Variadic (n-way) Unification

status update and preliminary results

Glenn Slayden

DELPH-IN Summit, June 2011

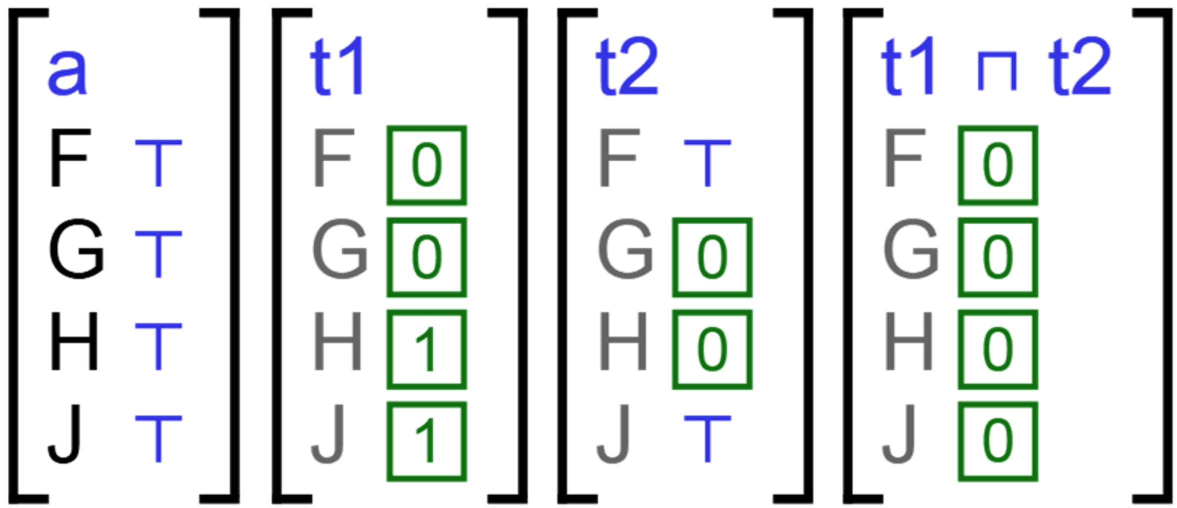
Ma.Sci. research at the *University of Washington*

Supervised by Emily Bender

Summary

- Unification is by far the most expensive part of parsing
- PET and LKB implement Tomabechi's (1991, 1992) "quasi-destructive" method
- I investigate a new algorithm where the fundamental recursive function accepts node arguments in variable arity
- The method is implemented in *agree*, a new DELPH-IN parser
- Early results are promising, especially during unpacking, where all rule daughter positions can be unified at once
- n-way unification outperforms a Wroblewski (1987)-style incremental unifier in controlled intra-system evaluation
- Unification satisfiability checker pseudo-code:
  `http://www.agree-grammar.com/n-way-unification/satisfiability.html`

What makes TFS unification difficult?



- Coreference spreading: The unification of *t1* and *t2* equates two coreference equivalence classes which remain distinct within *t1*
- This process can continue to chains of arbitrary length

Pereira 1985

"A Structure-sharing representation for unification-based grammar formalisms"

- Basic unification algorithm (from theorem proving work in the early 1970s) remains unchanged
- Instead, the underlying graph representation is changed to reduce the amount of new structure written
- This is done by maintaining each TFS as an $\langle update, skeleton \rangle$ tuple
- Applying updates incurs penalties when the derived instance is accessed

Wroblewski 1987

“Nondestructive Graph Unification”

- “Incremental” unification
- Build new structure as needed to avoid destroying old
- Generation counter invalidates all temporary structures associated with failed work in a single operation
- Incremental algorithms inherently suffer from “over-copying”

☞ For comparison with n-way, *agree* includes a (thread-safe) incremental-type unifier (results in this presentation)

Tomabechi 1991, 1992

With structure sharing adaptation (Malouf 2000)

