Adapting polarised disambiguation to surface realisation

Eric Kow
Langue et Dialogue - LORIA
615 rue du jardin botanique
54600 Villers-Lès-Nancy, France
Eric.Kow@loria.fr

Abstract. The aim of surface realisation is to produce a string from an input semantics. The task may be viewed as the inverse of parsing, and like parsing, it is made more difficult by the problem of lexical ambiguity in natural language. (Bonfante, Guillaume, and Perrier 2004) propose a polarisation technique for parsing that greatly reduces the effects of ambiguity. We show how polarisation can be adapted for surface realisation and how to address two complications which arise: multi and zero-literal semantic input.

1 Introduction

Surface realisation can be seen as parsing in reverse: given a grammar and some semantic input, the task is to construct the parse trees associated by the grammar with that semantics, and to output the resulting strings. Like parsing, surface realisation is severely affected by the possibility for each part of the semantics to be realised by several lexical entries. But unlike parsing, surface realisation is NP-complete (Koller and Striegnitz 2002), so that a solution to the lexical ambiguity problem becomes particularly desirable. A common response is to disambiguate the input with a separate preprocessing filter before parsing or realisation proper. The filter can be made to be very fast by using probabilistic methods (Joshi and Srinivas 1994); however, such methods carry the risk of discarding valid disambiguations along with the incorrect ones. We aim for a surface realiser that can produce all possible paraphrases, that is explore the entire search space, and so we eschew the use of probabilities in favour of an "exact method" (Bonfante, Guillaume, and Perrier 2004). The basic idea is to make the resource sensitivity of grammars explicit by annotating lexical items with polarities and using a filtering step

Proceedings of the Ninth ESSLLI Student Session
Paul Egré and Laura Alonso i Alemany (editors)
Chapter 1, Copyright © 2004, Eric Kow

to neutralise them. This has been successfully applied to parsing (Bonfante, Guillaume, and Perrier 2003), but adapting it to generation reveals some hidden complications which stem from lexical items spanning either multiple pieces of input, or none at all.

In this paper, we propose some modifications to the filtering algorithm that overcome these problems. The result is that the optimised surface realiser retains its ability to produce output with such items as pronouns and control verbs. We begin in sections 2 and 3 by briefly presenting the surface realiser and a basic version of its polarity filter. Then in sections 4 and 5, we discuss the modifications necessary to make the filter work for multi-literal and zero-literal semantic input.

2 The surface realiser

The surface realiser uses a Feature Based Lexicalised Tree Adjoining Grammar (FLTAG) (Vijay-Shanker and Joshi 1988) and a flat semantic representation (Copestake, Lascarides, and Flickinger 2001). The properties of FLTAG are largely irrelevant to this paper, except that a grammar is a set of lexical items where each item is a tree with one or more anchors. A flat semantics consists of a set of propositions called **literals**. A literal consists of a predicate followed by a list of indices, its arguments. If two literals have arguments with the same index, that index refers to the same entity. Below, for example, the index `p` in `picture(p)` and `cost(c,p,h)` refers to the same object.

(1) `picture(p), cost(c,p,h), high(h)` (*The picture is expensive*)

Given a semantics (the **input semantics**) and a set of lexical items retrieved from the FLTAG (the **lexical input**), the surface realiser produces sentences whose semantics is equal to the input semantics. It uses a tabular algorithm with bottom-up processing. The lexical input consists of the set of lexical items whose semantics subsumes the input semantics. Several lexical items may cover the same literal, raising ambiguities to be solved. For example, the lexical input associated with (1) could be $\tau_{painting}$ or $\tau_{picture}$ for `picture(p)`; τ_{cost} (the noun) or τ_{costs} for `cost(c,p,h)`; and τ_{high} or $\tau_{a\ lot}$ for `high(h)`.

3 Polarity filtering

We attempt to resolve all input lexical ambiguities into a set of disambiguations where each disambiguation is a combination of selected lexical items.

Given a lexical input, the number of lexical combinations is *a priori* exponential: $\prod_{1 \leq i \leq n} a_i$ with a_i the degree of lexical ambiguity of the i-th literal and n, the number of literals in the input semantics. In practice, however, many of these combinations are syntactically invalid: either some items have syntactic requirements which cannot be met, or some of the lexical items cannot be combined with others and stitched into a successful realisation (Perrier 2003). To reduce the combinations resulting from lexical ambiguity, we introduce a single filtering step whose role is to efficiently detect these invalid combinations so that they are not explored during realisation proper.

