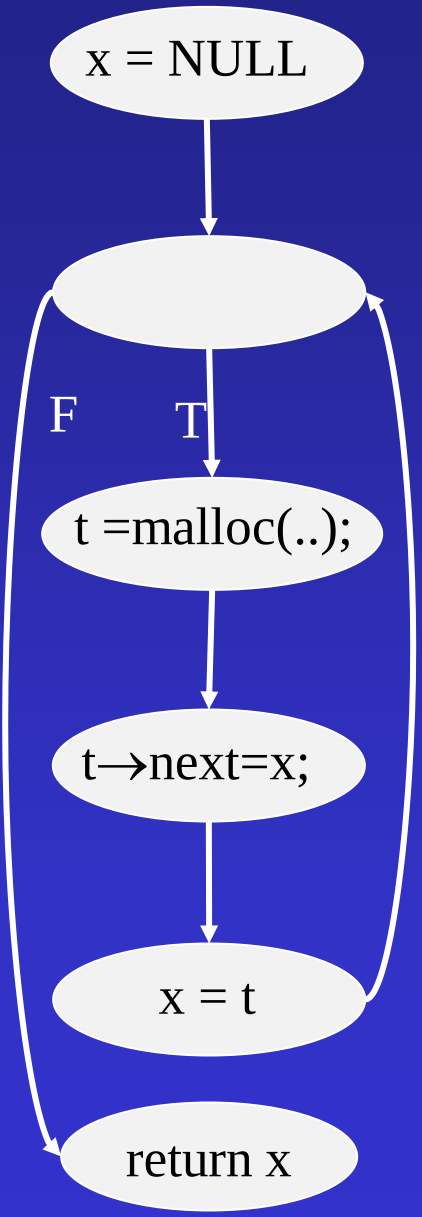
Data Representation Synthesis
PLDI'2011*, ESOP'12, PLDI'12*
CACM'12

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

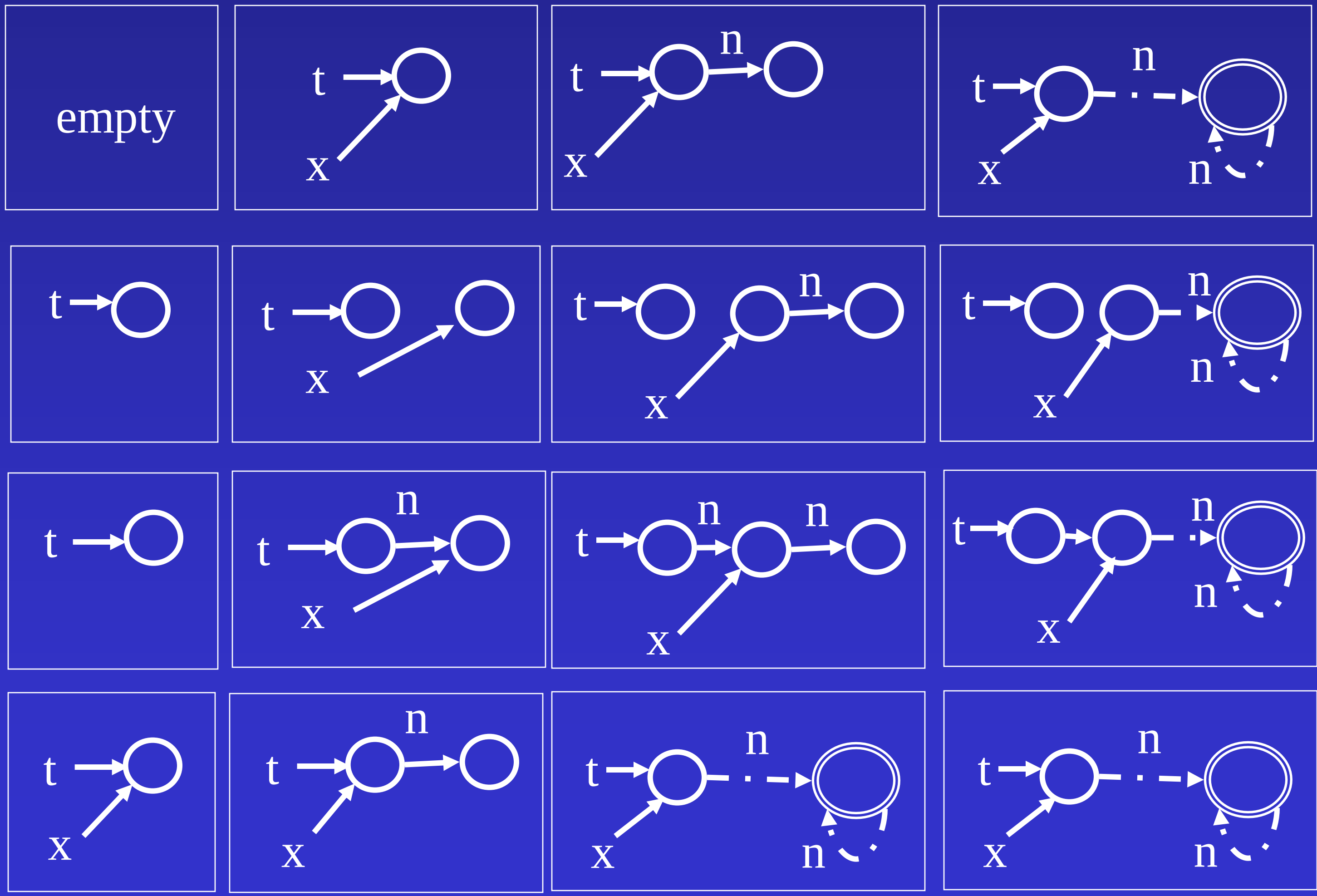
T

t =malloc(..);

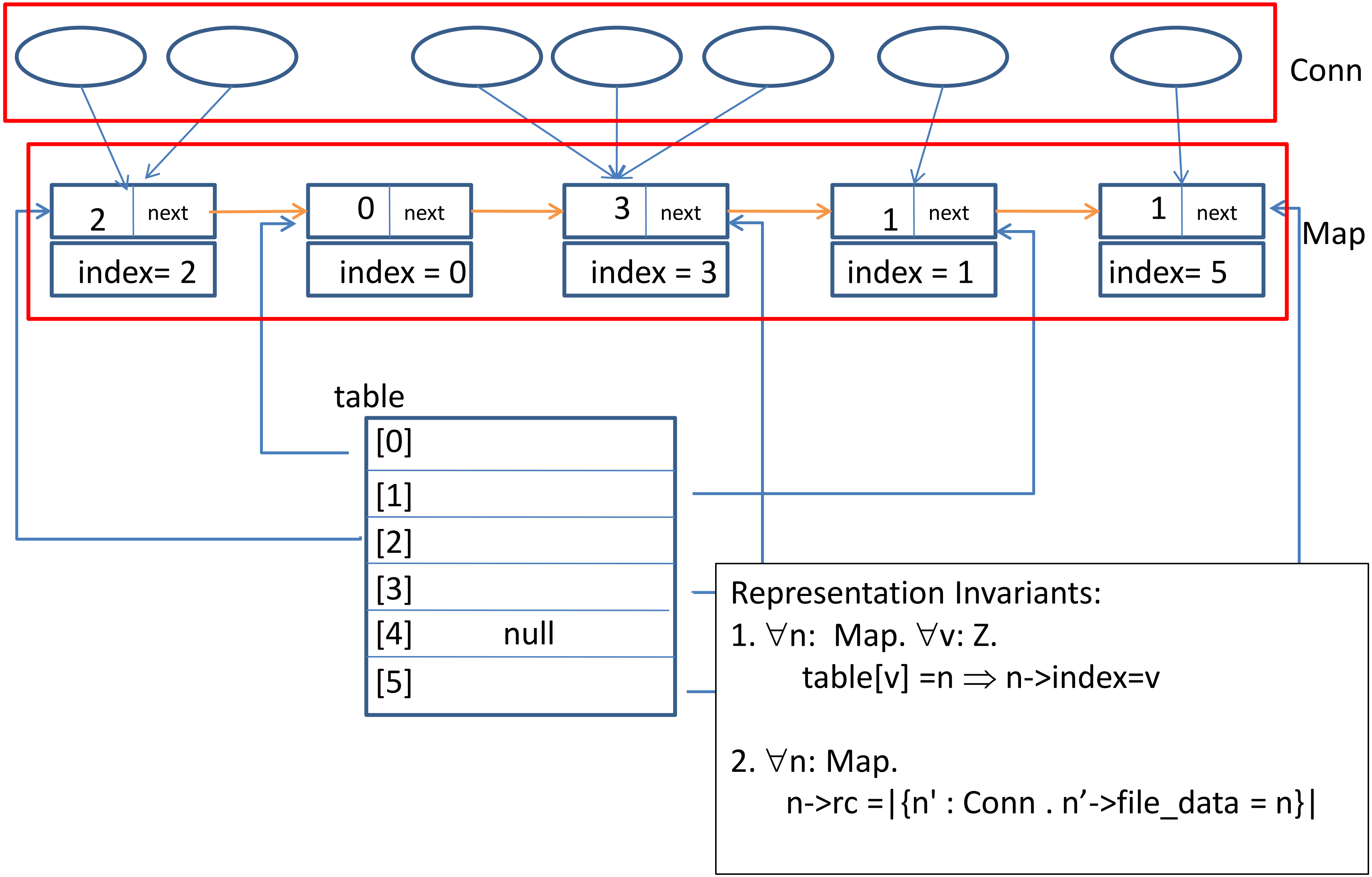
t→next=x;

x = t

return x



thttpd: Web Server

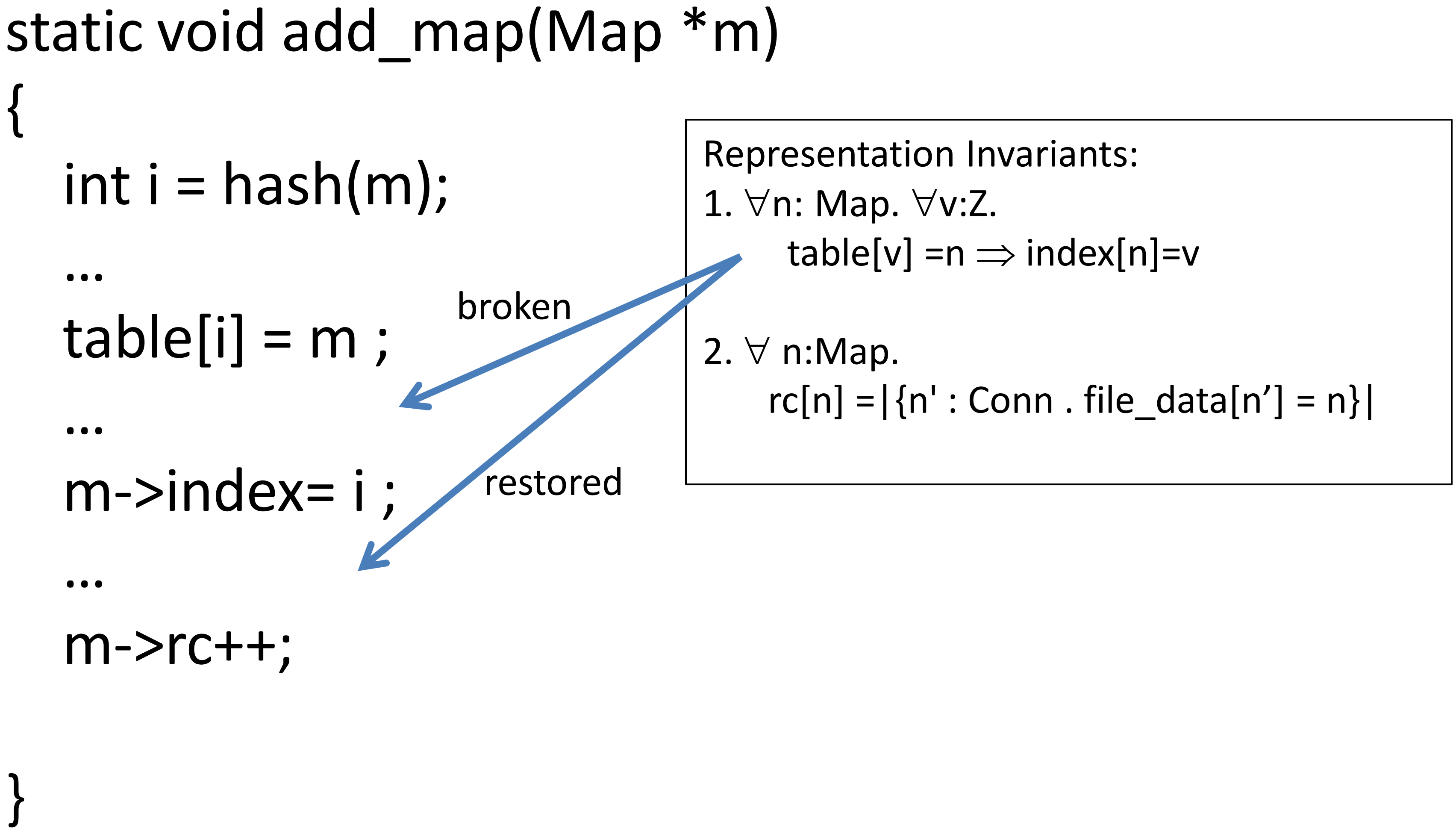


Representation Invariants:

1. $\forall n$: Map. $\forall v$: Z.
   table[v] =n $\Rightarrow$ n->index=v

2. $\forall n$: Map.
   n->rc =|{n' : Conn . n'->file_data = n}|

thttptd:mmc.c



```
static void add_map(Map *m)
{
    int i = hash(m);
    …
    table[i] = m ;
    …
    m->index= i ;
    …
    m->rc++;

}
```

broken

restored

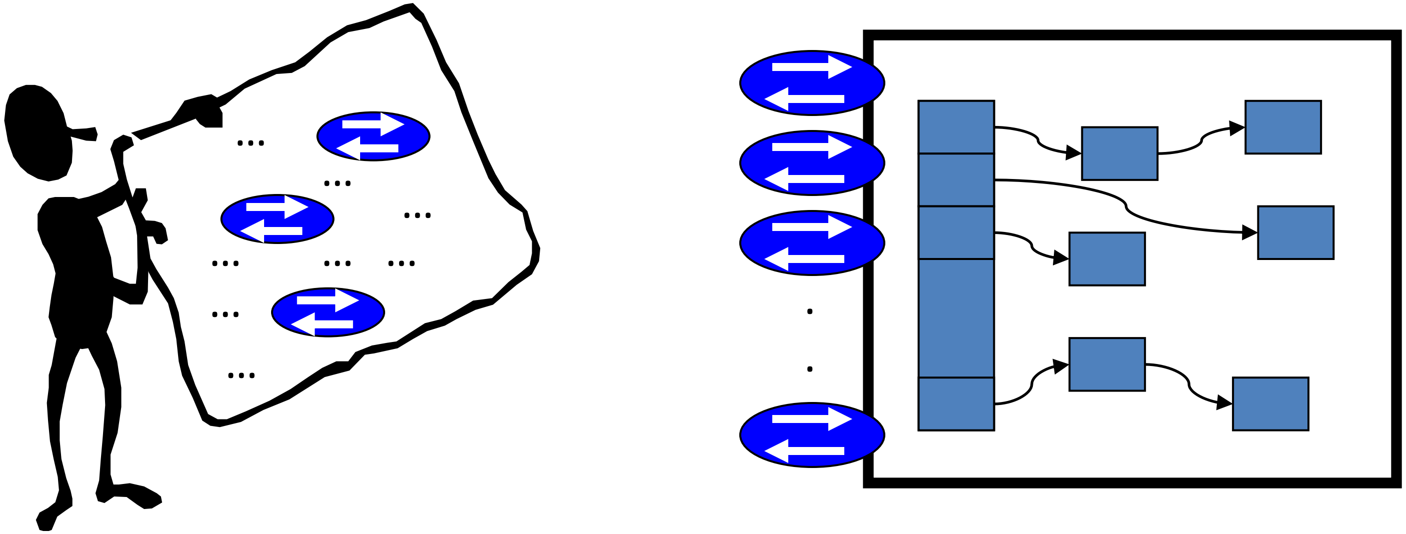
Representation Invariants:

1. $\forall n$: Map. $\forall v$:Z.
   table[v] =n $\Rightarrow$ index[n]=v

2. $\forall$ n:Map.
   rc[n] =|{n' : 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



