SHARED PREFERENCES

James Barnett and Inderjeet Mani
MCC
3500 West Balcones Center Dr.
Austin, TX 78759
barnett@mcc.com
mani@mcc.com

Abstract

This paper attempts to develop a theory of heuristics or preferences that can be shared between understanding and generation systems. We first develop a formal analysis of preferences and consider the relation between their uses in generation and understanding. We then present a bidirectional algorithm for applying them and examine typical heuristics for lexical choice, scope and anaphora in more detail.

1 Introduction

Understanding and generation systems must both deal with ambiguity. In understanding, there are often a number of possible meanings for a string, while there are usually a number of different ways of expressing a given meaning in generation. To control the explosion of possibilities, researchers have developed a variety of heuristics or preferences - for example, a preference for low attachment of modifiers in understanding or for concision in generation. This paper investigates the possibility of sharing such preferences between understanding and generation as part of a bidirectional NL system. In Section 2 we formalize the concept of a preference, and Section 3 presents an algorithm for applying such preferences uniformly in understanding and generation. In Section 4 we consider specific heuristics for lexical choice, scope, and anaphora. These heuristics have special properties that permit a more efficient implementation than the general algorithm from Section 3. Section 5 discusses some of the short-comings of the theory developed here and suggests directions for future research.

2 Preferences in Understanding and Generation

Natural language understanding is a mapping from utterances to meanings, while generation goes in the opposite direction. Given a set *String* of input strings (of a given language) and a set *Int* of interpretations or meanings, we can represent understanding as a relation $U \subseteq String \times Int$, and generation as $G \subseteq Int \times String$. U and G are relations, rather than functions, since they allow for ambiguity: multiple meanings for an utterance and multiple ways of expressing a meaning[1]. A minimal requirement for a reversible system is that U and G be inverses of each other. For all $s \in String$ and $i \in Int$:

$$(s, i) \in U \leftrightarrow (i, s) \in G \qquad (1)$$

Intuitively, preferences are ways of controlling the ambiguity of U and G by ranking some interpretations (for U) or strings (for G) more highly than others. Formally, then, we can view preferences as total orders on the objects in question (we will capitalize the term when using it in this technical sense).[2] Thus, for any $s \in String$ an understanding Preference P_{int} will order the pairs $\{(s, i) | (s, i) \in U\}$, while a generation Preference

[1] The definitions of U and G allow for strings with no interpretations and meanings with no strings. Since any meaning can presumably be expressed in any language, we may want to further restrict G so that everything is expressible: $\forall i \in Int\ (\exists s \in String\ [(s, i) \in G])$.

[2] We use total orders rather than partial orders to avoid having to deal with incommensurate structures. The requirement of commensurability is not burdensome in practice, even though many heuristics apparently don't apply to certain structures. For example, a heuristic favoring low attachment of post-modifiers doesn't clearly tell us how to rank a sentence without post-modifiers, but we can insert such sentences into a total order by observing that they have all modifiers attached as low as possible.

P_{str} will rank $\{(i,s)|(i,s) \in G\}$[3]. Thus we can view the task of understanding as enumerating the interpretations of a string in the order given by P_{int}. Similarly, generation will produce strings in the order given by P_{str}. Using $U_{P_{int}}$ and $G_{P_{str}}$ to denote the result of combining U and G with these preferences, we have, for all $s \in String$ and $i \in Int$:

$$U_{P_{int}}(s) =_{def} (i_1, \ldots, i_n) \quad (2)$$
$$\text{where } U(s) = \{i_1, \ldots, i_n\} \text{ and}$$
$$[j < k] \rightarrow [(s, i_j) \leq_{P_{int}} (s, i_k)]$$

$$G_{P_{str}}(i) =_{def} (s_1 \ldots s_m) \quad (3)$$
$$\text{where } G(i) = \{s_1, \ldots, s_m\} \text{ and}$$
$$[j < k] \rightarrow [(i, s_j) \leq_{P_{str}} (i, s_k))]$$

Alternatively, we note that any Preference P induces an equivalence relation $\doteq_P$ which groups together the objects that are equal under P.[4] We can therefore view the task of Generation and Understanding as being the enumeration of P's equivalence classes in order, without worrying about order within classes (note that Formulae 2 and 3 specify the order only of pairs where one member is less than the other under P.)

