Modular Reasoning about Aliasing using Permissions

John Boyland

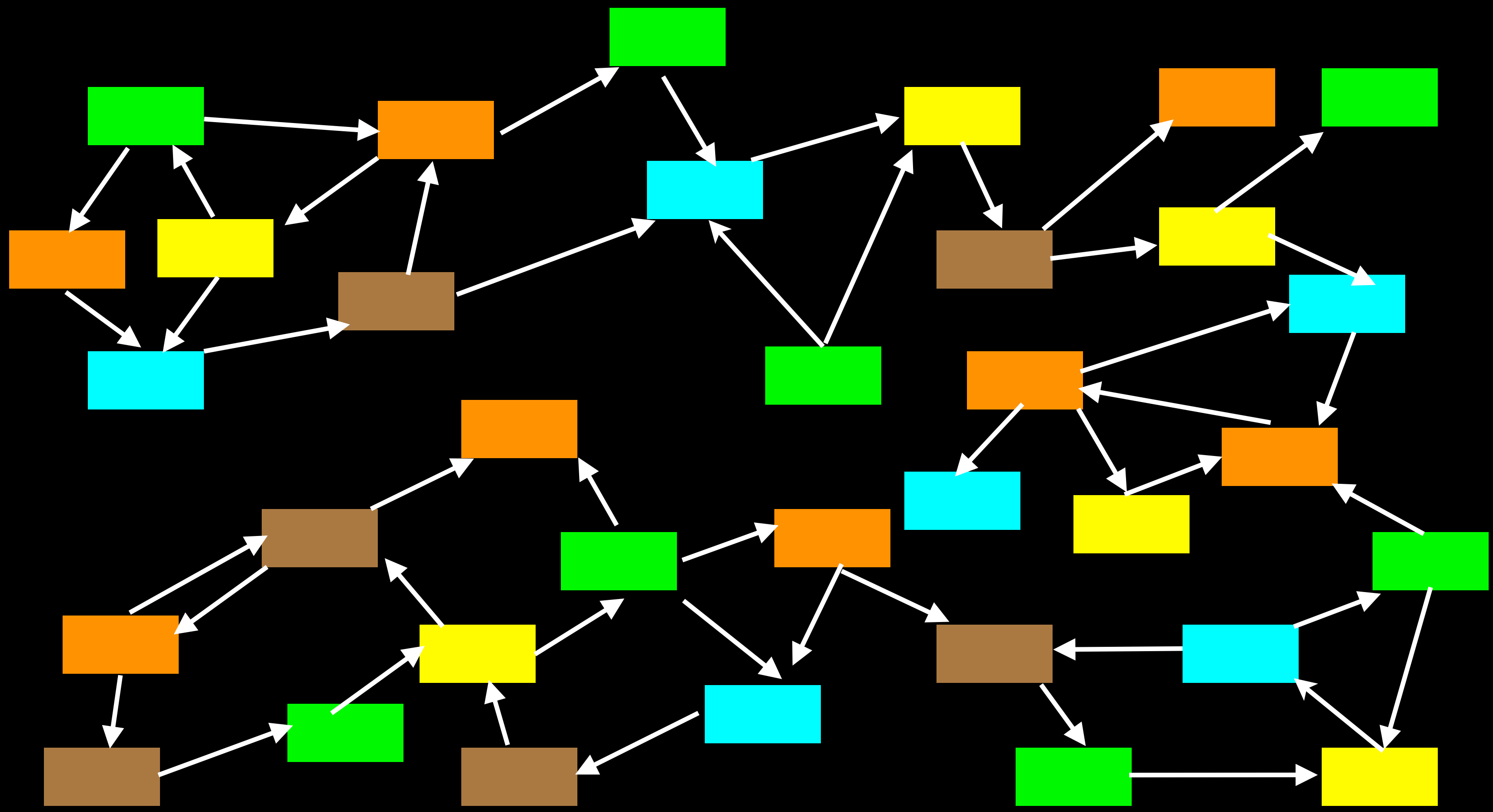
University of Wisconsin-Milwaukee

FOAL 2015

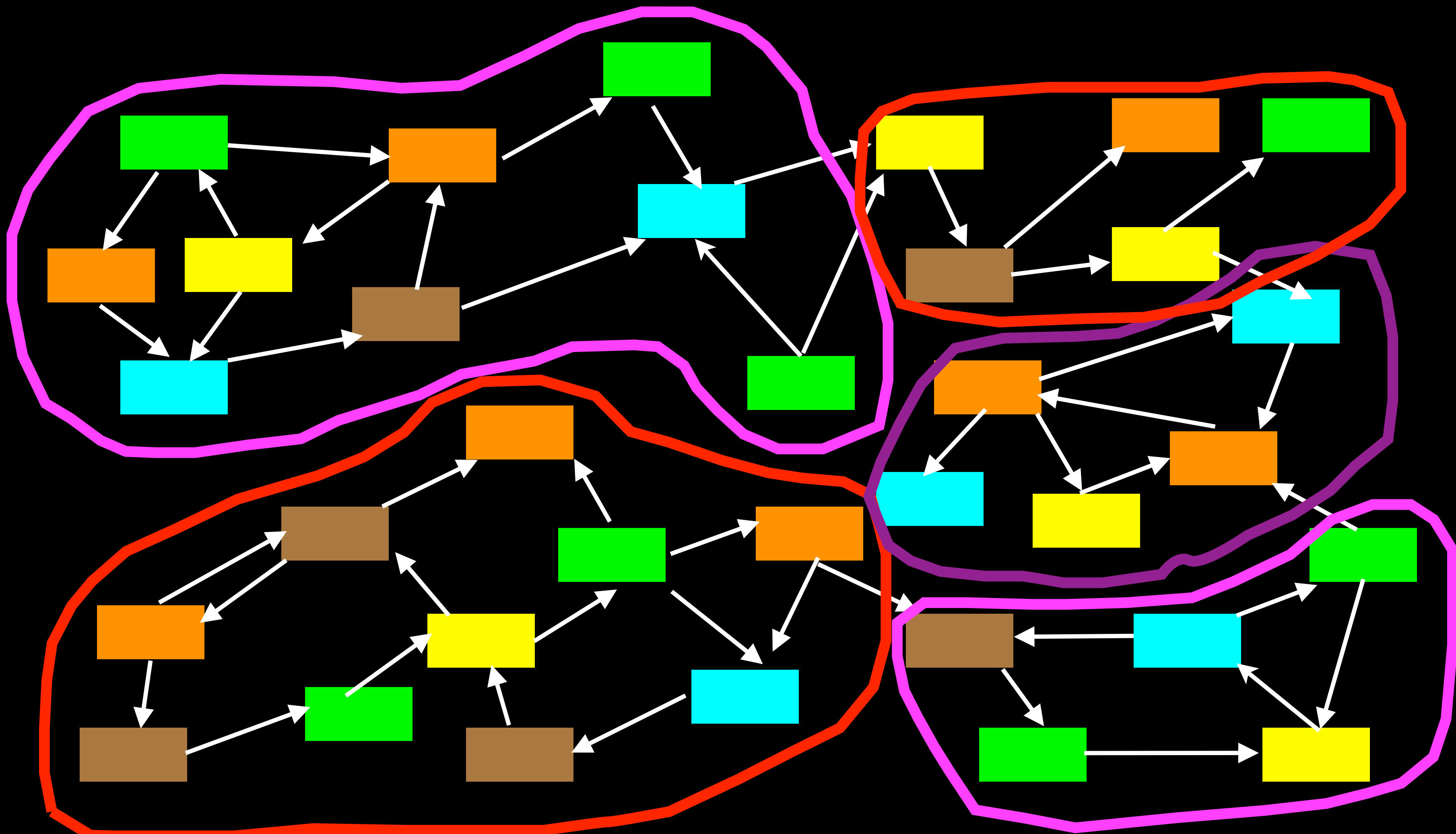
Summary

- Permissions are non-duplicable tokens that give access to state.
- Permissions give “effective” control over aliasing.
- Permission analysis determines whether code has access to state it uses.
- We use abstraction over permissions to have a uniform picture of method behavior.

Hidden Structure?



Hidden Structure?



Abstraction: a Problem

- Consider

```
interface Runnable {
  public void run();
}
```

- What does a call do?

```
r.run();
```

Permission Semantics

- In order to access mutable state, we need a
  - WRITE permission to write,
  - READ permission to read.

  Fractional permissions unify these.
- Permissions cannot be copied, only passed along with control flow.
- Read perm's can be split into "smaller" ones.

Permission Semantics

- In order to access mutable state, we need a
  - WRITE permission to write, 1
  - READ permission to read.

  Fractional permissions unify these.
- Permissions cannot be copied, only passed along with control flow.
- Read perm's can be split into "smaller" ones.

