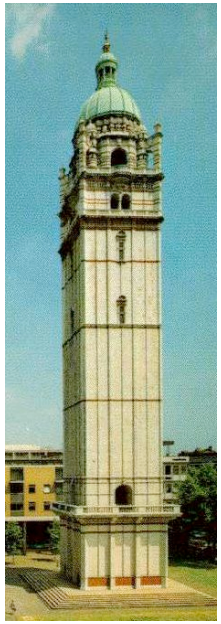


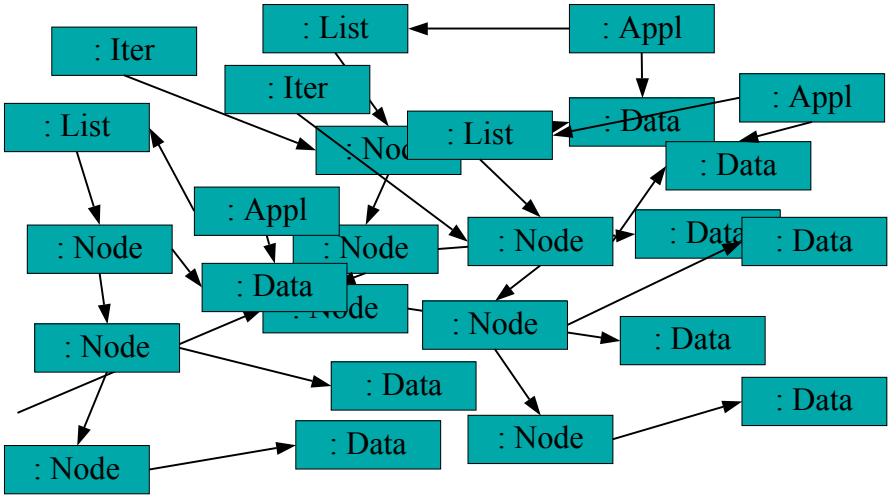
Types for Alias Control and Verification

Sophia Drossopoulou
Imperial College London

joint work with

Nick Cameron, IC
Dave Clake, Leuven
Dave Cunnigham, IC and IBM Research
Werner Dietl, ETH
Susan Eisenbach, IC
Adrian Francalanza, Univ. of Southampton
Peter Mueller, ETH and Microsoft Research
James Noble, Univ Wellington
Matthew Smith, IC
Alexander J Summers, IC
Tobias Wrigstad, Purdue

Motivation: Pointer Spaghetti



"Boxes" started 12 years ago ....

Flexible Alias Protection

James Noble[1], Jan Vitek[2], and John Potter[1]

[1] Microsoft Research Institute, Macquarie University, Sydney
kjx,potter@mri.mq.edu.au
[2] Object Systems Group, Université de Genève, Geneva.
Jan.Vitek@cui.unige.ch

Abstract. Aliasing is endemic in object oriented programming. Because an object can be modified via any alias, object oriented programs are hard to understand, maintain, and analyse. *Flexible alias protection* is a conceptual model of inter-object relationships which limits the visibility of changes via aliases, allowing objects to be aliased but mitigating the undesirable effects of aliasing. Flexible alias protection can be checked statically using programmer supplied *aliasing modes* and imposes no run-time overhead. Using flexible alias protection, programs can incorporate mutable objects, immutable values, and updatable collections of shared objects, in a natural object oriented programming style, while avoiding the problems caused by aliasing.

1 Introduction

I am who I am; I will be who I will be.

Object identity is the foundation of object oriented programming. Objects are useful for modelling application domain abstractions precisely because an object's identity always remains the same during the execution of a program —

...

OOOPSLA@08

Ownership Types for Flexible Alias Protection

David G. Clarke, John M. Potter, James Noble

Microsoft Research Institute, Macquarie University, Sydney, Australia
{clad,potter,kjx}@mri.mq.edu.au

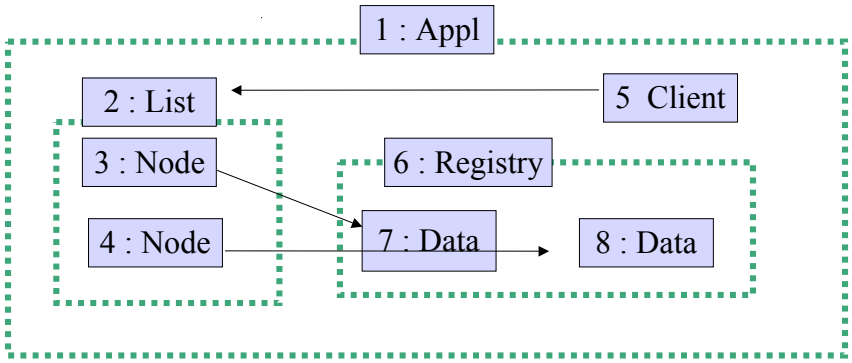
Abstract

Object-oriented programming languages allow inter-object aliasing. Although necessary to construct linked data structures and networks of interacting objects, aliasing is problematic in that an aggregate object's state can change via an

model of flexible alias protection, supported by illustrative examples, and suggested incorporating aliasing modes into programming languages. For flexible alias protection three key properties are required: no representation exposure; no role confusion between entities with different modes; and limited dependence on the mutable state of other objects

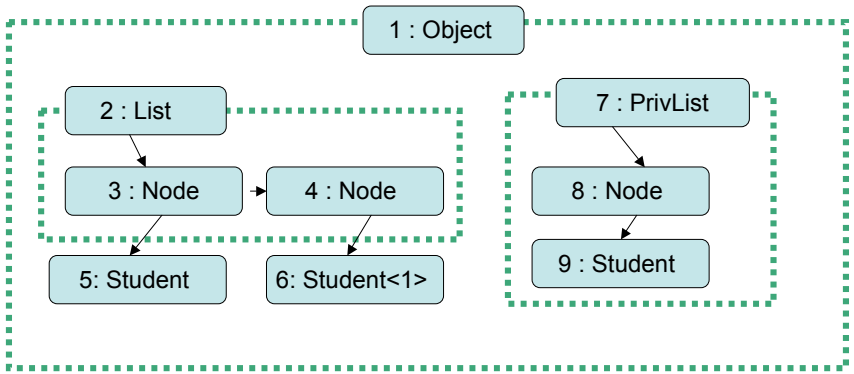
Box Types -- express structure

Heap organized into nested *boxes* (or contexts).
Boxes characterized by an object, which "owns" all the objects in the particular box.

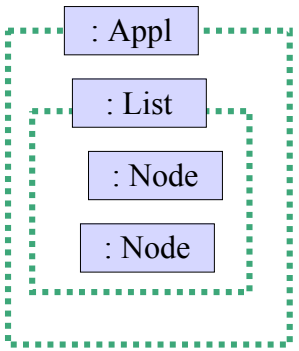


Boxes -- flavours

- Owners *as dominators*: eg forbid reference from 7 to 4
- Owners *as modifiers*: eg 4 only modified by 2 or 3
- Owners as a way to characterize heap areas



Box Types -- Uses



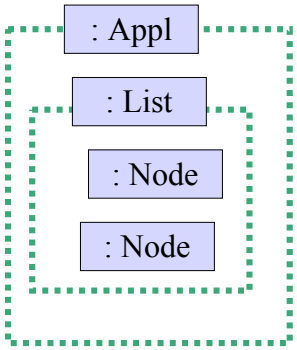
- Visualization: Noble, Potter, Clarke, ..
- **Owner as dominator**: memory management, Vitek, Noble, Palsberg,…
- Race Condition Detection: Boyapati, Rhinard, …
- Architectures: Aldrich, Chambers, Poetzsch-Heffter, …
- Reification: Naumann, Bannerjee
- Types: Clarke, Wrigstad, Drossopoulou, Cameron,…
- **Owner as modifier**: Verification: Leino, Mueller, Poetzsch-Hefter, …

Mueller: Modular Verification,
PhD thesis 2002

Peter Müller

Modular Specification and Verification of Object-Oriented Programs

Mueller: Modular Verification,
PhD thesis 2002



- Simple universe annotations to describe heap topology (rep, peer, any)
- **Owner as Modifier** Owners control modification of owned objects
- Invariants of objects only depend on the state of owned objects
- Execution outside the context of an object does not affect the object's invariants (modularity)

CONTENTS

- Improving expressivity of topology
  - Universe Types - type theoretic account
  - Generic Universe Types
  - Multiple Ownership
  - Universe Transfer Types
  - Tribe: Simple Dependent Types
- Universe Types for avoiding race conditions
- Universe/Ownership Types in Verification

CONTENTS

- Improving expressivity of topology
  - **Universe Types - type theoretic account**
  - Generic Universe Types
  - Multiple Ownership
  - Universe Transfer Types
  - Tribe: Simple Dependent Types
- Universe Types for avoiding race conditions
- Universe/Ownership Types in Verification

UT - FMCO'08

Universe Types for Topology and Encapsulation

Dave Cunningham[1], Werner Dietl[2], Sophia Drossopoulou[1], Adrian Francalanza[3], Peter Müller[2], and Alexander J. Summers[1]

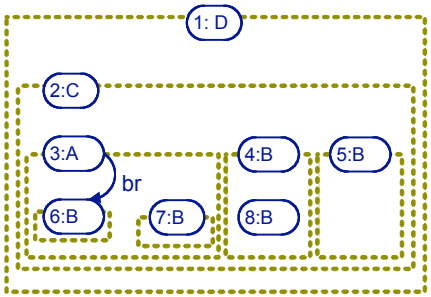
[1] Imperial College London
{david.cunningham04,s.drossopoulou,alexander.j.summers}@imperial.ac.uk
[2] ETH Zurich
{Werner.Dietl,Peter.Mueller}@inf.ethz.ch
[3] University of Southampton
af1@ecs.soton.ac.uk

Abstract. The Universe Type System is an ownership type system for object-oriented programming languages that hierarchically structures the object store; it is used to reason modularly about programs.