The question now arises of what the relation between understanding Preferences and generation Preferences should be. Understanding heuristics are intended to find the meaning that the speaker is most likely to have intended for an utterance, and generation heuristics should select the string that is most likely to communicate a given meaning to the hearer. We would expect these Preferences to be inverses of each other: if s is the best way to express meaning i, then i should be the most likely interpretation of s. If we don't accept this condition, we will generate sentences that we expect the listener to misinterpret. Therefore we define $class(Preference, pair)$ to be the equivalence class that $pair$ is assigned to under $Preference$'s ordering,[5] and link the the first (most highly ranked) classes under P_{int} and P_{str} as follows:

$$class(P_{str}, (i, s)) = 0 \quad (4)$$
$$\rightarrow class(P_{int}, (s, i)) = 0$$

It is also reasonable to require that opposing sets of preferences in understanding be reflected in generation. If string s_1 has two interpretations i_1 and i_2, with i_1 being preferred to i_2, and string s_2 has the same two interpretations with the preferences reversed, then s_1 should be a better way of expressing i_1 than i_2, and vice-versa for s_2:

$$\begin{array}{rcl} [\,(s_1, i_1) & <_{P_{int}} & (s_1, i_2) \\ & \& & \\ (s_2, i_2) & <_{P_{int}} & (s_2, i_1)\,] \\ & \leftrightarrow & \\ [\,(i_1, s_1) & <_{P_{str}} & (i_1, s_2) \\ & \& & \\ (i_2, s_2) & <_{P_{str}} & (i_2, s_1)\,] \end{array} \quad (5)$$

Formula 4 provides a tight coupling of heuristics for understanding and generating the most preferred structures, but it doesn't provide any way to share Preferences for secondary readings. Formula 5 offers a way to share heuristics for secondary interpretations, but it is quite weak and would be highly inefficient to use. To employ it during generation to choose between s_1 and s_2 as ways of expressing i_1, we would have to run the understanding system on both s_1 and s_2 to see if we could find another interpretation i_2 that both strings share but with opposite rankings relative to i_1.

If we want to share Preferences for secondary readings, we will need to make stronger assumptions. The question of ranking secondary interpretations brings us onto treacherous ground since most common heuristics (e.g., preferring low attachment) specify only the best reading and don't help choose between secondary and tertiary readings. Furthermore, native speakers don't seem to have clear intuitions about the relative ranking of lesser readings. Finally, there is some question about why we should care about non-primary readings, since the best interpretation or string is normally what we want. However, it is important to deal with secondary preferences, in part for systematic completeness, but mostly because secondary readings are vital in any attempt to deal with figurative language - humor, irony, and metaphor - which depends on the interplay between primary and secondary readings.

[3] Note that this definition allows Preferences to work 'across derivations.' For example, it allows P_{int} to rank pairs (s, i), (s', i') where $s \neq s'$. It permits a Preference to say that i is a better interpretation for s than i' is for s'. It is not clear if this sort of power is necessary, and the algorithms below require only that Preferences be able to rank different interpretations (strings) for a given string (interpretation).

[4] Any order P on a set of objects D partitions D into a set of equivalence classes by assigning each $x \in D$ to the set $\{y|y \leq_P x \,\&\, x \leq_P y\}$.

[5] $class(Preference, pair)$ is defined as the number of classes containing items that rank more highly than $pair$ under $Preference$.

To begin to develop a theory of secondary Preferences, we will simply stipulate that the heuristics in question are shared 'across the board' between understanding and generation. The simplest way to do this is to extend Formula 4 into a biconditional, and require it to hold of all classes (we will reconsider this stipulation in Section 5). For all $s \in String$ and $i \in Int$, we have:

$$class(P_{int}, (s, i)) = class(P_{str}, (i, s)) \quad (6)$$

Since Preferences now work in either direction, we can simplify our notation and represent them as total orderings of a set T of trees, where each node of each tree is annotated with syntactic and semantic information, and, for any $t \in T$, $str(t)$ returns the string in $String$ that t dominates (i.e., spans), and $sem(t)$ returns the interpretation in Int for the root node of t. For a preference P on T and trees t_1, t_2, we stipulate:

$$t_1 <_P t_2 \quad \& \quad str(t_1) = str(t_2) \quad (7)$$
$$\leftrightarrow$$
$$U_P(str(t_1)) = (\ldots sem(t_1) \ldots sem(t_2) \ldots)$$

$$t_1 <_P t_2 \quad \& \quad sem(t_1) = sem(t_2) \quad (8)$$
$$\leftrightarrow$$
$$G_P(sem(t_1)) = (\ldots str(t_1) \ldots str(t_2) \ldots)$$

We close this section by noting a property of Preferences that will be important in Section 4: an ordered list of Preferences can be combined into a new Preference by using each item in the list to refine the ordering specified by the previous ones. That is, the second Preference orders pairs that are equal under the first Preference, and the third Preference applies to those that are still equal under the second Preference, etc. If $P_1 \ldots P_n$ are Preferences, we define a new Complex Preference $P_{<1 \ldots n>}$ as follows:

$$t_1 <_{P_{(P_1 \ldots P_n)}} t_2 \quad (9)$$
$$\leftrightarrow \exists\, 1 \leq j \leq n\, [t_1 <_{P_j} t_2]$$
$$\& \neg \exists\, i < j\, [t_2 <_{P_i} t_1]$$

