam.ac.uk

Abstract

We report work on effectively incorporating linguistic knowledge into grammar induction. We use a highly interactive bottom-up inductive logic programming (ILP) algorithm to learn 'missing' grammar rules from an incomplete grammar. Using linguistic constraints on, for example, head features and gap threading, reduces the search space to such an extent that, in the small-scale experiments reported here, we can generate and store all candidate grammar rules together with information about their coverage and linguistic properties. This allows an appealingly simple and controlled method for generating linguistically plausible grammar rules. Starting from a base of highly specific rules, we apply least general generalisation and inverse resolution to generate more general rules. Induced rules are ordered, for example by coverage, for easy inspection by the user and at any point, the user can commit to a hypothesised rule and add it to the grammar. Related work in ILP and computational linguistics is discussed.

1 Introduction

A major advantage of inductive logic programming is the ability to incorporate domain knowledge (background knowledge) into the inductive process. In ILP domain knowledge is usually encoded by (i) a set of definite clauses declaring rules and facts which are true (or assumed to be true) in the domain and (ii) extra-logical constraints on the hypothesis space. The ILP approach thus allows a very direct and flexible method of expressing domain knowledge.

In this paper, we report on continuation of the work described in (Cussens and Pulman, 2000), which attempts to maximise the effectiveness of linguistic knowledge when inducing a grammar. We take an existing grammatical formalism (derived from the FraCaS Project (1996)) and extend it with inductive capabilities, rather than shoe-horning a grammar learning problem into a form suitable for some particular ILP algorithm. This has major practical benefits, since the required linguistic knowledge can be encoded in a *linguistically natural manner*. As in all real applications of ILP most effort is required in 'getting the background knowledge right'. Being able to express this knowledge in a representation specifically developed to enable linguists to write down a grammar makes this step easier and quicker.

The paper is organised in a manner analogous to that of our algorithm. In Section 2, we describe how to generate *naive* grammar rules directly from the chart produced during a failed parse. The essentials of this approach have already been described in (Cussens and Pulman, 2000), but we briefly describe it here for completeness and also because we have altered its implementation. Section 3 describes the most important step of the algorithm—the representation and use of linguistic constraints at an early stage in the inductive process. Section 4 describes the two generalisation operators currently used in the search by way of an example. Section 5 describes two further experiments very briefly. Most of the various components of our method have been investigated previously either in the ILP or the computational linguistics literature: in Section 6 we discuss this related work. In Section 7, we assess the current work and point to future work.

2 Generating naive rules

The first step in our algorithm can be described as *inductive chart parsing*. The details of integrating induction into chart parsing have been described in (Cussens and Pulman, 2000), here we give just a brief account. This first step of the algorithm is the only one that has been retained from this previous work. The basic idea is that, after a failed parse, we use abduction to find *needed edges*: which, if they existed, would allow a complete parse of the sentence. These are produced in a top-down manner starting with the initial need for a sigma edge spanning the entire sentence. If a need matches the mother of a grammar rule and edges for all the daughters bar one are in the chart, then the missing daughter edge is generated as a new need.

The process of generating naive rules is very simple, and we will explain it by way of an example. Suppose the `vp_vp_mod` grammar rule, shown in Fig 1, has been artificially removed from a grammar. The absence of this rule means

```
vp_vp_mod syn vp:[gaps=[A,B],mor=C,aux=n]==>
[ vp:[gaps=[A,D],mor=C,aux=n],
  mod:[gaps=[D,B],of=or(s,vp),type=_]].
```

Figure 1: Missing grammar rule (human-readable representation)

that, for example, the sentence *All big companies wrote a report quickly* can not be parsed, since we can not get the needed VP *wrote a report quickly* from the found VP *wrote a report* and the found MOD *quickly*. The corresponding needed and actual (complete) edges are given in Fig 2. A naive rule is constructed by putting a

```
%need(Sent,Cat,From,To).
need(1, vp([ng,ng],
  f(0,0,0,0,1,1,1,1,1),_), 3, 7).
%edge(Sent,Id,Origin,From,To,Cat,..)
ThCat,From,To).
edge(1, 39, vp_v_np, 3, 6,
  vp([_A,_A],f(0,0,0,0,_,_,_,1,1),n),...).
edge(1, 19, quickly, 6, 7,
  mod([_B,_B],f(0,0,0,1),f(0,1,1,1)),...).
```

