Compiling Typed Attribute-Value Logic Grammars

Bob Carpenter
Computational Linguistics Program, Philosophy Department
Carnegie Mellon University, Pittsburgh, PA 15213
carp@lcl.cmu.edu

Abstract

The unification-based approach to processing attribute-value logic grammars, similar to Prolog *interpretation*, has become the standard. We propose an alternative, embodied in the Attribute-Logic Engine (ALE) [Carpenter 1993], based on the Warren Abstract Machine (WAM) approach to *compiling* Prolog [Aït-Kaci 1991]. Phrase structure grammars with procedural attachments, similar to Definite Clause Grammars (DCG) [Pereira and Warren 1980], are specified using a typed version of Rounds-Kasper logic [Carpenter 1992]. We argue for the benefits of a strong and total version of typing in terms of both clarity and efficiency. Finally, we discuss the compilation of grammars into a few efficient low-level instructions for the basic feature structure operations.

Compiling Type Definitions

The first component of an ALE grammar is a type specification, which lays out the basic types of feature structures that will be employed in a grammar, along with the inheritance relations between these types and declarations of appropriate features and constraints on their values. Such a specification includes declarations such as the following for lists of atoms:

```
bot sub [atom,list].
  atom sub [a,b].
    a sub [].
    b sub [].
  list sub [ne_list,e_list].
    e_list sub [].
    ne_list sub []
            intro [hd:atom,tl:list].
```

The idea here is that `bot` is the most general type, with two subtypes `atom` and `list`. The type `atom` has two subtypes, `a` and `b`, which are maximally specific types. The list type also has two subtypes, `ne_list` and `e_list` for non-empty and empty lists, respectively. Note that the `ne_list type` introduces two features, `hd` and `tl`, whose values are required to be atoms and lists. The idea here is that the only type which has any appropriate features is the `ne_list` type, and it is appropriate for exactly two features, `hd` and `tl`.

Inheritance of appropriateness specifications is performed on the basis of the type hierarchy. For instance, consider the following declaration from HPSG:

```
sign sub [word,phrase]
     intro [phon:phon_list,
            synsem:synsem_obj,
            qstore:quant_list].
  word sub []
       intro [phon:singleton_phon_list].
  phrase sub []
         intro [dtrs:dtr_struct].
```

Here the type `sign` introduces three features and provides value restrictions. The subtype for words inherits these features and the associated value restrictions, imposing the additional condition that the phonology value be a singleton list. In addition, the subtype for phrases introduces an additional features for daughters, which is only appropriate for phrases. Thus, unlike the case for order-sorted terms (see, for instance, [Meseguer et al. 1987]), not every subtype of a type need have the same slots for values. This is significant in terms of implementations, as memory cells are only allocated on a structure for appropriate features.

The initial stage of compilation in ALE involves just the type hierarchy. First, the transitive closure of subsumption is calculated using Warshall's algorithm (see [O'Keefe 1990]). Second, least upper bounds are computed for each pair of consistent types. A condition on type hierarchies is that they form a bounded-complete partial order (BCPO), or

in other words, that every pair of bounded (consistent) types, those pairs of types with a common subtype, has a least upper bound. This ensures that the unification of two types always takes a unique value. This reduces non-determinism at run-time, but might require additional types to be declared by the user (see [Carpenter 1992]). Such hierarchies can be compiled automatically from either systemic networks or ISA/ISNOTA hierarchies, as shown in [Carpenter and Pollard 1991], and such a compiler has been developed and will be included in the next release of ALE [Carpenter and Penn forthcoming]. The final stage involves calculating which features are appropriate for each type and their appropriate values. This is done by collecting all of the declared features on subtypes and unifying their value restrictions. The second condition on type hierarchies, in addition to their forming a BCPO, is that they introduce each feature at a unique most general type. This, along with the BCPO condition, ensures a unique solution to the type inference problem. If we only knew that a feature `f` was defined, and nothing else about an object, then if there were two maximally general types for which `f` was appropriate, a decision could not be made as to which type it was and non-determinism would be introduced. As with the BCPO condition, this condition can be automatically eliminated by introducing a new type appropriate for the feature which is more general than the two existing ones (see [Carpenter 1992]). We also forbid appropriateness cycles such as:

```
person sub [male,female]
       intro [father:male,
              mother:female].
   male sub [].
   female sub [].
```

We rule out this situation because type inference, as we define it below, can not find most general well-typings in such cases. To be a well-formed object of type `male`, a requirement is that the `father` feature is defined and filled by another male, leading to a non-halting procedure. Again, if we wish to represent people with parents, the problem can be solved by adding types which are not required to have parents.

