# Adding state and visibility control to traits using lexical nesting

Tom Van Cutsem (VUB, Belgium)
Alexandre Bergel (INRIA Lille, France & PLEIAD, Chile)
Stéphane Ducasse (INRIA Lille, France)
Wolfgang De Meuter (VUB, Belgium)

# Introduction



- Traits: composable units of behavior *(Schärli et al., 2003)*
- Original model: no state, no visibility control (private/public methods)
    - Stateful Traits *(Bergel et al., 2007)*: operators to expose/merge state
    - Freezable Traits *(Ducasse et al., 2007)*: operators to change visibility of names

# Introduction



- Traits: composable units of behavior *(Schärli et al., 2003)*
- Original model: no state, no visibility control (private/public methods)
    - Stateful Traits *(Bergel et al., 2007)*: operators to expose/merge state
    - Freezable Traits *(Ducasse et al., 2007)*: operators to change visibility of names
- This work: state + visibility control without additional operators...
    - but: different trait model
    - state and visibility control through lexical nesting
    - implementation in AmbientTalk

# Traits (Schärli et al., ECOOP 03)



- Alternative to Multiple Inheritance
- Composition order of multiple traits does not matter
- Explicit resolution of conflicts:
  - Local redefinition (with alias to original method)
  - Exclusion

# Traits (Schärli et al., ECOOP 03)



- Alternative to Multiple Inheritance
- Composition order of multiple traits does not matter
- Explicit resolution of conflicts:
  - Local redefinition (with alias to original method)
  - Exclusion

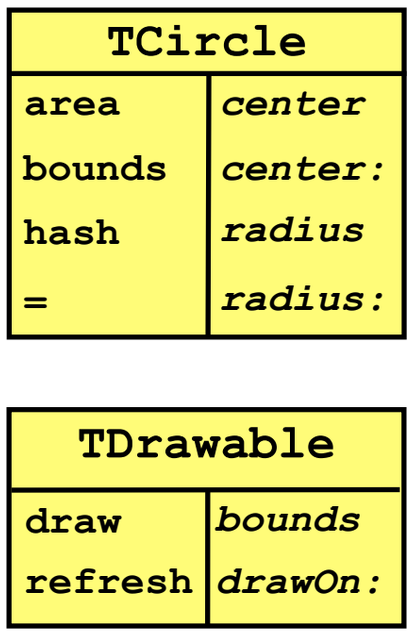
| TCircle | |
|---|---|
| area | *center* |
| bounds | *center:* |
| hash | *radius* |
| = | *radius:* |

| TDrawable | |
|---|---|
| draw | *bounds* |
| refresh | *drawOn:* |

# Traits (Schärli et al., ECOOP 03)



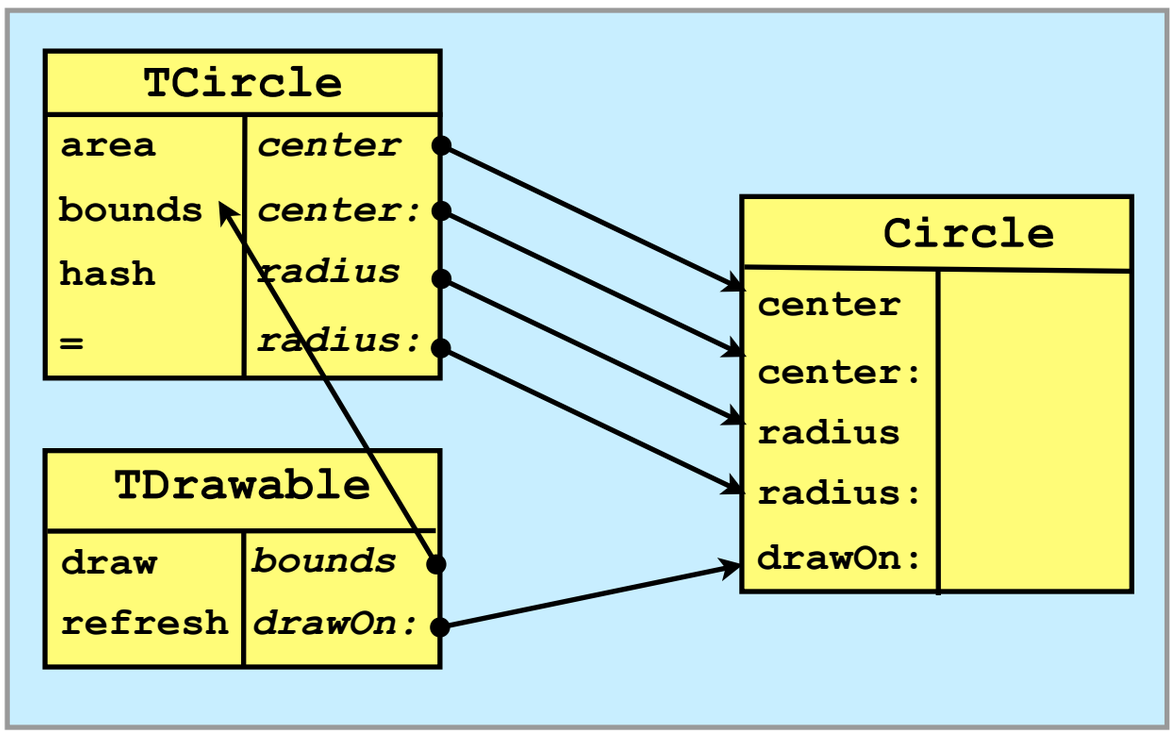
- Alternative to Multiple Inheritance
- Composition order of multiple traits does not matter
- Explicit resolution of conflicts:
  - Local redefinition (with alias to original method)
  - Exclusion



# Traits (Schärli et al., ECOOP 03)



- Alternative to Multiple Inheritance
- Composition order of multiple traits does not matter
- Explicit resolution of conflicts:
  - Local redefinition (with alias to original method)
  - Exclusion



# AmbientTalk



- Concurrent & Distributed Programming Language
- Object-based (no classes)
- Anonymous lexically nested objects

# AmbientTalk



- Concurrent & Distributed Programming Language
- Object-based (no classes)
- Anonymous lexically nested objects

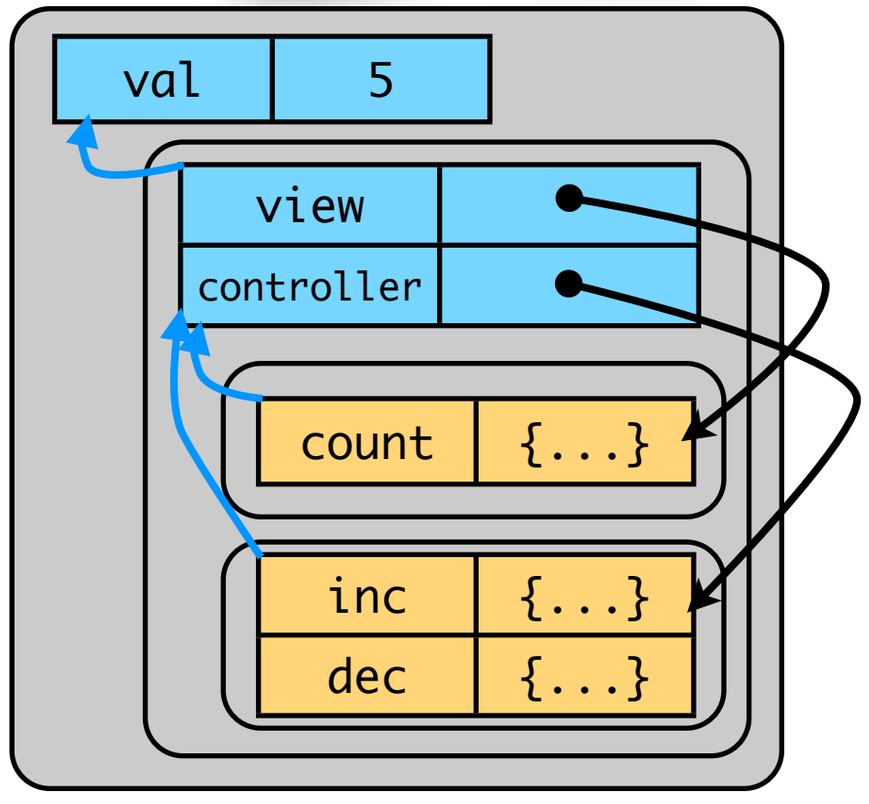

```
def makeCounter(val) {
  def view := object: {
    def count() { val }
  }
  def controller := object: {
    def inc() { val := val + 1 }
    def dec() { val := val - 1 }
  }
  [view, controller]
}
```

# AmbientTalk



- Concurrent & Distributed Programming Language
- Object-based (no classes)
- Anonymous lexically nested objects

```
def makeCounter(val) {
  def view := object: {
    def count() { val }
  }
  def controller := object: {
    def inc() { val := val + 1 }
    def dec() { val := val - 1 }
  }
  [view, controller]
}
```



# Traits in AmbientTalk



- Traits are ordinary objects (cf. Ungar & Smith's Self)
- Required methods are implicit (self sends in method implementations)

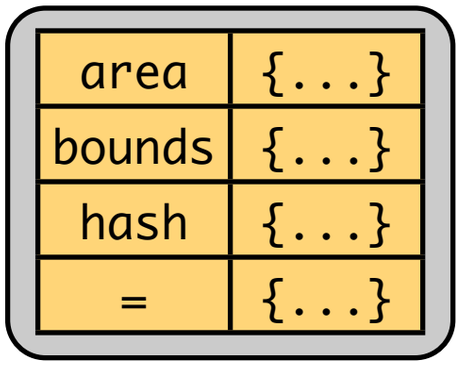
```
def TCircle := object: {
  def area() { PI * self.radius().square() }
  def bounds() { ... }
  def hash() { ... }
  def =(o) { ... }
}
```

# Traits in AmbientTalk



- Traits are ordinary objects (cf. Ungar & Smith's Self)
- Required methods are implicit (self sends in method implementations)

```
def TCircle := object: {
  def area() { PI * self.radius().square() }
  def bounds() { ... }
  def hash() { ... }
  def =(o) { ... }
}
```