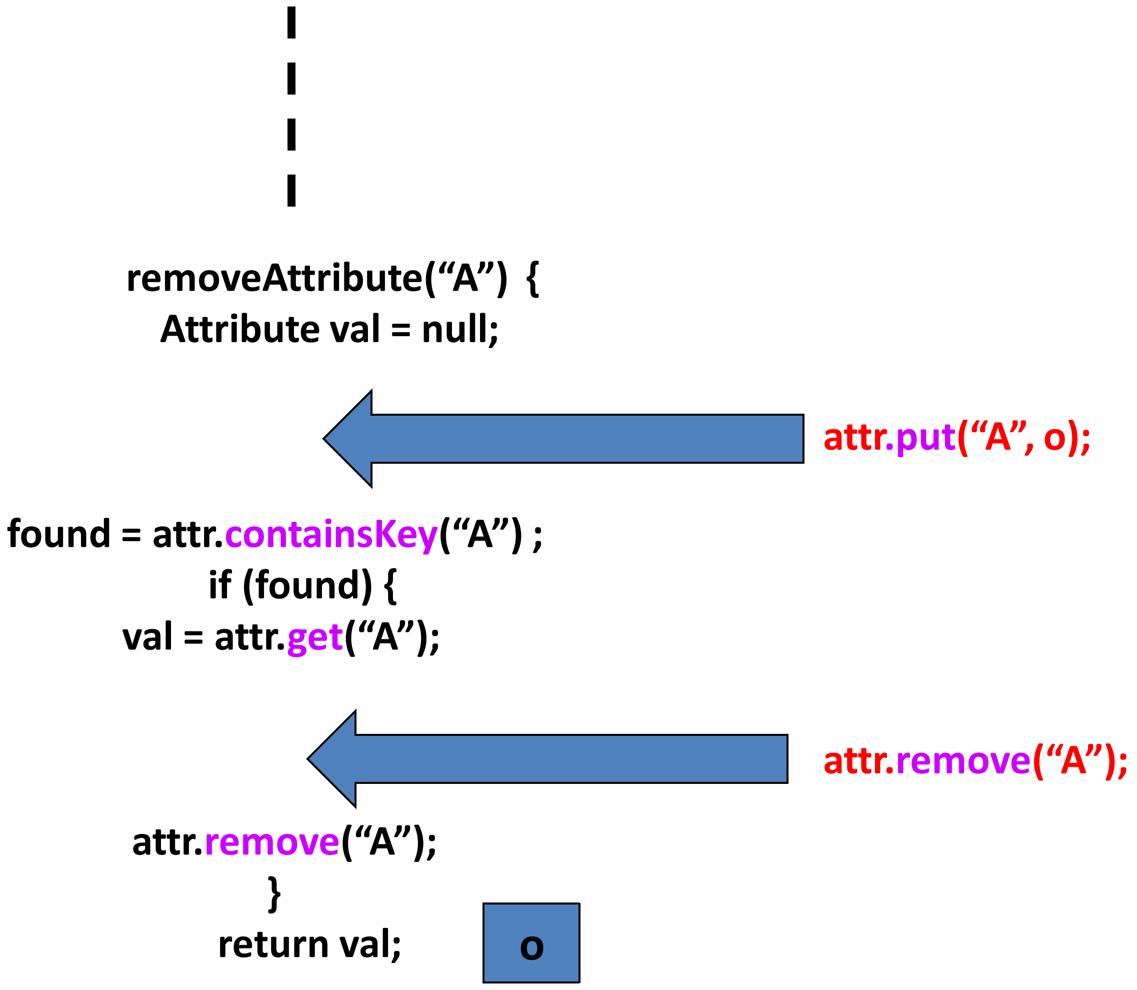
TOMCAT Motivating Example

TOMCAT 5.*

TOMCAT 6.*

```
attr = new HashMap();
attr = new ConcurrentHashMap();
                              ...
Attribute removeAttribute(String name){
  Attribute val = null;
  synchronized(attr) {
  /* 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;
```

```
attr.put("A", o);
```

```
found = attr.containsKey("A") ;
   if (found) {
   val = attr.get("A");
```

```
attr.remove("A");
```

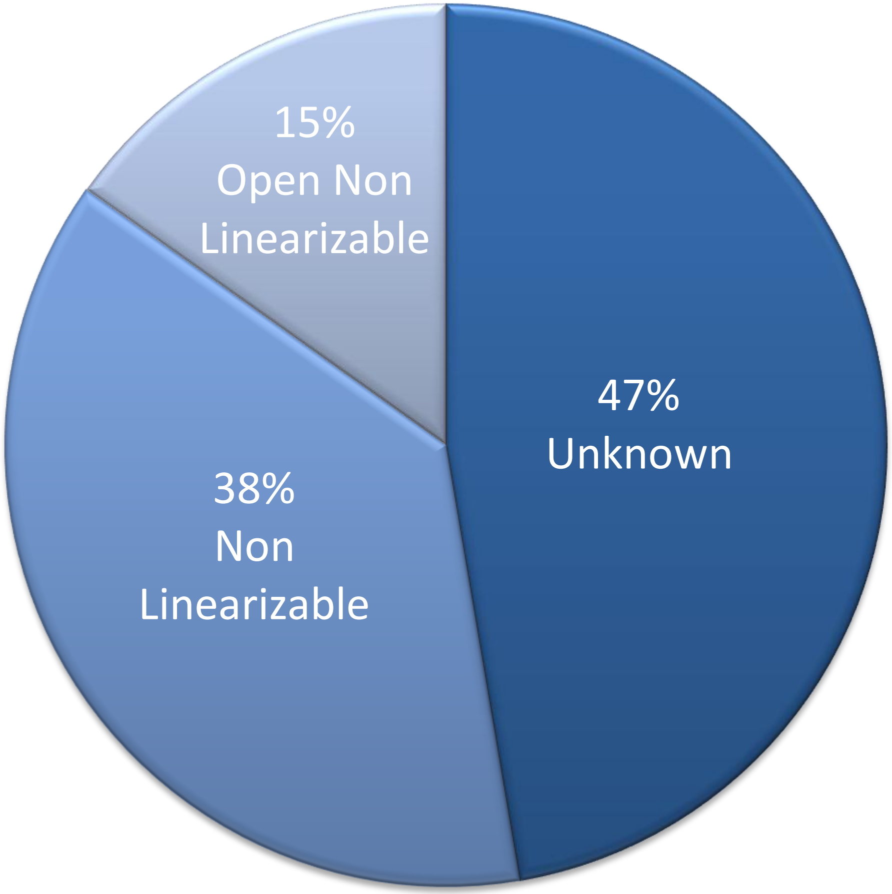
```
   attr.remove("A");
   }
   return val;
```

☒ *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



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!"

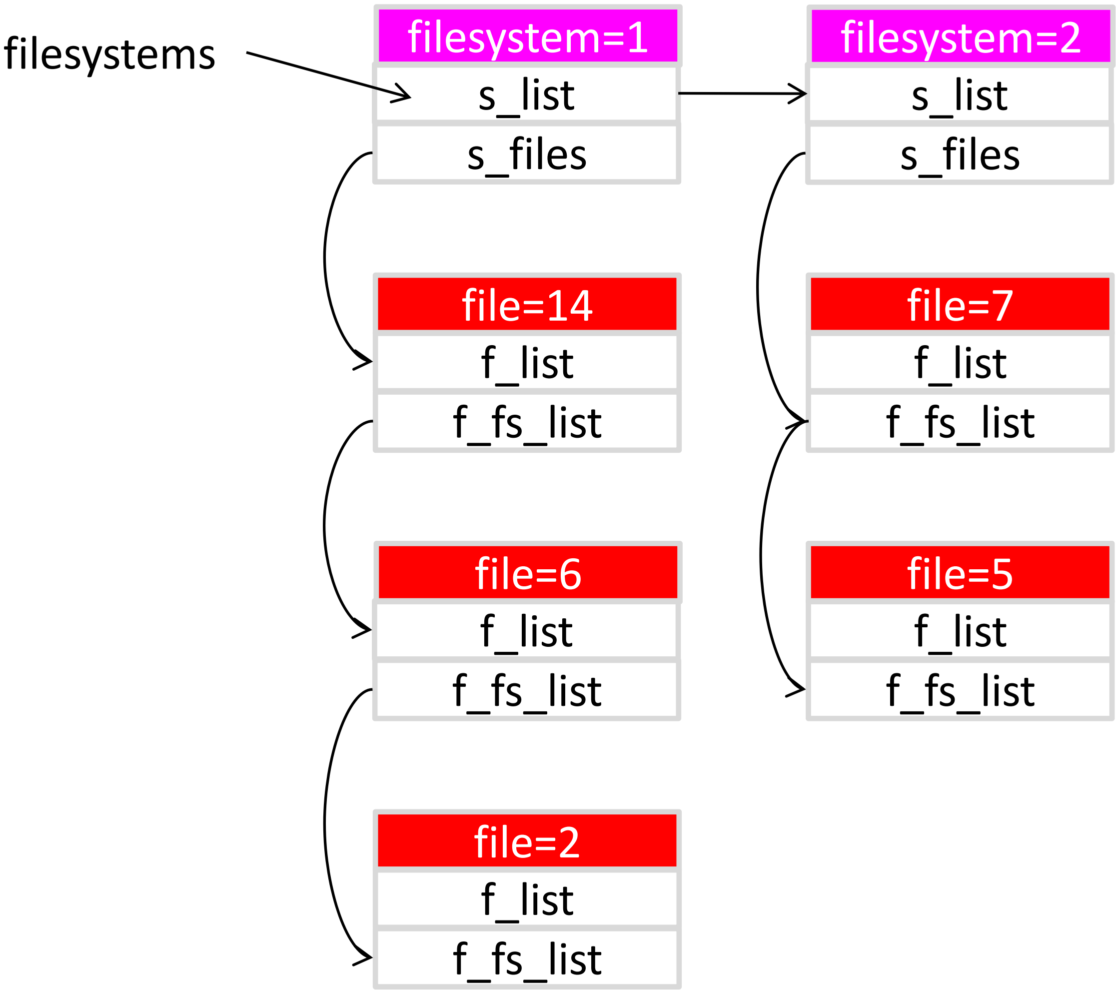
Specifying and Verifying Data Structure Composition

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

Research Questions

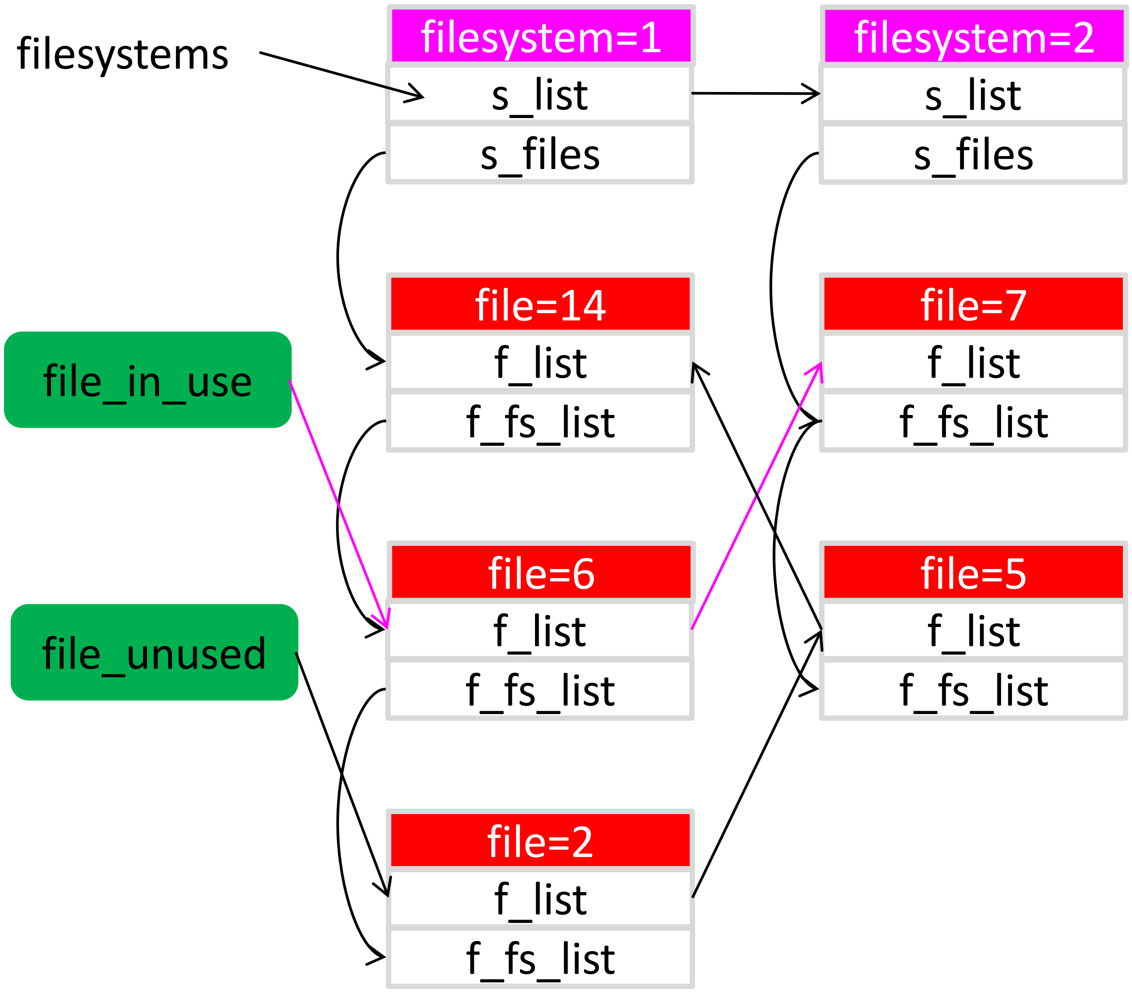
- How to compose several data structures?
  - Support shared data structures
- Hide the complexity of concurrent programming
- Provably correct code
- Simpler program reasoning

Composing Data Structures



Problem: Multiple Indexes

+Concurency



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

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

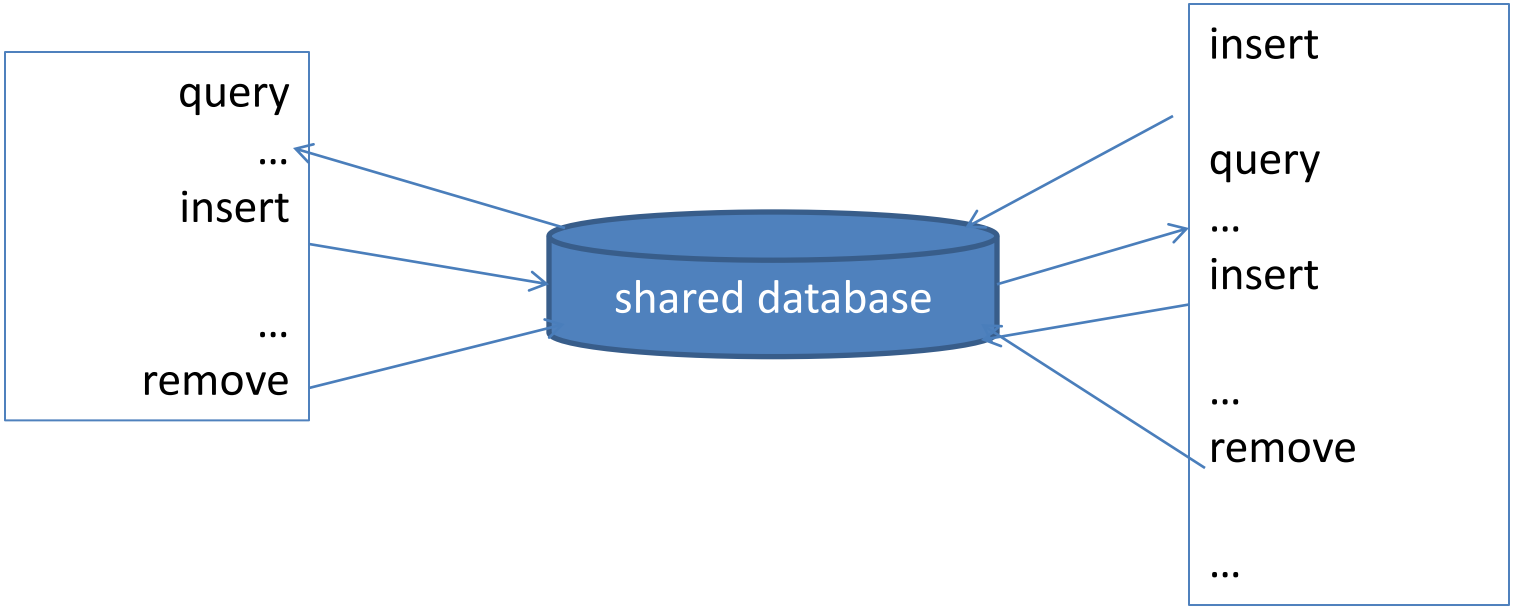
Our thesis

- 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

…

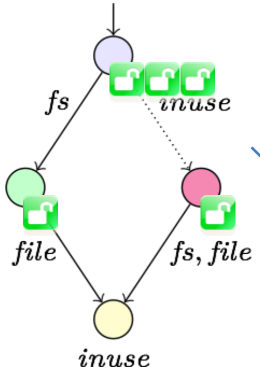
Relational Specification

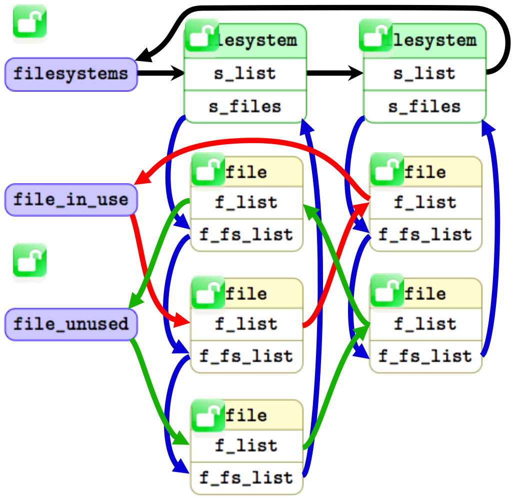
- Program states as relations
  - Columns correspond to properties
  - Functional dependencies define global invariants

