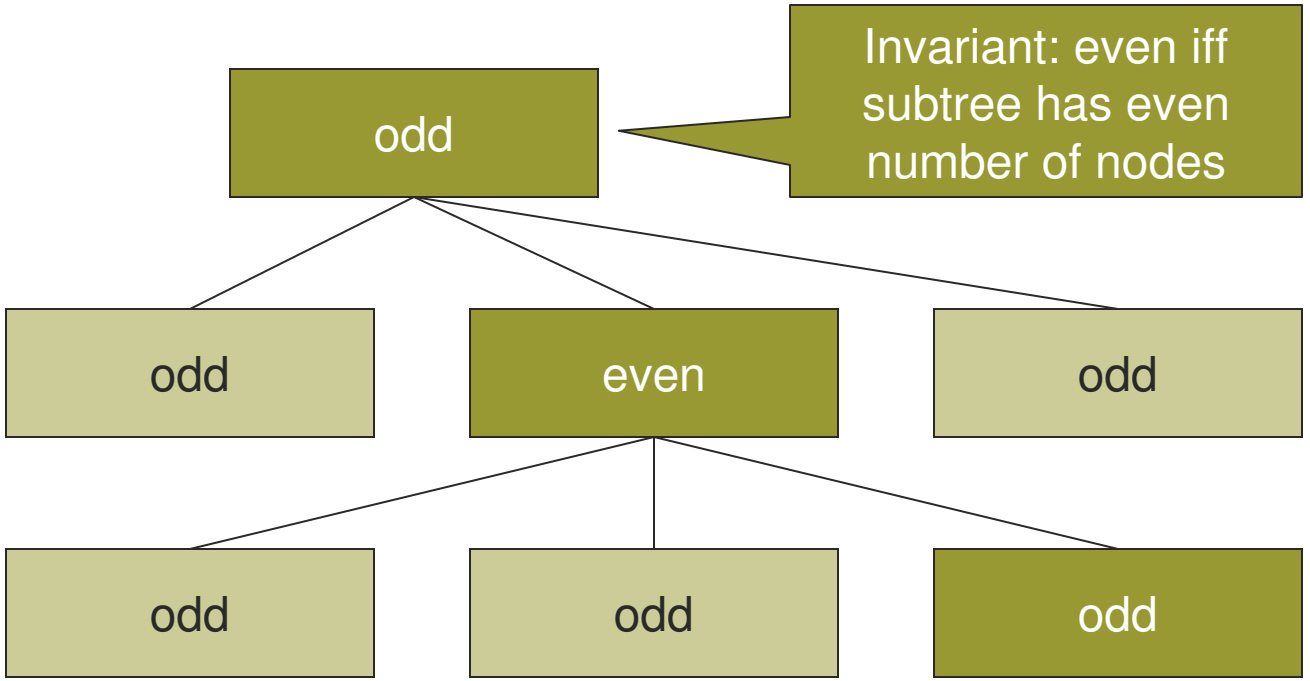
Permissions to Specify the Composite Design Pattern

Kevin Bierhoff & Jonathan Aldrich
Carnegie Mellon University
SAVCBS 9 November 2008

Composites: Whole-part relationships as object trees



Two conflicting goals:
Nodes depending on subnodes and adding children to any node

We will use permissions to force parent updates

- Two conflicting goals
  - Nodes depend on their subnodes in invariants
  - Allow adding new children to any node
- Solution: Do not add children without parents' knowledge
  - Update parent after adding new child
- Permissions to enforce parent update
  - Distribute permission to update node between node and its immediate parent

Sample Composite tracks even vs. odd number of nodes

```
public final class Composite {
boolean odd;
Composite parent, left, right;

public Composite() { odd = true; parent = null; left = null; right = null; }

public void setLeft(Composite c) {
    c.parent = this; left = c;
    if(c.odd) {
        odd = ! odd;
        Composite p = parent;
        while(p != null) {
            p.odd = ! p.odd;
            p = p.parent;
        }
    }
}
```

```
public boolean odd() {
    return odd;
}
```

Access Permissions: typestate tracking under aliasing [OOPSLA '07]

Other references	Current reference	
	Read/write	Read-only
Read/write	**share**	**pure**
Read-only	**full**	**immutable**