Figure 2: Needed and (abbreviated) actual edges

needed edge on its LHS and other edges on the RHS which in this case gives us the naive rule in Fig 3. In (Cussens and Pulman, 2000) only actual edges were allowed on the RHS of a naive rule, since this ensures that the naive rule suffices to allow a parse. Recently, we have added an option which allows *needed* edges to appear on the RHS, thus generating more naive rules. This amounts to conjecturing that the needed edges should actually be there, but are missing from the set of actual edges because some other grammar rule is missing: thus preventing the parser from producing them. Since all naive rules are subsequently constrained and evaluated on the data, and then not added to the grammar unless the user allows them, such bold conjectures can be retracted later on. From

```
cmp_synrule(r0,
vp([ng,ng],f(0,0,0,0,1,1,1,1,1),_),
  vp([_A,_A],f(0,0,0,0,_,_,_,1,1),n),
  mod([_B,_B],f(0,0,0,1),f(0,1,1,1))).
```

Figure 3: Naive VP $\rightarrow$ VP MOD rule in compiled form

an ILP perspective, the construction of naive rules involves repeated applications of inverse resolution (Muggleton and Buntine, 1988) until we produce a clause which meets extra-logical constraints on vertex connectivity. Abbreviating, we produce `vp(3,7) :- vp(3,6)` and then `vp(3,7) :- vp(3,6),mod(6,7)`. This is then followed by variabilising the vertices to give `vp(V1,V3) :- vp(V1,V2),mod(V2,V3)`. Exactly the same procedure can be implemented by building a 'bottom-clause' using the Progol algorithm. We previously used P-Progol (now called Aleph) to construct naive rules in this way, but have since found it more convenient to write our own code to do this.

3 Using linguistic constraints

3.1 Simple filter constraints

The user never sees naive rules; most are filtered out as linguistically implausible and those that survive have generally become specialised. Our basic motto is: *constrain early, constrain tightly*. The aim is that no linguistically implausible rule is ever added to the set of candidate rules. This allows an incremental approach to implementing the constraints. On observing a linguistically implausible rule in the candidate set, we have to specify what makes it implausible and then express this as a constraint in Prolog. In this way, we build up a set of filters which get rid of linguistically implausible naive rules as soon as they are produced.

Table 1 lists the constraints currently used. The *Head features* and *Gap threading* constraints are discussed later. *RHS length* simply limits the number of constituents on the RHS of a rule to some small user-defined integer (in the experiments described here it was equal to 4). *LHS ≠ RHS* filters out rules with a single daughter which is the same category as the mother. *Head OK* filters out rules where the LHS has a head category which is not found on the RHS. The last three constraints in Table 1 act on the LHS of potential rules (i.e. needs), filtering out, respectively, sigma categories, categories which do not appear as the LHS of existing rules (and so are probably lexical) and `s` (sentence) categories.

Constraint	Specialises	Defined on
Head features	Yes	Compiled
Gap threading	Yes	Compiled
RHS length	No	Compiled
LHS ≠ RHS	No	Compiled
Head OK	No	Readable
LHS not `sigma`	No	Needs
LHS not new	No	Needs
LHS not `s`	No	Needs

Table 1: Linguistic constraints

3.2 Gap threading and head feature constraints

Gap-threading is a technique originating with Pereira's 'extraposition grammars' (Pereira, 1981). It is an implementation technique commonly used for dealing with movement phenomena in syntax, as illustrated by a Wh-question like *What does Smith own _?*, where the Wh-word is logically associated with the gap marked '_'.