During the compilation of the type system, many different kinds of errors are detected, such as: two types which mutually subsume one another, violations of the BCPO condition where two types have multiple unifiers, cases where inconsistent constraints are inherited by a feature, where there are appropriateness cycles, where there is no most general type appropriate for a feature, and so on. Other errors such as undeclared types and multiple declarations are also recognized. In addition, a number of warnings are raised in cases which might not be desirable, such as a type with only one subtype or where dynamic type-inference during unification will be necessary. This latter condition arises when types `s` and `t` are both appropriate for `f`, with value restriction `s'` and `t'`, but the unification of `s` and `t`, `s+t`, has a more specific restriction than `s'+t'`. In this case, when an `s` and `t` object are unified, additional constraints on their value for a feature must be checked.

Compiling Basic Operations

As with other grammar formalisms based on attribute-value logics, the primary data structure used in ALE is the feature structure. The structures used in ALE are similar to those in other systems, with the primary difference being that they are required to be totally well-typed (see [Carpenter 1992]). In other words, every feature structure must be assigned a type and every feature appropriate for that type must appear with an appropriate value. This can be contrasted with sorted, but untyped systems, which allow sorts to label feature structures and participate in unification, but don't enforce any typing conditions. It can also be contrasted with systems which only perform type inference on values, but do not require every appropriate feature to be present. There are a number of benefits to typing a programming language. Not the least of these benefits is the ability to detect errors at compile-time. For instance, rules which can not be satisfied and lexical entries which are not well-typed are flagged as such. Practice has shown that this cuts down on grammar development time significantly, because one of the most prevalent grammar-writing errors is being inconsistent about which features appear at which level in a structure and how they are bundled together, especially when grammar formalisms approach the 200 node lexical entry level as found in significant fragments of HPSG (see [Penn 1993b]).

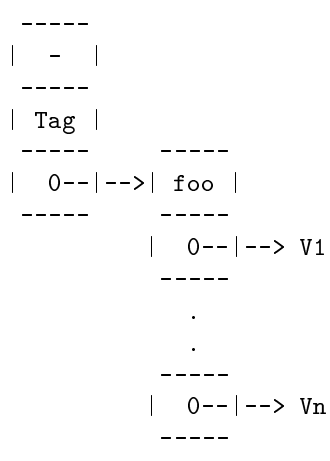
Another significant benefit of employing typed structures is that the features appropriate for a type can be determined at compile time. This has two advantages. First, it allows memory allocation and deallocation to be handled efficiently, as the type of each structure is known. Second, it allows unification to be greatly speeded up as there is no need

to merge features represented as lists; the positions of relevant features are known at compile time. We consider these two benefits in turn.

ALE is currently implemented in Prolog, though plans are underway to implement it in C, using WAM techniques directly. As things stand, the WAM implementation of Prolog is exploited heavily to develop WAM-like behavior for ALE. Using Prolog for feature structure unification systems has its advantages and drawbacks. The drawback is that there are no pointers in Prolog, and thus path compression during dereferencing can not be carried out efficiently (though it is carried out on inactive edges during parsing). The advantage is that Prolog is very good at structure copying, last call optimization, incremental clause evaluation and search. We will consider all of these topics. But first, we note that the data structure used for feature structures in ALE is:

```
Tag-foo(V1,...,Vn)
```