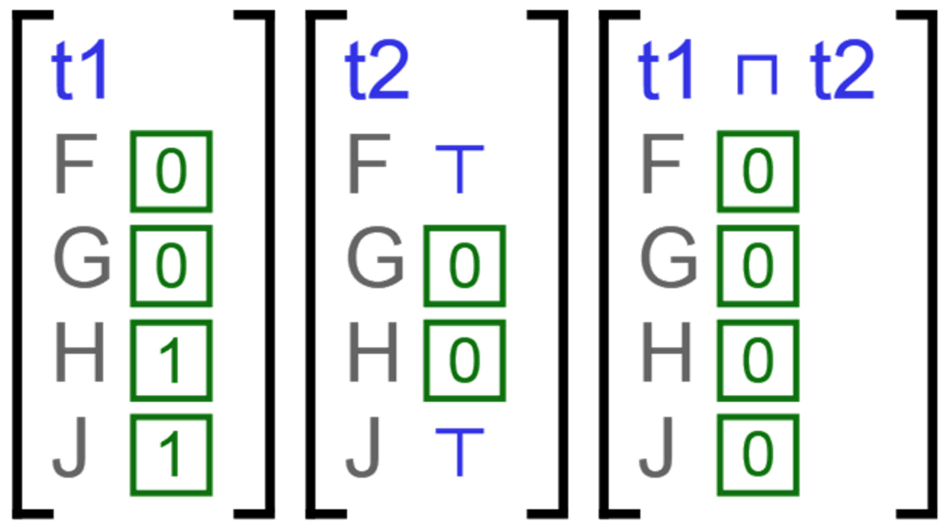
- This has been regarded as the state-of-the-art method for 20 years
- Unification is divided into two passes
  1. with no allocations, prepare data structures
  2. if successful, write new TFS
- Scratch fields are invalidated by Wroblewski's global counter technique
- Disadvantages:
  – As published, it is not thread-safe
  – Successful unification requires two passes

Other authors

- Godden (1990) "Lazy Unification" relies on language closures
  - Inefficiencies of this language construct probably nullify gains
- Emele (1991)
  - Extending Pererira's update/environment ideas; backtracking
- Kogure (1990, 1994)
- Tomuro and Lytinen (1997)
- Van Lohuizen (2000)
  - parallel adaptation of Tomabechi

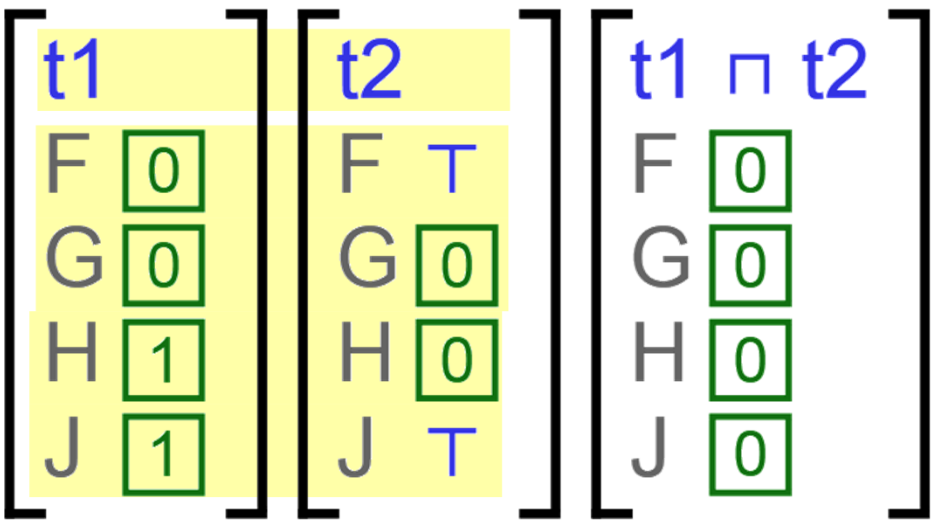
n-way unification: idea

- Observation: complexity in existing algorithms owes to the maintenance of temporary structures to account for pending equivalence classes that are subject to further spreading



