Improved Type Hierarchy Processing and Display

John Carroll

Department of Informatics, University of Sussex, UK

DELPH-IN Meeting, Paris, June 2018

With thanks to: Ann, Glenn, Petter, Uli C, Woodley

Outline

Improved type hierarchy display in the LKB

Efficient computation of BCPOs for large type hierarchies

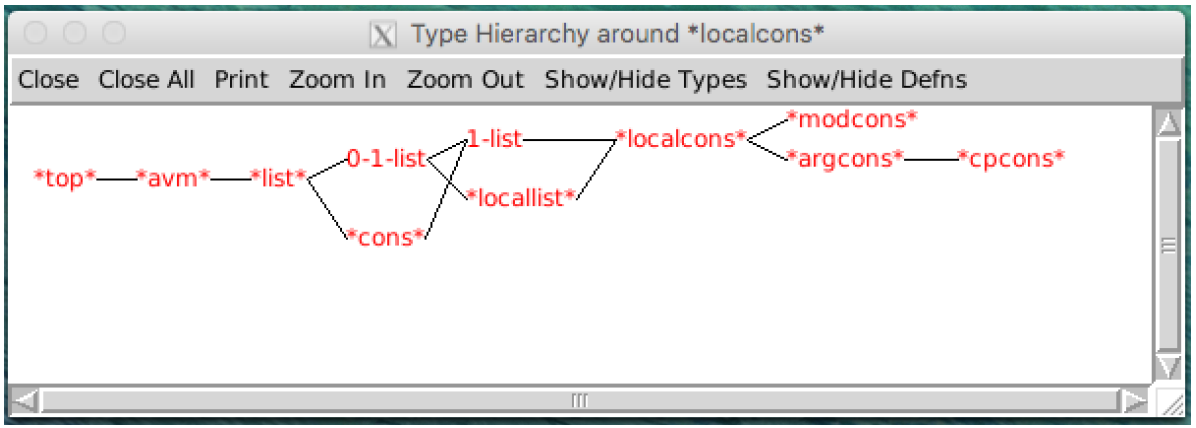
- Baseline algorithm
- Fast transitive reduction
- Improvements
- Empirical results

Improved Type Hierarchy Display

Part of recent enhancements to LKB-FOS

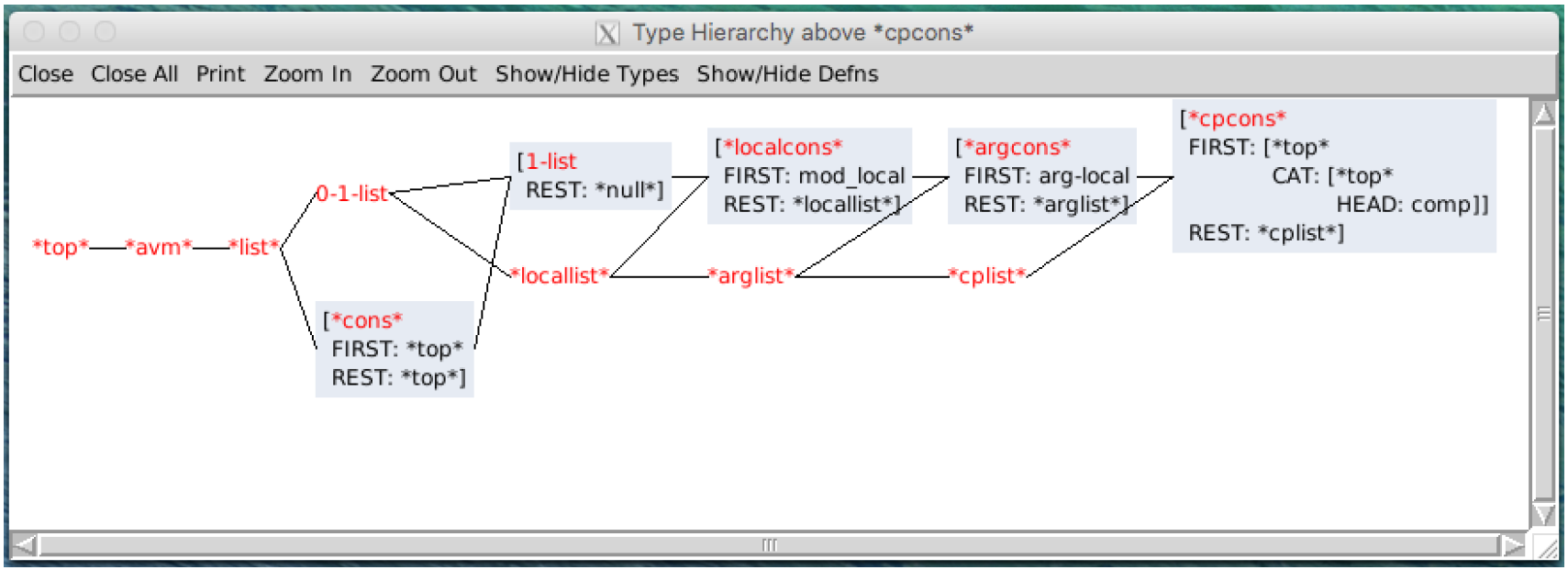
- intended to allow more effective debugging of type hierarchies
- possibly also useful for teaching and demos
- will be ported back to 'classic' LKB