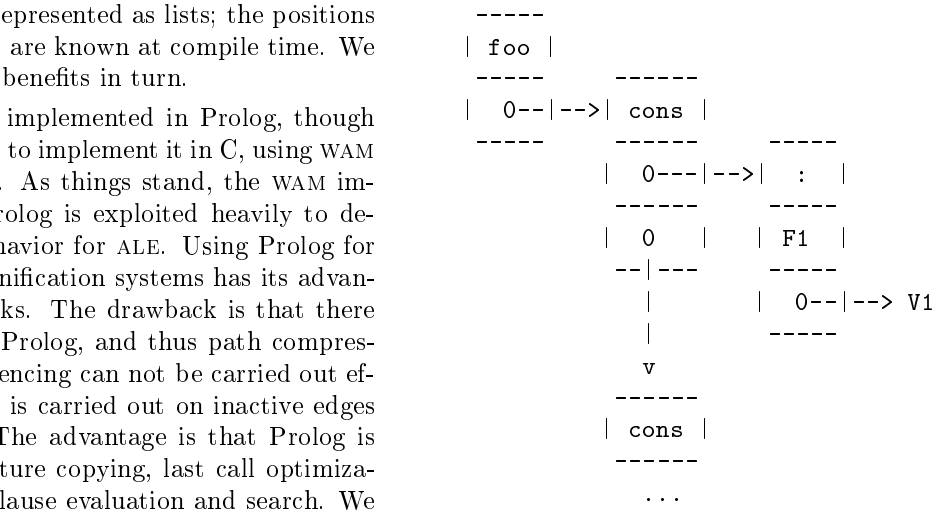
where `Tag` is a reference pointer, signalling the intensional identity of the structure, much as a position in memory in an imperative language would do, and where `foo` is the name of the type of the structure, which must be a Prolog atom, of course, and where `V1` through `Vn` are the values for the features `F1` through `Fn` that are appropriate for type `foo`. Given Prolog's compilation to the WAM, this amounts to having the following kind of record structure for feature structures:



Contrast this with a representation such as:

```
Tag-foo([F1:V1,...,Fn:Vn])
```

where the features are coded explicitly in terms of a list. Here the structure required is as follows (ignoring the tag):



In general, our representation requires $4 + n$ cells for a structure with n features, while the usual one requires $4 + 6n$ cells for the same structure. This constitutes a huge discrepancy when we consider the amount of overhead this induces throughout the grammar in areas such as lexical retrieval and copying edges into the chart. Note that this difference between using record-like structures as opposed to lists of feature-value pairs is Prolog-independent.

We have not said much about the tag. It is based on the same principle as O'Keefe's method of encoding arrays in Prolog using variables, which provides constant time access and update (see the Quintus library). The basic idea is that each slot in an array is associated with a value and a pointer, which is either a variable or a structure consisting of another value and a pointer. Updates are performed by instantiating the variable to a new pair consisting of a variable and value. Thus values are found by tracking the pointer until it's a variable. To maintain constant time, the entire array must be regularly updated. In our case, the tag plays the role of the pointer, and dereferencing is performed by following the tag value until it is a variable. The number of dereferencing steps needed at any stage is bounded by the depth of the inheritance hierarchy (of course, Prolog does its own internal dereferencing, so we can not statically bound the total number of dereferencing steps needed during unification). Path compression is then the equivalent of O'Keefe's array updating, and is performed when a completed edge is found during parsing. We will see examples of the use of tags shortly.

The second benefit we mentioned for typing structures is that we are able to carry out unification without merging feature-value lists. The standard method in unifying feature structures is to take two

lists of features, find the common elements, unify them, and take both symmetric differences and copy the results of this into the final result. Such tasks are extremely costly, especially as the number of features grows. Instead, our compiler will produce the following kind of code to perform unification (which has been simplified here, but will be expanded upon later):

```
unify(T-ne_list(H,R),T-ne_list(H2,R2)):-
  unify(H,H2),  unify(R,R2).
```

The positional encoding of feature values means that at compile time, we know which features any two types have. Also note that the tags are identified to make sure any update of one structure is felt by the other. Compare the number of operations required in the above procedure to one that had to look through two lists of feature value pairs, and act conditionally depending on the results of comparisons, and detect termination conditions. While our unification in the above case requires only two logical operations in Prolog, the list merging method would require at least two comparisons, a termination test and the same two logical operations as our method, more than doubling the cost in the best possible instance. Of course, matters are much worse if the features are out of order or don't line up exactly in the two structures.