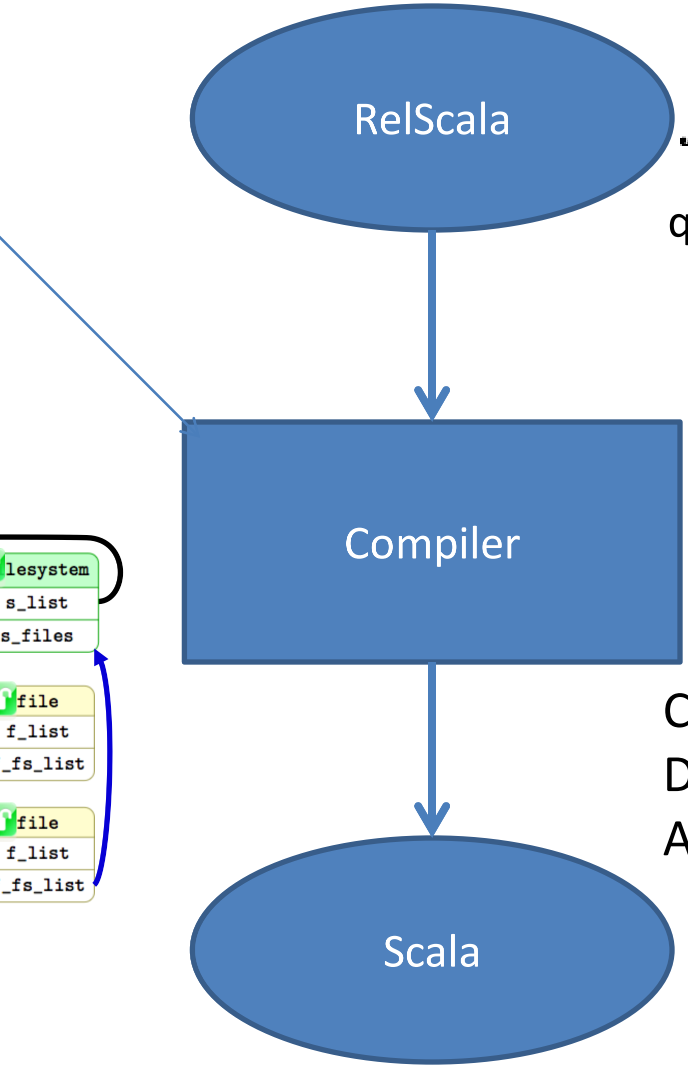
Atomic Operation	**meaning**
r= empty	r := {}
insert r s t	if s $\notin$r then r = r $\cup$ {<s.t>}
query r S C	The C of all the tuples in r matching tuple
remove r s	remove from r all the tuples which match s

The High Level Idea

Decomposition







RelScala

Compiler

Scala

$$\{fs, file, inuse\}$$

$$fs, file \rightarrow inuse$$

query <inuse:T> {fs, file}

Concurrent Compositions of Data Structures, Atomic Transactions

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

Filesystem

- Three columns {fs, file, inuse}
- fs:int × file:int × inuse:Bool
- Functional dependencies
  - {fs, file} → { inuse}

fs	file	inuse
1	14	F
2	7	T
2	5	F
1	6	T
1	2	F
1	2	T

Filesystem (operations)

fs	file	inuse
1	14	F
2	7	T
2	5	F
1	6	T
1	2	F

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

[<fs:2, file:7>, <fs:1, file:6>]

Filesystem (operations)

fs	file	inuse
1	14	F
2	7	T
2	5	F
1	6	T
1	2	F

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

fs	file	inuse
1	14	F
2	7	T
2	5	F
1	6	T
1	2	F
1	15	T

Filesystem (operations)

fs	file	inuse
1	14	F
2	7	T
2	5	F
1	6	T
1	2	F
1	15	T

remove <fs:1>

fs	file	inuse
2	7	T
2	5	F

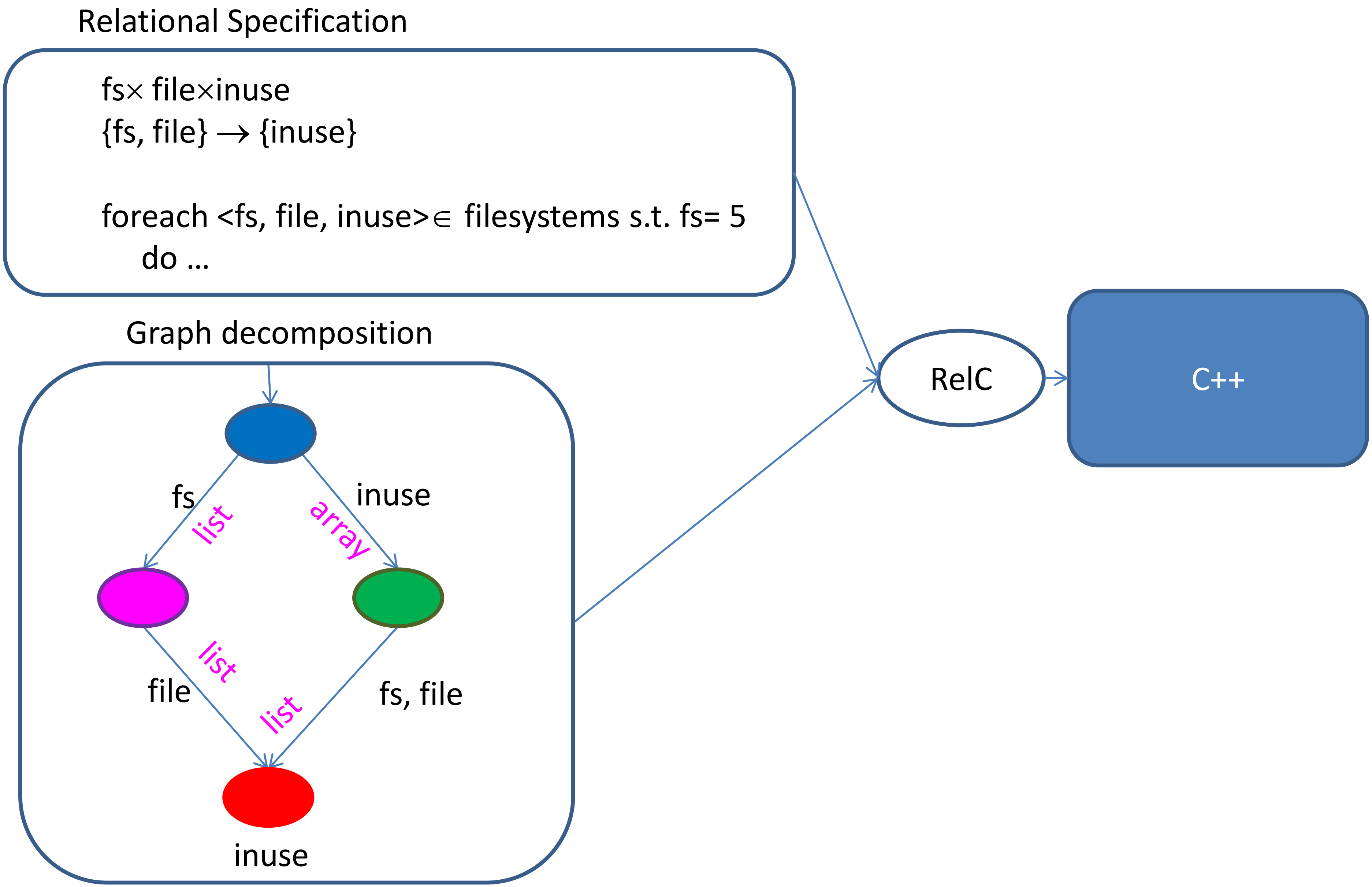
Plan

- 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× file×inuse
{fs, file} → {inuse}

foreach <fs, file, inuse>∈ filesystems s.t. fs= 5
do …

Graph decomposition

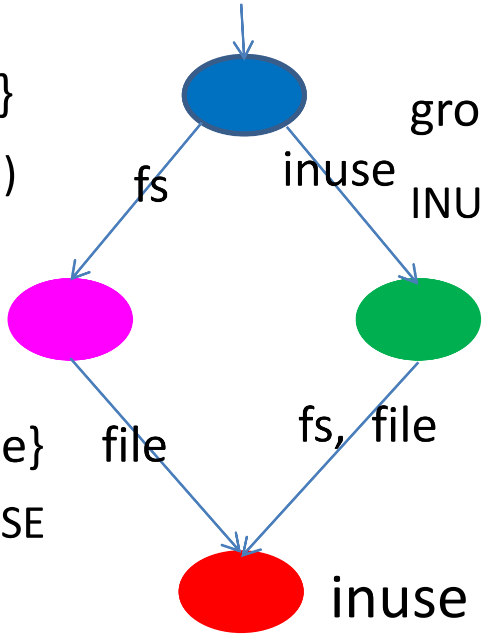
Decomposing Relations

- 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

fs× file×inuse

{fs, file} → {inuse}



fs× file×inuse

group-by {fs}

FS → (FILE×INUSE)

fs

inuse

group-by {inuse}

INUSE → FS × FILE

file×inuse

fs× file

group-by {file}

file

FILE→INUSE

fs, file

group_by {fs, file}

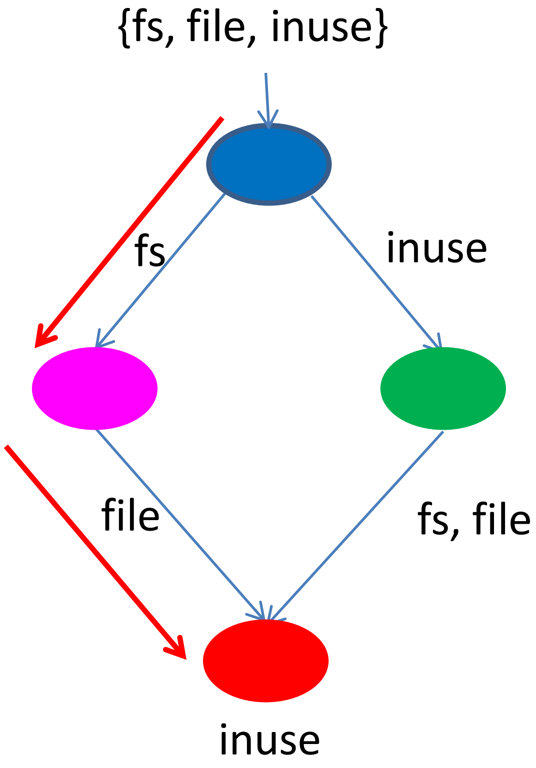
FS × FILE → INUSE

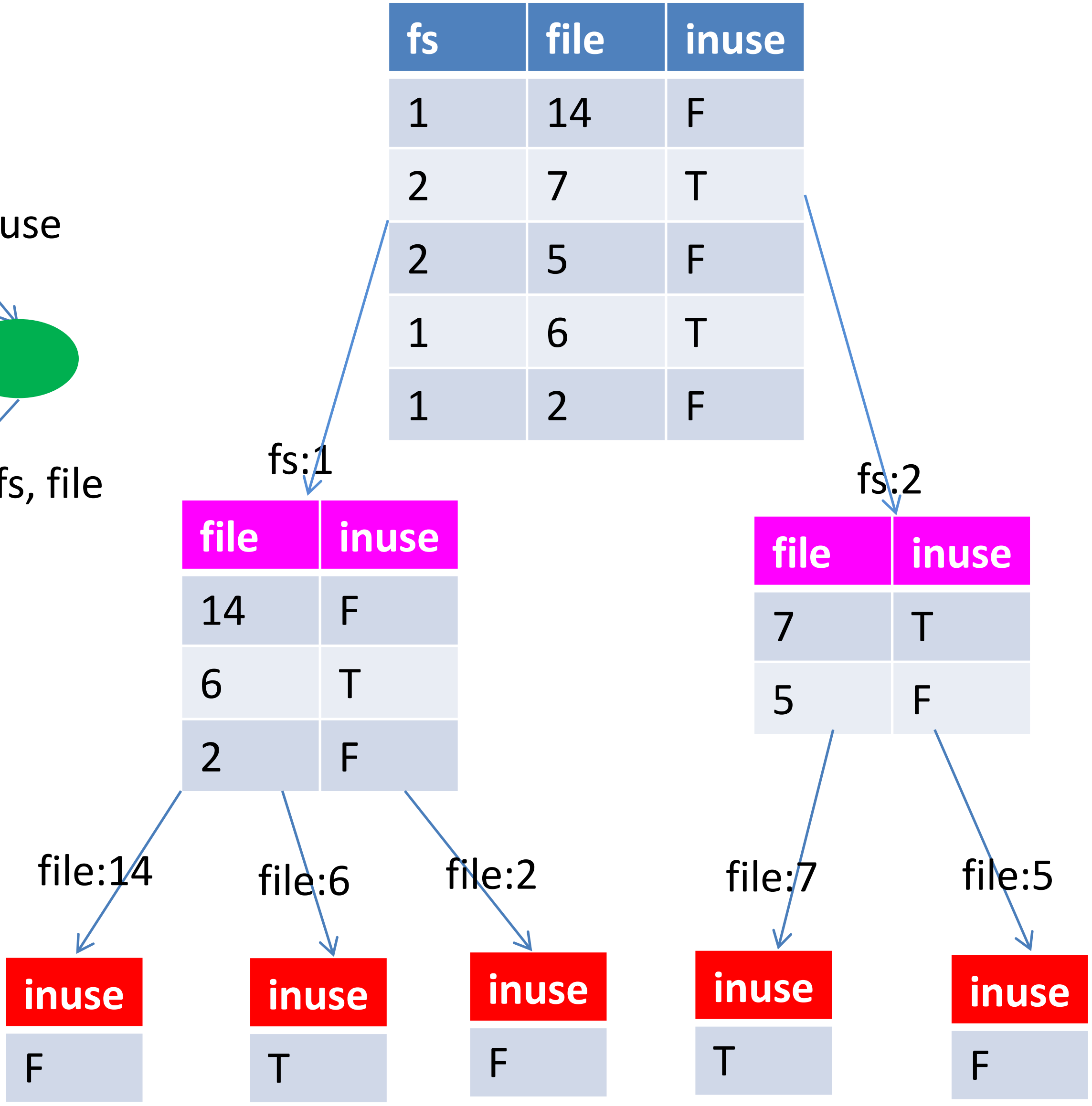
inuse

FS → (FILE→INUSE)

INUSE → (FS × FILE → INUSE)