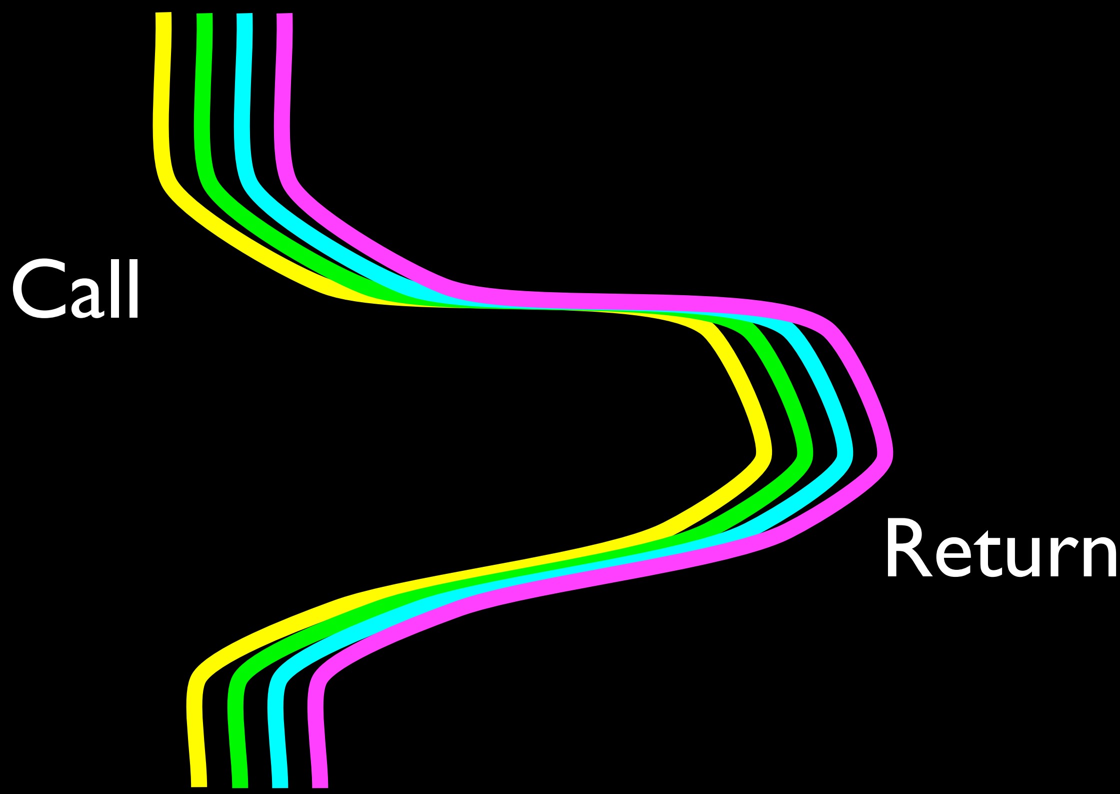
Permission Semantics

- In order to access mutable state, we need a
  - WRITE permission to write, 1
  - READ permission to read. <1

  Fractional permissions unify these.
- Permissions cannot be copied, only passed along with control flow.
- Read perm's can be split into "smaller" ones.

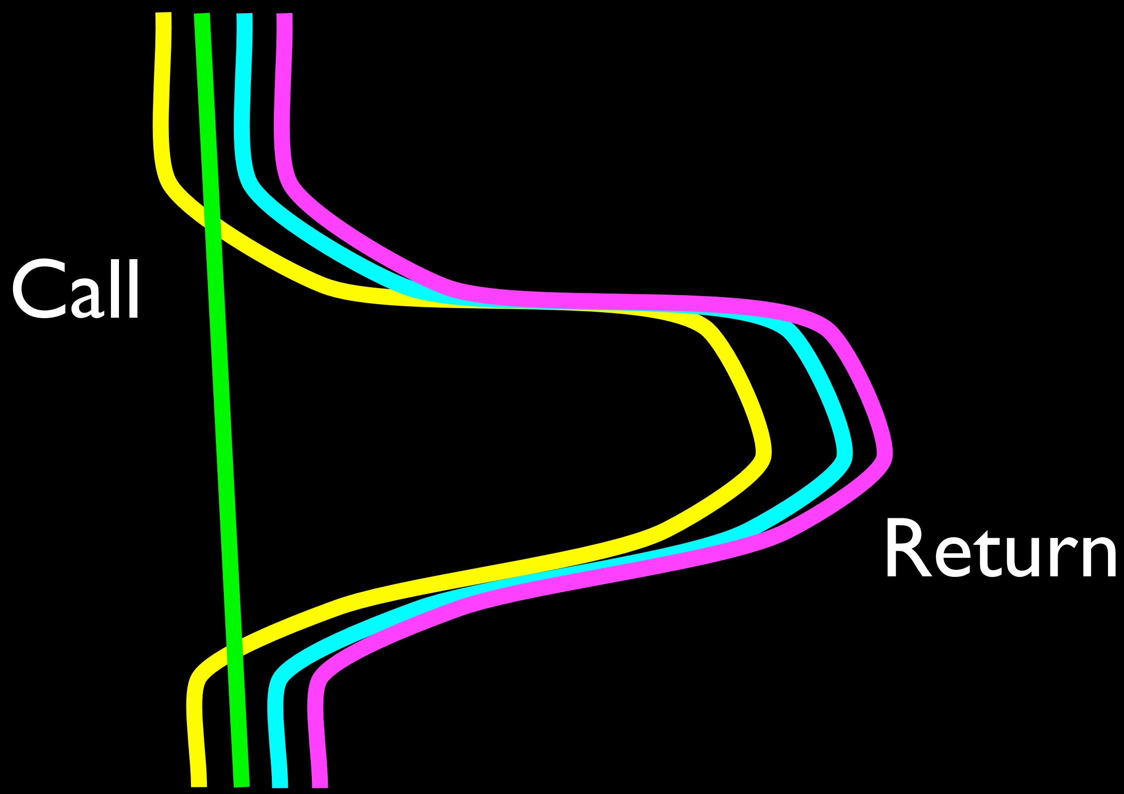
Permission Idioms (1)

- Single-threaded chaining: pass along all permissions with control-flow.



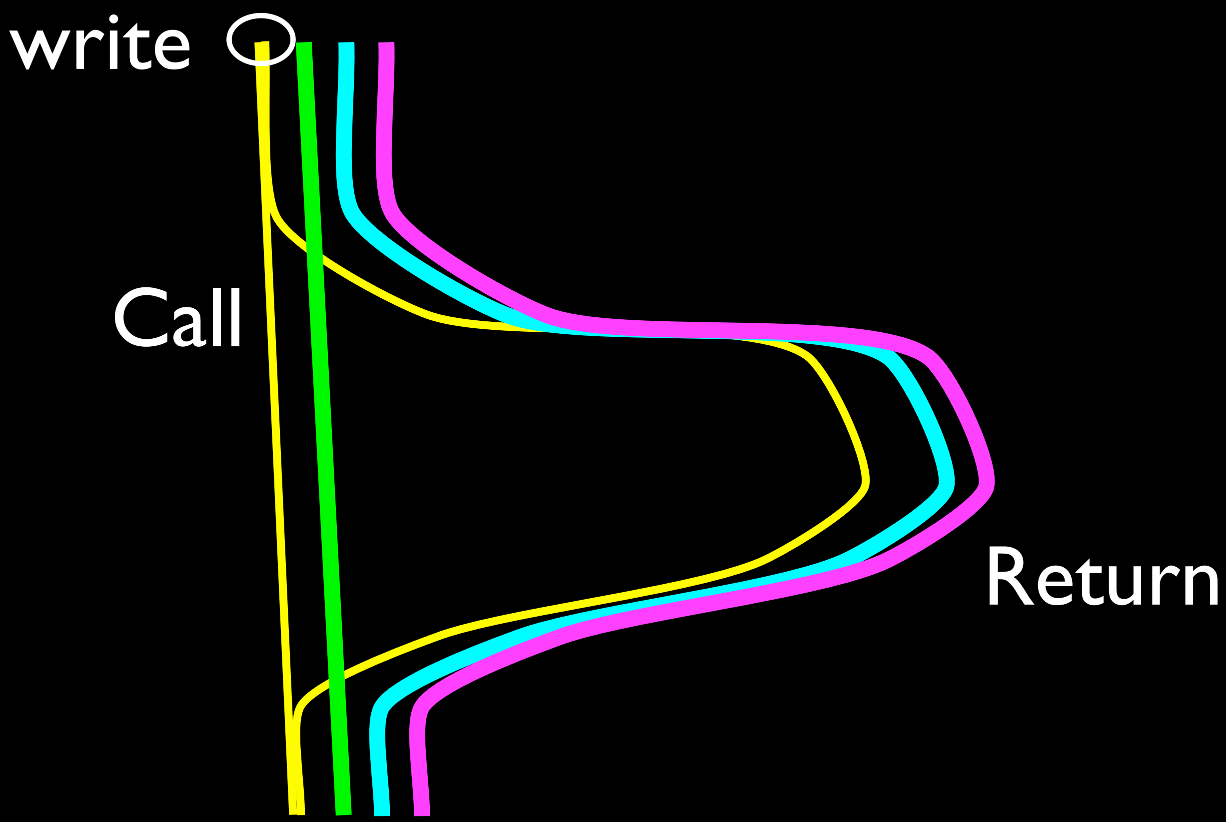
Permission Idioms (2)

- Framing: withhold some permissions before calling a method, add after call returns.



