Data Representation Synthesis
PLDI’2011*, ESOP’12, PLDI’12*
CACM’12

Peter Hawkins, Stanford University (google)
Alex Aiken, Stanford University
Kathleen Fisher, Tufts
Martin Rinard, MIT
Mooly Sagiv, TAU

http://theory.stanford.edu/~hawkinsp/

* Best Paper Award
Shape Analysis

x = NULL

F

T

t = malloc(..);

t → next = x;

x = t

return x

empty
t

x
t

n

x
t

n

x

n

n

n

x

n

n

n

x

n

n

n

x

n

n

x

n

n

x

n

n

x
thttpd: Web Server

Representation Invariants:
1. $\forall n: \text{Map. } \forall v: \mathbb{Z}$. 
   \[ \text{table}[v] = n \implies n\rightarrow\text{index}=v \]

2. $\forall n: \text{Map}$. 
   \[ n\rightarrow\text{rc} = |\{n' : \text{Conn. } n'\rightarrow\text{file_data} = n\}| \]
```c
static void add_map(Map *m)
{
    int i = hash(m);
    ...
    table[i] = m ;
    ...
    m->index= i ;
    ...
    m->rc++;
}
```

**Representation Invariants:**
1. \( \forall n: \text{Map. } \forall v: \text{Z. } \) 
   \[ \text{table}[v] = n \Rightarrow \text{index}[n]=v \]
2. \( \forall n: \text{Map. } \) 
   \[ \text{rc}[n] = \{n' : \text{Conn . file_data}[n'] = n\} \]
Concurrent Data Structures

- Writing highly concurrent data structures is complicated
- Modern programming languages provide efficient concurrent collections with atomic operations
TOMCAT Motivating Example

TOMCAT 5.*

```
attr = new HashMap();
...
Attribute removeAttribute(String name){
    Attribute val = null;
    synchronized(attr) {
        found = attr.containsKey(name);
        if (found) {
            val = attr.get(name);
            attr.remove(name);
        }
    }
    return val;
}
```

Invariant: removeAttribute(name) returns the removed value or null if it does not exist
```
removeAttribute("A") {
    Attribute val = null;
    found = attr.containsKey("A");
    if (found) {
        val = attr.get("A");
    }
    attr.remove("A");
    return val;
}
```

**Invariant:** `removeAttribute(name)` returns the removed value or null if it does not exist
OOPSLA’11 Shacham

• Search for all public domain collection operations methods with at least two operations
• Used simple static analysis to extract composed operations
  – 29% needed manual modification
• Extracted **112** composed operations from **55** applications
  – Apache Tomcat, Cassandra, MyFaces – Trinidad, ...

• Check Linearizability of **all** public domain composed operations
Results: OOPSLA’11 Shacham

- 47% Unknown
- 38% Non Linearizable
- 15% Open Non Linearizable
Impact OOPSLA’11 Shacham

• Reported the bugs
  – Even bugs in open environment were fixed
• As a result of the paper the Java library was changed

“A preliminary version is in the pre-java8 "jsr166e" package as ConcurrentHashMapV8. We can't release the actual version yet because it relies on Java8 lambda (closure) syntax support. See links from http://gee.cs.oswego.edu/dl/concurrency-interest/index.html including: http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166edocs/jsr166e/ConcurrentHashMapV8.html

Good luck continuing to find errors and misuses that can help us create better concurrency components!”
Specifying and Verifying Data Structure Composition

- Efficient libraries are widely available
- Composing operations in a way which guarantee correctness:
  - Specification
  - Verification
  - Synthesis
  - Performance
  - Handle concurrency
Research Questions

• How to compose several data structures?
  – Support shared data structures
• Hide the complexity of concurrent programming
• Provably correct code
• Simpler program reasoning
Composing Data Structures

```
filesystems
  filesystem=1
    s_list
    s_files
    file=14
      f_list
      f_fs_list
  filesystem=2
    s_list
    s_files
    file=7
      f_list
      f_fs_list
    file=5
      f_list
      f_fs_list
    file=6
      f_list
      f_fs_list
  file=2
    f_list
    f_fs_list
```
Problem: Multiple Indexes

