A TOOL FOR COLLECTING DOMAIN DEPENDENT SORTAL CONSTRAINTS FROM CORPORA

François Andry*, Mark Gawron, John Dowding, and Robert Moore

SRI International, Menlo Park, CA
*CAP GEMINI Innovation, Boulogne Billancourt, France
Internet: andry@capsogeti.fr

Topical paper : Tools for NL Understanding (Portability).

1 ABSTRACT

In this paper, we describe a tool designed to generate semi-automatically the sortal constraints specific to a domain to be used in a natural language (NL) understanding system. This tool is evaluated using the SRI Gemini NL understanding system in the ATIS domain.

2 INTRODUCTION

The construction of a knowledge base related to a specific domain for a NL understanding system is time consuming. In the Gemini system, the domain-specific knowledge base includes a *sort hierarchy* and a set of *sort rules* that provide (largely domain-specific) selectional restrictions for every predicate invoked by the lexicon and the grammar. The selectional restrictions provide a source of constraints over and above syntactic constraints for choosing the correct analysis of a sentence. The sort rules are generally entered by a linguist, by hand, from the study of a corpus and while tuning the grammar.

However, the use of an interactive tool that can help the linguist to acquire this knowledge from a corpus[3][5], can drastically reduce the time dedicated to this task, and also improve the quality of the knowledge base in terms of both accuracy and completeness. The reduction in the amount of effort to develop the knowledge base becomes obvious when porting an existing system to a new domain. At SRI, our main concern was to port Gemini, our NL understanding system to other domains without investing the same amount of work we put into the first domain application[1].

In this paper, we describe the results of using this semi-automatic tool to port the Gemini NL system to the ATIS domain, a domain that Gemini had already been ported to, and for which it had achieved high performance and grammatical coverage using hand-written sortal constraints. Chossing a known domain, rather than a new one, allowed us to compare the performance of the derived sorts to the hand-written ones, holding the domain, grammar, and lexicon constant. It also allowed us to evaluate the semi-automatically obtained coverage using the evaluation tools provided for the ATIS corpus.

3 PARSING WITH SORTS

Gemini[2] implements a clear separation between syntactic and semantic information. Each syntactic node invokes a set of semantic rules which result in the building of a set of logical forms for that node. Selectional restrictions are enforced on the logical forms through the sorts mechanism: All predications in a candidate logical form must be licensed by some sorts rule. The sorts are located in a conceptual hierarchy of approximately 200 concepts and are implemented as Prolog terms such that more general sorts subsume more specific sorts[6]. Failure to match any available sorts rule can thus be implemented as unification-failure.

Gemini parser creates logical forms expressions like the following one :

$exists((A;[flight])$,
$[and,[flight,(A;[flight])];[prop]$,
$[to,(A;[flight])$,
$('BOSTON';[city])];[prop]];[prop]);[prop]$

In these logical form expressions, every subexpression is assigned a sort, represented as the

[1]The actual domain is Air Transportation (ATIS) used as a benchmark in the ARPA community.

right-hand-side of a ';' operator[1]. Sorts rules for predicates are declared with sor/2 clauses:

$$\texttt{sor}('BOSTON', [city]).$$
$$\texttt{sor}(to, ([[flight]], [city]], [prop])).$$

The above declarations license the use of 'BOSTON' as a zero-ary predicate with "resulting" sort [*city*] and 'to' as a two-place predicate relating flights and cities with resulting sort [*prop*] (or proposition).

In the ATIS application domain, for example, the subject (or actor) of the verb *depart*, as in '*the morning flights departing for denver*', can be a flight. For this, we use the following set of sort definitions:

$$sor(depart, ([[departure]], [prop]))$$
$$sor(flight, ([[flight]], [prop]))$$
$$sor(actor, ([[departure], [flight]], [prop]))$$

The first two definitions make *depart* and *flight* predicates compatible with departure and flight events respectively, returning a proposition; the third makes *actor* a relation that can hold between flights and flights, also returning a proposition. A simple example of a logical form licensed by these rules follows (with the result sort [*prop*] suppressed):

$qterm(some, ((X; [flight]),$
  $[and, [flight, (X; [flight])],$
    $exists((Y; [flight]),$
      $[and, [depart, (Y; [departure])],$
        $(Y; [departure]),$
        $[actor, (Y; [departure]), (X; [flight])])])])$

Which would be roughly the logical form for '*a departing flight*'.