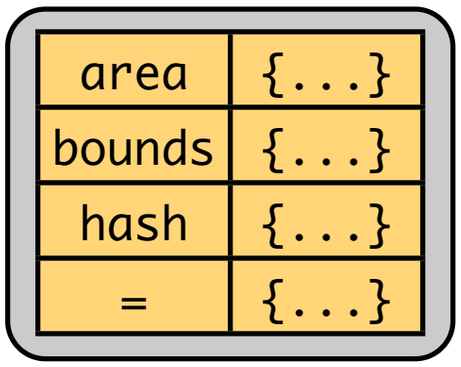


# Traits in AmbientTalk



- Traits are ordinary objects (cf. Ungar & Smith's Self)
- Required methods are implicit (self sends in method implementations)

```
def TCircle := object: {
  def area() { PI * self.radius().square() }
  def bounds() { ... }
  def hash() { ... }
  def =(o) { ... }
}

def makeCircle(c, r) {
  object: {
    import TCircle;
    import TDrawable;
    def radius() { r }
    ...
  }
}
```

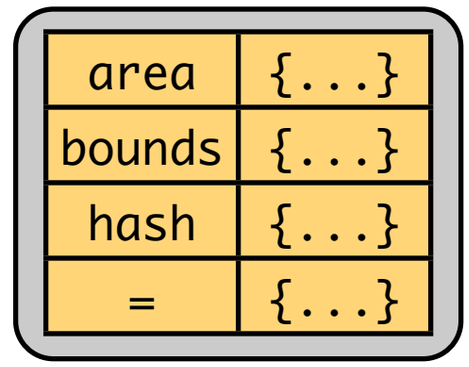


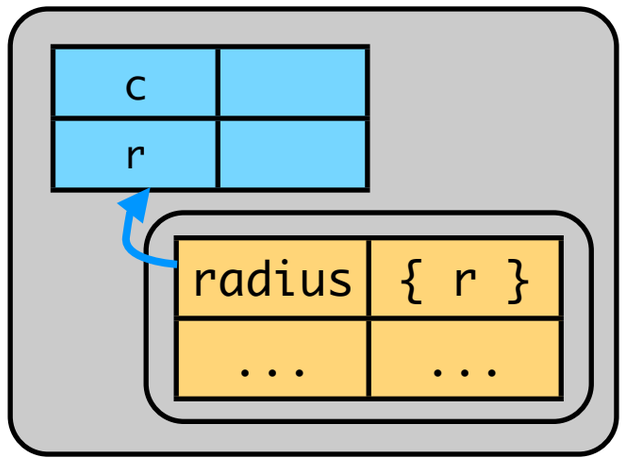
# Traits in AmbientTalk



- Traits are ordinary objects (cf. Ungar & Smith's Self)
- Required methods are implicit (self sends in method implementations)

```
def TCircle := object: {
  def area() { PI * self.radius().square() }
  def bounds() { ... }
  def hash() { ... }
  def =(o) { ... }
}

def makeCircle(c, r) {
  object: {
    import TCircle;
    import TDrawable;
    def radius() { r }
    ...
  }
}
```



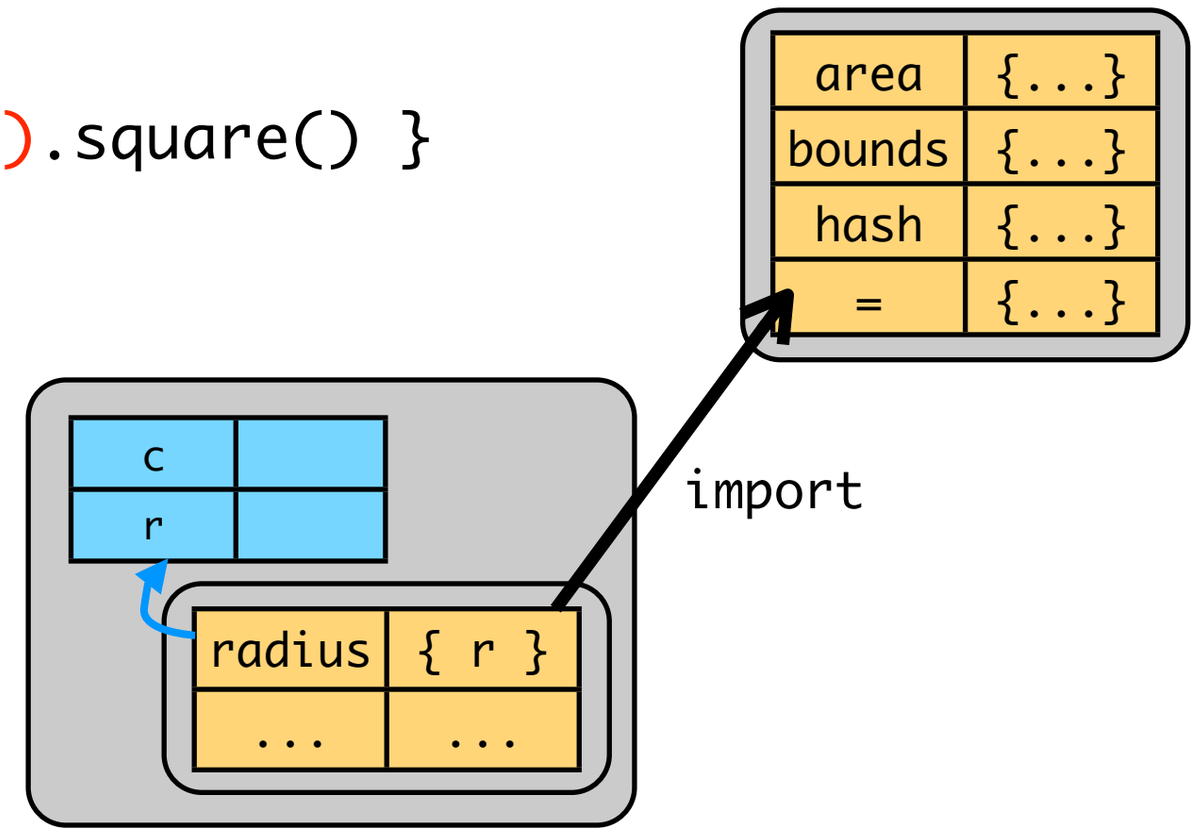



# Traits in AmbientTalk



- Traits are ordinary objects (cf. Ungar & Smith's Self)
- Required methods are implicit (self sends in method implementations)

```
def TCircle := object: {
  def area() { PI * self.radius().square() }
  def bounds() { ... }
  def hash() { ... }
  def =(o) { ... }
}

def makeCircle(c, r) {
  object: {
    import TCircle;
    import TDrawable;
    def radius() { r }
    ...
  }
}
```

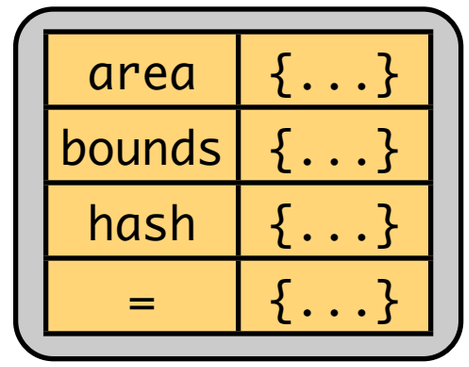


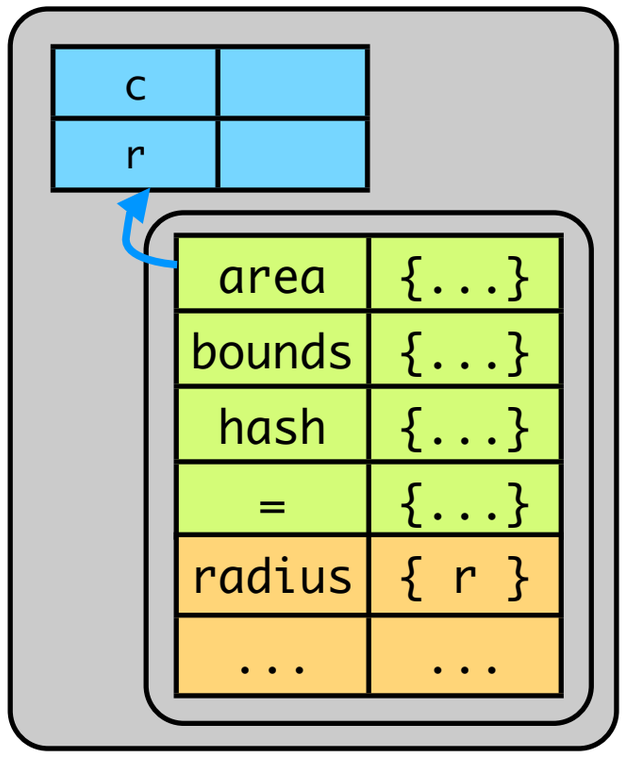
# Traits in AmbientTalk



- Traits are ordinary objects (cf. Ungar & Smith's Self)
- Required methods are implicit (self sends in method implementations)

```
def TCircle := object: {
  def area() { PI * self.radius().square() }
  def bounds() { ... }
  def hash() { ... }
  def =(o) { ... }
}

def makeCircle(c, r) {
  object: {
    import TCircle;
    import TDrawable;
    def radius() { r }
    ...
  }
}
```



