Types for Alias Control and Verification
Sophia Drossopoulou
Imperial College London

joint work with
Nick Cameron, IC
Dave Clarke, 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

Approach: Boxes

“Boxes” started 12 years ago ....
Ownership Types for Flexible Alias Protection

David G. Clarke, John M. Peucer, James Noble
Microsoft Research Institute, Macquarie University, Sydney, Australia
(ji@peucer/je@msr@mace.edu)

Abstract
Objects-oriented programming languages allow open object references. Although structuring constructs like those in
other languages and networks of communicating objects, allowing to perform
tasks in these object-oriented object, one can show the on
model of flexible alias protection, supported by Infrastructure
concepts, and orthogonal mechanisms, allowing objects to
be dependent without explicit knowledge of the other object.

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

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-Heffter, ...
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
Universe Types for Topology and Encapsulation

Dave Cunningham¹, Werner Dietl², Sophia Drossopoulou¹, Adrian Francalanza³, Peter Miller⁴, and Alexander J. Summers⁵

¹ Imperial College London
dave.cunningham04,a.drossopoulou,alexander.j.summers@imperial.ac.uk
² ETH Zurich
Werner.Dietl,Peter.Mueller@inf.ethz.ch
³ University of Southampton
al@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 modularity about programs.

13

Universe Types: rep

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

rep characterizes all "owned" objects

14

Universe Types: rep - 2

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

rep characterizes all "owned" objects

15

Universe Types: peer

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

peer characterizes objects with same owner

16
**Universe Types: peer - 2**

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

peer characterizes objects with same owner

**Universe Types: any**

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

any characterizes any objects - don't care

**Universe Types: any - 2**

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

any characterizes any objects - don't care

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

**UJ Types**

```c
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.
We define the operation $\uparrow$

$$
\begin{align*}
\text{peer} \uparrow \text{peer} &= \text{peer} \\
\text{rep} \uparrow \text{peer} &= \text{rep} \\
\text{u} \uparrow \text{u}' &= \text{any}
\end{align*}
$$

and prove Lemma 1:

$$
\begin{align*}
\text{h, o1} &\vdash \text{o2 : u1} \quad \text{and} \quad \text{h, o2} &\vdash \text{o3 : u2} \quad \text{implies} \quad \text{h, o1} \vdash \text{o3 : u1} \uparrow \text{u2}
\end{align*}
$$

Lemma 1:

$$
\begin{align*}
\text{h, o1} &\vdash \text{o2 : u1} \quad \text{and} \quad \text{h, o2} &\vdash \text{o3 : u2} \quad \text{implies} \quad \text{h, o1} \vdash \text{o3 : u1} \uparrow \text{u2}
\end{align*}
$$

justifies the type rule:

$$
\begin{align*}
e_2 : \text{u1} \\text{class c1 defines field f of type u2} \text{ c2}
e_2.f : (\text{u1} \uparrow \text{u2}) \text{ c2}
\end{align*}
$$
We also prove Lemma 2:
\[ h, o1 \vdash o2 : u1 \land h, o1 \vdash o3 : (u1 \parallel u2) \land \forall u : u1 \parallel u = u \Rightarrow u = \text{peer}. \]

implies \[ h, o2 \vdash o3 : u2 \]

We also prove Lemma 2:
\[ h, o1 \vdash o2 : u1 \land h, o1 \vdash o3 : u1 \parallel u2 \land \forall u : u1 \parallel u = u1 \parallel u2 \Rightarrow u = u2 \]

implies \[ h, o2 \vdash o3 : u2 \]

Theorem 1: UT Soundness
\[ h \land \Gamma \vdash e : t \land \Gamma, h \vdash s \land e, s, h \vdash t, h' \]

implies \[ h' \land \Gamma, h' \vdash t : t \]

Lemma 2:
\[ e2 : u1 \ c1 \land \text{class c1 defines field f of type u2 c2} \]
\[ e3 : (u1 \parallel u2) \ c3 \land \forall u : u1 \parallel u = u1 \parallel u2 \Rightarrow u = u2 \]
\[ e2. f := e3 : \text{void} \]
Owner as modifier

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

We extend the rule for field assignment

\[
e2 : u1 \triangleright c1 \quad \text{any} \quad \text{class } c1 \text{ defines field } f \text{ of type } u2 \quad c2
\]

\[
e3 : (u1 \triangleright u2) \triangleright c3
\]

\[
\forall u : u1 \triangleright u = u1 \triangleright u2 \Rightarrow u = u2