We identify the syntactic requirements of each lexical item and only retain the lexical combinations in which the requirements of every item are met. More precisely, each lexical item is assigned a bag of labels that have either a polarity + or −. These labels provide hints about which trees may combine with each other. For instance, an intransitive verb will have the label `-np`, meaning it "requires" a noun phrase (the subject), whereas a transitive verb will have the labels `-np -np` (the subject and object), requiring two noun phrases. In the case of TAG grammars, these polarities can be automatically extracted (Bonfante, Guillaume, and Perrier 2004) by awarding each tree with a $-f$ polarity for every foot or substitution node with category f and a $+r$ polarity, where r is the category of its root node. For instance, a tree like `s(np↓,vp(v(hates),np↓))` would have the polarities `-np -np +s`.

In the table below, we show how these polarities can be put to use for generation. Each column contains a single literal from the input semantics `picture(p), cost(c,p,h), high(h)`, along with the lexical items that realise that literal as well as their associated polarities. To simplify this example, we only consider a single label, `np`.

`picture(p)`	`cost(c,p,h)`	`high(h)`
$\tau_{picture}$ `+np`	τ_{cost} `+np -np`	τ_{high} `-np`
$\tau_{painting}$ `+np`	τ_{costs} `-np`	$\tau_{a\ lot}$

The problem at hand is to choose a combination that covers the entire semantics, that is, one lexical item per column. We can avoid the syntactically invalid combinations by counting polarities. If the sum of polarities (or **charge**) is greater than zero, then the lexical combination overall has more trees than it can use. Similarly, if the charge is less than zero, then the combination requires more trees than it can provide. The charge must be equal to zero or else the combination is syntactically invalid. For instance, the combination $\tau_{painting}\ \tau_{cost}\ \tau_{a\ lot}$ has a charge of $+1np$, so it is clearly not a solution. In contrast, $\tau_{painting}\ \tau_{costs}\ \tau_{a\ lot}$ has a charge of $0np$, which does not *guarantee* syntactic compatibility but suggests that the combination is worth exploring.

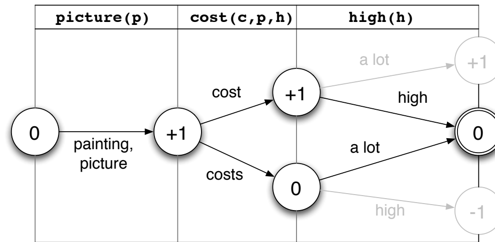

Figure 1.1: A minimised polarity automaton

Rather than performing a separate polarity count for every lexical combination, we factorise the work into an automaton which calculates the lexical combinations and their net charge. We arbitrarily impose an order on the input semantics into a list we call $InputSemantics$. Each automaton state has the form $\langle L, C \rangle$ and represents the set[1] of lexical combinations whose semantics is L and whose polarity charge is C. The approach is to construct the automaton in steps, one literal at a time, starting with the initial step $\langle \emptyset, 0 \rangle$. At each step, we build upon the states $\langle L, C \rangle$ created in the previous step. We select the next literal l from $InputSemantics$; for each lexical item lex_l which realises literal l, we add a transition to the state $\langle L + l, C + pol \rangle$ where pol is the polarity of lex_l. When there are no more literals left to select, we complete the process by declaring the final state to be $\langle InputSemantics, 0 \rangle$ and performing automaton minimisation (Hopcroft and Ullman 1979). What remains is a representation of all the lexical combinations that cover the input semantics with zero net charge (figure 1.1). [2] As shown in (Kow 2004), polarity filtering is an efficient method for dealing with lexical ambiguity. Given an input with 438 272 possible combinations, the polarity filter only passes 232 of these combinations on. As a result, the surface realisation time drops from 93.8 to 14.7 seconds (Gardent and Kow 2005).

4 Multi-literal semantic lexical items

Certain lexical items could have a semantics that spans multiple literals, for example `cost(c,p,h), high(h)` for the item $\tau_{expensive}$. These items are not correctly handled because the automaton construction algorithm relies on the assumption that at any given state $\langle L, C \rangle$, the lexical combinations

[1] We will use the following notation for lists: $\emptyset$ indicates the empty list and $L + i$ indicates appending item i to L. We also take the liberty of comparing sets and lists, e.g, if we say that a set S is equal to a list L, we merely treat L as a set.

