Fully Persistent Hash Tables

Eric Price, MIT
joint work with
Erik Demaine, MIT
Stefan Langerman,
Université Libre de Bruxelles
Functional Programming

• Haskell, ML, Scheme
• Use functional (read-only) data structures
  – Allows memoization, lazy evaluation
• Hard to make arbitrary data structures functional
  – Arrays and hash tables, in particular
  – Relax restriction to full persistence
Persistence Spectrum

- Partial persistence (version line)
  - Read old versions
- Full persistence (version tree)
  - Branch tree
  - Sufficient for mutation, lazy evaluation.
- Functional (no mutation)
Fully Persistent Arrays

- Prior work [Dietz 89]
- $O(\log \log MU)$ time per operation, $O(1)$ space
  - $M =$ size of version graph
  - $U =$ maximum index array
- Statically sized (in $M$ and $U$)
- Give $O(\log \log MN)$ static hash tables
  - $U = N^4$, pairwise independent hashing
    - Chance of collision is $1/N^2$, can rebuild.
- Want $O(\log \log N)$ dynamic hash tables
Outline

• Overview of static fully persistent arrays
• Technique for converting them to dynamic $O(\log \log N)$ hash tables.
Static Fully Persistent Arrays

• Van Emde Boas queue on (array index, version index)
  – Predecessor queries in $O(\log \log UM)$ time
  – Version indices in Eulerian tour of version graph

• Set($V$, $i$, $x$)
  – Creates new version $V'$ with index $a$ in Eulerian tour
  – Adds second copy of $V$ with index $b > a$ in tour
  – Insert $(i, a) = x$, $(i, b) = \text{value in cell } i \text{ in } V$. 
Example

- Tour:
  
  0 1 2 3 4 5 6 7 8 9 10
  1 2 3 2 4 2 1 5 6 5 1

- \((i, 0) = \text{NULL}\)
  \((i, 1) = x\)
  \((i, 6) = \text{NULL}\)
  \((i, 8) = y\)
  \((i, 9) = \text{NULL}\)

- More complicated to keep version indices ordered through insertion
Approach to improvement

- Dynamic sizing, $O(\log \log UM)$ to $O(\log \log UN) = O(\log \log U)$
- Size N array works for depth N/2 tree of size polynomial in N
- Rebuild array occasionally
  - Slow, but amortize
  - Amortization and persistence are tricky to mix
Trouble with amortization

• Suppose we rebuild whenever $\frac{3}{4}$ full.
  – Add elements to be just under $\frac{3}{4}$ full in version V.
  – Repeatedly add an element to version V.

• Results of amortization must spread out over large subtree.
Botany

- Bush (noun)
  - A small cluster of shrubs appearing as a single plant.
Once a shrub grows to some size, it flowers:

- Nodes become seeds of new shrubs
- Removed from enclosing bush to become its own bush
Augmented Data

- **Bush:**
  - *Pristine* DS P built for root and including seeds of shrubs
  - *Active* DS A built for root and including all nodes of shrubs

- **Shrub:**
  - List of updates in shrub

- **Operations act on A and append to the shrub list.**
Operation

• Shrub with seed size $N$ flowers at $N/5$ versions
  – Bush has $O(N^2)$ nodes, depth less than $N/2$
  – Pristine DS has $O(N)$ nodes, depth less than $N/5$.

• Flowering operation:
  – Rebuild pristine DS at the seed ($O(N)$ operations)
  – Insert shrub operations into DS ($N/5$ operations)
  – Run twice for new active, pristine data structures
  – Causes the creation of $\Omega(N)$ seeds.
Result of rebuilding technique

- Suppose we have a fully persistent DS that:
  - Can rebuild clean copy from $O(N)$ pristine DS and $O(N)$ update list in $O(N)$ operations
  - Clean copy works for $O(N^2)$ tree with depth $N/2$
- Then can run DS on arbitrary tree with the same time bounds in terms of $N$.
  - Fully persistent arrays
  - Fully persistent hash tables.