- Consistency
  - 1 full and many pure
  - Many immutables and many pures
- Example: **full**(**this**, PARENT) **in** orphan

Current state

Permission for PARENT dimension

Fractions enable permission splitting and merging

Other references	Current reference	
	Read/write	Read-only
Read/write	**share**	**pure**
Read-only	**full**	**immutable**

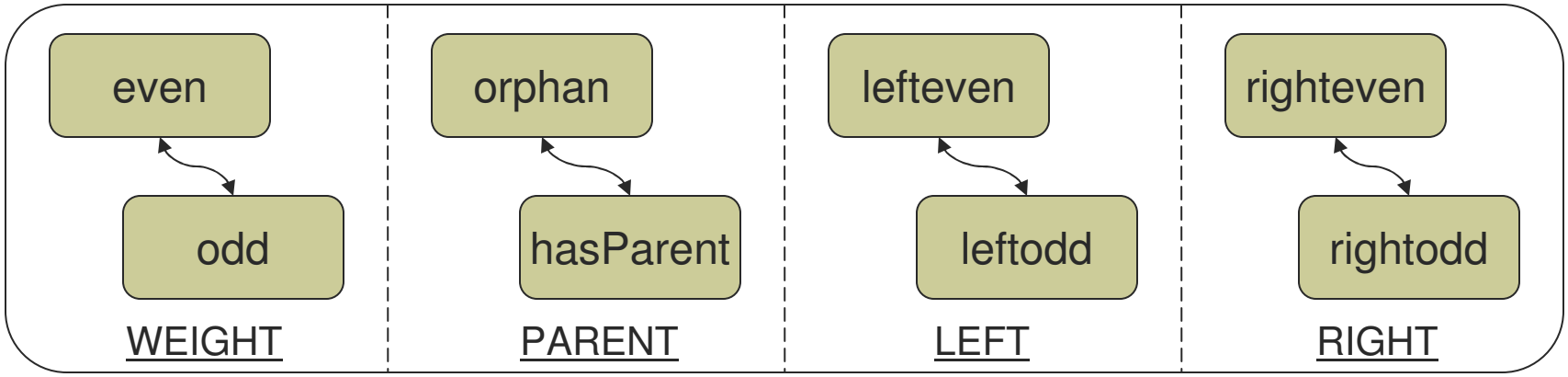
- Split permissions to introduce aliases
  - full(x, P) => full(x, P) ⊗ pure(x, P)
- Fractions allow permission merging [Boyland '03]
  - full(x, P) ⇔ immutable(x, P, ½) ⊗ immutable(x, P, ½)
  - Can recover write permission from read-only permissions!

Dimensions, states, and invariants

- Permissions for each dimension
- Invariants separate for each dimension
- State invariants only hold when in state

states WEIGHT = { even, odd }
states PARENT = { orphan, hasParent }
states LEFT = { lefteven, leftodd }
states RIGHT = { righteven, rightodd }

boolean odd; **in** WEIGHT;
Composite parent; **in** PARENT;
Composite left; **in** LEFT;
Composite right; **in** RIGHT;



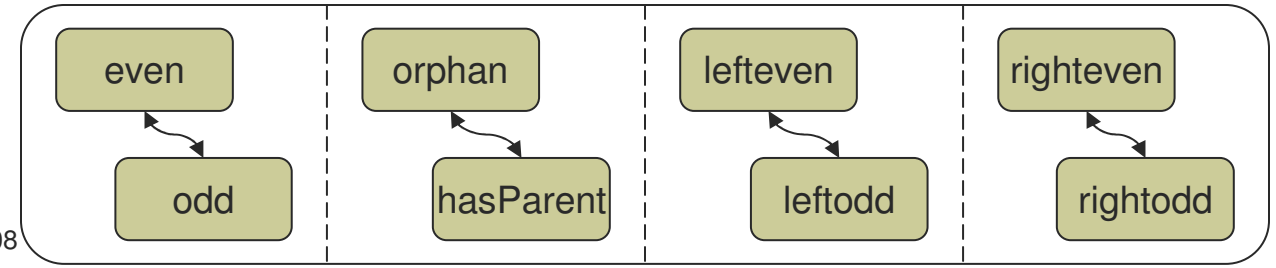
Linear logic for composing atomic predicates [Girard '87]