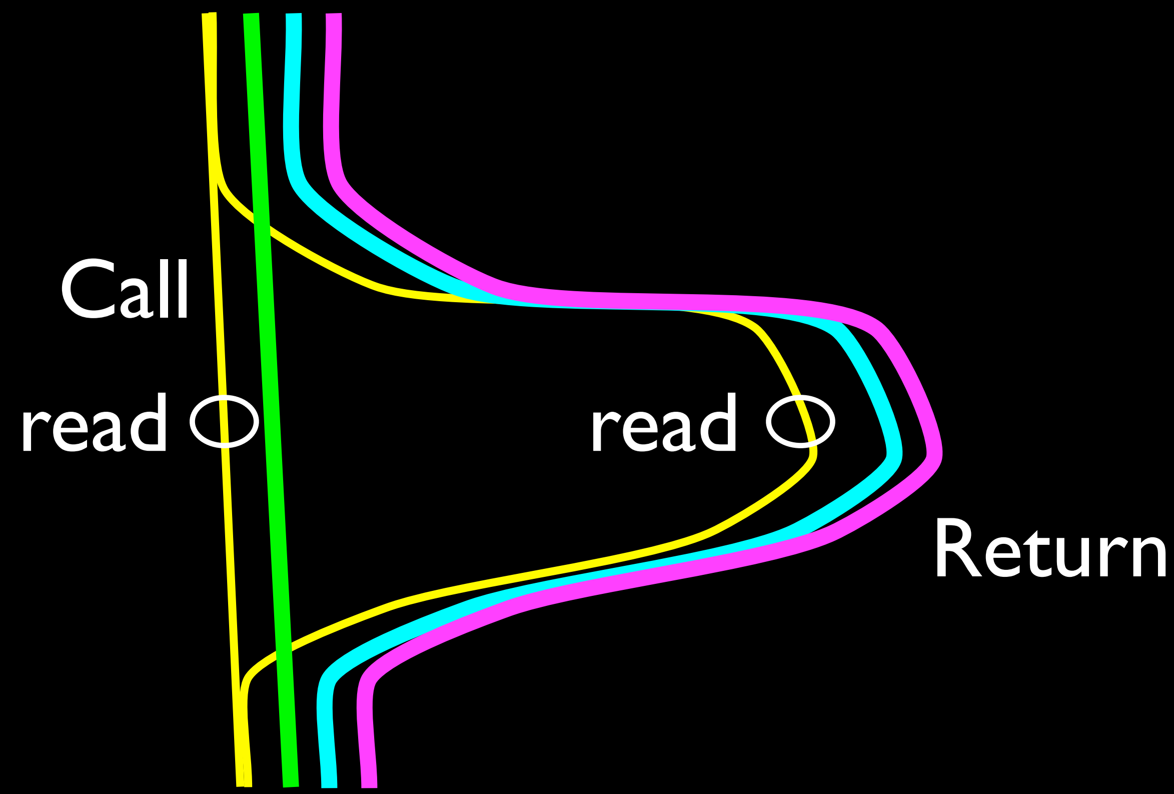
Permission Idioms (2)

- Framing: withhold some permissions before calling a method, add after call returns.



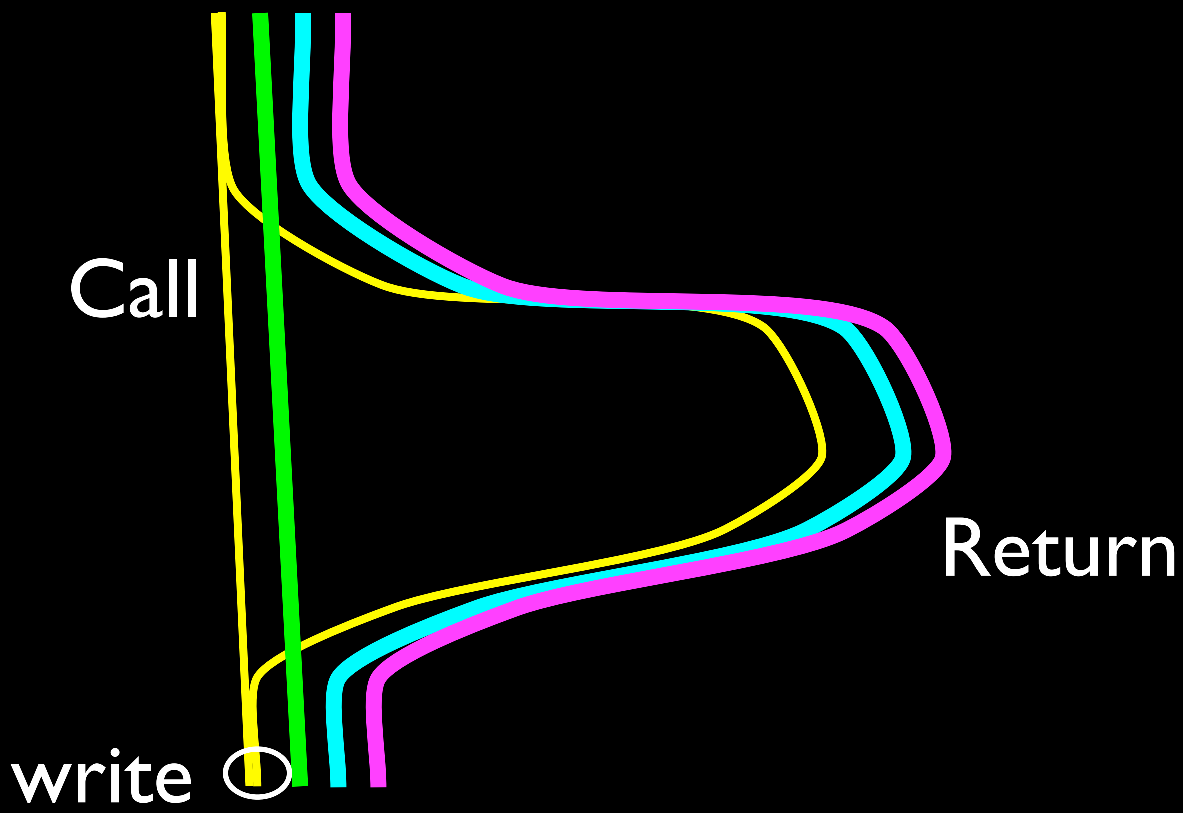
Permission Idioms (2)

- Framing: withhold some permissions before calling a method, add after call returns.



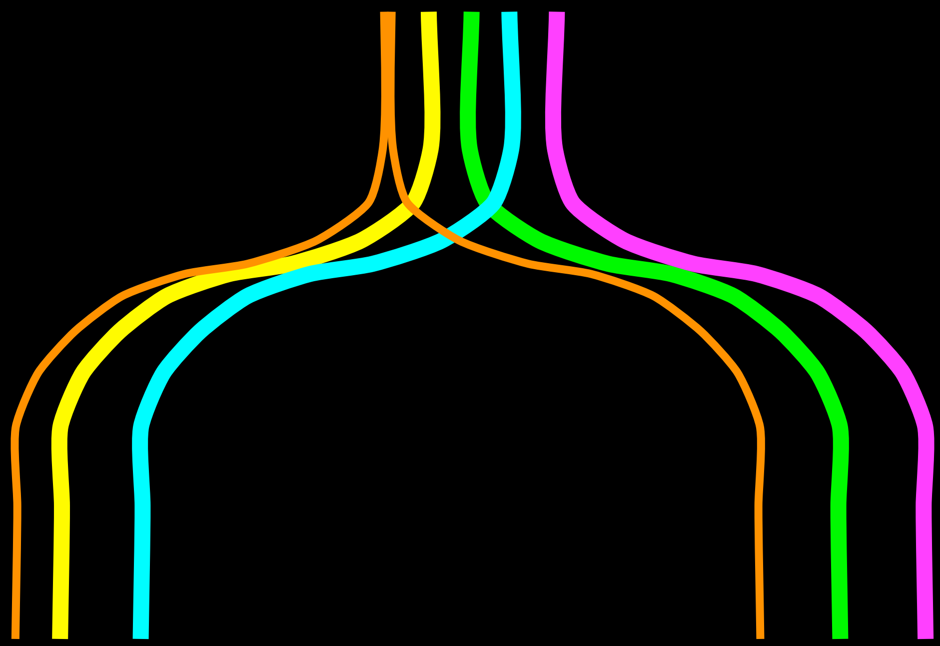
Permission Idioms (2)

- Framing: withhold some permissions before calling a method, add after call returns.



