

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

Humboldt 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

- L. Ciortuz. “LIGHT – a constraint language and compiler system for typed-unification grammars.” In LNAI vol. 2479, M. Jarke, J. Köhler, G. Lakemeyer (eds.), Springer-Verlag, 2002, pp. 3–17.
- L. Ciortuz. *On two classes of feature paths in large-scale unification grammars.* In *New Developments in Parsing Technologies*, Hurry Bunt, Giorgio Satta, John Carroll (eds.), Kluwer Academic Publishers, 2004, pp. 203–227.



1.1. LinGO – the English Resource Grammar

EUBP version, www.delph-in.net

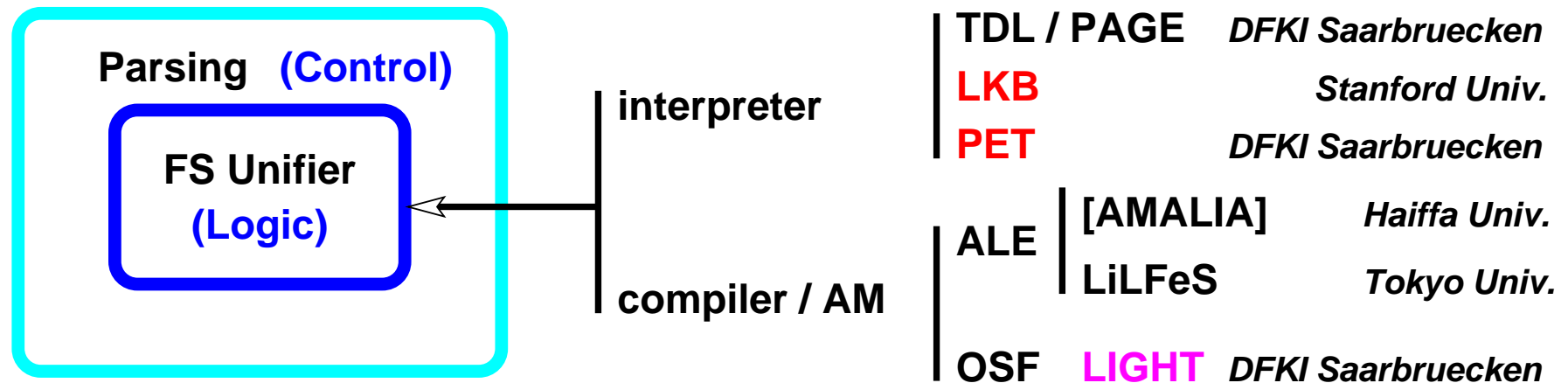
Short description: from “Efficiency in Unification-Based Parsing”,
Natural Language Engineering, special issue, 6(1), 2000

- **Support theory:** HPSG — Head-driven Phrase Structure Grammar
[Pollard and Sag, 1987, 1994]
- **Size:** un-expanded: 2.47MB, expanded: 40.34MB;
15059 types, 62 rules, 6897 lexical extrics
- **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



Some comparisons on performances in processing LinGO

reported by [Oepen, Callmeier, 2000]

<i>version</i>	<i>year</i>	<i>test suite</i>	<i>av. parsing time (sec.)</i>	<i>space (Kb)</i>
TDL / PAGE	1996	'tsnlp'	3.69	19016
		'aged'	2.16	79093
PET	2000	'tsnlp'	0.03	333
		'aged'	0.14	1435

**Performances of LIGHT
w.r.t. other systems processing LinGO**

<i>system</i>	<i>optimization</i>	<i>average parsing time on CSLI test-suite (sec./sentence)</i>
LIGHT	quick-check	0.04
PET	quick-check	0.04
LiLFeS	CFG filter	0.06
LIGHT	without quick-check	0.07
PET	without quick-check	0.11

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 ckeck (QC) paths, and
generalised reduction (GR) paths

2.1 Unification-based grammars in the small

Two sample feature structures OSF notation

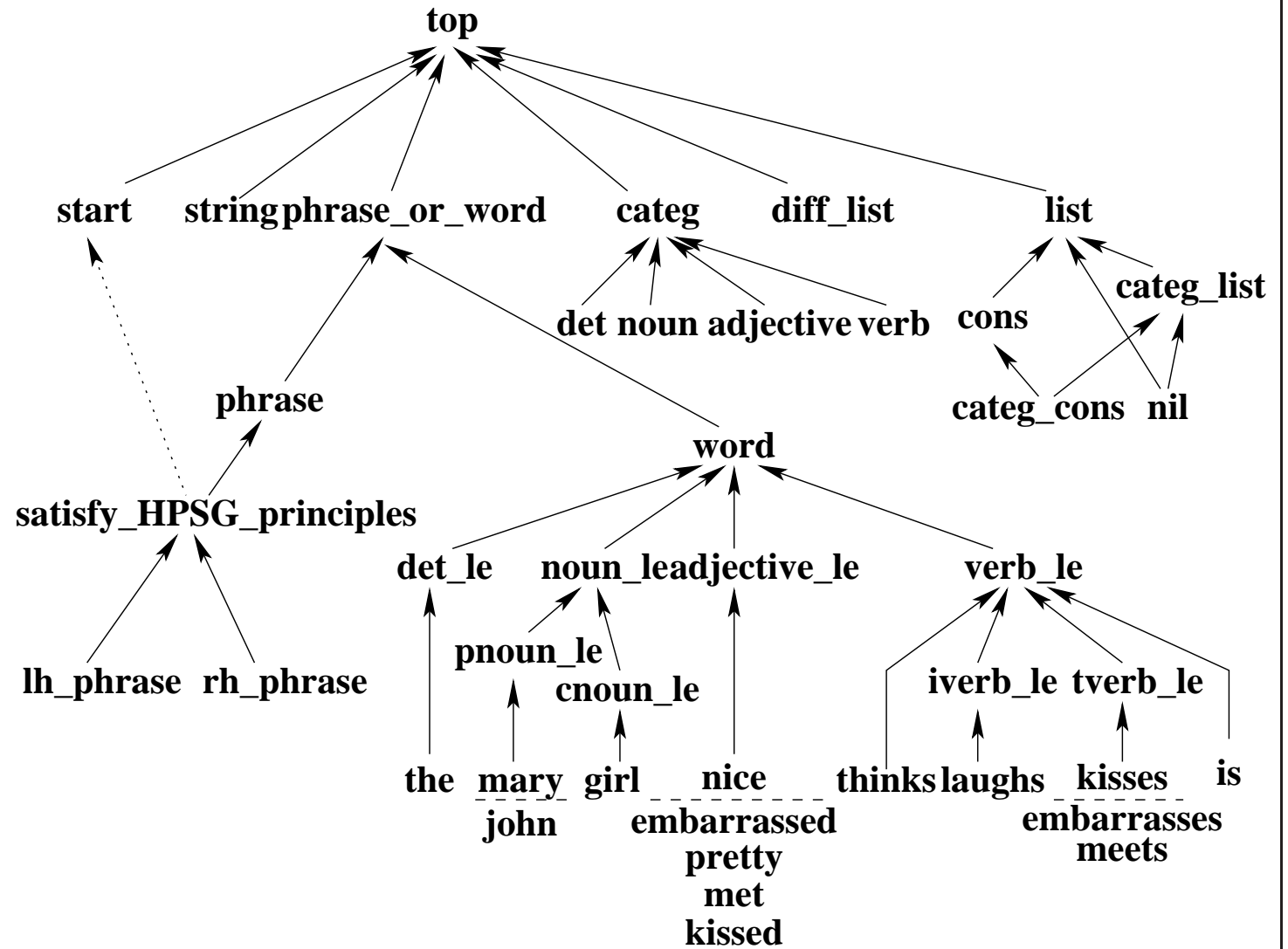
```
vp
[ ARGS      < verb
  [ HEAD     #1,
    OBJECT   #3:np,
    SUBJECT  #2:sign ],
  #3 >,
  HEAD      #1,
  SUBJECT   #2 ]
```

```
satisfy_HPSG_principles
[ CAT       #1,
  SUBCAT    #2,
  HEAD      top
    [ CAT #1,
      SUBCAT #3|#2 ],
  COMP      top
    [ CAT #3,
      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

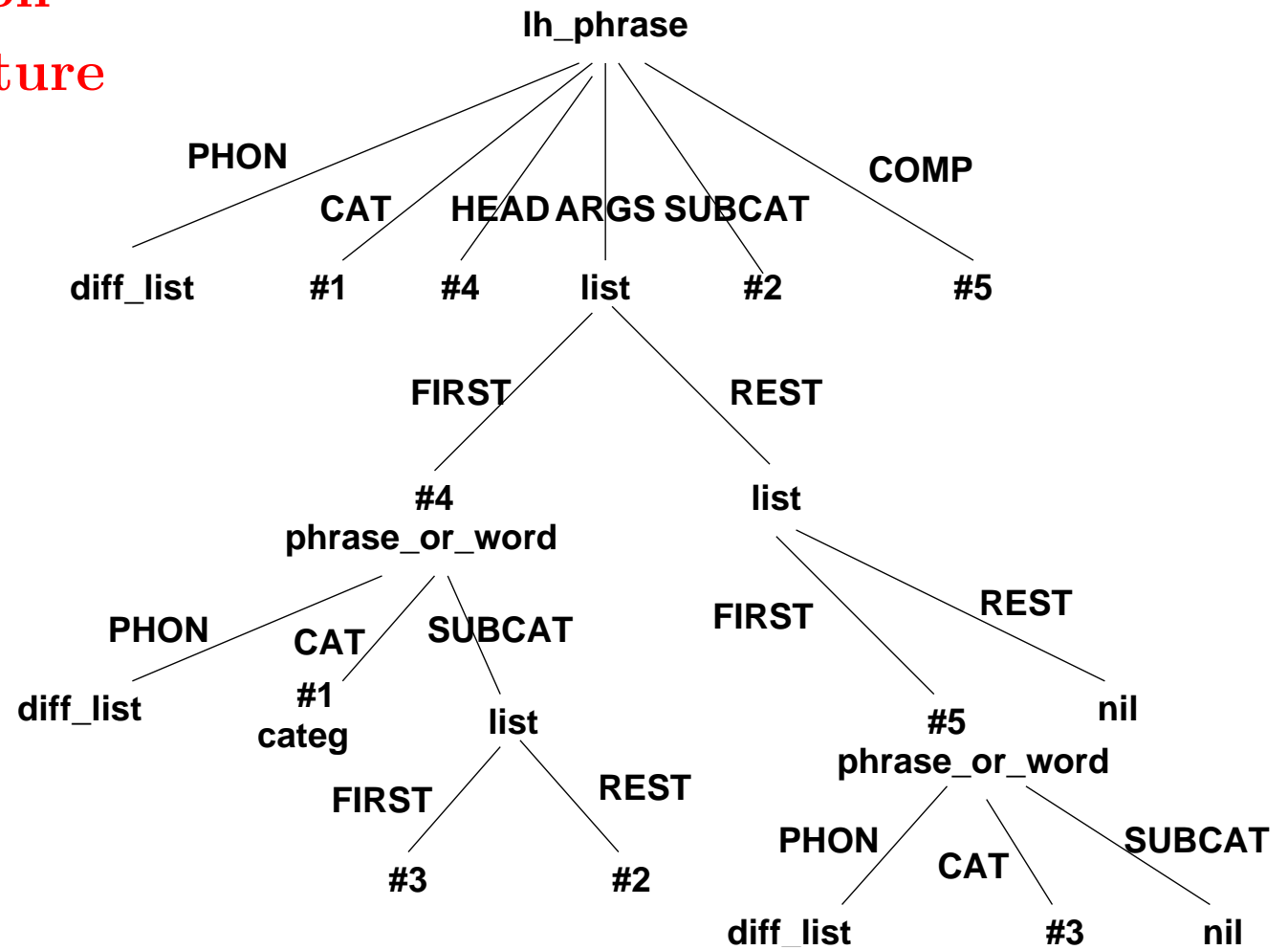


An expanded feature structure... rewritten as a rule

