Finite State Machines from Feature Grammars

Alan W Black
Centre for Speech Technology Research
and Dept of Artificial Intelligence
University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN
awb@eusip.ed.ac.uk

Abstract

This paper describes the conversion of a set of feature grammar rules into a deterministic finite state machine that accepts the same language (or at least a well-defined related language). First the reasoning behind why this is an interesting thing to do within the Edinburgh speech recogniser project, is discussed. Then details about the compilation algorithm are given. Finally, there is some discussion of the advantages and disadvantages of this method of implementing feature based grammar formalisms.

1 Background

Real-time continuous speech recognition is still not possible but is becoming more possible each year. One of the many problems in recognition is doing symbolic analysis in the higher levels of the system in a reasonable time.

Within CSTR, we are investigating analyses using high level GPSG-type formalisms (like that in [Gazdar85]) to describe the grammar of various restricted domains. This high level notation is then automatically compiled into a basic feature grammar formalism called FBF ([Thompson89]) thus compiling out aliases, feature passing conventions etc. This FBF grammar is then used directly in the run-time recogniser within a chart parser.

However, at run time, the many hypotheses predicted by the lower levels of the system give rise to many partial constituents in the chart. Thus a large amount of time was spent in the chart doing unification. However, when we look at the real requirements of the lower level of the system (lexical access), we note that what is required in the majority of cases is merely a simple prediction of the next possible symbol in a sentence from a given state.

Consequently we started to think about ways to provide this information as quickly as possible. Obviously representing the grammar as a Finite State Machine would make lexical access prediction significantly faster. As we currently write our grammars in a high level formalism it seems wrong to throw that information away and start again, so we hope to find some form of compilation from feature grammars to finite state grammars.

Of course, the first theoretical point to note is that feature grammars are, in essence, context-free thus allowing more complex languages to be described than FSGs. For example, there does not exist an equivalent finite state grammar for the (context-free) grammar

$$S \rightarrow a\ S\ b$$
$$S \rightarrow a\ b$$

Which describes the language $a^n b^n$ where n is greater than or equal to 1. However if we set a finite limit on n then there does exist a (possibly very large but *finite*) FSM. Thus we could accept $a^n b^n$ only where n is greater than or equal to one but less than some finite number d.

In terms of natural language, an equivalent example is the restriction that you can only have up to n levels of centre embedding within a language. This seems to be no less a restriction on a language than the restrictions you are imposing on that language when you try to write a grammar for it in the first place, irrespective of the grammar formalism.

Practically, there may be other problems in writing a compilation function from feature grammars to finite state grammars. There is of course the problem of the *size* of FSM created, as well as the *time* that is needed to generate it. Both these question were open at the start of our investigation.

Because we hoped that this compilation need only be run occasionally and that the high level formalism could be debugged using a conventional chart parser, we feel that compilation time can be up to 12 hours without any problem. As for the resulting FSM, it seems that with today's workstations up to 100,000 transitions might be acceptable. But the question still remained: how big a feature grammar can be compiled within these constraints?

2 The Initial Structures

The grammarian first writes a grammar in the high level GPSG-like notation which is then translated to FBF. This translation is relatively simple, it merely converts the user-written form into an internal Lisp form, expanding aliases, feature passing conventions etc. The FBF formalism seemed like a good input to the FSM compiler as it is well defined and quite fixed within our system.

FBF is effectively an *assembly language* for feature grammars. It is much in the spirit of PATR-II ([Shieber86]) but differs in that it uses term unification rather than graph unification as its basic operation, though that distinction if not important here.

The inputs to the FSM compilation are:

- a distinguished category
- a set of feature grammar rules.
- a set of lexical entries

The lexicon consists of a mapping of atomic symbols to categories. In actual fact within our system these atoms are not words but preterminals. It is these preterminals which label the arcs of the generated finite state machine.

It should be added that FBF is not a prerequisite for this technique. Any feature grammar notation would be suitable (though the code would have to be changed).

