

The ALEP Formalism in a Nutshell

Jörg Schütz
IAI, Saarbücken
Internet: joerg@iai.uni-sb.de

Contents

1	The Rational Background of ALEP	2
2	Type System Specifications	3
2.1	Primitive Types	3
2.2	Structured Types	6
2.3	Type Hierarchy	7
3	Defaults	8
4	Macros	9
5	Structures and Rules	10
5.1	Phrase Structure Rules	10
5.2	Lexical Entries	11
5.3	Transfer Rules	12
6	Procedural Issues	13
6.1	Parsing and Generation	13
6.2	Rule Head Declarations	14
6.3	Specifier Declarations	14
6.4	Key Declarations	15
	Bibliography	16

1 The Rational Background of ALEP

The formalism supported by the ALEP virtual machine (VM) has been developed on the basis of the ET-6.1 design study [Alshawi et al., 1991], and readers interested in more details than are given here, are also referred to that document¹.

During the implementation of the first ALEP prototype, known as ALEP-0 (cf. [Simpkins, 1992]), it was decided that parts of the original ET-6.1 specification had to be changed in order to produce an efficient prototype for groups involved in CEC projects to experiment with. The most significant changes were made to the *type system* and the definition of *categories*. Atomic syntactic categories were replaced by complex objects called *types* which are, possibly recursively, structured through attributes with values which also must be typed. This view was also adopted for the industrial ALEP-1 implementation within the ET-9.1 project.

The chosen approach corresponds to the one adopted by other current linguistic formalisms, such as HPSG ([Pollard and Sag, 1993]), and actual implemented systems, such as the ALE system designed by Carpenter ([Carpenter, 1992a]), the TFS system ([Emele et al., 1990], [Zajac, 1992]) of the University of Stuttgart (IMS), or the *TDL* system of the DFKI Saarbrücken ([Krieger and Schäfer, 1993]). The idea behind this model is the theory of abstract data types, and its adaptation in computer science, viewing, for the first time, the construction of (larger) grammatical descriptions as a software engineering task in a broader sense.

Under this model, the process of designing and implementing a grammar and a lexicon for a given language fragment, changes quite a bit from the way it used to be. Rather than collecting rules and lexical entries without further internal organisation of the sets of entities, the process is now divided into a step where a declaration defines which types (data structures) will be used (in the grammar and lexicon), followed by the specification of any other linguistic knowledge (grammar rules, translation rules and lexical entries) under the organisation declared before.

Writing grammars is thus much closer to writing a computer programme, or rather an abstract specification (*lingware*), since we try to abstract away as much as possible from procedural issues, than it was before; this is getting to be known as *grammar engineering* or, more general, as *linguistic engineering*.

If then the type system also allows for the incorporation of *inheritance*, we also gain a much better compactness of our description, because we state information only in one place and inherit it from there to other locations where it is needed.

The core of the ET-6.1 formalism which formed the basis of what is now implemented in ALEP, is kept simple and follows a rather conservative design. Nevertheless, the formalism is powerful enough to allow grammars with an uncontrollable runtime behaviour to be written, so-called *pathological grammars*. This, because the decision in the current ALEP implementation is, to put a ma-

¹The examples used in the description here are mostly taken from the user manual of the ALEP virtual machine and from IAI's lingware reports of the ET-9.1 project. A complete description of an implementation in the field of terminology can be found in [Schütz, 1994].

major part of the burden of ensuring good runtime behaviour of the grammar on the grammar writer, rather than extensively using compilation techniques and other software engineering techniques that would extract from the specification the information that the grammar writer now has to specify by hand².

2 Type System Specifications

The type system constitutes the locus where the user has to define the form and permitted content of complex linguistic objects. The design of the type system language of ALEP follows three basic principles:

- The system has the characteristics of a strongly typed programming language: the arity of every feature term, and the names of each of its arguments (attributes) is specified, as well as the type of values that each argument can hold.
- The instances of structures found in grammatical rules can be fully compiled into Prolog terms; actually they are compiled into native machine code. Those found in lexical rules are compiled into Prolog clauses which are stored in an external clausal database (ClauseDB by BIM).
- The usage of the type system is restricted to compile time, i.e. no use of type information, both about well-formedness and more importantly about inheritance, is being made at run time. This may help for efficiency, but takes away the possibility to utilise the compactness of the type specification during the debugging of a grammar.

