Efficient parsing with a large-scale unification-based grammar
Lessons from a multi-year, multi-team endeavour

Liviu Ciortuz
Department of Computer Science
University of Iasi, Romania

“ALEAR” Workshop, FP7 E.U. Project
Humbold Universität, Berlin, Germany
November 2008
PLAN

● **Fore-ground:**
  LinGO, the large-scale HPSG for English
  Key efficiency issues in parsing with large-scale unification grammars

● **Back-ground:**
  Unification-based grammars in the small
  OSF- and OSF-theory unification
  FS expansion
  Compilation of OSF- and OSF-theory unification
  **LIGHT:** The language and the system
  Two classes of feature paths: QC and GR
1. Fore-ground:

Based on

“Collaborative Language Engineering”
St. Oepen, D. Flickiger J. Tsujii, H. Uszkoreit (eds.), Center for Studies of Language and Information, Stanford, 2002


1.1. LinGO – the English Resource Grammar
EUBP version, www.delph-in.net

Short description: from “Efficiency in Unification-Based Parsing”, Natural Language Engineering, special issue, 6(1), 2000


• Size: un-expanded: 2.47MB, expanded: 40.34MB; 15059 types, 62 rules, 6897 lexical entries

• Developed within: \{ TDL / PAGE, [Kiefer, 1994], DFKI Type Description Language \\
LKB, [Copestake, 1999], CSLI Stanford Linguistic Knowledge Base \}

Applications: machine translation of spoken and edited language, email auto response, consumer opinion tracking, question answering
Systems running LinGO ERG

Parsing (Control)

FS Unifier (Logic)

interpreter

compiler / AM

compiler / AM

TDL / PAGE  
DFKI Saarbruecken

LKB  
Stanford Univ.

PET  
DFKI Saarbruecken

ALE  
[AMALIA]  
Haiffa Univ.

LiLFeS  
Tokyo Univ.

OSF  LIGHT  DFKI Saarbruecken
Some comparisons on performances in processing LinGO
reported by [Oepen, Callmeier, 2000]

<table>
<thead>
<tr>
<th>version</th>
<th>year</th>
<th>test suite</th>
<th>av. parsing time (sec.)</th>
<th>space (Kb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDL / PAGE</td>
<td>1996</td>
<td>‘tsnlp’</td>
<td>3.69</td>
<td>19016</td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘aged’</td>
<td>2.16</td>
<td>79093</td>
</tr>
<tr>
<td>PET</td>
<td>2000</td>
<td>‘tsnlp’</td>
<td>0.03</td>
<td>333</td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘aged’</td>
<td>0.14</td>
<td>1435</td>
</tr>
</tbody>
</table>
Performances of LIGHT
w.r.t. other systems processing LinGO

<table>
<thead>
<tr>
<th>system</th>
<th>optimization</th>
<th>average parsing time on CSLI test-suite (sec./sentence)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIGHT</td>
<td>quick-check</td>
<td>0.04</td>
</tr>
<tr>
<td>PET</td>
<td>quick-check</td>
<td>0.04</td>
</tr>
<tr>
<td>LiLFeS</td>
<td>CFG filter</td>
<td>0.06</td>
</tr>
<tr>
<td>LIGHT</td>
<td>without quick-check</td>
<td>0.07</td>
</tr>
<tr>
<td>PET</td>
<td>without quick-check</td>
<td>0.11</td>
</tr>
</tbody>
</table>
1.2 Key efficiency issues in parsing with large-scale (LinGO-like) unification-based grammars (I)

- choosing the right logical framework, and making your grammar a logical, declarative grammar
- grammar expansion: full vs. partial expansion
- sort lattice encoding
- FS unification: compilation
- FS sharing
- lexicon pre-compilation
Key efficiency issues in parsing with large-scale (LinGO-like) unification-based grammars (II)

- exploring grammar particularities:
  - quick check (QC) pre-unification filtering
    (generalised) grammar reduction (GR)

- two-step parsing
  - hyper-active parsing
  - ambiguity packing (based on FS subsumption)
  - grammar approximation: CFGs
2. Back-ground: PLAN

2.1 Unification-based grammars in the small

2.2 The Logics of feature structures
   2.2.1 OSF notions
   2.2.2 OSF- and OSF-theory unification
   2.2.3 The osf unify function
   2.2.4 The type-consistent OSF unifier
   2.2.5 Feature Structure expansion

2.3 Compiled OSF-unification

2.4 Compiled OSF-theory unification

2.5 LIGHT: the language and the system

2.6 Two classes of feature paths in unification grammars:
   quick check (QC) paths, and
   generalised reduction (GR) paths
2.1 Unification-based grammars in the small

Two sample feature structures OSF notation

\[
\text{vp} \\
\quad [ \text{ARGS} < \text{verb} \\
\qquad [ \text{HEAD} \#1, \\
\qquad \text{OBJECT} \#3:np, \\
\qquad \text{SUBJECT} \#2:\text{sign} ], \\
\qquad \#3 >, \\
\qquad \text{HEAD} \#1, \\
\qquad \text{SUBJECT} \#2 ]
\]

\[
\text{satisfy_HPSG_principles} \\
\quad [ \text{CAT} \#1, \\
\qquad \text{SUBCAT} \#2, \\
\qquad \text{HEAD top} \\
\qquad [ \text{CAT} \#1, \\
\qquad \text{SUBCAT} \#3|\#2 ], \\
\qquad \text{COMP top} \\
\qquad [ \text{CAT} \#3, \\
\qquad \text{SUBCAT nil} ] ]
\]
HPSG principles as feature constraints

• head principle:
  satisfy_HPSG_principles [HEAD.CAT = CAT]

• saturation principle:
  satisfy_HPSG_principles [COMP.SUBCAT = nil]