4 SORT ACQUISITION

The approach we have taken is to start from an initial "schematic" sorts file we call the signature file (explained below), which essentially allows all predicate argument combinations. We then harvest a set of preliminary sort rules by parsing a large corpus. The logical forms that induce these preliminary rules come from parses that essentially incorporate only syntactic constraints. The resulting sorts rules are filtered by hand and the process is iterated with an increasingly accurate sorts file, converging rapidly on the sorts file specific to the application domain (fig. 1).

4.1 Signature and Restrictions

If we started the above iteration process with *no* sortal information, then the logical forms resulting from a parse would contain no sortal information, and only vacuous sortal rules would be harvested.

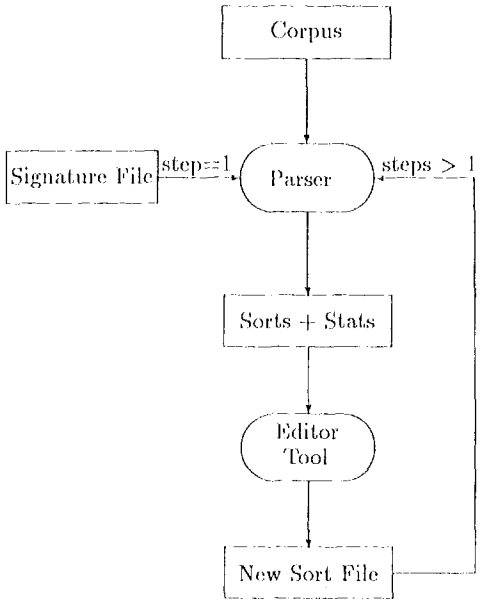

Figure 1: Iterative Acquisition of Sorts.

The first step is thus to build an initial sort file we call the *signature* file. The idea is to assign lexical predicates inherent sorts, but not to assign assign any rules which constrain which lexical items can combine with which. The signature file, then, is not just domain-independent. It has no information at all about semantic combinatorial possibilities, not even those determined by the language (for example, that the verb *break* does not allow propositional subjects). The reason for this is so that it can be generated largely automatically from the lexicon.

4.2 The Signature

Lets begin with certain inherently relational predicates, for which the signature file gives only an arity and the result sort. For example the signature for the predicates *at* (corresponding to the preposition) and *actor* (corresponding to logical subject) would be the same:

$$signature(at, ([X, Y], [prop])$$
$$signature(actor, ([X, Y], [prop])$$

This signature is used as the sort rule for *at* and *actor* in the sorts tool's first iteration. The effect is to limit the choice of sorts rules for these predicates to rules which are further instantiations their signatures, that is, to rules licensing them to

take two arguments of any sort to make a proposition. The object in successive iterations will be to assign these relational predicates substantive sortal constraints, thus constraining head modifier relations and the parse possibilities.

Verbs, nouns, some adjective and adverbs, on the other hand, have signatures with fully or partially instanciated arguments: For example, in the ATIS domain, the verbs *depart*, *get_in* and the nouns *data*, *flight* have the signatures:

signature(*depart*, ([[*departure*]], [*prop*]))
signature(*get_in*, ([[*arrival*]], [*prop*]))
signature(*data*, ([[*information*]], [*prop*]))
signature(*flight*, ([[*flight*]], [*prop*]))

These declarations have no effect on the combinatorial possibilities of these words (they tell us nothing about what can be the subject of the verb *depart* or what verbs the noun *flight* can be subject of), but when a logical form is built up from a syntactically licensed parse (like the one given above for *a departing flight*), these sortal declarations will "fill in" the sorts for the connecting predicate *actor*, generating the sort rule:

signature(*actor*, ([[*departure*], [*flight*]], [*prop*])

Thus in the signature file, lexical predicates have their own "inherent" sort rules, which then help build up the sort rules for the relational predicates. The inherent sort rules for adjectives like *cheap* and *late* will constrain only their first argument. The reason for this is that it is this first argument that modifiers (such as intensifying adverbs and specifiers), will hook on to.

signature(*cheap*, ([[*cost_soa*], *A*, *B*], [*prop*]))
signature(*late*, ([[*temporal_stage*], *A*, *B*], [*prop*]))

At the same time the argument position filled in by what the adjectives modify is left unconstrained. The signature file thus makes no commitment about what sorts of things can be *late* or *cheap*; it just needs to say there is such a thing as lateness and cheapness. This is why for a new domain the signature file can be generated largely automatically, using a new inherent sort for each new lexical item, assigning the type of predicate appropriate to its grammatical category.