3 An Algorithm for Sharing Preferences

If we consider ways of sharing Preferences between understanding and generation, the simplest one is to simply produce all possible interpretations(strings), and then sort them using the Preference. This is, of course, inefficient in cases where we are interested in only the more highly ranked possibilities. We can do better if we are willing to make few assumptions about the structure of Preferences and the understanding and generation routines. The crucial requirement on Preferences is that they be 'upwardly monotonic' in the following sense: if t_1 is preferred to t_2, then it is also preferred to any tree containing t_2 as a subtree. Using $subtree(t_1, t_2)$ to mean that t_1 is a subtree of t_2, we stipulate

$$[t_1 <_P t_2 \quad \& \quad subtree(t_2, t_3)] \quad (10)$$
$$\rightarrow \quad t_1 <_P t_3$$

Without such a requirement, there is no way to cut off unpromising paths, since we can't predict the ranking of a complete structure from that of its constituents.

Finally, we assume that both understanding and generation are agenda-driven procedures that work by creating, combining, and elaborating trees.[6] Under these assumptions, the following high-level algorithm can be wrapped around the underlying parsing and generation routines to cause the output to be enumerated in the order given by a Preference P. In the pseudo-code below, *mode* specifies the direction of processing and *input* is a string (if *mode* is understanding) or a semantic representation (if *mode* is generation). *execute_item* removes an item from the agenda and executes it, returning 0 or more new trees. *generate_items* takes a newly formed tree, a set of previously existing trees, and the mode, and adds a set of new actions to the agenda. (The underlying understanding or generation algorithm is hidden inside *generate_items*.) The variable *active* holds the set of trees that are currently being used to generate new items, while *frozen* holds those that won't be used until later. *complete_tree* is a termination test that returns True if a tree is complete for the mode in question (i.e., if it has a full semantic interpretation for understanding, or dominates a complete string for generation). The global variable *classes* holds a list of equivalence classes used by *equiv_class* (defined below), while *level* holds the number of the equivalence class currently being enumerated. *Thaw&restart* is called each time *level* is incremented to generate new agenda items for trees that may belong to that class.

ALGORITHM 1

[6] A wide variety of NLP algorithms can be implemented in this manner, particularly such recent reversible generation algorithms as [Shieber, van Noord, Moore, and Pereira, 1989] and [Calder, Reape, and Zeevat, 1989].

```
classes := Nil; solutions := Nil;
new-trees := Nil; agenda := Nil;
level := 1;
frozen := initialize_agenda(input, mode);
{end of global declarations}
while frozen do
 begin
  solutions := get_complete_trees
                  (frozen, level, mode);
  agenda := thaw&restart
            (frozen, level, agenda, mode);
  while agenda do
   begin
   new_trees := execute_item(agenda);
   while new_trees do
    begin
     new_tree := pop(new_trees);
     if equiv_class (P, new_tree)
                                 > level
     then push(new_tree, frozen);
     else if  complete_tree
                  (new_tree,mode)
       then push(new_tree, solutions);
     else generate_items
             (new_tree, active,
                 agenda, mode);
    end;
   end; {agenda exhausted for this level}
  {solutions may need partitioning}
  while  solutions do
    begin
     complete_tree := pop(solutions);
     if equiv_class(P, complete_tree)
                               > level
     then push(complete_tree, frozen);
     else output(complete_tree, level) ;
    end
  {increment level to output next class}
  level := level + 1;
 end;
```

The function *equiv_class* keeps track of the equivalence classes induced by the Preferences. Given an input tree, it returns the number of the equivalence class that the tree belongs to. Since it must construct the equivalence classes as it goes along, it may return different values on different calls with the same argument (for example, it will always return 1 the first time it is called, even though the tree in question may end up in a lower class.) However, successive calls to *equiv_class* will always return a non-decreasing series of values, so that a given tree is guaranteed to be ranked no more highly than the value returned (it is this property of *equiv_class* that forces the extra pass over the completed trees in the algorithm above: a tree that was assigned to class n when it was added to *solutions* may have been demoted to a lower class in the interim as more trees were examined). *Less_than* and *equal* take a Preference and a pair of trees and return True if the first tree is less than (equal to) the second under the Preference. *Create_class* takes a tree and creates a new class whose only member is that tree, while *insert* adds a class to *classes* in the indicated position (shifting other classes down, if necessary), and *select_member* returns an arbitrary member of a class.

```
function equiv_class (P: Preference, T: Tree)
begin
  class_num := 1;
  for class in  classes do
   begin
     if less_than
         (P, T, select_member(class))
     then
     begin
      insert(new_class(T),
            classes, class_num);
      return(class_num);
     end;
     else if equal
              (P, T, select_member(class))
     then
       begin
       add_member(T, class);
       return(class_num);
       end;
   else class_num := class_num + 1;
  end ;
  {T < all classes}
  insert(new_class(T),
            classes, class_num);
  return(class_num);
 end {equiv_class}
```