Having motivated our approach, we now consider two operations on feature structures which are calculated at compile time. The first of our operations on feature structures involves adding the information that it is of a given type. For instance, we might have a list and want to add the information that it is non-empty. In this case, the system produces the following code:

```
add_to(ne_list,Tag-TVs):-
  add_to_ne_list(TVs,Tag).

add_to_ne_list(list,
               _-ne_list(T1-bot,
                         T2-bot)):-
  add_to_atom(bot,T1),
  add_to_list(bot,T2).
```

Note that to add the information that a structure is a `ne_list` at the top level, we make the first argument the type-value term. The reason for doing this is that the WAM performs first-argument indexing. This means that given the current structure, of type `list`, hashing is done to find the code to add the fact that it is a non-empty list to it. In fact, the clauses for `add_to/2` are never used at run-time, as can be seen from the second of the above clauses, which call `add_to_atom(TVs,Tag1)` rather than `add_to(atom,Tag1-TVs)`. At this stage, we should point out that it would be more efficient to use the following code:

```
add_to_ne_list(list,
               _-ne_list(_-atom,_-list)).
```

For completely fresh features such as the head and tail above, there is really no reason to create a structure `Tag-bot` and then immediately add a type to it. The next release of ALE [Carpenter and Penn forthcoming] will have such an optimization, as it is statically computable.

On the other hand, consider the effect of adding the type `word` to the type `sign` given above:

```
add_to_word(sign(T1-Phon,SynSem,QSt),
            _-word(T-Phon,SynSem,QSt)):-
  add_to_singleton_phon_list(Phon,T).
```

Here we see that (pointers to) the feature values for `sign` are copied over into the new word structure created and the additional constraint that the `Phon value` be a singleton list must also be resolved. Note that this extra bit of (pointer) copying is something that is usually also done in encodings using feature-value pairs.

As we hinted at above, the procedure to perform unification on two structures is also compiled before run-time. In particular, consider the code to unify two `ne_lists`, in its full form:

```
unify_deref(FS1,FS2):-
  deref(FS1,Tag1,TVs1),
  deref(FS2,Tag2,TVs2),
  ( Tag1 == Tag2, !
  ; unify(TVs1,TVs2,Tag1,Tag2)
  ).

unify(ne_list(H1,T1),TVs2,Tag1,Tag2):-
  unify_ne_list(TVs2,H1,T1,Tag1,Tag2).

unify_ne_list(ne_list(H2,T2),H1,T1,T,T):-
  unify_deref(H1,H2),
  unify_deref(T1,T2).
```

The strange argument order and extra level of indirection comes about to exploit the first-argument indexing of the WAM. In effect, what happens when unifying two structures is that they are first dereferenced, then two hashings are performed, one on each

of their types, and finally their shared feature values are unified. This illustrates one of the simplest cases of unification. Note that absolutely no type inference is required at run time because the compiler knows that when two structures of the same type are unified, then their features already meet the type constraints, and hence so will the result of unifying them. Other cases might involve `add_to_`*sort*/2 goals being called and tags being instantiated, when unifying the two structures leads to a new structure with a type higher than each of the inputs. For instance, suppose we have:

```
b sub [c] intro [f:x,h:u].
d sub [c] intro [g:y,h:v].
  c sub [] intro [f:x2,h:u+v,j:z].
```

where `u+v` is the type unification of `u` and `v`. Then we would have:

```
unify(b(V1,V2),TVs2,Tag1,Tag2):-
  unify_b(TVs2,V1,V2,Tag1,Tag2).

unify_b(d(V3,V4),V1,V2,
        _-c(V1,V2,V3,T-bot),
        _-c(V1,V2,V3,T-bot)):-
  unify_deref(V2,V4),
  deref(V1,Tag1,SVs1),
  add_to_x2(SVs1,Tag1),
  add_to_z(bot,T).
```

When unifying structures of type `b` and `d`, we must instantiate both of their reference pointers to a new structure of type `c`, with a new feature `j`, and in addition, perform the extra type inference on the value of `f`. It is worth noting that all and only the necessary type inference is determined at compile-time. For instance, the fact that the `h` value of `c` is required to satisfy the unification of the constraints on `h` in `b` and `d` is enough to let the compiler determine that no additional type inference will be required.