Access Patterns

- Find all mounted filesystems
- Find cached files on each filesystem
- Iterate over all used or unused cached files in Least-Recently-Used order

+ Concurrency
Disadvantages of linked shared data structures

• Error prone
• Hard to change
• Performance may depend on the machine and workload
• Hard to reason about correctness
  – Low level representation invariants

• Concurrency makes it harder
  – Lock granularity
  – Aliasing
Our thesis

• Very high level programs
  – No pointers and shared data structures
  – Easier programming
  – Simpler reasoning
  – Machine independent

• The compiler generates pointers and multiple concurrent shared data structures

• Performance comparable to manually written code
Our Approach

• Program with “database”
  – States are tables
  – Uniform relational operations
    • Hide data structures from the program
    – Functional dependencies express program invariants
• The compiler generates low level shared pointer data structures with concurrent operations
  – Correct by construction
• The programmer can tune efficiency
• Autotuning for a given workload
Conceptual Programming Model

query
... 
insert
... 
remove

shared database

insert
query
... 
insert
... 
remove
...
Relational Specification

- Program states as relations
  - Columns correspond to properties
  - Functional dependencies define global invariants

<table>
<thead>
<tr>
<th>Atomic Operation</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>r= empty</code></td>
<td><code>r := {}</code></td>
</tr>
<tr>
<td><code>insert r s t</code></td>
<td><code>if s \notin r \text{ then } r = r \cup \{&lt;s,t&gt;\}</code></td>
</tr>
<tr>
<td><code>query r S C</code></td>
<td>The C of all the tuples in r matching tuple</td>
</tr>
<tr>
<td><code>remove r s</code></td>
<td>remove from r all the tuples which match s</td>
</tr>
</tbody>
</table>
The High Level Idea

Decomposition

Concurrent Compositions of Data Structures, Atomic Transactions

Compiler

RelScala

\( \{fs, file, inuse\} \)

\( fs, file \rightarrow inuse \)

query <inuse:T> \{fs, file\}

Scala

List * query(FS* fs, File* file) {
  lock(fs) ; for (q= file_in_use; ...) 
  ....
Filesystem

- Three columns \{fs, file, inuse\}
- $fs: \text{int} \times file: \text{int} \times inuse: \text{Bool}$
- Functional dependencies
  - $\{fs, file\} \rightarrow \{inuse\}$

<table>
<thead>
<tr>
<th>fs</th>
<th>file</th>
<th>inuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>T</td>
</tr>
</tbody>
</table>
Filesystem (operations)

<table>
<thead>
<tr>
<th>fs</th>
<th>file</th>
<th>inuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>F</td>
</tr>
</tbody>
</table>

query <inuse:T> {fs, file }=

[<fs:2, file:7>, <fs:1, file:6>]
## Filesystem (operations)

<table>
<thead>
<tr>
<th>fs</th>
<th>file</th>
<th>inuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>F</td>
</tr>
</tbody>
</table>

insert `<fs:1, file:15> <inuse:T>`

<table>
<thead>
<tr>
<th>fs</th>
<th>file</th>
<th>inuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>T</td>
</tr>
</tbody>
</table>
**Filesystem (operations)**

<table>
<thead>
<tr>
<th>fs</th>
<th>file</th>
<th>inuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>T</td>
</tr>
</tbody>
</table>

remove <fs:1>

<table>
<thead>
<tr>
<th>fs</th>
<th>file</th>
<th>inuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>F</td>
</tr>
</tbody>
</table>
Plan

• Compiling into sequential code (PLDI’11)
• Adding Locks concurrency (PLDI’12)
Mapping Relations into Low Level Data Structures

• Many mappings exist
• How to combine several existing data structures
  – Support sharing
• Maintain the relational abstraction
• Reasonable performance
• Parametric mappings of relations into shared combination of data structures
  – Guaranteed correctness
The RelC Compiler

Relational Specification

\( fs \times file \times inuse \)
\{fs, file\} → \{inuse\}

foreach \(<fs, file, inuse> \in \text{filesystems}\) s.t. \(fs = 5\)
do ...

