Parsing logic grammars

- Introduction
- Prolog as parser
- Parsing open FS
  - Schöter
- Parsing typed FS
  - ALE (Carpenter)
  - Amalia (Wintner)
Phrase Structure Grammars vs Logic Grammars

- terminal and non terminal vocabularies
  - Tags belonging to a close tagset
  - Open expressions
    - terms from a 1st order predicate logic (e.g. Prolog terms)
    - Feature Structures, FS
  - ...

...
• Logic Grammars vs Unification Grammars
  • How important is unification
• Adaptation of classical algorithms dealing with CFG
Unification importance

Tomabechi, 1991

- 85% - 90% of parsing time
- 75% - 95% from this time: structures copying
- > 60% of unifications in correct parses fail

- Other operations
  - subsunption, generalization, inheritance, disjuntion, negation, reescripture, assignment
Parsing logic grammars

- Dag unification vs term unification
- Representation of FS
- Compilation:
  - FS $\rightarrow$ Prolog terms, vectors of bits, logical predicates
  - P-Patr (Hirst, 1988), CLE (Hinrich, 1992), Schöter
  - HPSG $\rightarrow$ Tag (Kasper, Kiefer)
  - Typed FS: ALE (Carpenter, 1993), ALEP, TFS, CUF
- WAM (Warren Abstract Machine)
  Carpenter, Penn (ALE), Wintner (Amalia)
• Enrich parsing schema with information that can be added to the CF core of the grammar

• Feature addition
  \[ \forall \varphi_0 (A \rightarrow \alpha) \] for productions
  \[ \forall \varphi (X) \] for constituents
  \[ \forall \varphi ([A \rightarrow \alpha \cdot \beta, i, j]) \] for items

• Applicable to CKI, Earley, LR, ...

\[
\varphi(\xi) \sqsubseteq \varphi(\eta) \quad \text{subsumption}
\]

\[
\varphi(\xi) \sqcup \varphi(\eta) \quad \text{unification}
\]
Prolog as parser

• Pros
  • Parsing as theorem proving
  • A specific parser is not needed
  • Incorporated to formalism: DCG
  • (Relatively) easy to write the grammar
  • (Relatively) easy to integrate
Prolog as parser

- Cons
  - TD fixed strategy
  - Fixed left to right management of literals
  - Fixed order of clauses (grammar rules)
  - Problems with left recursivity
  - Backtracking
    - Computation redundancy
    - Generation of useless (sub) parse trees
  - Lack of flexibility on parsing
• Improvements
  • (Stabler, 1990)
  • Separation of grammar clauses from lexicon ones.
  • Reordering of objectives to satisfy (literals).
    • static and dynamic (when trying to satisfy right part of the clause)
  • Unfolding.
    • Replacing part of the literals of the right part of a clause by the unfolding of the involved clauses.
  • Partial evaluation.
• Improvements
  • Generation and test.
    • performing tests asap within a conjunction.
  • Using selectional functions domain specific
  • Literal reordering in the grammar using statistical techniques taking into account the number of arguments, their complexity, their instantiation level, etc...
  • dynamic pruning (e.g. limitations on the number of recursive steps)
  • Saving of intermediate results (positive or negative)
• Improvements
  • Kiefer et al, 1999
  • Precompiling the lexicon
  • expansion and application of lexical rules
  • removing parts of FS useless for parsing
  • Improvements in unification
    • quasi-destructive algorithm of Tomabechi
    • reusing parts of input structure in the output (Pereira)
  • Precompilation of type unification
  • Precompilation of filtering rules
Prolog as parser

• Improvements
  • unification dynamic filters ("quick check")
  • Reduction of the size of FS using restrictors
  • Limitation of the initial number of items of chart
  • Computing and saving best partial analysis
• Bottom Up Parser
  • Traduction, at compilation time, of grammar rules, in DCG, to a type of Horn clauses, BUP clauses, that will be further processed by Prolog.
  • The resulting parser incorporates in BUP clauses the control mechanism ("left corner")
  • [Matsumoto et al, 1983], [Matsumoto et al, 1985]
  • SAX (Matsumoto, Sagimura)
Prolog as parser

original DCG grammar

BUP grammar

Using the BUP grammar as a conventional DCG
P-Patr (Hirsh, 1988)
1) Computing arity and composition of all the complex FS
2) attribute/value pairs in correct position. Closed lists.
3) Attribute removing
4) Analysis following a left corner strategy
Parsing with open FS

sentence → np, vp
  <sentence head> === <vp head>
  <vp head agr> === <np head agr>

would be represented as:

```
sentence([head:[agr:Y]]) -->
  np([head:[agr:Y]]),
  vp([head:[agr:Y]]).
```

components of the backbone CFG perform as functors and equations are reflected into the arguments

look out!! Part of the information originally shared, is now opied.
• Compilation Patr → Prolog terms
  • From the grammar a set of reference DAGs is extracted.
  • A typed specification of the grammar is generated
  • This specification is used for building a set of translation schemas from dags to terms.
    • access functions
  • translation schemas and access functions are applied over the initial grammar for building the terms grammar