Universe Types: rep

```
class A {
      rep B br;
      peer B bp;
      any B ba;
 }
```

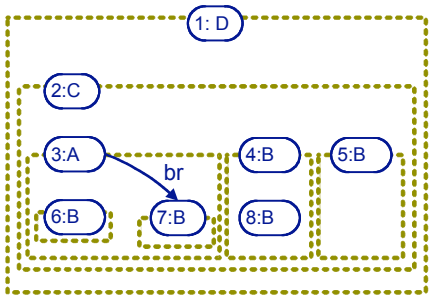
rep chacracterizes all "owned" objects



Universe Types: rep - 2

```
class A {
      rep B br;
      peer B bp;
      any B ba;
 }
```

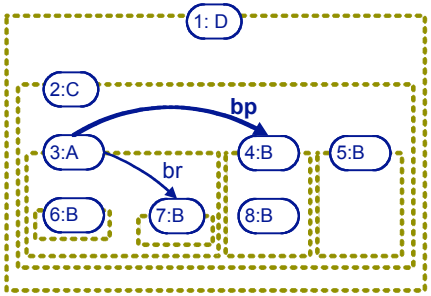
rep chacracterizes all "owned" objects



Universe Types: peer

```
class A {
      rep B br;
      peer B bp;
      any B ba;
 }
```

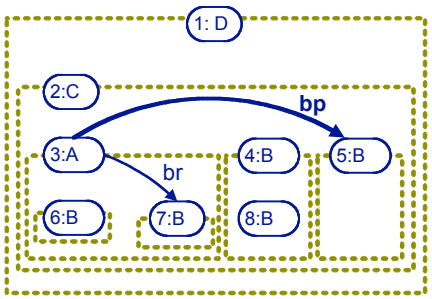
peer chacracterizes objects with same owner



Universe Types: peer - 2

```
class A {
      rep B br;
      peer B bp;
      any B ba;
 }
```

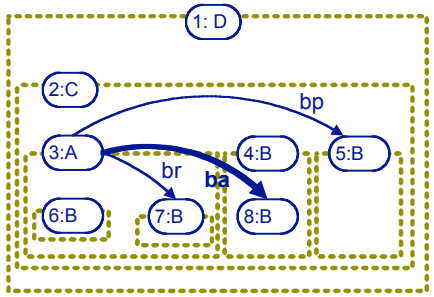
peer chacracterizes objects with same owner



Universe Types: any

```
class A {
      rep B br;
      peer B bp;
      any B ba;
 }
```

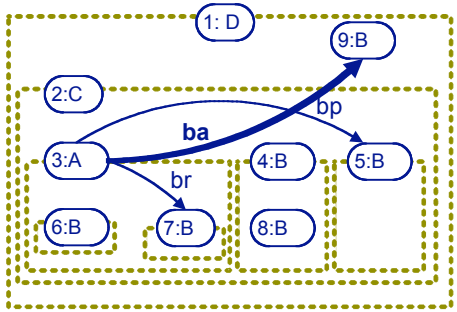
any chacracterizes *any* objects - don't care



Universe Types: any - 2

```
class A {
      rep B br;
      peer B bp;
      any B ba;
 }
```

any chacracterizes *any* objects - don't care



any allows to point anywhere, and in particular *inside* boxes.

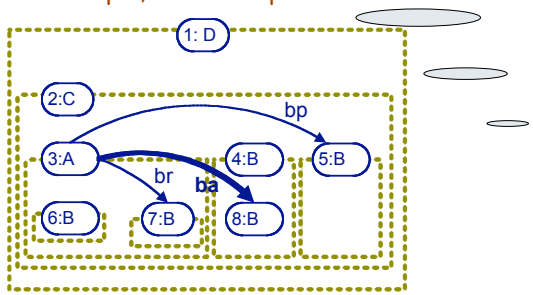
UJ Types

```
u  ::= any | rep | peer
t  ::= u c
```

types consist of a universe annotation and a class

SURPRISE: Types are relative!

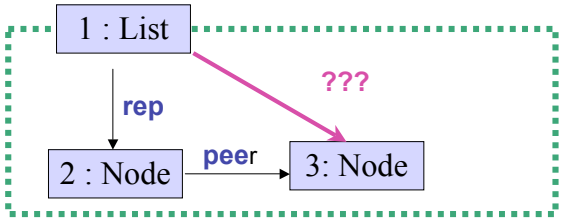
For example, in the heap h1:



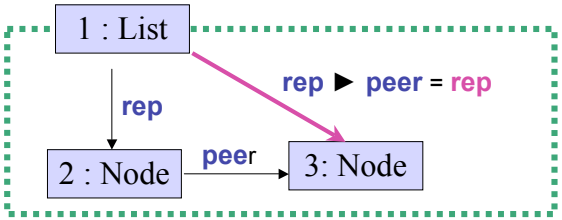

```
h1, 3 |-  4 : peer B
h1, 2 |-  4 : rep B
h1, 7 |-  4 : any B
```

NOTE: such "relative" types may (but need not) be used in generics.

UJ - Viewpoint adaptation



UJ - Viewpoint adaptation - 2



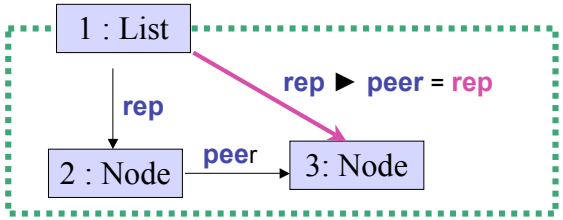
We define the operation ▶

```
peer ▶ peer  = peer
rep  ▶ peer  = rep
u    ▶ u'    = any
```

and prove Lemma 1:

h, o1|- o2 : u1 and h, o2|- o3 : u2 implies h, o1 |- o3 : u1 ▶ u2

UJ - Viewpoint adaptation - 3



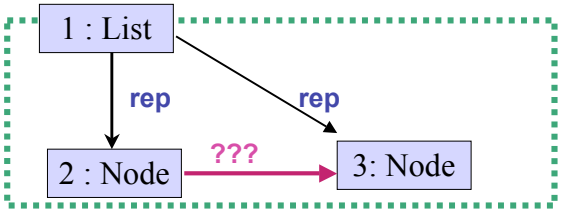
Lemma 1:

h, o1|- o2 : u1 and h, o2|- o3 : u2 implies h, o1 |- o3 : u1 ▶ u2

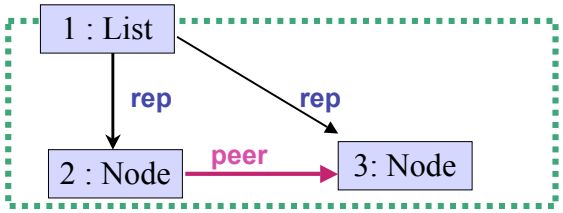
justifies the type rule:

$$\frac{e2 : u1\ c1 \qquad \text{class } c1 \text{ defines field } f \text{ of type } u2\ c2}{e2.f : (u1 \blacktriangleright u2)\ c2}$$

UJ - Viewpoint adaptation - 4



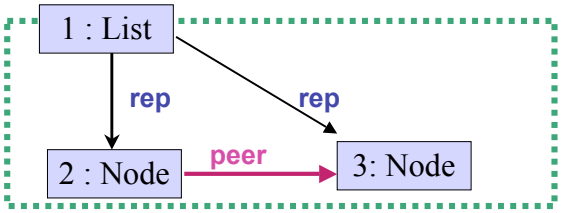
UJ - Viewpoint adaptation - 5



because

rep ▶ peer = rep and ∀u : rep ▶ u = rep ⇒ u = peer .

UJ - Viewpoint adaptation - 6



because

rep ▶ peer = rep and

∀u : rep ▶ u = rep ⇒ u = peer .

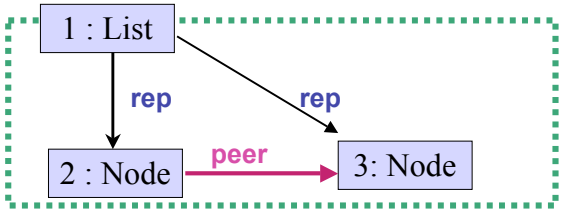
We also prove Lemma 2:

h, o1 |- o2 : u1 and h, o1 |- o3 : (u1 ▶ u2) and

∀u : u1 ▶ u = u1 ▶ u2 ⇒ u = u2

implies h, o2 |- o3 : u2

UJ - Viewpoint adaptation - 7



Lemma 2:

h, o1 |- o2 : u1 and h, o1 |- o3 : u1 ▶ u2 and

∀u : u1 ▶ u = u1 ▶ u2 ⇒ u = u2

implies h, o2 |- o3 : u2

justifies the type rule

$$\frac{e2 : u1\ c1 \qquad \text{class } c1 \text{ defines field } f \text{ of type } u2\ c2 \qquad e3 : (u1 \blacktriangleright u2)\ c3 \qquad \forall u : u1 \blacktriangleright u = u1 \blacktriangleright u2 \Rightarrow u = u2}{e2.f := e3 \ : \ void}$$

Theorem 1: UT Soundness

$\models h$ and

$\models \Gamma \vdash e : t$ and

$\models \Gamma, h \vdash s$ and

$\models e, s, h \nearrow \iota, h'$

imply

$\models h'$ and

$h', s(this) \vdash \iota : t$

Owner as modifier

We extend the type system to enforce the Owners as Modifiers discipline.