Filesystem Example





fs	file	inuse
1	14	F
2	7	T
2	5	F
1	6	T
1	2	F

fs:1

file	inuse
14	F
6	T
2	F

file:14

inuse
F

file:6

inuse
T

file:2

inuse
F

fs:2

file	inuse
7	T
5	F

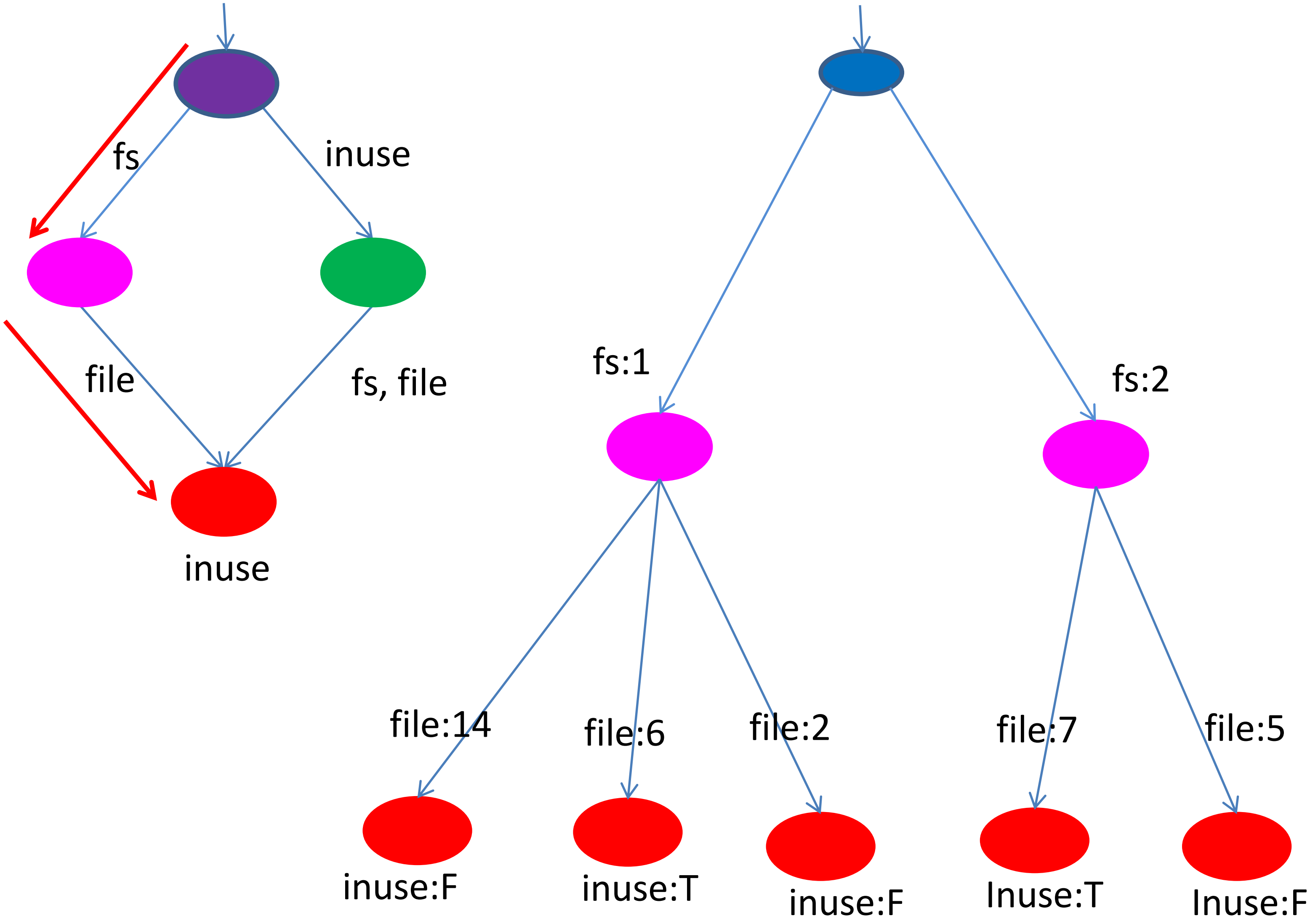
file:7

inuse
T

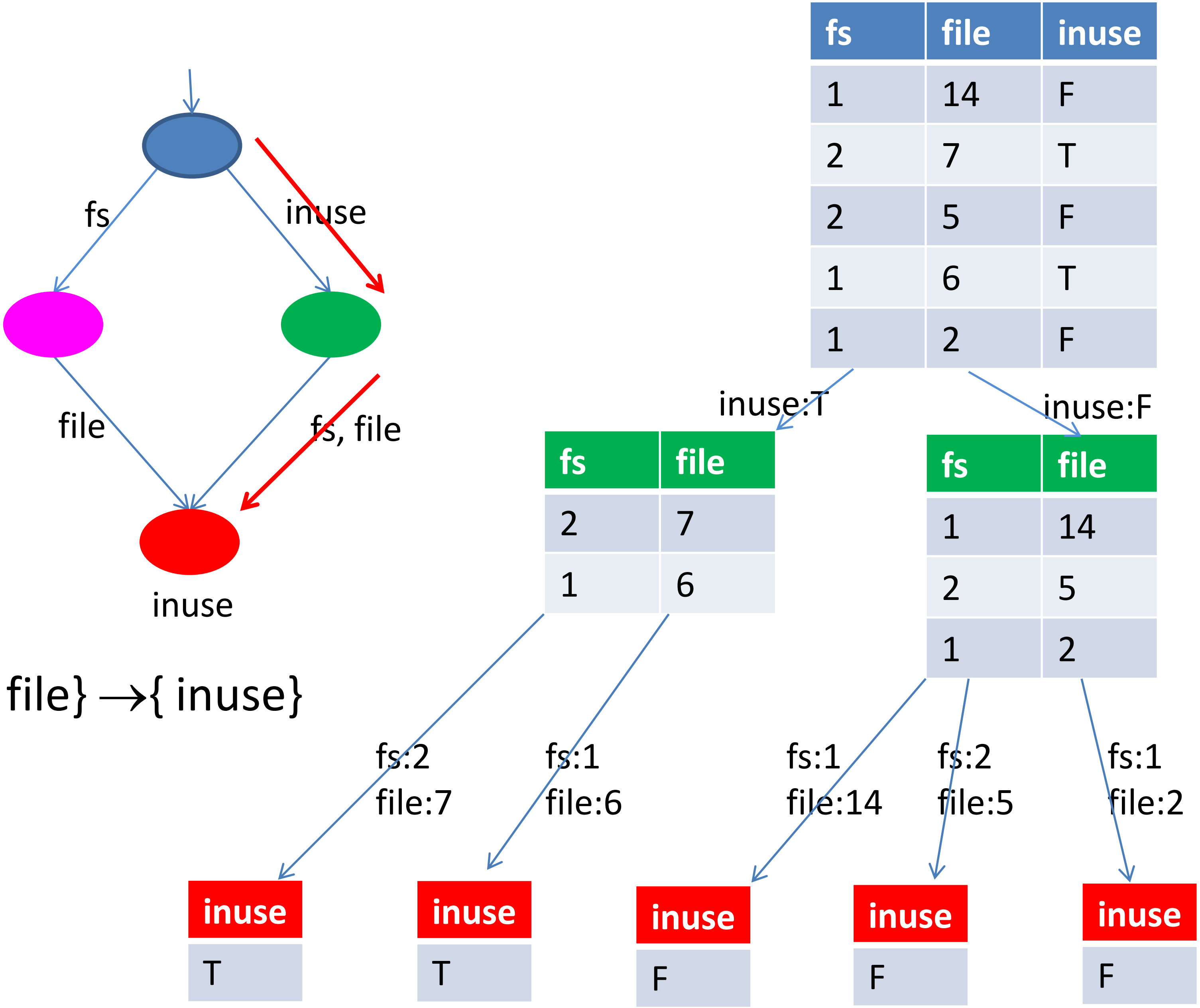
file:5

inuse
F

Memory Decomposition(Left)



Filesystem Example



{fs, file} →{ inuse}

fs	file	inuse
1	14	F
2	7	T
2	5	F
1	6	T
1	2	F

inuse:T

fs	file
2	7
1	6

inuse:F

fs	file
1	14
2	5
1	2

fs:2
file:7

inuse
T

fs:1
file:6

inuse
T

fs:1
file:14

inuse
F

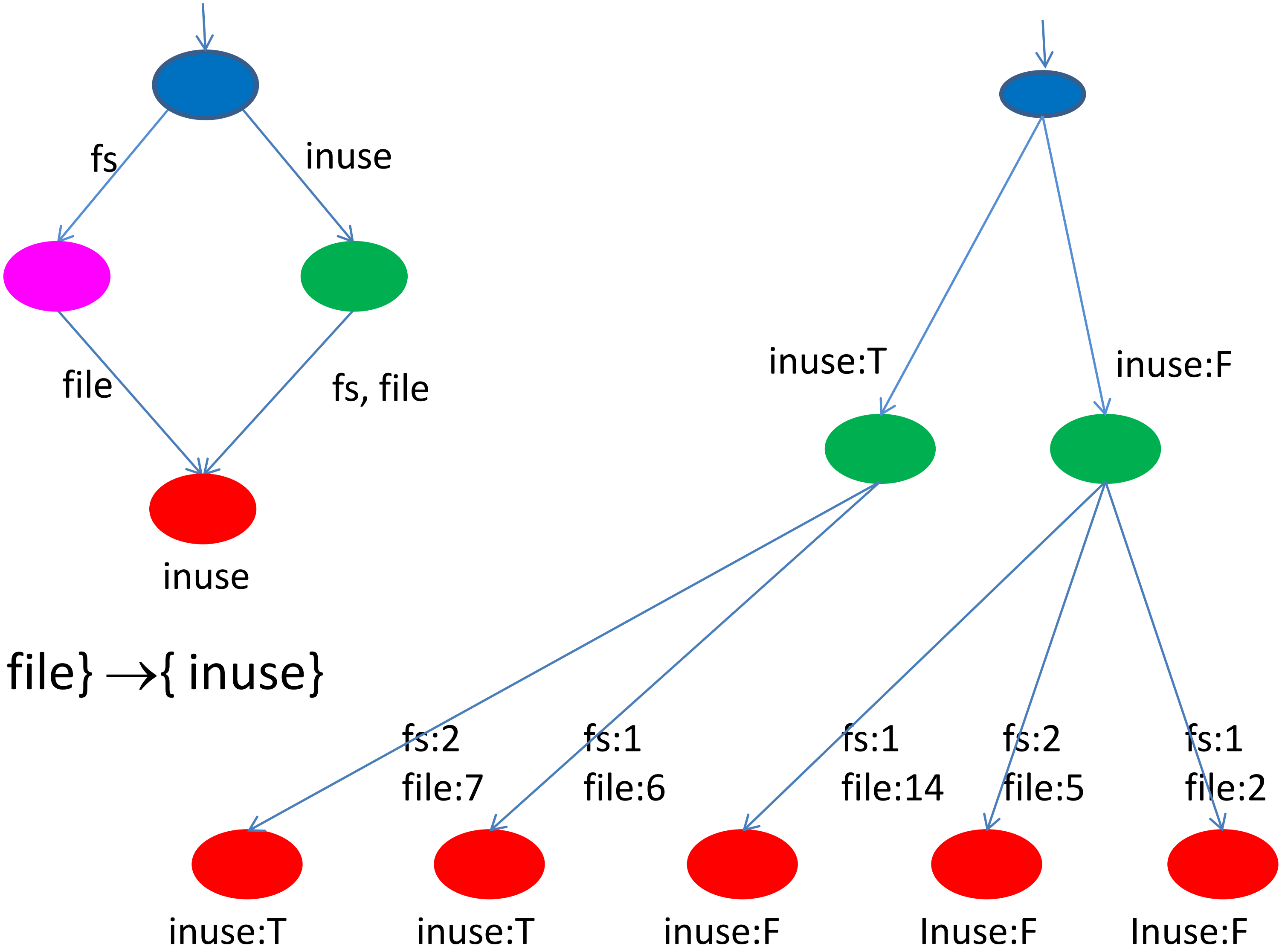
fs:2
file:5

inuse
F

fs:1
file:2

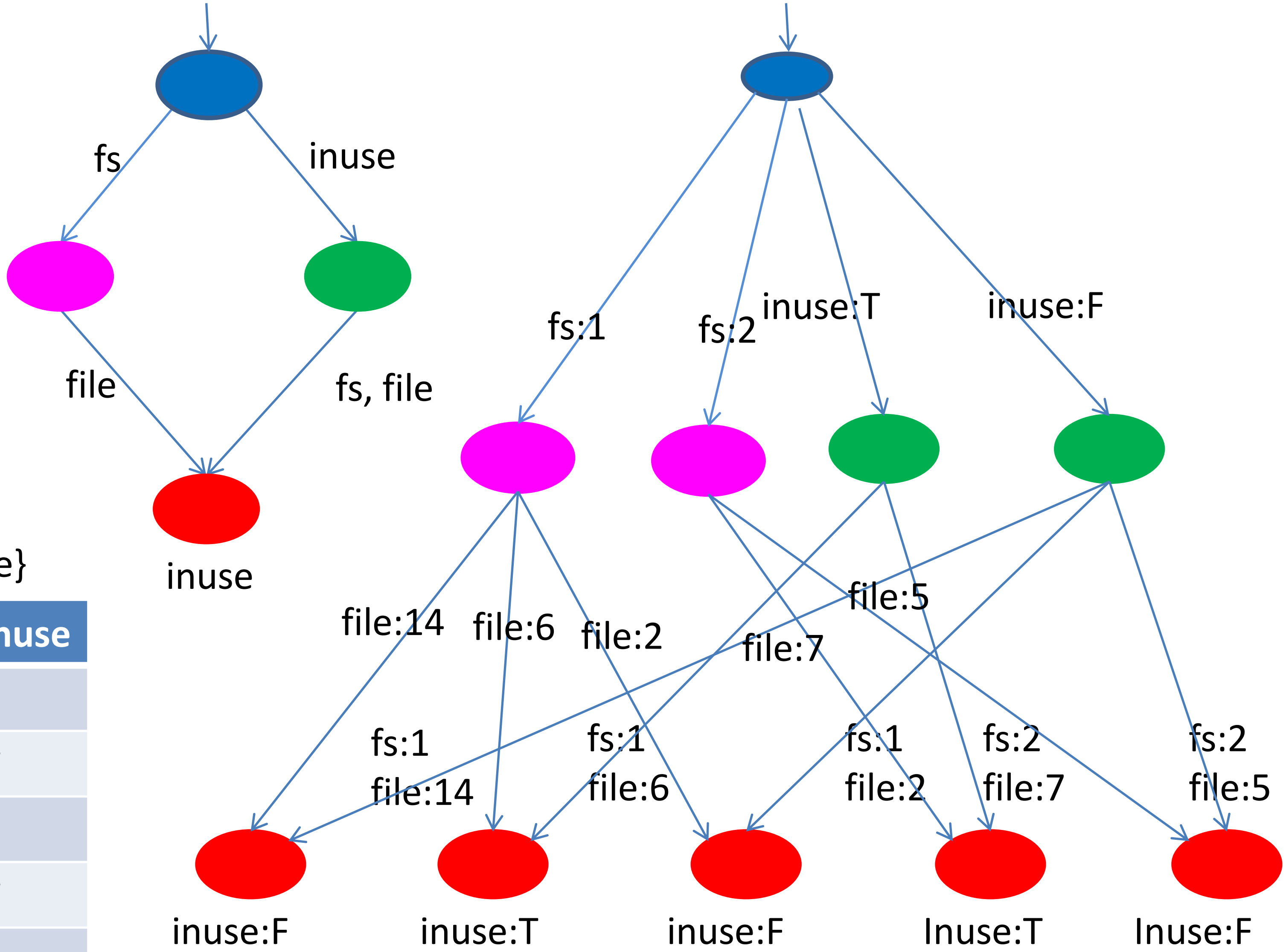
inuse
F

Memory Decomposition(Right)