```
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 ]
iverb_le
[ CAT verb,
  SUBCAT <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"> ]
pretty[ 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.

types:

3sing
[ NR   sing,
  PERS third ]

program: // rules

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

query: // lexical entries

the
[ HEAD top
  [ TRANS top
    [ DETNESS + ] ],
  PHON < "the" > ]

a
[ HEAD top
  [ TRANS top
    [ DETNESS - ] ],
  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 ] ] ]

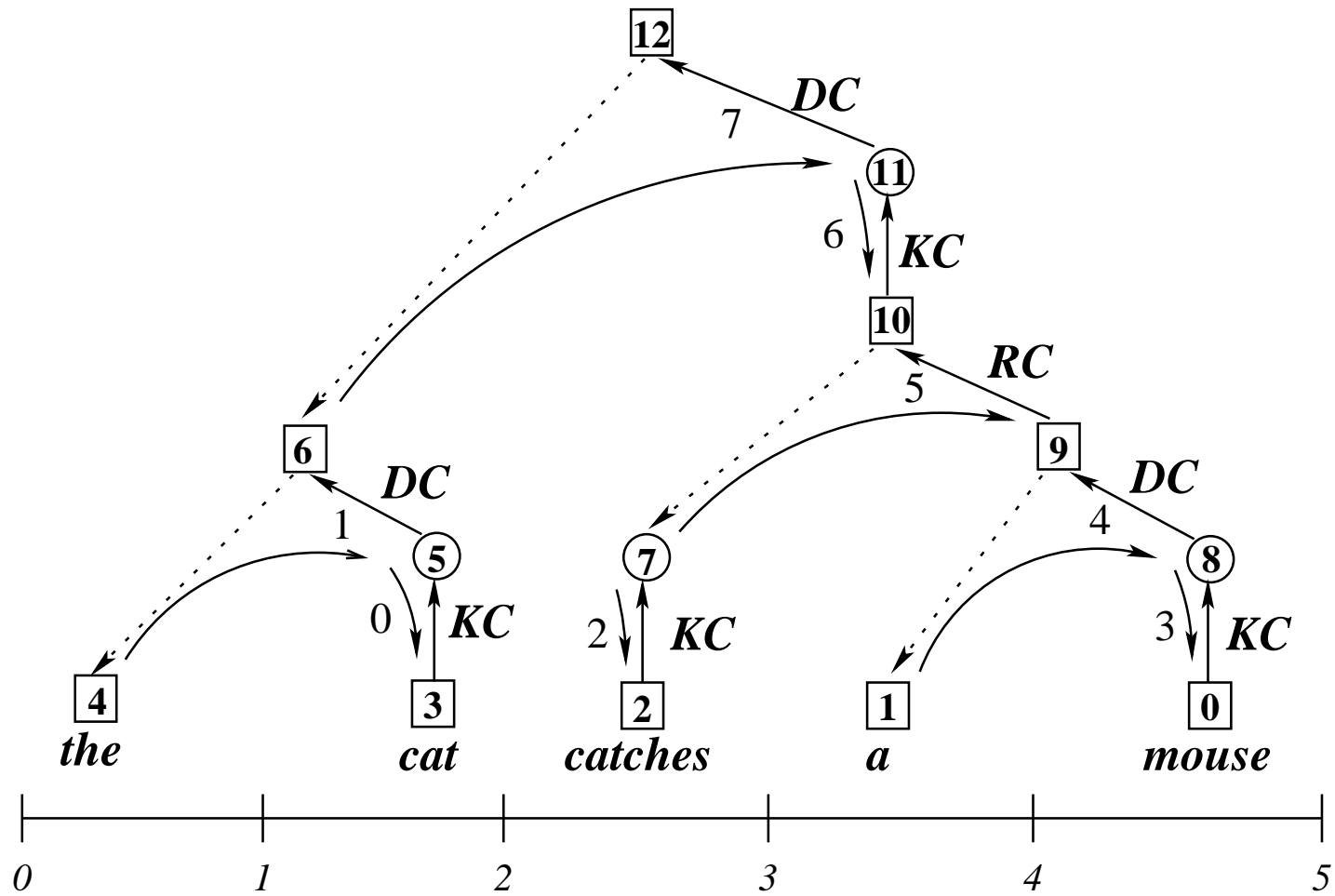
```

The context-free backbone of the above grammar

np → *det* **noun*
vp → **verb* *np*
s → *np* **vp*

det ⇒ *the* | *a*
noun ⇒ *cat* | *mouse*
verb ⇒ *catches*

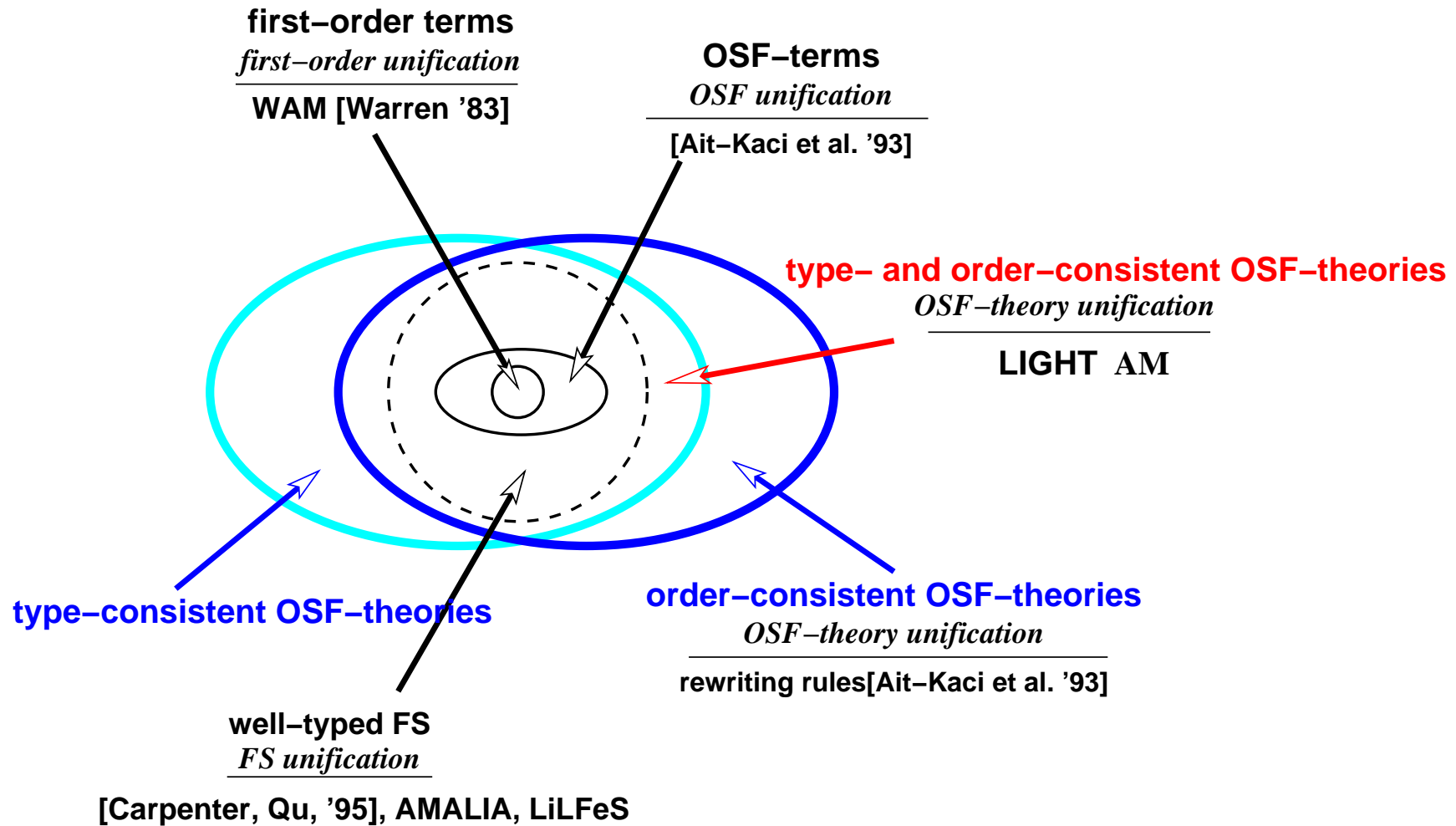
Parsing *The cat catches a mouse*



The final content of the
chart when parsing
The cat catches a mouse

	syn. rule / lex. categ.	start - end	env
12	$s \rightarrow .np\ vp.$	0 - 5	7
11	$s \rightarrow np\ .vp.$	2 - 5	6
10	$vp \rightarrow .verb\ np.$	2 - 5	5
9	$np \rightarrow .det\ noun.$	3 - 5	4
8	$np \rightarrow det\ .noun.$	4 - 5	3
7	$vp \rightarrow .verb.\ np$	2 - 3	2
6	$np \rightarrow .det\ noun.$	0 - 2	1
5	$np \rightarrow det\ .noun.$	1 - 2	0
4	$det \Rightarrow the$	0 - 1	
3	$noun \Rightarrow cat$	1 - 2	
2	$verb \Rightarrow catches$	2 - 3	
1	$det \Rightarrow a$	3 - 4	
0	$noun \Rightarrow mouse$	4 - 5	

2.2 The Logics of feature structures



2.2.1 OSF notions

- \mathcal{S} – sorts, \mathcal{F} – features, \mathcal{V} – variables/coreferences
 $\langle \mathcal{S}, \prec, \wedge \rangle$ – sort *signature*
- Atomic constraints:
 - sort constraint: $X : s$
 - feature constraint: $s.f \Rightarrow t$
 - equation (inside FS): $X \doteq Y$
- sort hierarchy: lower semi-lattice over \mathcal{S}
- OSF feature structure (OSF-term)