We extend the rule for field assignment

$$\frac{e2 : u1\ c1 \quad u1 \neq any \quad \text{class } c1 \text{ defines field } f \text{ of type } u2\ c2 \qquad e3 : (u1 \blacktriangleright u2)\ c3 \quad \forall u : u1 \blacktriangleright u = u1 \blacktriangleright u2 \Rightarrow u = u2}{e2.f := e3 \ : \ void}$$

We extend the rule for method call

.....

and can then prove ..

Theorem 2: Owner as modifier

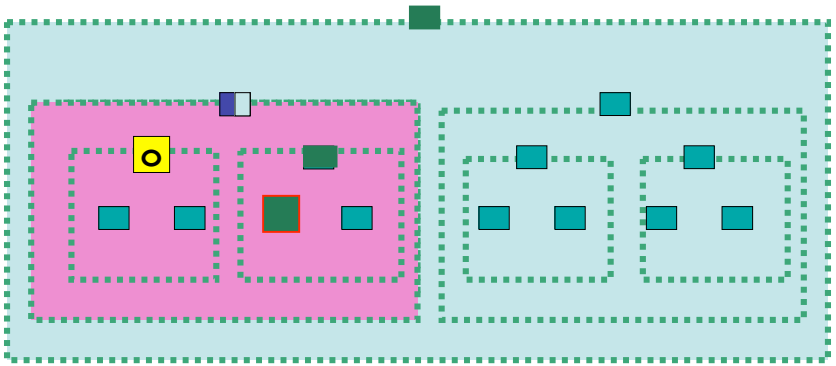
$\vDash h$ and $\Gamma \vdash e : t$ and $s(this) = o$ and $e, s, h \rightsquigarrow \iota, h'$ imply, for all objects o':

$Ownr(o) \in Ownr^*(o')$, or o' unmodified,

Theorem 2: Owner as modifier

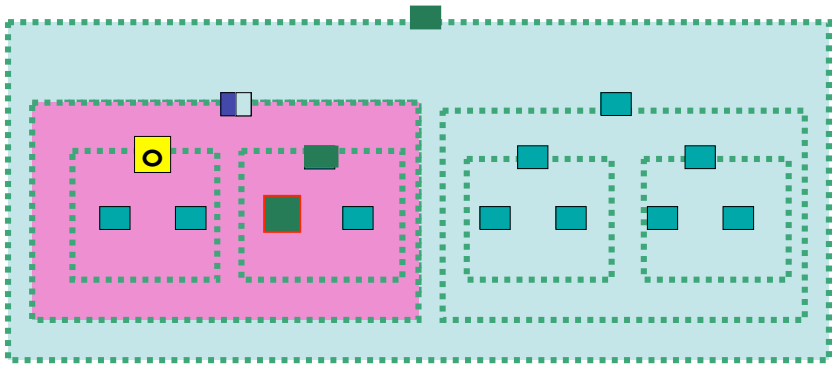
$\vDash h$ and $\Gamma \vdash e : t$ and $s(this) = o$ and $e, s, h \rightsquigarrow \iota, h'$ imply, for all objects o':

$Ownr(o) \in Ownr^*(o')$, or o' unmodified,



Owners as modifier - 2

Enables Verification of
Invariants and Frame Properties



CONTENTS

- Improving expressivity of topology
  - Universe Types - type theoretic account
  - **Generic Universe Types**
  - Multiple Ownership
  - Universe Transfer Types
  - Tribe: Simple Dependent Types
- Universe Types for avoiding race conditions
- Universe/Ownership Types in Verification

GUJ @ FOOL'07 and ECOOP'07

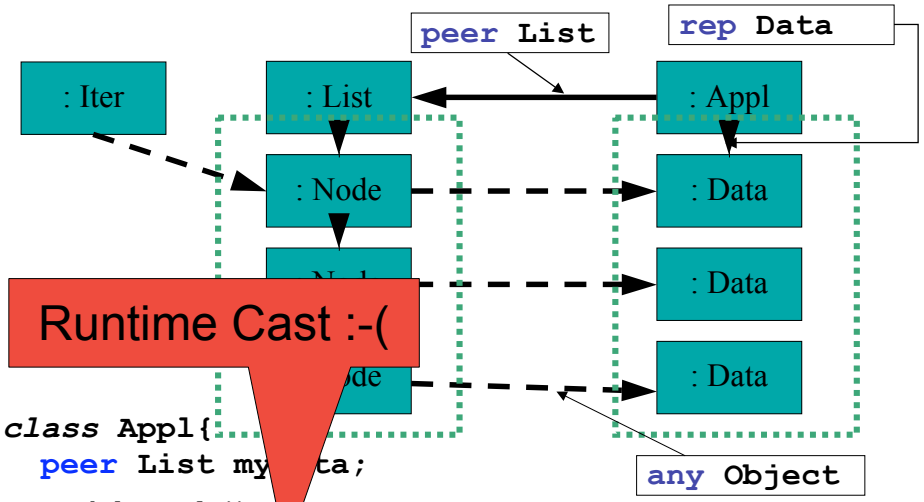
Generic Universe Types

Werner Dietl[1], Sophia Drossopoulou[2], and Peter Müller[1]

[1] ETH Zurich
{Werner.Dietl,Peter.Mueller}@inf.ethz.ch
[2] Imperial College London
S.Drossopoulou@imperial.ac.uk

Abstract. Ownership is a powerful concept to structure the object store and to control aliasing and modifications of objects. This paper presents an ownership type system for a Java-like programming language with

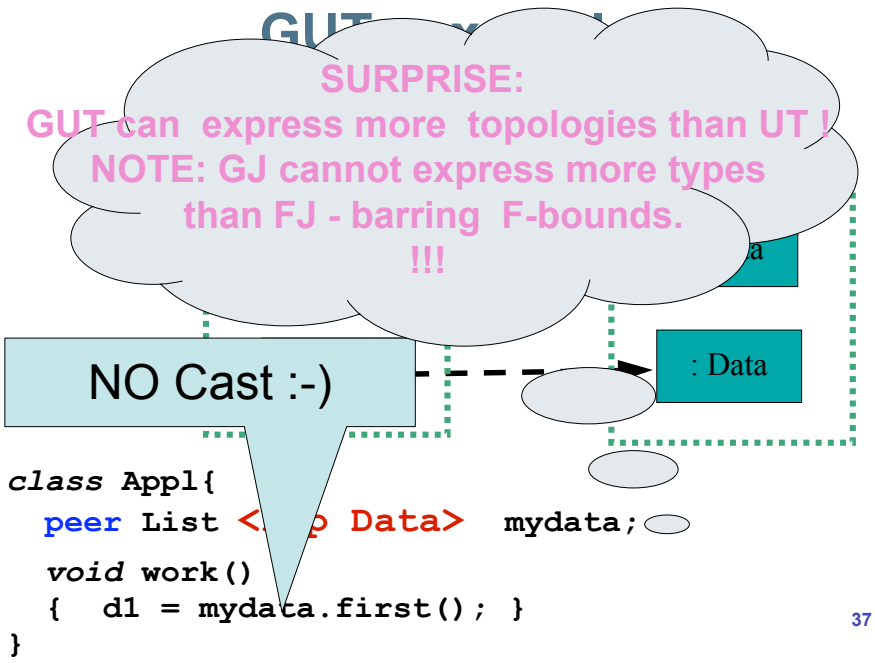
GUT Motivation: Lists in UT



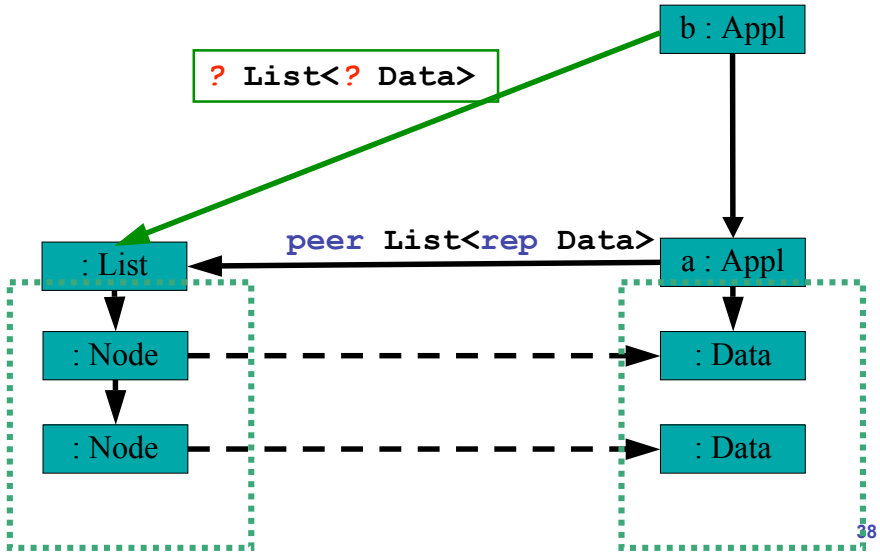

```
class Appl{
  peer List my   ta;
  void work()
  {  d1 = (rep Data) mydata.first(); }
}
```


