Selective Ownership: Combining Object and Type Hierarchies for Flexible Sharing

Stephanie Balzer, Thomas R. Gross, and Peter Müller

School of Computer Science, Carnegie Mellon University
Department of Computer Science, ETH Zurich

FOOL 2012
Ownership type systems
Ownership type systems

Structure program heap

- Facilitate reasoning about program
Ownership type systems

Structure program heap

- Facilitate reasoning about program

Application domains

- Thread synchronization
- Memory management
- Enforcement of architectural styles
- Program verification
Ownership type systems

Structure program heap
- Facilitate reasoning about program

Application domains
- Thread synchronization
- Memory management
- Enforcement of architectural styles
- Program verification

Proof obligations to verify, at compile-time, that invariants hold
Ownership-based verification
Ownership-based verification
Ownership-based verification

invariant on tree nodes
Ownership-based verification

invariant on tree nodes

Owner of children nodes
Ownership-based verification

invariant on tree nodes

Owner of children nodes
Ownership-based verification

invariant on tree nodes

Owner of children nodes

Owner of children nodes
Ownership-based verification

invariant on tree nodes

Owner of children nodes

n1

n2

n4

n5

n3

n6

n7
Ownership-based verification

invariant on tree nodes

Impose a tree topology on program heap
Ownership-based verification
Ownership-based verification

Modifications of objects are initiated by their owners
Ownership-based verification

Modifications of objects are initiated by their owners
Ownership-based verification

Modifications of objects are initiated by their owners
Ownership-based verification

Modifications of objects are initiated by their owners
Ownership-based verification
Ownership-based verification

No modifying call-backs into owners from owned objects
Ownership-based verification

No modifying call-backs into owners from owned objects
Ownership-based verification: summary

Guarantees by ownership

- Impose tree topology on program heap
- Owner-as-modifier discipline
- No modifying call-backs into owners from owned objects
Ownership-based verification: summary

Guarantees by ownership

- Impose tree topology on program heap
- Owner-as-modifier discipline
- No modifying call-backs into owners from owned objects

Sound, modular verification of invariants on owned objects
Ownership-based verification: summary

Guarantees by ownership

• Impose tree topology on program heap
• Owner-as-modifier discipline
• No modifying call-backs into owners from owned objects

Sound, modular verification of invariants on owned objects

However ...
Ownership-based verification: summary

Restrict sharing: modifying access only by owner and peers
Ownership-based verification: summary

Restrict sharing: modifying access only by owner and peers
Ownership-based verification: summary

Restrict sharing: modifying access only by owner and peers
About this paper
About this paper

Selective ownership

- Less rigid form of ownership
- Permits shared, modifying access to objects further “down” in heap topology
About this paper

Selective ownership

- Less rigid form of ownership
- Permits shared, modifying access to objects further “down” in heap topology
About this paper

Selective ownership

• Less rigid form of ownership
• Permits shared, modifying access to objects further “down” in heap topology
About this paper

Selective ownership

- Less rigid form of ownership
- Permits shared, modifying access to objects further “down” in heap topology
About this paper

Selective ownership

• Less rigid form of ownership
• Permits shared, modifying access to objects further “down” in heap topology

Selective ownership-based verification

• Enables sound, modular verification of invariants over shared, modifiable sub-structures

First-class relationships

• Naturally support selective ownership
Selective ownership in a nutshell
Selective ownership in a nutshell

Gives structure to program heap
Selective ownership in a nutshell

Gives structure to program heap

- By defining order on type declarations
- And, optionally, by imposing ownership on selected objects
Selective ownership in a nutshell

Gives structure to program heap

- By defining order on type declarations
- And, optionally, by imposing ownership on selected objects

E.g., subtyping, package structure
Selective ownership in a nutshell

Gives structure to program heap

• By defining order on type declarations
• And, optionally, by imposing ownership on selected objects
Selective ownership in a nutshell