- At each step, a duplex (two-argument) unifier can only join a single element to the class. Therefore:
  - scratch structures reflect the complexity of an arbitrary limitation
  - the number of recursive calls is unnecessarily high

duplex unification



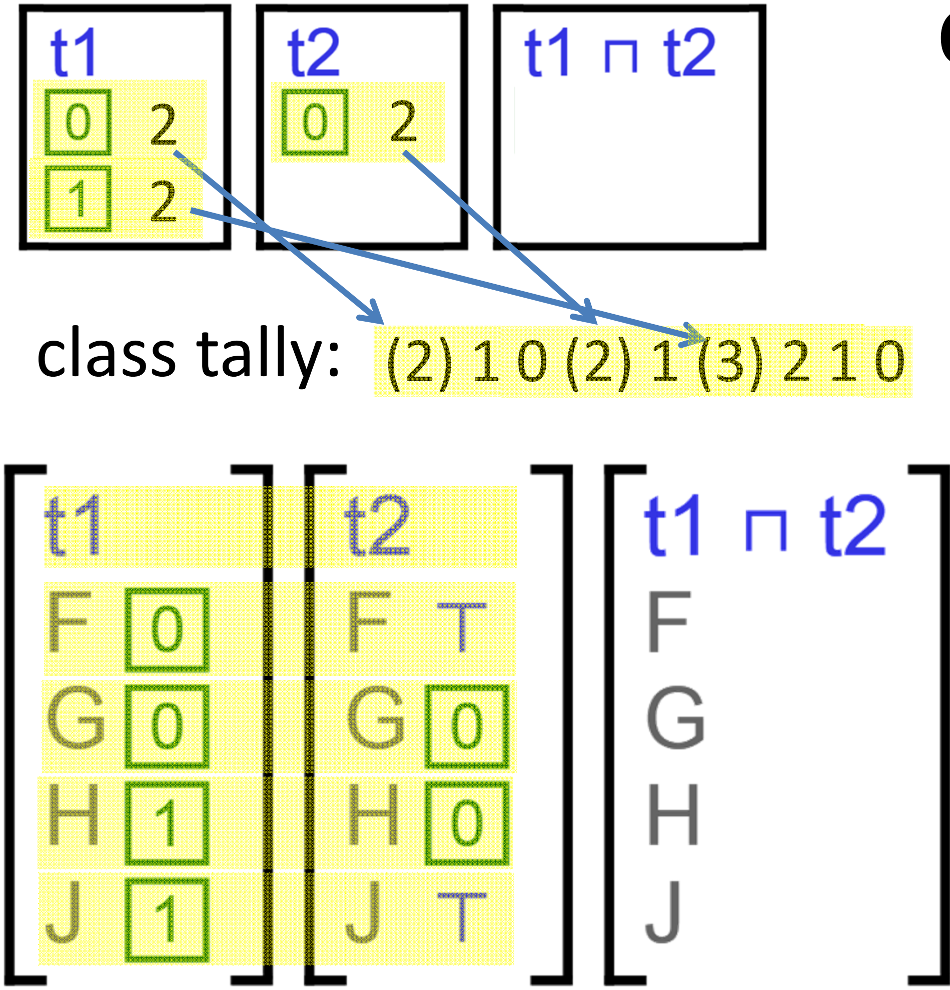
1. *unify*(t1, t2)
   F
   G
2. unify(t1-F/G, t2-G/H
   H
3. unify(#2, t1-H/J)
   J

The number of recursive calls in a top-level unification is $O(n)$ in the number of coreference equivalence classes in the *input* (3 in this case)

n-way unification

- It would preferable to unify the entire equivalence class at once, in a single function call
- Delay descent on the class until the equivalence class is definitive
- Only then, unify all nodes in the class and enter their sub-structure all at once
- To do this, a set of reentrancy tallies—invariant for each top-level TFS—is maintained and referenced during unification