Compiling Descriptions

In this section, we consider compiling descriptions taken from ALE's attribute-value logic:

```
<desc> ::= <type>
         | <var>
         | <feat>:<desc>
         | <desc> and <desc>
         | <desc> or <desc>
```

As was shown by Smolka [1988], the lack of variables can lead to a quadratic increase in the size of descriptions using only path equations; with variables, path equations are no longer necessary. A complete proof theory with respect to both an algebraic semantics and a feature-structure based interpretation can be found in [Carpenter 1992].

Descriptions are compiled into the operations of `add_to_`*sort*, `unify`, `deref`, and a combination of conjunction and disjunction in Prolog. In addition, to handle constraints of the form `<feat>:<desc>`, which tell us to add the description to the value of the feature, we need a procedure for extracting a feature's value from a structure. This is done with clauses such as:

```
featval(hd,FS,Val):-
  deref(FS,Tag,TVs),
  featval_hd(TVs,Tag,Val).

  featval_hd(ne_list(H,_),Tag,H).
```

Again, we present the first clause for convenience; only the second is used at run-time, combined with the necessary dereferencing. Note that if we look for the `hd` value of a structure of type `list`, we coerce `list` to `ne_list`:

```
featval_hd(list,_-ne_list(T-atom,_-list),
                T-atom).
```

Here we create a new structure of type `ne_list`, with a fresh head and tail, and return the fresh head as the result. In general, this might require additional type inference, as could be seen by considering what would happen if we took the value of the feature `j` in an object of type `d` in the above type system. In this case, the type `d` object would be coerced to one of type `c`, which in turn requires boosting the type of its `h` value and adding new `f` and `j` values:

```
featval_j(d(V1,T2-TVs2),
          _-c(T3-bot,V1,T2-TVs2,T-bot),
          T-bot):-
  add_to_u(TVs2,T2),
  add_to_x2(bot,T3),
  add_to_z(bot,T).
```

Again notice that the compiler determines exactly which type inferences to perform as part of finding a feature's value. Again, in the next release of ALE, the `add_to_`*sort*`(bot,T)` goals will be replaced with instantiated feature structures of type *sort*.

We are now in a position to see how descriptions get compiled into Prolog clauses. To add a description of the sort found on the left to a dereferenced structure `Tag-TVs`, the Prolog code on the right is generated:

```
sort          add_to_sort(TVs,Tag)

V             deref(V,Tag2,TVs2),
              unify(TVs1,TVs2,Tag1,Tag2)

f:D           featval_f(TVs,Tag,Val),
              deref(Val,Tag2,TVs2),
              [ add D to Tag2-TVs2 ]

D1 and D2     [ add D1 to Tag-TVs ],
              deref(Tag-TVs,Tag2,TVs2),
              [ add D2 to Tag2-TVs2 ]

D1 or D2      ( [ add D1 to Tag-TVs]
              ; [ add D2 to Tag-TVs]
              )
```

Sorts are straightforward, and simply invoke the appropriate `add_to` goal. Variables are such that they get instantiated to the feature structures which they describe. Thus adding a variable to a structure involves dereferencing the variable, which is instantiated to the current value it has, and unifying it with the structure to which it is being added. All variables are initialized to `Tag-bot` at compile-time for compatibility with the basic operations over feature structures. The last three cases are recursive. Adding a description to a feature's value requires finding the feature's value, dereferencing it, and adding the embedded description. Conjunction and disjunction in descriptions are translated into the corresponding Prolog control structures. In particular, this means that we treat disjunction in descriptions as introducing non-determinism in adding a description. In this way, Prolog backtracking, and its attendant efficient implementation of search and variables, will take care of the disjunction without any need for explicit copying in the program. Of course, it's still there — it's just that Prolog's doing it. In a non-Prolog implementation of this method, a programmer would have to be very clever to implement this kind of control structure, using some kind of lazy copying along the lines of Tomabechi [1992] or along the lines of the WAM itself. Conjunction, on the other hand, is treated as goal sequencing in Prolog.