Schöter 1993
Parsing with open FS

PATR equations \rightarrow Prolog DAGs

Partial evaluation \rightarrow Prolog DAGs

Partial evaluation \rightarrow flat terms

Partial evaluation \rightarrow Hirsh-style lists

Compilation \rightarrow flat terms

Compilation \rightarrow Hirsh-style lists
rule(1, tran, VB, V, NP):-
  V: subcat === 1,
  cat_macro([v, bar(0)], V),
  cat_macro([norm, bar(2)], NP),
  cat_macro(bar(1), VB),
  VB: head === V: head,
  V: subj ===('-',
  VB: subj === V: subj,
  NP: tree === A,
  V: tree === B,
  VB: tree ===
  vb(A, B).
Parsing with open FS

Partial evaluation

rule(1, tran, A, B, C):-
    C===[head:[cat:[maj:[n:+, v:-], D], min:nil | E]...],
    B===[subcat:1, head:[cat:[maj:[n:-, v:+], I]...],
    C===[bar:1, head:[cat:[maj:[n:-, v:+], I]...].
Partial evaluation

DAG:Path <= Value
(A===B;C===D)
  cannot be precompiled, they have to be evaluated during parsing

rule(nil,bare_np,A,B):-
  B===[head:[cat:[maj:[n:+,v:-|C],min:nil|D]...],
  A===[bar:2,head:[cat:[maj:[n:+,v:-|C],min:nil|D]...],
  D]...],
  (A:head:agr:num === pl
  ;      A:head:ntype === mass
  ).
Terms should be maximally instantiated
A stable feature inventory is needed
   → features set
   → enumeration

Using only the information implicit in the grammar
(there is no a previous type structure)

3 steps
1) Obtention of grammar's type structure (implicit in it)
2) Obtention of a set of templates for translating dags to terms
3) Apply the templates over all the dags of the grammar
**1st step**

- hierarchy type number (0 for simple terms)
- feature path
- set of values

- path(0, head:cat:n, [+])
- path(0, head:cat:v, [-])
- path(0, head:agr:per, [1,2,3])
- path(0, head:agr:num, [sg,pl])
- path(0, head:case, [nom,acc])
- path(0, head:nform, [it,norm])
- path(0, bar, [0])

---

Knowledge source: representative set of dags (by default the grammar, including the lexicon)

Possibility of manual tuning

Possibility of using a specification of the type structure
2nd step

1) Compiler template
   Skeletal maximal DAG whose values are replaced by variables
   Equivalent terms all of whose arguments are terms
   A set of type constraints on atomic values

2) Access functions

path(0, head:cat:n, [+]).
path(0, head:cat:v, [-]).
path(0, head:agr:per, [1,2,3]).
path(0, head:agr:num, [sg,pl]).
path(0, head:case, [nom,acc]).
path(0, head:nform, [it,norm]).
path(0, bar, [0]).

[dag0(A,B,C,D,E,F,G)]
Access functions
Replacing the functionality lost in moving from named grouped arguments to positional independent ones

```
a_func/3
a_func(Path, Term, Value)
```

```
a_func(head_cat, dag0(...,A,B,...), dag3(A,B)).
...
```

```
path(0, head:cat:n, [+]).
path(0, head:cat:v, [-]).
path(0, head:agr:per, [1,2,3]).
path(0, head:agr:num, [sg,pl]).
path(0, head:case, [nom,acc]).
path(0, head:nform, [it,norm]).
path(0, bar, [0]).
```

Functions (dagterm/1) are also generated to recognize terms as compiled DAGs
Using Access functions

rule(nil,bare_np,A,B):-
    B===[head:[cat:[maj:[n:+,v:-|C],min:nil|D]...],
         A===[bar:2,head:[cat:[maj:[n:+,v:-|C],min:nil|D]...],
              (                        
                 A:head:agr:num === pl
              ;
                 A:head:ntype === mass
              )].

rule(nil,bare_np,
    dag0(2,np(A),_,_,_,_,_,_,nil,-,+,norm,H,I,J,K,L,M,N,O,P,Q),
    dag0(1,A,_,_,_,_,_,_,nil,-,+,norm,H,I,J,K,L,M,N,O,P,Q)):-
    (a_func(head:agr:num,
             dag0(1,A,_,_,_,_,_,_,nil,-,+,norm,H,I,J,K,L,M,N,O,P,Q),
             pl);
     a_func(head:ntype,
             dag0(1,A,_,_,_,_,_,_,nil,-,+,norm,H,I,J,K,L,M,N,O,P,Q),
             mass)).
3rd step