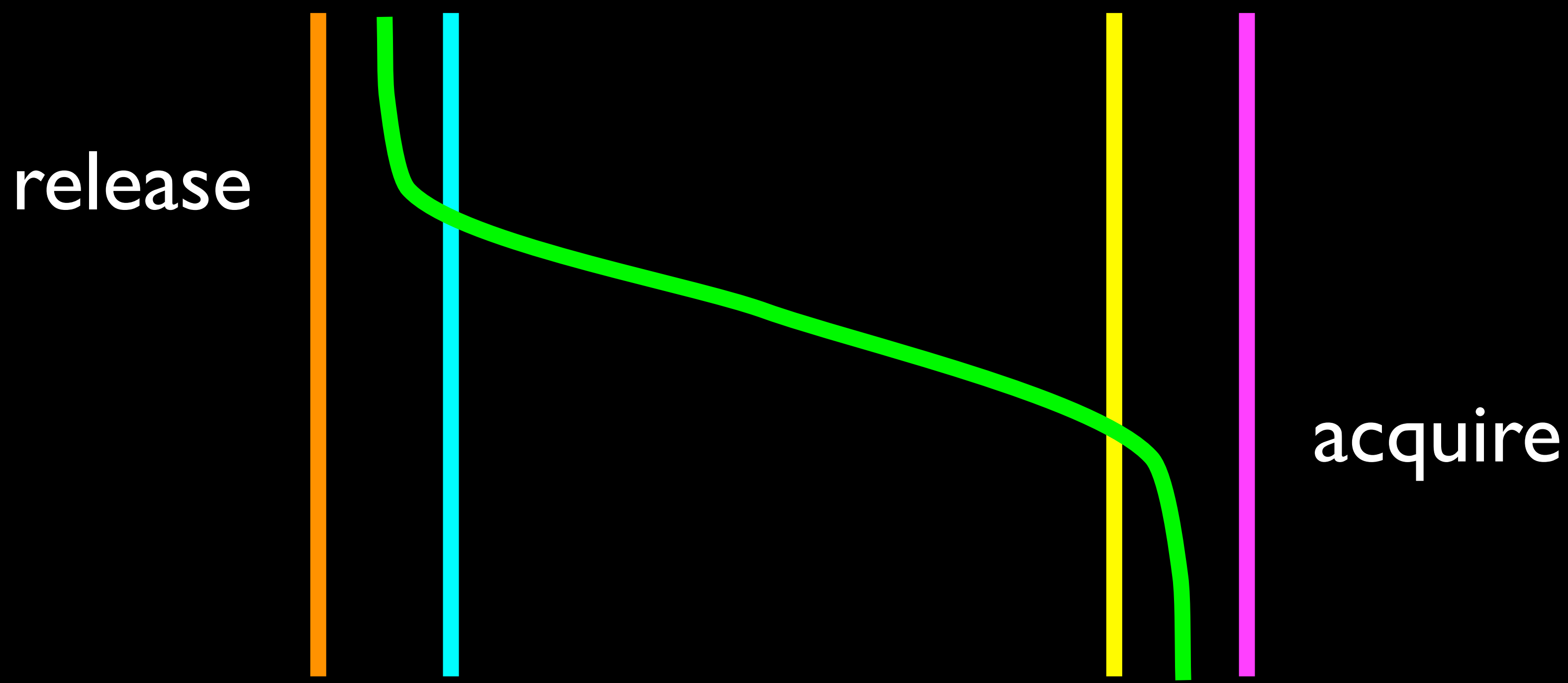
Permission Idioms (3)

- Fork: split permissions among threads



Permission Idioms (4)

- Transfer: pass permissions through synchronization points.



Permission Packaging

- Capability: pointer packaged with permission to access its contents.
  - effectively unique: aliases cannot be used.
- Self-framed assertion: program property/ invariant packaged with permission to access state described.
  - unframed properties are "ineffective"; they cannot be checked.

Permission Analysis

- Static analysis to determine whether permissions are always present.
- Sound analysis + program accepted → permissions can be ignored dynamically.
- Modularity requires description of input/output permissions of a method call.

Permission Analysis

- Static analysis to determine whether permissions are always present.
- Sound analysis + program accepted → permissions can be ignored dynamically.
- Modularity requires description of input/output permissions of a method call.

Annotations

Basic Annotations

- Method effects: perm's passed in and out
  - read (e.g. `reads this.f, arg.g`)
  - write

  Permissions are returned even upon abrupt termination.

- Immutable: read perm's passed one-way
- Unique: write perm's passed one-way

Abstraction (1)

- For modularity, we need annotations.
- For modularity, annotations need abstraction
  - we don’t want to list all (private?) fields
- What abstractions are appropriate?

Abstraction (1)

- For modularity, we need annotations.
- For modularity, annotations need abstraction
  - we don't want to list all (private?) fields
- What abstractions are appropriate?

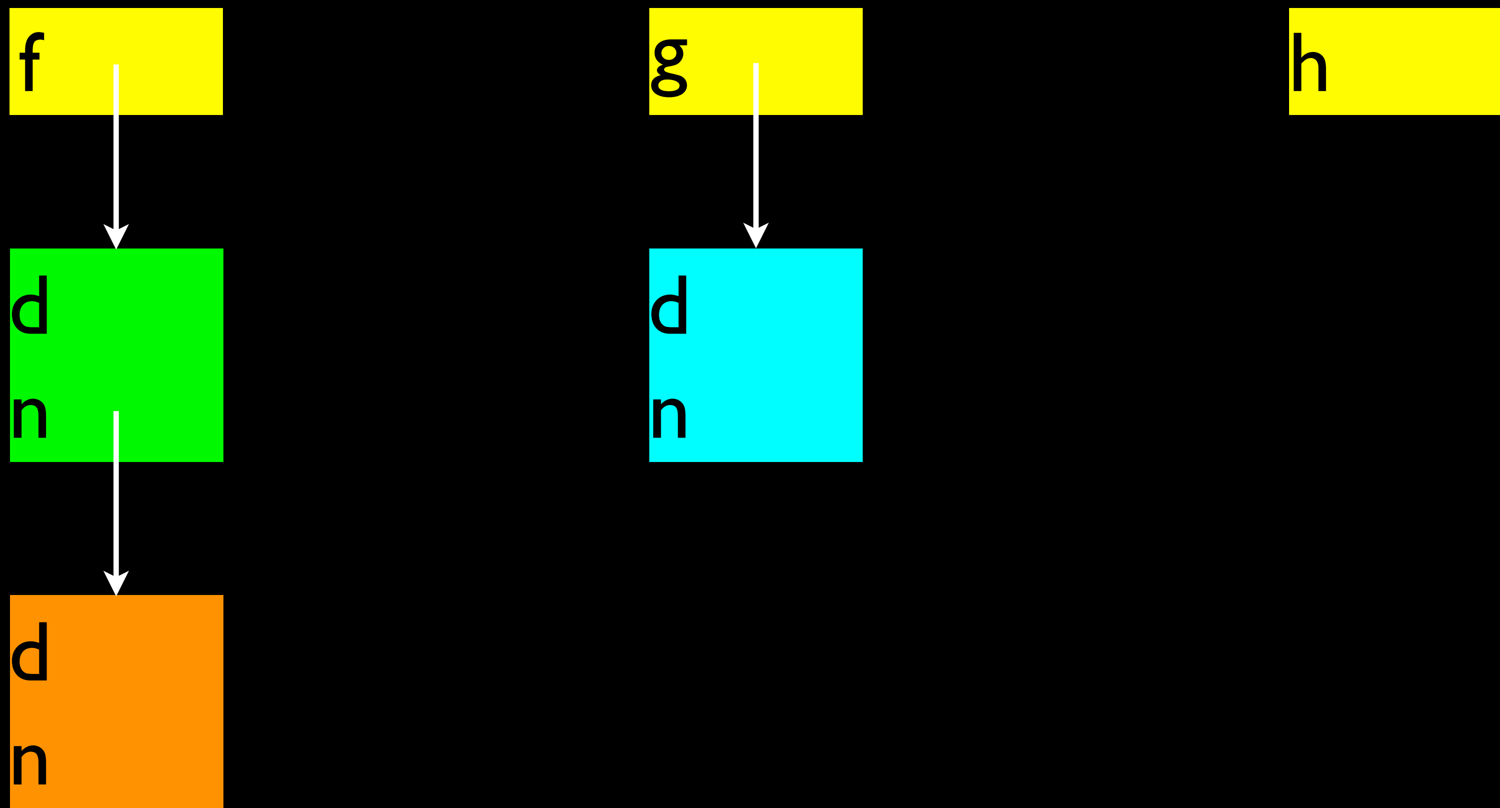
Regions / Data Groups

[Greenhouse&Boyland 1999, Leino 1998]