{fs, file} →{ inuse}

Decomposition Instance

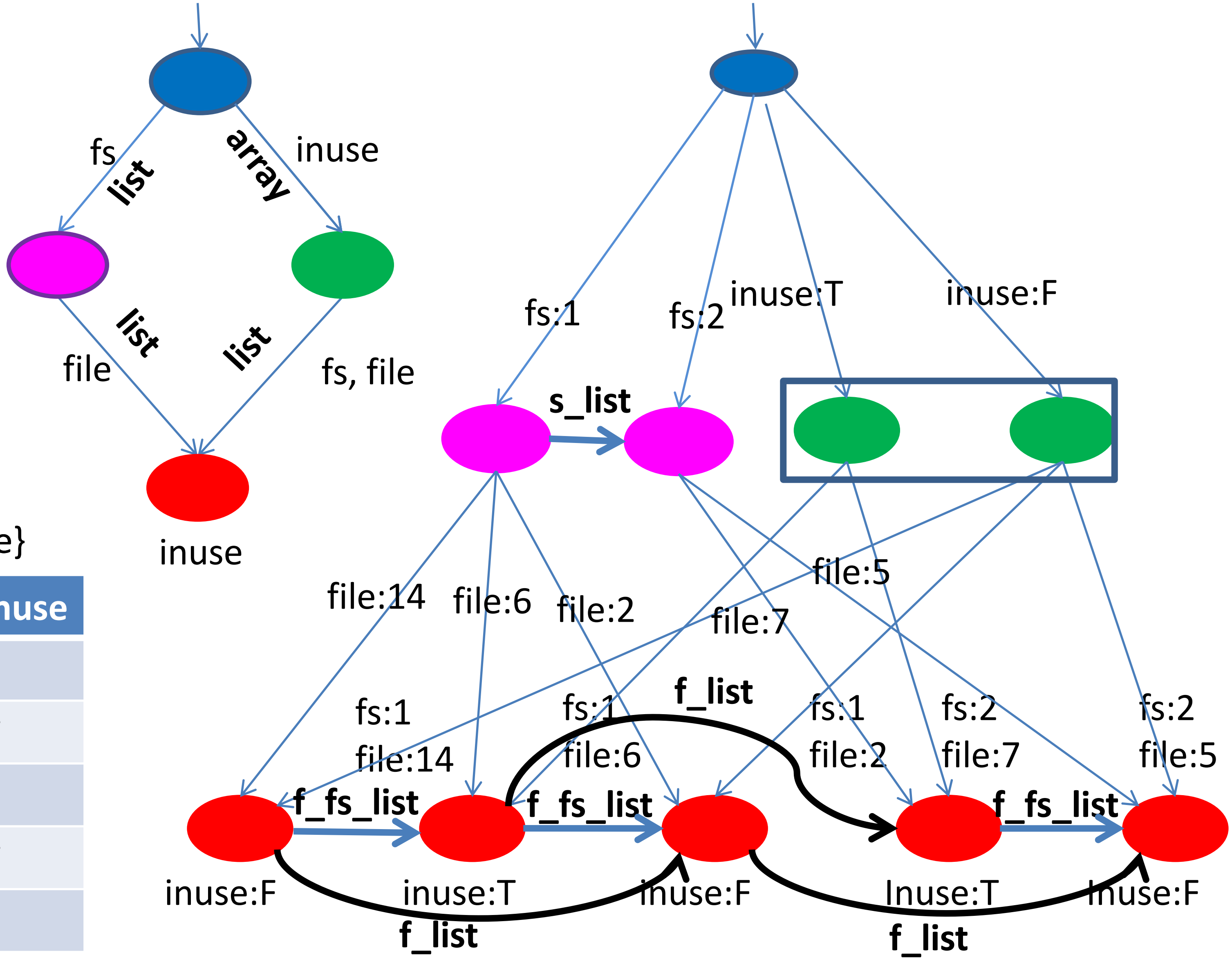


fs × file × inuse

{fs, file} →{ inuse}

fs	file	inuse
1	14	F
2	7	T
2	5	F
1	6	T
1	2	F

Decomposition Instance



fs × file × inuse

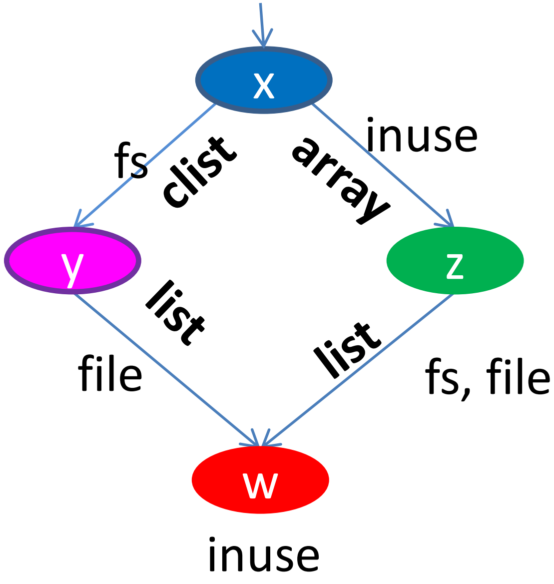
{fs, file} →{ inuse}

fs	file	inuse
1	14	F
2	7	T
2	5	F
1	6	T
1	2	F

Decomposing Relations Formally(PLDI’11)

$fs \times file \times inuse$

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



let w: $\{fs, file, inuse\} \triangleright \{inuse\} = \{inuse\}$ in

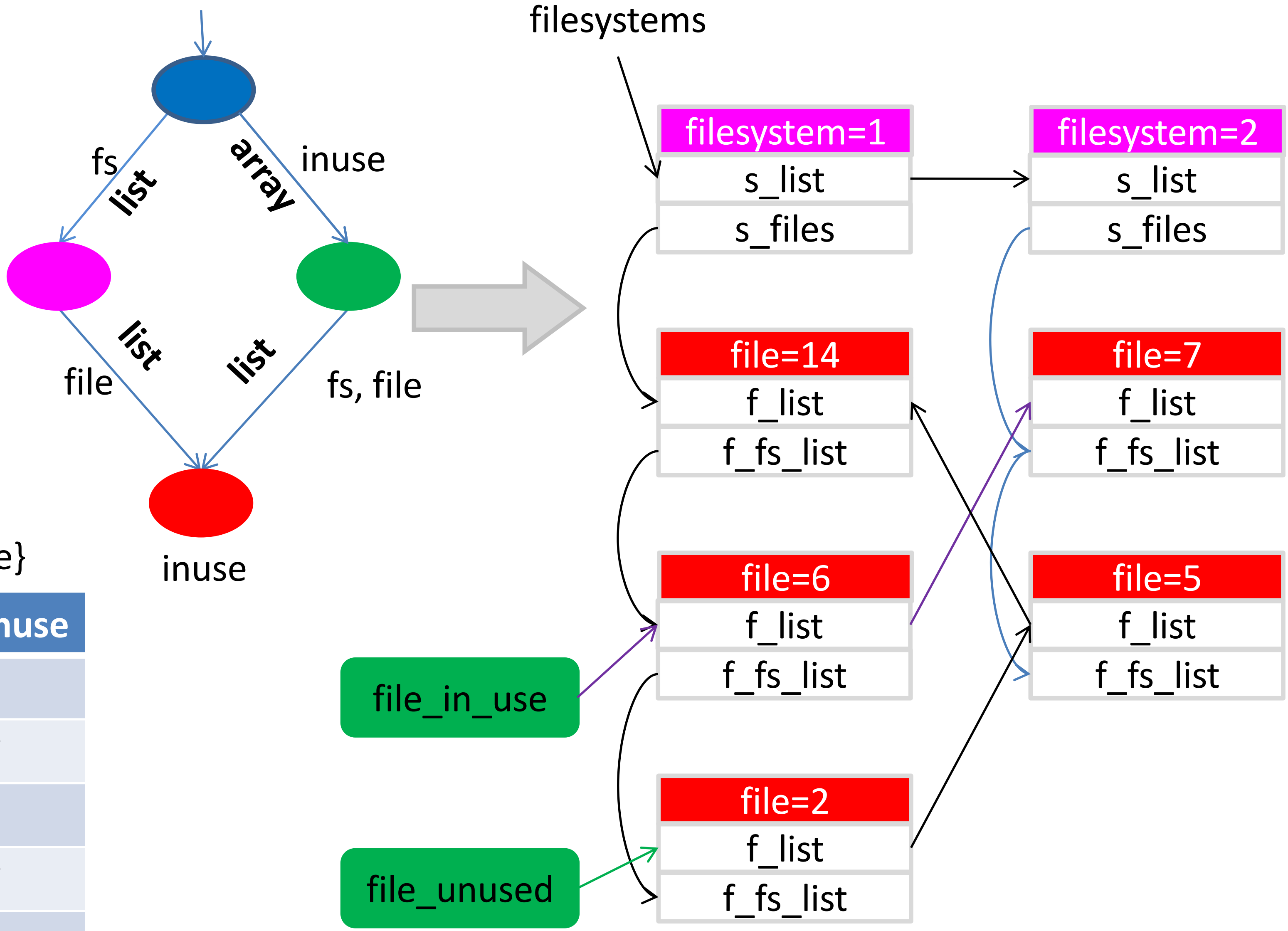
let y : $\{fs\} \triangleright \{file, inuse\} = \{file\} \rightarrow^{list} \{w\}$ in

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

let x: $\{\} \triangleright \{fs, file, inuse\} = \{fs\} \rightarrow^{clist} \{y\} \bowtie$

$\{inuse\} \rightarrow^{array} \{z\}$

Memory State



fs × file × inuse

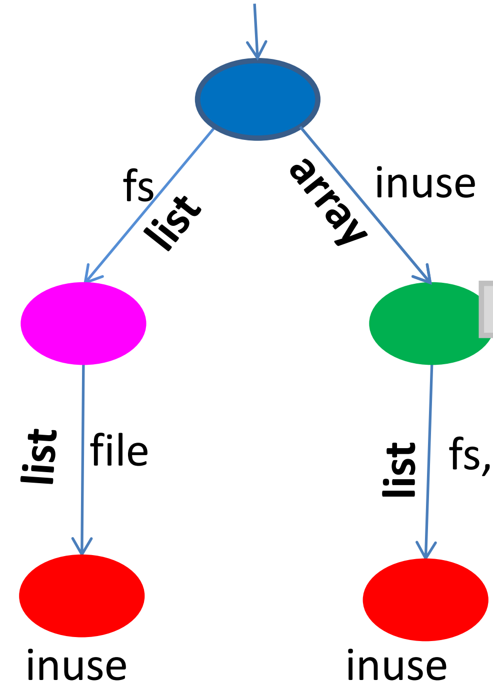
{fs, file} →{ inuse}

fs	file	inuse
1	14	F
2	7	T
2	5	F
1	6	T
1	2	F

Memory State(2)

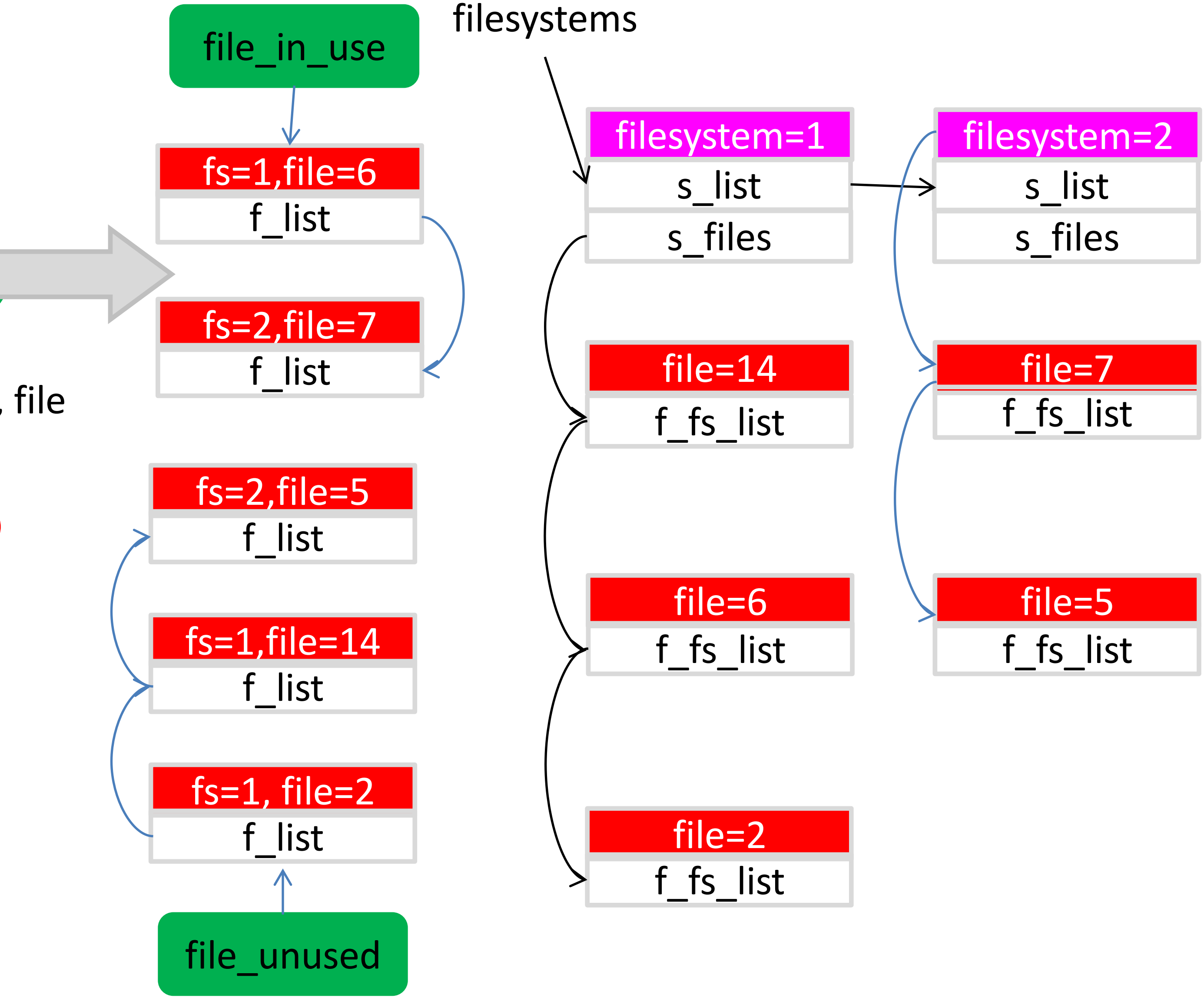
fs × file × inuse

{fs, file} →{ inuse}



fs
list
array
inuse
list
file
list
fs, file
inuse
inuse

fs	file	inuse
1	14	F
2	7	T
2	5	F
1	6	T
1	2	F



file_in_use
fs=1,file=6
f_list
fs=2,file=7
f_list
fs=2,file=5
f_list
fs=1,file=14
f_list
fs=1, file=2
f_list
file_unused

filesystems
filesystem=1
s_list
s_files
file=14
f_fs_list
file=6
f_fs_list
file=2
f_fs_list

filesystem=2
s_list
s_files
file=7
f_fs_list
file=5
f_fs_list

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

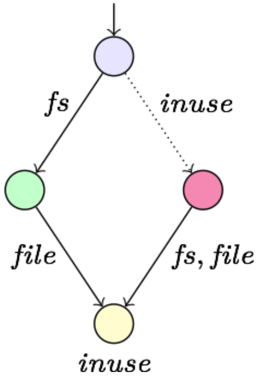
enforces sufficient conditions for adequacy

$$\{fs, file, inuse\}$$

$$fs, file \to inuse$$

Adequacy

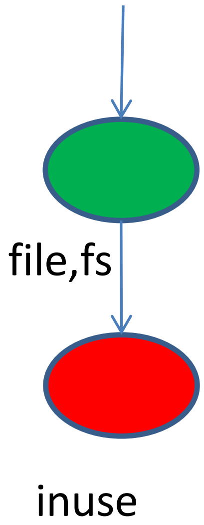
⟷



Adequacy of Decompositions

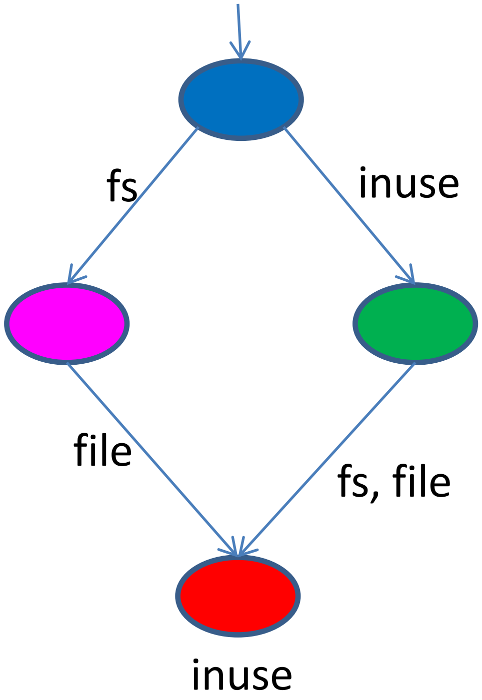
- All columns are represented
- Nodes are consistent with functional dependencies
  - Columns bound to paths leading to a common node must functionally determine each other