To see that the algorithm enumerates trees in the order given by $<_P$, note that the first iteration outputs trees which are minimal under $<_P$. Now consider any tree t_n which is output on a subseqent itertion N. For all other $t_{n'}$ output on that iteration, $t_n =_P t_{n'}$. Furthermore, t_n contains a subtree t_{sub} which was frozen for all levels up to N. Using $T(J)$ to denote the set of trees output on iteration J, we have: $\forall\, 1 \leq I < N$ $[\forall\, t_i \in T(I)\ t_i <_P t_{sub}]]$, whence, by stipulation 10, $t_n <_P t_i$. Thus t_n is greater than or equal to

all trees which were enumerated before it. To calculate the time complexity of the algorithm, note that it calls *equiv_class* once for each tree created by the underlying understanding or generation algorithm (and once for each complete interpretation). *Equiv_class*, in turn, must potentially compare its argument with each existing equivalence class. Assuming that the comparison takes constant time, the complexity of the algorithm depends on the number k of equivalence classes $<_P$ induces: if the underlying algorithm is $O(f(n))$, the overall complexity is $O(f(n)) \times k$. Depending on the Preference, k could be a small constant, or itself proportional to $f(n)$, in which case the complexity would be $O(f(n)^2)$.

4 Optimization of Preferences

As we make more restrictive assumptions about Preferences, more efficient algorithms become possible. Initially, we assumed only that Preferences specified total orders on trees, i.e., that would take two trees as input and determine if one was less than, greater than, or equal to the other[7]. Given such an unrestricted view of Preferences, we can do no better than producing all interpretations(strings) and then sorting them. This simple approach is fine if we want all possibilities, especially if we assume that there won't be a large number of them, so that standard n^2 or $n \log n$ sorting algorithms (see [Aho, Hopcroft, and Ullman, 1983]) won't be much of an additional burden. However, this approach is inefficient if we are interested in only some of the possibilities. Adding the monotonicity restriction 10 permits Algorithm 1, which is more efficient in that it postpones the creation of (successors of) lower ranked trees. However, we are still operating with a very general view of what Preferences are, and further improvements are possible when we look at individual Preferences in detail. In this section, we will consider heuristics for lexical selection, scope, and anaphor resolution. We do not make any claims for the usefullness of these heuristics as such, but take them as concrete examples that show the importance of considering the computational properties of Preferences.

Note that Algorithm 1 is stated in terms of a single Preference. It is possible to combine multiple Preferences into a single one using Formula 9, and we are currently investigating other methods of combination. Since the algorithms below are highly specialized, they cannot be combined with other Preferences using Formula 9. The ultimate goal of this research, however, is to integrate such specialized algorithms with a more sophisticated version of Algorithm 1.

[7] We also assume that this test takes constant time.

4.1 Lexical Choice

One simple preferencing scheme involves assigning integer weights to lexical items and syntactic rules. Items or rules with higher weights are less common and are considered only if lower ranked items fail. When combined with restriction 10, this weighting scheme yields a Preference $<_{wt}$ that ranks trees according to their lexical and rule weights. Using $max\_wt(T)$ to denote the most heavily weighted lexical item or rule used in the construction of T, we have:

$$t_1 <_{wt} t_2 \leftrightarrow_{def} max\_wt(t_1) < max\_wt(t_2) \quad (11)$$

The significant property here is that the equivalence classes under $<_{wt}$ can be computed without directly comparing trees. Given a lexical item with weight n, we know that any tree containing it must be in class n or lower. Noting that our algorithm works by generate-and-test (trees are created and then ranked by *equiv_class*), we can achieve a modest improvement in efficiency by not creating trees with level n lexical items or rules until it is time to enumerate that equivalence class. We can implement this change for both generation and understanding by adding *level* as a parameter to both *initialize_agenda* and *generate_items*, and changing the functions they call to consider only rules and lexical items at or below *level*. How much of an improvement this yields will depend on how many classes we want to enumerate and how many lexical items and rules there are below the last class enumerated.