There are three components to this type of analysis. Firstly, one rule must introduce the 'moved' constituent. This rule also sets up an expectation for a gap of the same type as the moved constituent elsewhere in the sentence. This expectation is coded as a set of features, or in our case, a single tuple-valued feature with 'GapIn' and 'GapOut' values. By setting the value of the 'GapIn' feature to be that of (a copy of) the moved constituent, and GapOut to be some null marker (here, `ng` = nogap) we can enforce that expectation. Secondly, rules which do not involve gaps directly pass the value of the GapIn and GapOut values along their daughters (this is the 'threading' part) making sure that the gap value is threaded everywhere that a gap is permitted to occur linguistically. Thirdly, there are rules which rewrite the type of constituent which can be moved as the empty string, discharging the 'gap' expectation. Example rules of all three types are as follows:

```
(i)   s:[gap=(G,G)]     ->
       np:[type=wh,agr=A,gap=(ng,ng)]
       s:[gap=(np:[type=wh,agr=A,
                   gap=(ng,ng)],ng)]

(ii) vp:[gap(In,Out)] ->
       v:[] np:[gap=(In,Next)]
       pp:[gap=(Nxt,Out)]

(iii) np:[gap=(np:[type=T,agr=A,
    gap=(ng,ng)],ng),type=T,agr=A] ->
        epsilon
```

Rule (i) introduces a fronted wh NP as sister to an S which must contain an associated NP gap agreeing in number etc. Rule (ii) passes the gap feature from the mother VP along the daughters that can in principle contain a gap. Rule (iii) rewrites an NP whose gap value indicates that a moved element precedes it as the empty string. Rules of these three types conspire to ensure that a moved constituent is associated with exactly one gap.

Constituents which cannot contain a gap associated with a moved element outside the constituent identify the In and Out values of the gap feature, and so a usual NP rule might be of the form: `np:[gap(G,G)] -> det:[...] n:[...]` In a sentence containing no gaps the value of In and Out will be `ng` everywhere.

Naive rules will not necessarily fall into one of the three categories above, because the categories that make up their components will have been instantiated in various possibly incomplete ways. Thus in Fig 3 the gaps values in the mother are (`ng,ng`), and those in the daughters are separately threaded (`A,A`) and (`B,B`).

We apply various checks and filters to candidate rules to ensure that the logic of the gap feature instantiations is consistent with the linguistic principles embodied in the gap threading analysis.

The gap threading logic is tested as follows. Firstly, rules are checked to see whether they match the general pattern of the three types above, gap-introduction, gap-threading, or gap-discharge rules. Secondly, in each of the three cases, the values of the gap features are checked to ensure they match the relevant schematic examples above.

The most frequently postulated type of rule is a gap threading rule. The rule in Fig 3 has the general shape of such a rule but the feature values do not thread in the appropriate way and so it will be in effect unified with a template that makes this happen. The effect here will actually be to instantiate all In and Out values to `ng`, thus specialising the rule. Hypothesised rules where the values are all variables will get the In and Out values unified analogously to the example threading rule (ii) above. Hypothesised rules where the gap values are not variables are checked to see that they are subsumed by the appropriate schema: thus all the different threading patterns in Fig 4 would be substitution instances of the pattern imposed by the example threading rule (ii). At the later generalisation stage the correct variable threading regime should be the only one consistent with all the observed instantiation patterns.

```
%[ng/ng,ng/ng,ng/ng]. 1
[all,big,companies,wrote,a,report,quickly].

%[np/ng,np/ng,ng/ng]. 2
[what,dont,all,big,companies,read,
                  with,a,machine].

%[np/ng,np/np,np/ng]. 3
[what,dont,all,big,companies,read,
                   a,report,with].

%[np/np,np/np,np/np]. 4
[what,dont,all,big,companies,read,
           a,report,quickly,from].
```

Figure 4: Artificial dataset showing 4 different patterns of gap threading

Our constraints on head feature agreement are similar to the gap threading constraints. The specialised version of the naive rule in Fig 3 is displayed in Fig 5. Note that although the rule in Fig 5 is not incorrect, it is overly specific, applying only to `mor=pl,aux=n` where there is no gap to thread. We now consider how to generalise rules.

```
vp:[gaps=[ng:[],ng:[]],mor=pl,aux=n]==>
 [vp:[gaps=[ng:[],ng:[]],mor=pl,aux=n],
 mod:[gaps=[ng:[],ng:[]],of=vp,type=n]]
```

Figure 5: VP $\rightarrow$ VP MOD rule specialised to meet head and gap constraints