- Permissions are linear resources
- Multiplicative conjunction
  - $A \otimes B$ (similar to separation logic's $A * B$)
  - A and B at the same time
- Additive conjunction (internal choice)
  - A & B (similar to separation logic's $A \wedge B$)
  - A and B available, but must choose one or the other
- Disjunction (external choice)
  - $A \oplus B$ (similar to separation logic's $A \vee B$)
  - Either A or B non-deterministically—be ready for either
- Linear implication
  - $A \multimap B$ (similar to separation logic's $A -\!\!* B$)
  - B holds if A can be consumed

Invariants we care about

invariant lefteven : left = **null** ⊕ (left ≠ **null** ⊗ left **in** even);
invariant leftodd : left ≠ **null** ⊗ left **in** odd;
invariant righteven : right = **null** ⊕ (right ≠ **null** ⊗ right **in** even);
invariant rightodd : right ≠ **null** ⊗ right **in** odd;

invariant WEIGHT:
(odd = **false** ⊸ ((**this in** leftodd ⊗ **this in** righteven) ⊕
(**this in** lefteven ⊗ **this in** rightodd))) &
(odd = **true** ⊸ ((**this in** lefteven ⊗ **this in** righteven) ⊕
(**this in** leftodd ⊗ **this in** rightodd)));
invariant even : odd = **false**;
invariant odd : odd = **true**;



Insight: Use immutable permissions for dependencies

- Invariants indicate dependencies
  - on other dimensions of the same object
  - on dimensions in other objects
- Need immutable permissions in those invariants
  - Guarantees that values do not change without knowledge of dependent object
  - Full permission would work, too

Immutable permissions to guarantee dependencies

invariant WEIGHT : immutable(**this**, LEFT) ⊗ immutable(**this**, RIGHT);
invariant LEFT : left ≠ **null** —∘ immutable(left, WEIGHT);
invariant RIGHT : right ≠ **null** —∘ immutable(right, WEIGHT);

invariant PARENT : parent ≠ **this**;
invariant PARENT :
parent ≠ **null** —∘ ((immutable(parent, LEFT) ⊗ parent.left = **this**) ⊕
(immutable(parent, RIGHT) ⊗ parent.right = **this**)));

Needed for verification

invariant lefteven : left = **null** ⊕ (left ≠ **null** ⊗ left **in** even);
invariant leftodd : left ≠ **null** ⊗ left **in** odd;
invariant righteven : right = **null** ⊕ (right ≠ **null** ⊗ right **in** even);
invariant rightodd : right ≠ **null** ⊗ right **in** odd;

invariant WEIGHT:
(odd = **false** —∘ ((**this in** leftodd ⊗ **this in** righteven) ⊕
(**this in** lefteven ⊗ **this in** rightodd))) &
…