As we can see, the designers of the formalism adopted a more conservative approach to the use of type systems in the ALEP environment than other systems, such as ALE, *TDL* or TFS, which integrate the type system more thoroughly with the rest of the specification and the processing machinery, paying a slightly higher price in terms of possible efficiency, but also gaining (probably) a lot in terms of expressivity and perspicuity of the linguistic specifications that can be achieved.

A very important restriction on the ALEP type system is that it has to be constructed in a strictly hierarchical manner, thus *multiple inheritance* is not supported.

2.1 Primitive Types

ALEP supports a small number of primitive types out of which more complex types can be defined by the user:

- any

²One example is the specification of which daughter node in a grammar is supposed to act as a *head* during analysis. The problem here is, that the head specification that gives the most efficient behaviour of the parsing algorithm not always corresponds to the *linguistic head* that the grammar writer's intuition and the linguistic theory would specify.

- atom
- boolean
- functor
- list
- tuple

Complex objects such as rules and lexical entries are internally structured by means of attributes and values associated with them. The values themselves have to be typed as well.

For example,

```
lu => atom
lu => Lu
subcat => list(atom)
subcat => [np, vp]
```

are all valid declarations for attributes. Since the notation is very close to Prolog (the implementation language of ALEP), the notation for variables and the rules for variable sharing used in Prolog hold here as well. In the strongly typed setting, shared variables have to be *type compatible*.

Boolean Type Declarations

In order to have simplified and more efficient access to boolean combinations of (mostly) atomic valued features, an attribute can be declared to be of type boolean:

```
<feature_name> => boolean([<atom_sets>]).
```

where <atom_sets> is any number of enumerations of atom sets (separated by ',') for example:

```
agr => boolean([ {sing, plural}, {'1', '2', '3'} ]).
```

The data type created by a boolean declaration can then subsequently be used in complex expressions using boolean (infix) operators:

```
A ; B           % (infix) disjunction (highest priority)
A & B           % (infix) conjunction
~ A             % (prefix) negation (lowest priority)
```

The agr type declaration above allows then the use of expressions such as

```
ld:{ ... , agr => sing ; '3', ... }
ld:{ ... , agr => ~plural, ... }
ld:{ ... , agr => ~(plural & '1'), ... }
```

which would internally be compiled out into the corresponding boolean compilation of atomic expressions.

Functor Type Declarations

The built-in/primitive type functor is used to specify that an attribute takes an ordinary Prolog term as its value:

```
<term> => <atom> ['(' <args> ')'] | <variable> .
<args> => <term> [' , ' <args> ] .
```

For example, given the declaration

```
lf => functor
```

we can write for example:

```
lf => _
lf => A % a shared variable.
lf => man(X) % a logical form.
lf => exists(X, man(X)) % a complex logical form.
```

The most frequent use for functor type values is in the specification of *semantic forms*, which are supposed to have no further internal structure relevant to the processing inside the ALEP setting.

Tuple Type Declarations

This type allows for values being a tuple, for example

```
gaps => tuple
```

which then allows a value of the form

```
gaps => (gaps_in:{ ... }, gaps_out:{ ... })
```

in lexical entries and rules.

Enumerated Data Types

For the atomic types, it is possible to define subtypes by enumerating subsets of their extension. This is called the unstructured enumerated type and has the form:

```
<primitive_type> <sub_range>
```

That is, a primitive type (`atom`, `list`, etc.) is further restricted/enumerated to a specific set of values (`atoms`, `element types`). For example:

```
num => atom({sing, plural})
cat => atom({n, v, adj, adv})
lex => atom
```

Note that the builtin type functor cannot be enumerated. In particular, it is not possible to restrict terms to have a particular functor, or particular instances of arguments.