Compiling Grammars and Programs

This compilation of descriptions into Prolog code rather than into feature structures is where ALE departs most radically from other attribute-value-based parsers of which we are familiar. The traditional method, say for chart parsing, involves taking an inactive edge which has just been created and trying to unify it with the feature structures corresponding to the heads of rules in the grammar. Instead, our system will execute the Prolog code compiled from the description of the head of a grammar rule. There are two principal benefits to our approach. These stem from the fact that we reduce the copying and search methods to those of the WAM itself by compiling the Prolog clauses generated. The first benefit is that early failures in matching a description to a goal do not result in any overcopying — in fact there is really no copying done at all — it's all handled in the heap mechanism of the WAM. The second benefit is that if we have deeply embedded disjunctions in our descriptions, we do not need to expand to a disjunctive normal form or invoke one of the many approaches to disjunctive unification. In particular, if we have a description with an embedded disjunction and the first disjunct fails, then we only backtrack to the second disjunct, not all the way back to the beginning of the structure. Again, this operation is very efficient in the WAM. It should be noted that nothing here depends on using a chart parser as the control strategy — similar benefits would accrue to any other parsing strategy. In fact, the same benefits could also be gained by using this kind of strategy in generation, say along the lines of van Noord et al. [1992].

The chart parsing strategy used in ALE is not particularly significant qua parser, as it was primarily motivated by Prolog considerations. What is significant is the way in which descriptions are compiled and made available to the parser, a strategy which can be maintained using many different parsers. For instance, we are also working on a left-corner parser which will not require any copying or manipulation of the database. The most significant thing to note about ALE's parser is that it employs a dynamic chart, where inactive edges are asserted into the database,[1] and parses according to a bottom-up strategy, from right to left in the chart, and from left to right through individual rules. Active edges

[1] Current versions of the WAM in SICStus and Quintus index asserted clauses, allowing the edges beginning at a particular position to be easily retrieved by hashing — a method with explicit copying would most likely be faster than the one with `assert`, and we plan to explore this possibility.

are truly active, being represented only by the current position in a Prolog clause compiled from a rule description. Rules are of the form:

```
D0 ===> D1, ..., DN.
```

where `D0` is the description of the mother category and the `Di` are descriptions of the daughter categories. The grammar rules are then compiled into the `rule/3` predicate. The goal `rule(L,R,C)` is called whenever an inactive edge `C` is added from position `L` to `R` in the chart, either by the lexicon or by `rule/3` itself. The code produced for the above rule is:

```
rule(C1,Left,Mid1):-
  [ add D1 to C1 ],
  edge(Mid1,Mid2,C2),
  [ add D2 to C2 ],
  ...
  edge(MidN-1,Right,CN),
  [ add DN to CN ],
  [ add D0 to Tag-bot ],
  fully_deref(Tag-bot,C0),
  assert(edge(Left,Right,C0)),
  rule(Left,Right,C0).
```

When `rule(C1,Left,Mid1)` is called, the first thing that happens is the description `D1` being added to the feature structure `C1`. Assuming this fails, no other work is done, and no copying is performed. Instead, the code generated by the description `D1` is simply executed, and failure causes Prolog backtracking either to earlier disjunctions in the description `D1`, or to other clauses for `rule/3` generated by other rules. Assuming `D1` is successfully added to `C1`, `rule/3` looks for an inactive edge directly to the right of `C1` in the chart. The fact that parsing is done right to left ensures that the chart has been completed to the right of any inactive edge which is being considered. If an inactive edge of category `C2` to the right is found, `rule/3` attempts to add the description `D2` to a copy of `C2`. The current bottleneck in this process is the inordinate amount of copying required, especially when many empty categories are present. A better solution would be to add the descriptions in a more lazy fashion without eagerly copying the whole structure, but Prolog does not provide that kind of fine control of its database. This process continues until the right hand side of the rule is completely matched. At this point, the mother category is constructed by adding the compiled description `D0` to a fresh category, fully dereferencing (path compressing), asserting it into the database of inactive edges, and recursively calling `rule/3`. As there are no base cases to `rule`, it will eventually fail and backtrack through all of the disjunctive choice points and alternative rules.