4.2 Scope

Scope is another place where we can improve on the basic algorithm. We start by considering scoping during Understanding. Given a sentence s with operators (quantifiers) $o_1 \ldots o_n$, assigning a scope amounts to determining a total order on $o_1 \ldots o_n$[8]. If a scope Preference can do

[8] Note that this ordering is not a Preference. A Preference will be a total ordering of trees, each of which contains such a scope ordering, i.e., a scope Preference will be an ordering of orderings of operators.

no more than compare and rank pairs of scopings, then the simple generate-and-test algorithm will require $O(n!)$ steps to find the best scoping since it will potentially have to examine every possible ordering. However, the standard heuristics for assigning scope (e.g., give "strong" quantifiers wide scope, respect left-to-right order in the sentence) can be used to directly assign the preferred ordering of $o_1 \ldots o_N$. If we assume that secondary readings are ranked by how closely they match the preferred scoping, we have a Preference $<_{sc}$ can be defined. In the following $(o_i, o_j) \in Sc(s)$ means that o_i preceeds o_j in scoping Sc of sentence s, and $Sc_{best}(s)$ is the preferred ordering of the operators in s given by the heuristics:

$$Sc_1(s) \leq_{sc} Sc_2(s) \leftrightarrow_{def} \quad (12)$$
$$\forall o_i, o_j \; [(o_i, o_j) \in Sc_{best}(s) \rightarrow$$
$$[(o_i, o_j) \in Sc_2(s) \rightarrow (o_i, o_j) \in Sc_1(s)]\,]$$

Given such a Preference, we can generate the scopings of a sentence more efficiently by first producing the preferred reading (the first equivalence class), then all scopes that have one pair of operators switched (the second class), then all those with two pairs out of order, etc. In the following algorithm, *ops* is the set of operators in the sentence, and *sort* is any sorting routine. *switched?* is a predicate returning True if its two arguments have already been switched (i.e., if its first arg was to the right of its second in $Sc_{best}(s)$), while *switch(o_1, o_2, ord)* is a function that returns new ordering which is the same as *ord* except that o_2 precedes o_1 in it.

```
{the best scoping}
root_set := sort(operators, Sc_best(s));
level := 1;
output(root_set, level);
new_set := Nil;
old_set := add_item(root_set, Nil);
{loop will execute n! - 1 times }
while old_set do
begin
  for ordering in old_set do
  begin
    for op in ordering do
    begin
      {consider adjacent pairs of operators}
      next := right_neighbor(op, ordering);
      {switch any pair that hasn't already been}
      if next and not(switched?(op, next))
      then do
        begin
        new_scope := switch(op, next, ordering);
        add_item(new_scope, new_set);
        output(new_scope, level) ;
        end
    end
  end
old_set := new_set;
new_set := Nil;
end
```

While the Algorithm 1 would require $O(n!)$ steps to generate the first scoping, this algorithm will output the best scoping in the n^2 or $n \log n$ steps that it takes to do the sort (cf [Aho, Hopcroft, and Ullman, 1983]), while each additional scoping is produced in constant time.[9] The algorithm is profligate in that it generates all possible orderings of quantifiers, many of which do not correspond to legal scopings (see [Hobbs and Shieber, 1987]). It can be tightened up by adding a legality test before scope is output.

When we move from Understanding to Generation, following Formula 6, we see that the task is to take an input semantics with scoping Sc and enumerate first all strings that have Sc as their best scoping, then all those with Sc as the second best scoping, etc. Equivalently, we enumerate first strings whose scopings exactly match Sc, then those that match Sc except for one pair of operators, then those matching except for two pairs, etc. We can use the Algorithm 1 to implement this efficiently if we replace each of the two conditional calls to *equiv_class*. Instead of first computing the equivalence class and then testing whether it is less than *level*, we call the following function *class_less_than*:

```
{True iff candidate ranked at level or below}
{Target is the desired scoping}
function class_less_than(candidate, target, level)
 begin
  switch_limit := level; {global variable}
  switches := 0; {global variable}
  return test_order(candidate, target, target);
 end {class_less_than }

function test_order(cand, targ_rest, targ)
begin
 if null(cand)
  return True;
 else
```

[9] *switched?* can be implemented in constant time if we record the position of each operator in the original scoping Sc_{best}. Then *switched?*(o_1, o_2) returns True iff *position*(o_2) < *position*(o_1).

```
 begin
   targ_tail := member(first(cand), targ_rest);
   if targ_tail
    return test_order(rest(cand), targ_tail, targ);
   else
   begin
    switches := switches + 1;
    if >(switches, switch_limit)
     return False;
   end
   else
    if (simple_test(rest(cand), targ_rest)
     return test_order(cand, targ, targ);
    else return False;
 end
end {test_order}

function simple_test(cand_rest, targ_rest)
begin
 for cand in cand_rest do
  begin
  if not(member(cand, targ_rest))
   begin
    switches := switches + 1;
    if >(switches, switch_limit)
     return false;
   end
  end
 return true;
end {simple_test}
```