example

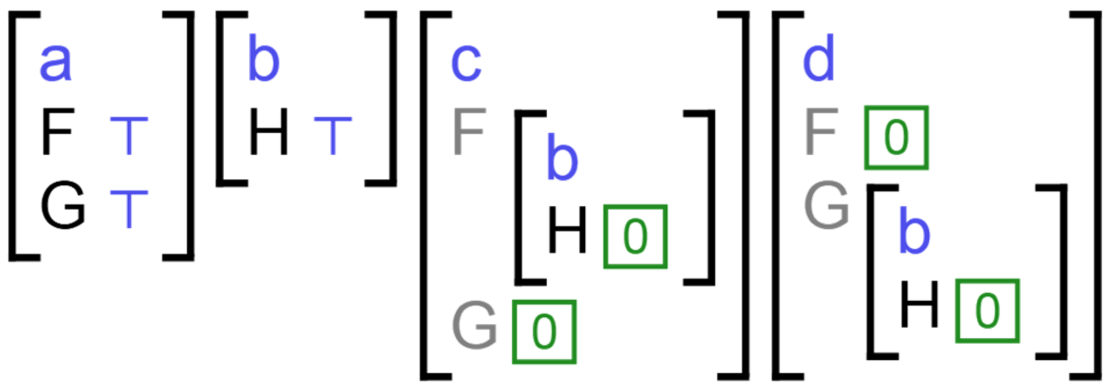


1. *unify*(t1, t2)
   F
   G
   H
   J
2. *unify*(t1-F/G,
   t2-G/H,
   t1-H/J)
3. completeness-check

The number of recursive calls is $O(n)$ in the number of equivalence classes in the ***output*** (2 in this case). In unification, the number of output classes is always $\leq$ the number of input classes.

n-way completeness check

- When the traversal is complete, remaining reentrancies for all classes must be zero
- If any are not, this indicates that parts of one or more inputs were not visited
- Unvisited parts occur when there are mutually-blocking structures:



- This condition is a true-positive for unification failure
- The cost of the check—$O(n)$ integer tests for zero, in the number of input classes—is only borne for putative successes

Determinism guarantees

- For inputs that can be unified:
  - when any classes remain, at least one of them will be exposed and completed
  - such a class will always be accessible via a prospective (not yet visited) node
- The completeness check is the key to the single-traversal guarantee:
  - n-way unification requires only one single, step-wise traversal of the input TFSes, greedily descending only on completed classes
  - Traversal order—for discovering the exposed, completed classes—is irrelevant

Space analysis - satisfiability

- For satisfiability checking, the class list, plus a single integer tally is the entire scratch requirement
- Worst-case $O(n)$ in the number of input classes
- Best case $O(n)$ in the number of output classes
- For best performance, *the class lists are directly maintained in the variadic format of the (eventual) recursive call*

Persistent space analysis

- Tally sets are an additional persistent storage cost for each top level TFS
  - $O(n)$ in the number of coreferenced nodes
  - agree uses 1 byte tallies, allowing a single coreference to have up to 255 reentrancies
- Computing these tally sets are a “free” product of the unification that produces any TFS
- But they require administration: a more generally pervasive association between nodes and their top-level TFS
  - However, carrying this association also solves the problem of spurious structure sharing (Malouf 2000, aka theorem proving’s “renaming problem” Pereira 1985)
  - Consistently distinguishing nodes by $< TFS, node >$ tuple within the unifier allows aggressive (extra-linguistic) structure sharing

Implementation options

- Class lists can be discarded after descent is undertaken
- In the minimal requirement, type unification (in addition to descent) is deferred until the class is definitive
  - Depending on storage details, this may incur extra node accesses
  - This increases the number of failures detected solely by the completeness check
  - To detect overall failures earlier, and avoid extra node accesses, it is trivial to maintain a running type unification with each class
  - For the above reasons, *agree* implements this variation

Extending the n-way satisfiability checker to the full case of writing the result TFS

- For each class, also maintain a list of referring nodes
- In the *agree* implementation, this is a linked list which adds time $O(1)$ and space $O(n)$ in the number of input classes
- When a coreferenced node is definitively "published," an $O(n)$ walk of the list writes all of its inward arcs
  - This is trivially deferred until unification success is known

agree: n -way full implementation notes

- The *agree* implementation is vastly complicated by simultaneously implementing the parse restrictor, so that restricted nodes are never written in the first place
  - Only referring nodes in non-restricted areas are recorded in the class
  - Traversal into restriction is still required, so writing is switched off when entering restriction—but then back on when popping out of any coreference that is not subject to restriction
  - The re-enabling case is detected by the presence of > 0 referring nodes
- Sharing the invariant tally set amongst rule daughters is ok, but may lead to reentrancy tallies of ‘1’, which can be ignored