[2] See (Kow 2004) for a description of how this technique can be generalised for multiple labels.

represented by that state all have a semantics equal to L. The assumption breaks down when one of the combinations has an item lex_m whose semantics includes some literal x which is not in L. If the algorithm later visits literal x, whatever representative it selects for the literal will cause a polarity miscount: either lex_m is reselected and its polarities are double-counted, or some other item with a redundant semantics is selected and its polarities are included in the count. The correct behaviour in this case is neither to double-count polarities nor build lexical combinations with redundant lexical items, but not to select any item at all.

We achieve this by augmenting the automaton states with a third element E, used to keep track of the lexical combinations with extra semantic literals. Given a state $\langle L_1, C_1, E_1 \rangle$, the literal being visited l_1 and the selected lexical item lex_1 with polarity pol and semantics $\{l_1\} \cup extra$; we build a transition via lex_1 to the state $\langle L_1 + l_1, C_1 + pol, E_1 \cup extra \rangle$. Now, given a step with state $\langle L_2, C_2, E_2 \rangle$; if the literal being visited $l_2 \in E_2$, we build a null transition to the state $\langle L_2 + l, C_2, E_2 \setminus \{l_2\} \rangle$.

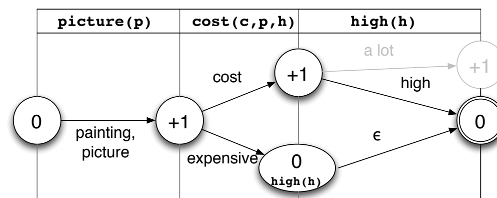

Figure 1.2: A minimised polarity automaton with multi-literal semantics

It may seem desirable to simplify this mechanism by replacing the tracking of semantic literals and use of null transitions with a single transition that spans all the semantic literals covered by a lexical item. In figure 1.2 above, this would mean a direct transition *expensive* between the first +1 and final 0 states. But this simplification is not viable, because lexical items may span overlapping sets of semantics literals. In (Shemtov 1996) for example, *quickly moved into* can also be expressed as *rushed into* or *quickly entered*. Given the ordering below, it would not be possible to build a direct transition for $\tau_{entered}$ across `move(m,x), into(x,y)` without erroneously skipping over the literal `quick(m)`.

lexical item	**semantics**		
moved	`move(m,x)`		
rushed	`move(m,x)`	`quick(m)`	
entered	`move(m,x)`		`into(x,y)`

5 Null and zero-literal semantic lexical items

Lexical items with a **null semantics** typically correspond to function words: complementisers *(John likes **to** read.)*, subcategorised prepositions *(Mary accuses John **of** cheating.)*. Such items need not be lexical items at all. We can exploit TAG's support for trees with multiple anchors, by treating them as co-anchors to some primary lexical item. The English infinitival *to*, for example, can appear in the tree $\tau_{to\ take}$ as `s(comp(to),v(take),np↓)`.

On the other hand, pronouns have a **zero-literal** semantics, one which is not null, but which consists only of a variable index. For example, the pronoun *her* in (2b) has semantics `s` and in (3), *he* has the semantics `j`.

(2) a. `joe(j), sue(s), book(b), lend(l,j,b,s), boring(b)`
Joe lends Sue a boring book.

b. `joe(j),          book(b), lend(l,j,b,s), boring(b)`
Joe lends her a boring book.

(3) `joe(j), sue(s), leave(l,j), promise(p,j,s,l)`
Joe promises Sue to leave.
or *Joe promises Sue that he would leave.*

In figure 1.3, we compare the construction of polarity automata for (2a, left) and (2b, right). Building an automaton for (2b) fails because τ_{sue} is not available to cancel the negative polarities for τ_{lends}; instead, a pronoun must be used to take its place. The problem is that the selection of a lexical item is only triggered when the construction algorithm visits one of its semantic literals. Since pronoun semantics have zero literals, they are *never* selected. Making pronouns visible to the construction algorithm would require us to count the indices from the input semantics. Each index refers to an entity. This entity must be "consumed" by a syntactic functor (e.g. a verb) and "provided" by a syntactic argument (e.g. a noun).