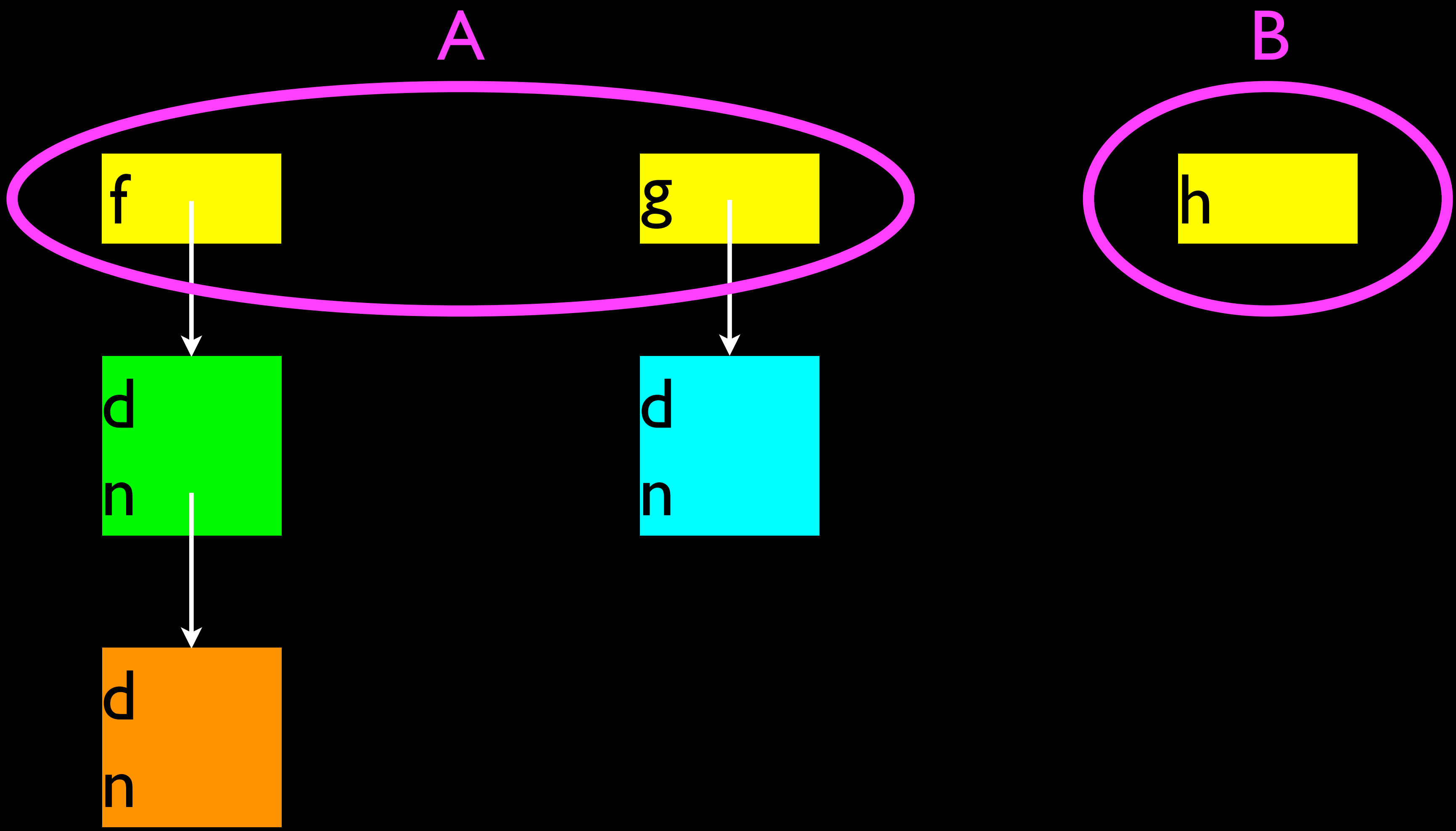
Abstraction (2)

- Internal Objects (e.g. Nodes in a TreeMap)
  - Option 1: Ownership
  - Option 2: Uniqueness
- Concurrency Related
  - Transfer through locks / volatiles
  - Thread-local objects