Graph decomposition

RelC

C++
Decomposing Relations

• Represents subrelations using container data structures
• A directed acyclic graph (DAG)
  – Each node is a sub-relation
  – The root represents the whole relation
  – Edges map columns into the remaining sub-relations
  – Shared node = shared representation
Decomposing Relations into Functions

Currying

\[
\begin{align*}
& \text{fs} \times \text{file} \times \text{inuse} \\
& \{\text{fs, file}\} \rightarrow \{\text{inuse}\} \\
& \text{group-by} \{\text{fs}\} \\
& \text{INUSE} \rightarrow \text{FS} \times \text{FILE} \\
& \text{FILE} \rightarrow \text{INUSE} \\
& \text{FILE} \rightarrow \text{INUSE} \\
& \text{FS} \rightarrow (\text{FILE} \rightarrow \text{INUSE}) \\
& \text{INUSE} \rightarrow (\text{FS} \times \text{FILE} \rightarrow \text{INUSE})
\end{align*}
\]
Filesystem Example

\[
\{\text{fs, file, inuse}\}
\]

\[
\begin{array}{c|c|c}
\text{fs} & \text{file} & \text{inuse} \\
1 & 14 & F \\
2 & 7 & T \\
2 & 5 & F \\
1 & 6 & T \\
1 & 2 & F \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{fs:1} & \text{file} & \text{inuse} \\
14 & F & \\
6 & T & \\
2 & F & \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{fs:2} & \text{file} & \text{inuse} \\
7 & T & \\
5 & F & \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{file:14} & \text{inuse} & \\
F & \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{file:6} & \text{inuse} & \\
T & \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{file:2} & \text{inuse} & \\
F & \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{file:7} & \text{inuse} & \\
T & \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{file:5} & \text{inuse} & \\
F & \\
\end{array}
\]
Memory Decomposition (Left)

Diagram showing memory decomposition with nodes labeled "fs", "inuse", "file", and "fs, file". The diagram consists of red and green nodes, with additional file labels such as "file:14", "file:6", "file:2", "file:7", and "file:5".
{fs, file} \rightarrow \{ \text{inuse} \}
Memory Decomposition (Right)

\{fs, file\} \rightarrow \{ inuse\}

- \text{fs:2} file:7 \text{ inuse:T}
- \text{fs:1} file:6 \text{ inuse:T}
- \text{fs:1} file:14 \text{ inuse:F}
- \text{fs:2} file:5 \text{ Inuse:F}
- \text{fs:1} file:2 \text{ Inuse:F}
Decomposition Instance

fs × file × inuse

\{fs, file\} \rightarrow \{inuse\}

<table>
<thead>
<tr>
<th>fs</th>
<th>file</th>
<th>inuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>F</td>
</tr>
</tbody>
</table>
Decomposition Instance

\[
\text{fs} \times \text{file} \times \text{inuse} \\
\{\text{fs, file}\} \rightarrow \{\text{inuse}\}
\]

<table>
<thead>
<tr>
<th>fs</th>
<th>file</th>
<th>inuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>F</td>
</tr>
</tbody>
</table>
Decomposing Relations Formally (PLDI’11)

\[ \text{let } w : \{ \text{fs, file, inuse} \} \triangleright \{ \text{inuse} \} = \{ \text{inuse} \} \text{ in} \]

\[ \text{let } y : \{ \text{fs} \} \triangleright \{ \text{file, inuse} \} = \{ \text{file} \} \rightarrow^{\text{list}} \{ w \} \text{ in} \]