GUT - example

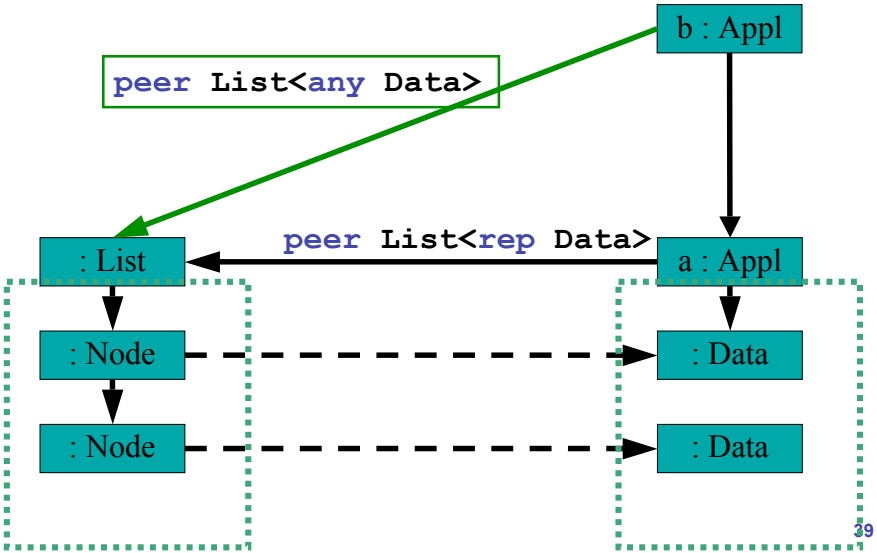
```
class Node<Y> {
  peer Node<Y> next;
  Y elem;
  Node(peer Node<Y> n, Y e){...}
}

class List<X> {
  rep Node<X> head;
  void add(X o) {
    head = new rep Node<X>(head, o);
  }
}
```



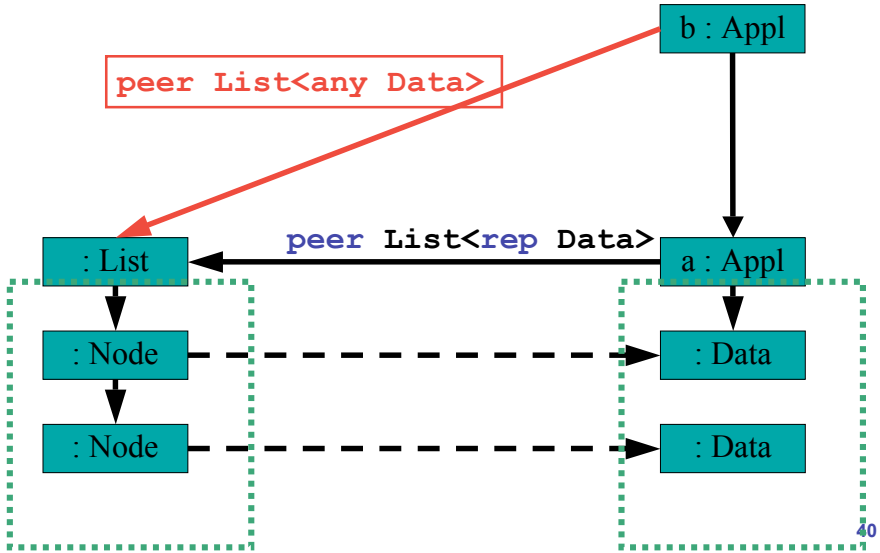
GUT - viewpoint adaptation revisited



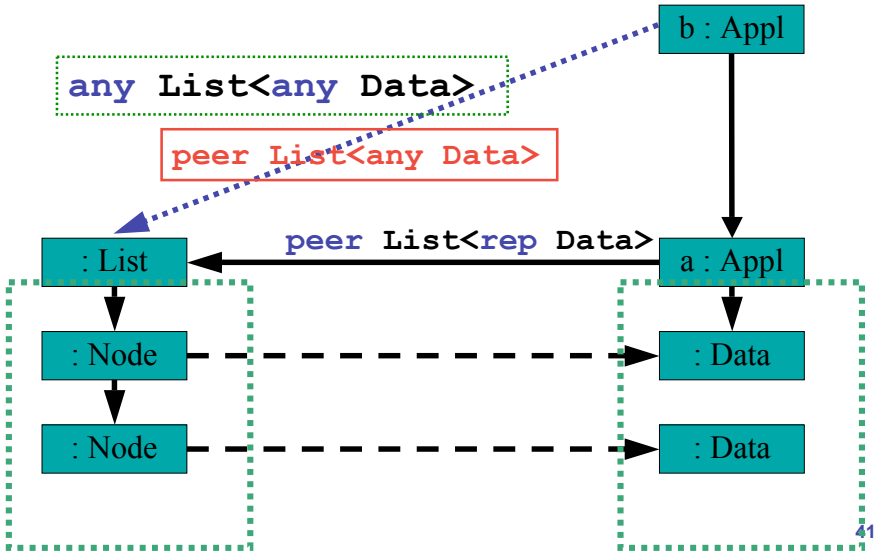
GUT - viewpoint adaptation 1st attempt



GUT - viewpoint adaptation 1st attempt is unsound

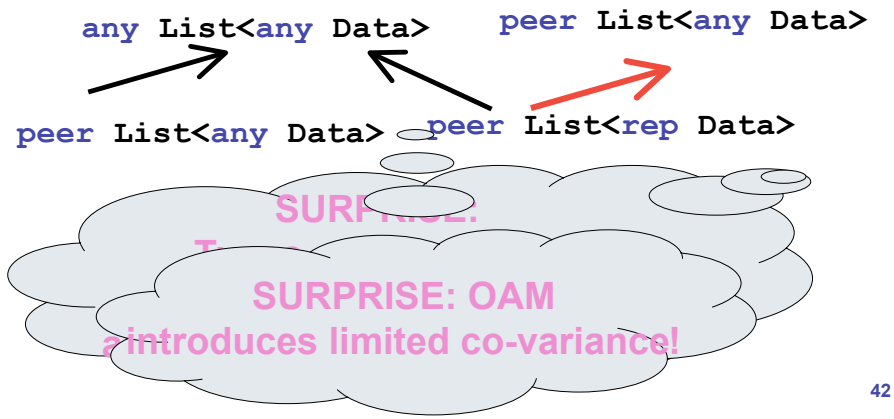


1st Solution, using OAMs



GUT1st Solution (using OAMs) and subtypes

`any` means a) do not care where,
b) not allowed to modify its fields (OAM).



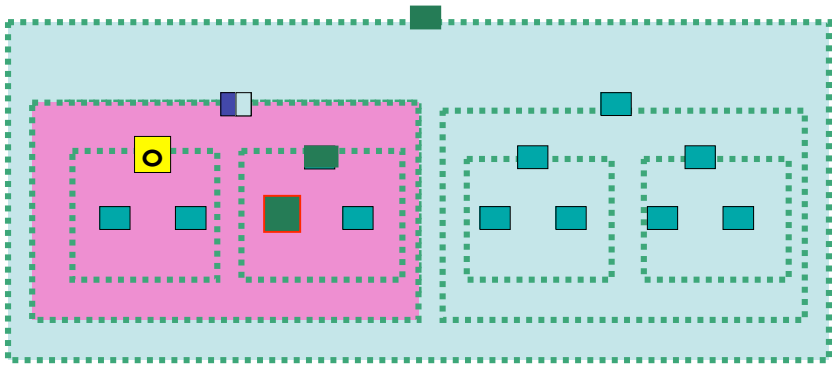
Thorem 3: GUT Soundness

$\vDash h$ and $\Gamma \vdash e : t$ and $\Gamma, h \vdash s$ and
$e, s, h \rightsquigarrow \iota, h'$
imply
$\vDash h'$ and $h', s(\mathtt{this}) \vdash \iota : t$

Theorem 4: Owner as modifier

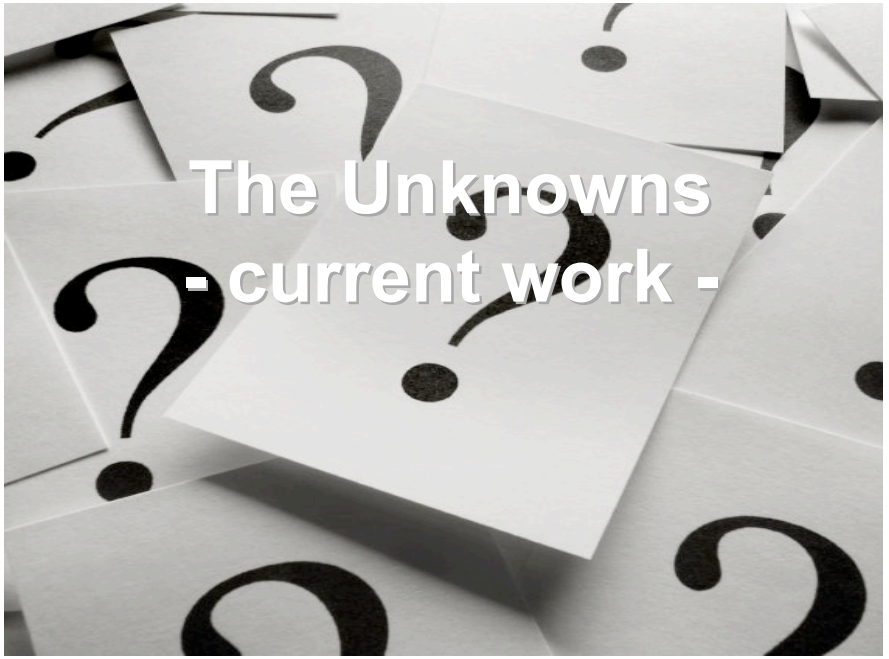
$\vDash h$ and $\Gamma \vdash e : t$ and $s(\mathtt{this}) = o$ and $e, s, h \rightsquigarrow \iota, h'$
imply, for all objects o':