3 The Compilation Process

The compilation takes place in five stages:

- conversion into internal structures for fast access. This consists of the conversion of categories in the grammar and lexicon into an internal form, consisting of an atomic type and a list of feature values, thus unification can be done more efficiently. Also, two indexes are created – one for the grammar and one for the lexicon – both indexed by category type, allowing efficient access to them.
- conversion of the grammar to a non-deterministic finite state machine. This is the main part — see the the next section for details about this.
- removal of error states from the non-deterministic finite state machine. States can be created which cannot lead to final states, these are removed as well as all arcs pointing to them.
- determinising. Standard determinising of the finite state machine (as described in [Hopcroft79, p. 22])
- analysis to produce statistics, this finds the size, average and maximum branching rates.

4 The Actual Conversion

The conversion is done by building "agenda states" on an agenda and processing them until the agenda is empty. An "agenda state" consists of the following:

- A depth — the number of rewrites that are required to get the first category in the remainder
- a list of remaining categories — these are the categories (preterminal or otherwise) that have yet to be found before the end of a sentence is reached
- A set of variable bindings
- a state in the non-determinised machine

The basic loop starts with an initial "agenda state" with the following settings:

- a depth of 0
- a list containing only the distinguished category
- a set of empty bindings
- the initial state of the (non-deterministic) FSM

The processing is as follows:

Take an "agenda state" from the agenda and take its remainder. Rewrite the first category in the remainder, using the grammar, in all ways, recursively until either the depth limit is met or a lexical category is found (i.e. a category which is in the lexicon).

Rewrites are made by replacing the first category with the right hand side of a grammar rule, whose left hand side unifies with the first category. Thus a rewrite changes the first category, increments the depth, and possibly binds some variables[1]. Also, in addition to the right hand side, a special "end-subrule" marker (*em*) is added so that we can tell when to decrease the depth count. For example: S may rewrite as follows[2]

$$\begin{array}{ll} \text{S} & \Longrightarrow \\ \text{NP VP } em & \Longrightarrow \\ \text{Det Noun } em \text{ VP } em & \end{array}$$

Then for each rewrite, check the lexicon and find all entries that can match the first category. Add a transition to the state in the current "agenda state", labelled with that lexical item, to a new state, in the non-deterministic FSM.

This may be a (truly) new state or an already existing state. Each state in the non-deterministic FSM has a "state descriptor" which symbolizes which categories from this state would lead to a final state. The state descriptor is constructed by taking the remaining categories list and dereferencing the variables, removing the "end-subrules" markers, and replacing any unbound variables with a unique atom name representing a variable[3]. Thus no unification is required in searching, a simple Lisp `EQUAL` is adequate (actually a more complex indexing system is used).

When looking for a "new state", the state descriptor of the required state is constructed and a (rather large) index is checked to find if such a state already exists, if so the new transition points to the state related to that "state descriptor".

If a truly new state is required a corresponding new "agenda state" is created. The "cdr" of the remaining categories list is taken: that is the next category is found in the remainder list, any "end-subrule" markers which precede it are removed and the depth is decremented.

5 An Example

For the sake of brevity the example grammar used here is only a standard context-free grammar with atomic categories rather than a feature grammar. Thus we use `EQUAL` as our test operator, while with feature grammars we would use unification, and record any resulting bindings.

Given the following grammar:

S $\rightarrow$ NP VP
NP $\rightarrow$ Det Noun
NP $\rightarrow$ PropNoun
VP $\rightarrow$ Verb NP

And a lexicon as follows:

`the` $\rightarrow$ Det

[1]Because variables are "uniquified" at each instantiation of a rule the correct bindings are ensured throughout the conversion.

[2]Atomic symbols are used here as categories for brevity

[3]This is actually over-general, as variables which have been bound to one variable, and hence co-referenced, but not (yet) bound to a literal, will still be treated as distinct by this method.

`boy` → Noun
`Hanako` → PropNoun
`saw` → Verb

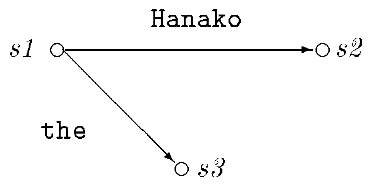
Let us go through some of the steps. The first stage is an agenda state of the form[4]:

depth: `0` remainder: `(S)` state: *s1*

There are two possible rewrites

depth: `2` remainder: `(PropNoun em VP em)`
depth: `2` remainder: `(Det Noun em VP em)`

We then add transitions from *s1* to two new states labelled with "`the`" and "`Hanako`" like so:



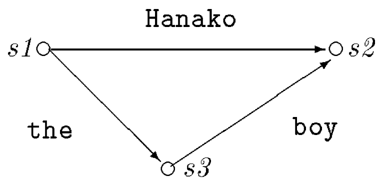
We then create two new "agenda states" and add them to the agenda

depth: `1` remainder: `(VP em)` state: *s2*
depth: `2` remainder: `(Noun em VP em)` state: *s3*

Now consider the second one. As `Noun` is already a lexical category, there is no need to rewrite it. We can add a transition from *s3* to a "new state". To find the "state descriptor" of this "new state" we first remove the first category, and then remove any "`em`" markers, decrementing the depth accordingly. The resulting remainder and depth is

depth: `1` remainder: `(VP em)`

Then we create the "state descriptor" from this new remainder, which will give simply `(VP)`, which is the same as the descriptor of *s2*. Thus this new arc labelled with "`boy`" will go from *s3* to *s2*. Like this:



Thus we only need one occurrence of the VP despite there being two "types" of NP. Of course in larger grammars, we would probably have two parts of the FSM representing VPs, one dealing with singular subject VPs, and the other with plural VPs (actually there may be more depending on the distinctions made in the grammar). This of course means building a large FSM, but that is, in part, the object of this exercise, trading space (i.e. the size of the FSM) with time (reducing the number of unifications required).

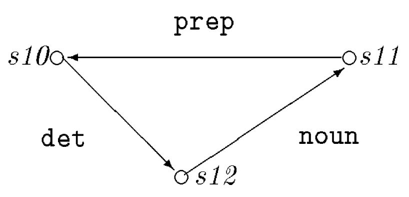
[4]no bindings are shown as we dealing with a simple atomic CFG

5.1 Getting Loops from Recursion

Consider the following three rules in isolation:

NP → NP PP
NP → Det Noun
PP → Prep NP

If we can collapse recursion into loops, we can represent these three rules by the very simple FSM



We have two problems to deal with here, left recursion, and right recursion. Left recursion is a lot harder to deal with than right recursion. With left recursion, during the rewrite stage we must check to see if we have already used the rule during this rewrite. If we detect this, we construct the new rewrite in a different way.

Instead of replacing the first category with its expansion, we find: what the non-recursive rewrites are; and the rules which introduce the rewrites. For the sake of description we will consider the case where there is only one non-recursive and one recursive rule, as in this example. Thus we have a "non-recursive rewrite" (`Det Noun em`) and a "non-recursive part of a recursive rule" (`PP em` — from the rule NP → NP PP). We then construct a new remainder (for an "agenda state")

("non-recursive rewrite"
     ("non-recursive part of a recursive rule")
     "top remainder"
)

When there are multiple occurrences of the first two parts we must form remainders for the cross-product of them. However in our example, suppose we start with the remainder `(NP VP em)`, the three parts are

non-recursive rewrite	`Det Noun em`
non-recursive part of recursive rule	`PP em`
top remainder	`VP em`

Thus the complete rewrite is

```
(Det Noun em (PP em) VP em)
```

The "looping part" in brackets, `(PP em)`, does not appear in the "state descriptor" and hence this state is treated the same as `(Det Noun em VP em)`. The important feature is this: when the categories before the bracketed part have been dealt with and we have remainder of the form `((PP em) VP em)`, we construct *two* new "agenda states", one with remainder `(PP em VP em)` and the other `(VP em)`.

This of course is too general as we are now treating the states with the "state descriptors" `(Det Noun em VP em)` and `(Det Noun em (PP em) VP em)` as the same, which may not be true. What we need to do is ensure that after the "looping part" we can get back to the same state which did not follow that part. (Assuming no variable bindings have made that join inappropriate).

Right recursion is a lot easier, having generated a state with the remainder `(PP em VP em)`, we rewrite to `(prep NP em em VP em)`. After removing the `prep` we will be left with a remainder of `(NP em em VP em)`. Because we ignore "depth" and the "end-subrule" markers in generating "state descriptors", the "state descriptor" of `(NP em em VP em)` is the same as that of `(NP em VP em)`, despite the different depths and number of "end-subrule" markers. Thus after the preposition we can return to the point in the FSM where we require an NP followed by a VP.