Translating DAGs of the grammar into terms
Unifying the DAG to be compiled with the template's skeletal maximal DAG

```
rule(1,tran,A,B,C):-
    C===[head:[cat:[maj:[n:+,v:-|D],min:nil|E]...],
    B===[subcat:1,head:[cat:[maj:[n:-,v:+|I]...],
    C===[bar:1,head[cat:[maj:[n:-,v:+|I]...].
```

```
rule(1,tran,
    dag0(1,vb(A,B),_,C,_,_,_,_,nil,-,+D,E,F,G,H,I,J,K,L,M,N),
    dag0(0,A,1,C,_,_,_,_,nil,-,+D,E,F,G,H,I,J,K,L,M,N),
    dag0(2,B,_,_,_,_,_,_,nil,+,-,norm,_,_,_,_,_,_,_,_),
```
Transformations in the parser

**DAG1:** Path1 $\rightarrow$\ $\rightarrow$ DAG2: Path2

**DAG:** Path $\leftarrow$ Value

\[
\text{path\_to\_pathname(Path1,PathName1),} \\
\text{path\_to\_pathname(Path2,PathName2),} \\
\text{a\_func(PathName1,Term1,V),} \\
\text{a\_func(PathName2,Term2,V)}
\]

\[
\text{path\_to\_pathname(Path,PathName),} \\
\text{a\_func(PathName,Term,V0),} \\
\text{nonvar(V0),} \\
\text{V0 = Value}
\]
• **Restrictors:**
  - Forms of reduce search space
  - [Shieber, 1985], [Seiffert, 1987], [Gerdemann, 1993], [La Serna, 1998]
  - Potentially infinite domains
  - Algorithms implying prediction (e.g. Earley, TD charts)
• $R = \text{Restrictor} = \text{finite set of paths (subset of the set of paths, potentially infinite)}$
• $RD = \text{Restricted Dag (relative to } R)$
  - $RD$ subsumes $D$
  - For each path $p$ in $RD$, $\exists q$ in $R$ such that $p$ is prefix of $q$
  - $RD$ is the most specific dag satisfying these conditions
• Using $RD$ for prediction and $D$ for scanning and completion
• How to select restrictors:
  - cat: CFG backbone
  - Statistical model on the instanciation degree of features
• Typed FS
  • Any FS $f$ owns a typo $t$
    • $t \in T$, finite no empty set of types usually organized as a hierarchy
    • A FS $f$, with type $t$ allows a finite set of features, appropriate to $t$
    • A specific value, belonging to an appropriate type can be associated to each feature of FS

• Examples
  • ALE, Carpenter
  • Alep, Alshawi
  • AMALIA, Wintner
  • CUF, Dörre
  • TFS, Zajac
Example of type hierarchy in ALE

```
bot
  / \
list  atom
  / \     / \
 ne-list e-list a b
     \   \   
      bot sub [list, atom]
        list sub [e-list ne-list]
          e-list sub []
            ne-list sub []
              intro [hd:bot, tl:list]
                atom sub [a, b]
                  a sub []
                  b sub []
```
Parsing with typed FS

- Compiling Type structure $T$
  - transitive closure of subsumption in $T$
    - Wharshall’s algorithm
  - least upper bounds for each pair of consistent types
  - Appropriateness
    - appropriate features for each type
    - appropriate value for each feature

- Compiling basic operations
- Compiling descriptions
- Compiling grammar and programs
Implementing FS in ALE

Following Carpenter, 1993, a FS is represented as:
   Tag - foo (V1, ... Vn)

where
- Tag: reference pointer
- foo: type of the structure

Implementing FS in ALE
The following kind of code is generated:

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

```
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_atom(bot,T2).
```

```
add_to_word(sign(T1-Phon, Synsem, QSt), _-word(T1-Phon, Synsem, QSt)):-
    add_to_singleton_phon_list(Phon,T).
```
Abstractive Feature Structures

unifying two TFS: a program and a query => new TFS

Grammar rules and lexicon entries follow ALE format

The main difference between AMALIA and abstract machines devised for variants of Prolog (e.g. ALE) is that AMALIA is not based on resolution but on parsing with respect to an input grammar.
Example of Amalia type hierarchy

\[
\begin{align*}
\text{myth} & \quad f: \text{bool} \\
\text{bird} & \quad \text{wings: yes} \\
\text{beast} & \\
\text{adj} & \quad \text{n} \\
\text{cat} & \\
\downarrow & \\
\text{anim} & \quad \text{prd: bool} \\
\text{bool} & \\
\text{sign} & \quad \text{syn: cat} \\
& \quad \text{sem: anim} \\
\text{yes} & \quad \text{no} \\
\end{align*}
\]

\[
\text{anim sub[bird,beast] intro[prd:bool]}
\]
Parsing with typed FS

Representation in memory of a FS

e.g. TFS sign(myth(®yes,®),n)