Unification in DELPH-IN parsers

- The LKB and PET use Tomabechi's method
- van Lohuizen (2000) made some modifications to PET to support concurrent unification
  - is this version still supported?
- *agree* is a new parser supporting DELPH-IN research standards
- *agree* supports two unifier test configurations
  - incremental (duplex) unifier
  - new n-way unifier (with running type carry)
  - both are thread-safe, supporting intrinsic concurrency when or if the parser initiates multiple tasks

Evaluating n-way unification

- When testing diverse parsers, it is not possible to decisively control for performance of the unification algorithm alone
- Comparative evaluation of distinct parsing systems is already notoriously difficult (Dridan 2010)
- This is true even with identical grammars and testsuites

  Uncontrolled variables include: operating system; programming language; compiler options; runtime environment; storage and access methods for GLBs, type hierarchy, and TFS; parser configuration options; and numerous internal parser implementation details, such as chart storage, chart access, etc.
- Therefore, conclusive evaluation of n-way unification requires intra-system testing
  - An incremental unifier is in place in the *agree* system (results today)
  - An in-system quasi-destructive unifier must be implemented (work underway)

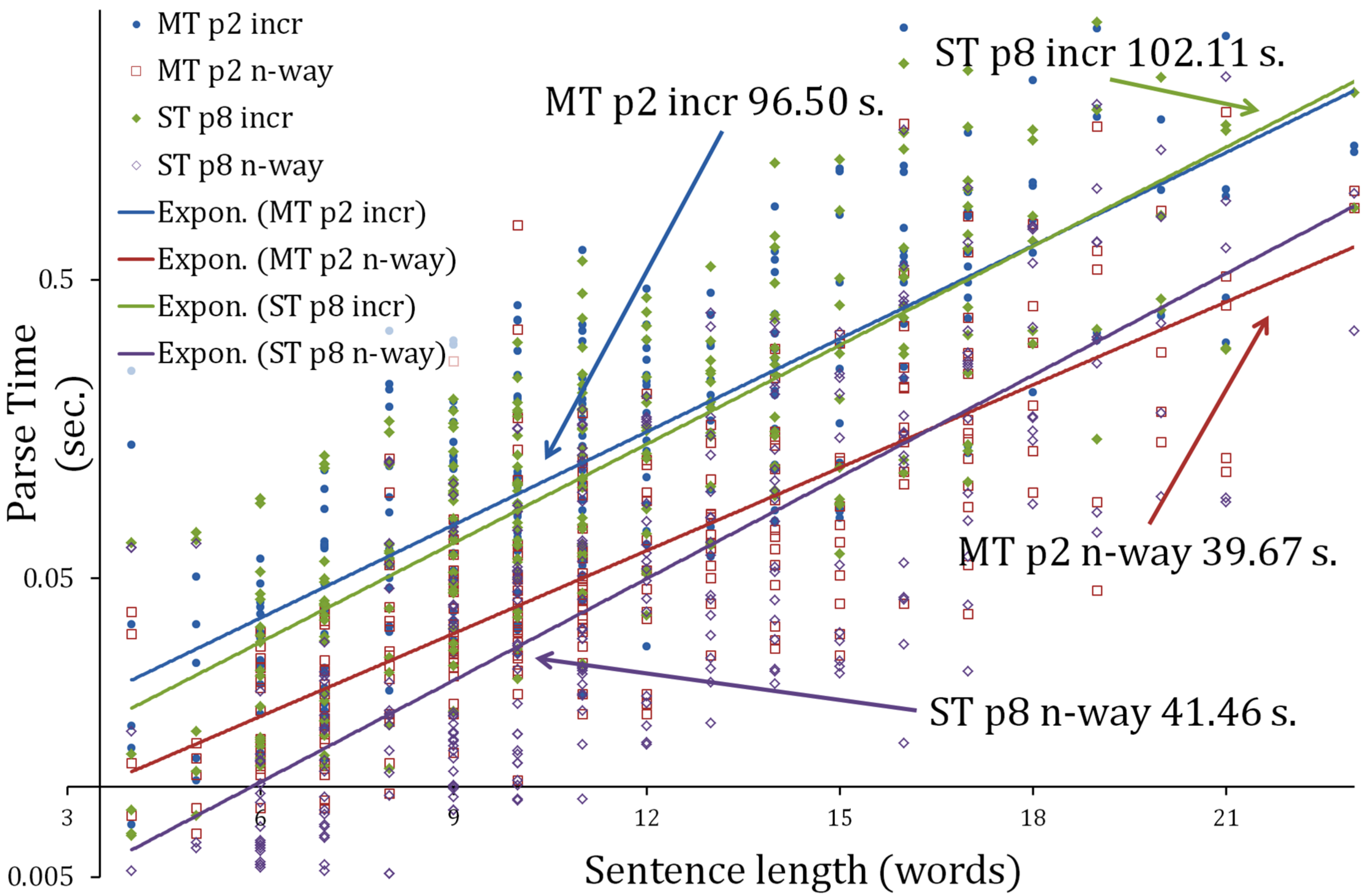
Evaluation methodology

- ERG rev. 8962
- ‘Hike’ corpus
  - except sentences containing numerals (287 sentences)

  http://www.agree-grammar.com/corpora/hike/hike-input-PET.txt
- Full packing, exhaustive unpacking
  - *agree* currently does not support parse selection
- Windows Server 2008 x64
- .NET 4.0
- gcServer
  - this is a more intrusive, but higher-performance garbage collector
- Hardware: 8-way (2 × Xeon 5460), 3.17GHz, 32GB

incremental duplex vs. n-way unification, $\log t$

agree parser: multi-threaded, pipeline 2 vs. single-threaded pipeline 8

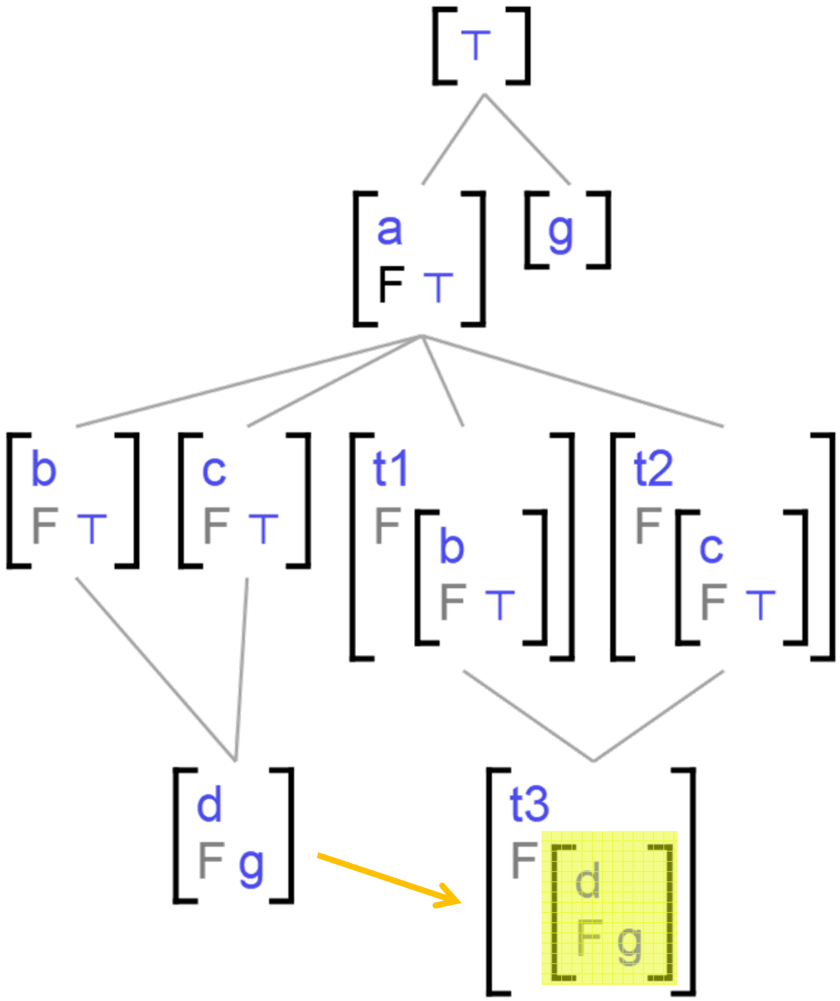


n-way: Opportunities in the parser

Although experiments evaluating the intrinsic performance of n-way unification continue, the algorithm does enable at least two intriguing operational benefits:

1. Simplified treatment, during unification, of well-formedness constraints
2. Synchronous unification of all rule-daughters during unpacking

Well-formed unification



Our formalism enforces *well-formedness* during unification. Because the type unification of *t1* and *t2* yields a third type, *t3*, unification must automatically introduce the canonical constraint on *t3* as well. Therefore, *t3* ends up with *g* for feature F, even though this constraint is specified by neither *t1* nor *t2*.