Type Hierarchy below / above / around; e.g. ERG (1214), types around `*localcons*`:

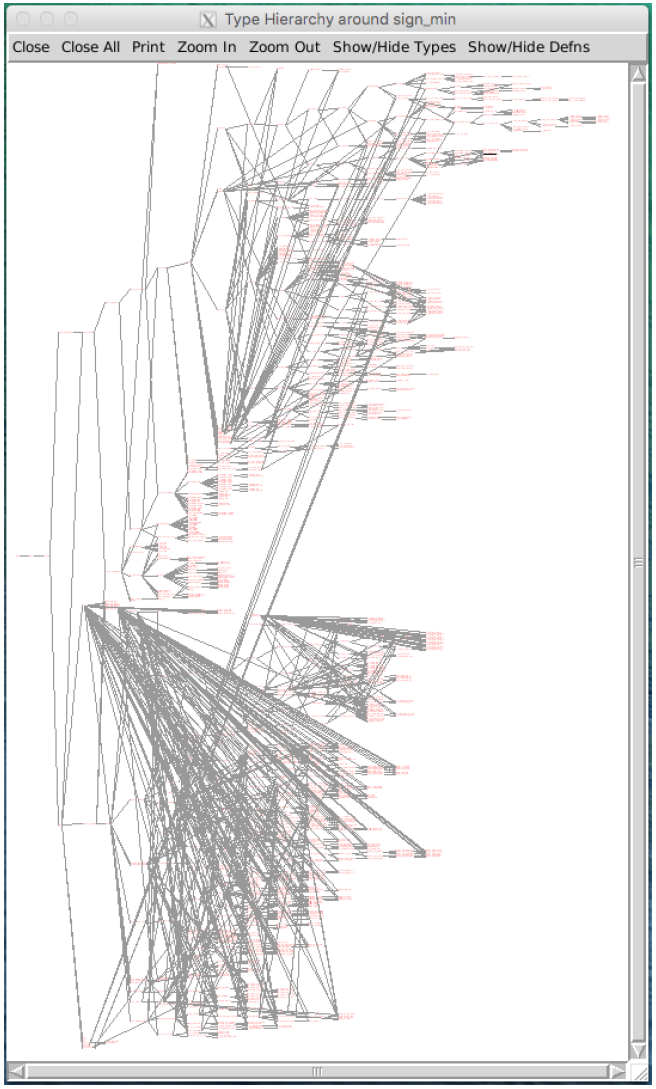


Show/Hide Types; as before, but much more efficient for large hierarchies

Show/Hide Defns; e.g. hierarchy above `*cpcons*` showing type constraints:



Zoom In / Zoom Out; e.g. hierarchy around `sign_min` zoomed out 5 times:



Efficient computation of BCPOs

Computation performed during grammar loading (creates 'GLB types')

- so it's inside the grammar testing and debugging cycle

Tolerably efficient in all four DELPH-IN parsing/generation systems for most current grammars

- but **not** for Norsyg

In early August 2017, BCPO computation for Norsyg took around 1 min 30 sec (agree), 30 minutes (LKB-FOS), and 2 hours or more (ACE)

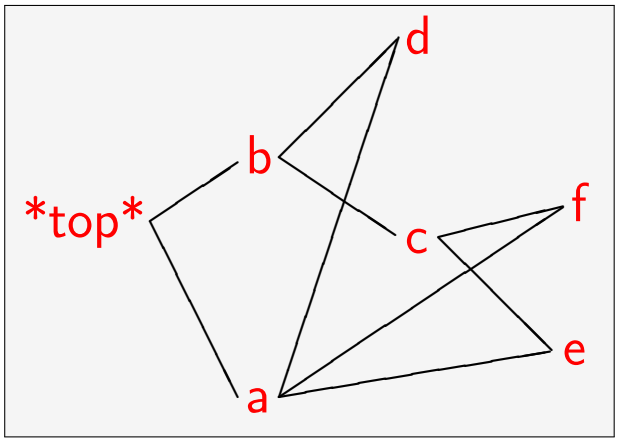
- why? it's an $O(n^3)$ algorithm applied to a 40,000-type hierarchy

Baseline Algorithm

```
1: Assign a unique bit code to each 'authored' type
2: repeat
3:     for each pair of bit codes do
4:         if their bitwise AND is not the code for an existing type then
5:             create a GLB type and add it to the pool of bit codes
6: until no new GLB types
7: Integrate GLB types into hierarchy through subsumption relationship
8: Remove redundant links using transitive reduction
```

Ulrich Callmeier (2001) *Efficient Parsing with Large-Scale Unification Grammars*. Diploma Dissertation, Saarland University. http://www.coli.uni-saarland.de/~uc/thesis/thesis.ps

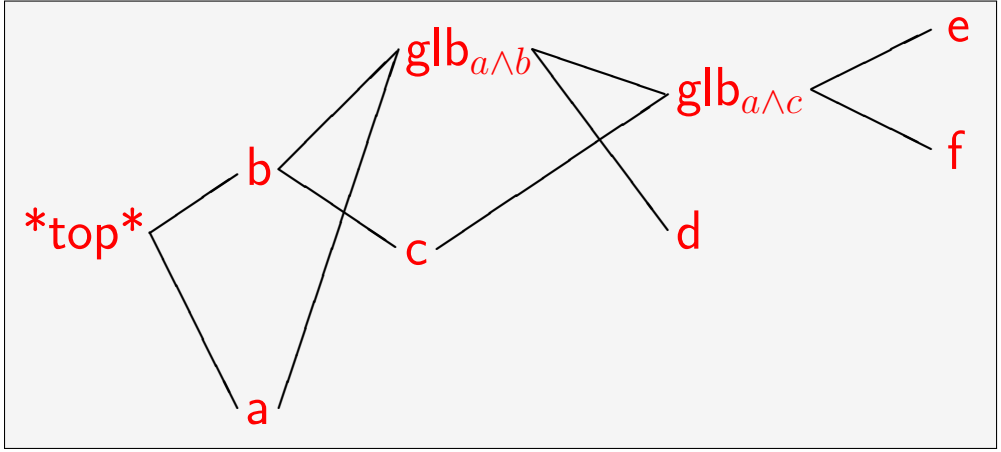
```
a := *top*.
b := *top*.
d := a & b.
c := b.
e := a & c.
f := a & c.
```



	top	a	b	c	d	e	f
top	1	1	1	1	1	1	1
a		1			1	1	1
b			1	1	1	1	1
c				1		1	1
d					1		
e						1	
f							1
a ∧ c						1	1
a ∧ b					1	1	1

	top	a	b	c	d	e	f	$\text{glb}_{a\wedge c}$	$\text{glb}_{a\wedge b}$
top		1	1	1	1	1	1	1	1
a					1	1	1	1	1
b				1	1	1	1	1	1
c						1	1	1	
d									
e									
f									
$\text{glb}_{a\wedge c}$						1	1		
$\text{glb}_{a\wedge b}$					1	1	1	1	

blue entries
removed by
transitive reduction



Fast Transitive Reduction

We want to apply transitive reduction to a large, though sparse DAG; let's first consider transitive closure, which is highly related