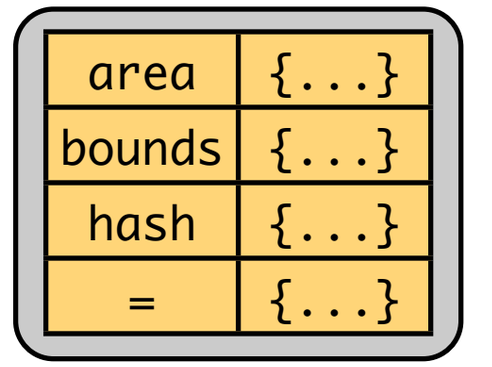



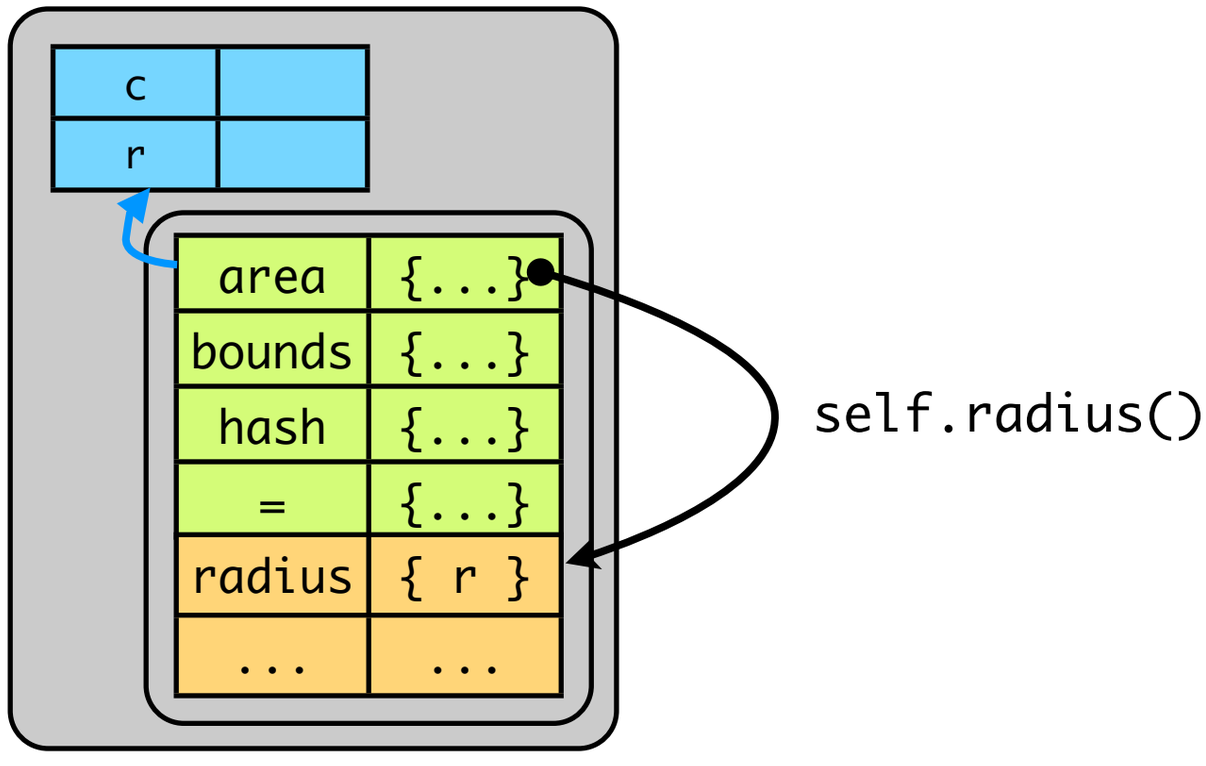
# Traits in AmbientTalk



- Traits are ordinary objects (cf. Ungar & Smith's Self)
- Required methods are implicit (self sends in method implementations)

```
def TCircle := object: {
  def area() { PI * self.radius().square() }
  def bounds() { ... }
  def hash() { ... }
  def =(o) { ... }
}

def makeCircle(c, r) {
  object: {
    import TCircle;
    import TDrawable;
    def radius() { r }
    ...
  }
}
```



# Traits in AmbientTalk: Conflict Resolution



- Methods may be aliased or excluded

```
def TCircle := object: {
  def area() { PI * self.radius().square() }
  def bounds() { ... }
  def hash() { ... }
  def =(o) { ... }
}


def TColor := object: {
  def red() { self.color.red }
  ...
  def hash() { ... }
  def =(o) { ... }
}
```

# Traits in AmbientTalk: Conflict Resolution



- Methods may be aliased or excluded

```
def TCircle := object: {
  def area() { PI * self.radius().square() }
  def bounds() { ... }
  def hash() { ... }
  def =(o) { ... }
}
```

```
def TColor := object: {
  def red() { self.color.red }
  ...
  def hash() { ... }
  def =(o) { ... }
}
```

# Traits in AmbientTalk: Conflict Resolution



- Methods may be aliased or excluded

```
def TCircle := object: {
  def area() { PI * self.radius().square() }
  def bounds() { ... }
  def hash() { ... }
  def =(o) { ... }
}

def TColor := object: {
  def red() { self.color.red }
  ...
  def hash() { ... }
  def =(o) { ... }
}
```

```
def makeCircle(c, r, col) {
  object: {
    import TCircle exclude hash;
    import TColor alias = → color=;
    def radius() { r }
    def color() { col }
    ...
  }
}
```

# Traits in AmbientTalk: Conflict Resolution



- Methods may be aliased or excluded

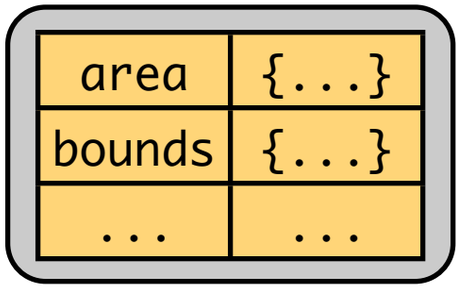
```
def TCircle := object: {
  def area() { PI * self.radius().square() }
  def bounds() { ... }
  def hash() { ... }
  def =(o) { ... }
}

def TColor := object: {
  def red() { self.color.red }
  ...
  def hash() { ... }
  def =(o) { ... }
}

def makeCircle(c, r, col) {
  object: {
    import TCircle exclude hash;
    import TColor alias = → color=;
    def radius() { r }
    def color() { col }
    ...
  }
}
```

# Traits and lexical nesting



- Traits can hide state and methods in their lexical scope

```
def TCircle := object: {
  def area() { PI * self.radius().square() }
  ...
}
```

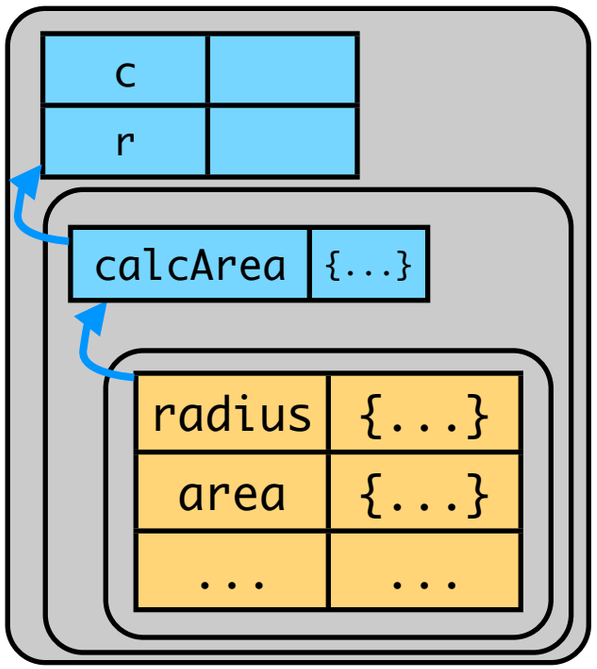


# Traits and lexical nesting



- Traits can hide state and methods in their lexical scope

```
def makeCircleTrait(c, r) {
  def calcArea() { PI * r * r }
  object: {
    def radius() { r }
    def area() { calcArea() }
    ...
  }
}
```

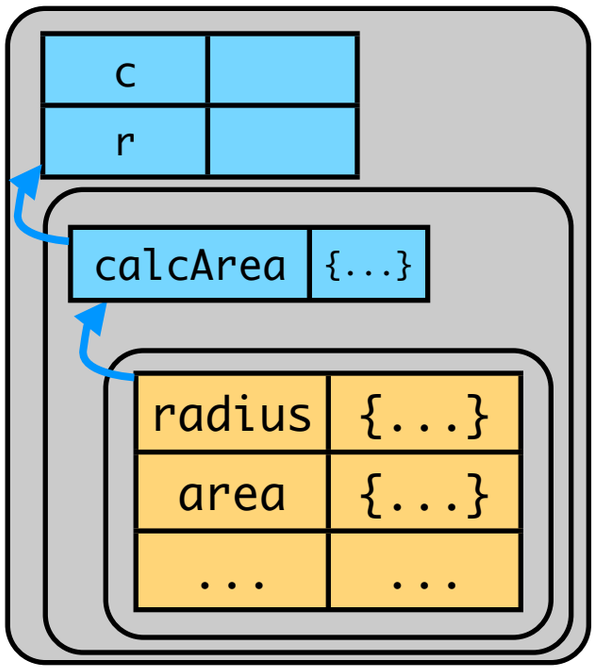


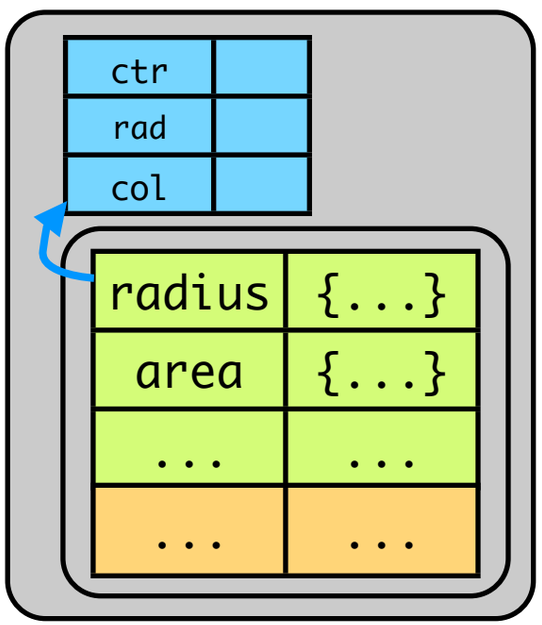
# Traits and lexical nesting



- Traits can hide state and methods in their lexical scope

```
def makeCircleTrait(c, r) {
  def calcArea() { PI * r * r }
  object: {
    def radius() { r }
    def area() { calcArea() }
    ...
  }
}

def makeCircle(ctr, rad, col) {
  object: {
    import makeCircleTrait(ctr, rad);
    import TDrawable;
    ...
  }
}
```