Evaluating well-formedness checks with n-way unification

- With the ERG 'Hike' corpus, well-formedness checks account for 1.41% of duplex unification time
  - This was measured using the *agree* incremental unifier but the result should apply in general
- When n-way naturally incorporates well-formed constraints, their provenance is lost
  - This is the aesthetic benefit of the method…
  - …but it essentially precludes direct measurement of the improvement
  - However, any improvement would likely be small
  - Therefore, evaluation of this effect was not pursued

n-way and Unpacking

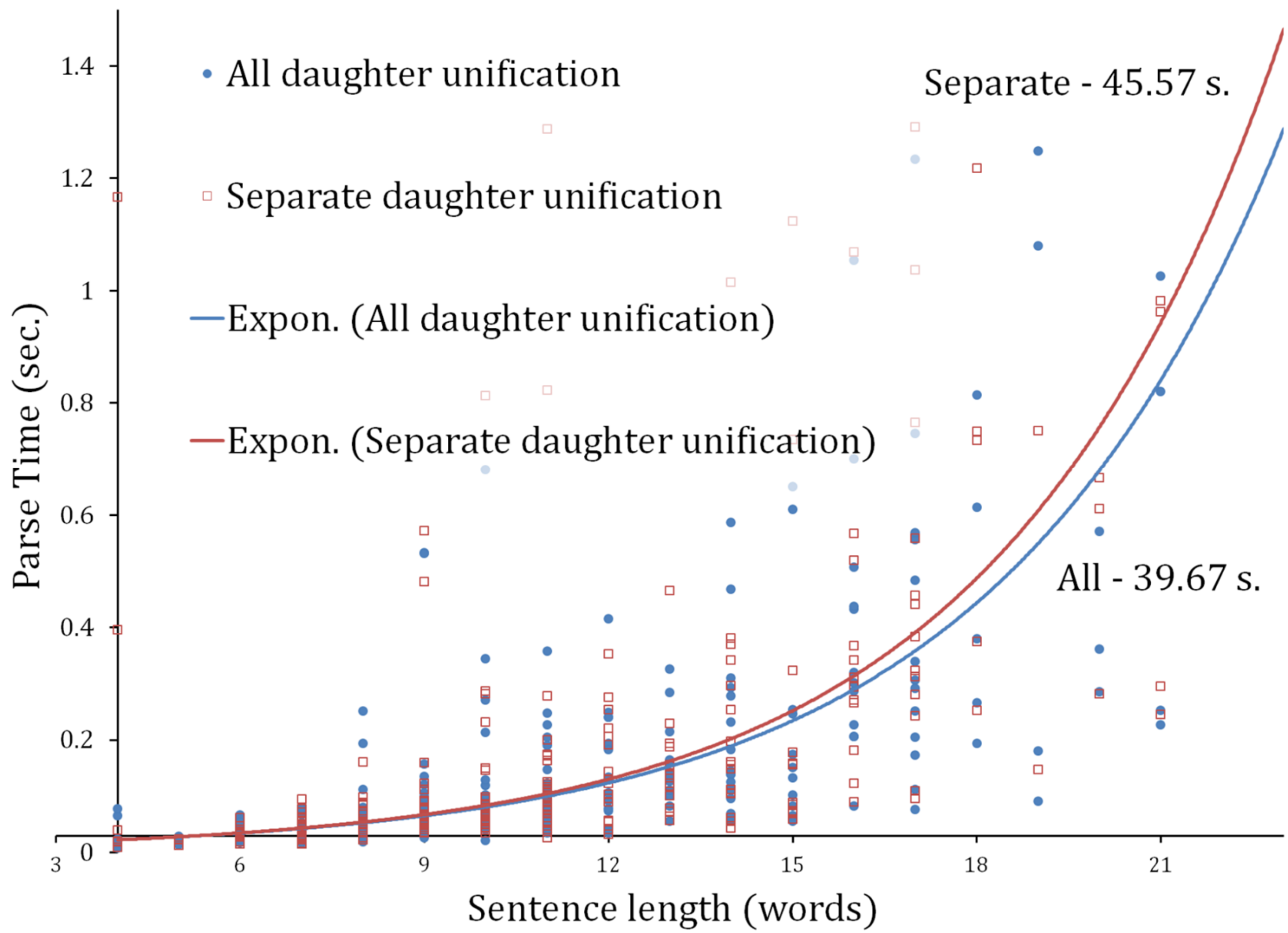
- n-way satisfaction checking trivially supports parse forest validating with $O(1)$ top-level unification operation per derivation
  - In practice, memoization at each level of the tree is desired, so $O(n)$ operations would be used per derivation
- Duplex unifiers require $O(n)$ in the rule arity *for each node* in the derivation tree
  - Memoization is not opt-out, so, $O(n^2)$ operations per derivation

