TRALE feature logic

Johannes Dellert

Seminar für Sprachwissenschaft, Universität Tübingen

January 09, 2008

Facts about ALE

- ALE stands for Attribute Logic Engine
- it is the name for both a formalism and its Prolog implementation
- a logic programming environment in which terms are typed feature structures
- the feature logic we will have a look at is in essence an attribute-value logic with variables
- understanding this logic is essential to be able to write grammars in ALE
- ALE will compile such grammars into parsers

Facts about TRALE

- TRALE is an extension of the ALE system that supports some extra functionality
- on the logical level, the main difference between ALE and TRALE is the view on subtyping
- since we will mainly be dealing with TRALE, everything presented today will be from the TRALE perspective
- we will mostly be concerned with TRALE signatures and their meaning

Outline

1. Basic type system, type hierarchies, TRALE signatures
2. Feature structures
3. Subsumption und unification
4. Enhanced type system, feature appropriateness in signatures
5. Outlook: TRALE descriptions, Attribute-Value Logic

Overview: ALE type system

- ALE is a language with **strong typing** - every structure it uses comes with a type
- the user must declare all the of types that will be used
- types are arranged in an **inheritance hierarchy**
- many types will typically have **subtypes**
- constraints on more general types are inherited by their subtypes: **inheritance-based polymorphism**

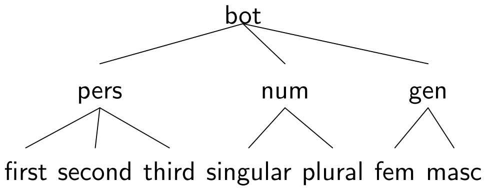
Inheritance Hierarchies

- subtyping relation only specified by **immediate** subtyping declarations
- however, the subtyping itself is **transitive**
- it is also **anti-symmetric** ($a < b \wedge b < a \rightarrow a = b$)

$\longrightarrow$ subtyping constitutes a **partial order** on the types

- there are additional restrictions:
  - there must be a unique most general type named **bot**
  - type hierarchies must be **bounded complete** (explanation to come)

A simple example



The signature in TRALE format

```
type_hierarchy
bot
    per
        first
        second
        third
    num
        singular
        plural
    gen
        fem
        masc
.
```

Bounded completeness

- if a poset is **bounded complete**, every subset that has some upper bound must also have a least upper bound
- for type hierarchies: every collection of types with a common subtype must have a **unique most general common subtype**

Problematic:

```
type_hierarchy
bot
  a
    c
    d
  b
    c
    d
.
```

Solution:

```
type_hierarchy
bot
  a
    e
      c
      d
  b
    e
.
```

Questions?

Feature Structures in ALE

- primary representational device in ALE
- serve as universal data structure
- a feature structure consists of
  - a type drawn from the inheritance hierarchy
  - a finite (possibly empty) set of **feature/value pairs** where each value is in itself a feature structure ($\longrightarrow$ **recursion**)
- Examples in ALE output notation:

```
noun              verb
CASE akk          PERS first
NUM pl            NUM pl
GEN f             SUBCAT ne_list
                         HD noun
                            CASE akk
                         TL e_list
```

Substructures and structure sharing

- a **substructure** is the value of a feature at some level of nesting
- two different substructures can be **token-identical** $\longrightarrow$ **substructure sharing**
- a **path** is a sequence of features designating a substructure
- if two paths are in a structure sharing relation, their values are token-identical
- structure sharing is indicated by numbered **tags** in the feature structures

Cyclic structures

"This sentence is false."

[0] false
    ARG1 [0]

- this is a **legal** feature structure!
- structure sharing between paths: ϵ, ARG1, ARG1 ARG1, ARG1 ARG1 ARG1 ...

"It is false that this sentence is false."

false
ARG1 [0] false
            ARG1[0]

- the two structures are not treated as identical, a cyclic structure is not conflated with its infinite unfolding

Questions?