Warshall's Algorithm

for $j = 1 \ldots n$ **do** ▷ Process each column
  for $i = 1 \ldots n$ **do** ▷ Process each row
    for $k = 1 \ldots n$ **do**
      $M(i, j) \leftarrow M(i, j) \vee (M(i, k) \wedge M(k, j))$

Warshall (improved)

for $j = 1 \ldots n$ **do**
  for $i = 1 \ldots n$ **do**
    if $M(i, j) = 1$ **then**
      $M(i, *) \leftarrow M(i, *) \vee M(j, *)$ ▷ 'OR' row j into row i

Warren's Algorithm

for $i = 2 \ldots n$ **do** ▷ Row-wise, below main diagonal
  for $j = 1 \ldots i-1$ **do**
    if $M(i,j) = 1$ **then**
      $M(i,*) \leftarrow M(i,*) \vee M(j,*)$
for $i = 1 \ldots n-1$ **do** ▷ Row-wise, above main diagonal
  for $j = i+1 \ldots n$ **do**
    if $M(i,j) = 1$ **then**
      $M(i,*) \leftarrow M(i,*) \vee M(j,*)$

For sparse DAGs, we need the following to be efficient:

- find next 1 in a bit vector
- compute (in-place) the bit-wise OR of two bit vectors

Transitive reduction is very similar, except for $M(i,*) \leftarrow M(i,*) \wedge \neg M(j,*)$

Are there any existing comparative evaluations? Yes!

Gerald Penn (2006) Efficient transitive closure of sparse matrices over closed semirings. *Theoretical Computer Science*, 354, 72–81. https://doi.org/10.1016/j.tcs.2005.11.008

[1] Version of ERG with 4,305 types (11,720 entries including main diagonal), C implementation by G. Penn, on 32-bit 2.4 GHz Xeon server

[2] ERG (1214), 6,016 types (19,107 entries including main diagonal), SBCL implementation, on 64-bit 2.5 GHz i5 desktop

Implementation	*CPU time (sec)*
Warshall [1]	481
Warshall, naive [2]	668
Warshall, improved [2]	0.14
Warren [2]	0.012

Improvements

A. Partition type hierarchy into independent sub-graphs

- For each descendant d of x, are each of d's parents also one of x's descendants? (i.e. do d's parents remain within the partition's 'envelope')
- For ERG (1214), there are 134 non-trivial partitions

  `SIGN_MIN`: 1149 types
  `PREDSORT`: 1016 types
  `SYNSEM_MIN`: 866 types
  ...
  `XMOD`: 7 types

- apply the BCPO algorithm to each partition individually

B. Add 'summary words' to each bit code: each successive bit represents whether *any* bit is set in each successive 64-bit word in the bit code (wrapping around at end)

E.g. determining whether the bitwise AND of a pair of bit codes is all zero (on a 4-bit computer):

```
0000 1101 0000 1111 0000 0011
AND
0000 0000 1011 0000 1100 0000
->
0000 0000 0000 0000 0000 0000
```

We can speed this up by first checking the AND of the bit codes' summary words:

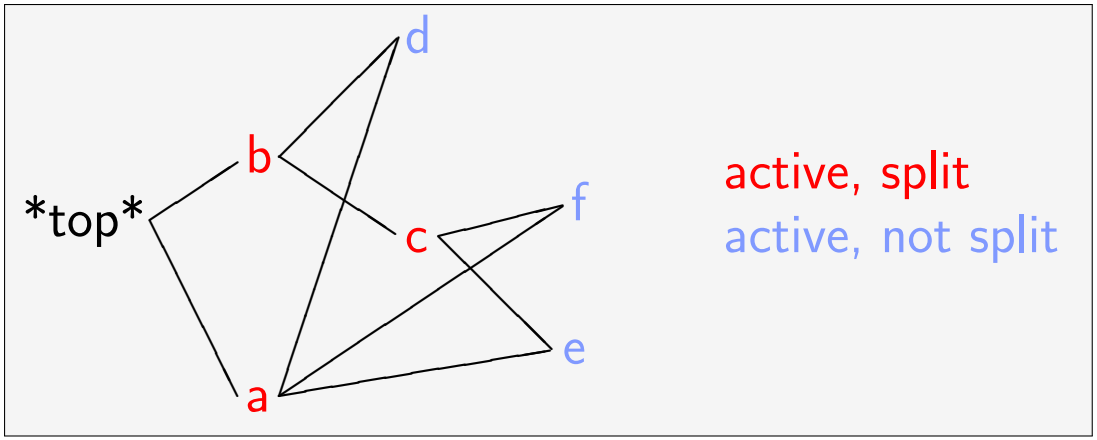
```
0101
AND
1010
->
0000
```

C. Only consider a subset of 'active' types for computing GLBs; only some of these ('split' types) need participate in the pairwise AND tests to find new GLBs

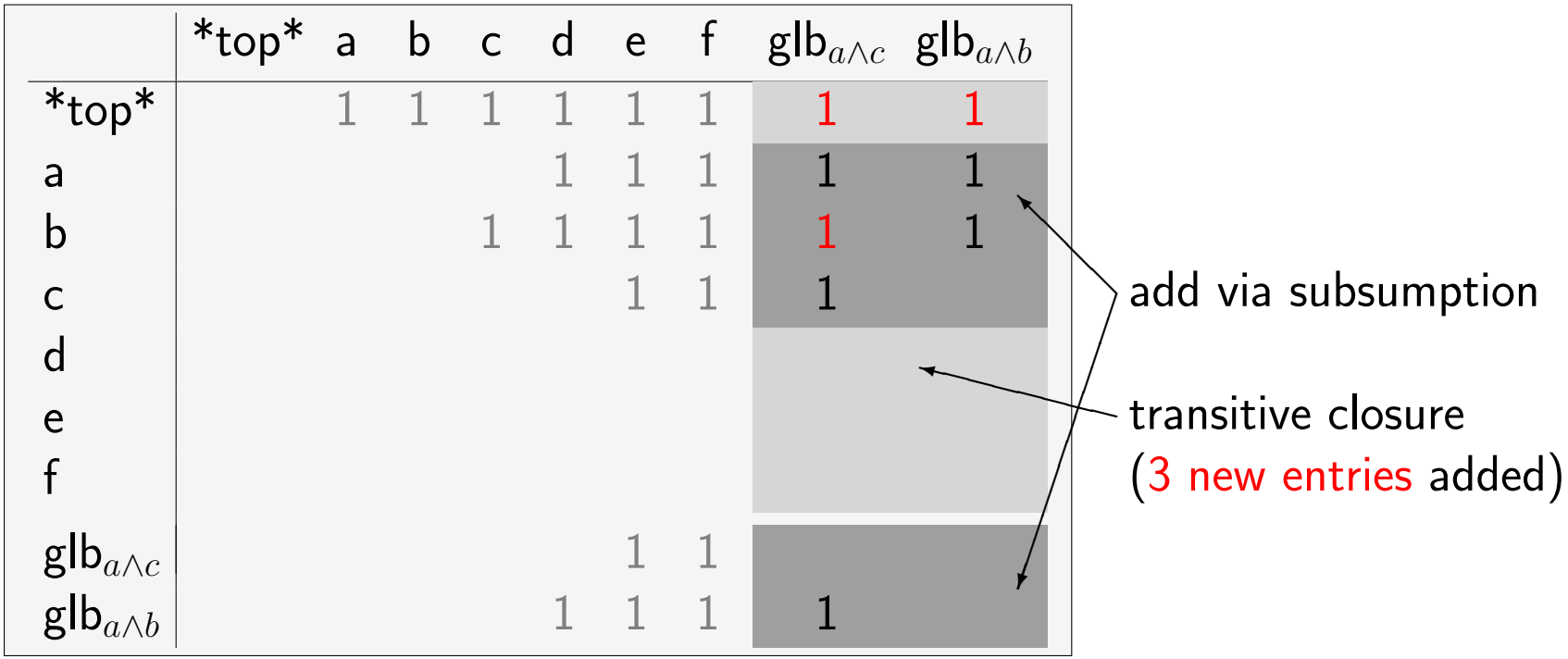
More precisely, we classify types according to 3 tests:

1. more than 1 parent and/or more than 1 daughter
2. not inside a tree-shaped part of hierarchy? (nor a leaf type with 1 parent)
3. more than 1 daughter that is not in a tree-shaped part of hierarchy

Active types satisfy tests 1 and 2; split types are active and additionally satisfy 3

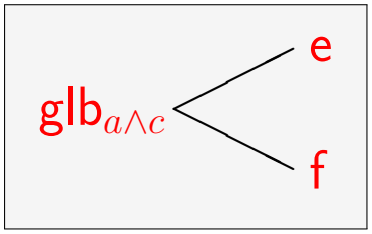


D. When integrating new GLB types into the hierarchy, test subsumption against split types, and compute the remaining authored type $\rightarrow$ GLB links via (partial) transitive closure



E. Ignore the result of a bitwise AND containing only a single 1 bit, since it must be the code for an existing (authored) type

Also, if a new GLB type has only 2 bits set (i.e. has only 2 active daughters) then don't add it to the pool, and don't test whether it subsumes any other GLBs



F. Record start and end indices for bit codes, and use these to skip over leading and trailing all-zero words in AND / subsumption tests

```
0000 0000 1011 0000 1100 0000
->
start=2, end=4, code=0000 0000 1011 0000 1100 0000
```

G. Use bit code start and end indices to filter out non-overlapping / non-contained bit codes in AND / subsumption tests

H. Compress bit codes (after initial assignment step), starting each code at its first non-zero word and finishing at its last

- requires careful programming to correctly align codes

```
0000 0000 1011 0000 1100 0000
->
start=2, end=4, code=1011 0000 1100
```

Empirical Results

Tests with Norsyg (1708) (42,300 authored types, 21,498 GLBs) on 3.3 GHz i5

Baseline: faithful implementation of PET's algorithm in LKB-FOS, without improvements but with fast transitive reduction

System	*CPU time (sec)*
Baseline	982
+ A. Partition hierarchy	866
+ B. Summary words (3)	82
+ C. Active and split types	43
+ D. Splits-GLB partial closure	32
+ E. Filter 1-bit ANDs, 2-bit GLBs	23
+ F. Skip all-zero bit code words	12
+ G. Filter on bit code indices	7.4
+ H. Compressed bit codes	3.9

Baseline + C +...+ H
6.6 sec

Baseline + F + G + H
30 sec

Results for some other DELPH-IN grammars:

Grammar (version)	*CPU time (sec)*	
	Baseline	*Improved*
ERG (1214)	1.6	0.06
JACY (2016-11-17)	0.03	0.003
GG (Oct_2008)	2.6	0.07
HAG (1607)	0.4	0.07

Summary

Improved type hierarchy display

- types below / above / around, show/hide type constraints, zoom in / zoom out

Efficient computation of BCPOs for large type hierarchies

- massively combinatorial
- algorithmic and data structure improvements make a big difference
- reduced processing time for all grammars, and for Norsyg from 16 minutes to 4 seconds

Available now in LKB-FOS, and will be ported back to 'classic' LKB

Thanks! Any questions?