<table>
<thead>
<tr>
<th>Address</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>STR</td>
<td>REF</td>
<td>REF</td>
<td>STR</td>
<td>REF</td>
<td>REF</td>
<td>STR</td>
<td>STR</td>
</tr>
<tr>
<td>contents</td>
<td>sign</td>
<td>4</td>
<td>5</td>
<td>n</td>
<td>myth</td>
<td>8</td>
<td>8</td>
<td>yes</td>
</tr>
</tbody>
</table>

Flattening FS:

<table>
<thead>
<tr>
<th>Linear representation</th>
<th>Set of equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign(myth(®yes,®),n)</td>
<td>$X_1 = \text{sign}(X_2, X_3)$</td>
</tr>
<tr>
<td></td>
<td>$X_2 = n$</td>
</tr>
<tr>
<td></td>
<td>$X_3 = \text{myth}(X_4, X_4)$</td>
</tr>
<tr>
<td></td>
<td>$X_4 = \text{yes}$</td>
</tr>
</tbody>
</table>
compiled code for the query \text{sign(myth(\textsf{yes},\textsf{yes}),n)}

<table>
<thead>
<tr>
<th>Set of equations</th>
<th>Compiled code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1 = \text{sign}(X_2, X_3)$</td>
<td><code>put_node sign/2, X_1</code></td>
</tr>
<tr>
<td></td>
<td><code>put_arc X_1, 1, X_2</code></td>
</tr>
<tr>
<td></td>
<td><code>put_arc X_1, 2, X_3</code></td>
</tr>
<tr>
<td>$X_2 = n$</td>
<td><code>put_node n/0, X_2</code></td>
</tr>
<tr>
<td>$X_3 = \text{myth}(X_4)$</td>
<td><code>put_node myth/2, X_3</code></td>
</tr>
<tr>
<td></td>
<td><code>put_arc X_3, 1, X_4</code></td>
</tr>
<tr>
<td></td>
<td><code>put_arc X_3, 2, X_4</code></td>
</tr>
<tr>
<td>$X_4 = \text{yes}$</td>
<td><code>put_node yes/0, X_4</code></td>
</tr>
</tbody>
</table>
Parsing with typed FS

Parsing with typed FS

Least upper bounds (partial)

<table>
<thead>
<tr>
<th></th>
<th>⊥</th>
<th>g</th>
<th>d</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>e</th>
<th>d1</th>
<th>d2</th>
<th>arity</th>
<th>appropriateness</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊥</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>2</td>
<td></td>
<td>f3:d, f1: ⊥</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>f3:d, f1: ⊥, f2: ⊥, f4: ⊥</td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A series of abstract machine language functions are generated: e.g. 
\[ \text{unify_type}[t1,t2] \] with one parameter \( addr \), where \( addr \) is the address of some TFS of type \( t2 \)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unify_type<a href="beast_addr">bird, beast</a></td>
<td>build_str(myth); build_self_ref; % since the LUB is ( \text{myth} ) unknown build_var(bool); % the value of ( f ) is a new structure return;</td>
</tr>
<tr>
<td>unify_type<a href="bird_addr">beast, bird</a></td>
<td>build_str(myth); build_ref 1; % ( \text{wings} ) is the first feature of ( \text{bird} ) build_var(bool); % the value of ( f ) is a new structure return;</td>
</tr>
</tbody>
</table>
Compiling a program

\[ \text{e.g. sign(myth(} \text{yes,}\text{)}, \text{n}) \]

\[ \Rightarrow \]

\[ X_1 = \text{sign(} \]
\[ \quad X_2, \]
\[ \quad X_3 \]
\[ X_2 = \text{n} \]
\[ X_3 = \text{myth(} \]
\[ \quad X_4, \]
\[ \quad X_4 \]
\[ X_4 = \text{yes} \]

\begin{align*}
&\text{get\_structure \text{sign/2, X1}} \\
&\text{unify\_variable X2} \\
&\text{unify\_variable X3} \\
&\text{get\_structure \text{n/0, X2}} \\
&\text{get\_structure \text{myth/2, X3}} \\
&\text{unify\_variable X4} \\
&\text{unify\_value X4} \\
&\text{get\_structure \text{yes/0, X4}}
\end{align*}