• subcategorization principle:
  satisfy_HPSG_principles [HEAD.SUBCAT = COMP.CAT | SUBCAT]
A sample sort hierarchy

start stringphrase_or_word categ diff_list list

phrase

satisfy_HPSG_principles

lh_phrase rh_phrase

det_le noun_leadjective_le verb_le

det noun adjective verb

categ_cons nil

categ_list

word

the mary girl nice thinks

john embarrassed pretty met

kisses laughed

met kissed

embarrasses

meets

is
An expanded feature structure... rewritten as a rule

lh_phrase
[ PHON list,
  CAT #1:categ,
  SUBCAT #2:categ_list,
  HEAD #4:phrase_or_word
    [ PHON list,
      CAT #1,
      SUBCAT #3|#2 ],
  COMP #5:phrase_or_word
    [ PHON list,
      CAT #3,
      SUBCAT nil ],
  ARGS <#4, #5> ]

lh_phrase
[ PHON list,
  CAT #1:categ,
  SUBCAT #2:categ_list,
  HEAD #4,
  COMP #5 ]
<- #4:phrase_or_word
  [ PHON list,
    CAT #1,
    SUBCAT #3|#2 ],
#5:phrase_or_word
  [ PHON list,
    CAT #3,
    SUBCAT nil ].
Tree representation of a feature structure
A simple typed-unification HPSG-like grammar

types:

start[ SUBCAT nil ]
cons
  [ FIRST top,
    REST list ]
diff_list
  [ FIRST_LIST list,
    REST_LIST list ]
categ_cons
  [ FIRST categ,
    REST categ_list ]
phrase_or_word
  [ PHON list,
    CAT categ,
    SUBCAT categ_list ]
phrase
  [ HEAD #1:phrase_or_word,
    COMP #2:phrase_or_word,
    ARGS cons ]
satisfy_HPSG_principles
  [ CAT #1,
    SUBCAT #2,
    HEAD top
      [ CAT #1,
        SUBCAT #3|#2 ],
    COMP top
      [ CAT #3,
        SUBCAT nil ] ]
det_le
  [ CAT det,
    SUBCAT nil ]
noun_le
  [ CAT noun ]
pnoun_le
  [ SUBCAT nil ]
cnoun_le
  [ SUBCAT <det> ]
adjective_le
  [ CAT adjective,
    SUBCAT nil ]
dverb_le
  [ CAT verb,
    SUBCAT <noun, noun> ]
tverb_le
  [ CAT verb,
    SUBCAT <noun, noun> ]

program: // rules

lh_phrase
  [ HEAD #1,
    COMP #2,
    ARGS <#1,#2> ]
rh_phrase
  [ HEAD #1,
    COMP #2,
    ARGS <#2,#1> ]
query: // lexical entries

the[ PHON <"the"> ]
girl[ PHON <"girl"> ]
john[ PHON <"john"> ]
mary[ PHON <"mary"> ]
nice[ PHON <"nice"> ]
embarrassed[ PHON <"embarrassed"> ]
nice[ PHON <"pretty"> ]
met[ PHON <"met"> ]
kissed[ PHON <"kissed"> ]
is[ PHON <"is">,
    CAT verb,
    SUBCAT <adjective, noun> ]
laughs[ PHON <"laughs"> ]
kisses[ PHON <"kisses"> ]
thinks[ PHON <"thinks">,
    CAT verb,
    SUBCAT <verb, noun> ]
meets[ PHON <"meets"> ]
embarrasses[ PHON <"embarrasses"> ]
A simple typed-unification grammar

sorts:

sign:top.
rule:sign.
np:rule.
vp:rule.
s:rule.
lex_entry:sign.
det:lex_entry.
noun:lex_entry.
verb:lex_entry.
the:det.
a:det.
cat:noun.
mouse:noun.
catches:verb.

query: // lexical entries

the
[ HEAD top
  [ TRANS top
    [ DETNESS + ]
    PHON < "the" > ]
  PHON < "a" > ]
cat
[ HEAD top
  [ AGR 3sing,
    TRANS top
    [ PRED cat ]
    PHON < "cat" > ]
mouse
[ HEAD top
  [ AGR 3sing,
    TRANS top
    [ PRED mouse ]
    PHON < "mouse" > ]
catches
[ HEAD top
  [ AGR 2:3sing,
    TENSE present,
    TRANS top
    [ ARG1 #3,
      ARG2 #1,
      PRED catches ]
    OBJECT sign
    [ HEAD top
      [ TRANS #1 ]
      PHON < "catches" >,
      SUBJECT sign
      [ HEAD top
        [ AGR #2,
          TRANS #3 ] ] ]

program: // rules

np
[ ARGS < det
  [ HEAD top
    [ TRANS #1 ]
    noun
    [ HEAD #2:top
      [ TRANS #1 ]
      KEY-ARG + ] >,
    HEAD #1 ]
  HEAD #2 ]

vp
[ ARGS < verb
  [ HEAD #1,
    OBJECT #3:np,
    SUBJECT #2:np,
    KEY-ARG + ]
  #3 >,
  HEAD #1,
  SUBJECT #2 ]

s
[ ARGS < #2:np,
  vp
  [ HEAD #1,
    SUBJECT #2,
    KEY-ARG + ] >,
  HEAD #1 ]

The context-free backbone of the above grammar

\[
\begin{align*}
np & \to \text{det} \ast \text{noun} \\
vp & \to \ast \text{verb} \ np \\
s & \to \ np \ast \ vp
\end{align*}
\]

\[
\begin{align*}
det & \to \text{the} \mid \text{a} \\
noun & \to \text{cat} \mid \text{mouse} \\
verb & \to \text{catches}
\end{align*}
\]
The cat catches a mouse
The final content of the chart when parsing