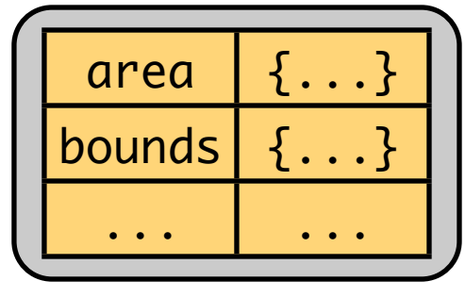



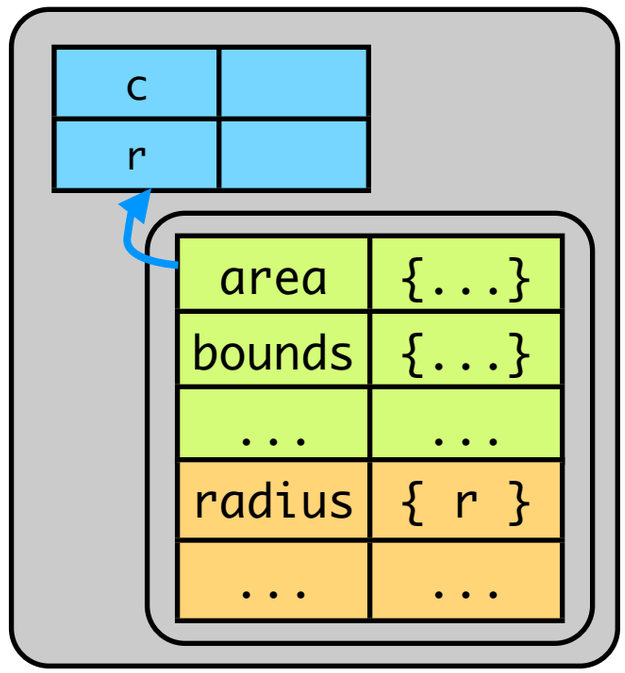
# The Flattening Property



- “the semantics of a class defined using traits is exactly the same as that of a class constructed directly from all of the non-overridden methods of the traits” *(Schärli et al., 2003)*

```
def TCircle := object: {
  def area() { PI * self.radius().square() }
  ...
}

def makeCircle(c, r) {
  object: {
    import TCircle;
    def radius() { r }
    ...
  }
}
```



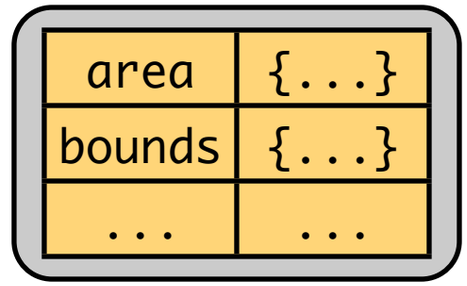



# The Flattening Property

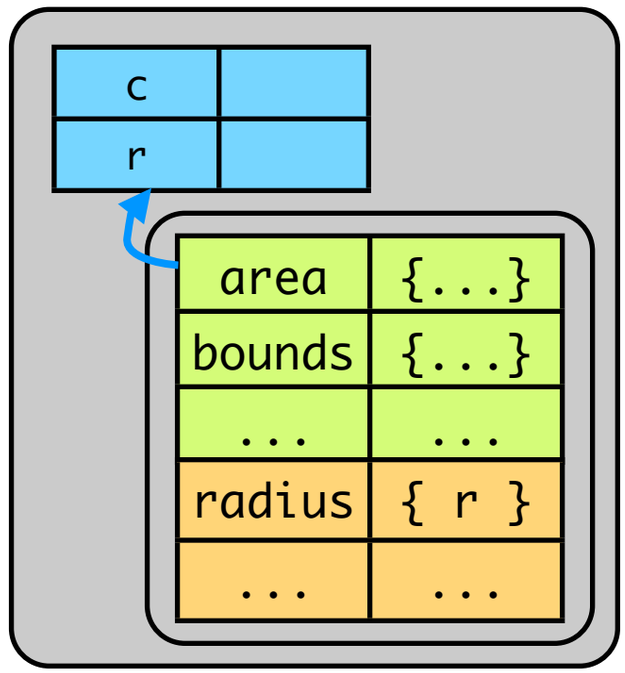


- “the semantics of a class defined using traits is exactly the same as that of a class constructed directly from all of the non-overridden methods of the traits” *(Schärli et al., 2003)*

```
def TCircle := object: {
  def area() { PI * self.radius().square() }
  ...
}
```




```
def makeCircle(c, r) {
  object: {
    def area() { PI * self.radius().square() }
    def radius() { r }
    ...
  }
}
```

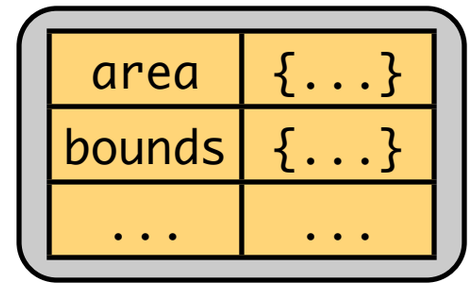


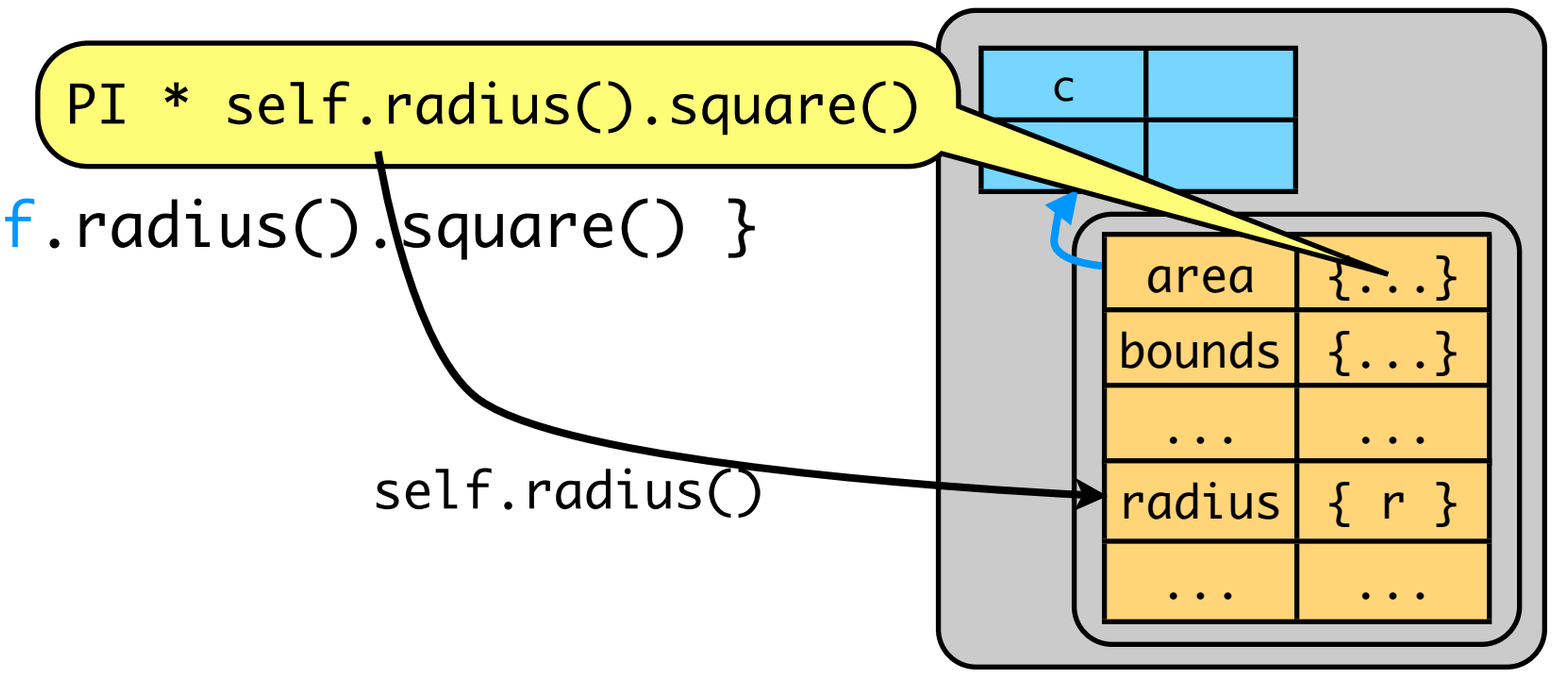
# The Flattening Property



- “the semantics of a class defined using traits is exactly the same as that of a class constructed directly from all of the non-overridden methods of the traits” *(Schärli et al., 2003)*

```
def TCircle := object: {
  def area() { PI * self.radius().square() }
  ...
}

def makeCircle(c, r) {
  object: {
    def area() { PI * self.radius().square() }
    def radius() { r }
    ...
  }
}
```



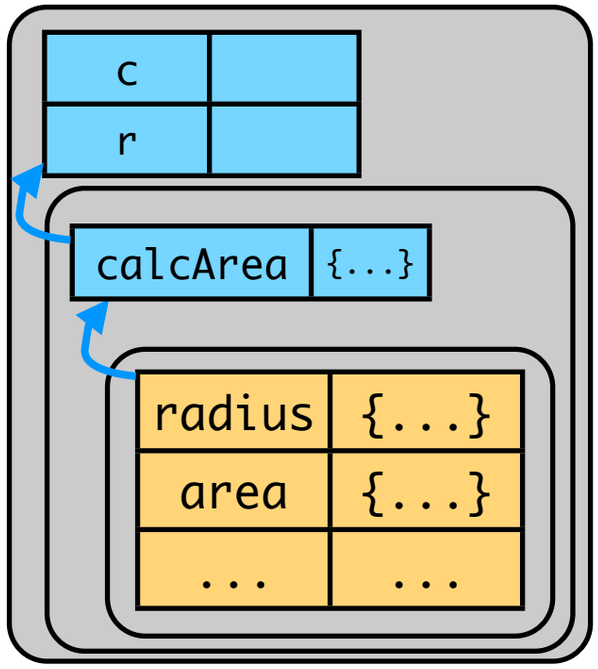



# The Flattening Property



- No longer holds in the case of lexical nesting

```
def makeCircleTrait(c, r) {
  def calcArea() { PI * r * r }
  object: {
    def radius() { r }
    def area() { calcArea() }
    ...
  }
}
```