\[ \text{let } z : \{ \text{inuse} \} \triangleright \{ \text{fs, file, inuse} \} = \{ \text{fs, file} \} \rightarrow^{\text{list}} \{ w \} \text{ in} \]

\[ \text{let } x : \{ \} \triangleright \{ \text{fs, file, inuse} \} = \{ \text{fs} \} \rightarrow^{\text{clist}} \{ y \} \equiv \]

\[ \{ \text{inuse} \} \rightarrow^{\text{array}} \{ z \} \]
Memory State

<table>
<thead>
<tr>
<th>fs</th>
<th>file</th>
<th>inuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>F</td>
</tr>
</tbody>
</table>

fs × file × inuse

{fs, file} →{ inuse}
Memory State(2)

{fs, file} \rightarrow \{\text{inuse}\}

<table>
<thead>
<tr>
<th>fs</th>
<th>file</th>
<th>inuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>F</td>
</tr>
</tbody>
</table>
Adequacy

Not every decomposition is a good representation of a relation.

A decomposition is *adequate* if it can represent every possible relation matching a relational specification.

enforces sufficient conditions for adequacy
Adequacy of Decompositions

• All columns are represented
• Nodes are consistent with functional dependencies
  – Columns bound to paths leading to a common node must functionally determine each other
Respect Functional Dependencies

file, fs

inuse

✓ \{file, fs\} \rightarrow \{inuse\}
Adequacy and Sharing

Columns bound on a path to an object $x$ must functionally determine columns bound on any other path to $x$

✓ $\{fs, file\} \leftrightarrow \{inuse, fs, file\}$
Adequacy and Sharing

Columns bound on a path to an object \( x \) must functionally determine columns bound on any other path to \( x \)

\[ \{\text{fs, file}\} \leftrightarrow \{\text{inuse, fs}\} \]
Sequential Compositions of Data Structures
foreach <fs, file, inuse> ∈ filesystems
  if inuse=T do ...

Cost proportional to the number of files
foreach <fs, file, inuse>ε filesystems
if inuse=T do ...

Cost proportional to the number of files in use
Completeness

- The representation is adequate $\rightarrow$ the compiler can always generate correct code
- But the code may be slow

\[
\text{foreach } \langle \text{fs, file, inuse} \rangle \in \text{filesystems s.t. } \text{fs}=1 \text{ do}
\]
Removal and graph cuts

<table>
<thead>
<tr>
<th>fs</th>
<th>file</th>
<th>inuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>F</td>
</tr>
</tbody>
</table>
Abstraction Theorem

• If the programmer obeys the relational specification and the decomposition is **adequate** and if the individual containers are correct

• Then the generated low-level code maintains the relational abstraction
Simplified Compilation Strategy

• Specify provably correct program transformations
• Select the best compiled code using a workload
Autotuner

• Given a fixed set of primitive types
  – list, circular list, doubly-linked list, array, map, ...
• A workload
• Exhaustively enumerate all the adequate decompositions up to certain size
• The compiler can automatically pick the best performing representation for the workload
Directed Graph Example (DFS)

- **Columns**
  - \( \text{src} \times \text{dst} \times \text{weight} \)

- **Functional Dependencies**
  - \( \{ \text{src}, \text{dst} \} \rightarrow \{ \text{weight} \} \)

- **Primitive data types**
  - map, list

---

![Directed Graph Example](image-url)
Synthesizing Concurrent Programs

PLDI’12
The High Level Idea

Concurrent Decomposition

Concurrent Compositions of Data Structures, Atomic Transactions

Compiler

Scala

List * query(FS* fs, File* file) {
    lock(...) for (q= file_in_use; ...)
    ....