Lists must only have elements of one type. It may thus be necessary to introduce a type just for this purpose, which would otherwise not be independently motivated. The notation allows to enumerate them in terms of the type of their elements, as a sort of on-the-fly type declaration for the elements without naming this type:

```
subcat => list(type({sign:{}}))
subcatp => list(atom({s, np, vp}))
nested => list(list)
```

2.2 Structured Types

A formalism useful for the description of complex linguistic objects has to allow for the definition of types with some amount of internal structure. Structured types (user defined types) group and name a number of attributes (their names and types) into a single data type. The basic form of a type declaration is:

```
'type(' <type_name> ':'{
    <attribute_name1> => <attribute_type1>,
    <attribute_name2> => <attribute_type2>,
    ...
    <attribute_nameN> => <attribute_typeN> }',
<comment_string> ').'
```

For example, a type `syn` might be declared as:

```
type(syn:{ cat    => atom({s, np, vp, n, v, adj, pp, p, det}),
           morph  => list(atom)}, '').
```

For instance, a declaration such as

```
type(sign:{cat    => atom({n, v, det, pn, s, np, vp}),
           lu     => atom,
           case   => atom({subj, obj, iobj}),
           vform  => atom({bse, fin, ing, en}),
           subcat => list(type({sign:{}}),
                          lf => functor}, '').
```

would allow us to define the following:

```
sign:{ cat => v,
       lu => eat,
       vform => bse,
       subcat => [sign:{cat => np, case => obj, lf => 0},
                 sign:{cat => np, case => subj, lf => S}
                ],
       lf => eat_ingest(S, 0)}.
```

The structured type defined above can then be used to describe the values of other attributes in subsequent type definitions.

2.3 Type Hierarchy

Besides supporting the definition of structures, the type system is able to express relations between types in a hierarchical manner, thus maintaining a scheme of inheritance of information between the types. The design decision in the current version of ALEP foresees that this hierarchical organisation and the inheritance scheme can only be used at compile time; all the inherited information is fully compiled out and the structures are present in the database in their fully expanded form.

As an example consider a structured type declared as follows:

```
type(n_v_heads:{num  => atom({sing, plural}),
                pers => atom({'1', '2', '3'})}, '').
```

The fact that `n_v_heads` does have sub-types can be declared by specifying the hierarchical relationship:

```
n_v_heads > {n_head, v_head}.
```

The declaration of the two sub-types itself then is:

```
type(n_head:{gender => atom({masc, fem}),
         case  => atom({acc, gen, dat})}), '')).
type(v_head:{tense => atom({pres, past})}, '').
```

Note that the inheritance information has to be specified through the hierarchy specification. There is always only exactly one supertype for each given type, since the ALEP type scheme does not allow multiple inheritance.

Attributes are classified as either being local (for the type they are directly declared for) or inherited (for types, where the attribute is declared only at one of the supertypes). Given the declaration above, the three types can then be used in rules, for example:

```
sign:{..., head => n_v_heads:{ num => sing }, ... }
sign:{..., head1 => n_head:{ num => sing, gender => masc }, ... }
sign:{..., head2 => v_head:{ num => plural, tense => past }, ... }
```

A declaration like

```
sign:{..., head => n_v_heads:{ gender => masc }, ... }
```

is not permitted, and is captured at compile time, because the type `n_v_heads` has no local and no inherited definition for the attribute `gend`.

3 Defaults

Within a type declaration it is possible to define default values for attributes. At compile time, if an instance of an attribute does not have a value then the default value is assigned. This, however, does not apply to specified variable sharings, i.e. they will not be overwritten by default values. The declaration specification then is:

```
<feature_name> => <feature_type> => <default_value>
```

The default value must be type compatible and can be complex, like in:

```
head => type({n_head:{}, v_head:{}) => n_head:{head_type => n,  
                                             num           => sing}
```

4 Macros

Macro definitions are an integral part of the ALEP formalism. They allow for abbreviations and ease lingware maintenance. Macros may be defined in a type system and apply over all linguistic specifications compiled under that type system. All macros are expanded at compile time. A macro definition has the form:

```
'macro(' <macro_name> '[' <macro_args> '], ' <macro_body> ').'
```

The argument part of a macro definition can be empty (cf. the examples below). The body of a macro can include calls to other macros; cyclic macro expansions are detected by the compiler and a warning message is given.