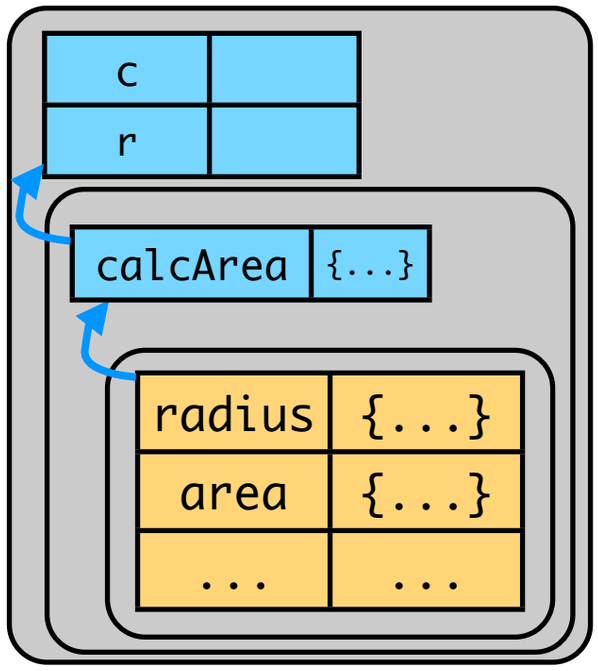


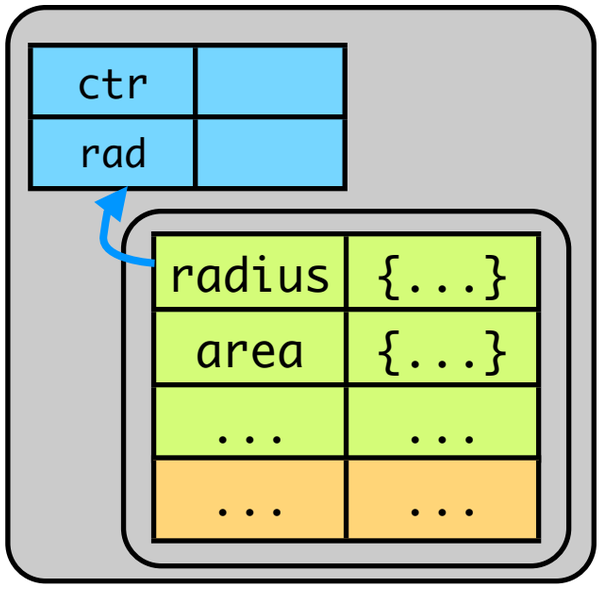
# The Flattening Property



- No longer holds in the case of lexical nesting

```
def makeCircleTrait(c, r) {
  def calcArea() { PI * r * r }
  object: {
    def radius() { r }
    def area() { calcArea() }
    ...
  }
}

def makeCircle(ctr, rad, col) {
  object: {
    import makeCircleTrait(ctr,rad);
    ...
  }
}
```



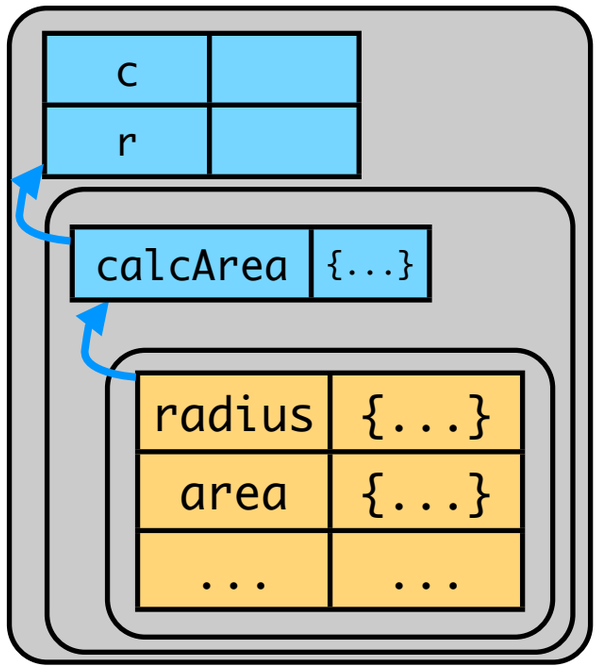



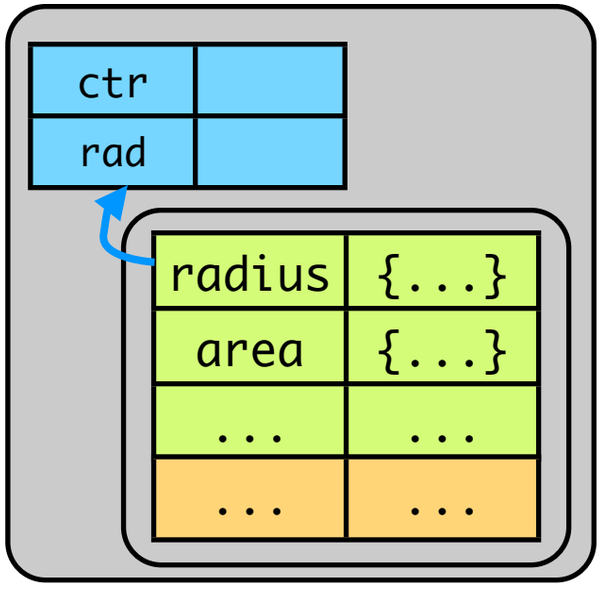
# The Flattening Property



- No longer holds in the case of lexical nesting

```
def makeCircleTrait(c, r) {
  def calcArea() { PI * r * r }
  object: {
    def radius() { r }
    def area() { calcArea() }
    ...
  }
}

def makeCircle(ctr, rad) {
  object: {
    def radius() { r }
    def area() { calcArea() }
    ...
  }
}
```



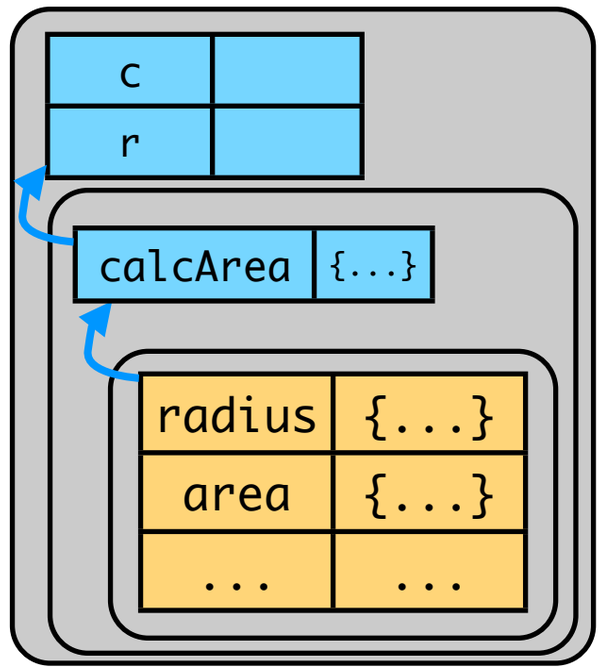



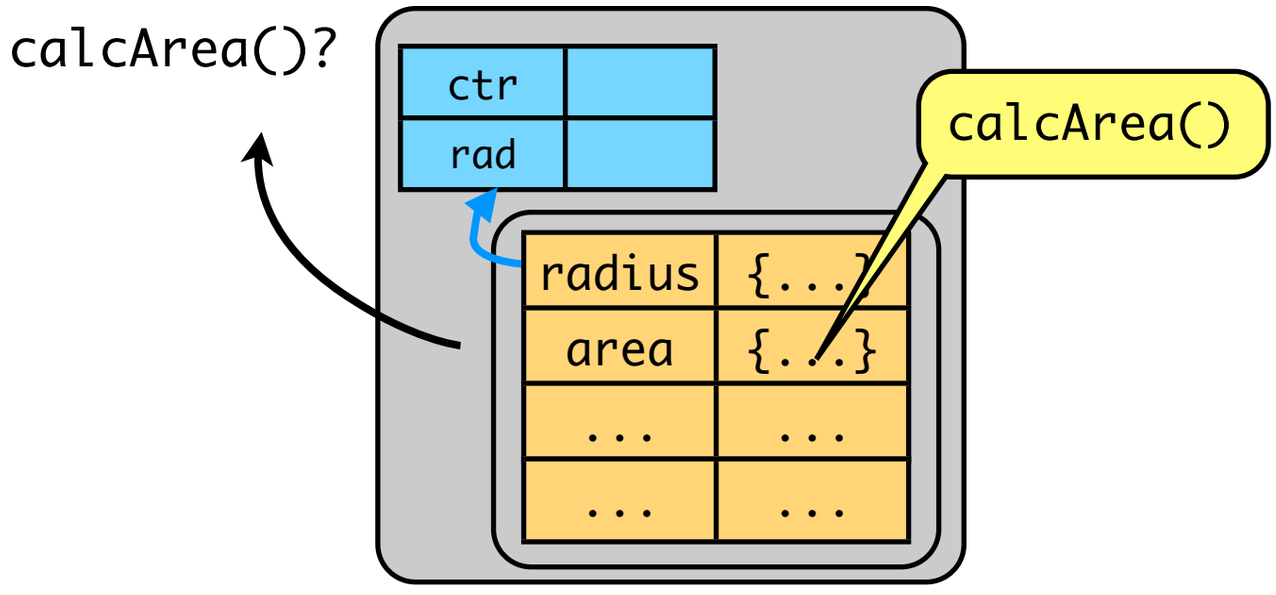
# The Flattening Property



- No longer holds in the case of lexical nesting

```
def makeCircleTrait(c, r) {
  def calcArea() { PI * r * r }
  object: {
    def radius() { r }
    def area() { calcArea() }
    ...
  }
}

def makeCircle(ctr, rad) {
  object: {
    def radius() { r }
    def area() { calcArea() }
    ...
  }
}
```