*The cat catches a mouse*

<table>
<thead>
<tr>
<th></th>
<th>syn. rule / lex. categ.</th>
<th>start - end</th>
<th>env</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>$s \rightarrow .np \ vp.$</td>
<td>0 – 5</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>$s \rightarrow np .vp.$</td>
<td>2 – 5</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>$vp \rightarrow .verb \ np.$</td>
<td>2 – 5</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>$np \rightarrow .det \ noun.$</td>
<td>3 – 5</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>$np \rightarrow det .noun.$</td>
<td>4 – 5</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>$vp \rightarrow .verb. \ np$</td>
<td>2 – 3</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>$np \rightarrow .det \ noun.$</td>
<td>0 – 2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>$np \rightarrow det .noun.$</td>
<td>1 – 2</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>det $\Rightarrow$ the</td>
<td>0 – 1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>noun $\Rightarrow$ cat</td>
<td>1 – 2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>verb $\Rightarrow$ catches</td>
<td>2 – 3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>det $\Rightarrow$ a</td>
<td>3 – 4</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>noun $\Rightarrow$ mouse</td>
<td>4 – 5</td>
<td></td>
</tr>
</tbody>
</table>
2.2 The Logics of feature structures

- First-order terms
  - First-order unification
  - WAM [Warren '83]

- OSF-terms
  - OSF unification
  - [Ait-Kaci et al. '93]

- Type- and order-consistent OSF-theories
  - OSF-theory unification
  - LIGHT AM

- Type-consistent OSF-theories

- Order-consistent OSF-theories
  - OSF-theory unification
  - rewriting rules [Ait-Kaci et al. '93]

- Well-typed FS
  - FS unification
  - [Carpenter, Qu, '95], AMALIA, LiLFeS
2.2.1 OSF notions

- $S$ – sorts, $F$ – features, $V$ – variables/coreferences
  $< S, \prec, \land >$ – sort signature

- Atomic constraints:
  - sort constraint: $X : s$
  - feature constraint: $s.f \Rightarrow t$
  - equation (inside FS): $X \equiv Y$

- sort hierarchy: lower semi-lattice over $S$

- OSF feature structure (OSF-term)

- the logical form associated to an OSF-term:
  $\psi \equiv s[f_1 \rightarrow \psi_1, ..., f_n \rightarrow \psi_n]$

  $\text{Form}(\psi, X) \equiv \exists X_1 \ldots \exists X_n((X.f_1 \equiv \text{Form}(\psi_1, X_1) \land ... \land X.f_n \equiv \text{Form}(\psi_n, X_n)) \leftarrow X : s)$

- FS subsumption
- FS unification
OSF notions (cont’d)

- **OSF-theory**: \(\{\Psi(s)\}_{s \in S}\) with \(\text{root}(\Psi(s)) = s\)
- **OSF-theory unification**: 
  \(\psi_1\) and \(\psi_2\) unify w.r.t. \(\{\Psi(s)\}_{s \in S}\) if \(\exists \psi\) such that 
  \(\psi \sqsubseteq \psi_1, \psi \sqsubseteq \psi_2,\) and \(\{\Psi(s)\}_{s \in S} \models \psi.\)
- **order-consistent OSF-theory**: \(\{\Psi(s)\}_{s \in S}\) such that 
  \(\Psi(s) \sqsubseteq \Psi(t)\) for any \(s \preceq t\)
- **type-consistent OSF-theory**: 
  for any non-atomic subterm \(\psi\) of a \(\Psi(t)\), 
  if the root sort of \(\psi\) is \(s\), then \(\psi \sqsubseteq \Psi(s)\)
2.2.2 OSF- and OSF-theory unification

Let us consider two OSF-terms and a sort signature in which $b \land c = d$ and the symbol $+$ is a subsort of the sort $bool$. We consider the OSF-theory made (uniquely) of

$$
\Psi(d) = d[ \text{FEAT2} \rightarrow + ].
$$

The $\text{glb}$ (i.e. OSF-term unification result) of $\psi_1$ and $\psi_2$

$$
\psi_1 = a[ \text{FEAT1} \rightarrow b ],
\psi_2 = a[ \text{FEAT1} \rightarrow c[ \text{FEAT2} \rightarrow bool ] ],
$$

is

$$
\psi_3 = a[ \text{FEAT1} \rightarrow d[ \text{FEAT2} \rightarrow bool ] ],
$$

while the $\{ \Psi(d) \}$ OSF-theory relative $\text{glb}$ (i.e. unification result) for $\psi_1$ and $\psi_2$ is

$$
\psi_4 = a[ \text{FEAT1} \rightarrow d[ \text{FEAT2} \rightarrow + ] ].
$$
Internal representation of the $vp$ feature structure
The effect of OSF-theory unification on $\psi_1$ and $\psi_2$

<table>
<thead>
<tr>
<th>cell_heap</th>
<th>frame_heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SORT</strong></td>
<td><strong>FTAB</strong></td>
</tr>
<tr>
<td>0</td>
<td>nil</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
</tr>
<tr>
<td>2</td>
<td>+</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
</tr>
<tr>
<td>6</td>
<td>c d</td>
</tr>
<tr>
<td>7</td>
<td>bool +</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.2.3 The osf_unify function

```java
boolean osf_unify( int a1, int a2 )
{
    boolean fail = FALSE;
    push_PDL( &PDL, a1 ); push_PDL( &PDL, a2 );
    while non_empty( &PDL ) ∧¬fail {
        d1 = deref( pop( &PDL ) ), d2 = deref( pop( &PDL ) );
        if d1 ≠ d2 {
            new_sort = heap[d1].SORT ∧ heap[d2].SORT;
            if new_sort = BOT
                fail = TRUE;
            else {
                bind_refine( d1, d2, new_sort )
                if deref( d1 ) = d2
                    carry_features( d1, d2 );
                else carry_features( d2, d1 );
            }
        }
    }
    return ¬fail;
}
```
Routines needed by the osf_unify function