- the *logical form* associated to an OSF-term:

$$\psi \equiv s[f_1 \rightarrow \psi_1, \dots, f_n \rightarrow \psi_n]$$

$$\mathbf{Form}(\psi, X) \equiv \exists X_1 \dots \exists X_n ((X.f_1 \doteq \mathbf{Form}(\psi_1, X_1) \wedge \dots \wedge X.f_n \doteq \mathbf{Form}(\psi_n, X_n)) \leftarrow X : s)$$
- FS subsumption
- FS unification

OSF notions (cont'd)

- **OSF-theory:** $\{\Psi(s)\}_{s \in \mathcal{S}}$ with $\text{root}(\Psi(s)) = s$
- **OSF-theory unification:**

ψ_1 and ψ_2 unify w.r.t. $\{\Psi(s)\}_{s \in \mathcal{S}}$ if $\exists \psi$ such that $\psi \sqsubseteq \psi_1, \psi \sqsubseteq \psi_2$, and $\{\Psi(s)\}_{s \in \mathcal{S}} \models \psi$.
- **order-consistent OSF-theory:** $\{\Psi(s)\}_{s \in \mathcal{S}}$ such that $\Psi(s) \sqsubseteq \Psi(t)$ for any $s \preceq t$
- **type-consistent OSF-theory:**

for any non-atomic subterm ψ of a $\Psi(t)$,
if the root sort of ψ 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 $\mathbf{b} \wedge \mathbf{c} = \mathbf{d}$ and the symbol $+$ is a subsort of the sort *bool*. We consider the OSF-theory made (uniquely) of

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

The *glb* (i.e. OSF-term unification result) of ψ_1 and ψ_2

$$\psi_1 = \mathbf{a}[\text{FEAT1} \rightarrow \mathbf{b}],$$

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

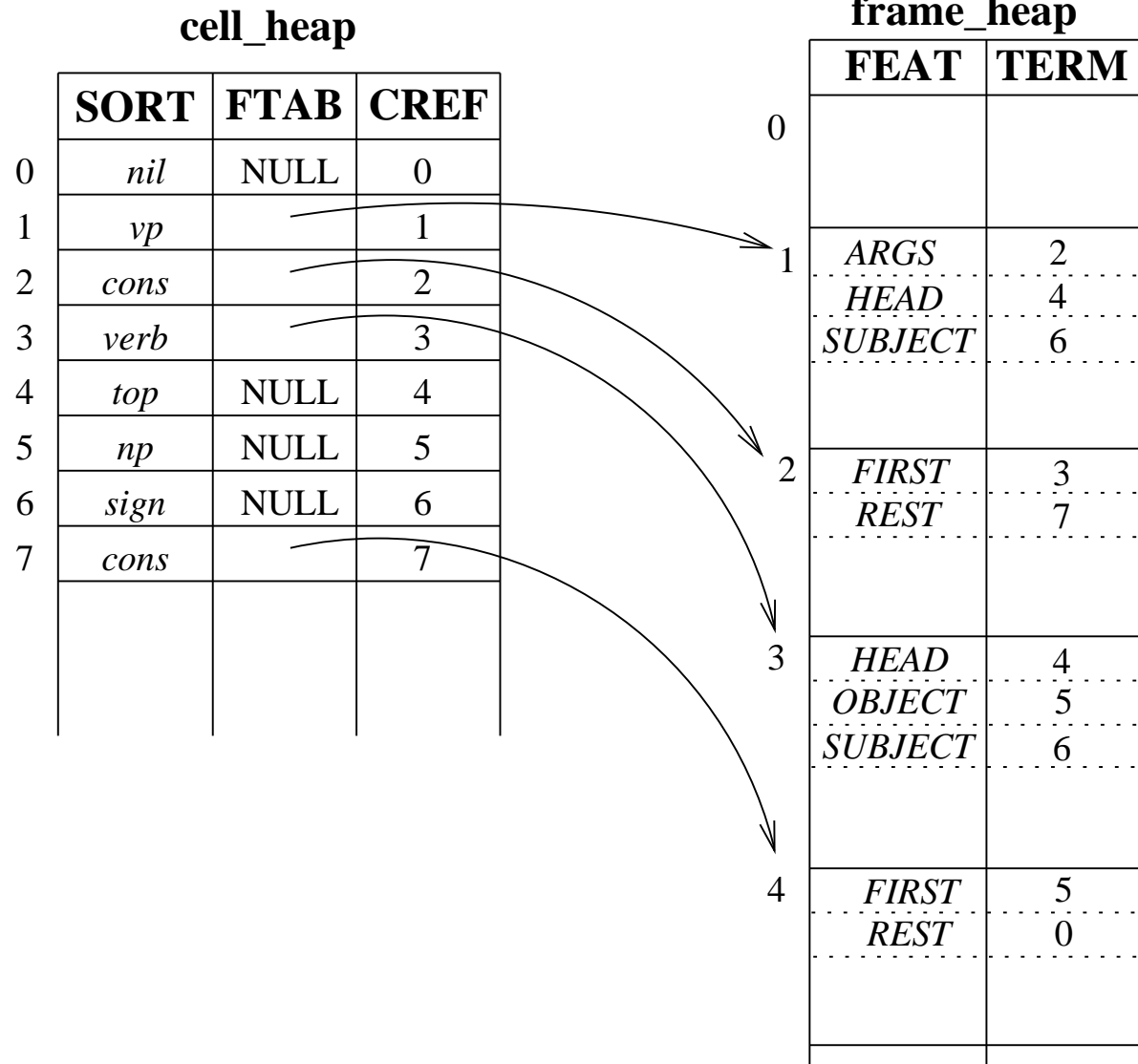
is

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

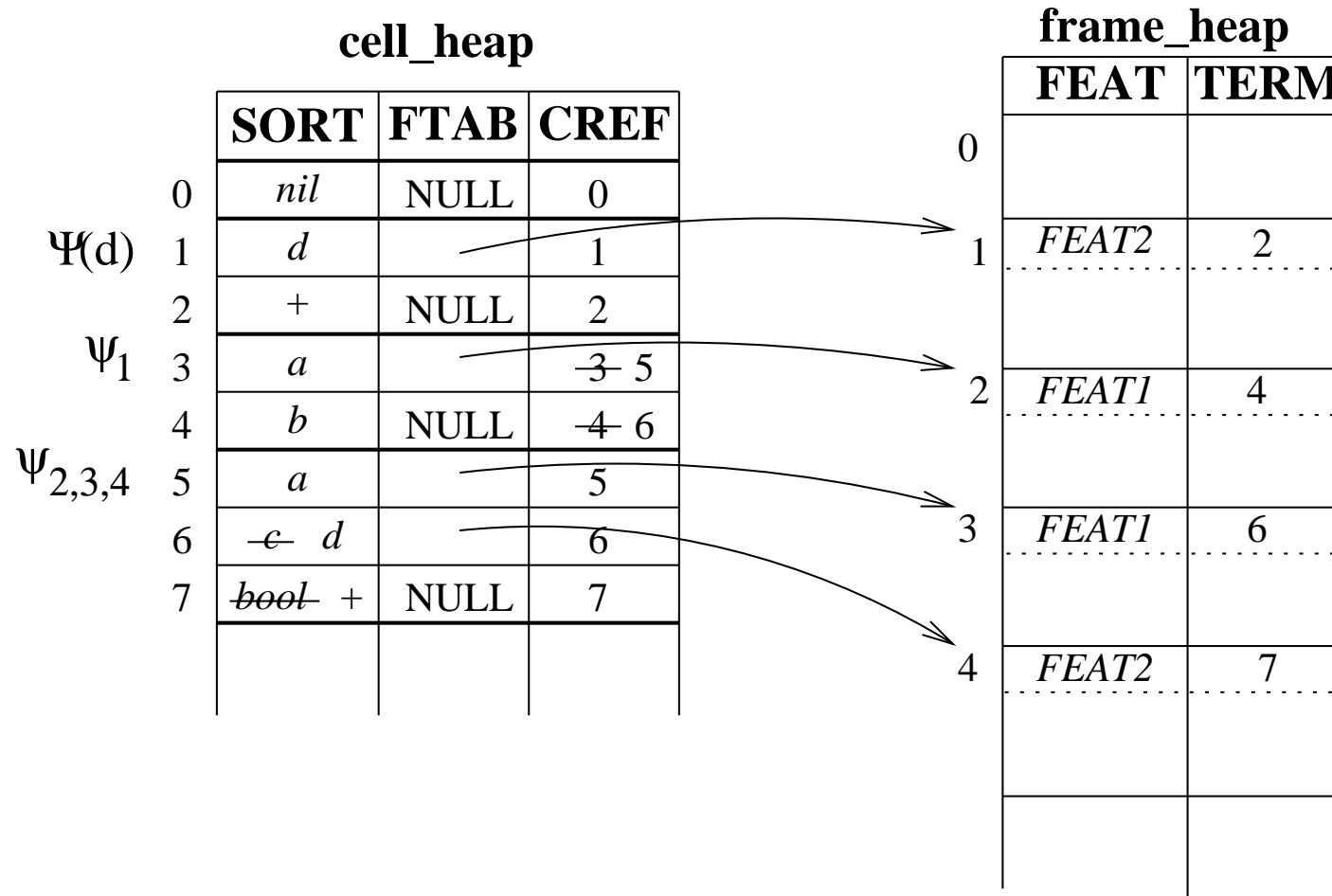
while the $\{\Psi(\mathbf{d})\}$ OSF-theory relative *glb* (i.e. unification result) for ψ_1 and ψ_2 is

$$\psi_4 = \mathbf{a}[\text{FEAT1} \rightarrow \mathbf{d}[\text{FEAT2} \rightarrow +]].$$

Internal representation of the *vp* feature structure



The effect of OSF-theory unification on ψ_1 and ψ_2



2.2.3 The osf_unify function