The input string is consumed from right to left, at each step adding inactive edges until no more edges can be added. This gives the parser as a whole a mix of breadth-first and depth-first search, to best exploit the inherent behavior of the WAM. The top level control strategy is quite straightforward:

```
parse(Words,C):-
  reverse(Words,WordsRev),
  length(Words,N),
  build_chart(Words,N),
  edge(0,N,C).

build_chart(_,N):-
  empty(C),
  assert(edge(N,N,C)),
  rule(N,N,C).
build_chart([],_).
build_chart([W|Ws],N):-
  M is N-1,
  ( lex(W,C),
    assert(edge(M,N,C)),
    rule(M,N,C)
  ; build_chart(Ws,M)
  ).
```

The words are reversed and counted, and the chart is built from the right to left, taking lexical entries for each word and firing `rule/3`. Before considering lexical entries, empty categories are asserted into the chart and processed using `rule/3`. All lexical and empty category alternatives will be considered during backtracking before proceeding leftward to the next word. We should also mention that lexical entries and empty categories are fully expanded as path-compressed feature structures at compile time.

In addition to allowing categories in a rule, ALE also allows definite clause goals to be invoked, in a way similar to DCG rules such as:

```
f(Z) ---> h(Y), g(X), {foo(X,Y,Z)}, j(Z).
```

In this rule, as soon as the `h(Y)` and `g(X)` categories are found, the goal `foo(X,Y,Z)` is invoked and solved before going on to consider `j(Z)`. The change to `rule/3` is minimal; the code for solving `foo(X,Y,Z)` is simply inserted in between the code generated by the categories `g(X)` and `j(Z)`.

Definite clause programs can be defined in ALE, where instead of Prolog terms, feature structure descriptions are used. For instance, we can define standard predicates such as:

```
append(e_list,X,X) if true.

append(hd:X and tl:Xs,
       Ys,
       hd:X and tl:Zs) if
  append(Xs,Ys,Zs).
```

The logical variables are used as in Prolog, with the result being an instance of constraint logic programming over the typed feature structure logic. This bears a close similarity to the LOGIN language of Aït-Kaci and Nasr [1986], who point out a number of benefits of using an order-sorted notion of feature structure for logic programming. A general CLP scheme suiting this application was defined by Höhfeld and Smolka [1988] and this particular application is detailed in [Carpenter 1992]. The previous two clauses will translate into the following pieces of code, following O'Keefe's [1990] meta-interpreters (and omitting all of the dereferencing):

```
solve([]).
solve([G|Gs]):-
  solve(G,Gs).

solve(append(FS1,FS2,FS3),Goals):-
  add_to_e_list(TVs1,Tag1),
  unify(FS2,FS3),
  solve(Goals).
solve(append(FS1,FS2,FS3),Goals):-
  featval_hd(TVs1,Tag1,FS1H),
  featval_hd(TVs3,Tag3,FS3H),
  unify(FS1H,FS3H),
  featval_tl(TVs1,Tag1,FS1T),
  featval_tl(TVs3,Tag3,FS3T),
  solve(append(FS1T,FS2,FS3T),Goals).
```

The coding used, with goals being threaded, is to ensure that last call optimization takes place. While ALE does not perform indexing, it does support full cuts, disjunctions, negations by failure and last call optimization.

Such procedural attachments can be interspersed into rules just as in DCGs. This mechanism has been used in ALE grammars for purposes such as quantifier scoping using Cooper Storage, for treating the maximal onset principal in syllabification in attribute-value phonology [Mastroianni 1993], and for implementing principles such as the non-local feature principle (for slashes) and the binding theory of HPSG [Penn 1993b]. Procedures can even be used to postpone some of the unifications in a rule until after all of the categories have been found, thus encoding a form of restriction similar to that used by Shieber [1985]. Such procedures will allow general hooks to Prolog in the next release of ALE, and as the definite clause component of a grammar can be arbitrary, can also be used for interleaving online semantic processing with syntactic processing as in Pereira and Pollack [1990].

Before concluding, we should also point out that ALE has a number of other useful features. One of the most interesting of these is the use of lexical rules, which are loosely based on those of PATR-II, in that they map one lexical entry to another at compile-time. In ALE, such rules may involve procedural attachments just as other rules, and contain a rudimentary morphological component based on string unification. ALE also fully supports parametric macros which are compiled out statically into the descriptions they abbreviate.

The next release of ALE, scheduled for Summer 1993, will also include more general constraints on types, following Aït-Kaci 1986 (see also [Carpenter 1992]), inequations and extensionality (see [Carpenter 1992] for theoretical details, and [Penn and Carpenter forthcoming] and [Penn 1993a] for implementation details and motivation).