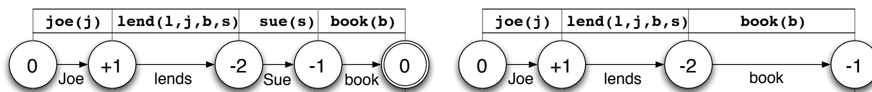

Figure 1.3: Difficulty with zero-literal semantics.

We make this explicit by annotating the semantics of the lexical input (that is, the set of lexical items selected on the basis of the input semantics) with a form of polarities. Roughly, nouns provide indices[3] (+), modifiers leave them unaffected, and verbs consume them (−). Predicting pronouns

[3]except for predicative nouns, which like verbs, are semantic functors

is then a matter of counting the indices. If the positive and negative indices cancel each other out, no pronouns are required. If there are more negative indices than positive ones, then as many pronouns are required as there are negative excess indices. In the table below, we show how the example semantics above may be annotated and how many negative excess indices result:

semantics	b	j	s
joe(+j) sue(+s) book(+b) lend(l,-j,-b,-s) boring(b)	0	0	0
joe(+j) book(+b) lend(l,-j,-b,-s) boring(b)	0	0	1
joe(+j) sue(+s) leave(l,-j,-s) promise(p, j,-s,l)	0	0	0
joe(+j) sue(+s) leave(l,-j,-s) promise(p,-j,-s,l)	0	1	0

Counting surplus indices allows us to establish the number of pronouns used and thus gives us the information needed to build polarity automata. We implement this by introducing a virtual literal for negative excess index, and having that literal be realised by pronouns. Building the polarity automaton as normal yields lexical combinations with the required number of pronouns, as in figure 1.4.

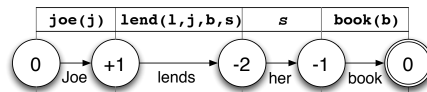

Figure 1.4: Constructing a polarity automaton with zero-literal semantics.

This becomes more complicated when the lexical input contains lexical items with different annotations for the same semantics. For instance, the control verb *promise* has two forms: one which solicits an infinitive as in *promise to leave*, and one which solicits a declarative clause as in *promise that he would leave*. This means two different counts of subject index `j` in (3) : zero for the form that subcategorises for the infinitive, or one for the declarative. But to build a single automaton, these counts must be reconciled, i.e., how many virtual literals do we introduce for `j`, zero or one?

The answer is to introduce enough virtual literals to suppport the largest count (in this case one), and to balance them by adding the virtual literals to the lexical semantics of the smaller counts. To handle example (3), we introduce one virtual literal for `j` in order to select the pronoun *he* in *promise that he would leave*. This extra pronoun is not selected for the infinitive form *promises to leave*, because it is accounted for in the semantics of lexical item $\tau_{promise_to}$, which now consists of `promise(p,j,s,l)` as well as the virtual literal `j`.

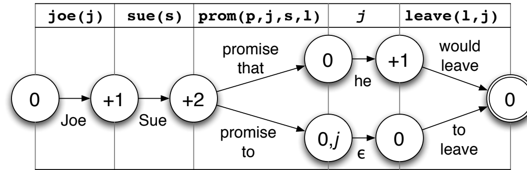

Figure 1.5: Constructing a polarity automaton with zero-literal semantics.

6 Related work

Polarity filtering is not the only mechanism for dealing with ambiguity. (Kay 1996) proposes a chart generation algorithm which groups intermediate results into equivalence classes according to their semantic coverage, syntactic category and a distinguished semantic index. (Shemtov 1996) improves on this approach by using a coarser representation of semantic coverage to produce larger classes. Equivalence classes allow for the variants of an intermediate solution, for example those which arise from lexical ambiguity, to be packed together and processed as a single group. The more structures are recognised as equivalent, the more redundant computation is avoided. Our surface realisation algorithm does not keep track of equivalence classes, but we plan to investigate how they can be incorporated, how usefully they would combine with polarity filtering in practice.