```

boolean osf_unify( int a1, int a2 )
{
    boolean fail = FALSE;
    push_PDL( &PDL, a1 ); push_PDL( &PDL, a2 );
    while non_empty( &PDL )  $\wedge$   $\neg$ fail {
        d1 = deref( pop( &PDL ) ), d2 = deref( pop( &PDL ) );
        if d1  $\neq$  d2 {
            new_sort = heap[d1].SORT  $\wedge$  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  $\neg$ 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 (( $\neg$  isAtomicFS( d1 )  $\vee$   $\neg$  isAtomicFS( d2 ))  $\wedge$ 
      heap[ d1 ].SORT  $\neq$  s  $\wedge$  heap[ d2 ].SORT  $\neq$  s)  $\vee$ 
      (isAtomicFS( d1 )  $\wedge$   $\neg$  isAtomicFS( d2 )  $\wedge$  heap[ d2 ].SORT  $\neq$  s)  $\vee$ 
      ( $\neg$  isAtomicFS( d1 )  $\wedge$  isAtomicFS( d2 )  $\wedge$  heap[ d1 ].SORT  $\neq$  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  $\neq$  NULL  $\wedge$  expansionCondition( d1, d2, s )
    *toBeChecked = cons( d2, *toBeChecked ); }
  heap[ d2 ].SORT = s;
}

```

The type-consistent OSF unifier (cont'd)

```

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:

	grammar size (nodes)	tcpu (sec)	space (KB)
expansion	524,145	0.928	7,028
“partial” expansion	305,836	0.742	5,846
unfilling	171,195	0.584	4,683

Note: reproduced from [Callmeier, 2000], Natural Language Engineering

2.3 Compiled OSF-unification

The ‘query’ OSF abstract code
for ψ_1 (left) and ψ_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 ψ_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:
```

READ abstract instructions in OSF AM

```

push_cell i:int ≡
  if i+Q ≥ MAX_HEAP ∨ H ≥ MAX_HEAP
    error( "heap allocated size exceeded\n" );
  else {
    heap[ H ].SORT = TOP;
    heap[ H ].FTAB = FTAB_DEF_VALUE;
    heap[ H ].CREF = H;
    setX( i+Q, H++ ); }

set_sort i:int, s:sort ≡
  heap[ X[ i+Q ] ].SORT = s;

set_feature i:int, f:feat, j:int ≡
  int addr = deref( X[ i+Q ] );
  FEAT_frame *frame = heap[ addr ].FTAB
  push_TRAIL( &TRAIL, addr, FEAT,
              (frame ≠ FTAB_DEF_VALUE ? frame->nf : 0) );
  add_feature( addr, f, X[ j+Q ] );

```

WRITE abstract instructions in OSF AM (I)

```
intersect_sort i:int, s:sort ≡
  int addr = deref( X[ i+Q ] ), p;
  sort new_sort = glb( s, heap[ addr ].SORT );
  if new_sort = ⊥
    fail = TRUE;
  else {
    if s ≠ new_sort
      push_TRAIL( &TRAIL, addr, SORT, heap[ addr ].SORT );
    heap[ addr ].SORT = new_sort; }

write_test level:int, l:label ≡
  if D ≥ level
    goto Rl;
```

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 )
begin
  heap[ d1 ].CREF = d2;
  heap[ d2 ].SORT = s;
end

```

```

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 ≠ s then
    push( trail, d2, SORT, heap[ d2 ].SORT );
  if expansionCondition( d1, d2, s ) then
    if onLineExpansion then
      begin
        heap[ d2 ].SORT = s;
        return on_line_ID_expansion( s, d2 );
      end
    else
      begin
        if toBeChecked then
          *toBeChecked = cons( d2, *toBeChecked );
          heap[ d2 ].SORT = s;
          return TRUE;
        end
      end
    else
      return TRUE;
    end
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

```

intersect_sort i:int, s:sort ≡
  begin
    addr = deref( X[ i ] );
    old_sort = heap[ addr ].SORT;
    new_sort = glb( s, old_sort );
    if new_sort = ⊥ then
      fail = TRUE;
    else
      if old_sort ≠ s new_sort then
        begin
          push( trail, addr, SORT, heap[ addr ].SORT );
          if NOT(isAtomicFS( addr )) then
            begin
              heap[ addr ].SORT = new_sort;
              r = program_id( new_sort );
              if NOT(addr = Q) AND
                r = programPDL.array[ programPDL.top-1 ] then
                fail = NOT(on_line_ID_expansion( new_sort, addr ));
              else heap[ d2 ].SORT = s;
            end
          else heap[ d2 ].SORT = s;
        end
      end
    else ;
  end
end

```

Example:

$\psi_2 = a[\text{FEAT1} \quad c[\text{FEAT2} \quad \textit{bool}]]$ on the heap
 $\psi_1 = a[\text{FEAT1} \quad b]$ 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:

```

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
  end
end

```

Augmented OSF AM
Abstract Instructions (III)
on-line expansion stuff

Example:

$\psi_1 = a[\text{FEAT1} \quad b]$ on the heap

$\psi_2 = a[\text{FEAT1} \quad c[\text{FEAT2} \quad \textit{bool}]]$ 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*:

$\psi_0 :- \psi_1 \psi_2 \dots \psi_n$ ($n \geq 0$) with

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

$root(\psi_i) \preceq rule\text{-}sign$ or $root(\psi_i) \preceq lexical\text{-}sign$, $i = \overline{1, n}$

Remark: there are no predicate symbols

Inference-based parsing with LIGHT grammars

input: $\langle w_1 w_2 \dots w_n \rangle$

lexical item: $(\epsilon, \psi', i-1, j)$, with $\psi.\text{PHON} = \langle w_i w_{i+1} \dots w_j \rangle$

- **Head-corner:**

(σ, ψ, i, j) a passive item,

$\psi_0 :- \psi_1 \dots \psi_r$ with ψ_k its head/key arg;

if (there is) $\varphi = \text{glb}(\text{ of } \psi_k, \psi)$, with $\tau\psi_k = \text{glb}(\psi_k, \psi)$, then

$(\tau\sigma, \psi_0 :- \psi_1 \dots \psi_k \dots \psi_r, i, j)$ is an item, passive... or active... .

- **Right complete:**

$(\sigma, \psi_0 :- \psi_1 \dots \psi_p \dots \psi_q \dots \psi_r, i, j)$ an active item,

a passive item, either (τ, ψ, j, k) or $(\tau, \psi :- \psi'_1 \dots \psi'_m, j, k)$;