4 Generalisation operators

In this section, we show how to generate grammar rules by generalising overly specific rules using the VP $\rightarrow$ VP MOD running example. Our target is to generate the missing grammar rule displayed in Fig 1. We will use the artificial dataset given in Fig 4 which displays 4 different patterns of gap threading. From the first three sentences we generate the expected overly specific grammar rules which correspond to the three patterns of gap threading. These are given, in abbreviated form, in Fig 6. We use least general generalisation (lgg)

```
%Covers sentence 1
vp:[gaps=[ng,ng],mor=pl,aux=n]==>
 [vp:[gaps=[ng,ng],mor=pl,aux=n],
 mod:[gaps=[ng,ng],of=vp,type=n]]

%Covers sentence 2
vp:[gaps=[np,ng],mor=inf,aux=n]==>
 [vp:[gaps=[np,ng],mor=inf,aux=n],
 mod:[gaps=[ng,ng],of=or(nom,vp),type=n]]

%Covers sentence 3
vp:[gaps=[np,ng],mor=inf,aux=n]==>
 [vp:[gaps=[np,np],mor=inf,aux=n],
 mod:[gaps=[np,ng],of=or(nom,vp),type=n]]
```

Figure 6: Overly specific gap threading rules (in abbreviated form)

as our basic generalisation operator. This is implemented (for terms) in the Sicstus `terms` library built-in `term_subsumer/3`. Lgg operates on the compiled form of the rules (such as the `cmp_synrule/3` unit clause displayed in Fig 5), not the human-readable form as in Fig 6. The lgg of the first two rules produces the following rule (translated back into human-readable form):

```
vp:[gaps=[_282,ng:[]],mor=or(inf,pl),aux=n]
==>
[
vp:[gaps=[_282,ng:[]],mor=or(inf,pl),aux=n],
mod:[gaps=[ng:[],ng:[]],of=or(nom,vp),type=n]
]
```

The lgg of this rule with the third is:

```
vp:[gaps=[_286,ng:[]],mor=or(inf,pl),aux=n]
==>
[
vp:[gaps=[_286,_270],mor=or(inf,pl),aux=n],
mod:[gaps=[_270,ng:[]],of=or(nom,vp),type=n]
]
```

This rule covers the first three sentences but is not general enough to cope with the situation where the gap is not discharged on the mother VP—a pattern present in the fourth sentence.

Unfortunately, the fourth sentence needs to use the missing rule twice to get a parse, and it is a fundamental limitation of our approach that a missing rule can only be recovered from a failed parse if it is required only once. Note that to induce a rule we only need one sentence where the rule is needed once—our assumption is that in real (large) training datasets there will be enough sentences for this to be true for any missing grammar rule.

Although this assumption seems reasonable, we have decided to experiment with a generalisation operator, which is helpful when the assumption does not hold true. A rule with a context-free skeleton of VP $\rightarrow$ VP MOD MOD *is* generated from the fourth sentence. This corresponds to the two applications of the target VP $\rightarrow$ VP MOD rule. The rule we have, can be derived by having the target rule resolve on itself. It follows that we can inductively generate the target rule from VP $\rightarrow$ VP MOD MOD by implementing a special inverse resolution operator which produces the most specific clause C_2 from a clause C_1, when C_1 can be produced by C_2 resolving with itself. Applying this operator to the VP $\rightarrow$ VP MOD MOD rule renders:

```
vp:[gaps=[np,_342],mor=inf,aux=n]==>
 [vp:[gaps=[np,np],mor=inf,aux=n],
  mod:[gaps=[np,_342],of=or(nom,vp),type=n]]
```

'Lggifying' this latest rule with the lgg of the 3 other rules finally generates a grammar rule with the correct gap threading, which we display in Fig 7 as it appears to the user (with a few added line breaks). However, this rule is not general enough simply because our training data is not general enough. Adding in the sentences *All big companies will write a report quickly*, *All big companies have written a report quickly* and *All big companies wrote a report incredibly* generates a more general version covering these various cases. However, there is still a problem because our induced rule allows the modifier to be modifying either a `nom` or a `vp` (represented by the term `f(0,_280,_280,1)` in the compiled form), where the correct rule allows the modifier to modify an `s` or a `vp` (represented by the term `f(0,0,_280,1)` in the compiled form). This is because our constraints still need to be improved.

```
| ?- display_rules.

r158 vp ==> [vp,mod]
vp:[gaps=[_384,_368],mor=or(inf,pl),aux=n]==>
[vp:[gaps=[_384,_366],mor=or(inf,pl),aux=n],
mod:[gaps=[_366,_368],of=or(nom,vp),type=n]]

cmp_synrule(r158,vp([_324,_322],
                f(0,0,_316,_316,1,1,1,1,1),n),
[vp([_324,_302],f(0,0,_316,_316,1,1,1,1,1),n),
mod([_302,_322],f(0,_280,_280,1),f(0,1,1,1))])

INFO: [head_feature_status(good,
[mor/f(0,0,_316,_316,1,1,1,1,1),aux/n]=
[mor/f(0,0,_316,_316,1,1,1,1,1),aux/n]),
gap_feature_status(gap_threading_rule),score(2)]
Covers: 4 sentences:
[4,3,2,1]
*******************
** Hit ENTER to continue, anything else to stop **
```