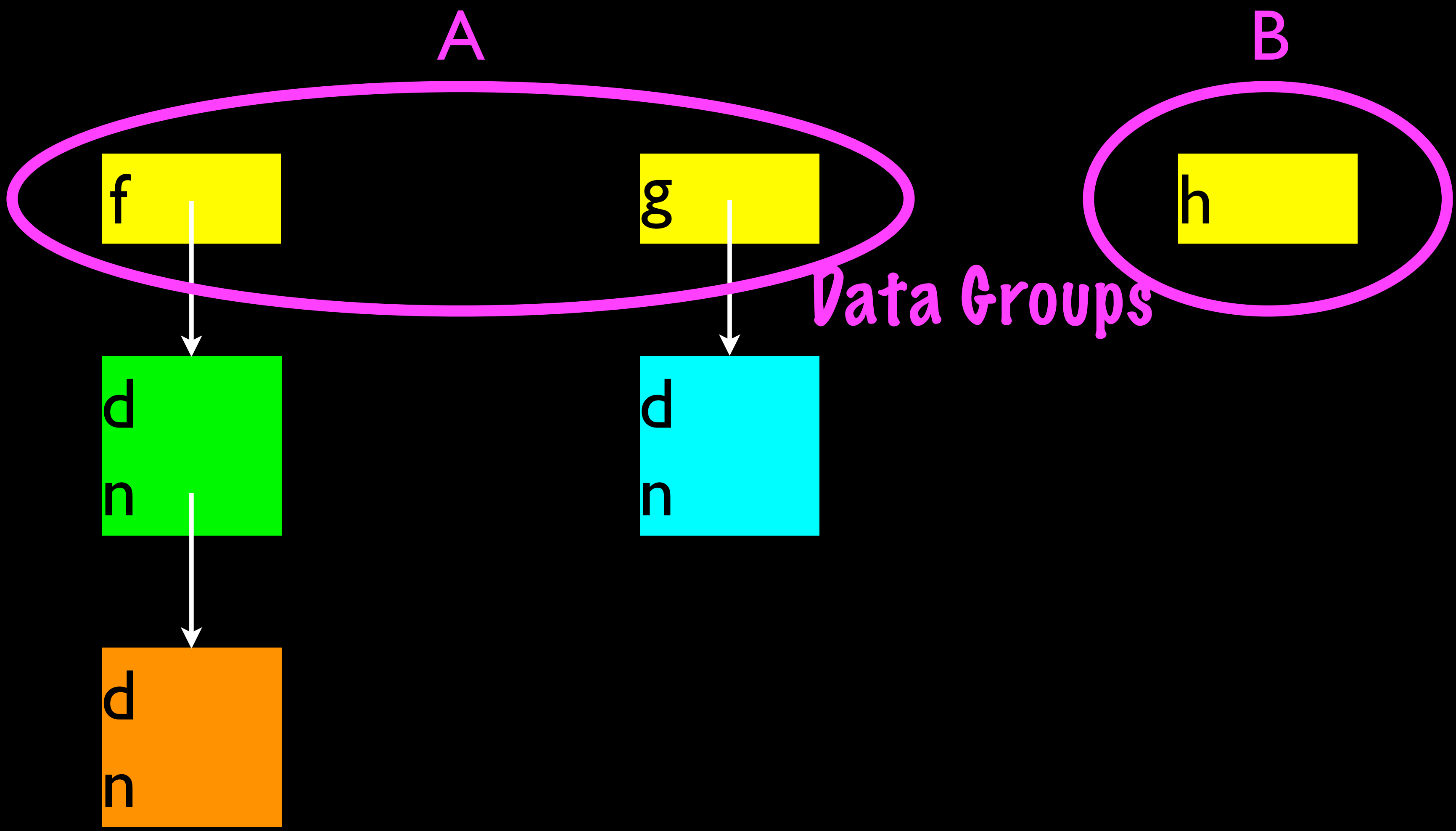
Two Dimensions



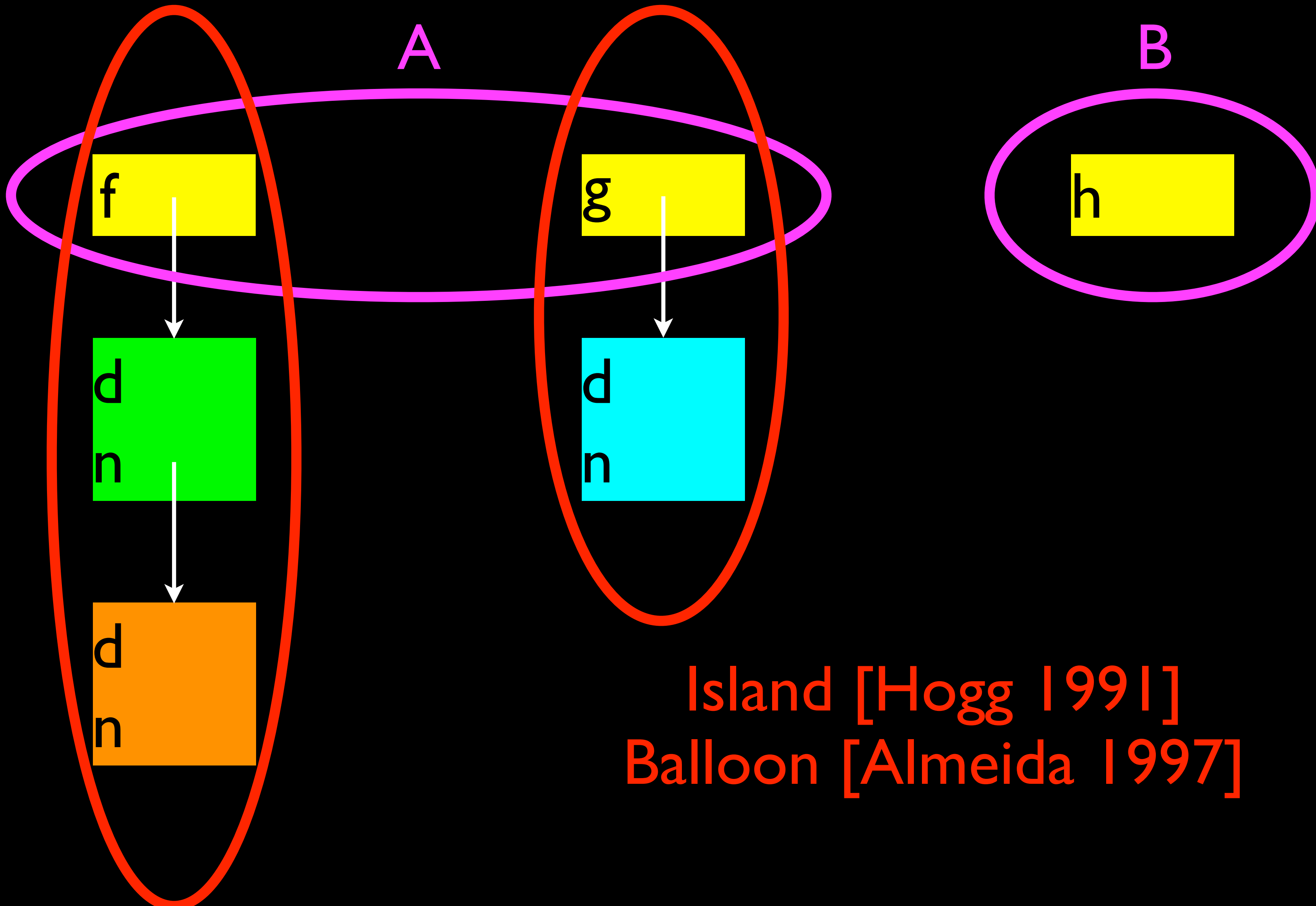
Two Dimensions



Two Dimensions

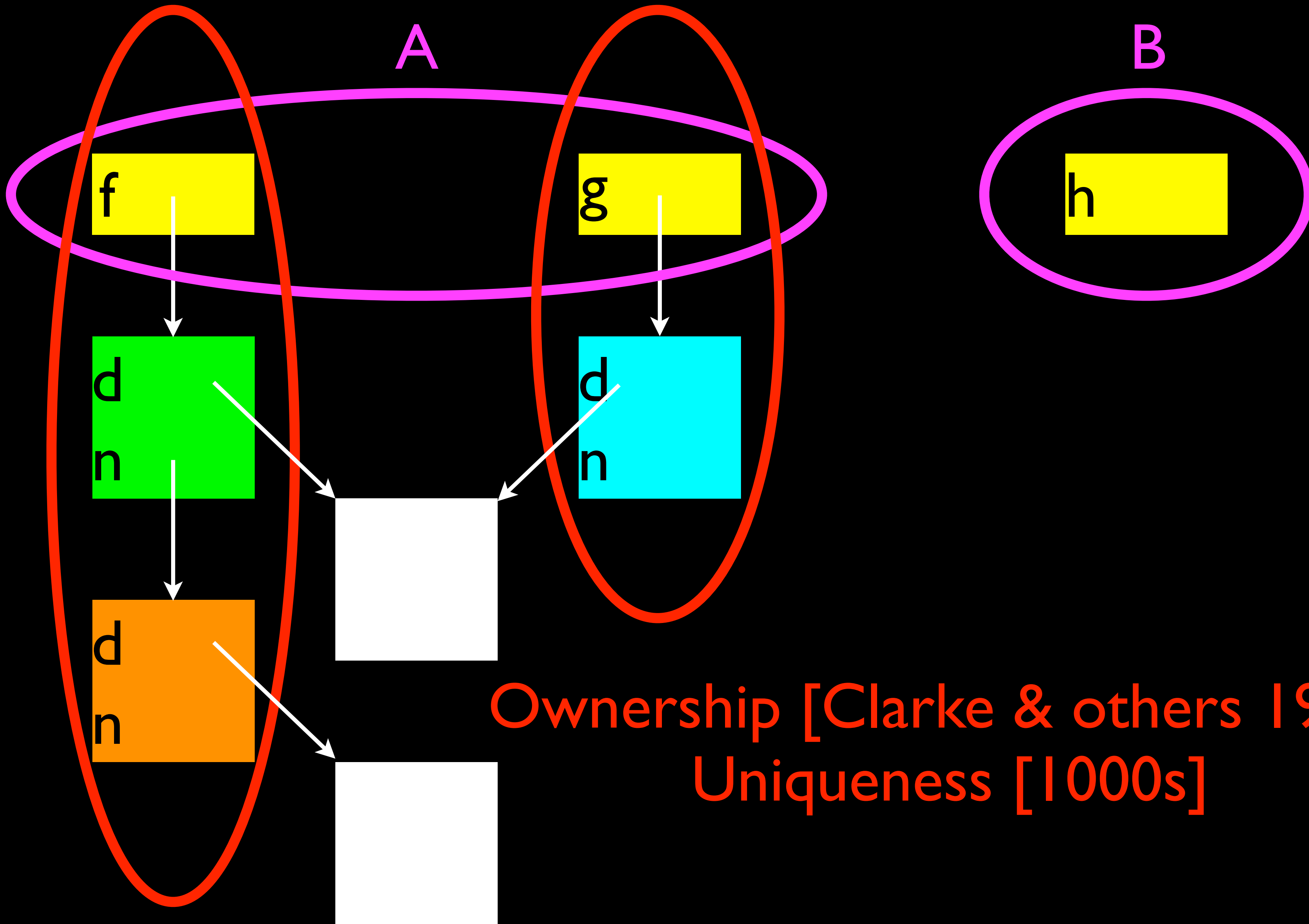


Two Dimensions



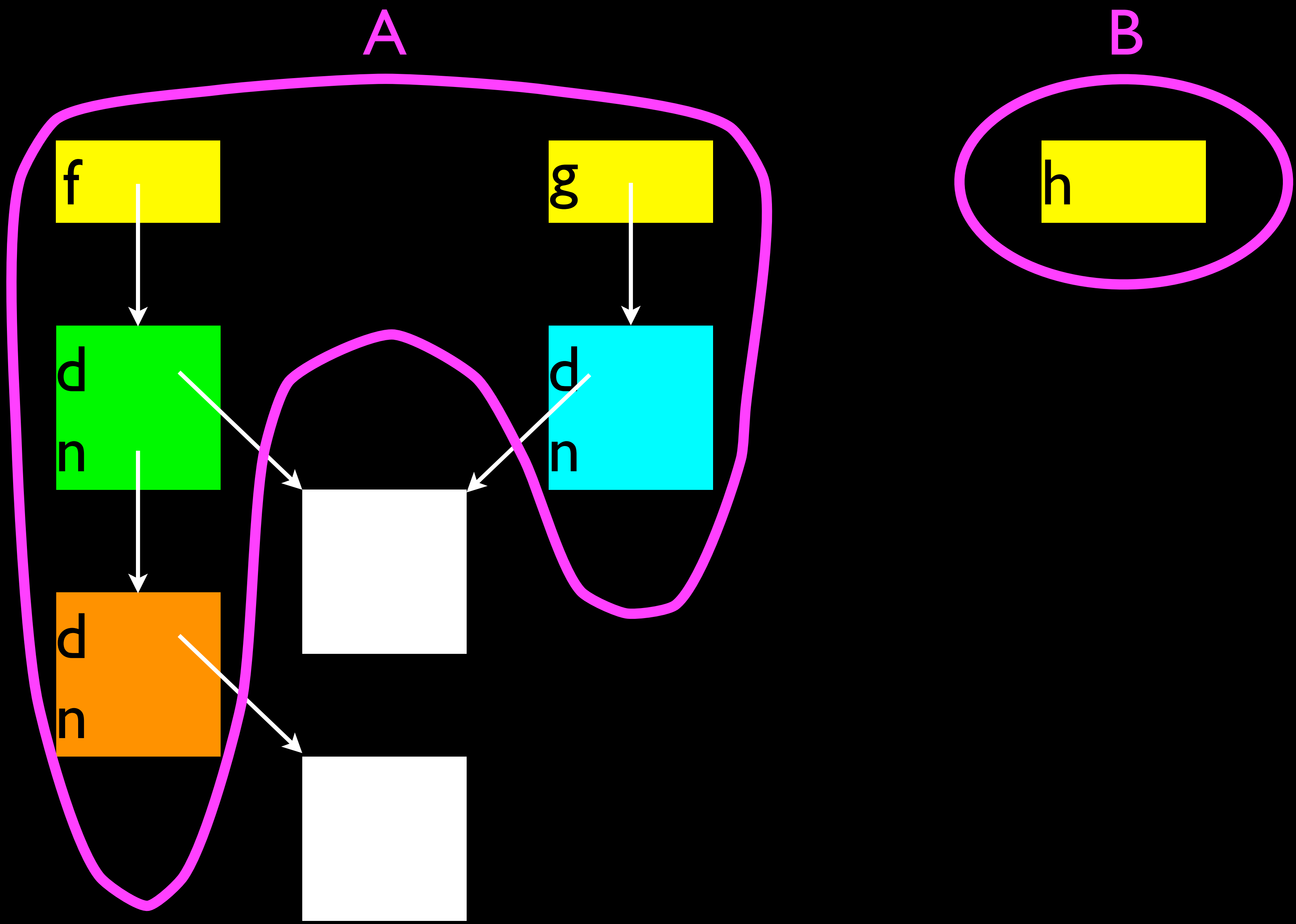
Island [Hogg 1991]
Balloon [Almeida 1997]