Respect Functional Dependencies



✓ {file, fs} → {inuse}

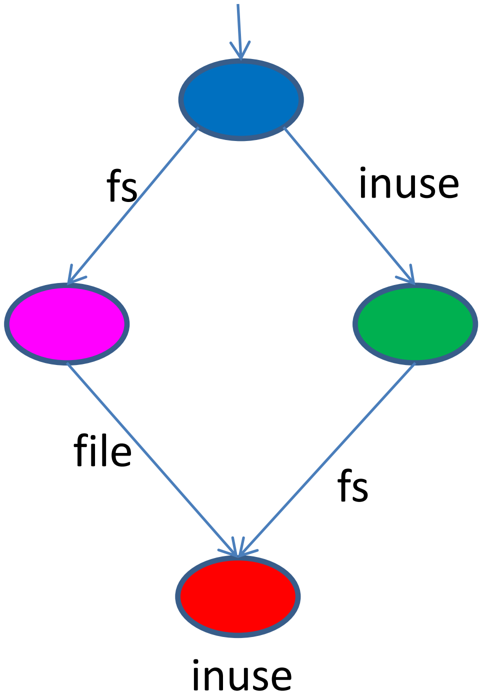
Adequacy and Sharing



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

- ✓ {fs, file}↔{inuse, fs, file}

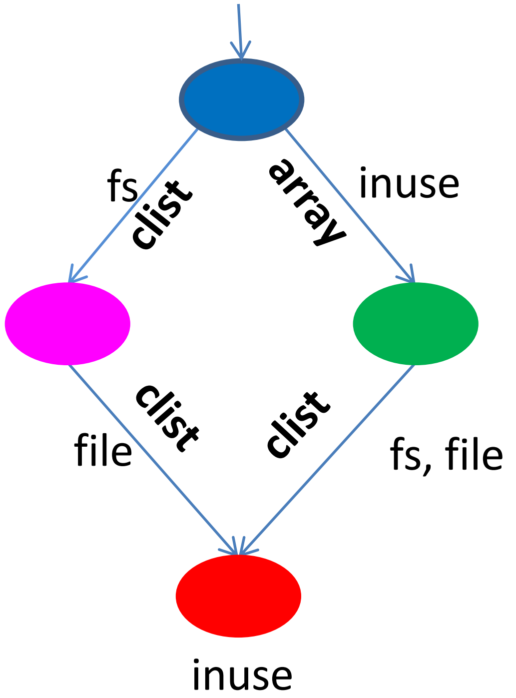
Adequacy and Sharing

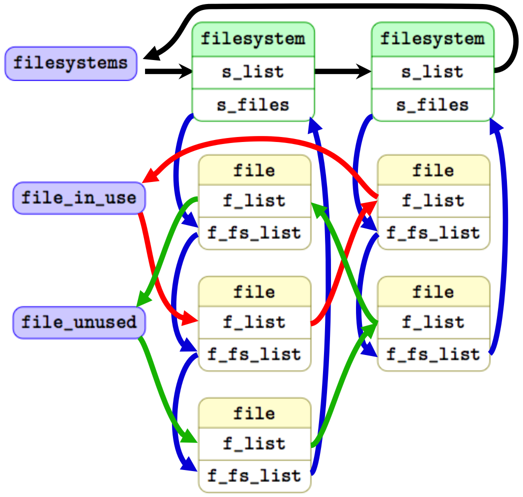


Columns bound on a path to an object *x must functionally* determine columns bound on any other path to *x*

☒ {fs, file} ↮ {inuse, fs}

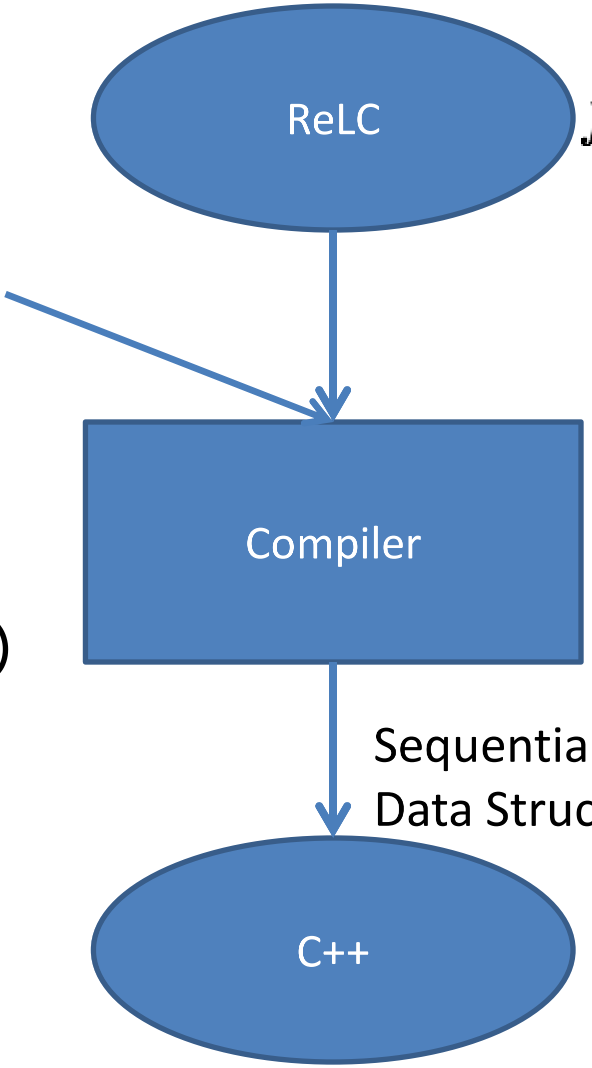
The RelC Compiler PLDI'11





$$\{fs, file, inuse\}$$

$$fs, file \rightarrow inuse$$



ReLC

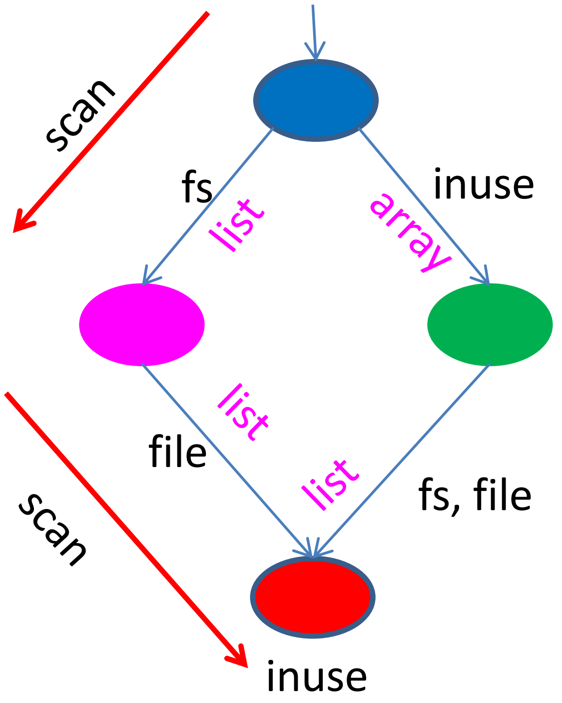
Compiler

Sequential Compositions of
Data Structures

C++

Query Plans

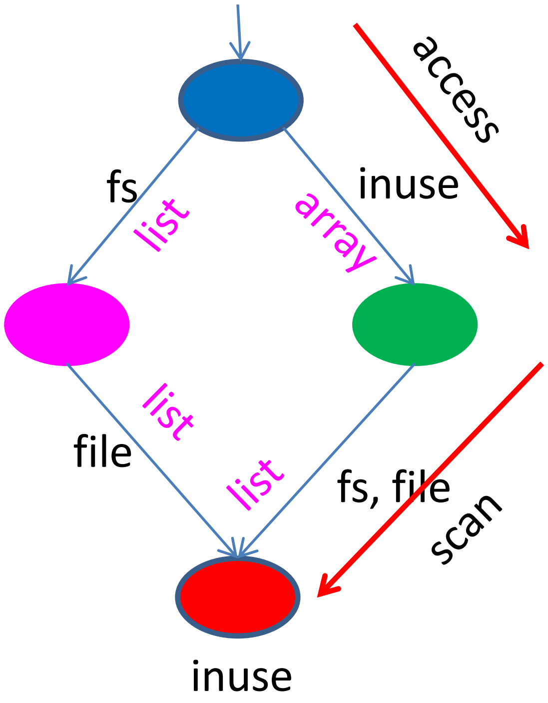
foreach <fs, file, inuse>∈ filesystems
  if inuse=T do …



Cost proportional to the number of files

Query Plans

foreach <fs, file, inuse>∈ filesystems
  if inuse=T do …

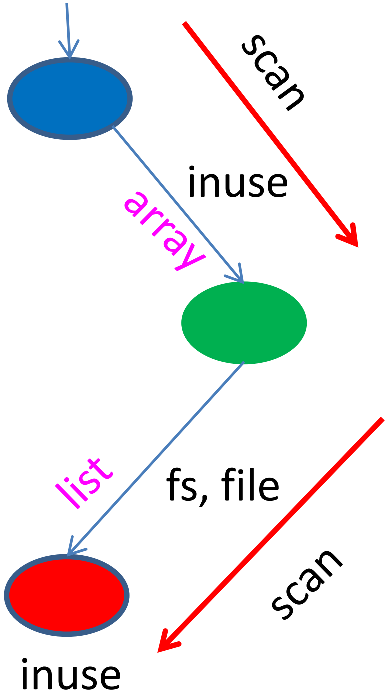


Cost proportional to the number of files in use

Completeness

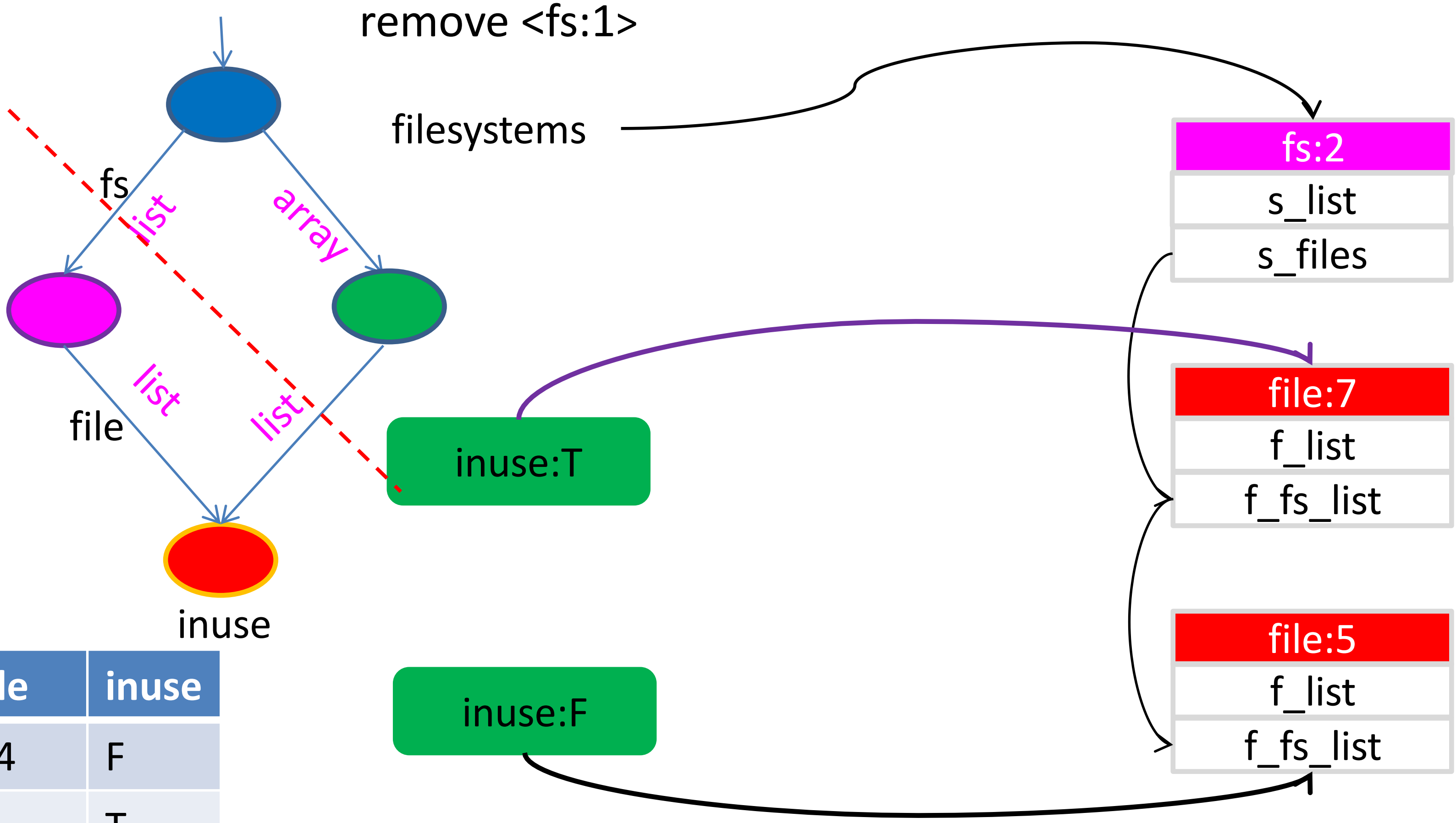
- The representation is adequate → the compiler can always generate correct code
- But the code may be slow

foreach <fs, file, inuse>$\in$ filesystems s.t. fs=1 do



Removal and graph cuts

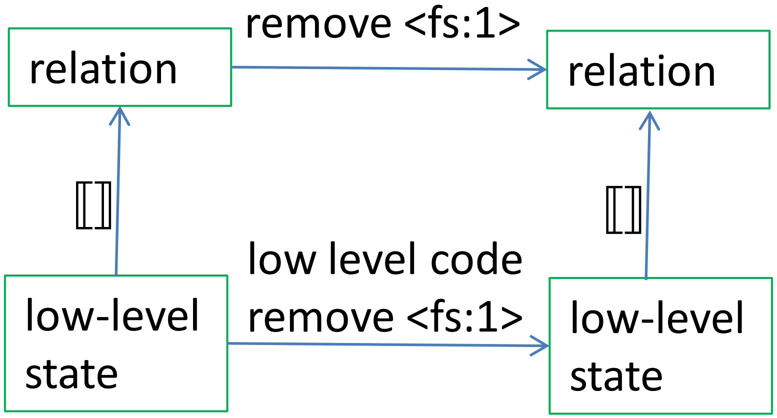
remove <fs:1>



fs	file	inuse
1	14	F
2	7	T
2	5	F
1	6	T
1	2	F

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

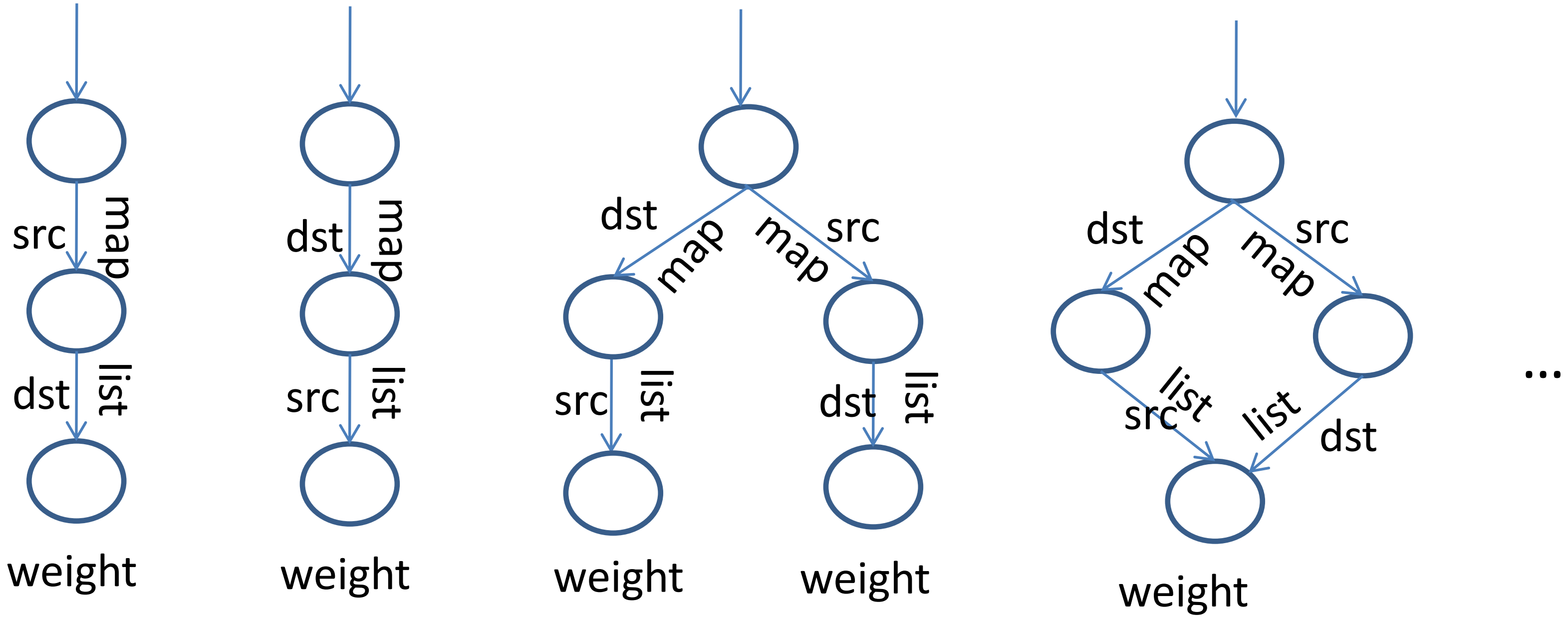
- Specify provably correct program transformations
- Select the best compiled code using a workload