Figure 7: Almost finding the missing grammar rule

5 Two experiments

Our experiments consist of (i) randomly generating 50 sentences from a grammar, (ii) deleting some grammar rules and (iii) seeing whether we can recover the missing grammar rules using the 50 sentences. Our approach is interactive with *the user* making the final decision on which hypothesised rules to add to the grammar. Hypothesised rules are currently ordered by coverage and presented to the user in that order. In our artificial experiments the earlier the missing rule is presented to the user the more successful the experiment.

In the first experiment we deleted the VP $\rightarrow$ VP MOD rule in Fig 1 and the rule

```
np_det_nom syn
np:[gaps=[A,A],mor=B,type=C,case=_]==>
  [det:[type=C,mor=B],nom:[mor=B]].
```

After generalisation of naive rules, the rule with the largest cover was

```
np:[gaps=[ng:[],ng:[]],mor=or(pl,s3),
                type=_414,case=_415]==>
  [det:[type=or(n,q),mor=_405],
   nom:[mor=or(pl,s3)]]
```

which is over-general since the morphology feature of the determiner is not constrained to equal that of the mother. However, the third most general rule covered 24 sentences and was:

```
np:[gaps=[ng:[],ng:[]],mor=or(pl,s3),
               type=n,case=_442]==>
  [det:[type=n,mor=or(pl,s3)],
   nom:[mor=or(pl,s3)]]
```

which does have agreement on morphology. Committing to this latter rule by asserting it as a grammar rule, removing newly parsable sentences and re-generating rules produced a `vp ==> [vp,mod]` rules which was more general in terms of morphology than the one in Fig 7, but less general in terms of gap threading. This just reflects the sensitivity of our learning strategy on the particular types of sentences in the training data.

In a second experiment, we deleted the rules:

```
nom_nom_mod syn nom:[mor=A]==>
 [nom:[mor=A],
  mod:[gaps=[ng:[],ng:[]],of=nom,
                  type=or(n,q)]].

vp_v_np syn vp:[gaps=A,mor=B,aux=C]==>
 [v:[mor=B,aux=C,inv=n,subc=[np:[gaps=_,
                 mor=_,type=_,case=_]]],
  np:[gaps=A,mor=_,type=or(n,q),
                          case=nonsubj]].

s_aux_np_vp syn
s:[gaps=A,mor=or(pl,or(s1,or(s2,s3))),
           type=or(n,q),inv=y]==>
 [v:[mor=or(pl,or(s1,or(s2,s3))),
   aux=y,inv=y,
   subc=[vp:[gaps=_,mor=B,aux=_]]],
 np:[gaps=[ng:[],ng:[]],
   mor=or(pl,or(s1,or(s2,s3))),
   type=or(n,q),case=subj],
 vp:[gaps=A,mor=B,aux=_]].
```

Our algorithm failed to recover the `s_aux_np_vp` rule but did find close approximations to the other two rules:

```
vp:[gaps=[_418,_420],
     mor=or(inf,or(ing,s3)),aux=n]==>
 [v:[mor=or(inf,or(ing,s3)),aux=n,inv=n,
          subc=[np:[gaps=_430,mor=_431,
          type=or(n,q),case=nonsubj]]],
 np:[gaps=[_418,_420],mor=or(pl,s3),
                   type=n,case=_407]]

nom:[mor=or(pl,s3)]==>
 [nom:[mor=or(pl,s3)],
 mod:[gaps=[_339,_339],
    of=or(nom,vp),type=or(n,q)]]
```

6 Related work

The strong connections between proving and parsing are well known (Shieber et al., 1995), so it is no surprise that we find related methods in both ILP and computational linguistics. In ILP the notion of inducing clauses to fix a failed proof, which is the topic of Section 2, is very old dating from the seminal work of Shapiro (1983). In NLP, Mellish (1989) presents a method for repairing failed parses in a relatively efficient way based on the fact that, after a failed parse, the information in the chart is sufficient for us to be able to determine what constituents would have allowed the parse to go through if they had been found.