Two Dimensions



Ownership [Clarke & others 1998]
Uniqueness [1000s]

Two Dimensions



My Permission System

- (Positive) Fractions (with +, * and /)
- Packaging using existentials; encumberance with (linear) implications.
- Nesting: X < Y, a generalization of

1. Adoption [Fähndrich & DeLine 2002]
2. (effective) Ownership
3. Data Groups

My Permission System

- (Positive) Fractions (with +, * and /)
- Packaging using existentials; encumberance with (linear) implications.
- Nesting: X < Y, a generalization of

  1. Adoption [Fähndrich & DeLine 2002]
  2. (effective) Ownership
  3. Data Groups

Fact of nesting is “nonlinear”, that is, persistent/duplicable

Object Invariants

- Self-Framed Assertion P(r), e.g. $\exists n \cdot r.x \rightarrow n$
- Nesting fact: P(r)<r.All

1. If you have permission to the state (r.All), then you can access the invariant, including permissions to the fields involved. During access, you temporarily give up r.All
2. If you don't have permission to the state, you know nothing.

Object Invariants

- Self-Framed Assertion P(r), e.g. $\exists n \cdot r.x \rightarrow n$
- Nesting fact: P(r)<r.All
  1. If you have permission to the state (r.All), then you can access the invariant, including permissions to the fields involved. During access, you ~~temporarily give up~~ encumber r.All
  2. If you don't have permission to the state, you know nothing.

Case Studies

- Immutable Compounds
- Collections and Iterators
- GUI Event Thread
- Multi-Thread Broadcast
- Thread Communication with `volatile`

Immutable Compounds

```
class Period {
  final Time start;
  final Duration length;
  public Period(Time t, Duration l) {
    check for errors (null, empty)
    time = t;
    length = l;
  }
  ...
}
```

- We want everything immutable.

Partial Ownership

- Designate a special owner for immutables:

  System.Immutable ("I" for short)

- An immutable object x has partial nesting:

  $\exists q \cdot (qx.\mathrm{All}) \prec I$

- Every method passed a non-zero fraction:

  $\exists q \cdot qI$

Partial Ownership

- Designate a special owner for immutables:

  System.Immutable ("I" for short)

- An immutable object x has partial nesting:

  $$\exists q \cdot (qx.\mathrm{All}) \prec I$$

- Every method passed a non-zero fraction:

  $$\exists q \cdot qI$$ **Implicitly!**

Access Immutable State

- Get fraction of fraction of nested state

$$q_1 q_2 x.\mathrm{All}$$

- Get field read permissions from there.
- Rational numbers can get arbitrarily small.

Constructing Immutable

- Immutable constructor contract:
  - pre: *write permission for all fields + read permission for all parameter objects*
  - post: $(invariant \prec \texttt{this}.\mathrm{All}) + (\exists q \cdot q\texttt{this}.\mathrm{All} \prec I)$

Constructing Immutable

- Immutable constructor contract:
  - pre: *write permission for all fields + read permission for all parameter objects*
  - post: $(invariant \prec \mathtt{this}.\mathrm{All}) + (\exists q \cdot q\mathtt{this}.\mathrm{All} \prec I)$

Effective invariant:
true when one has (read) access

Collections & Iterators

```
for (Iterator<String> it = c.iterator();
     it.hasNext();) {
    String v = it.next();
    ...
    if (...) {
        it.remove();
    }
}
```

- While any (other) iterator is active, collection must not be mutated (or else possible CME).

Collections & Iterators

```
for (Iterator<String> it = c.iterator();
     it.hasNext();) {
    String v = it.next();
    ...
    if (...) {
        it.remove();
    }
}
```

- While any (other) iterator is active, collection must not be mutated (or else possible CME).

(dual to normal restriction)

Collections & Iterators

```
for (Iterator<String> it = c.iterator();
      it.hasNext();) {
     String v = it.next();
     ...
     if (...) {
         it.remove();
     }
}
```

can read c but
not write c here

- While any (other) iterator is active, collection must not be mutated (or else possible CME).

Collections & Iterators

```
for (Iterator<String> it = c.iterator();
      it.hasNext();) {
    String v = it.next();
    ...
    if (...) {
        it.remove();
    }
}
```

No other iterator
can be active here.

- While any (other) iterator is active, collection must not be mutated (or else possible CME).

Collection Permissions

- ρ is a collection of *E* and we have access

$$\mathrm{Collection}(\rho, E) + \rho.\mathrm{All}$$

$$C \equiv \mathrm{Collection}(\mathtt{c}, \mathrm{String}) + \mathtt{c}.\mathrm{All}$$

- ρ is a q iterator of *E* on ρ' and we have access

$$\mathrm{Iterator}(\rho, q, \rho', E) + \rho.\mathrm{All}$$

$$I_q \equiv \mathrm{Iterator}(\mathrm{it}, q, \mathtt{c}, \mathrm{String}) + \mathtt{it}.\mathrm{All}$$

Collection Permissions

- ρ is a collection of *E* and we have access

$$\text{Collection}(\rho, E) + \rho.\text{All}$$

higher order

$$C \equiv \text{Collection}(\texttt{c}, \text{String}) + \texttt{c}.\text{All}$$

- ρ is a q iterator of *E* on ρ' and we have access

$$\text{Iterator}(\rho, q, \rho', E) + \rho.\text{All}$$

$$I_q \equiv \text{Iterator}(\text{it}, q, \texttt{c}, \text{String}) + \texttt{it}.\text{All}$$

`iterator()` encumbers

- Permission contract of **`iterator()`**
  - before: qC
  - after: $I_q + (I_q \multimap qC)$
- Expresses restriction on iterator as a restriction on the collection: we can't get permission back until we give up iterator access.

`iterator()` encumbers

- Permission contract of **`iterator()`**
  - before: qC
  - after: $I_q + (I_q \multimap qC)$

write ($q = 1$) or read ($q < 1$)
access to **`c`**

- Expresses restriction on iterator as a restriction on the collection: we can't get permission back until we give up iterator access.

`iterator()` encumbers

- Permission contract of **`iterator()`**
  - before: qC
  - after: $I_q + (I_q \multimap qC)$

linear implication: consumes the premise

- Expresses restriction on iterator as a restriction on the collection: we can't get permission back until we give up iterator access.

`iterator()` encumbers

- Permission contract of **`iterator()`**
  - before: qC
  - after: $I_q + (I_q \multimap qC)$
- Expresses restriction on iterator as a restriction on the collection: we can't get permission back until we give up iterator access.