$Ownr(o') \in Ownr^*(o)$ or o' unmodified,

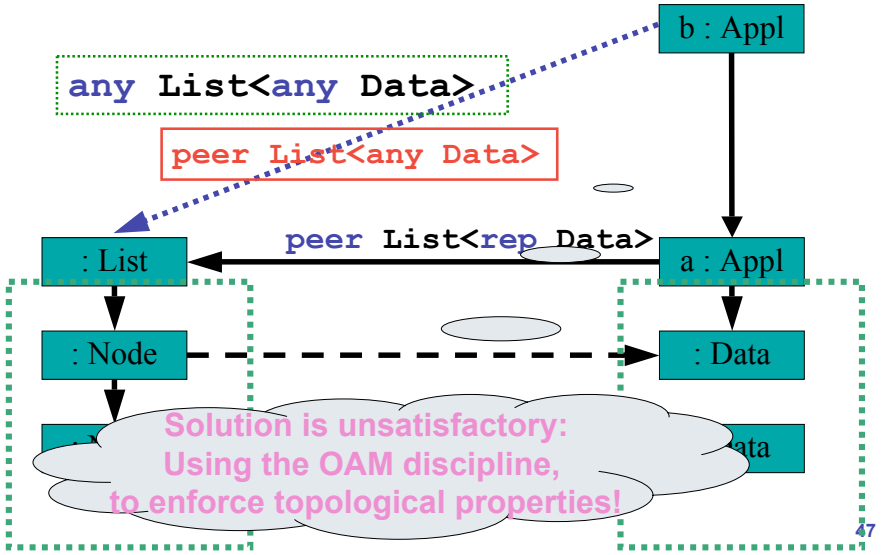


Formalization

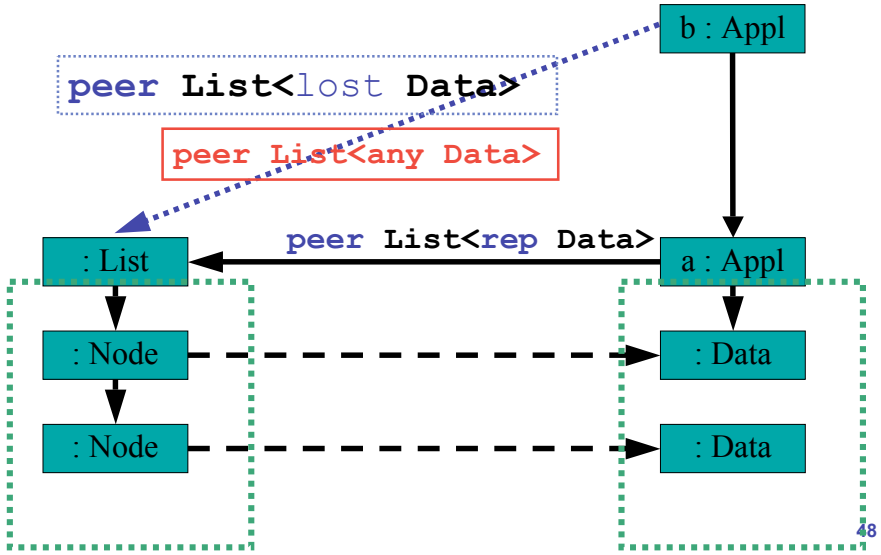
Details in the ECOOP paper!



1st Solution, using OAMs



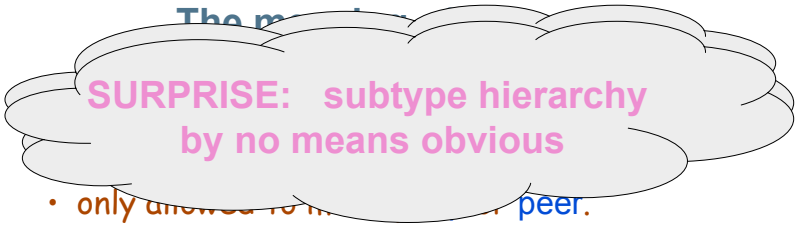
GUT - viewpoint adaptation
2nd Solution, introduce `lost`



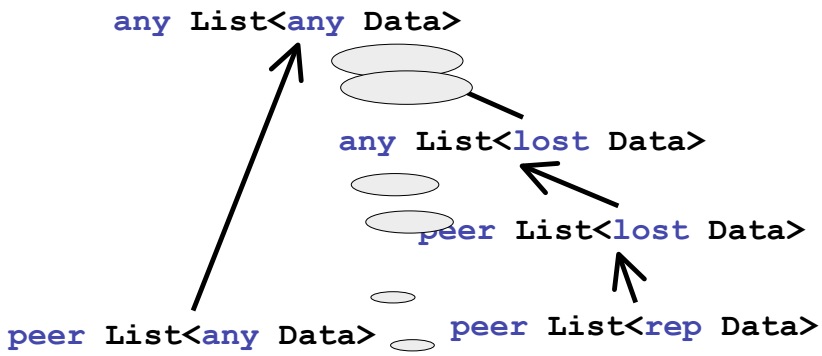
The meaning of lost and any

- So far, any was overloaded; it meant
  a) do not care where,
  b) not allowed to modify it.

- We refine the above
  - **any** means *do not care* where,
  - **lost** means *do not know* where,
  - only allowed to modify **rep** or **peer**.



- only allowed to [illegible] peer.



We need a further modifier: some

We refine the above

List<**any** E> Es may have different owners

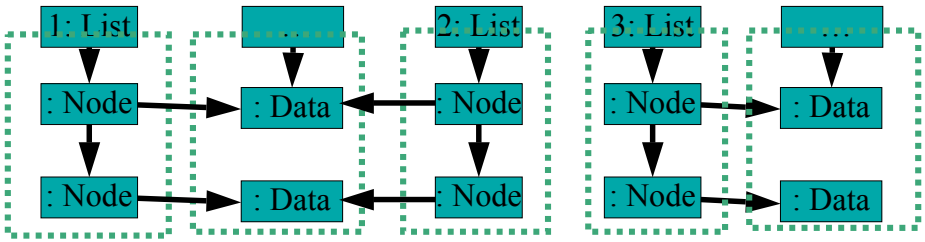
List<**some** E> Es have same owner,
do not care where this owner is

List<**lost** E> Es have same owner,
do not know where this owner is

Subtypes in the presence of some, lost and any

List<**some** E> Es same owner, do *not care* where it is
List<**lost** E> Es same owner, do *not know* where it is

Is it List<**some** E> <: List<**lost** E>
or
List<**lost** E> <: List<**some** E> ??



[[List<**some** E>]] = { 1, 2, 3 } = [[List<**lost** E>]] .
The two types represent the *same set* of values!

PER models (and Dezani) to the rescue!

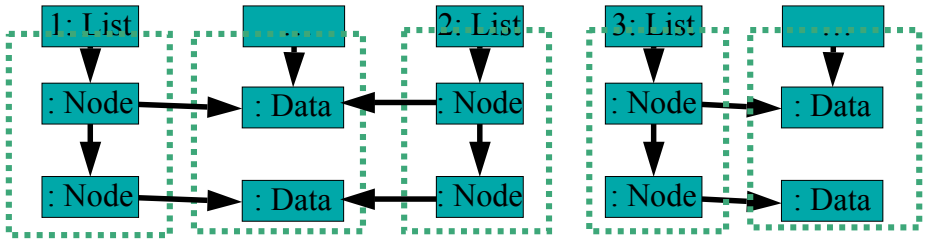
List<**some** E> Es same owner, do *not care* where it is

List<**lost** E> Es same owner, do *not know* where it is

Is it List<**some** E> <: List<**lost** E>

or

List<**lost** E> <: List<**some** E>



In List<**some** E> : 1 ~ 2 ~ 3

In List<**lost** E> : 1 ~ 2, 3 ~ 3

… a more traditional approach ..

(Cameron, Drossopoulou, ESOP 09)

Existential Quantification for Variant Ownership

Nicholas Cameron* and Sophia Drossopoulou

Imperial College London
{ncameron, scd}@doc.ic.ac.uk

Abstract. Ownership types characterize the topology of objects in the heap, through a characterization of the context to which an object belongs. They have been used to support reasoning, memory management, concurrency, etc. Subtyping is traditionally invariant w.r.t. contexts,

… would use existential types

List<∃o.o E > List<**any** E>

ie Es may have different owners

∃o. List<o E > List<**some** E>, List<**some** E>

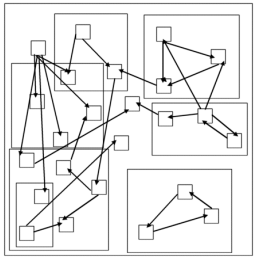
ie Es have the same owner.
The distinction between do not know and do not care is made through packing and unpacking.