bind_refine( int d1, int d2, sort s )
{
    heap[ d1 ].CREF = d2;
    heap[ d2 ].SORT = s;
}

carry_features( int d1, int d2 )
{
    FEAT_frame *frame1 = heap[ d1 ].FTAB, *frame2 = heap[ d2 ].FTAB;
    FHEAP_cell *feats1 = frame1 -> feats, *feats2 = frame2 -> feats;
    int feat, nf = frame1 -> nf;
    for (feat = 0; feat < nf; ++feat) {
        int f, f1 = feats1[ feat ].FEAT, v1 = feats1[ feat ].TERM, v2;
        if ((f = get_feature( d2, f1 )) ≠ FAIL) {
            v2 = feats2[ f ].TERM;
            push_PDL( &PDL, v2 );
            push_PDL( &PDL, v1 );
        } else add_feature( d2, f1, v1 );
    }
}
2.2.4 The type-consistent OSF unifier

boolean expansionCondition( int d1, int d2, sort s )
{
    if ((¬ isAtomicFS( d1 ) ∨ ¬ isAtomicFS( d2 )) ∧
        heap[ d1 ].SORT ≠ s ∧ heap[ d2 ].SORT ≠ s) ∨
    (isAtomicFS( d1 ) ∧ ¬ isAtomicFS( d2 ) ∧ heap[ d2 ].SORT ≠ s) ∨
    (¬ isAtomicFS( d1 ) ∧ isAtomicFS( d2 ) ∧ heap[ d1 ].SORT ≠ s)
        return TRUE;
    else return FALSE;
}

bind_refine( int d1, int d2, sort s )
{
    push_TRAIL( &TRAIL, d1, LINK, heap[ d1 ].CREF );
    heap[ d1 ].CREF = d2;
    if toBeChecked ≠ NULL ∧ expansionCondition( d1, d2, s )
        *toBeChecked = cons( d2, *toBeChecked ); }
    heap[ d2 ].SORT = s;
}
The type-consistent OSF unifier (cont’d)

```c
boolean check_osf_unify_result( int r, int_list *toBeChecked, int *representation )
{
    boolean result = TRUE;
    int_list *l;
    for (l = toBeChecked; result ∧ *l ≠ NIL; l = l -> next) {
        int k = l -> value; // take the 1st elem from *l
        int s = heap[ k ].SORT;
        int_list new_list = NIL;
        if osf_unify( representation[ s ], k, &new_list ) = -1
            result = FALSE;
        else
            append( toBeChecked, new_list );
    }
    return result;
}

boolean consistent_osf_unify( int i, int j, int *representation )
{
    int_list toBeChecked = NIL;
    return
        osf_unify( i, j, &toBeChecked ) ∧
        (toBeChecked = NIL ∨ check_osf_unify_result( h, &toBeChecked, representation ));
}
```
2.2.4 Feature Structure Expansion: An example

lh_phrase
[ PHON list,
  CAT #1:categ,
  SUBCAT #2:categ_list,
  HEAD #4:phrase_or_word
    [ PHON list,
      CAT #1,
      SUBCAT #3|#2 ],
  COMP #5:phrase_or_word
    [ PHON list,
      CAT #3,
      SUBCAT nil ],
  ARGS <#4, #5> ]
EXPANSION in Typed Feature Structure Grammars vs. Order- and Type-consistent OSF-theories

**computational effects:**

<table>
<thead>
<tr>
<th></th>
<th>grammar size (nodes)</th>
<th>tcpu (sec)</th>
<th>space (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>expansion</td>
<td>524,145</td>
<td>0.928</td>
<td>7,028</td>
</tr>
<tr>
<td>“partial” expansion</td>
<td>305,836</td>
<td>0.742</td>
<td>5,846</td>
</tr>
<tr>
<td>unfilling</td>
<td>171,195</td>
<td>0.584</td>
<td>4,683</td>
</tr>
</tbody>
</table>

*Note:* reproduced from [Callmeier, 2000], Natural Language Engineering
2.3 Compiled OSF-unification

The ‘query’ OSF abstract code for $\psi_1$ (left) and $\psi_2$ (right)

```
push_cell 0
set_sort 0, a
push_cell 1
set_feature 0, FEAT1, 1
set_sort 1, b

push_cell 0
set_sort 0, a
push_cell 1
set_feature 0, FEAT1, 1
set_sort 1, c
push_cell 2
set_feature 1, FEAT2, 2
set_sort 2, bool
```
Abstract ‘program’ code for the term $\psi_1$

R0: intersect_sort 0, a
    test_feature 0, FEAT1, 1, 1, W1, a
    intersect_sort 1, b

R1: goto W2;

W1: push_cell 1
    set_feature 0, FEAT1, 1
    set_sort 1, b

W2:
push_cell i:int \equiv \\
if i+Q \geq \text{MAX}_\text{HEAP} \lor H \geq \text{MAX}_\text{HEAP} \\\n\quad \text{error( "heap allocated size exceeded\n" );} \\
else \{ \\
\quad \text{heap}[H].\text{SORT} = \text{TOP}; \\
\quad \text{heap}[H].\text{FTAB} = \text{FTAB}_\text{DEF}_\text{VALUE}; \\
\quad \text{heap}[H].\text{CREF} = H; \\
\quad \text{setX}( i+Q, H++ ); \} \\
set_sort i:int, s:sort \equiv \\
heapp{X[i+Q]} .= s; \\
set_feature i:int, f:feat, j:int \equiv \\
\quad \text{int addr} = \text{deref( } X[i+Q] \text{ );} \\
\quad \text{FEAT\_frame *frame} = \text{heap[ addr ].FTAB} \\
\quad \text{push\_TRAIL( } \&\text{TRAIL, addr, FEAT,} \\
\quad \quad \quad \quad \text{(frame \neq \text{FTAB}_\text{DEF}_\text{VALUE} ? frame->nf : 0)} \); \\
\quad \text{add\_feature( } addr, f, X[j+Q] \text{ );}
WRITE abstract instructions in OSF AM (I)

\[
\text{intersect\_sort } i: \text{int}, s: \text{sort} \equiv \\
\text{int } \text{addr} = \text{deref( } X[ i+Q ] , p; \\
\text{sort } \text{new\_sort} = \text{glb( } s, \text{heap[ addr }.\text{SORT } ); \\
\text{if } \text{new\_sort} = \bot \\
\text{fail} = \text{TRUE}; \\
\text{else } \{ \\
\text{if } s \neq \text{new\_sort} \\
\text{push\_TRAIL( } &\text{TRAIL, addr, SORT, heap[ addr }.\text{SORT } ); \\
\text{heap[ addr }.\text{SORT} = \text{new\_sort}; \} \\
\]