All zero-arity predicates (names) need to have inherent sorts. Certain general 'tool words' which include numbers, dates, time, and commons words, will receive the same signatures in any domain :

signature(3, ([*number*]))
signature(*friday*, ([[*day*]], [*prop*]))
signature(*pm*, ([*nonagent*]))
signature(*yes*, ([*prop*]))

In addition to this, however, there is a whole list of words specific to the domain which need to be inherently sorted. This part of creating a signature file will need to be done by hand:

signature('*NASHVILLE*', ([*city*]))
signature('*AIR_CANADA*', ([*airline*]))
signature('*LA_GUARDIA*', ([*airport*]))

4.3 Extracting the Sorts

We now give a more detailed example of how sort rules are extracted from logical forms (LFs) built by the parser. For *'the morning flights flying to denver'*, we obtain roughly the following Logical Form :

qterm(*the*; [*non_symmetric_determiner*],
  A; [*flight*],
    [*and*,
     [*flight*, (*A*; [*flight*])],
     [*n_n_rel*,
       (*B*; [*day_part*]) [*and*,
                  [*morning*,
                  (*B*; [*day_part*])]]
      ; [[*day_part*]], [*prop*],
      A; [*flight*]],
     exists(*C*; [*flight*],
              [*and*,
               [*fly*, (*C*; [*flight*])],
               [*actor*, (*C*; [*flight*]),
                     (*A*; [*flight*])],
               [*has_aspect*,
                    (*C*; [*flight*]),
                    (*in_progress*; [*aspect*])],
               [*to*, (*C*; [*flight*]),
                    ('*DENVER*'; [*city*])]]))]))
; [*flight*]

The extraction process consists of a recursive exploration of the logical form and retrieval of each predicate and its arguments. For example, from the LFs above, our tool would extract the following sort definitions set[2] :

sor(*flight*, [[*flight*]], [*prop*])
sor(*morning*, [[*day_part*]], [*prop*])
sor(*n_n_rel*, [([[*day_part*]], [*prop*]), [*flight*]], [*prop*])
sor(*fly*, [[*flight*]], [*prop*])
sor(*actor*, [[*flight*], [*flight*]], [*prop*])
sor(*to*, [[*flight*], [*city*]], [*prop*])
sor(*frag_np*, [[*flight*]], [*prop*])

[2] For reason of efficiency and simplification, we exclude some very common predicates independent of the domain, such as *'and'*, *'equal'*, *'exists'*, *'has_aspect'*, and *'qterm'*.

sor(np_frag, [[prop]], [prop])

When constrained only by signatures, the parser typically finds a large number of logical forms. The sorts tool provides the option of harvesting sort rules in one of two ways, either from all generated logical forms, or only from the Preferred Logical Form (PLF). The parse preference component implemented in Gemini chooses the best intepretation from the chart, based on syntactic heuristics[2], and provides a set of PLFs.

In addition to the extraction of the sort rules, we also calculate the occurrence Θ_i of each sort rule for all the sentences of the corpus. We then normalized Θ_i by the number of logical forms that include the sort rule ($\overline{\Theta}_i$). Each value $\overline{\Theta}_i$ is stored along with its sort rule and used to calculate the probabilities related to the sort rule :

$$Prob(Sort_i) = \frac{\overline{\Theta}_i}{\sum_{i=0}^{n} \overline{\Theta}_i}$$

In fact three sets of probabilitilies are calculated for each rule R: (1) Global probability of sort rule R: the number of invocations of rule R normalized by the number of LFs containing R and divided by the total number of rule invocations in the corpus; (2) Conditional probability of rule R given a particular predicate; (3) Conditional probability of R given the predicate in R and an argument of the same sort as the first argument of R.

Also, associated to each sort definition, we keep the list of the indexes of a small set of sentences which contain the corresponding sort definition in its logical form. This set is used as a sample for the set editor tool.

4.4 The Argument Restrictions

The argument restrictions are instantiated versions of the signatures for each predicate. For example, after parsing and extraction from the logical forms, the arguments X and Y of the signature associated to the preposition *at* will help to generate a list of several sort definitions such as :

sor(at, ([[airport], [city]], [prop])
as in : *'the aiport at Dallas'*,

sor(at, ([[domain_event], [time_point]], [prop])
as in : *'departure at 9pm'*.