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
  src × dst × weight
- Functional Dependencies
  - $\{src, dst\} \rightarrow \{weight\}$
- Primitive data types
  - map, list

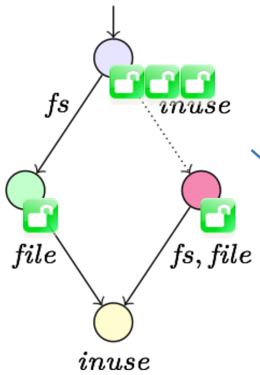


Synthesizing Concurrent Programs

PLDI’12

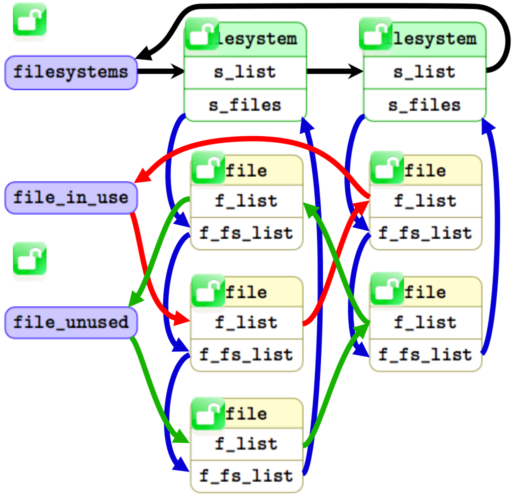
The High Level Idea

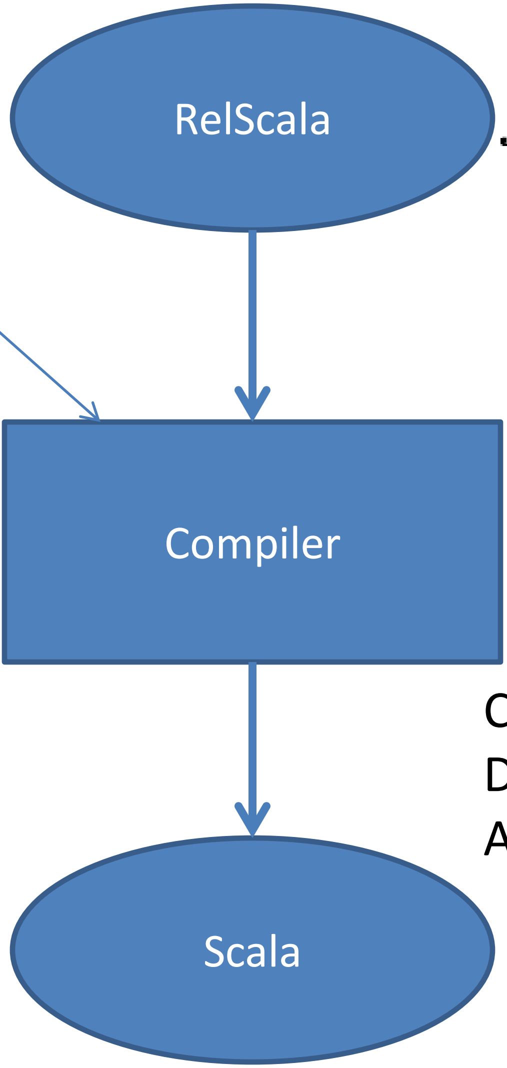
Concurrent Decomposition



⟶ ConcurrentHashMap

⇢ HashMap





RelScala

$$\{fs, file, inuse\}$$

$$fs, file \rightarrow inuse$$

query <inuse:T> {fs, file}

Compiler

Concurrent Compositions of
Data Structures,
Atomic Transactions

Scala

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

Two-Phase Locking

Attach a lock to each piece of data

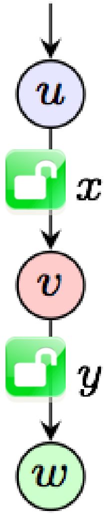
Two phase locking protocol:

- **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

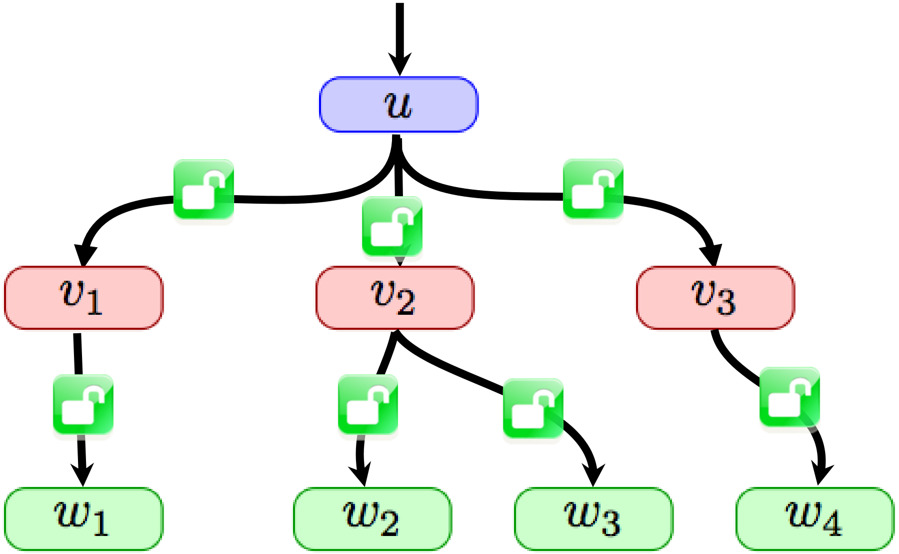
Theorem [Eswaran et al., 1976]: Well-locked, two-phase transactions are serializable

Two Phase Locking

Decomposition



Decomposition Instance



Attach a lock to every edge

Two Phase Locking ➔ Serializability

We're done!

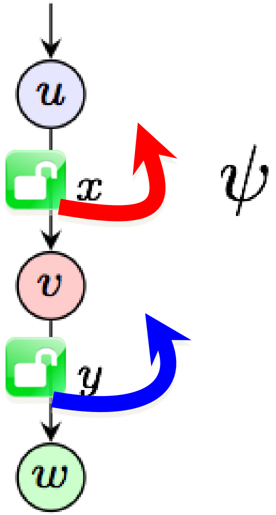
Problem 1: Can't attach locks to container entries

Problem 2: Too many locks

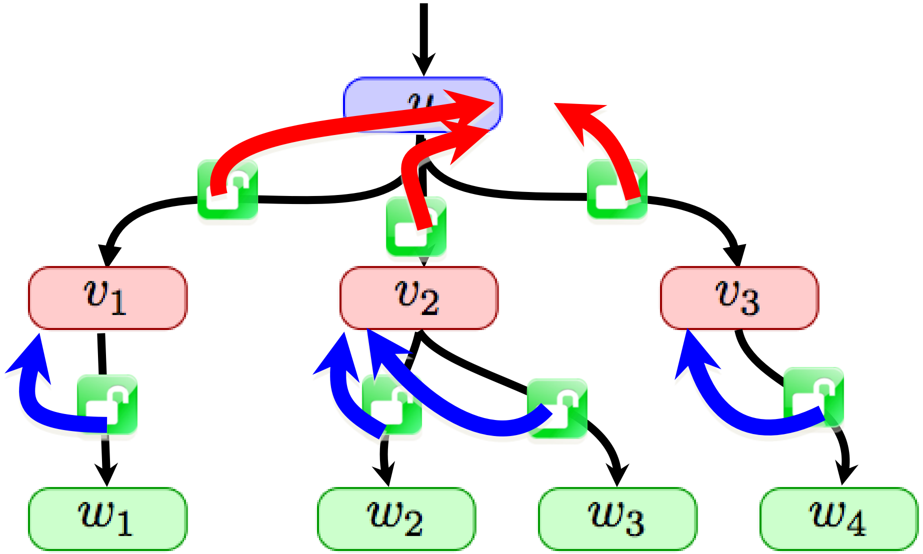
Butler Lampson/David J. Wheeler: "Any problem in computer science can be solved with another level of indirection."

Lock Placements

Decomposition



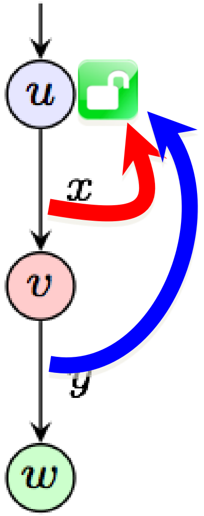
Decomposition Instance



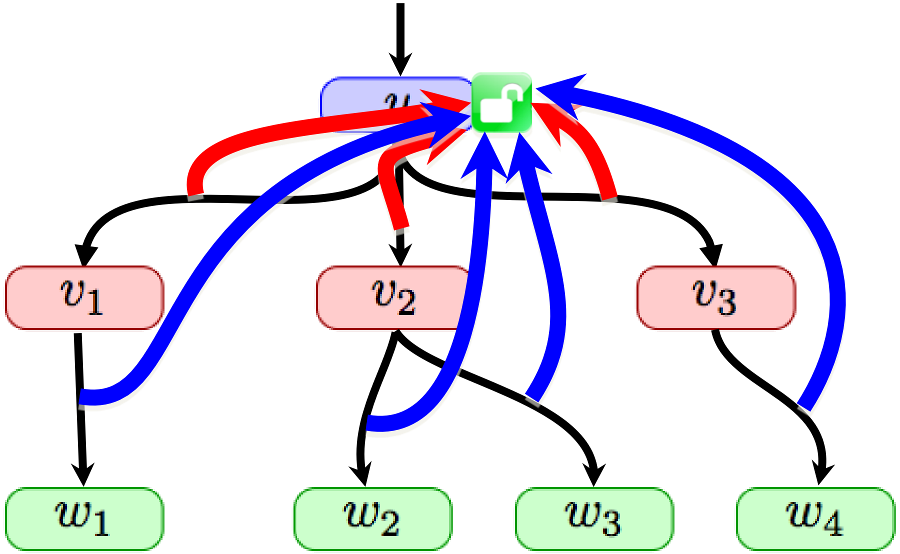
1. **Attach locks to nodes**
2. Use a *lock placement* ψ to map data (on edges) to locks (on nodes)

Coarse-Grained Locking

Decomposition



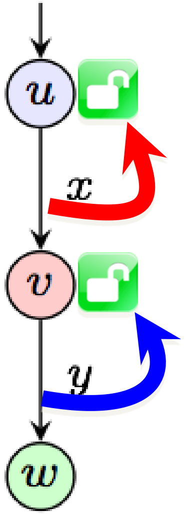
Decomposition Instance



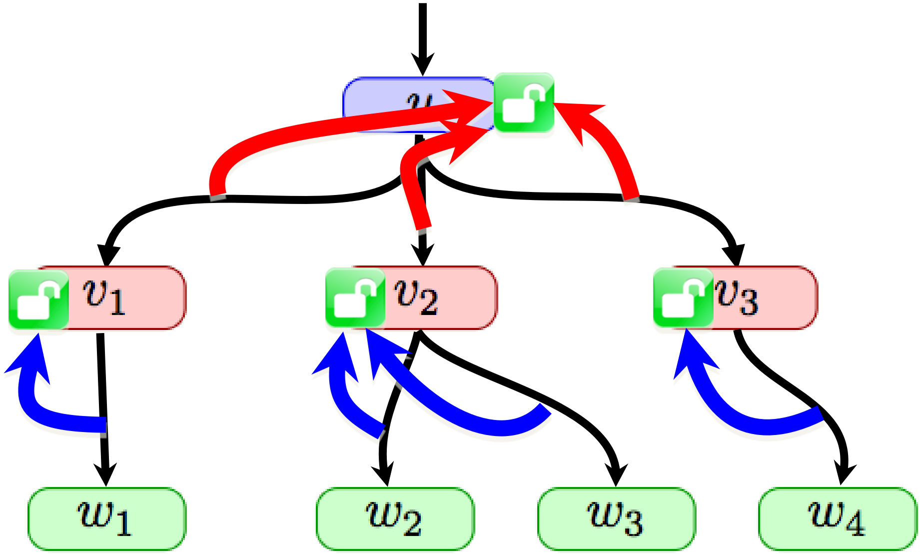
$$\psi = \{uv \mapsto u,\ vw \mapsto u\}$$

Finer-Grained Locking

Decomposition



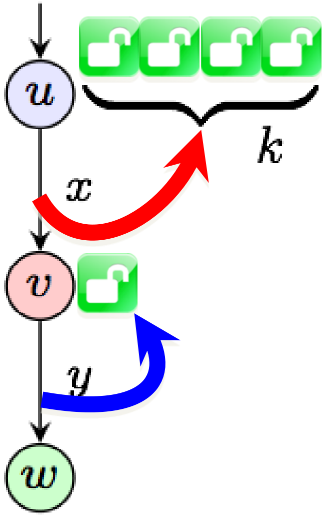
Decomposition Instance



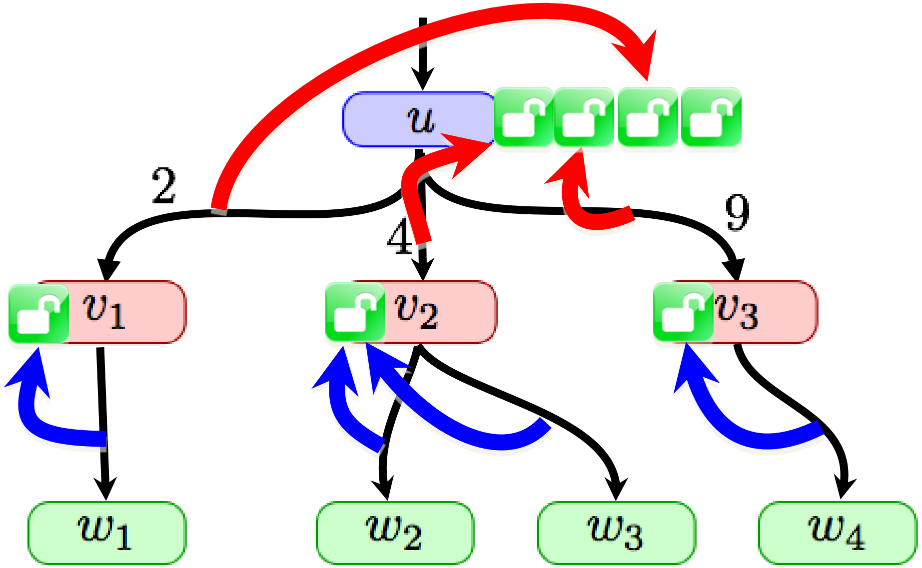
$$\psi = \{uv \mapsto u,\ vw \mapsto v\}$$