if exists $\text{glb}(\tau\psi, \sigma\psi_{q+1})$, and v is the corr. matching subst., then

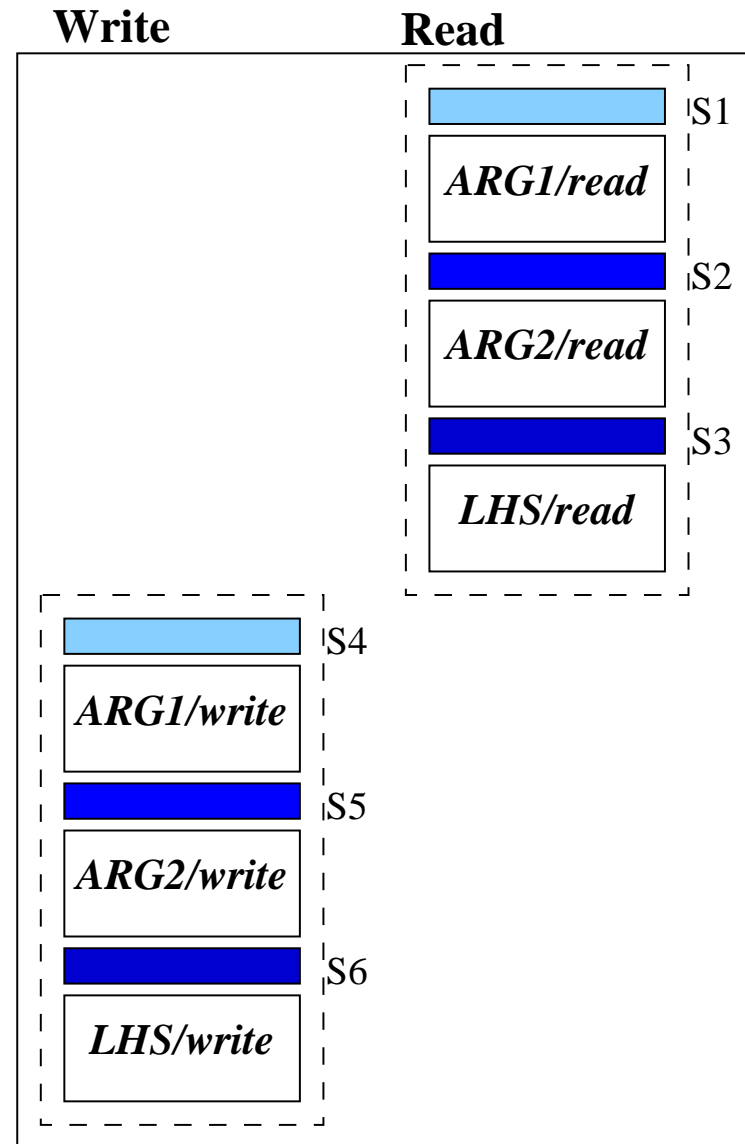
$(v\sigma, \psi_0 :- \psi_1 \dots \psi_{p-1} \dots \psi_q \dots \psi_r, i, k)$ is an item

- **Left complete**

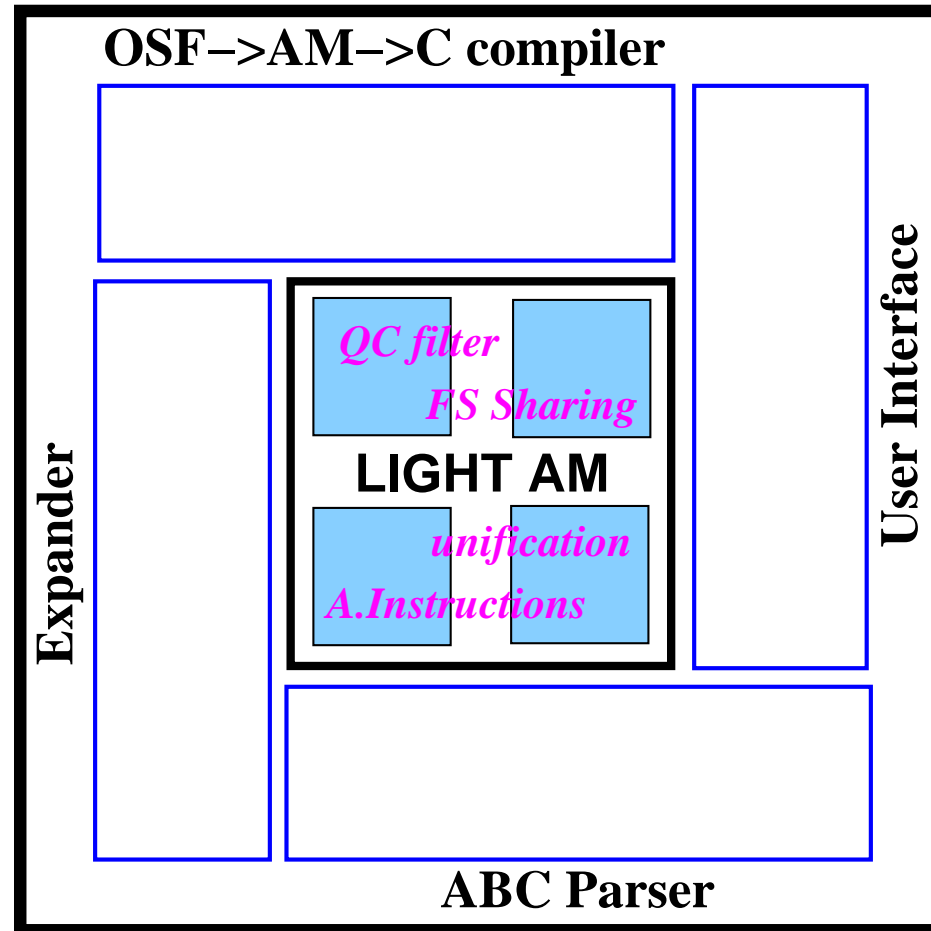
derivation

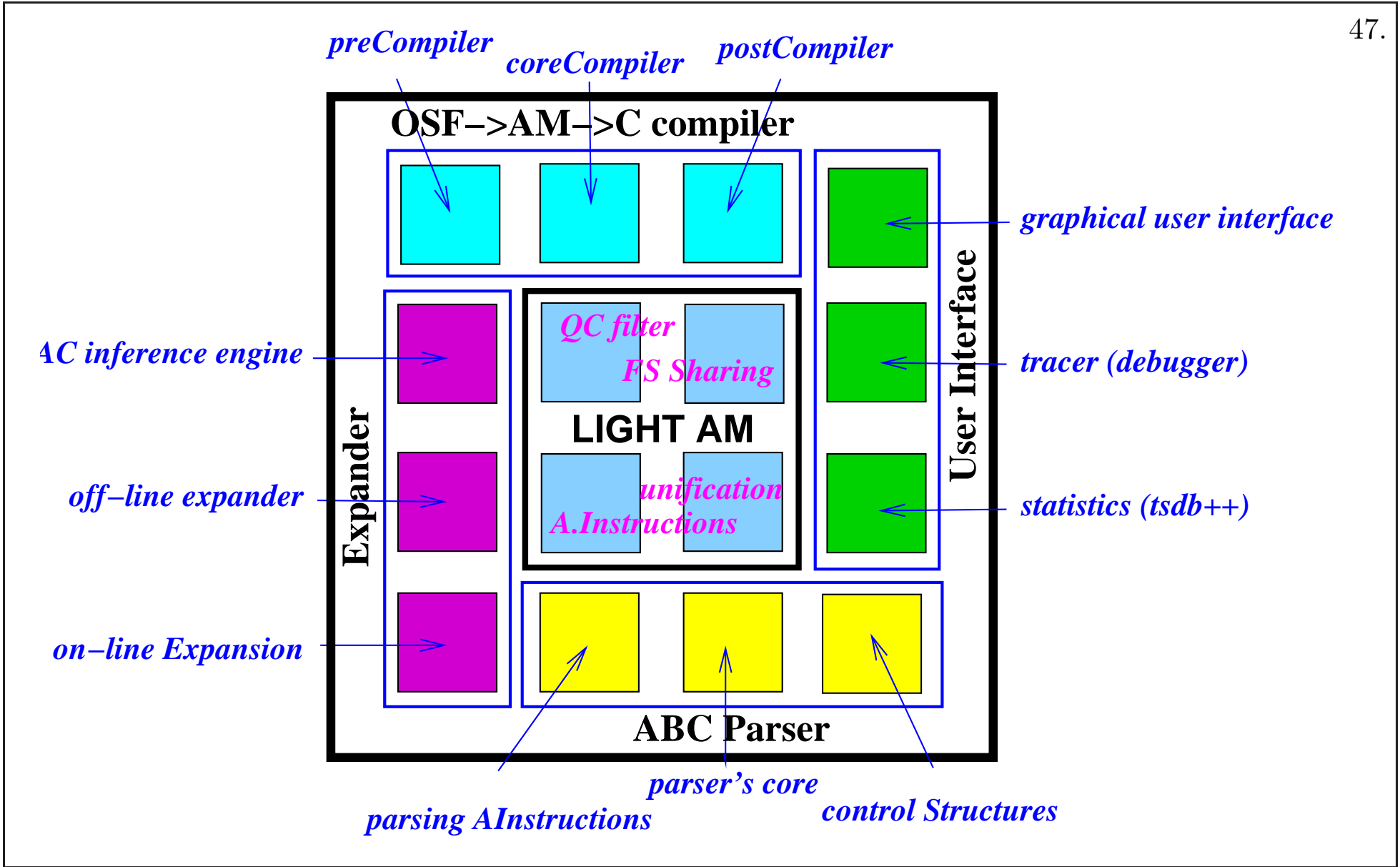
parse: $(\sigma, \psi, 0, n)$, where ψ is *start*-sorted

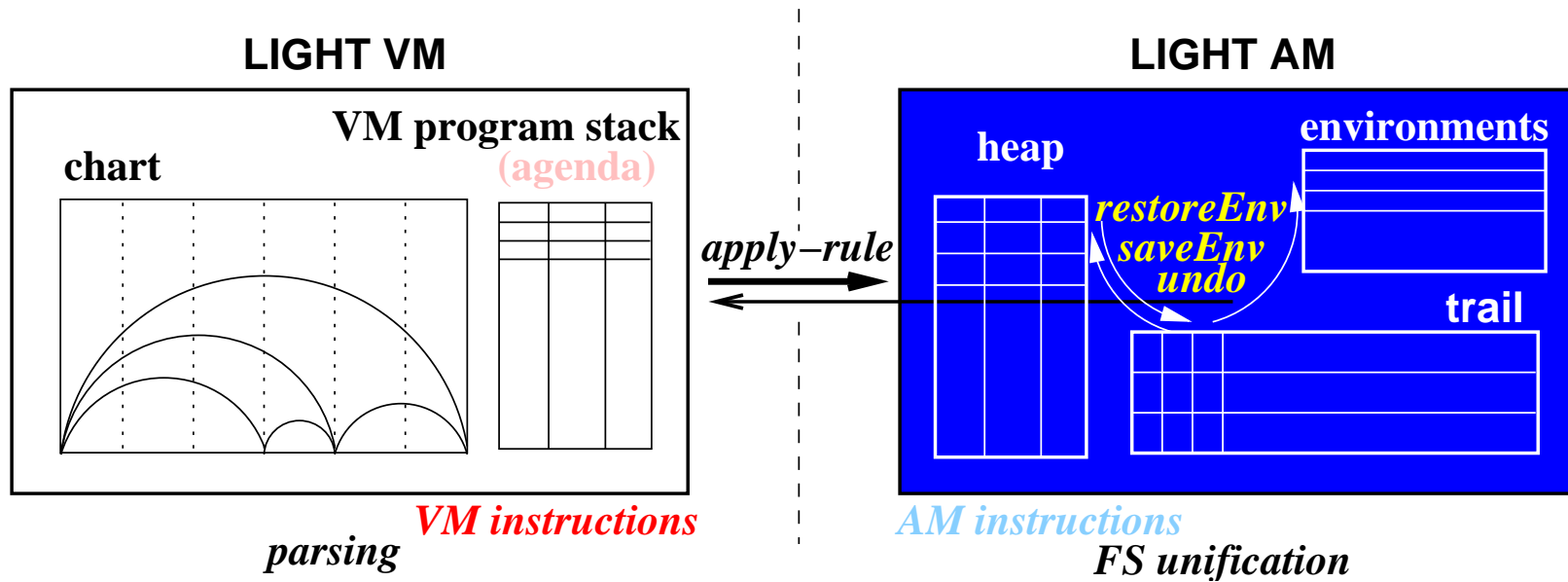
The abstract code for a binary rule



An overview of LIGHT system's architecture







Instructions in LIGHT VM and LIGHT AM

<i>VM Instructions</i>		<i>AM Instructions</i>	
<i>parsing</i>	<i>interface</i>	<i>READ-stream</i>	<i>WRITE-stream</i>
keyCorner	undo	push_cell	<u>intersect_sort</u>
directComplete	saveEnvironment	set_sort	<u>test_feature</u>
reverseComplete	restoreEnvironment	set_feature	unify_feature
apply_rule		write_test	

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 $\#(m - 1), \#(m - 2), \dots, \#0$, 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 $\#(m - 1)$, $\#(m - 2)$, ..., $\#0$, 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 .

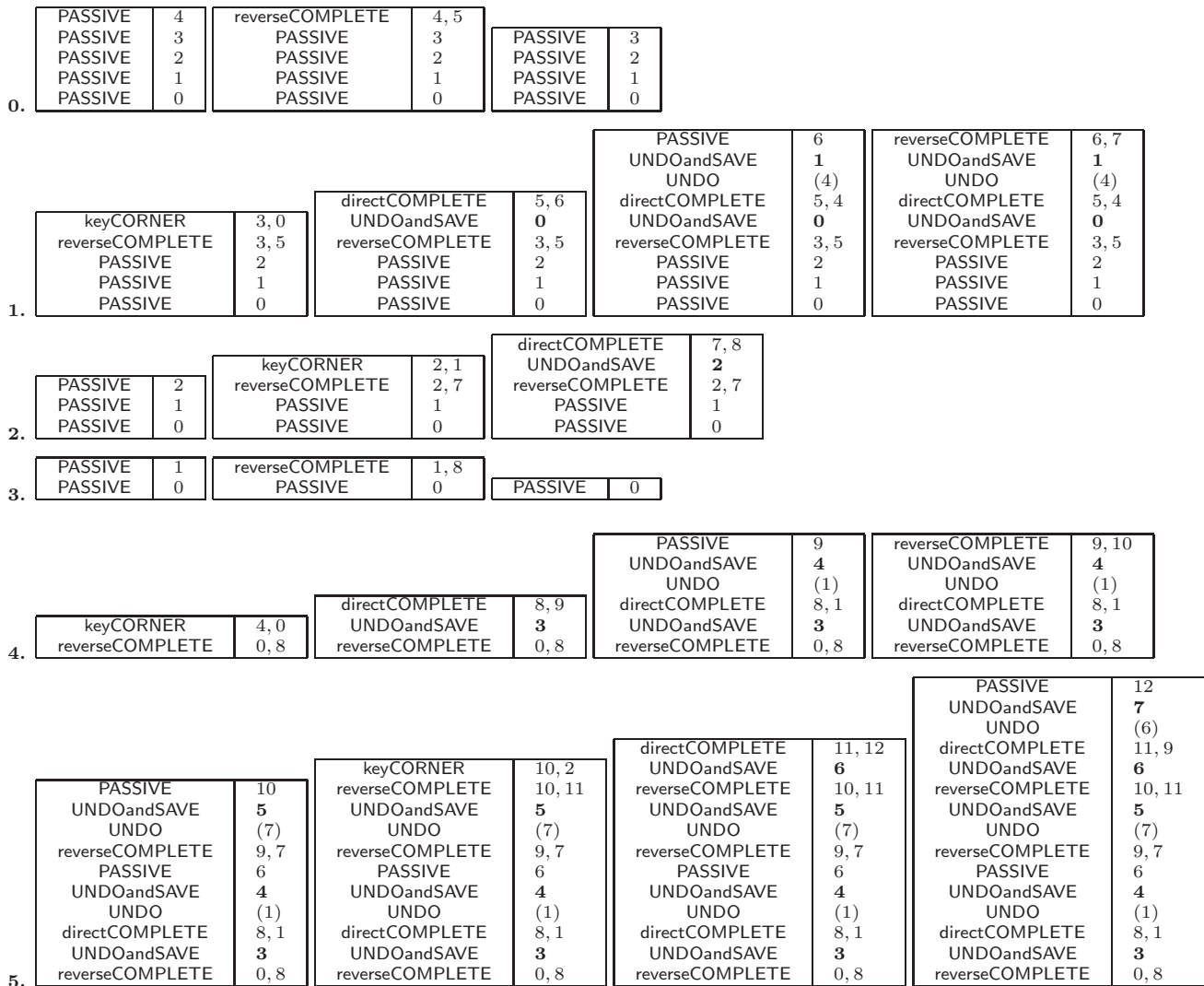
FS sharing-oriented VM instructions: Basic specifications

- `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 *parse* (control) procedure