Selective ownership := type order [ + object ownership]
Type order
Type order

A

B

C

D

E
Type order

Legend:  type

A

B

C

D

E
Type order

Legend: 

(type) type 

(type order) type order

A

B

C

D

E
Type order

Legend: \( \text{\textfilled{gray}} \) type \( \rightarrow \) type order

A → B

D → C → E
Type order

Legend: type order
Type order

Legend:  type  type order
Type order

Legend:  type    type order
Type order

Legend:  

$$\{A \leftrightarrow D, B \leftrightarrow C, B \leftrightarrow E, C \leftrightarrow D, C \leftrightarrow E\}$$
Type order

Legend:  type  type order
Type order forms a strict partial order
Type order

Legend: 🟥 type 🍁 type order

A

B

C

D

E
Type order

Legend: type → type order instance
Type order

Legend: type order

A

a1 a2

B

b1 b2 b3

C

c1 c2

D

d1

E

e1 e2 e3
Type order

Legend:  
- **type**
- **type order**
- **instance**

A

- a1
- a2

B

- b1
- b2
- b3

C

- c1
- c2

D

- d1

E

- e1
- e2
- e3

Modifying references comply with type order
Type order

Legend:  type  type order  instance
         modifying reference

Modifying references comply with type order
Type order

Legend:  type  type order  instance  modifying reference

Modifying references comply with type order
Type order

Modifying references comply with type order
Type order

Legend:  
- type
- type order
- instance
- modifying reference

Modifying references comply with type order
Type order

Legend:  
- **type**  
- **type order**  
- **instance**  
- **modifying reference**

A
- a1
- a2

B
- b1
- b2
- b3

C
- c1
- c2

D
- d1

E
- e1
- e2
- e3
Type order

Legend:  
- Gray circle: type  
- Black arrow: type order  
- Gray dot: instance  
- Black double arrow: modifying reference

Read-only references are unrestricted
Type order

Legend:
- **type**
- **type order**
- **instance**
- **modifying reference**
- **read-only reference**

Read-only references are unrestricted
Type order

Legend:  

- type order
- modifying reference
- read-only reference

Read-only references are unrestricted
Type order

Legend: type □ type order → instance ● modifying reference

A

a1 □ a2

B

b1 ● b2 ● b3

C

c1 ● c2

D

d1

E

e1 ● e2 ● e3
Type order

Legend:  
- **type**  
- **type order**  
- **instance**  
- **modifying reference**

Heap forms DAG (w.r.t. modifying access)
Type order

Legend:
- Light grey: type
- Dark grey: type order
- Dark grey circle: instance
- Red arrow: modifying reference

A

a1

a2

B

b1

b2

b3

C

c1

c2

D

d1

E

e1

e2

e3
Type order + ownership

Legend:
- type
- type order
- instance
- modifying reference
Type order + ownership

Legend:  
- type  
- type order  
- instance  
- modifying reference

A

a1  a2

B

b1  b2  b3

C

c1  c2

D

d1

E

e1  e2  e3
Type order + ownership

Legend:
- type
- type order
- instance
- modifying reference
Type order + ownership

Legend: type order instance
modifying reference

A

B

C

D

E

F

G

a1 a2

b1 b2 b3

c1 c2
d1
e1 e2 e3

8
Type order + ownership

Legend:
- □ type
- ← type order
- ● instance
- → modifying reference
Type order + ownership

Legend: 
- type
- type order
- instance
- modifying reference

Diagram:

A: a1 a2
C: c1 c2
D: d1

B: b1 b2 b3
E: e1 e2 e3

F: f1 f2
G: g1
Type order + ownership

Legend:
- type
- type order
- instance
- modifying reference
- ownership (modifying ref)
Type order + ownership

Legend:
- type
- type order
- instance
- modifying reference
- ownership (modifying ref)
Type order + ownership

Legend:
- type
- type order
- instance
- modifying reference
- ownership (modifying ref)