CONTENTS

- Improving expressivity of topology
  - Universe Types - type theoretic account
  - Generic Universe Types
  - **Multiple Ownership**
  - Universe Transfer Types
  - Tribe: Simple Dependent Types
- Universe Types for avoiding race conditions
- Universe/Ownership Types in Verification

Objects in *Multiple* Boxes

- 75% of ownership structures require multiple ownership
  - [34] *Mitchell, ECOOP, '06*
- Trees can not describe non-hierarchical structures, e.g. tasks executed by workers, and owned by projects



MOJO, OOPSLA'07

Multiple Ownership

Nicholas Cameron, Sophia Drossopoulou
Imperial College London
{ncameron, sd}@doc.ic.ac.uk

James Noble *
Victoria University of Wellington, New Zealand
kjx@mcs.vuw.ac.nz

Matthew Smith
Imperial College London
mjs198@doc.ic.ac.uk

Abstract

Existing ownership type systems require objects to have precisely one primary owner, organizing the heap into an ownership tree. Unfortunately, a tree structure is too restrictive for many programs, and prevents many common design patterns where multiple objects interact.

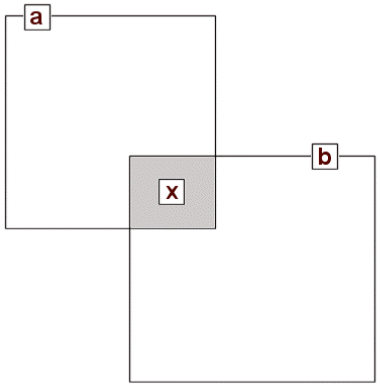
Multiple Ownership is an ownership type system where objects can have more than one owner, and the resulting ownership structure forms a DAG. We give a straightforward model for multiple ownership, focusing in particular on how multiple ownership can support a powerful effect system that determines when two

that this ownership structure does not suit all programs [5, 30, 38]. In this paper we present an ownership type system that removes this restriction, so that an object may have multiple owners, and the ownership relation forms a DAG. We make the following contributions:

- the *objects in boxes* model, a simple, straightforward model of object ownership based on sets of objects, which describes the fundamental features of single ownership, and generalises smoothly to multiple ownership.
- a language design incorporating Multiple Ownership

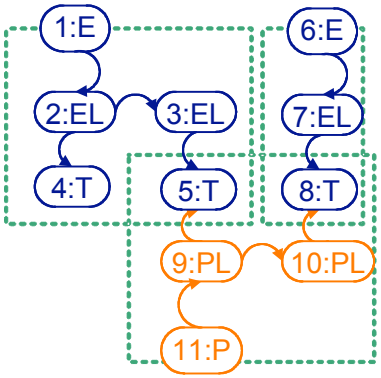
Objects in *Multiple* Boxes

$$x : \texttt{C<a \& b>} \Rightarrow x \in [\![\texttt{a}]\!] \cap [\![\texttt{b}]\!]$$



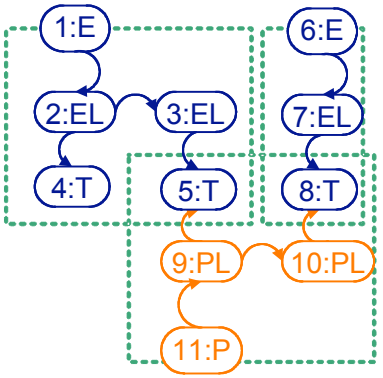
Task belongs to an employee and a project

```
class Employee<o> {
  List<this,this&any> tasks;
  void delay( ){ ... } }
class Project<o> {
   List<this,this&any> tasks;
   void delay( ){ ... } }
class List<o1,o2> {
   List<o1,o2> next;
   Task<o1,o2> task;
   void delay( ){ ... } }
class Task<o1>{  ...
   void delay(){ ... }; }
```



Effects

```
class Employee<o> {
   …
   void delay() this&any.undr
     {..}
}
class Project<o> {
 ...
 void delay() this&any.undr
     {..}
}
class List<o1,o2> {
   ...
   void delay()o2.undr {..}
}
class Task<o>{  ...
   void delay()this.undr{..}
}
```



Delaying an employee does not affect another

Thus,

```
e1.delay()  : (e1&any).under
e2.OK()     : (e2&any).under
```

Because

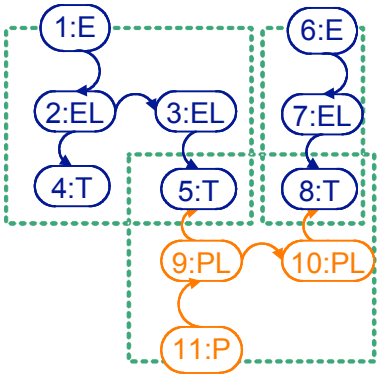
```
e1#e2 ⊢
  (e1&any).under #
  (e2&any).under
```

we have

```
e1#e2 ⊢ e1.delay() # e2.OK()
```

Similarly,

```
p1#p2 ⊢ p1.delay() # p2.OK()
```



Subtyping

- Subtyping with wildcards is variant with respect to owners:

```
C<a> <: C<?>
C<a> <: C<a & ?>
C<a & b & ?> <: C<a & ?>
```

- But:

```
C<?> ⊄: C<a>
C<a> ⊄: C<a & b>
C <a & b> ⊄: C<a>
```

CONTENTS

- Improving expressivity of topology
  - Universe Types - type theoretic account
  - Generic Universe Types
  - Multiple Ownership
  - **Universe Transfer Types**
  - Tribe: Simple Dependent Types
- Universe Types for avoiding race conditions
- Universe/Ownership Types in Verification

UTT OOPSLA'07

Ownership Transfer in Universe Types

Peter Müller
Microsoft Research, USA
mueller@microsoft.com

Arsenii Rudich
ETH Zurich, Switzerland
arsenii.rudich@inf.ethz.ch

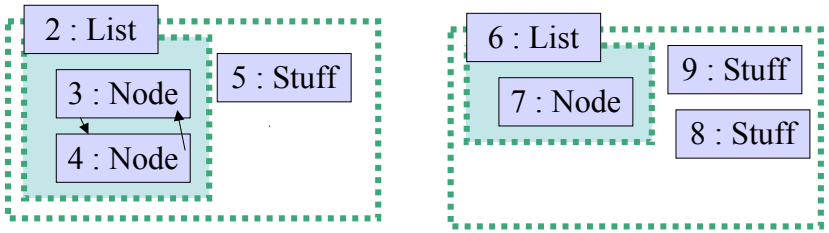
Abstract

Ownership simplifies reasoning about object-oriented programs by controlling aliasing and modifications of objects. Several type systems have been proposed to express and check ownership statically.

verify object invariants [29, 31, 33, 34], to show the absence of data races in multi-threaded programs [7, 26], to facilitate memory management for real-time programs [3, 9], to check object immutability [24], and to prove representation independence [4].

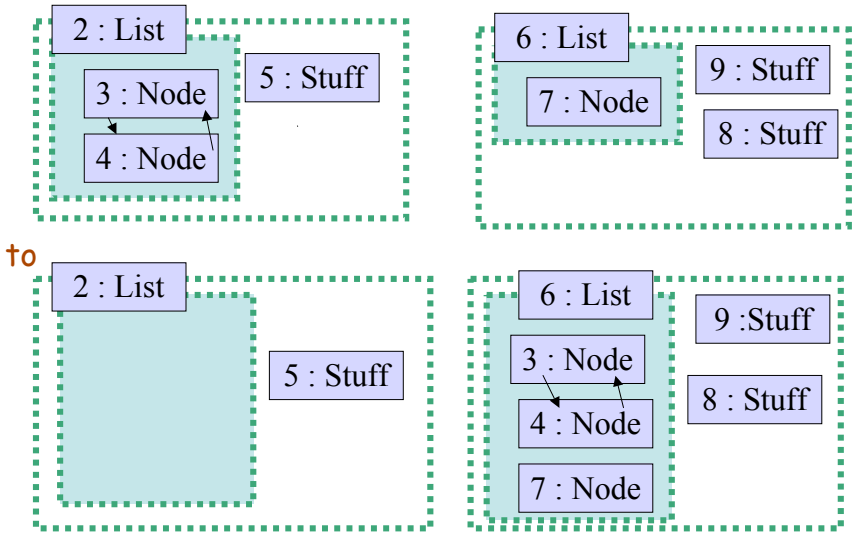
Supports transfer of whole clusters objects.

Eg, from



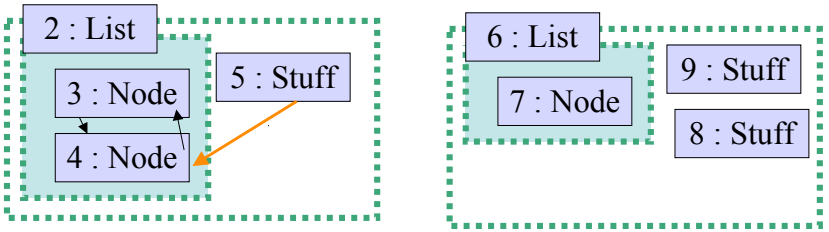
Supports transfer of whole clusters objects.

Eg, from

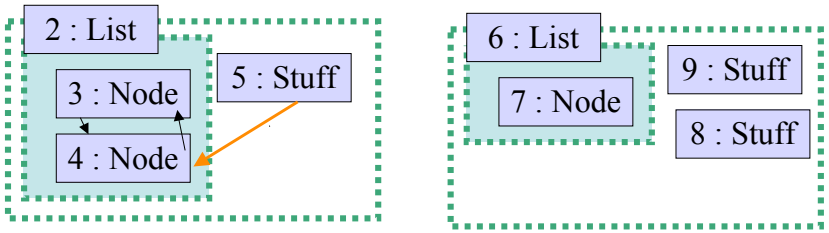


Transfer of clusters illegal, if at point of cluster there is non-any reference into cluster (static analysis checks).