\text{write\_test } \text{level:int}, \text{l:label} \equiv \\
\text{if } D \geq \text{level} \\
\text{goto } R_l; \]
WRITE abstract instructions in OSF AM (II)

test_feature i:int, f:feat, j:int, level:int, l:label ≡
    int addr = deref( X[i+Q] ), p;
    int k = get_feature( addr, f );
    if k ≠ FAIL
        X[j+Q] = heap[addr].FTAB.features[k].VAL;
    else
        { D = level; goto Wl; }

unify_feature i:int, f:feat, j:int ≡
    int addr = deref( X[i+Q] ), k;
    FEAT_frame *frame = heap[addr].FTAB;
    if (k = (get_feature( addr, f )) ≠ FAIL)
        fail = osf_unify( heap[addr].FTAB.feats[k].TERM, X[j+Q] );
    else {
        push_TRAIL( &TRAIL, addr, FEAT,
            (frame ≠ FTAB_DEF_VALUE ? frame->nf : 0) );
        add_feature( addr, f, X[j+Q] );
    }
2.4 Compiled OSF-theory unification
Augmented OSF AM
Abstract Instructions (I)
on-line expansion
and FS sharing stuff

bind_refine\((d1:int, d2:int, s:sort)\): boolean
begin
push( trail, d1, LINK, heap[d1].CREF );
heap[d1].CREF = d2;
if heap[d2].SORT \neq s then
push( trail, d2, SORT, heap[d2].SORT );
if expansionCondition( d1, d2, s ) then
if onLineExpansion then
begin
heap[d2].SORT = s;
return onLineID_expansion( s, d2 );
end
else
begin
if toBeChecked then
*toBeChecked = cons( d2, *toBeChecked );
heap[d2].SORT = s;
return TRUE;
end
else
return TRUE;
end
On-line expansion stuff

on_line_ID_expansion( s:sort, addr:int ):boolean
begin
  r = program_id( s );
  oldQ = Q;
  Q = addr;
  saveXregisters;
  push( programPDL, r );
  result = program( r );
  pop( programPDL );
  restoreXregisters;
  Q = oldQ;
end
Augmented OSF AM Abstract Instructions (II)

on-line expansion and FS sharing stuff

\[ \text{intersect\_sort} \ i : \text{int}, \ s : \text{sort} \equiv \]
\[
\begin{align*}
\text{begin} \\
\text{addr} &= \text{deref}( X[ i ] ); \\
\text{old\_sort} &= \text{heap}[ \text{addr} ].\text{SORT}; \\
\text{new\_sort} &= \text{glb}( s, \text{old\_sort} ); \\
\text{if new\_sort} &= \perp \text{ then} \\
\text{fail} &= \text{TRUE}; \\
\text{else} \\
\text{if old\_sort} &\neq s \text{ new\_sort then} \\
\text{begin} \\
\text{push}( \text{trail}, \text{addr}, \text{SORT}, \text{heap}[ \text{addr} ].\text{SORT} ); \\
\text{if NOT(isAtomicFS( addr )) then} \\
\text{begin} \\
\text{heap}[ \text{addr} ].\text{SORT} &= \text{new\_sort}; \\
r &= \text{program\_id}( \text{new\_sort} ); \\
\text{if NOT(addr = Q) AND} \\
r &= \text{programPDL.array}[ \text{programPDL.top-1} ] \text{ then} \\
fail &= \text{NOT(on\_line\_ID\_expansion( new\_sort, addr ))}; \\
\text{else} \text{ heap}[ \text{d2} ].\text{SORT} &= s; \\
\text{end} \\
\text{else} \text{ heap}[ \text{d2} ].\text{SORT} &= s; \\
\text{end} \\
\text{else} ; \\
\text{end} \\
\text{end} 
\end{align*}
\]
Example:

\[ \psi_2 = \text{a[ FEAT1 c[ FEAT2 bool ]]} \text{ on the heap} \]
\[ \psi_1 = \text{a[ FEAT1 b]} \text{ compiled as a program term:} \]

R0:intersect_sort X[0], a
    test_feature X[0], FEAT1, X[1], 1, W1, a
    intersect_sort X[1], b
R1:goto W2;
W1: push_cell X[1]
    set_feature X[0], FEAT1, X[1]
    set_sort X[1], b
W2:
augmented OSF AM
abstract instructions (III)

on-line expansion stuff

declare test_feature i, feat, j, level, label, sort
begin
    addr = deref( X[ i ] );
    f = get_feature( addr, feat );
    if f ≠ FAIL then
        X[ j ] = heap[ addr ].FTAB.features[ f ].TERM
    else
        if new_sort ≠ sort AND isAtomicFS( addr ) then begin
            new_sort = heap[ addr ].SORT;
            r = program_id( new_sort );
            if addr ≠ Q AND
                programPDL.array[ programPDL.top-1 ] = r then begin
                on_line_ID_expansion( new_sort, addr )
                F = get_feature( addr, feat );
                X[ j ] = heap[ addr ].FTAB.features[ f ].TERM;
            end
        else begin D = level; goto label; end
    end
else begin D = level; goto label; end
end
Example:

\[ \psi_1 = a[ \text{FEAT1} \ b ] \text{ on the heap} \]
\[ \psi_2 = a[ \text{FEAT1} \ c[ \text{FEAT2} \ \text{bool} ] ] \text{ comp. as prog. term:} \]