Iterator methods

- hasNext() reads this.All
  - before/after: $q' I_q$
- next() writes this.All
  - before/after: I_q
- remove writes this.All, requires “write” iter
  - before/after: I_1

Temporary Read Access

- We need a way to permit read access to the collection while not removing elements.

$$I_1 \rightsquigarrow I_{\frac{1}{2}} + \frac{1}{2}C + \left(I_{\frac{1}{2}} + \frac{1}{2}C -\!\!+ I_1 \right)$$

Temporary Read Access

- We need a way to permit read access to the collection while not removing elements.

$$I_1 \rightsquigarrow I_{\frac{1}{2}} + \frac{1}{2}C + \left(I_{\frac{1}{2}} + \frac{1}{2}C \mathrel{-\!\!+} I_1\right)$$

(Requires an extension)

Swing Event Thread

```
SwingUtilities.invokeLater(new Runnable() {
      JFrame f = new MyFrame();
      f.setSize(500,300);
      f.setVisible(true);
});
```

- Only Swing event thread can
  1. create instances of GUI classes;
  2. mutate state of GUI instances.
- Not a fixed thread. “setVisible” may yield control.

GUI owns its state

- Designate a global field as owner of all GUI state, e.g. Swing.GUI (“G” for short)
- Every GUI class constructor has contract:
  - pre: G + *write permission to all fields*
  - post: G + ($inv't$ < this.All) + (this.All < G)
- Can’t access state of instances without G.

GUI owns its state

- Designate a global field as owner of all GUI state, e.g. Swing.GUI ("G" for short)
- Every GUI class constructor has contract:
  - pre: G + *write permission to all fields*
  - post: G + ($inv't$ < this.All) + (this.All < G)

Object "owned" by GUI

- Can't access state of instances without G.

Chaining of GUI

- Any method that could lead to yielding control will require G including state of any GUI classes (hence invariants in effect).
  - This prevents call-back problems.
- If a new thread is given responsibility for the GUI, it is passed the GUI permission.

Runnable is Generic

- Java interface Runnable is generic in effect:

```
interface Runnable<effect E> {
    public void run() writes E
}
```

- SwingUtilities.invokeLater requires

  `Runnable<` G `+E>` task

Runnable is Generic

- Java interface Runnable is generic in effect:

```
interface Runnable<effect E> {
    public void run() writes E
}
```

- SwingUtilities.invokeLater requires

  `Runnable<` G `+E> task`

  Permissions from caller

Multi-Thread Broadcast

```
void observe(Observer<T> ob, int i) ... {
  for (;;++i) {
     T elem;
     synchronized (cont) {
       while (i >= cont.size()){
         cont.wait();
       }
       elem = cont.get(i);
     }
     ob.update(this,elem);
   }
 }
```

Multi-Thread Broadcast

Basis of distributed COMMAND pattern

```
void observe(Observer<T> ob, int i) ... {
  for (;;++i) {
     T elem;
     synchronized (cont) {
       while (i >= cont.size()){
         cont.wait();
       }
       elem = cont.get(i);
     }
     ob.update(this,elem);
   }
 }
```

Multi-Thread Broadcast

```
void observe(Observer<T> ob, int i) ... {
  for (;;++i) {
     T elem;
     synchronized (cont) {
       while (i >= cont.size()){
         cont.wait();
       }
       elem = cont.get(i);
     }
     ob.update(this,elem);
   }
 }
```

Could throw
InterruptedException

Multi-Thread Broadcast

```
void observe(Observer<T> ob, int i) ... {
  for (;;++i) {
     T elem;
     synchronized (cont) {
       while (i >= cont.size()){
         cont.wait();
       }
       elem = cont.get(i);
     }
     ob.update(this,elem);
   }
 }
```

private list

used as mutex

Multi-Thread Broadcast

```
void observe(Observer<T> ob, int i) ... {
  for (;;++i) {
     T elem;
     synchronized (cont) {
       while (i >= cont.size()){
         cont.wait();
       }
       elem = cont.get(i);
     }
     ob.update(this,elem);
   }
 }
```

During this call,
add() could add an element
and notifyAll()

Multi-Thread Broadcast

```
void observe(Observer<T> ob, int i) ... {
  for (;;++i) {
     T elem;
     synchronized (cont) {
       while (i >= cont.size()){
         cont.wait();
       }
       elem = cont.get(i);
     }
     ob.update(this,elem);
   }
 }
```

observer called while
lock *not* held

RT owns mutexes

- Designate a special owner for mutexes:

  System.Lock (“L” for short)

- Synchronization only allowed on x if

  x.All < L

  Gives access to x.All (unless re-entered).

- No one ever gets access to L.

Mutex Methods

- wait() writes this.All, requires this.All $< L$
  - ensures invariants re-established.
- notify() writes this.All, requires this.All $< L$
  - just to ensure we have acquired mutex.
- lock()
  - pre: this.All $< L$ + (*lock-order* ? 0 : this.All)
  - post: this.All

Mutex Methods

- wait() writes this.All, requires this.All $< L$
  - ensures invariants re-established.
- notify() writes this.All, requires this.All $< L$
  - just to ensure we have acquired mutex.
- lock()
  - pre: this.All $< L$ + (*lock-order* ? 0 : this.All)
  - post: this.All

deadlock check:
current lock level is less than this

Mutex Methods

- wait() writes this.All, requires this.All $< L$
  - ensures invariants re-established.
- notify() writes this.All, requires this.All $< L$
  - just to ensure we have acquired mutex.
- lock()
  - pre: this.All $< L$ + (*lock-order* ? 0 : this.All)
  - post: this.All + (*lock-order*' ? (this.All $-\!\!+$ U) : U)

Mutex Methods

- wait() writes this.All, requires this.All $< L$
  - ensures invariants re-established.
- notify() writes this.All, requires this.All $< L$
  - just to ensure we have acquired mutex.
- lock()
  - pre: this.All $< L$ + (*lock-order* ? 0 : this.All)
  - post: this.All + (*lock-order'* ? (this.All $\dashv$ U) : U)

old lock-order

Mutex Methods

- wait() writes this.All, requires this.All $< L$
  - ensures invariants re-established.
- notify() writes this.All, requires this.All $< L$
  - just to ensure we have acquired mutex.
- lock()
  - pre: this.All $< L$ + (*lock-order* ? 0 : this.All)
  - post: this.All + (*lock-order'* ? (this.All $-\!\!+$ U) : U)

Token to permit **`unlock()`**

Observer is Generic

- As with **`Runnable`**, this interface also needs a generic effect parameter.
- And **`observe()`** is generic as well:
  - permissions passed along to **`update()`**.

Volatile Communication

```
private volatile List<Connection> connex;
public void addConnection(Connection x){
    List<Connection> newL =
        new ArrayList<>(connex);
    newL.add(x);
    connex = newL;
}
public void paintComponent(Graphics g) {
   for (Connection c : connex) { ... }
}
```

- List struct. must not change, elements may.

Volatile Communication

Called by thread monitoring a ServerSocket

```
private volatile List<Connection> connex;
public void addConnection(Connection x){
    List<Connection> newL =
        new ArrayList<>(connex);
    newL.add(x);
    connex = newL;
}
public void paintComponent(Graphics g) {
   for (Connection c : connex) { ... }
}
```

- List struct. must not change, elements may.

Volatile Communication

```
private volatile List<Connection> connex;
public void addConnection(Connection x){
    List<Connection> newL =
        new ArrayList<>(connex);
    newL.add(x);
    connex = newL;
}
public void paintComponent(Graphics g) {
   for (Connection c : connex) { ... }
}
```

a fresh object

- List struct. must not change, elements may.

Volatile Communication

```
private volatile List<Connection> connex;
public void addConnection(Connection x){
    List<Connection> newL =
        new ArrayList<>(connex);
    newL.add(x);
    connex = newL;
}
public void paintComponent(Graphics g) {
   for (Connection c : connex) { ... }
}
```

Called by GUI

- List struct. must not change, elements may.

Volatile Communication

```
private volatile List<Connection> connex;
public void addConnection(Connection x){
    List<Connection> newL =
        new ArrayList<>(connex);
    newL.add(x);
    connex = newL;
}
public void paintComponent(Graphics g) {
   for (Connection c : connex) { ... }
}
```

(Connections may be mutated elsewhere in GUI)

- List struct. must not change, elements may.

Handling Volatile

- Volatile fields may be read/updated without any permission from any thread at any time.
- The value written to a volatile field must satisfy the field's invariant.
- The value read from a volatile field can be assumed to meet the field's invariant.
- The invariant must be a "fact" (duplicable).

Volatile: Solution

- "connex" invariant: immutable list of connection objects owned by GUI.
- "Connection x" added to list is owned by GUI (any unique object can be made to fit).
- GUI code can traverse (immutable) list and update fields of objects without interference.

Conclusion

- Permissions make state access explicit.
- Static permission analysis requires an expressive static permission system.
- Even without analysis, thinking about permissions makes software cleaner.
- We have "faith" that a well-written program has a reasonably simple explanation of its permission behavior.

Questions?

- See theoretical paper (w/ mechanized proof):

  http://www.cs.uwm.edu/~boyland/papers/frac-nesting.html

Aspect-Oriented Permissions?

- Adding fields/behavior using those fields is handled by adding to invariant (e.g. a new data group).
- Adding new synchronization is less modular because requirements on lock-level.