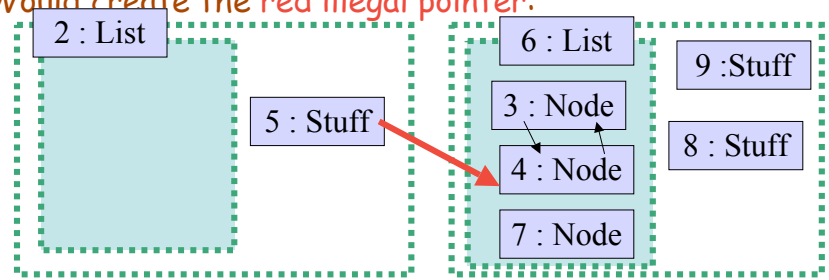
Transfer of clusters illegal, if at point of cluster there is non-any reference into cluster (static analysis checks).
For example, the following non-any pointer



Transfer of clusters illegal, if at point of cluster there is non-any reference into cluster (static analysis checks).
For example, the following pointer



Would create the red illegal pointer:



CONTENTS

- Improving expressivity of topology
  - Universe Types - type theoretic account
  - Generic Universe Types
  - Multiple Ownership
  - Universe Transfer Types
  - **Tribe: Simple Dependent Types**
- Universe Types for avoiding race conditions
- Universe/Ownership Types in Verification

Tribe @ AOSD'07

Tribe: More Types for Virtual Classes*

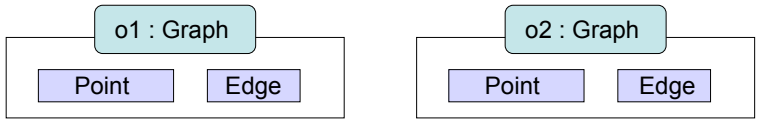
Dave Clarke[1], Sophia Drossopoulou[2], James Noble[3], and Tobias Wrigstad[4]

[1] CWI, Amsterdam, The Netherlands
[2] Imperial College London, UK
[3] Victoria University of Wellington, Wellington, NZ
[4] Stockholm University, Stockholm, Sweden

Abstract. Beginning with BETA, a range of programming language mechanisms have been developed to allow inheritance in the presence of mutually dependent classes. This paper presents Tribe, a type system

Tribe -2

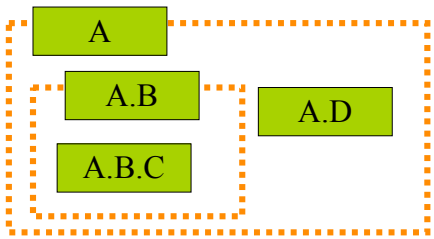
Classes depend on enclosing object



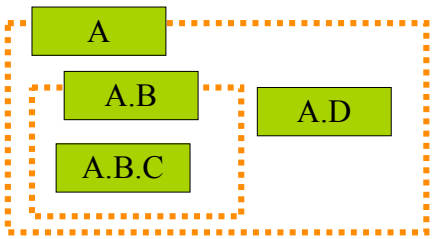
Types describe a path through the tree of objects

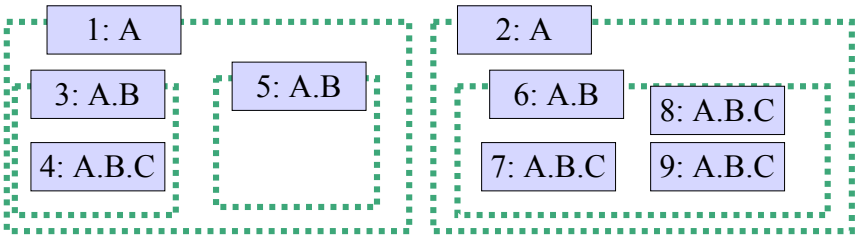
`T ::= ( x | this | up | C ).( C | ff )*`

Tribe classes are nested

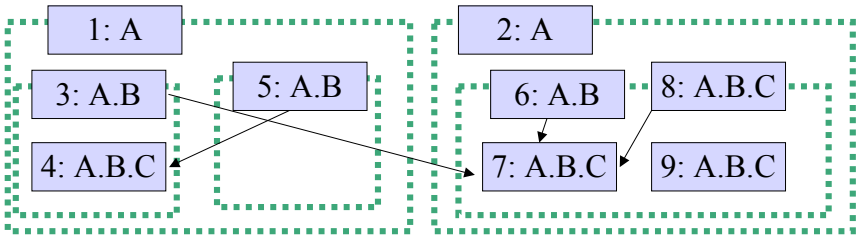


Object nesting reflects class nesting





Object nesting and field values give meaning to types



Consider the meaning of
`this.up.up.B.ff1`

from the viewpoint of object `4` or from `7`.

Tribe types for design patterns

Subject/Observer in Tribe

```
class SubjObs{
  class Subject{  … this.up.Observer o; … }
  class Observer{ … this.up.Subject s; … }
}
```


Tribe types for design patterns

Subject/Observer in Tribe

```
class SubjObs{
  class Subject{  … this.up.Observer o; … }
  class Observer{ … this.up.Subject s; … }
}
```

Subject/Observer in GJ

```
class Subject<O extends Observer<S,O>,
              S extends Subject<O,S>>
{ … Observer<S,O> o; … }
class Observer<S extends Subject<O,S>,
               O extends Observer<S,O>>
{ … Subject<O,S> … }
```


CONTENTS

- Improving expressivity of topology
  - Universe Types - type theoretic account
  - Generic Universe Types
  - Multiple Ownership
  - Universe Transfer Types
  - Tribe: Simple Dependent Types
- **Universe Types for avoiding race conditions**
- Universe/Ownership Types in Verification

VAMP 2007

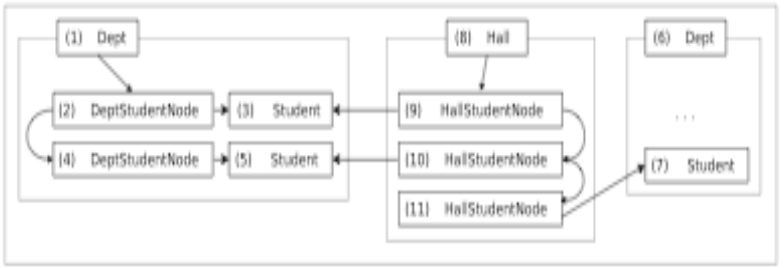
Universes for Race Safety

D. Cunningham [a,1] S. Drossopoulou[a,2] S. Eisenbach[a,3]

[a] *Department of Computing*
Imperial College London
U.K.

Abstract

Race conditions occur when two incorrectly synchronised threads simultaneously access the same object. Static type systems have been suggested to prevent them. Typically, they use annotations to determine the relationship between an object and its "guard" (another object), and to guarantee that the guard has been locked before the object is accessed. The object-guard relationship thus forms a tree similar to an ownership type hierarchy.
Universe types are a simple form of ownership types. We explore the use of universe types for static identification of race conditions. We use a small, Java-like language with universe types and concurrency

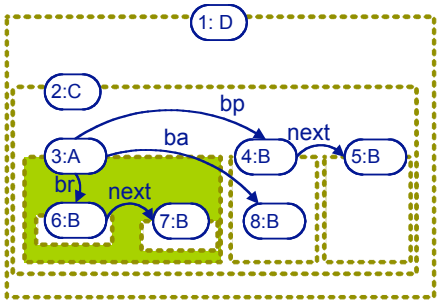



```
18 void releaseMarks () {
19   sync (this) {
20     rep DeptStudNode i = this.first;
21     sync (i) {
22       while (i!=null) {
23         i.s.mark = ...;
24         i = i.next;
25 } } } }
```

```
26 void cleanRooms () {
27   sync (this) {
28     rep HallStudNode i = this.first;
29     sync (i) {
30       while (i!=null) {
31         sync (i.s)
32           { i.s.roomClean = true; }
33         i = i.next;
34 } } } }
```

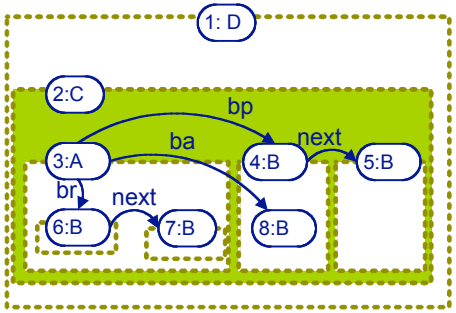

A thread may sync on an object, and thus lock all objects owned by it (but not object itself) -- 1

```
class B { peer B next; }

class A {
  rep B br;
  peer B bp;
  any B ba;

  void m1(){
   rep B br1 = br;
   sync this{
     while (br1){
            br1 = br1.next;
         }
    // above is atomic :-)
    // no data races   :-)
}
```



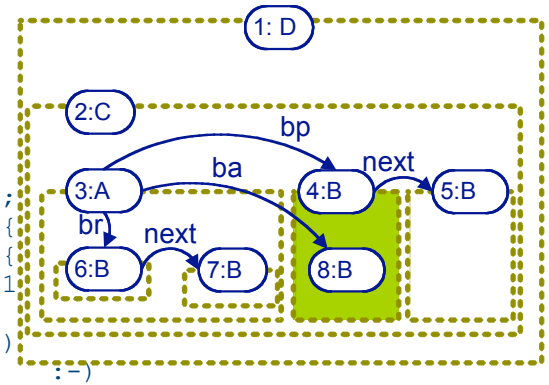
A thread may sync on an object, and thus lock all objects owned by it (but not object itself) -- 2

```
class B { peer B next; }

class A {
       rep B br;
       peer B bp;
       any B ba;

       void m2(){
      peer B bp1 = bp;
      sync bp1.owner{
         while (bp1){
            bp1 = bp1.next;
         }
    // above is atomic   :-)
    // no data races     :-)
}
```