A more similar approach is that of (Koller and Striegnitz 2002). The TAG lexical input is converted into a set of lexical entries of a dependency grammar. These entries are then parsed by an efficient constraint-based dependency parser and the output is converted back into a set of TAG derived trees representing the surface realisation output. The main similarity between this and our approach is that they both use a global mechanism for filtering out combinations of lexical entries that cannot possibly lead to a syntactically valid sequence. Both approaches involve the cancelling out of syntactic resources and requirements. Interestingly, Koller et al. take adjunction into account, whereas our approach largely ignores auxiliary trees. Their treatment of auxiliary trees could be a useful addition to the polarity filter. A key difference is that though Koller et al. handle null semantics, and offer some thoughts on multiple literal semantics, they do not account for zero-literal semantics. It would be worthwhile to see if semantic index counting can also be used within a constraint propagation framework.

7 Conclusion

Polarity filtering was introduced for parsing in (Bonfante, Guillaume, and Perrier 2004). In (Kow 2004), we transpose this approach to surface realisation and show that it greatly enhances efficiency. However, several linguistic issues arise which stem from lexical items displaying multi-literal semantics, null semantics or zero-literal semantics. In this paper, we show how the polarity algorithm can be extended to deal with such items. Briefly, multi-literal semantic items are catered for by introducing a sort of literal counting in the automaton; null semantic lexical items by using TAG support for multiple anchors and zero-literal lexical items by introducing index counting and modifying automaton construction accordingly.

One point is worth noting. Whereas null semantic and multiple literal items have been discussed before (Shieber 1988; Calder et al. 1989; Carroll et al. 1999), zero-literal items have been paid scant attention in the surface realisation literature. Their handling, however, is important for both linguistic and computational reasons. Linguistically, it is required to support the generation of sentences containing pronouns and control verbs as well as full noun phrases and non-control verbs. Computationally, it is necessary to restrain the semantic wildcard behaviour of pronouns. In a naive algorithm, a pronoun is selected for every index in the input semantics, which means any intermediary structure created during surface realisation can be combined with a pronoun, correctly or not. This is further aggravated because every lexical combination is explored and the number of intermediary structures is itself very large. The approach presented here addresses these two problems as follows. On the one hand, the wildcard effect is eliminated because index counting only allows pronouns to be selected if they can be associated with a *specific* "excess" index in the semantics. On the other hand, polarity filtering is used to drastically reduce the number of lexical combinations to be explored.

Bibliography

Bonfante, G., B. Guillaume, and G. Perrier (2003). Analyse syntaxique électrostatique. *Évolutions en analyse syntaxique, Revue TAL (Traitement Automatique des Langues) 44*(3).

Bonfante, G., B. Guillaume, and G. Perrier (2004). Polarization and abstraction of grammatical formalisms as methods for lexical disambiguation. In *Proceedings of CoLing 2004*.

Calder, J., M. Reape, and H. Zeevat (1989). An algorithm for generation in unification categorial grammar. In *Proceedings of the fourth conference on EACL*, pp. 233–240. ACL.

Carroll, J., A. Copestake, D. Flickinger, and V. Poznański (1999). An efficient chart generator for (semi-)lexicalist grammars. In *Proceedings of EWNLG '99*.

Copestake, A., A. Lascarides, and D. Flickinger (2001). An algebra for semantic construction in constraint-based grammars. In *Proceedings of the 39th ACL*, Toulouse, France.

Gardent, C. and E. Kow (2005). Generating and selecting grammatical paraphrases. *ENLG (in submission)*.

Hopcroft, J. and J. D. Ullman (1979). *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Publishing Company.

Joshi, A. and B. Srinivas (1994). Disambiguation of super parts of speech (or supertags): almost parsing. In *Proceedings of the 15th conference on Computational linguistics*, pp. 154–160. ACL.

Kay, M. (1996). Chart Generation. In *34th ACL*, Santa Cruz, California, pp. 200–204.

Koller, A. and K. Striegnitz (2002). Generation as dependency parsing. In *Proceedings of the 40th ACL*, Philadelphia.

Kow, E. (2004). Optimising a surface realiser. Master's thesis, Université de Nancy I.

Perrier, G. (2003). Les grammaires d'interaction. Habilitation à diriger les recherches en informatique, université Nancy 2.

Shemtov, H. (1996). Generation of paraphrases from ambiguous logical forms. In *COLING*, pp. 919–924.

Shieber, S. (1988). A uniform architecture for parsing and generation. In *Proceedings of the 12th conference on Computational linguistics*, pp. 614–619. ACL.

Vijay-Shanker, K. and A. Joshi (1988). Feature based tags. In *Proceedings of the 12th ACL*, Budapest, pp. 573–577.