R0:intersect_sort X[0], a
    test_feature X[0], FEAT1, X[1], 1, W1, a
    intersect_sort X[1], c
    test_feature X[1], FEAT2, X[2], 2, W2, c
    R1:goto W3;

W1: push_cell X[1]
    set_feature X[0], FEAT1, X[1]
    set_sort X[1], c
W2: push_cell X[2]
    set_feature X[1], FEAT2, X[2]
    set_sort X[2], bool
W3:
2.5 LIGHT: the language and the system

Logic, Inheritance, Grammars, Heads and Types

LIGHT grammar:

an order- and type-consistent OSF-theory, with:

reserved sorts: sign, rule-sign, lexical-sign, start

reserved features: PHON, ARGS

all leaf rule-sign-descendants being rules:

\[ \psi_0 : \psi_1 \psi_2 \ldots \psi_n \ (n \geq 0) \text{ with} \]

root(\psi_0) \leq \text{rule-sign}, and root(\psi_0) leaf node in (S, \prec)

root(\psi_i) \leq \text{rule-sign} \lor \text{root}(\psi_i) \leq \text{lexical-sign}, \ i = 1, n

Remark: there are no predicate symbols
Inference-based parsing with \texttt{LIGHT} grammars

input: $<w_1 w_2 \ldots w_n>$
lexial item: $(\varepsilon, \psi', i-1, j)$, with $\psi.PHON = <w_i w_{i+1} \ldots w_j>$

- **Head-corner:**
  $(\sigma, \psi, i, j)$ a passive item,
  $\psi_0 := \psi_1 \ldots \psi_r$ with $\psi_k$ its head/key arg;
  if (there is) $\varphi = \text{glb( of } \psi_k, \psi \text{ )}$, with $\tau\psi_k = \text{glb}(\psi_k, \psi)$, then
  $(\tau\sigma, \psi_0 := \psi_1 \ldots \psi_k \ldots \psi_r, i, j)$ is an item, passive... or active...

- **Right complete:**
  $(\sigma, \psi_0 := \psi_1 \ldots \psi_p \ldots \psi_q \ldots \psi_r, i, j)$ an active item,
  a passive item, either $(\tau, \psi, j, k)$ or $(\tau, \psi := \psi_1' \ldots \psi_m', j, k)$;
  if exists $\text{glb}(\tau\psi, \sigma\psi_{q+1})$, and $\upsilon$ is the corr. matching subst., then
  $(\upsilon\sigma, \psi_0 := \psi_1 \ldots \psi_{p-1} \ldots \psi_q \ldots \psi_r, i, k)$ is an item ...

- **Left complete**

derivation parse: $(\sigma, \psi, 0, n)$, where $\psi$ is start-sorted
The abstract code for a binary rule
An overview of LIGHT system’s architecture
LIGHT VM

chart

VM program stack (agenda)

parsing VM instructions

LIGHT AM

heap

environments

trail

AM instructions FS unification

apply-rule
Instructions in LIGHT VM and LIGHT AM

<table>
<thead>
<tr>
<th>VM Instructions</th>
<th>AM Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>parsing</strong></td>
<td><strong>interface</strong></td>
</tr>
<tr>
<td>keyCorner</td>
<td>undo</td>
</tr>
<tr>
<td>directComplete</td>
<td>saveEnvironment</td>
</tr>
<tr>
<td>reverseComplete</td>
<td>restoreEnvironment</td>
</tr>
<tr>
<td>apply_rule</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>READ-stream</strong></td>
</tr>
<tr>
<td></td>
<td>push_cell</td>
</tr>
<tr>
<td></td>
<td>set_sort</td>
</tr>
<tr>
<td></td>
<td>set_feature</td>
</tr>
<tr>
<td></td>
<td>write_test</td>
</tr>
<tr>
<td></td>
<td><strong>WRITE-stream</strong></td>
</tr>
<tr>
<td></td>
<td>intersect_sort</td>
</tr>
<tr>
<td></td>
<td>test_feature</td>
</tr>
<tr>
<td></td>
<td>unify_feature</td>
</tr>
</tbody>
</table>
Parsing-oriented VM instructions: Basic specifications (I)

Legend: \( n \) denotes the current number of items on the chart; \( t \) is the current value of the top index for the unifier’s trail.

- **keyCorner** \( i, r \)
  1. apply the rule \( r \) (in ‘key’ mode) on passive chart item \( \#i \);
  2. if this rule application is successful, push on agenda UNDOandSAVE \( n \), and either PASSIVE \( i \) or directCOMPLETE \( n - 1, n \), according to the arity of the rule \( r \) (1, respectively 2).

- **directComplete** \( i, m \)
  1. find \( \#j \), the first (if any) passive item among \( \#(m - 1), \#(m - 2), ..., \#0 \), such that item \( \#j \) completes the (active) item \( \#i \);
  2. if there is \( \#j \) as stated above, then (in the place of the directCOMPLETE \( i, m \) program word) push directCOMPLETE \( i, j \);
     on top of it, push successively: UNDO \( j \), UNDOandSAVE \( n, t \), and PASSIVE \( j \).
Parsing-oriented VM instructions: Basic specifications (II)

- **reverseComplete** $i, m$
  1. find $j$, the first (if any) active item among $(m-1), (m-2), \ldots, 0$, such that the (passive) item $i$ completes the item $j$;
  2. if there is $j$ as stated above, then (in the place of the reverseCOMPLETE $i,m$ program word) push directCOMPLETE $i,j$; on top of it, push successively: UNDO $(j)$, UNDOandSAVE $n,t$, and PASSIVE $j$.