```
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  $\neg$  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
}
```

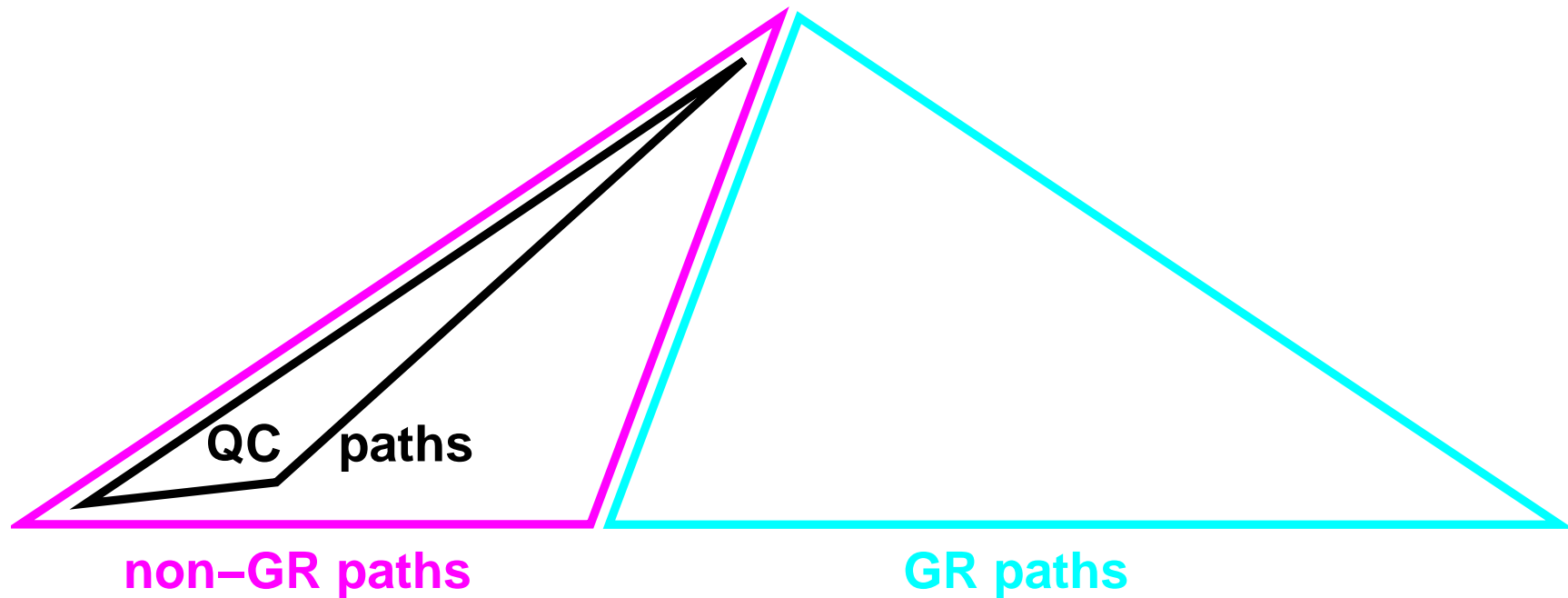
The evolution of the VM program stack (agenda) when parsing *The cat catches a mouse*



2.6 Two classes of feature paths: quick ckeck (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



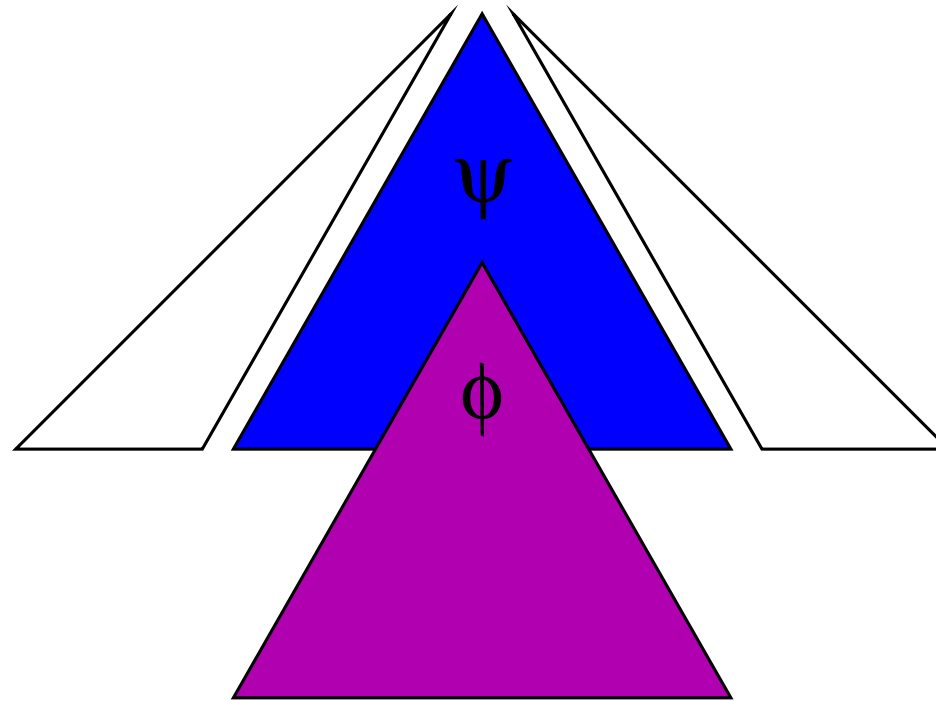
Quick-Check (QC)

63%(interp.), 42%(comp.)

Generalised Reduction (GR)

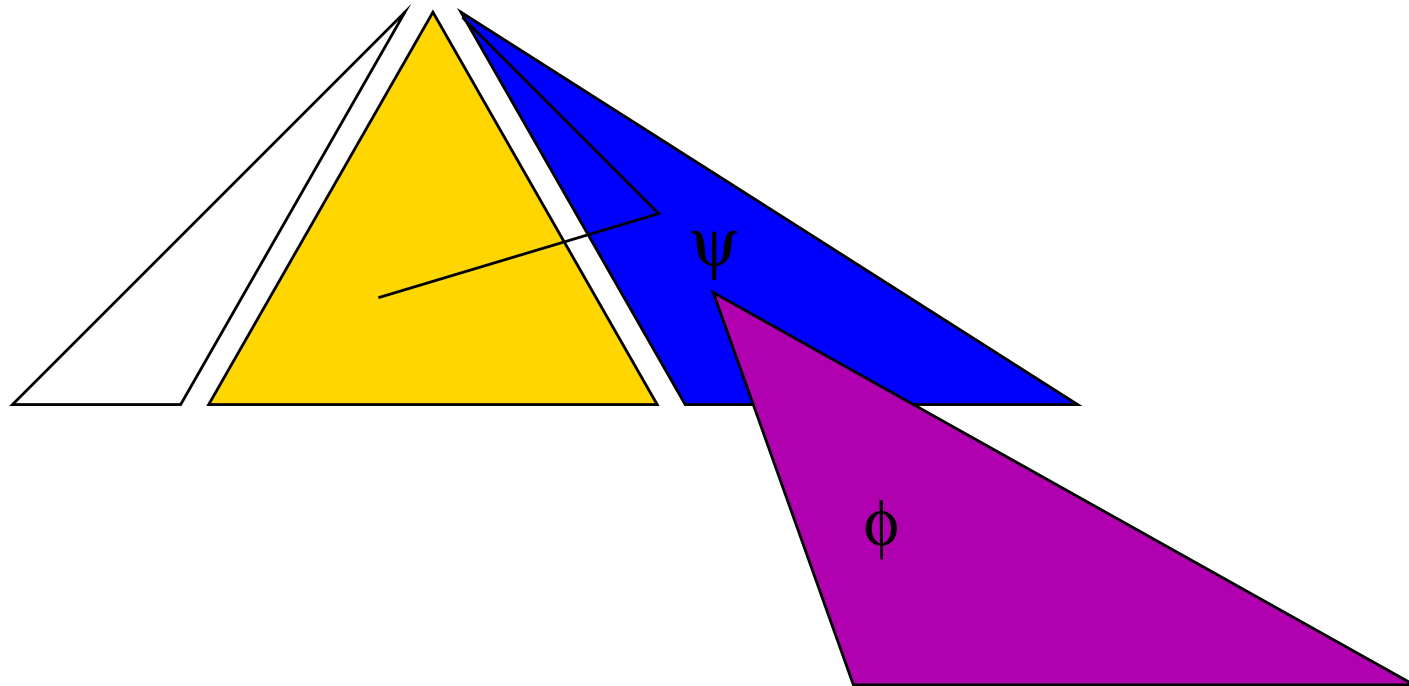
23% speed-up factor

The Quick-Check pre-unification filter



if $root-sort(\psi.\pi) \wedge root-sort(\phi.\pi) = \perp$ then $\neg unify(\psi, \phi)$

Compiled Quick-Check



$$\text{QC}_{\pi}(\psi) = \textit{on-lineQC}(\textit{compiledQC}_{\pi}(\psi))$$

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

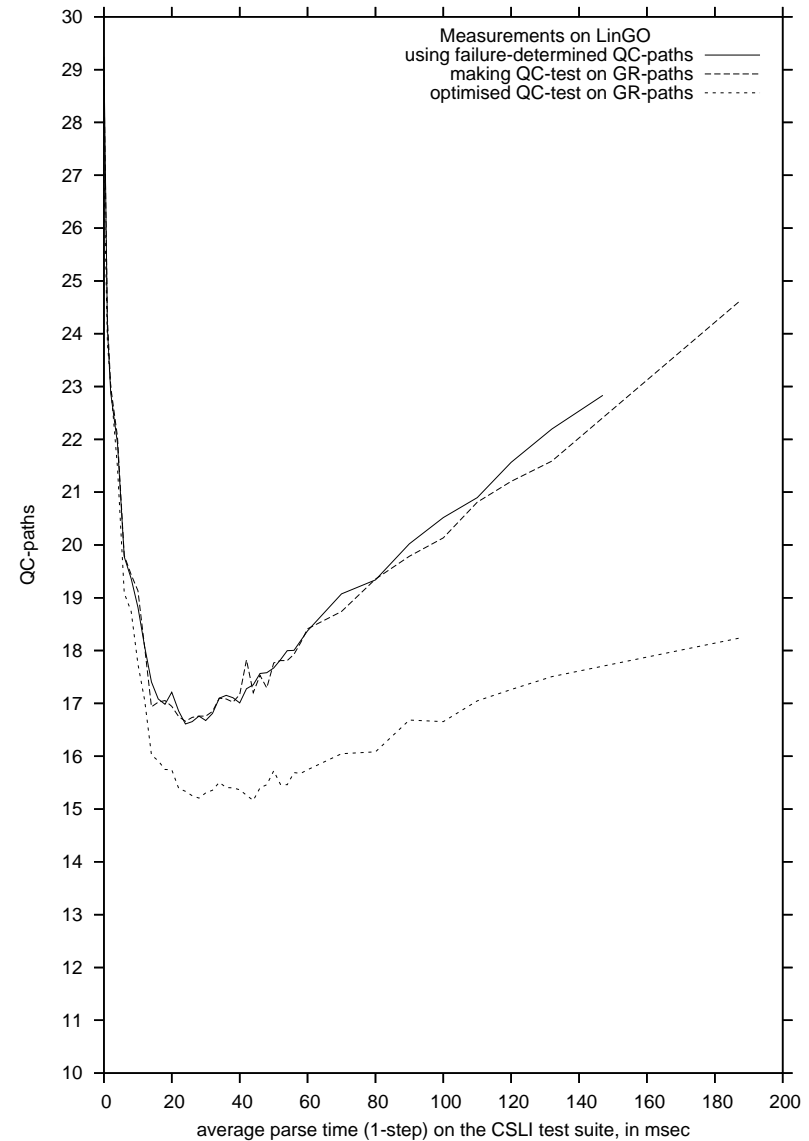
Effect: 9% additional speed up for (full) parsing with the LinGO grammar on the CSLI test suite.

Comparing the average parsing times for the GR-reduced form of LinGO on the CSLI test suite, using the LIGHT system:

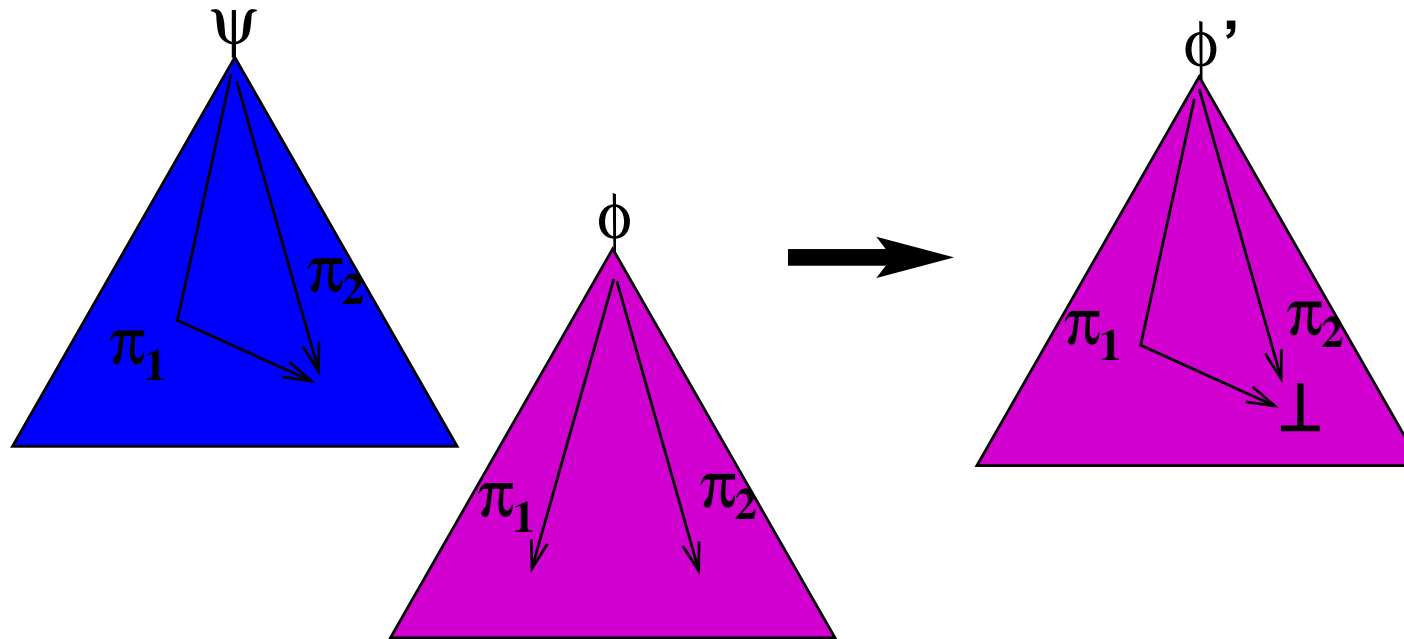
simply compiled QC

vs.

further compiled, rule-sensitive QC



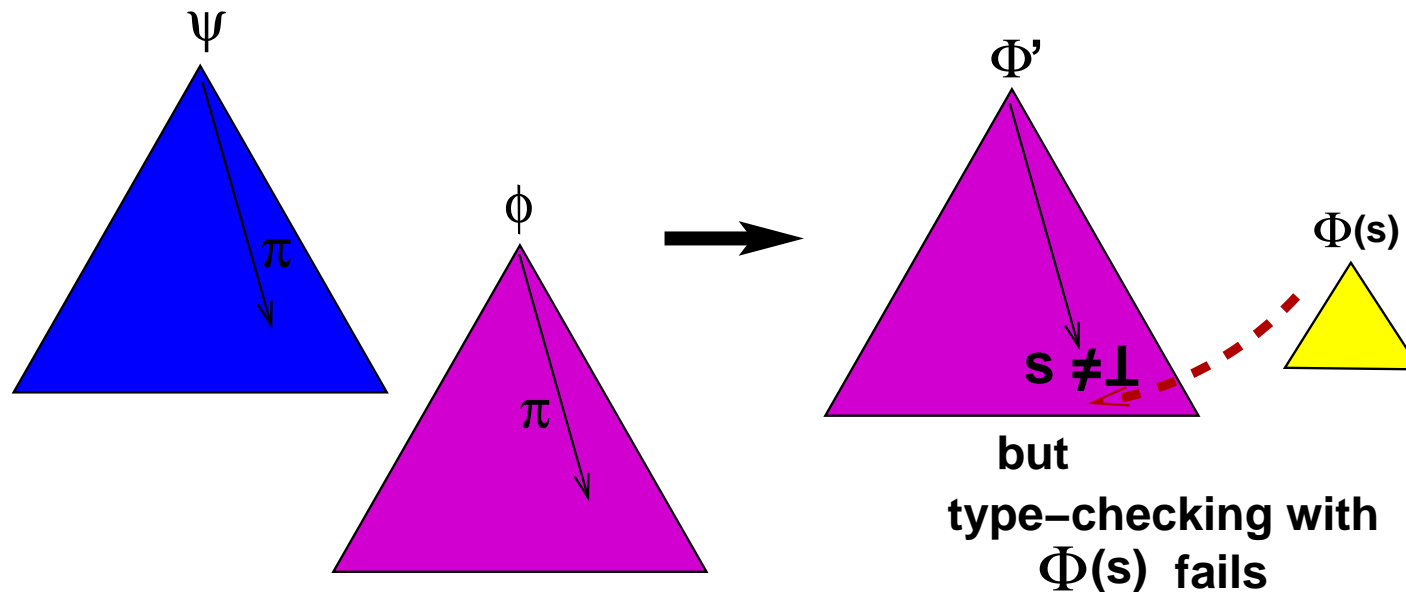
Two complementary forms to the “basic” QC test (I)
Coreference-based Quick-Check



if $\psi.\pi_1 \doteq \psi.\pi_2$ and $root\text{-}sort(\phi.\pi_1) \wedge root\text{-}sort(\phi.\pi_2) = \perp$
 then $\neg unify(\psi, \phi)$

Two complementary forms to the “basic” QC test (II)

Type-checking-based Quick-Check



if $s = \text{root-sort}(\phi.\pi) \wedge \text{root-sort}(\psi.\pi)$, and
 $s \neq \perp$ 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 $unify(\psi, \phi)$,

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 Θ , a test suite
- Output: \mathcal{G}' , a “reduced” form of \mathcal{G} , yielding the same parsing results on Θ
- Procedure:
 - for each rule $\Psi(r)$ in the grammar \mathcal{G}
 - for each elementary feature constraint φ in $\Psi(r)$
 - if removing φ from $\Psi(r)$
 - preserves the parsing (evaluation) results
 - for each sentence in the test-suite Θ
 - then $\Psi(r) := \Psi(r) \setminus \{\varphi\}$;

GR Procedure A'

65.

a **parameterized version** of the GR procedure A

- Additional input (w.r.t procedure A):
 Π , a subset of the elementary feature constraints in the grammar's rule FSs: $\Pi \subseteq \bigcup_{r \in \mathcal{G}} \Psi(r)$, and
 $\Sigma \subseteq \Theta$
- Procedure:
 - for each rule $\Psi(r)$ in the grammar \mathcal{G}
 - for each elementary feature constraint $\varphi \in \Psi(r) \cap \Pi$
 - if removing φ 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: $\text{GR}_A(\mathbf{G}, \Theta) \equiv \text{GR}_{A'}(\mathbf{G}, \Theta, \bigcup_{r \in \mathcal{G}} \Psi(r)), \Theta)$

GR Procedure B

an **incremental** GR procedure