Subsumption and Unification

- feature structures provide partial information (Why?)
- we can order feature structures based on the amount of information they provide
- this order is called the **subsumption ordering**, providing a formal notion of **information containment**
- with subsumption, we can define **information combination** (**unification**)
- unification is the standard mechanism to perform computations with feature structures

Subsumption

- let F, G feature structures
- F subsumes G iff
  - the type of F is more general than the type of G
  - if a feature f is defined in F then f is also defined in G such that the value in F subsumes the value in G
  - if two parts are shared in F then they are also shared in G

Subsumption - Examples

agr PERS first	<	agr PERS first NUM plu

sign
SUBJ agr
    PERS first
    NUM plu
OBJ agr
    PERS first
    NUM plu
	<	sign
		SUBJ [0] agr
		    PERS first
		    NUM plu
		OBJ [0]

Unification

- **unification** is an operation defined over pairs of feature structures returning a feature structure combining the information contained in the input structures if they are consistent and fails otherwise
- the result is the most general feature structure subsumed by both input structures
- from an operational point of view:
  - unify the types of the structures according to the hierarchy (unique result required, this is why we need bounded completeness)
  - recursively unify the values of the features which occur in both structures
  - if a feature only occurs in one structure, copy it over into the result

Unification - Examples

agr PERS first	+	agr PERS second	=	*failure*
agr PERS first	+	agr NUM plu	=	agr PERS first NUM plu

sign
SUBJ agr
    PERS 1st
OBJ agr
    NUM plu
	+	sign
		SUBJ [0] bot
		OBJ [0]
			=	sign
				SUBJ [0] agr
				    PERS first
				    NUM plu
				OBJ [0]

Unification - Examples

t
F [0] t
G [0]
	+	t
		F t
		    F [1]
		G [1]
			=	t
				F[1] t
				    F[1]
				G [1]

bot	+	t	=	t
e_list	+	ne_list HD a TL e_list	=	*failure*

Questions?

Extending the type system

- so far, our notion of the type system was built almost entirely around the notion of subtyping
- with the features in place, we can now add the notion of **feature appropriateness**
- each type must specify which features it can be defined for, and which types of values these features can take
- these appropriateness specifications are inherited in the type hierarchy

A TRALE signature with features

```
type_hierarchy
bot
    list
        e_list
        ne_list hd:bot tl:list
    atom
        a
        b
.
```

Restrictions on appropriateness conditions

- appropriateness conditions must be acyclic
- every feature must be introduced at a unique most general type
- each type must specify which features it can be defined for, and which types of values these features can take
- these appropriateness specifications are inherited in the type hierarchy

Well-typedness

- every feature structure must respect the appropriateness restrictions
- this amounts to two conditions on feature structures:
  - if a feature is defined on a structure, its type must be appropriate for the feature and the value of the feature must have the appropriate type
  - if a type was declared with a feature, every feature structure of that type must have a value for the feature
- feature structures fulfilling these conditions are called **totally well-typed**

Subtype covering

- TRALE assumes that subtypes exhaustively cover their supertypes
  $\rightarrow$ every object of a non-maximal type is also of one of the maximal types subsumed by it
- the following hierarchy fragment illustrates what this implies:

t f:bool g:bool
    t_1 f:+ g:-
    t_2 f:- g:+

- this signature will not allow structures like the following:

t
F +
G +

Questions?

From types to descriptions

- so far, the feature structures we allowed for where only specified by a signature
- the constraints we can impose in this way are rather restricted
- trying to define a language like this inevitably leads to severe overgeneration
- to write grammars, we need another device that allows us to impose much more complex conditions on valid feature structures
- to achieve this, we need a syntax to write rules and principles in a way similar to logic programming
- to generalize over feature structures, we need a description language that allows us to select classes of structures by means of properties that we are interested in
- for this purpose, we will introduce TRALE descriptions

Conclusion

- We know
  - how signatures are written and what they mean
  - how feature structures and their unification are defined
  - a lot of basic facts about TRALE's feature logic
- In the next session, we will learn about
  - atomic values and why they are not in the signature
  - the difference between token identity and structural identity
  - the syntax of basic descriptions in TRALE, allowing us to start writing grammars

Thank you.

TRALE descriptions

- the set of descriptions used in ALE can be described by the following grammar:

```
<desc> ::= <type>
         | <variable>
         | (<feature>:<desc>)
         | (<desc>, <desc>)
         | (<desc>; <desc>)
         | (=\= <desc>)
```