To estimate the complexity of *class_less_than*, note that if no switches are encountered, *test_order* will make one pass through *targ_rest* (= *targ*) in $O(n)$ steps, where n is the length of *targ*. Each switch encountered results in a call to *simple_test*, $O(n)$ steps, plus a call to *test_arg* on the full list *targ* for another $O(n)$ steps. The overall complexity is thus $O((j+1) \times n)$, where $level = j$ is the number switches permitted. Note that *class_less_than* tests a candidate string's scoping only against the target scope, without having to inspect other possible strings or other possible scopings for the string. We therefore do not need to consider all strings that can have Sc as a scoping in order to find the most highly ranked ones that do. Furthermore, *class_less_than* will work on partial constituents (it doesn't require that *cand* have the same number of operators as *targ*), so unpromising paths can be pruned early.

4.3 Anaphora

Next we consider the problem of anaphoric reference. From the standpoint of Understanding, resolving an anaphoric reference can be viewed as a matter of finding a Preference ordering of all the possible antecedents of the pronoun. Algorithm 1 would have to produce a separate interpretation for each object that had been mentioned in the discourse and then rank them all. This would clearly be extremely inefficient in any discourse more than a couple of sentences long. Instead, we will take the anaphora resolution algorithm from [Rich and Luperfoy, 1988], [Luperfoy and Rich, 1991] and show how it can be viewed as an implementation of a Complex Preference, allowing for a more efficient implementation.

Under this algorithm, anaphora resolution is entrusted to Experts of three kinds: a Proposer finds likely candidate antecendents, Filters provide a quick way of rejecting many candidates, and Rankers perform more expensive tests to choose among the rest. Recency is a good example of a Proposer; antecedents are often found in the last couple of sentences, so we should start with the most recent sentences and work back. Gender is a typical Filter; given a use of "he", we can remove from consideration all non-male objects that the Proposers have offered. Semantic plausibility or Syntactic parallelism are Rankers; they are more expensive than the Filters and assign a rational-valued score to each candidate rather than giving a yes/no answer.

When we translate these experts into our framework, we see that Proposers are Preferences that can efficiently generate their equivalence classes in rank order, rather than having to sort a pre-existing set of candidates. This is where our gain in efficiency will come: we can work back through the Proposer's candidates in order, confident that any candidates we haven't seen must be ranked lower than those we have seen. Filters represent a special class of Preference that partition candidates into only two classes: those that pass and those that are rejected. Furthermore, we are interested only in candidates that *all* filters assign to the first class. If we simply combine n Filters into a Complex Preference using Formula 9, the result is not a Filter since it partitions the input into 2^n classes. We therefore define a new simple Filter $F_{(f_1 \ldots f_n)}$ that assigns its input to class 1 iff $F_1 \ldots F_n$ all do. Finally, Rankers are Preferences of the kind we've been discussing so far. When we observe that the effect of running a Proposer and then removing all candidates that the Filters reject is equivalent to first running the Filter and then using the Proposer to refine its first

class[10], we see that the algorithm above, when run with Proposer Pr, Filters $F_1 \ldots F_n$ and Rankers $R_1 \ldots R_j$, implements the Complex Preference $P_{(F_{(f_1 \ldots f_n)}, Pr, R_1 \ldots R_j)}$, defined in accordance with Formula 9. We thus have the following algorithm, where *next_class* takes a Proposer and a pronoun as input and returns its next equivalence class of candidate antecedents for the pronoun.

```
class := 1; {global variable}
cand := next_class(Proposer, pronoun);
filtered_cand := cand;
while (cand) do
begin
  for cand in  cands do
  begin
   for  filter in  Filters do
   begin
    if not(Filter(cand))
     then remove(cand, filtered_cand);
   end
  end
  {filtered_cand now contains class n under}
  {P(F(f1...fn),Pr). Rankers R1...Rj}
  {may split it into several classes}
  refine&output(filtered_cand, Rankers);
  cand := next_class(Proposer);
  end


function Refine&Output(cands, Rankers)
begin
 refined_order := sort(cands, Rankers);
 if rest(Rankers)
 then refine&output(refined_order,
                                rest(Rankers));
 else
 begin
   loc_class := 1; for cand in refined_order do
      if >(equiv_class
                  first(Rankers), cand),
      loc_class)
      then
      begin
       loc_class := loc_class + 1;
       class := class + loc_class;
      end
      output(cand, class);
 end
end {Refine&Output}
```