$i = 0, \mathcal{G}_0 = \mathcal{G}, \Pi_0 = \cup_{r \in \mathcal{G}} \Psi(r);$
do 1. apply the GR procedure $A'(\mathcal{G}, \Theta, \Pi_i, \Sigma_i = \{s_i\})$,
 where s_i is a sentence chosen from Θ ;
 let \mathcal{G}_{i+1} be the result;
 2. eliminate from Θ the sentences for which
 \mathcal{G}_{i+1} provides the same parsing results as \mathcal{G} , and
 take $\Pi_{i+1} = \Pi_0 \setminus \{\cup_{r \in \mathcal{G}_i} \Psi_i(r)\}$, where
 $\Psi_i(r)$ denotes the FS associated to rule r in \mathcal{G}_i
 3. $i = i + 1$;
until $\Theta = \emptyset$

On the incremental nature of GR procedure B

- as sets of elementary constraints:

$$\Psi_0(r) \supseteq \Psi_1(r)$$

$$\Psi_0(r) \supseteq \dots \supseteq \Psi_{i+1}(r) \supseteq \Psi_i(r) \supseteq \dots \supseteq \Psi_2 \supseteq \Psi_1$$

$$\Pi_0 \subseteq \Pi_1(r)$$

$$\Pi_0 \subseteq \dots \subseteq \Pi_{i+1} \subseteq \Pi_i \subseteq \dots \subseteq \Pi_2(r) \subseteq \Pi_1(r)$$

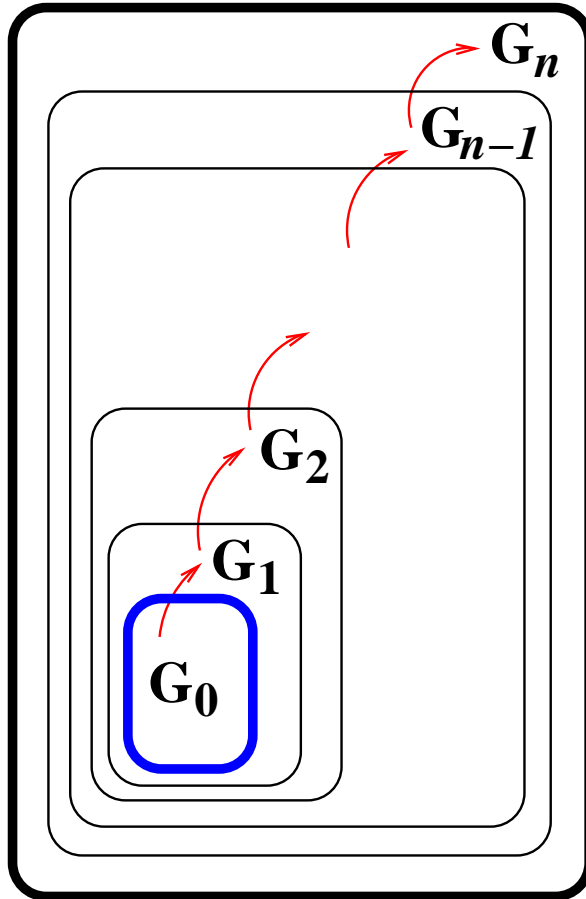
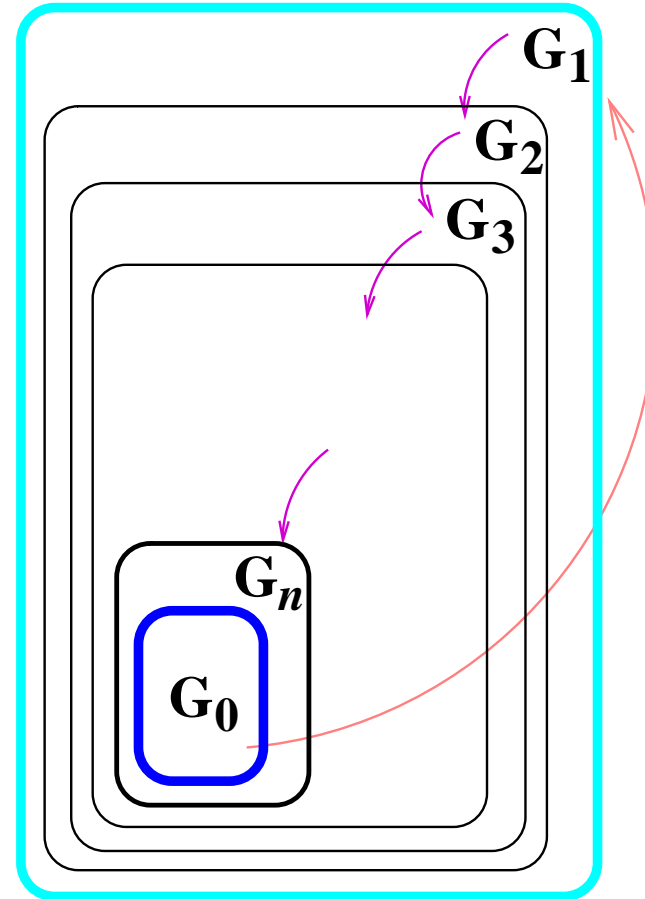
- as logical models:

$$\mathcal{G}_0 \models \dots \models \mathcal{G}_{i+1} \models \mathcal{G}_i \models \dots \models \mathcal{G}_2 \models \mathcal{G}_1$$

- as parsing results:

$$\mathcal{G}_{i+1}(\cup_{j=1}^i \Sigma_j) = \mathcal{G}_0(\cup_{j=1}^i \Sigma_j)$$

Note: Σ_i can be thought of as $\{s_i\}$ extended with the sentences which were eliminated from Θ at step 2 in GR_B following the obtention of \mathcal{G}_{i+1}

GR procedure A**GR procedure B**

Improvements to the GR procedure B

1. **sort the test suite Θ** on the number of unification failures per each sentence, in decreasing order; therefore get a “heavy”, very effective reduction first (\mathcal{G}_1)
2. **lazy elimination on sentences from Θ :**
at step 2 in GR_B , eliminate only(!) from the beginning of Θ those sentences which are correctly parsed by \mathcal{G}_{i+1}

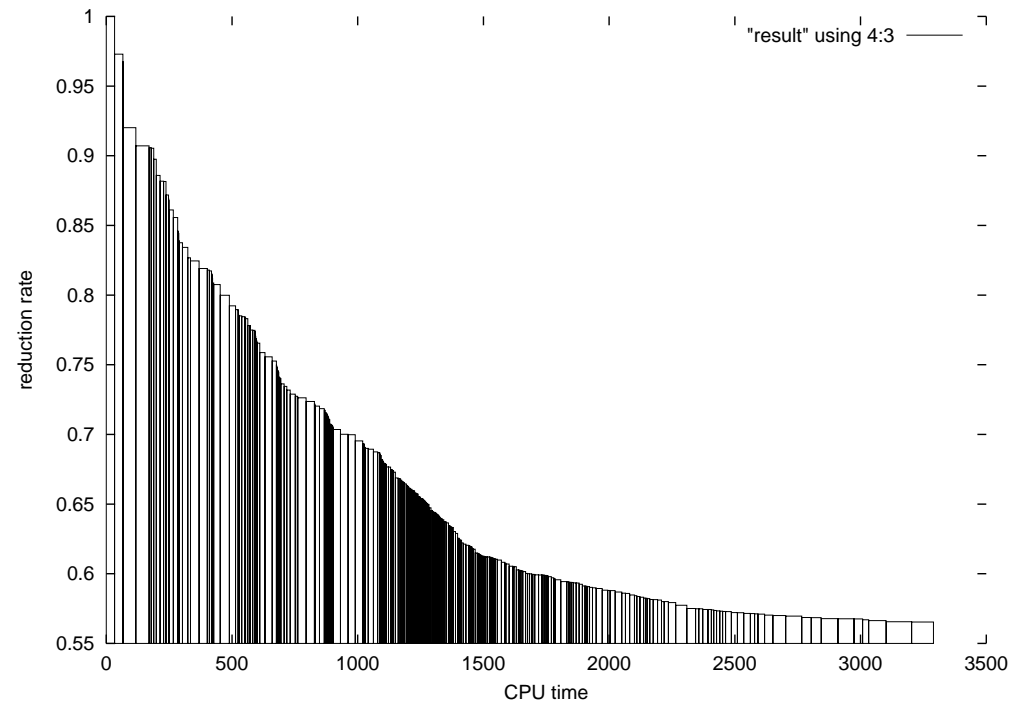
Note: 1. & 2. also contribute to reducing the effect of **resource exhaustion** which may appear due to **grammar over-reduction**

Improvements to the GR procedure B (Cont'd)

70.

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 upwords/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

<i>GR procedure</i>	<i>A</i>	<i>B</i>
FC reduction rate: average non-GR vs. all paths in r.arg.	58.92% 260/494 (52.64%)	56.64% 176/494 (35.62%)
<i>average parsing time, in msec.</i>		
using full-form rule FSs		21.617
1-step parsing (reduction %)	16.662 (22.24%)	16.736 (22.07%)
2-step parsing (red. %)	18.657 (13.12%)	18.427 (14.20%)

The GR effect on the LinGO grammar:
memory usage reduction for LIGHT AM

	<i>full-form rules</i>	<i>GR-restricted rules</i>	
		<i>1-step parsing</i>	<i>2-step parsing</i>
heap cells	101614	38320 (62.29%)	76998 (24.23%)
feature frames	60303	30370 (49.64%)	58963 (02.22%)

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)