A

a1
a2

d1

C

b1
b2
b3

e1 e2 e3

c1

c2

D

E

8
Type order + ownership

Legend:
- type
- type order
- instance
- modifying reference
- ownership (modifying ref)
Type order + ownership

Legend:
- type
- type order
- instance
- modifying reference
- ownership (modifying ref)
Type order + ownership

Legend:
- type
- type order
- instance
- modifying reference
- ownership (modifying ref)
Type order + ownership

Legend:
- type
- type order
- instance
- modifying reference
- ownership (modifying ref)
Type order + ownership

Legend:
- type
- type order
- instance
- modifying reference
- ownership (modifying ref)
Type order + ownership

Legend:
- type
- type order
- instance
- modifying reference
- ownership (modifying ref)
Type order + ownership

Legend:
- **type**
- **type order**
- **instance**
- **modifying reference**
- **ownership (modifying ref)**

---

A

- a1
- a2

B

- b1
- b2
- b3

C

- c1
- c2

D

- d1

E

- e1
- e2
- e3

F

- f1
- f2

G

- g1
Type order + ownership

Legend:
- type
- type order
- instance
- modifying reference
- ownership (modifying ref)
Type order + ownership

Heap forms DAG with “sub-trees” (w.r.t. modifying access)
Heap forms DAG with “sub-trees” (w.r.t. modifying access)
Type order + ownership

Legend:
- type
- type order
- instance
- modifying reference
- ownership (modifying ref)

Heap forms DAG with “sub-trees” (w.r.t. modifying access)
Recap
Recap

OO verification challenges

• Visible-state semantics in presence of call-backs
• Modular verification of multi-object invariants

Ownership-based verification

• Leverages tree topology to prevent modifying call-backs
• Leverages tree topology to encapsulate ownership-based invariants
Recap

OO verification challenges

• Visible-state semantics in presence of call-backs
• Modular verification of multi-object invariants

Ownership-based verification

• Leverages tree topology to prevent modifying call-backs
• Leverages tree topology to encapsulate ownership-based invariants

Selective ownership

• Leverages type order to prevent modifying call-backs
• Leverages ownership to encapsulate ownership-based invariants
Verification: type order
Verification: type order
Verification: type order

No transitive call-backs
Verification: type order
Verification: type order
Verification: type order

invariant (b2)
Visible-state semantics for single-object invariants is sound
Verification: type order + object ownership
Verification: type order + object ownership

A

a1 a2

C

c1 c2

d1

D

B

b1 b2 b3

e1 e2 e3

E

F

f1 f2

g1

G
Verification: type order + object ownership
Verification: type order + object ownership

A

a1  a2

C

c1  c2

D
d1

E
e1  e2  e3

F

B

b1  b2  b3

G

f1  f2  f3

g1
Verification: type order + object ownership

invariant (b2)
Verification: type order + object ownership

invariant (b2,d1)
Verification: type order + object ownership

Modular verification of multi-object invariants
Relationships & selective ownership
Relationships & selective ownership

Relationship-based programming languages

• Successors to OO languages
• Relationships as first-class citizens

First-class relationships

• Naturally give rise to a type order
Relationships & selective ownership

Relationship-based programming languages

• Successors to OO languages
• Relationships as first-class citizens

First-class relationships

• Naturally give rise to a type order

Naturally support selective ownership
In the paper

Rumer

- Relationship-based programming and specification language

Running example: tree

- Specification in Rumer
- Tree invariants
- Selective ownership-based verification
Relationships & type order
class Node {
...
}

relationship Parent participants (Node child, Node parent) {
    void link(Node c, Node p) {...}
}

relationship Tree participants (Node root, Set<Parent> tree) {
    void appendTree(Tree t, Node p) {...}
}
class Node {...}

relationship Parent participants (Node child, Node parent) {
    void link(Node c, Node p) {...}
}

relationship Tree participants (Node root, Set<Parent> tree) {
    void appendTree(Tree t, Node p) {...}
}
class Node {...}