6.1 Related work in ILP

The use of abduction to repair proofs/parses has been extensively researched in ILP as has the importance of abduction for multiple predicate learning. De Raedt (1992), for example, notes that "Roughly speaking, combining abduction with single predicate-learning leads to multiple concept-learning". This paper, where abduction is used to learn, say, verb phrases and noun phrases from examples of sentences is an example of this. Recent work in this vein includes (Muggleton and Bryant, 2000) and the papers in (Flach and Kakas, 2000).

Amongst this work a particularly relevant paper for us is (Wirth, 1988). Wirth's *Learning* *by Failure to Prove* (LFP) approach finds missing clauses by constructing partial proof trees (PPTs) and hence diagnosing the source of incompleteness. A clause representing the PPT is constructed (called the resolvent of the PPT) as is an approximation to the resolvent of the complete proof tree. Inverse resolution is then applied to these two clauses to derive the missing clause. Wirth explains his method by way of a small context-free DCG completion problem.

Our approach is similar to Wirth's in the dependence on abduction to locate the source of proof (i.e. parse) failure. Also both methods use a meta interpreter to construct partial proofs. In our case the meta-interpreter is the chart parser augmented with the generation of needs and the partial proof is represented by the chart augmented with the needs. In Wirth's work the resolvent of the PPT represents the partial proof and a more general purpose meta-interpreter is used. (We conjecture that our tabular representation has a better chance of scaling up for real applications.) Thirdly, both methods are interactive. Translating his approach to the language of this paper, Wirth asks the user to verify that proposed needed atoms (our needed edges) are truly needed. The user also has to evaluate the final hypothesised rules. We prefer to have the user only perform the latter task, but the advantage of Wirth's approach is that the user can constrain the search at an earlier stage. Wirth defends an interactive approach on the grounds that "A system that learn[s] concepts or rules from looking at the world is useless as long as the results are not verified because a user who feels responsible for his knowledge base rarely use these concepts or rules".

In contrast to (Cussens and Pulman, 2000) we now search bottom-up for our rules. This is because the rules we are searching for are near the bottom of the search space, and also because bottom-up searching effects a more constrained, example-driven search. Bottom-up search has been used extensively in ILP. For example, the GOLEM algorithm (Muggleton and Feng, 1990) used relative least general generalisation (rlgg). However, bottom-up search is rare in modern ILP implementations. This is primarily because the clauses produced can be unmanageably large, particularly when generalisation is performed relative to background knowledge, as with rlgg. Having grammar rules encoded as unit clauses alleviates this problem as does our decision to use lgg rather than rlgg.

Zelle and Mooney (1996) provides a bridge between ILP and NLP inductive methods. Their CHILL algorithm is a specialised ILP system that learns control rules for a shift-reduce parser. The connection with the approach presented here (and that of Wirth) is that intermediate stages of a proof/parse are represented and then examined to find appropriate rules. In CHILL these intermediate stages are states of a shift-reduce parser.

6.2 Related work in NLP

Most work on grammar induction has taken place using formalisms in which categories are atomic: context-free grammars, categorial grammars, etc. Few attempts have been made at rule induction using a rich unification formalism. Two lines of work that are exceptions to this, and thus comparable to our own, are that of Osborne and colleagues; and the work of the SICS group using SRI's Core Language Engine and similar systems.

Osborne (1999) argues (correctly) that the hypothesis space of grammars is sufficiently large that some form of bias is required. The current paper is concerned with methods for effecting what is known as declarative bias in the machine learning literature, i.e. hard constraints that reduce the size of the hypothesis space. Osborne, on the other hand, uses the Minimum Description Length (MDL) principle to effect a preferential (soft) bias towards smaller grammars. His approach is incremental and the induction of new rules is triggered by an unparsable sentence as follows:

1. Candidate rules are generated where the daughters are edges in the chart after the failed parse, and the mother is one of these daughters, possibly with its bar level raised.