Evaluating n-way synchronous unpacking

- Test n-way unification with and without synchronous unpacking, in the *agree* parser
- Synchronous unpacking was 13% faster over the whole corpus
- As expected, maximum improvement was for longer sentences, as high as 94%

Results: synchronous unpacking

agree parser, n-way unification, multi-threaded, pipeline 2



Future work

- Intra-system evaluation of n-way vs. quasi-destructive unification
  - implement Tomabechi (1991, 1992) method in *agree*
- Exploit aggressive structure sharing potential in n-way unification

agree parser – overview and eval presented at breakout tomorrow

Thank you!

References

Hassan Ait-Kaci, Robert Boyer, Patrick Lincoln, Roger Nasr. 1989. *Efficient Implementation of Lattice Operations*

Ulrich Callmeier. 2001. *Efficient Parsing with Large-Scale Unification Grammars*. MA Thesis, Universität des Saarlandes - Fachrichtung Informatik.

Ulrich Callmeier. 2000. PET: a platform for experimentation with efficient HPSG processing techniques. *Natural Language Engineering* 6(1): 99-107.

Rebecca Dridan. 2010. Using Lexical statistics to improve HPSG Parsing. PhD Thesis, Universität des Saarlandes.

M. Emele. (1991) “Unification with lazy non-redundant copying.” In Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics, Berkeley, CA, 323–330.

Dan Flickinger (2000). English Resource Grammar. In *Flickinger, Oepen, Tsujii, Uszkoreit, eds*.

References

Godden, K. (1990) “Lazy unification.” In Proceedings of the 28th Annual Meeting of the Association for Computational Linguistics, Pittsburgh, PA, 180–187.

Kogure, K. (1990) “Strategic lazy incremental copy graph unification.” In Proceedings of the 13th Conference on Computational Linguistics (COLING), Helsinki, Finland, 223–228.

Fernando C. N. Pereira. 1985. A structure-sharing representation for unification-based grammar formalisms. In *Proc. of the 23rd Annual Meeting of the Association for Computational Linguistics*. Chicago, IL, 8-12 July 1985, pages 137-144.

Hideto Tomabechi. 1991. Quasi-destructive graph unification. In *Proc. of the 29th Annual Meeting of the Association for Computational Linguistics*, Berkeley, CA.

References

Hideto Tomabechi. 1992. Quasi-destructive graph unifications with structure-sharing. In *Proc. of the 15th International Conference on Computational Linguistics (COLING-92)*, Nantes, France.

Hideto Tomabechi. 1995. Design of efficient unification for natural language. *Journal of Natural Language Processing*, 2(2):23-58.

Marcel P. van Lohuizen. 2000. Memory-efficient and Thread-safe Quasi-Destructive Graph Unification. *Proc. of the 38th Meeting of the Association for Computational Linguistics*.

David A. Wroblewski. 1987. Nondestructive graph unification. In *Proc. of the 6th National Conference on Artificial Intelligence (AAAI-87)*, 582-589. Morgan Kaufmann.