# Trait composition via delegation



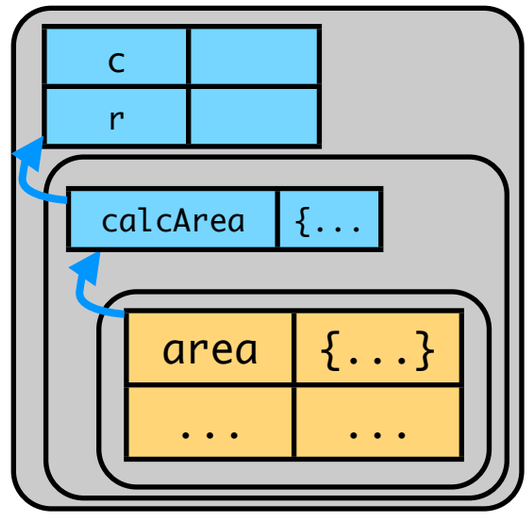
- In a trait method, self must be bound to the composite, yet the method's lexical scope should not change => use delegation *(Lieberman, 1986)*

# Trait composition via delegation



- In a trait method, self must be bound to the composite, yet the method's lexical scope should not change => use delegation *(Lieberman, 1986)*

```
def makeCircleTrait(c, r) {
  def calcArea() { PI * r * r }
  object: {
    def area() { calcArea() }
    ...
  }
}
```

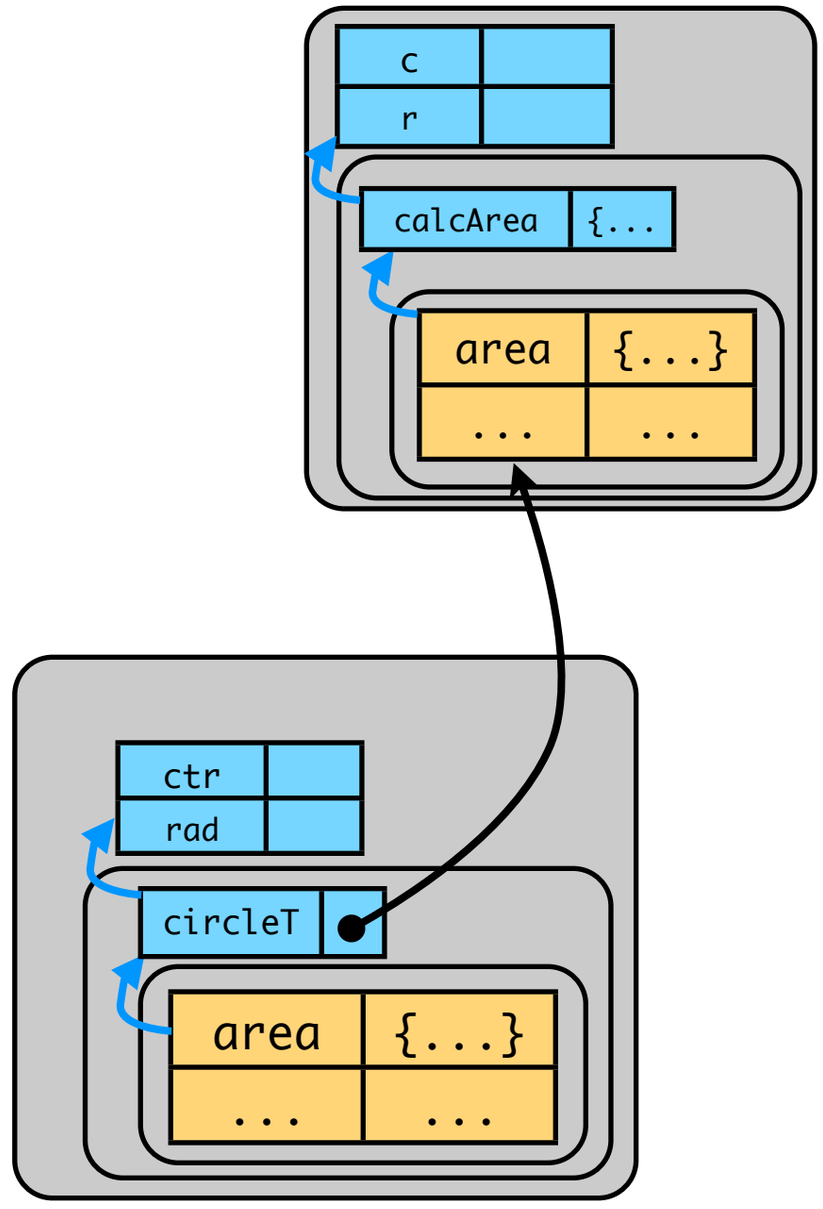


# Trait composition via delegation



- In a trait method, self must be bound to the composite, yet the method's lexical scope should not change => use delegation *(Lieberman, 1986)*

```
def makeCircleTrait(c, r) {
  def calcArea() { PI * r * r }
  object: {
    def area() { calcArea() }
    ...
  }
}

def makeCircle(ctr,rad) {
  def circleT := makeCircleTrait(ctr,rad);
  object: {
    def area() { circleT^area() }
    ...
  }
}
```

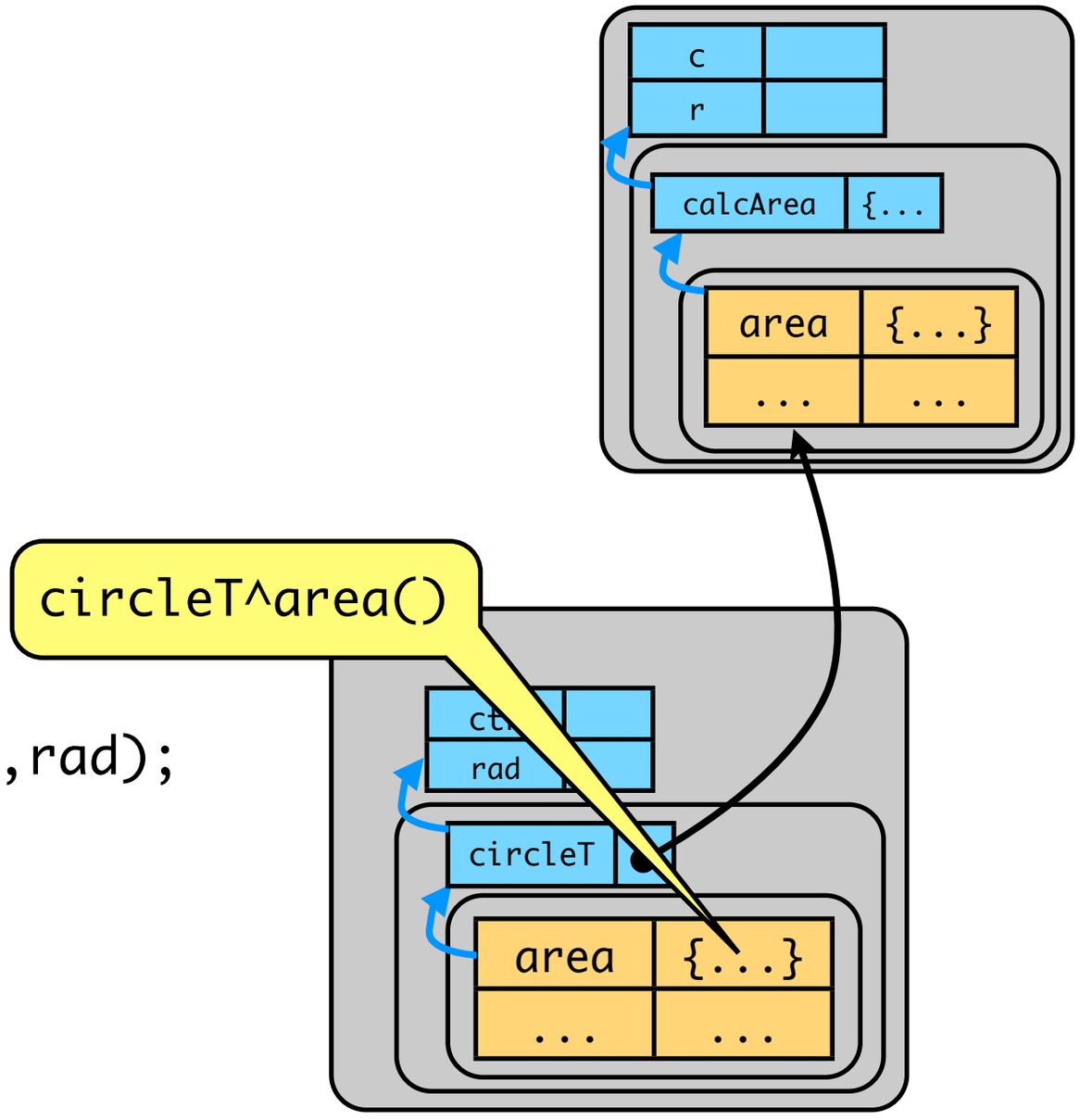


# Trait composition via delegation



- In a trait method, self must be bound to the composite, yet the method's lexical scope should not change => use delegation *(Lieberman, 1986)*

```
def makeCircleTrait(c, r) {
  def calcArea() { PI * r * r }
  object: {
    def area() { calcArea() }
    ...
  }
}

def makeCircle(ctr,rad) {
  def circleT := makeCircleTrait(ctr,rad);
  object: {
    def area() { circleT^area() }
    ...
  }
}
```