5 SORT EDITING

At each step of the process, after parsing, the linguist, using the interactive sort editor, can examine the new sort file which has been generated and choose which sortal definition need to be eliminated. Statistical information associated to each sort definition helps him decide which ones are revelant or not. We have also included the possiblility of adding a sort definition, although this kind of operations should be very rare. In fact the main activity of the linguist using the sort editor tool, will be to filter the sort definitions generated by the parsing of the corpus.

5.1 Description of the tool

The sort editor tool is an interactive, window-based program. It has a main window for displaying and editing the sorts and a set of buttons that help the user to either display additional information or perform actions such as :

- load or save a sort file,
- select a functor among the list of all functors and display the list of its possible arguments, result and probabilities,
- deletion and insertion of a sort definition,
- display a sample of sentences associated to a specific sort definition,
- mapping between the sort definitions and a reference sort file (for evaluation),
- changing the way the sort definitions are displayed (result or not, mapping or not, global probability, conditional to a functor, or relative to the first argument of a definition),
- use of a threshold on the probabilities to filter the sort definitions,
- retrieve the list of functors given a certain argument,
- display the sentences associated to a sort definition,
- display the list of predicates which have been excluded form the extraction,
- specification of a sortal hierarchy to be used with the sort definitions for the next iteration,
- use of a whiteboard to save specific sentences and information during a session.

The tool uses ProXT, the Quintus Prolog interface to MOTIF widget set and the X-Toolkit.

6 EVALUATION AND RESULTS

Evaluate the porting to a new domain require measuring how the new sort file contributes to perform the target task within the new domain. This kind of evaluation is difficult because it is hard to separate the contribution of the grammar and the contribution of the sorts constraints. One way to evaluate our tool would be to have a file of " correct" sortal constraints that we use as a reference to check the ones we generate with our tool. The problem is that this kind of file does not exist for new domains, since obtaining such file is precisely the purpose of our tool.

The approach we have chosen was to use the sort file built by hand for the ATIS corpus and to check this 'reference file' against the new sort file we intend to build, using our tool on a corpus of the same domaine.

6.1 Building the signature file

For the this first experimental exercise with the sort tool, we built the signature file somewhat differently than we would build it for a new application. In order to facilitate evaluating the tool, our goal this time was to come up with a signature file be compatible with the reference file built by hand.

The first step in the experiment was to automatically extract the signatures from the lexicon and reference sorts file, which contains nearly 2200 sort definitions. Signatures are largely predictable from the grammatical category of a word For example, most of the verbs (except the auxiliaries) with one argument, received a signature identical to the sort definition. On the other hand, most of the prepositions received a signature with all their arguments replaced by a variable (since they are domain-specific). In this maiden voyage of the sort acquisition system, the signatures chosen for verbs, adjectives and nouns were made compatible with the sort hierarchy used by the reference sorts file. In porting to a new domain, the lexical signatures would presumably use an automatically generated sort hierarchy, almost entirely flat, with a unique lexical sort for each lexical item.

In addition to this, some signatures, for logical predicates and predicates introduced in semantic rules, were added by hand. These represent a little bit more than 15% of the final signature file which contains a total of 1357 signatures. Half of these signatures are zero-arity predicates mostly automatically built from the lexicon.

6.2 Parsing Madcow

The next step of our experiment was to parse a corpus from the ATIS domain using the signature file we have built. For this, we have used the MADCOW corpus[4], that includes 7243 sentences of various length (from 1 to 36 words) with a large linguistic coverage from this domain. This process had been done in both modes LFs and PLFs. The idea was to compare the result in both modes, to check whether the use of parsing preferences was relevant for the extraction of the sort definitions or if we had to use all the Logical Forms from the parsing.

The first iteration of parsing MADCOW produced 5917 and 2275 sort rules[3] respectively for the LFs and PLFs modes.

6.3 Mapping corpus and reference rules

For this first evaluation, we also used a feature of our tool which can map each sort rule produced by the extraction phase against the rules of a reference sort file. The mapping consists of assigning one of the following categories to each corpus acquired sort rule :

- Exact : the corpus rule match exactly with a reference rule,
- Incompatible : the corpus rule does not match with any reference rule,
- Subsumed-by : the corpus rule is subsumed by at least one reference rule,
- Subsumes : the corpus rule subsumes at least one reference rule,
- Incomparable : the corpus rule is incomparable[4] with at least one reference rule.