An example for a macro definition is:

```
macro(agr[G, N, P, D], agr:{group=>G, num=>N, pers=>P, def=>D}).
```

with it's associated type declaration:

```
type(agr:{group => atom({proper, common}),  
       num     => atom({sing, plural}),  
       pers   => atom({'1', '2', '3'}),  
       def    => atom({def, indef})  
     }, 'feature structure describing noun agreement').
```

Other macro definitions can be based on this definition, for example

```
macro(sing_n[G, P, D], agr[G, sing, P, D]).
```

An example of a macro with no arguments is:

```
macro(third_person_sing[], {num => sing, pers => '3'}).
```

which can be called within any type where these specifications are legal, for example:

```
ld:{syn => syntax:{gender => masc, third_person_sing[]}}.
```

This will be expanded to:

```
ld:{syn => syntax:{gender => masc, num => sing, pers => '3'}}.
```

An alternative approach for the above ‘adding’ of features to an existing macro is supported by the ‘where-clause’. For example given a macro

```
macro(agr[], t:{num => sing}
```

another macro can be defined which is like the agr macro but which also defines a value for pers:

```
macro(p_agr[], agr[], where({pers => '3'})).
```

The use of arguments in such macros is prohibited. The compiler of the current system version (ALEP release 1.2) allows values defined in one macro to be overwritten by a ‘where’-statement.

5 Structures and Rules

5.1 Phrase Structure Rules

The description of the type system and type declarations has introduced the notion of types, which describe what is usually termed (complex) feature term or feature structure. In modern computational linguistic theories these data structures are the basic building blocks of nearly all linguistic specifications. Since the ALEP designers decided to base the formalism on an obligatory well-elaborated context-free backbone architecture, definitions of dominance and order are needed and are introduced by an extension of the traditional context-free skeleton.

Notational conventions such as repetition (Kleene star), optionality and disjunction are either not present or only usable in a limited fashion. These must either be expressed within the core formalism (cf., for example, the discussion of boolean types and expressions and Kleene rules in [Alshawi et al., 1991]) or they will be compiled out; for example, disjunction over types in a rule produces multiple rules.

Rules are then built from types as defined above, also referred to as *Linguistic Descriptions* (LD), instead of atomic categories, incorporated in the context-free skeleton. The BNF for grammar rules thus is as follows:

```
<rule> ::= <LD> '<' '[' <daughters> ] ']' .'
```

A simple 's → np vp' rule example would then look as follows:

```

/**** s_np_vp1 ****/

ph:{syn => syntax:{cat=>s, gaps => gaps:{in=>[], out=>[]}},
  log => logform:{lf=>VP},
  spec => specifier:{lang=>en, gram=>toy, id => 'S_NP_VP' }
}
< [
  ph:{syn => syntax:{cat=>np},
    log => logform:{lf=>NP}
  },
  ph:{syn => syntax:{cat =>vp},
    sem => semantics:{subj=>NP},
    log => logform:{lf=>VP}
  },
  wd:{syn => syn_lex:{cat=>punct, lu=>&fs}}
].

```

In this rule the second (vp) daughter of the rule would function as head in analysis and synthesis mode.³

5.2 Lexical Entries

A lexical entry is simply an LD with an identifier (usually the surface string), for example:

```

every ~
  sign:{syn => syn_lex:{cat => det,
    lu => every},
  log => logform:{lf => forall(X,N),
    var => X,
    scope => N}
}.

```

³More on the definition and use of the head specification is said in section 6.1.

5.3 Transfer Rules

A transfer rule defines a mapping of a linguistic structure from one type system to a partial linguistic structure in another type system. The form is:

```
'trule( ' <specifier_source>,  
        <specifier_target>,  
        <direction>,  
        <source_expression>,  
        <target_expression>,  
        '[' <conditions> ' ]'  
    ).'
```

The specifier are the identifier for the source and target type system as defined in the `ld_specifier_feature` (cf. below). `<direction>` is an operator defining the direction of the rule's mapping; it can either `=>`, `<=` or `<=>`. The expressions are compiled according to the respective type system. `<condition>` is a (possibly empty) list of transfer conditions; we distinguish between recursive conditions - indicated by `<expr1> '==' <expr2>` - and matching conditions - indicated by `'='` between the expressions - where `<expr>` is either a type (LD), list, tuple or variable which is compiled according to the respective type system.

