Immutability for Family History Data

Dr. Luther A. Tychonievich, Ph.D.
University of Virginia
Family History Information Standards Organisation

Outline

- **Introduction**
- Data Refactoring
- Immutable CRUD
- Benefits to Family History Software
- Immutable Data Communication
- Implementation outlines

Historical fiction

- In the beginning, there were two languages:
  - Lisp used mathematics-like variables: if x was 3, it was always 3. Obviously.
  - Fortran used assembly-like variables: $x = x + 1$ was not false, it was an "update" *changing* a variable's value. (huh?)
- Their kids had a fight, Fortran's kids won.
- Since then, most PL advances are adding Lisp back in:
  - Trees, recursive code, recursive data types, garbage collection, callbacks, dynamic typing, metaprogramming, strong typing…
- Mathematics-like variables haven't gained traction
- This talk: case for their semantics in data

Immutability of Data

- There is no update, no change of value
  - `struct`s must be created with all data in place
  - Implies acyclic pointers
- A.K.A "purely functional" or "persistent"
- Many benefits over mutability (more on this later)

Semantic Immutability

- To public interface, appears immutable
- May contain mutable caching, other invisible mutability
- Combines performance of mutability with robustness and other benefits of immutability

---

Immutable Data Organization

- Immutable data looks different:
  - Nested structures unwise
  - Mutual pointers impossible
  - Top-level objects generally small
- We'll see why when discussing CRUD…
- For now, let's look at what and how

Refactoring to be Immutable

- Consider the following data example:

```
p1- Person                p2- Person
  Name: Jane /Doe/          Name: John /Doe/
  Birth:                    Children:
    Date: 1789-01-23          p1
    Father: p2
```

- Nesting unwise if immutable; make pointer

```
p1- Person                p2- Person
  Name: Jane /Doe/          Name: John /Doe/
  Birth: e1                 Parent in:
e1- Birth                    e1
  Date: 1789-01-23
  Child: p1
  Father: p2
```

- Mutual pointers problematic; make link objects

```
p1- Person                p2- Person
  Name: Jane /Doe/          Name: John /Doe/

l1- p1 is child in e1     e1- Birth
l2- p2 is father in e1      Date: 1789-01-23
```

- Starting to look RDF-like…

How RDF-like should it be?

- Matter of preference
- At least remove mutual pointers
- Suggestion: principle of sensible disbelief
  - If you could disagree with part of an object, make that part its own object

---

Immutable CRUD

- CRUD - Create, Read, Update, Delete
- **Create:** simple
  - Never need to update data for consistency
  - Might need to update caches
- **Read:** Bit more work
  - Follow pointers, create aggregate view
  - Similar to SQL, RDF view processing
- Update
  - Two approaches: logging and full
  - **Logging-style update:**
    - Add a "set field x of y to be z" object
    - Similar to journaling
    - Inherits incoming edges automatically
    - More work on read
  - **Full update:**
    - Create a new object with the updated data
    - Add an "is update of" link object
    - Then either
      - Treat pointed-to-old-version as pointing-to-this on read; or
      - Update objects that pointed to old version
    - More data, but more flexible than logging
- **Delete:** just add "is deleted" object pointing to deleted object
  - Semantically sufficient
  - Use "not-deleted" filter on read
- **Purge:** remove purged object and purge all objects that used to point to it
  - For removing sensitive data
  - Or as garbage collection

Benefits to Family History Software

Versioning, but Better

- All old versions still exist
- History of changes integral to data
- But also: pointers are version-stable
  - Can't change from under me
  - Can use pointers as rationales, archival snapshots of reasoning, etc.

Distributed Processing

- Out of sync = has nodes the other does not
  - No mutual-change rectification needed
  - Resync = both share new data
- Content = identity, so can avoid ID allocation conflicts too
- Replication is free
- Most concurrency issues go away

Reducing Edit Wars

- Can share data with no chance someone changes it
- Disagreement results in both versions of data
  - Granularity: only disagreed-on details split
  - Visibility: others can see all opinions
- If users can white- or black-list their view of other's data, they get a "private tree" made entirely of shared data

Aside: about merges

- "The" question: "is that *my* John?"
- On a merge
  - Could update pointers to point to same person object
    - Reads simple, but IoUS problem
  - Could add "are same" node pointing to both
    - Undoable and transparent to user
    - Slightly complicates reads

Immutable Data Communication

Communication without IDs

- Because identity = content, IDs are optional
- No sending IDs simplifies merging and cross-tool communication
- Pointers in data can be replaced by local-to-communication indices, anchors, etc.

Sending

- Send node and all nodes it references, recursively
- By construction, references are acyclic, so the recursion will terminate
- Serialise pointers as local-to-sent-data references
- Sending pointed-to before pointed-from not necessary, but makes receipt single-pass

Receiving

- Import nodes that link to nothing first, then nodes that link to already-processed nodes
- Track import-ID → recipient-pointer
- For each imported node
  - Let n by the node, replacing IDs with pointers
  - If already have node with same value, use it
  - Otherwise, add to local store
  - Remember the new ID ↦ pointer

Filtering

- Sender can omit to send any node
  - e.g., for privacy or IP protection
  - also omit nodes pointing to omitted nodes
- Receiver can ignore any received node
  - e.g., for limited tool focus or support
  - e.g., because nodes were custom extensions
  - also ignore nodes pointing to ignored nodes

Implementation outlines

Efficient Implementation

- Underlying data = directed acyclic property graph
  - Need labelled directed edges
  - Outgoing edges fixed at node creation
  - Incoming edges increase over time
- The following should be fast
  - Iterate over incoming edges
  - Follow an edge (either direction)

Implementations

- **Graph DB**: store graphs natively
- **Relational DB**: edges in link table
- **Document DB**:
  - Outgoing = pointer (or ID if no pointers)
  - Incoming = per-document hidden, mutable cache of points-to-me pointers
  - Update caches on document creation/import
  - Cache not needed on export
- **In-memory OO**:
  - Outgoing = pointer fields
  - Incoming = one of
    - Per-object cache of points-to-me pointers
    - External cache
      `map<object, set<object>> incoming`
  - May want to cache (object, field) pairs instead of just objects

Summary

- Immutable data simplifies distribution, collaboration, and cross-tool communication
- Data organization might change, but in deterministic ways
- CRUD: C easy, R more involved, UD reversible; merge can be reversible too
- Immutability implementable in all major data stores (efficiently with mutable caching)