- **passive** $i$
  2. push on agenda reverseCOMPLETE $#i, i$, and for every rule $r$ having the key argument compatible with the chart item $#i$, push keyCORNER $r, i$. 
FS sharing-oriented VM instructions: Basic specifications

- **undo** \( t \)
  undo changes on the unifier’s heap, based on popping trail records down to the \( t \) value of the trail’s top index.

- **undoANDsave** \( e, t \)
do the same action as undo, after having saved those changes in the trailTrace field of the environment \( e \).
The VM \textit{parse} (control) procedure

\begin{verbatim}
parse( char **tokenizedInput )
{
    init_chart( tokenizedInput );
    for each (lexical) item on the chart (i = number_of_items-1, ..., 0)
        push_agenda( PASSIVE, i, 0 );
    while ¬ empty( agenda ) {
        agendaRecord AR = pop( agenda );
        switch (AR.type) {
            case PASSIVE:
                passive( AR.index ); break;
            case keyCORNER:
                keyCorner( AR.index, AR.arg ); break;
            case reverseCOMPLETE:
                reverseComplete( AR.index, AR.arg ); break;
            case directCOMPLETE:
                directComplete( AR.index, AR.arg ); break;
            case UNDOandSAVE:
                undoANDsave( AR.index, AR.arg ); break;
            default undo( AR.arg ); } % UNDO
    }
\end{verbatim}
The evolution of the VM program stack (agenda) when parsing

The cat catches a mouse
2.6 Two classes of feature paths: quick check (QC) paths, and generalised reduction (GR) paths

- **Problem** with large-scale typed-unification grammars:
  unification of (typed) FSs is a much time consuming operation w.r.t. parsing itself ($\approx 95\%$)

- **Evidence** towards eventually speeding up unification:
  - most of the unifications attempted during parsing fail ($\approx 90\%$), and
  - they fail on a limited number of paths ($\approx 7\%$) in the rule FSs!

- **On the contrary**...
  there seems to be (actually quite many!) feature paths that never lead to unification failure!
QC-paths vs GR-paths

Quick-Check (QC)  Generalised Reduction (GR)
63% (interp.), 42% (comp.)  23% speed-up factor
The Quick-Check pre-unification filter

\[
\text{if } \text{root-sort}(\psi.\pi) \land \text{root-sort}(\phi.\pi) = \bot \text{ then } \neg \text{unify}(\psi, \phi)
\]
Compiled Quick-Check

\[ QC_\pi(\psi) = \text{on-line} QC(\text{compiled} QC_\pi(\psi)) \]
Improvements to the Compiled Quick Check

• an advanced compilation phase
can eliminate much of the redundancy appearing in on-line computation of QC-path values

• the rule-sensitive application of the QC test:
  making the QC test order rule-dependent eliminates superfluous QC tests on certain rules

Effect: 9% additional speed up for (full) parsing with the LinGO grammar on the CSLI test suite.
Comparing the average parsing times for the GR-reduced form of LinGO on the CSLI test suite, using the LIGHT system:

- simply compiled QC
- vs.
- further compiled, rule-sensitive QC
Two complementary forms to the “basic” QC test (I)

Coreference-based Quick-Check

\[
\text{if } \psi.\pi_1 \sqsubseteq \psi.\pi_2 \text{ and } \text{root-sort}(\phi.\pi_1) \land \text{root-sort}(\phi.\pi_2) = \bot \text{ then } \neg \text{unify}(\psi, \phi)
\]
Two complementary forms to the “basic” QC test (II)

Type-checking-based Quick-Check

\[
\psi \pi \quad \Phi \\
\phi \pi \\
\Phi'
\]

if \( s = \text{root-sort}(\phi.\pi) \land \text{root-sort}(\psi.\pi) \), and
\( s \neq \bot \) but type-checking \( \psi.\pi \) with \( \Phi(s) \) fails,
then \( \neg \text{unify}(\psi, \phi) \)
The unresponsive LinGO!!

Coreference-based Quick-Check: Evaluation

In practice, on the LinGO grammar, using the CSLI test suite, the coreference-based Quick-Check is an effective filter for $\text{unify}(\psi, \phi)$,

Type-checking-based Quick-Check: Evaluation

It is a costly procedure, worth using only if

- type-checking causes very frequent failure
- and it is not “shadowed” by classical QC (on other paths)

A simplified but effective version of type-checking-based QC can be easily designed if type-checking with $\Phi(s)$ fails very frequently on very few (1,2) paths inside $\Phi(s)$. 
Generalised Reduction
Procedure A: simple, non-incremental

- **Input**: \( \mathcal{G} \), a typed-unification grammar; and \( \Theta \), a test suite
- **Output**: \( \mathcal{G}' \), a “reduced” form of \( \mathcal{G} \), yielding the same parsing results on \( \Theta \)
- **Procedure**:

\[
\text{for each rule } \Psi(r) \text{ in the grammar } \mathcal{G} \\
\quad \text{for each elementary feature constraint } \varphi \text{ in } \Psi(r) \\
\quad \quad \text{if removing } \varphi \text{ from } \Psi(r) \\
\quad \quad \quad \text{preserves the parsing (evaluation) results} \\
\quad \quad \quad \quad \text{for each sentence in the test-suite } \Theta \\
\quad \quad \text{then } \Psi(r) := \Psi(r) \setminus \{\varphi\};
\]
GR Procedure $A'$

a parameterized version of the GR procedure $A$

- Additional input (w.r.t procedure $A$):
  $\Pi$, a subset of the elementary feature constraints in the grammar’s rule FSs: $\Pi \subseteq \bigcup_{r \in G} \Psi(r)$, and $\Sigma \subseteq \Theta$