An example of a transfer rule is:

```

/**** trans_pred_senden ****/

trule(de, en, <=>,
      sem_fs:{gov => v_sem:{pred => senden,
                          predtype => PT1},
             args => ARGS1 => trival:{},
             mods => MODS1},
      sem_fs:{gov => v_sem:{pred => send,
                          predtype => PT2},
             args => ARGS2 => trival:{},
             mods => MODS2}
      [ARGS1 == ARGS2,
       MODS1 == MODS2,
       PT1 = PT2]).

```

Here, a feature structure of the analysis process is translated into an appropriate target feature structure which then is input to the generation process; this transfer rule is reversible.

6 Procedural Issues

6.1 Parsing and Generation

For efficient lingware execution it is important to understand something of the internal algorithms of the ALEP prototype and to reflect these algorithms in the approach to coding. This does not necessarily mean that linguistic notions, approaches and concepts must be compromised, but that there is some difference between a linguistic notion and how it should be formally expressed in the user language for optimum efficiency.

The ALEP system includes two different analysis parsing algorithms, a non-deterministic bottom-up head-out parser (basic version) and a structure sharing algorithm where records are packed by the subsumption relation (record version), and one generation (synthesis) algorithm. All three of these are based around the notion of *head*.

The two analysis parsers use what is basically the same algorithm in terms of string concatenation. It is important to note that the choice of a rule head element can cause failure respectively success of parsing.

A head element here is not the same notion as a linguistic head. It can be thought of as a *starting point* for parsing or generation.

Bearing in mind the complexity of the analysis algorithms, a grammar should be such that ambiguity is reduced and based on a *shallow* analysis, i.e. many constituent analyses are packable. Deeper analyses can then be produced by applying rules in a subsequent *refinement* operation. This mechanism can be used in an implementation to account for an instance of *performance control*. With *refinement* a given linguistic structure can be tested according to

specific features which can be accessed through *specifier* declarations (cf. below), and thus being subject to a well-formedness test and additional information assignment. This method is used as a device for *performance control* which is facilitated by general semantic information and domain-specific information in [Schütz, 1994].

6.2 Rule Head Declarations

Heads are described by `select_<mode>_head` declarations where `<mode>` is either `analysis` or `synthesis` and specifies the compilation mode of the rule set. As parameters we have to specify a `<root_ld>` which identifies a left-hand side of a grammar rule and a `<head_ld>` which identifies a daughter of the right-hand side of the rule as being a *head daughter*. Based on these specifications the reflexive transitive closure of the head relation is computed (at compile time) which then serves as a kind of predictor for the parsing respectively generation algorithm. In order to avoid misleading head relations (and thus predictions) within the closure we have to be as specific as possible when declaring head selections.

6.3 Specifier Declarations

A specifier is a generalisation of a rule's or lexical entry's identifier. It allows for a partitioning of a grammar into subgrammars; for example, to divide a grammar into rule sets for analysis and synthesis, or rules for syntactic and semantic processing. A specifier is the value of some attribute within the left-hand side of a rule. The path to this attribute is specified by a declaration of the form:

```
'ld_specifier_feature(' <variable> ',' <ld> ').'
```

where `<ld>` is a linguistic description which contains the variable `<variable>` as the value of the specifier attribute. Having declared this path a number of named specifier values can be declared by:

```
'spec(' <name> ',' <value> ').'
```

where `<value>` is the name of the specifier and `<value>` is a legal value of the specifier attribute as defined in the path declaration.