2. The sentence is parsed and for each successful parse, the set of candidate rules used in that parse constitutes a model.

3. The 'best' model is found using a Minimum Description Length approach and is added to the existing grammar.

So Osborne, like us, uses the edges in the chart after a failed parse to form the daughters of hypothesised rules. The mothers, though, are not found by abduction as in our case, also there is no subsequent generalisation step.

Unlike us Osborne induces a probabilistic grammar. When candidate rules are added, probabilities are renormalised and the n most likely parses are found. If annotated data is being used, models that produce parses inconsistent with this data are rejected. In (Osborne, 1999), the DCG is mapped to a SCFG to compute probabilities. In very recent work a stochastic attribute-value grammar is used (Osborne, 2000). Giving the increasing sophistication of probabilistic linguistic models (for example, Collins (1997) has a statistical approach to learning gap-threading rules) a probabilistic extension of our work is attractive—it will be interesting to see how far an integration of 'logical' and statistical can go.

Thalmann and Samuelsson (1995) describe a scheme which combines robust parsing and rule induction for unification grammars. They use an LR parser, whose states and actions are augmented so as to try to recover from situations that in a standard LR parser would result in an error. The usual actions of shift, reduce, and accept are augmented by

hypothesised shift: shift a new item on to the stack even if no such action is specified in that state

hypothesised unary reduce: reduce a symbol Y as if there was a rule X $\rightarrow$ Y, where the value of X is not yet determined.

hypothesised binary reduce: reduce a symbols Y Z as if there was a rule X $\rightarrow$ Y Z, where the value of X is not yet determined.

The value of the X symbol is determined by the next possibilities for reduction.

To illustrate, consider the grammar

1	S	$\rightarrow$	NP VP
2	NP	$\rightarrow$	Name
3	VP	$\rightarrow$	Vi

and a sentence 'John snores loudly'. Assume that all the words are known (though this is not necessary for their method). The sequence of events will be:

Operation	Stack
1. Shift	Name:john
2. Reduce with 2	NP[Name:john]
3. Shift	NP[Name:john] Vi:snores
4. Reduce with 3	NP[Name:john] VP[Vi:snores]
5. HShift	NP VP Adv:loudly
6. HReduce	NP X[VP Adv]
7. Reduce with 1	S[NP [VP VP Adv]]

After stage 4 we could reduce with 1 but this would not lead to an accepting state. Instead we perform a hypothesised shift at stage 5 followed by a hypothesised binary reduce with X $\rightarrow$ VP Adv in stage 6. Next we reduce with rule 1 which instantiates X to VP and we have a complete parse provided we hypothesise the rule VP $\rightarrow$ VP Adv.

Two more hypothesised actions are used to account for gap threading:

hypothesised move: put the current symbol on a separate movement stack (i.e. hypothesise that this constituent has been fronted)

hypothesised fill: move the top of the movement stack to to top of the main stack

These actions have costs associated with them and a control regime so that the 'cheapest' analysis will always be preferred. An analysis which uses none of the new actions will be cost-free. Unary reduction is more expensive than binary reduction because the consequent unary rules may lead to cycles, and such rules are often redundant.

These actions hypothesise only the context free backbone of the rules. Feature principles analogous to those we described above are used, along with hand editing, to get the final form of the hypothesised rule. Presumably the information hypothesised by the move and fill operations as to be translated somehow into the gap threading notation which is also used by their formalism. No details are given of the results of this system, nor any empirical evaluation.

This work shares many of the goals of the approach we describe, in particular the use of explicit encoding of background knowledge of feature principles. The main difference is that the technique they describe only hypothesises the context free backbone of the necessary rules, whereas in our approach the feature structures are also hypothesised simultaneously.

Asker et al. (1992) also describe a method for inducing new lexical entries when extending coverage of a unification grammar to a new domain, a task which is also related to our work in that they are using a full unification formalism and using partial analyses to constrain hypotheses. Firstly, they use 'explanation based generalisation' to learn a set of sentence templates for those sentences in the new corpus that can be successfully analysed. This process essentially takes commonly occurring trees and 'flattens' them, abstracting over the content words in them. Secondly they use these templates to analyse those sentences from the new corpus which contain unknown words, treating the entries implied by the templates for these words as provisionally correct. Finally these inferred entries are checked against a set of hand-coded 'paradigm' entries, and when all the entries corresponding to a paradigm have been found, a new canonical lexical entry for this word is created from the paradigm.