- Procedure:

  for each rule $\Psi(r)$ in the grammar $G$
  for each elementary feature constraint $\varphi \in \Psi(r) \cap \Pi$
  if removing $\varphi$ from $\Psi(r)$
    preserves the parsing (evaluation) results
    for each sentence in the subset $\Sigma \subseteq \Theta$
  then $\Psi(r) := \Psi(r) \setminus \{\varphi\}$;

- Note: $GR_A(G, \Theta) \equiv GR_{A'}(G, \Theta, \bigcup_{r \in G} \Psi(r)), \Theta)$
GR Procedure B

an incremental GR procedure

\[ i = 0, \ G_0 = \mathcal{G}, \ \Pi_0 = \bigcup_{r \in \mathcal{G}} \Psi(r); \]

**do**

1. apply the GR procedure A'(\mathcal{G}, \Theta, \Pi_i, \Sigma_i = \{s_i\}),
   where \(s_i\) is a sentence chosen from \(\Theta\);
   let \(G_{i+1}\) be the result;

2. eliminate from \(\Theta\) the sentences for which
   \(G_{i+1}\) provides the same parsing results as \(\mathcal{G}\), and
   take \(\Pi_{i+1} = \Pi_0 \setminus \{\bigcup_{r \in G_i} \Psi_i(r)\}\), where
   \(\Psi_i(r)\) denotes the FS associated to rule \(r\) in \(G_i\)

3. \(i = i + 1;\)

**until** \(\Theta = \emptyset\)
On the incremental nature of GR procedure B

- as sets of elementary constraints:
  \[ \Psi_0(r) \supseteq \Psi_1(r) \]
  \[ \Psi_0(r) \supseteq \ldots \supseteq \Psi_{i+1}(r) \supseteq \Psi_i(r) \supseteq \ldots \supseteq \Psi_2 \supseteq \Psi_1 \]
  \[ \Pi_0 \subseteq \Pi_1(r) \]
  \[ \Pi_0 \subseteq \ldots \subseteq \Pi_{i+1} \subseteq \Pi_i \subseteq \ldots \subseteq \Pi_2(r) \subseteq \Pi_1(r) \]

- as logical models:
  \[ G_0 \models \ldots \models G_{i+1} \models G_i \models \ldots \models G_2 \models G_1 \]

- as parsing results:
  \[ G_{i+1}(\bigcup_{j=1}^{i} \Sigma_j) = G_0(\bigcup_{j=1}^{i} \Sigma_j) \]

Note: \( \Sigma_i \) can be thought of as \( \{s_i\} \) extended with the sentences which were eliminated from \( \Theta \) at step 2 in GR_B following the obtention of \( G_{i+1} \)
Improvements to the GR procedure B

1. sort the test suite $\Theta$ on the number of unification failures per each sentence, in decreasing order; therefore get a “heavy”, very effective reduction first ($G_1$)

2. lazy elimination on sentences from $\Theta$:
   at step 2 in GR$_B$, eliminate only(!) from the beginning of $\Theta$ those sentences which are correctly parsed by $G_{i+1}$

Note: 1. & 2. also contribute to reducing the effect of resource exhaustion which may appear due to grammar over-reduction
Improvements to the GR procedure B (Cont’d)

3. at step 1 in GR$_{A'}$ (called at step 1 in GR$_B$), consider as candidates for elementary feature constraints only the rules which were involved in the parsing of the sentence $s_i$, in case it did not cause resource exhaustion.

4. halving the way up/down the rule FS
   if reduction succeeds for an elementary feature constraint which is “terminal” in the tree representing a rule’s FS, try to do a more extensive reduction/pruning:
   (a) check whether the feature constraint which is at the halfway distance from the rule’s (LHS/arg) root can also be eliminated;
   (b) continue upwards/downwards on the feature path if this check was successful/unsuccessful.
GR Procedure B: grammar reduction rate vs. CPU time consumption on LinGO on the CSLI test suite
The GR effect on the LinGO grammar: parsing with LIGHT on the CSLI test-suite

<table>
<thead>
<tr>
<th>GR procedure</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>FC reduction rate: average non-GR vs. all paths in r.arg.</td>
<td>58.92% 260/494 (52.64%)</td>
<td>56.64% 176/494 (35.62%)</td>
</tr>
<tr>
<td>average parsing time, in msec. using full-form rule FSs</td>
<td></td>
<td>21.617</td>
</tr>
<tr>
<td>1-step parsing (reduction %)</td>
<td>16.662 (22.24%)</td>
<td>16.736 (22.07%)</td>
</tr>
<tr>
<td>2-step parsing (red. %)</td>
<td>18.657 (13.12%)</td>
<td>18.427 (14.20%)</td>
</tr>
</tbody>
</table>
The GR effect on the LinGO grammar: memory usage reduction for LIGHT AM

<table>
<thead>
<tr>
<th></th>
<th>full-form rules</th>
<th>GR-restricted rules</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1-step parsing</td>
</tr>
<tr>
<td>heap cells</td>
<td>101614</td>
<td>38320 (62.29%)</td>
</tr>
<tr>
<td>feature frames</td>
<td>60303</td>
<td>30370 (49.64%)</td>
</tr>
</tbody>
</table>
3. Personal conclusions/opinions:
Killers of the efficiency-oriented work in unification-based parsing

- insufficient/bad/hidden/no communication
- non-integrative view on parsing, both scientifically and humanly
- use of few, biased grammars
- un-realising that certain optimisations that worked well for other domains don’t have the same effect on our grammars (e.g. feature indexing inside FSs, advanced forms of QC filtering, FS sharing, look-up tables for variables – feature paths values etc.)
Other/Future Work on LIGHT

- Inductive-based grammar learning with LIGHT (GS)
- Transforming LIGHT into an engineering platform to implement other unification-based grammars, e.g. Fluid Construction Grammars (L. Steels, 2008)