RelScala

\{fs, file, inuse\}

fs, file \rightarrow inuse

query <inuse:T> \{fs, file\}
Two-Phase Locking

Attach a lock to each piece of data

Two phase locking protocol:

- **Well-locked**: To perform a read or write, a thread must hold the corresponding lock
- **Two-phase**: All lock acquisitions must precede all lock releases

**Theorem** [Eswaran et al., 1976]: Well-locked, two-phase transactions are serializable
Two Phase Locking

Decomposition

Attach a lock to every edge

Two Phase Locking $\Rightarrow$ Serializability

**Problem 1:** Can’t attach locks to container entries

**Problem 2:** Too many locks

Butler Lampson/David J. Wheeler: “Any problem in computer science can be solved with another level of indirection.”
1. Attach locks to nodes
2. Use a lock placement $\psi$ to map data (on edges) to locks (on nodes)
Coarse-Grained Locking

Decomposition

Decomposition Instance

\[ \psi = \{ uv \leadsto u, v w \leadsto u \} \]
Finer-Grained Locking

Decomposition

Decomposition Instance

\[ \psi = \{ uv \mapsto u, vw \mapsto v \} \]
Lock Striping

Decomposition

\[ \psi = \{ uv_x \mapsto u_x \mod k, \; vw \mapsto v \} \]
Lock Placements: Domination

Locks must dominate the edges they protect

Decomposition

Decomposition Instance
Lock Placements: Path-Closure

All edges on a path between an edge and its lock must share the same lock

If $\psi(vw) = u$, then $\psi(uv) = u$ also.
Lock Ordering

Prevent deadlock via a topological order on locks

\[ t < u < v < w \]
Queries and Deadlock

Query plans must acquire the correct locks in the correct order

\[ t < u < v < w \]

1. acquire(t)
2. lookup(tv)
3. acquire(v)
4. scan(vw)

Example: find files on a particular filesystem
Deadlock and Aliasing

\[
\begin{align*}
\{ & \text{lock(a1)} \\
& \text{lock(b1)} \\
& \text{// do something} \\
& \text{unlock(b1)} \\
& \text{unlock(a1)} \\
\} \\
\{ & \text{lock(a2)} \\
& \text{lock(b2)} \\
& \text{// do something} \\
& \text{unlock(b2)} \\
& \text{unlock(a2)} \\
\}
\end{align*}
\]
Decompositions and Aliasing

• A decomposition is an abstraction of the set of potential aliases
• Example: there are exactly two paths to any instance of node $w$
Concurrent Synthesis (Autotuner)

Find optimal combination of

- Decomposition
- Container Data Structures
- Lock Placement

- Lock Implementations
- Lock Striping Factors
Concurrent Graph Benchmark

\{src, dst, weight\}

src, dst \rightarrow weight

- Start with an empty graph
- Each thread performs $5 \times 10^5$ random operations
- Distribution of operations a-b-c-d (a% find successors, b% find predecessors, c% insert edge, d% remove edge)
- Plot throughput with varying number of threads

Based on Herlihy’s benchmark of concurrent maps
Results: 35-35-20-10

35% find successor, 35% find predecessor, 20% insert edge, 10% remove edge

ConcurrentHashMap
HashMap

Throughput (ops/sec)

Number of Threads
(Some) Related Projects

- In-memory databases [DB-toaster, Kemper, ...]
- SETL [Paige, Schwartz, Schonberg]
- Relational synthesis: [Cohen & Campbell 1993], [Batory & Thomas 1996], [Smaragdakis & Batory 1997], [Batory et al. 2000] [Manevich, 2012] ...
- Two-phase locking and Predicate Locking [Eswaran et al., 1976], Tree and DAG locking protocols [Attiya et al., 2010], Domination Locking [Golan-Gueta et al., 2011]
- Lock Inference for Atomic Sections: [McCloskey et al., 2006], [Hicks, 2006], [Emmi, 2007]
Further Work

• Synchronization with Foresight
  [G. Gueta, OOPSLA’11, PLDI’13, PPOPP’13’15]

• Combining Optimistic and Pessimistic
  Synchronization [PLDI’15]
Summary

• Programming with uniform relational abstraction
  – Increase the gap between data abstraction and low level implementation
• Comparable performance to manual code
• Easier to evolve
• Automatic data structure selection
• Easier for program reasoning