Lock Striping

Decomposition



Decomposition Instance

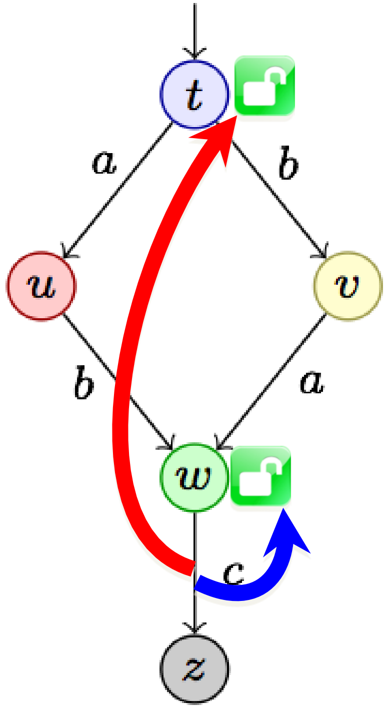


$$\psi = \{uv_x \mapsto u_{x \bmod k},\ vw \mapsto v\}$$

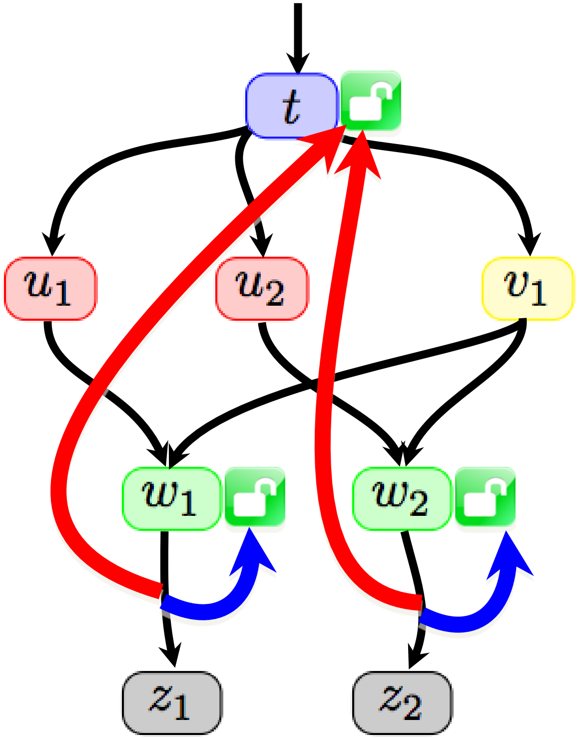
Lock Placements: Domination

Locks must dominate the edges they protect

Decomposition

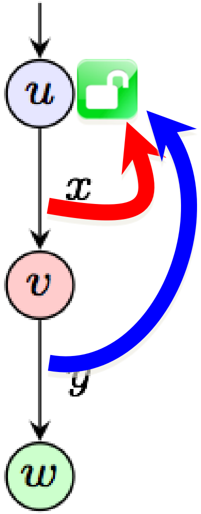


Decomposition Instance



Lock Placements: Path-Closure

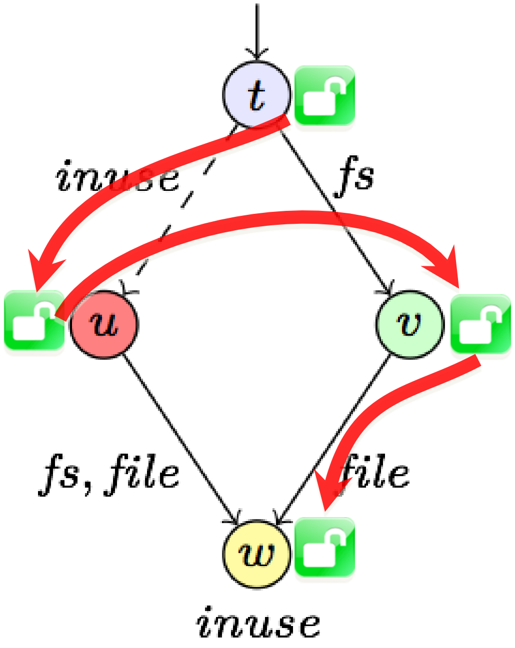
All edges on a path between an edge and its lock must share the same lock



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

Lock Ordering

Prevent deadlock via a topological order on locks

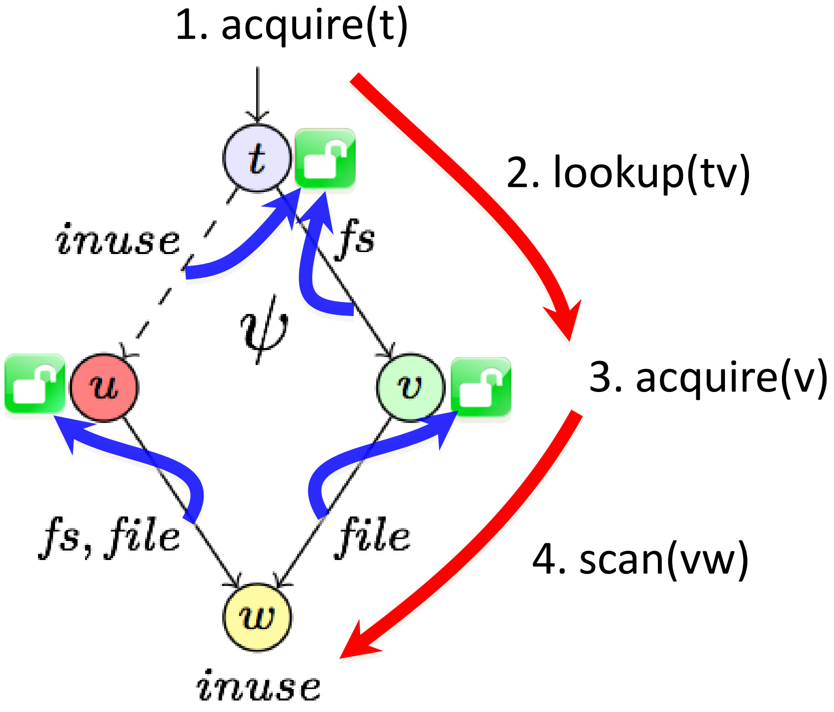


$$t \prec u \prec v \prec w$$

Queries and Deadlock

Query plans must acquire the correct locks in the correct order

$$t \prec u \prec v \prec w$$



1. acquire(t)

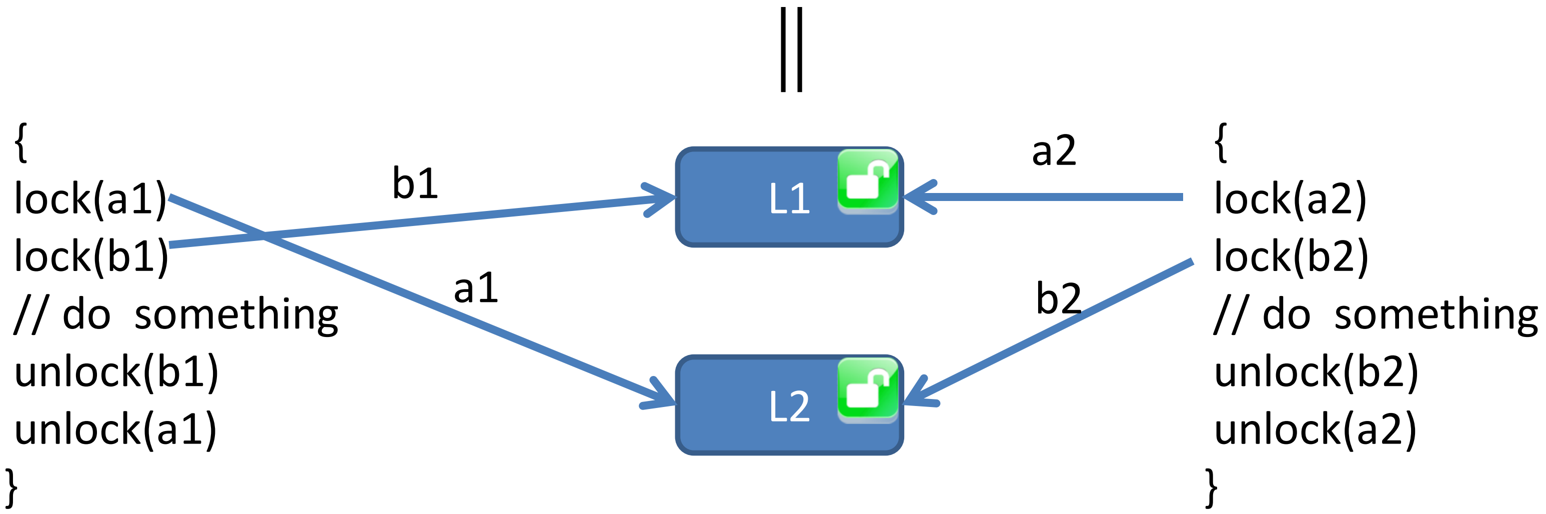
2. lookup(tv)

3. acquire(v)

4. scan(vw)

Example: find files on a particular filesystem

Deadlock and Aliasing



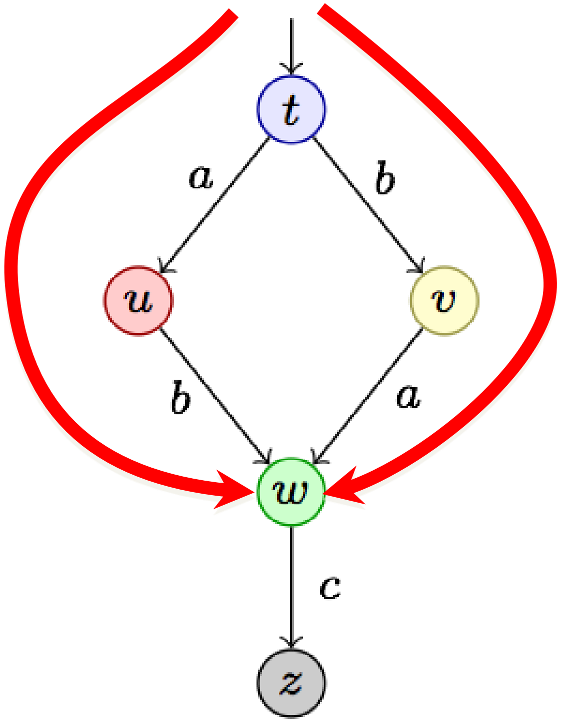
```
{
lock(a1)
lock(b1)
// do  something
unlock(b1)
unlock(a1)
}
```

||

```
{
lock(a2)
lock(b2)
// do  something
unlock(b2)
unlock(a2)
}
```

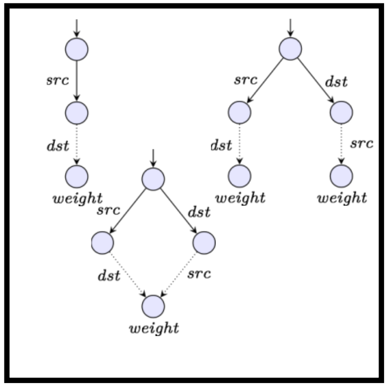
Decompositions and Aliasing

- 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

×

`Array` `TreeMap`

`HashMap`

`LinkedList`

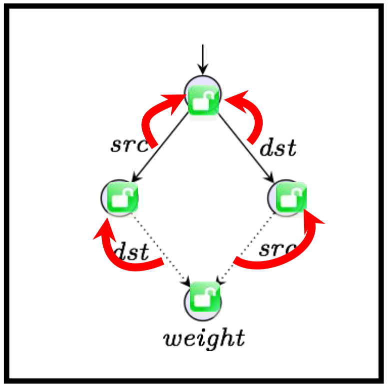
`ConcurrentHashMap`

`ConcurrentSkipListMap`

`CopyOnWriteArrayList`

Container Data Structures

×



Lock Placement

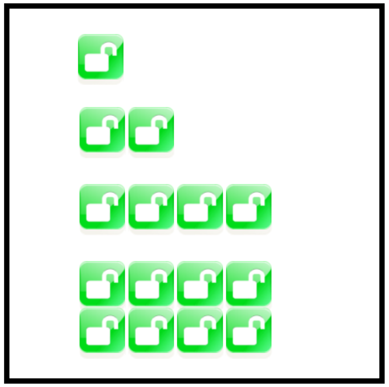
×

`ReentrantLock`

`ReentrantReadWriteLock`

Lock Implementations

×



Lock Striping Factors

Based on Herlihy’s benchmark of concurrent maps

Concurrent Graph Benchmark

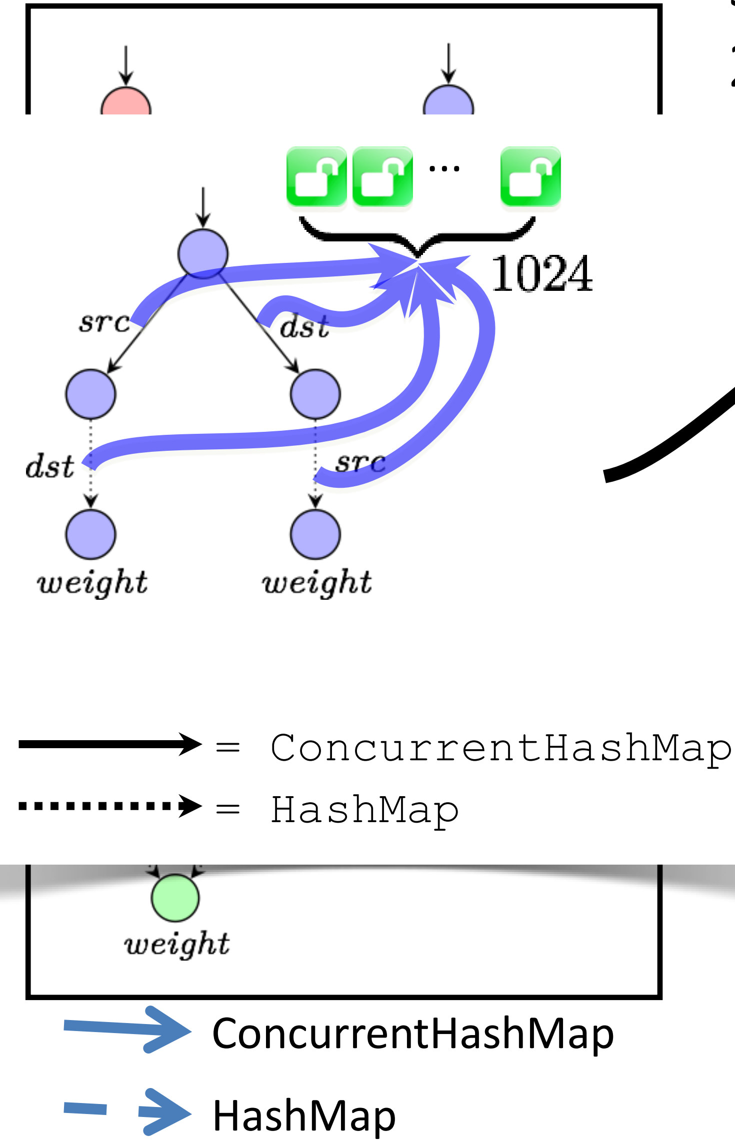
$$\{src, dst, weight\}$$

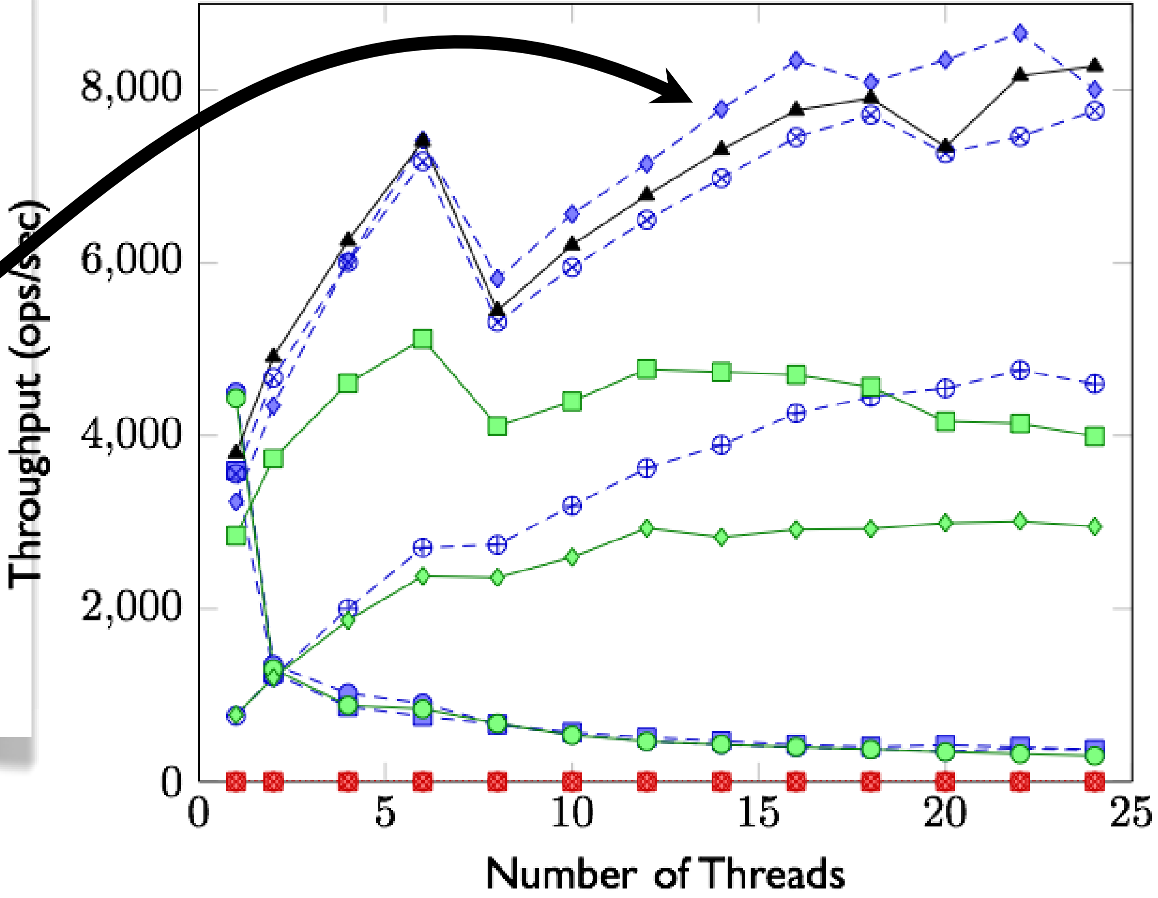
$$src, dst \rightarrow weight$$

- Start with an empty graph
- Each thread performs 5 x 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

Results: 35-35-20-10

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





(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