It is true that this NP is "different" from the other. One is an NP within a PP the other is the subject of a sentence, but because we are merely doing *recognition* this is all we need.

Notice that this matching of states by a state descriptor is not *guaranteed* to merge similar states, since there may be cases where one remainder does not start with a lexical category and another does. These may represent the same state if the first category can be written to the a remainder the same as the other (and only that remainder). This means that we will not guarantee the most minimal FSM during compilation, but will collapse many states.

6 Complexity Results

It is not surprising that this is possible. The really interesting part is whether useful grammars can be converted to reasonably sized finite state machines in reasonable time.

The code is written in Common Lisp and runs on a number of different machines. It had to be re-written a number of times to get the performance we wished. It has been true that the spectre of unacceptable computational complexity has been just round the corner a number of times but so far we have kept it at bay.

Describing the size of a grammar is difficult, but to give some idea of the feasibility of this method of running feature grammars, one of our current grammars, which consists of 31 GPSG-like rules, describes declarative sentences with the following features:

transitive and intransitive verbs
copula sentences
multiple adjectives, and intensifiers in NPs
quantifiers
noun compounding
NP conjunction

The NP conjunction was quite a drastic addition, which increased the size of the resulting FSM by an order of magnitude.

The grammar described above can be converted to a non-deterministic FSM of about 9,000 states[5] in around one hour on a Sun 4/260 with 32Megabytes of memory. We feel this is well within our 12 hour/ 100,000 state limit. But although this grammar is bigger than many "toy grammars", it is still rather small and not really large enough to cover a significant proportion of the domain we wish to cover.

[5]without conjunction the FSM is less than 1,000 states

It should be added that we have had problems in determinising some of the generated FSMs. Though the conversion stage has taken around an hour, determinising has failed to finish in 75 hours, producing a much larger FSM than its non-determinised equivalent. This does suggest that perhaps we should only produce non-deterministic FSMs as output.

7 Comment

So the basic question is, "is it worth it?"

The major loss in moving from a chart parser using a feature grammar to a finite state machine is the loss of a parse tree. One of the reasons for adding a sentence grammar to a speech recogniser is to enable (eventually) some form of semantic analysis. There is an argument that because vast numbers of hypotheses have to be dealt with by a speech recogniser, perhaps running with a FSM as a grammar would be effective during recognition, and that post-processing of the few sentences found could be done with a chart parser.

Then again perhaps speed is not the real thing to worry about, a fast chart parser and unification algorithm might work almost as well (especially if machines are doubling in speed every year).

It is true that the technique is practically limited, no matter how fast machines get there will always be grammars which cannot be converted in reasonable time and/or produce finite state machines with too many states.

And as noted before, the algorithm does produce a FSM which accepts the subset of the language described by the feature grammar where the "depth" less than the given limit, *plus* some extra sentences not originally accepted by the feature grammar. These extras are because of two faults in the conversion algorithm, namely in joining the end of left recursive rules and not constraining where variables have been co-indexed by another variable (and not an atomic value).

This over-generation seems to encourage the idea of using a real chart parser to post-process and correct the sentences accepted by the FSM (though the types of grammars which cause these problems are not common in our domain, so far).

Within our working framework (speech *recognition*) this method does produce useful results. As we can still allow our grammarians to write a high level description, but still have a fast implementation of their grammar. So in spite of the short comings we will probably use this technique for the foreseeable future.

8 Acknowledgements

This work was supported by the UK Information Engineering Directorate/ Science and Engineering Research Council as part of the IED/SERC Large Scale Integrated Speech Technology Demonstrator Project (SERC grants D/29604, D/29611, D/29628, F/10309, F/10316), in which Marconi Speech and Information Systems are the industrial partner.

References

[Gazdar85] G. Gazdar, E Klein, F. Pullum and I. Sag *Generalized Phrase Structure Grammar*

Blackwell, Oxford, 1985

[Hopcroft79] J. Hopcroft and J. Ullman *An Introduction to Automata Theory, Languages and Computation* Addison Wesley, Reading 1979.

[Shieber86] S. Shieber *An Introduction to Unification-based Approaches to Grammar* CSLI Lecture notes Number 4, 1986

[Thompson89] H. Thompson *FBF – A Micro-formalism for grammar: Syntax, Semantics and Metatheory* Dept of AI, University of Edinburgh, forthcoming