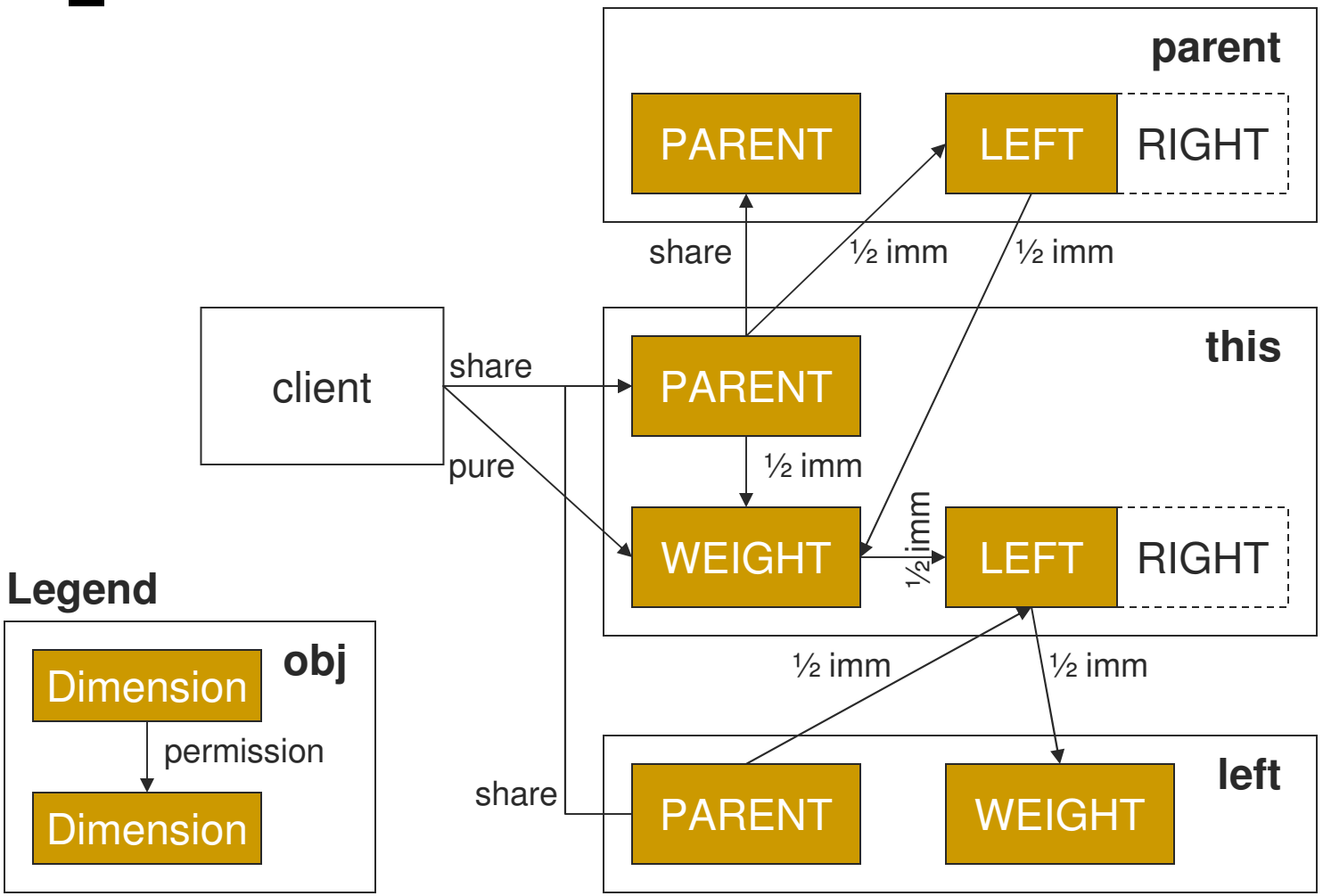
Trick: Use half fractions to allow modification

- We can only modify a field if we have a full permission for its dimension
- Solution: Use ½ fractions
  - full(x, P) ⇔ immutable(x, P, ½) ⊗ immutable(x, P, ½)