For example, given the following type declarations

```
type(sign:{spec => type({apply:{}}, ... }, 'ld').
type(apply:{rules => atom({thls, seg, analysis, refine}),
        language => atom({de, en})
      }, 'rule specifier').
```

the path to the specifier attribute can be defined as:

```
ld_specifier_feature(Spec, sign:{spec => Spec}).
```

and some specifier can be defined:

```
spec(segment, apply:{rules => seg, language => de}).  
spec(parse, apply:{rules => analysis, language => de}).
```

6.4 Key Declarations

Another procedural element of the ALEP formalism is the notion of keys. Keys are given as part of a type system and are used for efficient lingware execution, but they are not required. Keys are used to index rules and lexical entries, thus allowing for an efficient retrieval during runtime. Like head selection declarations keys are declared according to their application mode (analysis and synthesis). For lexical entries key declarations have the form:

```
ld_<mode>_key( <variable>, <ld> ).
```

where <mode> is either ana or syn, and <ld> is a linguistic description which specifies the path to the chosen key attribute value. For example:

```
ld_ana_key(A_Key, sign:{pho => pho_fs:{string => [A_Key|_]}}).  
ld_syn_key(S_Key, sign:{sem => sem_fs:{gov => gov_fs:{pred => S_Key}}}).
```

In the case of the (abbreviated) lexical entry:

```
gibt ~  
  sign:{spec => spec_fs:{lang => de, ... },  
        pho => pho_fs:{string => [gibt | _]},  
        sem => sem_fs:{gov => gov_fs:{pred => geben, ... }}}.
```

the analysis key will get the value `gibt` and the synthesis key the value `geben`.

Keys for rules are also declared according to the mode of compilation. Rules can be indexed either by their mother LD (left-hand side) or by their head daughter LD (of the right-hand side):

```
'ld_ana_mother_key(' <variable> ',' <ld> ').'  
'ld_ana_head_key(' <variable> ',' <ld> ').'
```

For example:

```
ld_ana_mother_key(Key, sign:{syn => syn_fs:{mkey => Key}}).  
ld_ana_head_key(Key, sign:{syn => syn_fs:{hkey => Key}}).
```

If for a particular rule or lexical entry the compiler cannot find an applicable key path specification, or if a key attribute is uninstantiated then a warning message is given.

References

- [Alshawi et al., 1991] H. Alshawi, D. J. Arnold, R. Backofen, D. M. Carter, J. Lindop, K. Netter, S. G. Pulman, J. Tsujii and H. Uszokoreit, 1991. ET-6/1 Final Report. CEC, DG-XIII, Luxembourg.
- [ALEP, 1993] **ALEP Documentation Package, Vol. I and II**. CEC and PE International, Luxemburg.
- [Carpenter, 1992] R. Carpenter, 1992. **The Logic of Typed Feature Structures**. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, Cambridge, MA.
- [Carpenter, 1992a] R. Carpenter, 1992. **ALE - The Attribute Logic Engine. User's Guide Version β** . CL Lab, Carnegie Mellon University, Pittsburgh.
- [Emele et al., 1990] M. Emele, U. Heid, W. Kehl, S. Momma and R. Zajac, 1990. **Organization Linguistic Knowledge for Multilingual Generation**. Technical Report, Project POLYGLOSS, University of Stuttgart.
- [Krieger and Schäfer, 1993] H. -U. Krieger and U. Schäfer, 1993. ***TDL* - A Type Description Language for Unification-Based Grammars**. Ms., DFKI, Saarbrücken.
- [Pollard and Sag, 1993] C. J. Pollard and I. A. Sag, 1993. **Head-Driven Phrase Structure Grammar**. CSLI Lecture Notes, Stanford, California.
- [Schütz, 1994] J. Schütz, 1994. **Terminological Knowledge in Multilingual Language Processing**. Studies in Machine Translation and Natural Language Processing, Vol. 5. CEC, DG-XIII, Luxembourg.
- [Simpkins, 1992] N. K. Simpkins. 1992. **ALEP-0 Version 2.2 – Prototype Virtual Machine, User Guide**. CEC DG-XIII, Luxembourg.

[Theofilidis, 1993] A. Theofilidis, 1993. ET-9.1 Lingware Report - Phase 2. IAI Working Papers, No. 26, IAI, Saarbrücken.

[Zajac, 1992] R. Zajac. 1992. Inheritance and Constrained-based Grammar Formalisms. In: Computational Linguistics, 18(2), p. 159-182.