Moving to Generation, we use this Preference to decide when to use a pronoun. Following Formula 6, we want to use a pronoun to refer to object x at level n iff that pronoun would be interpreted as referring to x in class n during Understanding. First we need a test *occurs?(Proposer, x)* that will return True iff *Proposer* will eventually output x in some equivalence class. For example, a Recency Proposer will never suggest a candidate that hasn't occurred in the antecedent discourse, so there is no point in considering a pronoun to refer to such an object. Next, we note that the candidates that the Proposer returns are really pairs consisting of a pronoun and an antecedent, and that Filters work by comparing the features of the pronoun (gender, number, etc.) with those of the antecedent. We can implement Filters to work by unifying the (syntactic) features of the pronoun with the (syntactic and semantic) features of the antecedent, returning either a more fully-specified set of features for the pronoun, or $\perp$ if unification fails. We can now take a syntactically underspecified pronoun and x and use the Filter to choose the appropriate set of features. We are now assured that the Proposer will suggest x at some point, and that x will pass all the filters.

Having established that x is a reasonable candidate for pronominal reference, we need to determine what class x will be assigned to as an antecedent. Rankers such as Syntactic Parallelism must look at the full syntactic structure[11], so we must generate complete sentences before doing the final ranking. Given a sentence s contaning pronoun p with antecedent x, we can determine the equivalence class of (p, x) by running the Proposer until it (p, x) appears, then running the Filters on all other candidates, and passing all the survivors and (p, x) to *refine&output*, and then seeing what class (p, x) is returned in. Alternatively, if we only want to check whether (p, x) is in a certain class n or not, we can run the resolution algorithm given above until n classes have been enumerated, quitting if (p, x) is not in it. (See the next section for a discussion of this algorithm's obvious weaknesses.)

[10] In both cases, the result is: $p_1 \cap f_1, \ldots p_n, \cap f_1$, where $p_1 \ldots p_n$ are the equivalence classes induced by the Proposer, and f_1 is the Filter's first equivalence class.

[11] The definitions we've given so far do not specify how Preferences should rank "unfinished" structures, i.e., those that don't contain all the information the Preference requires. One obvious solution is to assign incomplete structures to the first equivalence class; as the structures become complete, they can be moved down into lower classes if necessary. Under such a strategy, Preferences such as Syntactic Parallelism will return high scores on the incomplete constituents, but these scores will be meaningless, since many of the resulting complete structures will be placed into lower classes.

5 Discussion

Related Work: There is an enormous amount of work on preferences for understanding, e.g., [Whittemore, Ferrara, and Brunner, 1990], [Jensen and Binot, 1988], [Grosz, Appelt, Martin, and Pereira, 1987] for a few recent examples. In work on generation preferences (in the sense of rankings of structures) are less clearly identifiable since such rankings tend to be contained implicitly in strategies for the larger problem of deciding what to say (but see [Mann and Moore, 1981] and [Reiter, 1990].) Algorithm 1 is similar in spirit to the "all possibilities plus constraints" strategy that is common in principle-based approaches (see [Epstein, 1988]), but it differs from them in that it imposes a preference ordering on interpretations, rather than restricting the set of legal interpretations to begin with.

Strzalkowski [Strzalkowski, 1990] contrasts two strategies for reversibility: those with a single grammar and two intepreters versus those with a single interpreter and two grammars. Although the top-level algorithm presented here works for both understanding and generation, the underlying generation and understanding algorithms can belong to either of Strzalkowski's categories. However, the more specific algorithms discussed in Section 4 belong to the former category. There is also a clear "directionality" in both the scope and the anaphora Preferences; both are basically understanding heuristics that have been reformulated to work bi-directionally. For this reason, they are both considerably weaker as generation heuristics. In particular, the anaphora Preference is clearly insufficient as a method of choosing when to use a pronoun. At best, it can serve to validate the choices made by a more substantial planning component.

The Two Directions: In general, it is not clear what the relation between understanding and generation heuristics should be. Formulae 4 and 5 are reasonable requirements, but they are too weak to provide the close linkage between understanding and generation that we would like to have in a bi-directional system. On the other hand, Formula 6 is probably too strong since it requires the equivalence classes to be the same across the board. In particular, it entails the converse of Formula 4, and this has counter-intuitive results. For example, consider any highly convoluted, but grammatical, sentence: it has a best interpretation, and by Formula 6 it is therefore one of the best ways of expressing that meaning. But if it is sufficently opaque, it is not a good way of saying *anything*. Similarly, a speaker may suddenly use a pronoun to refer to an object in a distant part of the discourse. If the anaphora Preference is sophisticated enough, it may resolve the pronoun correctly, but we would not want the generation system to conclude that it should use a pronoun in that situation. One way to tackle this problem is to observe that understanding systems tend to be too loose (they accept a lot of things that you don't want to generate), while generation systems are too strict (they cover only a subset of the language.) We can therefore view generation Preferences as restrictions of understanding Preferences. On this view, one may construct a generation Preference from one for understanding by adding extra clauses, with the result that its ordering is a refinement of that induced by the understanding Preference.