Conclusion

We have shown how grammars based on attribute-value logic descriptions can be efficiently compiled into low-level Prolog instructions which exploit the inherent efficiency of the WAM. Unfortunately, there are a few inefficiencies stemming from this encoding due to Prolog's logical variables and its lack of control over copying structures from the database. The ideal solution will be to build a WAM-like abstract machine language directly for typed feature structures and their associated descriptions. The WAM has proved to be the most efficient architecture yet developed for implementing "unification-based" programs, even though, as we saw, it often relies on structure copying and creation rather than unification (the only cases of unification in ALE arise from shared variables in a structure — everything else is structure copying).

Current benchmarks, using the standard naive reverse, with just the definite clause component of ALE, place it at roughly 1000 logical inferences per second (LI/s) on a DEC 5100 running SICStus 2.1, which is roughly 1.5% of the speed of the SICStus compiler itself. HPSG grammars where lexical entries run between 100 and 200 words, all of the local principles have been implemented according to Pol-

lard and Sag [in press], process 15 word sentences, creating 40-50 inactive edges, at times under 2 seconds.

ALE Version β, as described in this paper, is available from the author without charge for research purposes. It runs under SICStus and Quintus Prologs. It is distributed with roughly 100 pages of documentation and sample grammars. Version 1.0 is scheduled for release in August 1993.

References

Aït-Kaci, H. (1986a). An algebraic semantics approach to the effective resolution of type equations. *Theoretical Computer Science*, 45.

Aït-Kaci, Hassan and Roger Nasr (1986) LOGIN: A logic programming language with built-in inheritance. Journal of Logic Programming 3.

Aït-Kaci, Hassan (1991) *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press.

Carpenter, Bob (1992) *The Logic of Typed Feature Structures.* Cambridge University Press.

Carpenter, Bob (1993) ALE User's Guide — β. Laboratory for Computational Linguistics Technical Report, Carnegie Mellon University, Pittsburgh.

Carpenter, Bob and Gerald Penn (forthcoming) ALE User's Guide Version 1.0. Laboratory for Computational Linguistics Technical Report, Carnegie Mellon University, Pittsburgh.

Carpenter, Bob and Carl Pollard (1991) Inclusion, disjointness and choice: the logic of linguistic classification. *Proceedings of the ACL.*

Höhfeld, M. and Gert Smolka (1988) Definite relations over constraint languages. LILOG Report 53, IBM, Stuttgart.

Kasper, Bob and Bill Rounds (1990) The logic of unification in grammar. Linguistics and Philosophy 13.

Mastroianni, Michael (1993) Attribute-logic Phonology. MS Thesis, Computational Linguistics Program, Carnegie Mellon University, Pittsburgh.

Meseguer, J. and J. Goguen and G. Smolka (1987) Order-sorted unification. CSLI Report 87-86, Stanford.

O'Keefe, Richard (1990) *The Craft of Prolog.* MIT Press.

Penn, Gerald (1993a) A utility for typed feature structure-based grammatical theories. MS Thesis. Computational Linguistics Program, Carnegie Mellon University, Pittsburgh.

Penn, Gerald (1993b) A comprehensive HPSG grammar in ALE. Laboratory for Computatonal Linguistics Technical Report. Carnegie Mellon University, Pittsburgh.

Pereira, Fernando and David H. D. Warren (1980) Definite clause grammars for language analysis. *Artificial Intelligence* 13.

Pereira, Fernando and Martha Pollock (1991) Incremental interpretation. *Artificial Intelligence* 50.

Pollard, Carl and Ivan Sag (in press) *Head-Driven Phrase Structure Grammar.* CSLI/University of Chicago Press.

Shieber, Stuart (1985) Using restriction to extend parsing algorithms for complex-feature-based formalisms. Proceedings of the ACL.

Smolka, Gert (1988) A feature logic with subsorts. LILOG Report 55. IBM, Stuttgart.

Tomabechi, Hideto (1992) *Quasi-Destructive Unification.* PhD Thesis, Computational Linguistics Program, Carnegie Mellon University, Pittsburgh.