The following table shows the repartition of mapping categories modes LFs and PLFs :

	LFs	*PLFs*
Exact	409	362
Incompatible	3055	691
Subsumed-by	1557	888
Subsumes	375	156
Incomparable	521	178
Total	5917	2275

[3]Since zero-arity sort predicates have a signature identical to their sort rule, only sorts rules with at least an argument were extracted during the parsing of MADCOW.

[4]Two sort rules are incomparable, when they unify each other while none of them subsumes the other one.

The first comments concerning these figures is that the percentage of incompatible rules is higher for the LFs than the PLFs mode (respectively 52% vs 30%), and the number of 'exact' sorts is more than half for LFs than PLFs. This shows that the use of Preferred Logical Forms for parsing is more efficient in extracting the 'good sorts'.

However, the figures do not give an exact idea of the completeness and precision of our tool, since there is a large number of rules subsumed by other ones (more than 30% for LFs and almost 50% for PLFs mode). In fact, some of the corpus rules are subsumed by more general rules in the reference sort file while providing the same coverage as the reference sort rules.

Therefore, the **precision** of our tool for the PLFs mode just after the extraction phase can be estimated between 16% (exacts rules) and 55% (exact rules plus subsumed rules). This number gets better and more precise very quickly after the first iteration of editing since the work of the linguist is precisely to remove most of the incompatible and incomparable rules and rules which are either too general or too specific.

The **overgeneration** of the tool just after parsing, for the PLFs mode, can be estimated to at least 30% (the percentage of incorrect rules). After the first iteration of editing, this number decreases very quickly since low probabilities help the linguist to eliminate rules that are incompatible or incomparable.

The **recall** for the PLFs mode after parsing, which is the ratio of the 'Exact' corpus rules by the number of reference rules used for the mapping in our evaluation (636 non zero-arity sorts rules), can be estimated to at least 57%.

A more precise estimation of the exact number of 'Exact' rules could be computed by using the sortal hierarchy, and generate for the two sets of rules (corpus and reference) all the rules that can be subsumed, and realize the mapping only with these rules.

7 CONCLUSION

This first evaluation of our tool in the ATIS domain shows that the acquisition of sorts from a corpus can be partially automated, reducing drastically the time the linguistic dedicates to this task (the precision converges in few editing iteration). In addition to this, the possibility of a systematic examination for all predicates with crosschecking tools such as sentence visualisation and functor browing helps the linguist to establish strict aquisition methods for the knowledge base in new domains.

In addition to this, the tool can also be used to improve an existing knowledge base. For example, the study of the incompatible rules during this first evaluation helped us to discover new rules that will increase the coverage of Gemini in the ATIS system.

8 Acknowledgements

This research was supported by the Advanced Research Projects Agency under contract with the Office of Naval Research, and by a grant of the Lavoisier Program from the French Foreign Office. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency of the U.S. Government, or those of the Scientific Mission of the French Foreign Office.

References

[1] Alshawi, H. (ed.), *The Core Language Engine*, MIT Press, 1992.

[2] Dowding J., Gawron J.M., Appelt D., Bear J., Cherny L., Moore R. and Moran D., "GEMINI : A Natural Language System For Spoken-Language Understanding", Proceedings of the 31st Meeting of the Association for Computational Linguistics, Ohio State University, Columbus, Ohio, pp. 54-61, 1993

[3] Grishman R., Hirschman L. and Ngo T.N., "Discovery Procedures for Sublanguage Selectional Patterns : Initial Experiments", *Computational Linguistics*, Vol. 12:3 pp. 205, 1986.

[4] Hirschman L., "Multi-Site Data Collection for a Spoken Language Corpus", MADCOW, in *Proceedings of the DARPA Speech and Natural Language Workshop*, pp. 7-14, Feb. 1992.

[5] Lang F.M., Hirschman L., "Improved Portability and Parsing Through Interactive Acquisition of Semantic Information", In *Second Conference on Applied Natural Language Processing'*, Feb. 1988.

[6] Mellish, C., "Implementing Systemic Classification by Unification". *Computational Linguistics*, Vol. 14, pp. 40-51, 1988.