Internal Structure: Further research is necessary into the internal structure of Preferences. We chose a very general definition of Preferences to start with, and found that further restrictions allowed for improvements in efficiency. Preferences that partition input into a fixed set of equivalence classes that can be determined in advance (e.g., the Preference for lexical choice discussed in Section 4) are particularly desireable since they allow structures to be categorized in isolation, without comparing them to other alternatives. Other Preferences, such as the scope heuristic, allow us to create the desired structures directly, again without need for comparison with other trees. On the other hand, the anaphora Preference is based on an algorithm that assigns rational-valued scores to candidate antecedents. Thus there can be arbitrarily many equivalence classes, and we can't determine which one a given candidate belongs to without looking at all higher-ranked candidates. This is not a problem during understanding, since the Proposer can provide those candidates efficiently, but the algorithm for generation is quite awkward, amounting to little more than "make a guess, then run understanding and see what happens."

The focus of our future research will be a formal analysis of various Preferences to determine the characteristic properties of good understanding and generation heuristics and to investigate methods other than Formula 9 of combining multiple Preferences. Given such an analysis, Algorithm 1 will be modified to handle multiple Preferences and to treat the different types of Preferences differently, thus reducing the need for

the kind of heuristic-specific algorithms seen in Section 4. We also plan an implementation of these Preferences as part of the KBNL system [Barnett, Mani, Knight, and Rich, 1990].

References

[Aho, Hopcroft, and Ullman, 1983] Alfred Aho, John Hopcroft, and Jeffrey Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[Barnett, Mani, Knight, and Rich, 1990] Jim Barnett, Inderjeet Mani, Kevin Knight, and Elaine Rich. Knowledge and natural language processing. *CACM*, August 1990.

[Calder, Reape, and Zeevat, 1989] J. Calder, M. Reape, and H. Zeevat. An algorithm for generation in unification categorial grammar. In *Proceedings of the 4th conference of the European Chapter of the ACL*, 1989.

[Epstein, 1988] Samuel Epstein. Principle-based interpretation of natural language quantifiers. In *Proceedings of AAAI 88*, 1988.

[Grosz, Appelt, Martin, and Pereira, 1987] Barbara Grosz, Douglas Appelt, Paul Martin, and Fernando Pereira. Team: an experiment in the design of portable natural language interfaces. *Artificial Intelligence*, 1987.

[Hobbs and Shieber, 1987] Jerry Hobbs and Stuart Shieber. An algorithm for generating quantifier scopings. *Computational Linguistics*, 13(1-2):47–63, 1987.

[Jensen and Binot, 1988] Karen Jensen and Jean-Louis Binot. Dictionary text entries as a source of knowledge for syntactic and other disambiguations. In *Second Conference on Applied Natural Language Processing*, Austin, Texas, 9-12 February, 1988.

[Luperfoy and Rich, 1991] Susan Luperfoy and Elaine Rich. Anaphora resolution. *Computational Linguistics*, to appear.

[Mann and Moore, 1981] William Mann and James Moore. Computer generation of multiparagraph english text. *American Journal of Computational Linguistics*, 7(1):17–29, 1981.

[Reiter, 1990] Ehud Reiter. The computational complexity of avoiding conversational implicatures. In *Proceedings of the ACL*, Pittsburgh, 6-9 June, 1990.

[Rich and Luperfoy, 1988] Elaine Rich and Susan Luperfoy. An architecture for anaphora resolution. In *Second Conference on Applied Natural Language Processing*, Austin, Texas, 9-12 February, 1988.

[Shieber, van Noord, Moore and Pereira, 1989] S. Shieber, G. van Noord, R. Moore, and F. Pereira. A semantic head-driven generation algorithm for unification-based formalisms. In *Proceedings of the ACL*, Vancouver, 26-29 June, 1989.

[Strzalkowski, 1990] Tomek Strzalkowski. Reversible logic grammars for parsing and generation. *Computational Intelligence*, 6(3), 1990.

[Whittemore, Ferrara, and Brunner, 1990] Greg Whittemore, Kathleen Ferrara, and Hans Brunner. Post-modifier prepositional phrase ambiguity in written interactive dialogues. In *Proceedings of the ACL*, Pittsburgh, 6-9 June, 1990.