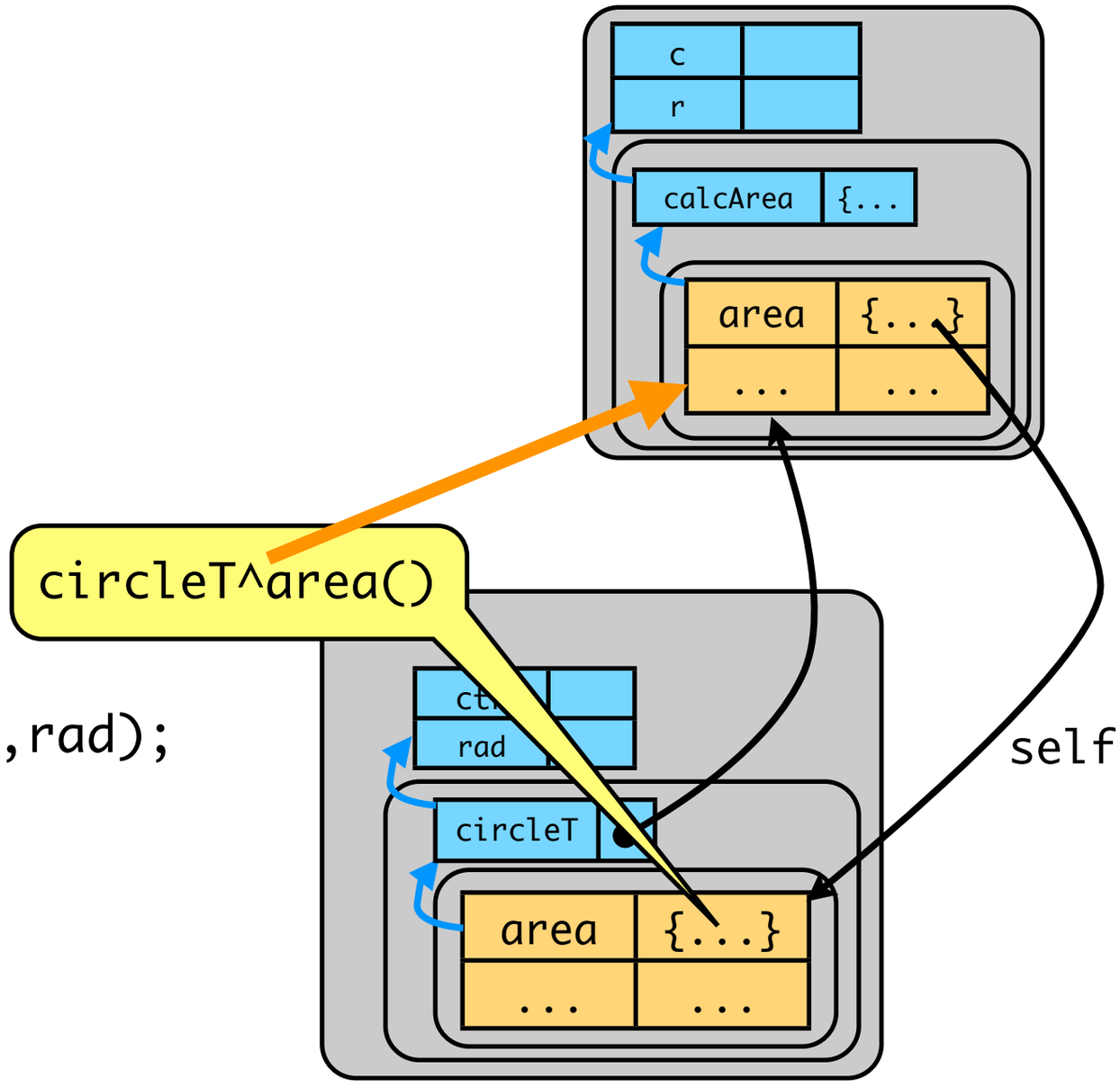


# Trait composition via delegation



- In a trait method, self must be bound to the composite, yet the method's lexical scope should not change => use delegation *(Lieberman, 1986)*

```
def makeCircleTrait(c, r) {
  def calcArea() { PI * r * r }
  object: {
    def area() { calcArea() }
    ...
  }
}

def makeCircle(ctr,rad) {
  def circleT := makeCircleTrait(ctr,rad);
  object: {
    def area() { circleT^area() }
    ...
  }
}
```

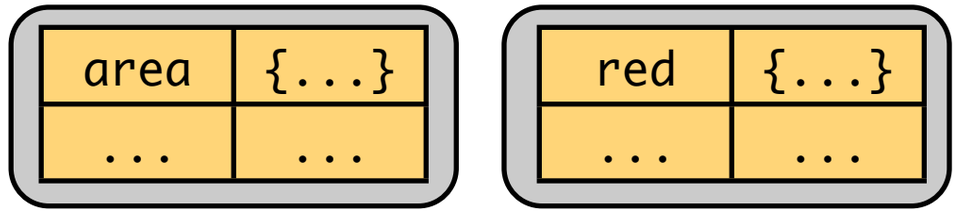


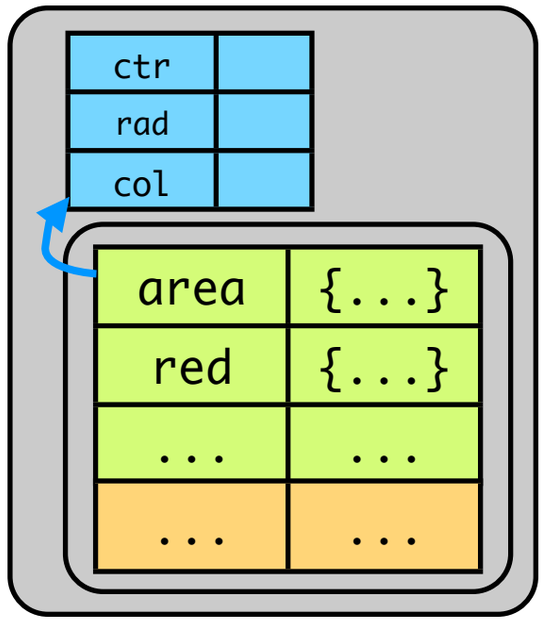
# Trait composition as delegate method generation



- Import clause expanded into delegating methods

```
def makeCircle(ctr,rad,col) {
  object: {
    import makeCircleTrait(ctr,rad);
    import makeColorTrait(col);
    ...
  }
}
```



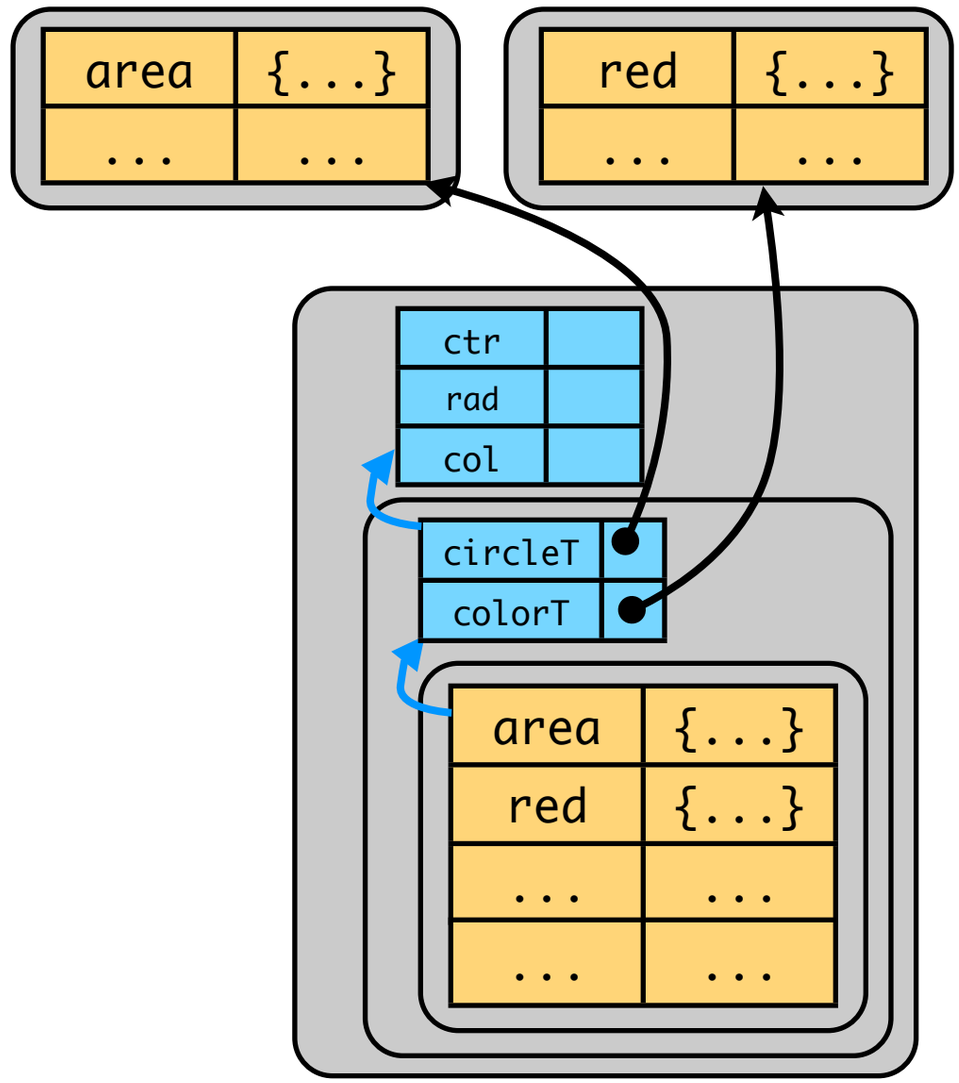



# Trait composition as delegate method generation



- Import clause expanded into delegating methods

```
def makeCircle(ctr,rad,col) {
  def circleT := makeCircleTrait(ctr,rad);
  def colorT := makeColorTrait(col);
  object: {
    def area() { circleT^area() }
    def red() { colorT^red() }
    ...
  }
}
```