A thread may sync on an object, and thus lock all objects owned by it (but not object itself) -- 3

```
class B { peer B next; }

class A {
   rep B br;
   peer B bp;
   any B ba;

   void m3(){
      any B ba1 = ba;
      lock ba1.owner{
         while (ba1){
            ba1 = ba1
         }
    // NOT atomic :-)
    // no data races   :-)
}
```



CONTENTS

- Improving expressivity of topology
  - Universe Types - type theoretic account
  - Generic Universe Types
  - Multiple Ownership
  - Universe Transfer Types
  - Tribe: Simple Dependent Types
- Universe Types for avoiding race conditions
- **Universe/Ownership Types in Verification**

Framework @ FOOL'08 and ECOOP'08

A Unified Framework for Verification Techniques for Object Invariants

S. Drossopoulou[1], A. Francalanza[2], P. Müller[3], and A. J. Summers[1]

[1] Imperial College London,
[2] University of Southampton,
[3] Microsoft Research, Redmond

Abstract. Object invariants define the consistency of objects. They have subtle semantics because of call-backs, multi-object invariants and

Object invariants - originally

```
class A{
   ....
   T m(T' x){
      // INV_A(this) expected
      ...
      ...
      prove INV_A(this)
   }
}
```

- invariant expected at method body begin/end (*visible states*)
- invariant temporarily broken within method body
- invariant to be proven at end of method

Object Invariants - Challenges

- **Call-backs** a method called when o-s invariant temporarily broken, might call back into o.
- **Multi-object invariant** when o-s invariant depends on o' state, then o-s invariant broken even when o not executing method.
- **Subclasses** if subclass invariant depends on superclass' fields, then superclass method might break subclass invariant.

These challenges increased by wish to offer *modular verification*, ie verification without knowledge of complete program.

Visible States Verification Techniques

Researchers have suggested a number of verification techniques to address these issues, eg Barnett&Nauman@MPCS'04, Husing&Kuiper@FASE'00, Leaves&Mueller@ICSE'07, Middlecoop&Huizing&Kuiper&Luit@SMBF'07,Poetzsch-HeffterHabil'97, Mueller&Petzsch-Heffter&Leavens@SCP'06, Lu&Potter@ECOOP'07,

- These techniques share commonalities but differ in
  - **Invariant Semantics** What invariant expected to hold when?
  - **Invariant Restrictions** Which objects may an invariant depend on?
  - **Proof Obligations** What has to be proven when?
  - **Program Restrictions** Which objects may receive method calls or field update?
  - **Type Systems**

- Invariants
  - Expected at the start of a method: X
  - Vulnerable to a method: V

```
void withdraw(int amount)
{
  ….

  this.balance -= amount;

  …
  this.card.pay(amount);

  ..
}
```

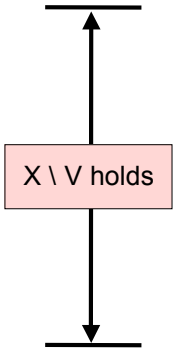

- Invariants
  - Expected at the start of a method: X
  - Vulnerable to a method: V

```
void withdraw(int amount)
{
  ….                                  poss break V

  this.balance -= amount;             poss break V

  …                                   poss break V
  this.card.pay(amount);              poss break V

  ..                                  poss break V
}
```

- Invariants
  - Expected at the start of a method: X
  - Vulnerable to a method: V

```
void withdraw(int amount)
{
  ….
  this.balance -= amount;
  …
  this.card.pay(amount);
  ..
}
```

poss break V
poss break V
poss break V
poss break V
poss break V



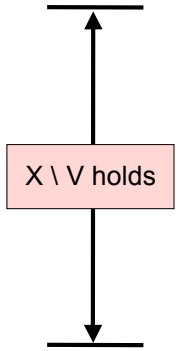
- Proof obligations
  - Before a method call: B
  - At the end of a method body: E

prove B

```
void withdraw(int amount)
{
  ….
  this.balance -= amount;
  …
  this.card.pay(amount);
  ..
}
```

prove B'

prove E



- Restrictions (control which invariants may be broken)
  - Receivers of method calls: C
  - Receivers of field updates: U

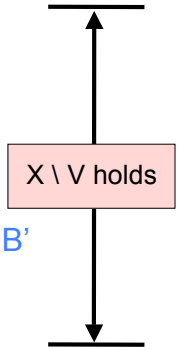
prove B

```
void withdraw(int amount)
{
  ….
  this.balance -= amount;
  …
  this.card.pay(amount);
  ..
}
```

check U

check C; prove B'

prove E



Our contributions

- formalize such techniques through a framework, independent of type-system and verification logic,
- characterize techniques in terms of 7 framework parameters,
- give 5 criteria guarantee soundness,
- instantiate the framework parameters and obtain six techniques from literature,
- find and repair an error in such a technique.

Soundness Conditions - 1

$\mathbb{F} \subseteq C_{c,m,c',m'} \Rightarrow (\mathbb{F} \rhd X_{c',m'}) \setminus (X_{c,m} \setminus V_{c,m}) \subseteq B_{c,m,\mathbb{F}}$

Soundness Conditions - 1a

$\mathbb{F} \subseteq C_{c,m,c',m'} \Rightarrow (\mathbb{F} \rhd X_{c',m'}) \setminus (X_{c,m} \setminus V_{c,m}) \subseteq B_{c,m,\mathbb{F}}$

If method m *may legally call method* m'

Soundness Conditions - 1b

$\mathbb{F} \subseteq C_{c,m,c',m'} \Rightarrow (\mathbb{F} \rhd X_{c',m'}) \setminus (X_{c,m} \setminus V_{c,m}) \subseteq B_{c,m,\mathbb{F}}$

If method m *may legally call method* m' *, then any invariants expected by* m'

Soundness Conditions - 1c

$\mathbb{F} \subseteq C_{c,m,c',m'} \Rightarrow (\mathbb{F} \rhd X_{c',m'}) \setminus (X_{c,m} \setminus V_{c,m}) \subseteq B_{c,m,\mathbb{F}}$

If method m *may legally call method* m' *, then any invariants expected by* m' *, which are not known to currently hold*

Soundness Conditions - 1d

$\mathbb{r} \subseteq C_{c,m,c',m'} \Rightarrow (\mathbb{r} \rhd X_{c',m'}) \setminus (X_{c,m} \setminus V_{c,m}) \subseteq B_{c,m,\mathbb{r}}$

If method m *may legally call method* m'*, then any invariants expected by* m'*, which are not known to currently hold, must be proved before calling* m'

Soundness Conditions 1-5

$\mathbb{r} \subseteq C_{c,m,c',m'} \Rightarrow (\mathbb{r} \rhd X_{c',m'}) \setminus (X_{c,m} \setminus V_{c,m}) \subseteq B_{c,m,\mathbb{r}}$

$V_{c,m} \cap X_{c,m} \subseteq E_{c,m}$

$C_{c,m,c',m'} \rhd (V_{c',m'} \setminus E_{c',m'}) \subseteq V_{c,m}$

$U_{c,m,c'} \rhd D_{c'} \subseteq V_{c,m}$

$c' \leq c \Rightarrow X_{c',m} \subseteq X_{c,m}$ & $(V_{c',m} \setminus E_{c',m}) \subseteq (V_{c,m} \setminus E_{c,m})$

Soundness Conditions 1-5

$\mathbb{r} \subseteq C_{c,m,c',m'} \Rightarrow (\mathbb{r} \rhd X_{c',m'}) \setminus (X_{c,m} \setminus V_{c,m}) \subseteq B_{c,m,\mathbb{r}}$

$V_{c,m} \cap X_{c,m} \subseteq E_{c,m}$

$C_{c,m,c',m'} \rhd (V_{c',m'} \setminus E_{c',m'}) \subseteq V_{c,m}$

$U_{c,m,c'} \rhd D_{c'} \subseteq V_{c,m}$

$c' \leq c \Rightarrow X_{c',m} \subseteq X_{c,m}$ & $(V_{c',m} \setminus E_{c',m}) \subseteq (V_{c,m} \setminus E_{c,m})$

A verification technique whose instantiation satisfies these conditions, is sound

However …

There techniques work only for hierarchical data structures; several patterns are not hierarchical, cf Composite design pattern

Invariants may sometimes be encoded as pre-post-conditions on methods; and thus may be considered as unnecessary

However …

There techniques work only for hierarchical data structures; several patterns are not hierarchical, cf Composite design pattern

Invariants may sometimes be encoded as pre-post-conditions on methods; and thus may be considered as unnecessary

We believe that invariant based techniques can (and should) be expanded to deal with combination of hierachical/non hierarchical structures

Summers, Drossopoulou, Mueller: The need for flexible object invariants, IWACO'09

Summers, Drossopoulou: Verification of the Composite Design Pattern, based on Object Invariants, done in Boogie, COST'09

Conclusions

- Universe/ownership type systems can express the topology of objects in the heap
- Such systems may be used to express refined effects, argue disjointness, and help for framing
- Several visible states semantics techniques are expressed as extensions of such topological systems
- Further work need to combine with non-hierarchical, and non-exclusive topologies
- Towards an architecture language for program specification?