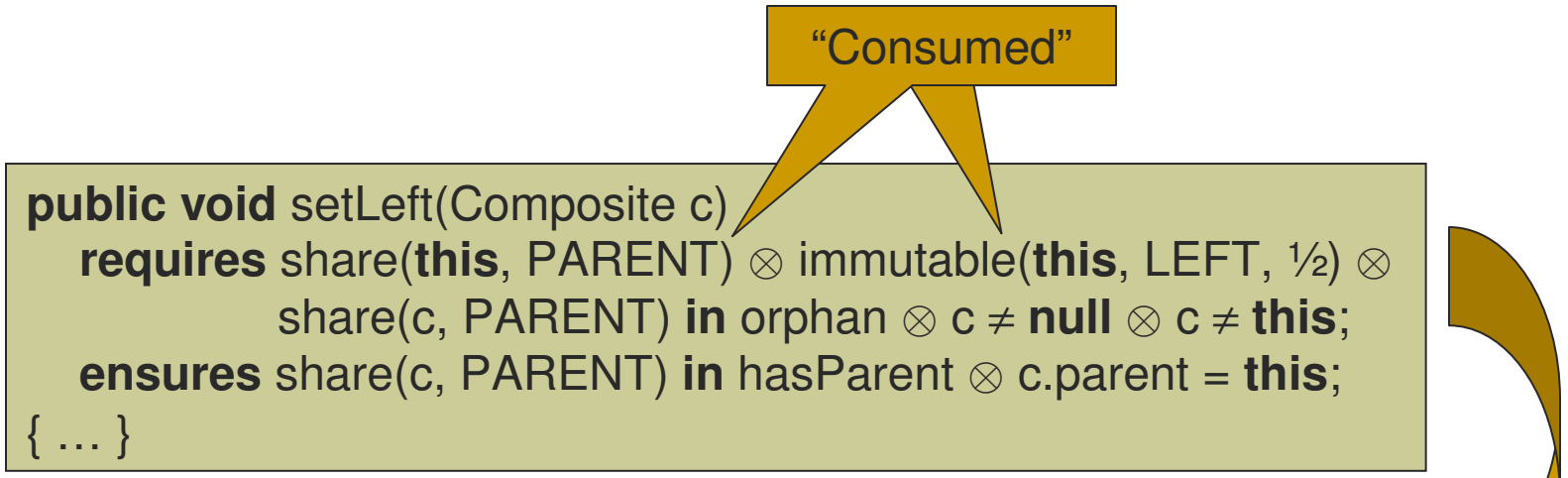
```
invariant WEIGHT : immutable(this, LEFT, ½) ⊗ immutable(this, RIGHT, ½);
invariant LEFT : left ≠ null ⊸ immutable(left, WEIGHT, ½);
invariant RIGHT : right ≠ null ⊸ immutable(right, WEIGHT, ½);

invariant PARENT : immutable(this, WEIGHT, ½) ⊗ parent ≠ this;
invariant PARENT :
    (parent = null ⊸ immutable(this, WEIGHT, ½)) &
    (parent ≠ null ⊸ (share(parent, PARENT) ⊗
                      ((immutable(parent, LEFT, ½) ⊗ parent.left = this) ⊕
                       (immutable(parent, RIGHT, ½) ⊗ parent.right = this))));
```

Permissions between a composite node and its neighbors



Specification for adding a child driven by invariants



invariant PARENT :
parent ≠ **null** ⊸ (share(parent, PARENT) ⊗
((immutable(parent, LEFT, ½) ⊗ parent.left = **this**) ⊕
(immutable(parent, RIGHT, ½) ⊗ parent.right = **this**)));
invariant orphan : parent = **null**;
invariant hasParent : parent ≠ **null**;

A taste of verifying implementation for new child

```
public void setLeft(Composite c)
   requires share(this, PARENT) ⊗ immutable(this, LEFT, ½) ⊗
            share(c, PARENT) in orphan ⊗ c ≠ null ⊗ c ≠ this;
{
   unpack(c, PARENT);
   c.parent = this;
   unpack(this, PARENT);
   if(parent != null) {
      unpack(parent, PARENT); unpack(parent, WEIGHT);
      unpack(parent, LEFT); unpack(parent, RIGHT); }
   unpack(this, WEIGHT); unpack(this, LEFT);
   left = c;
   pack(this, LEFT); unpack(c, WEIGHT);
   if(c.odd) {
      pack(c, WEIGHT); pack(c, PARENT);
      odd = ! odd;
      pack(this, WEIGHT);
      if(parent != null) { pack(parent, LEFT); pack(parent, RIGHT); }
      pack(this, PARENT); … } }
```

immutable(this, WEIGHT, ½) ⊗
(parent ≠ **null** —∘ (share(parent, PARENT)
⊗ immutable(parent, LEFT, ½)
⊗ parent.left = **this**))

mutable(parent.left, WEIGHT, ½)
erged: full(**this**, WEIGHT)

immutable(this, LEFT, ½)
merged: full(**this**, LEFT)

Future work: Non-typestate invariants

- Permissions form frame of dependencies
- We think that any property could be tracked on top of them
  - **invariant** WEIGHT : weight = lw() + rw() + 1;
  - **int** lw() { **return** left == **null** ? 0 : left.weight; }
  - **int** rw() { **return** right == **null** ? 0 : right.weight; }
- May require theorem prover to reason about desired property
  - Or extend Boogie or ESC/Java to permissions

Permissions to Specify the Composite Design Pattern

- Permissions allow nodes to depend on their children *and* adding new nodes
  - Immutable permissions allow nodes to depend on other nodes
  - ½ fractions allow modifications when needed
- Paper discusses how current limitations could be overcome
  - More complex invariants
  - Support for list of children
  - Specification overhead reduction