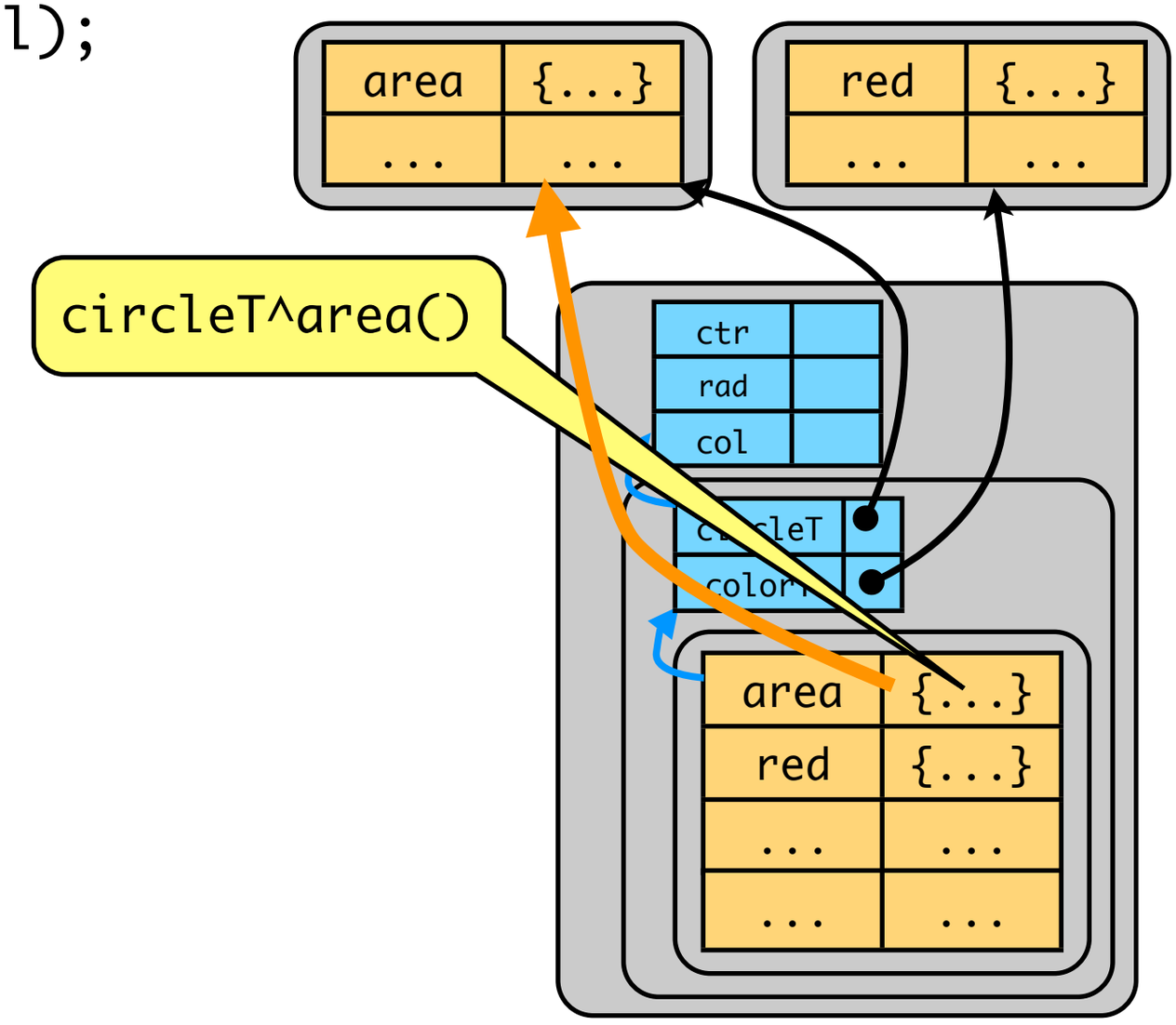


# Trait composition as delegate method generation



- Import clause expanded into delegating methods

```
def makeCircle(ctr,rad,col) {
  def circleT := makeCircleTrait(ctr,rad);
  def colorT := makeColorTrait(col);
  object: {
    def area() { circleT^area() }
    def red() { colorT^red() }
    ...
  }
}
```

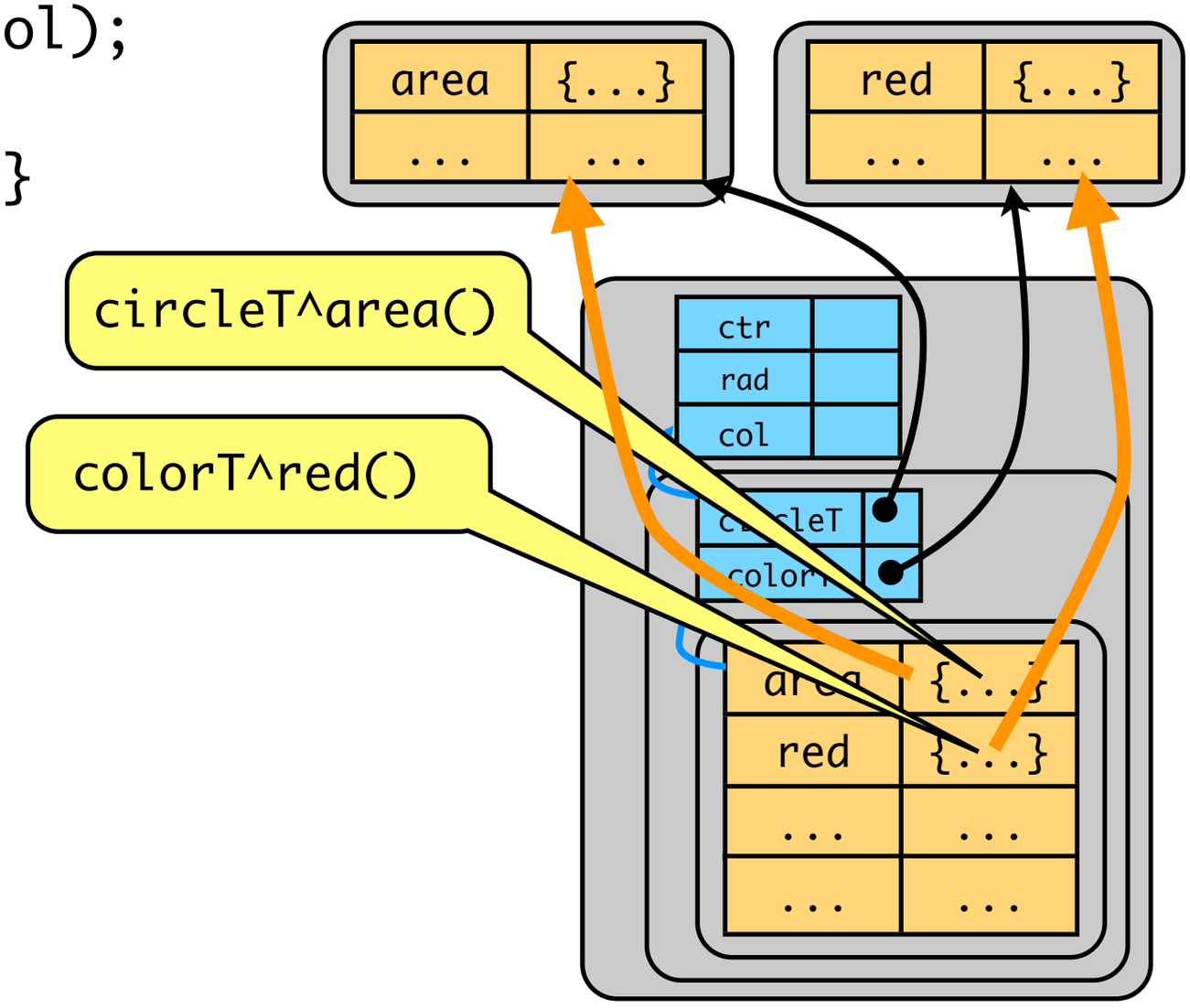


# Trait composition as delegate method generation



- Import clause expanded into delegating methods

```
def makeCircle(ctr,rad,col) {
  def circleT := makeCircleTrait(ctr,rad);
  def colorT := makeColorTrait(col);
  object: {
    def area() { circleT^area() }
    def red() { colorT^red() }
    ...
  }
}
```



# Trait composition as delegate method generation



- Exclusion = don’t generate a delegate method
- Aliasing = generate a delegate method with an aliased name

```
def makeCircle(c, r, col) {
  object: {
    import TCircle exclude hash;
    import TColor alias = → color=;
    ...
  }
}
```

# Trait composition as delegate method generation



- Exclusion = don’t generate a delegate method
- Aliasing = generate a delegate method with an aliased name

```
def makeCircle(c, r, col) {
  object: {
    import TCircle exclude hash;
    import TColor alias = → color=;
    ...
  }
}
```

```
def makeCircle(c, r, col) {
  def circleT := TCircle;
  def colorT := TColor;
  object: {
    def area() { circleT^area() }
    def red() { colorT^red() }
    ...
    def hash() { colorT^hash() }
    def =(o) { circleT^=(o) }
    def color=(o) { colorT^=(o) }
    ...
  }
}
```

# Trait composition as delegate method generation



- Exclusion = don’t generate a delegate method
- Aliasing = generate a delegate method with an aliased name

```
def makeCircle(c, r, col) {
  object: {
    import TCircle exclude hash;
    import TColor alias = → color=;
    ...
  }
}
```

```
def makeCircle(c, r, col) {
  def circleT := TCircle;
  def colorT := TColor;
  object: {
    def area() { circleT^area() }
    def red() { colorT^red() }
    ...
    def hash() { colorT^hash() }
    def =(o) { circleT^=(o) }
    def color=(o) { colorT^=(o) }
    ...
  }
}
```

# Trait composition as delegate method generation



- Exclusion = don't generate a delegate method
- Aliasing = generate a delegate method with an aliased name

```
def makeCircle(c, r, col) {
  object: {
    import TCircle exclude hash;
    import TColor alias = → color=;
    ...
  }
}
```

```
def makeCircle(c, r, col) {
  def circleT := TCircle;
  def colorT := TColor;
  object: {
    def area() { circleT^area() }
    def red() { colorT^red() }
    ...
    def hash() { colorT^hash() }
    def =(o) { circleT^=(o) }
    def color=(o) { colorT^=(o) }
    ...
  }
}
```

# Limitations



- Diamond “imports” with stateful traits: no merging of state
- Unlike freezable traits, no way to make private state/methods public again
- No additional composition operators, but reliance on lexical nesting instead
  - Limits number of languages to which our model can be applied
  - But not limited to prototype-based languages (classes can be nested too!)

# Conclusion



- Added state and visibility control to traits
- Different trait model: traits as *lexically nestable* objects
- In a language with nesting, traits cannot simply be flattened:
  - used explicit delegation of messages instead
  - explicit composition/conflict detection maintained by generating delegate methods (vs. Self's use of implicit delegation)
- Operational semantics: cf. paper

code.google.com/p/ambienttalk