Again, no results are evaluation are given, but it is clear that this method is likely to yield similar results to our own for inference of lexical entries.

7 Future directions

We find our preliminary results encouraging because (i) we usually get close to missing rules, (ii) the rules are fairly linguistically sophisticated, for example, involving gap threading and (iii) the burden on the user is light—by ordering induced rules by their coverage, the user sees the best rules first, and does not have to bother inspecting the mass of highly specialised rules produced. The work is incomplete and ongoing, and we conclude by listing three important tasks for the next phase of our work where we intend to do thorough empirical testing on real data.

(1) In (Cussens and Pulman, 2000), edges were re-used to speed up cover testing. This is still not working in the newer implementation. (2) In real applications missing lexical items are more significant than missing grammar rules. Although one can easily learn lexical items by encoding them as grammar rules it should be more efficient to replace an unknown word by a variable, and then just see how it gets instantiated as we parse. (3) In these small experiments we could get away with an appealingly simple learning strategy: produce and store all naive rules then produce and store all possible lggs. To scale up we will probably need to use a greedier approach.

Acknowledgements

Thanks to Christer Samuelsson and Björn Gambäck for pointing us to relevant literature.

References

Lars Asker, Björn Gambäck, and Christer Samuelsson. 1992. EBL^2: An approach to automatic lexical acquisition. In *Proceedings of the 14th International Conference on Computational Linguistics*, pages 1172–1176.

Michael Collins. 1997. Three generative, lexicalised models for statistical parsing. In *Proc. ACL'97*.

James Cussens and Stephen Pulman. 2000. Experiments in inductve chart parsing. In James Cussens and Sašo Džeroski, editors, *Learning Language in Logic*. Springer.

Luc De Raedt. 1992. *Interactive Theory Revision: An Inductive Logic Programming Approach*. Academic Press, London.

Peter A. Flach and Antonis C. Kakas, editors. 2000. *Abduction and Induction: Essays on their Relation and Integration*, volume 18 of *Applied Logic Series*. Kluwer, Dordrecht.

Chris Mellish. 1989. Some chart based techniques for parsing ill-formed input. In *Proc 27th ACL*, pages 102–109, Vancouver, BC. ACL.

Stephen Muggleton and Christopher Bryant. 2000. Theory completion using inverse entailment. In James Cussens and Alan Frisch, editors, *Proceedings of the 10th International Conference on Inductive Logic Programming*, pages 130–146, London, August. Springer.

Stephen Muggleton and Wray Buntine. 1988. Machine invention of first-order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 339–352. Kaufmann.

Stephen Muggleton and Cao Feng. 1990. Efficient induction of logic programs. In *Proc. of the First Conference on Algorithmic Learning Theory*, pages 473–491, Tokyo.

Miles Osborne. 1999. MDL-based DCG Induction for NP Identification. In Miles Osborne and Erik Tjong Kim Sang, editors, *CoNLL99*, pages 61–68, Bergen, Norway, June. EACL.

Miles Osborne. 2000. Estimation of Stochastic Attribute-Value Grammars using an Informative Sample. In *Coling 2000*.

F.C.N. Pereira. 1981. Extraposition grammars. *Computational Linguistics*, 7:243–256.

FraCaS project. 1996. Fracas: A framework for computational semantics. http://www.cogsci.ed.ac.uk/~fracas/.

Ehud Shapiro. 1983. *Algorithmic Program Debugging*. MIT Press, Cambridge.

Stuart M. Shieber, Yves Schabes, and Fernando C. N. Pereira. 1995. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1–2):3–26. Available at the Computation and Language e-print archive as cmp-lg/9404008.

Lars Thalmann and Christer Samuelsson. 1995. A uniform framework for grammar induction and robust parsing. In *Proceedings of the 5th Scandinavian Conference on Artificial Intelligence,*, pages 293–304.

Ruediger Wirth. 1988. Learning by failure to prove. In Derek Sleeman, editor, *Proceedings of the 3rd European Working Session on Learning*, pages 237–251, Glasgow, October. Pitman.

J. M. Zelle and R. J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, OR, August.