relationship Parent participants (Node child, Node parent) {
    void link(Node c, Node p) {...}
}

relationship Tree participants (Node root, Set<Parent> tree) {
    void appendTree(Tree t, Node p) {...}
}
class Node {
  ...
}

relationship Parent participants (Node child, Node parent) {
  void link(Node c, Node p) {...}
}

relationship Tree participants (Node root, Set<Parent> tree) {
  void appendTree(Tree t, Node p) {...}
}
Relationships & type order

class Node {...}

relationship Parent participants (Node child, Node parent) {
    void link(Node c, Node p) {...}
}

relationship Tree participants (Node root, Set<Parent> tree) {
    void appendTree(Tree t, Node p) {...}
}
class Node {...}

relationship Parent participants (Node child, Node parent) {
    void link(Node c, Node p) {...}
}

relationship Tree participants (Node root, Set<Parent> tree) {
    void appendTree(Tree t, Node p) {...}
}
class Node {...}

relationship Parent participants (Node child, Node parent) {
    void link(Node c, Node p) {...}
}

relationship Tree participants (Node root, Set<Parent> tree) {
    void appendTree(Tree t, Node p) {...}
}
Relationships & type order

```java
class Node {...}

relationship Parent participants (Node child, Node parent) {
    void link(Node c, Node p) {...}
}

relationship Tree participants (Node root, Set<Parent> tree) {
    void appendTree(Tree t, Node p) {...}
}
```

Type order defined by relationship declarations

{Tree ➔ Parent, Tree ➔ Node}
Relationships & type order

```
class Node {...}

relationship Parent participants (Node child, Node parent) {
    void link(Node c, Node p) {...}
}

relationship Tree participants (Node root, Set<Parent> tree) {
    void appendTree(Tree t, Node p) {...}
}
```

Type order defined by relationship declarations

{Tree → Parent, Tree → Node}
Relationships & type order

```java
class Node {...}

relationship Parent participants (Node child, Node parent) {
    void link(Node c, Node p) {...}
}

relationship Tree participants (Node root, Set<Parent> tree) {
    void appendTree(Tree t, Node p) {...}
}
```

Type order defined by relationship declarations

{Tree → Parent, Tree → Node, Parent → Node}
Relationships & instance ownership
class Node {...}

relationship Parent participants (Node child, Node parent) {
    void link(Node c, Node p) {...}
}

relationship Tree participants (Node r, owned Set<Parent> t) {
    void appendTree(Tree t, Node p) {...}
}
Relationships & instance ownership

```java
class Node {...}

relationship Parent participants (Node child, Node parent) {
    void link(Node c, Node p) {...}
}

relationship Tree participants (Node r, owned Set<Parent> t) {
    void appendTree(Tree t, Node p) {...}
}
```

Ownership relation must be subset of type order
class Node {...}

relationship Parent participates (Node child, Node parent) {
    void link(Node c, Node p) {...}
}

relationship Tree participates (Node r, owned Set<Parent> t) {
    void appendTree(Tree t, Node p) {...}
}
Conclusions
Conclusions

Selective ownership = type [+ object ownership]

• Leverages type order to prevent modifying call-backs
• Leverages ownership to encapsulate ownership-based invariants
Conclusions

Selective ownership = type [+ object ownership]

• Leverages type order to prevent modifying call-backs
• Leverages ownership to encapsulate ownership-based invariants

Sound modular verification of multi-object invariants
Conclusions

Selective ownership = type [+ object ownership]

• Leverages type order to prevent modifying call-backs
• Leverages ownership to encapsulate ownership-based invariants

Sound modular verification of multi-object invariants

Permits shared, modifying access to objects further down
Conclusions

Selective ownership = type [+ object ownership]

• Leverages type order to prevent modifying call-backs
• Leverages ownership to encapsulate ownership-based invariants

Sound modular verification of multi-object invariants
Permits shared, modifying access to objects further down
Type order naturally arises from first-class relationships
Thank you for your attention!