\]

\[
e2.f := e3 : \text{void}
\]

Theorem 2: Owner as modifier

\[\vdash h \quad \text{and} \quad \Gamma \vdash e : t \quad \text{and} \quad s(\text{this}) = o \quad \text{and} \quad e, s, h \triangleleft h', h'' \quad \text{imply, for all objects } o':\]

\[\text{Ownr}(o) \subseteq \text{Ownr}^*(o') \quad \text{or} \quad o' \text{ unmodified.}\]
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

GUT @ FOOL’07 and ECOOP’07

Generic Universe Types

Werner Dietl¹, Sophia Drossopoulou², and Peter Müller¹

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 Node<X> {
  peer Node<X> next;
  X elem;
  Node(peer Node<X> n, X e){...}
}
class List<X> {
  rep Node<X> head;
  void add(X o) {
    head = new rep Node<X>(head, o);
  }
}
GUT - example

List

Node

Data

Appl

class Appl{
    peer List<rep Data> mydata;
    void work()
    { d1 = mydata.first(); }
}

NO Cast :-)

SURPRISE:
GUT can express more topologies than UT!
NOTE: GJ cannot express more types than FJ - barring F-bounds.

class Appl{
    peer List<rep Data> mydata;
    void work()
    { d1 = mydata.first(); }
}

GUT - viewpoint adaptation revisited

peer List<rep Data>

GUT - viewpoint adaptation
1st attempt

peer List<any Data>

GUT - viewpoint adaptation
1st attempt is unsound

peer List<any Data>
1st Solution, using OAMs

GUT 1st Solution (using OAMs) and subtypes

Thorem 3: GUT Soundness

Theorem 4: Owner as modifier
Formalization

1st Solution, using OAMs

Solution is unsatisfactory:
Using the OAM discipline,
to enforce topological properties!

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.

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\(<\text{some } E>\) Es same owner, do not care where it is
List\(<\text{lost } E>\) Es same owner, do not know where it is

Is it List\(<\text{some } E>\) \(<: \) List\(<\text{lost } E>\)
or
List\(<\text{lost } E>\) \(<: \) List\(<\text{some } E>\)

In List\(<\text{some } E>\): 1 ~ 2 ~ 3
In List\(<\text{lost } E>\): 1 ~ 2, 3 ~ 3

... would use existential types

List\(<\exists o. \text{o } E >\) List\(<\text{any } E >\)
\ie Es may have different owners

\exists o. List\(<\text{o } E >\) List\(<\text{some } E >\), List\(<\text{some } E >\)
\ie Es have the same owner.
The distinction between do not know and do not care is made through packing and unpacking.

... a more traditional approach ..
(Cameron, Drossopoulou, ESOP 09)

**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 cannot describe non-hierarchical structures, e.g. tasks executed by workers, and owned by projects
Task belongs to an employee and a project

```java
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

```java
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{ .. } }
```

Subtyping

- Subtyping with wildcards is variant with respect to owners:
  - C<a> <: C<?>
  - C<a> <: C<a & ?>
  - C<a & b & ?> <: C<a & ?>

- But:
  - C<?> $\not<$ C<a>
  - C<a> $\not<$ C<a & b>
  - C <a & b> $\not<$ 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

---

<table>
<thead>
<tr>
<th>Supports transfer of whole clusters objects.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eg, from</td>
</tr>
</tbody>
</table>

```
2 : List
3 : Node
4 : Node

5 : Stuff

6 : List
7 : Node
8 : Stuff

9 : Stuff
```

---

<table>
<thead>
<tr>
<th>Supports transfer of whole clusters objects.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eg, from</td>
</tr>
</tbody>
</table>

```
2 : List
3 : Node
4 : Node

5 : Stuff

6 : List
7 : Node
8 : Stuff

9 : Stuff
```

---

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Abstract</strong></td>
</tr>
<tr>
<td>Ownership simplifies reasoning about object-oriented programs by controlling aliasing and modifi-</td>
</tr>
<tr>
<td>cations of objects. Several type sys-</td>
</tr>
<tr>
<td>tems have been proposed to express and</td>
</tr>
<tr>
<td>check ownership variance.</td>
</tr>
</tbody>
</table>

Peter Müller  
Microsoft Research, USA  
muller@microsoft.com

Armeni Rudich  
ETH Zurich, Switzerland  
armeni.rudich@ethz.ch

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 [5, 9], to check object invariance [28], and to prove representation independence [4].
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:

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: More Types for Virtual Classes

Abstract. Beginning with BiTa, 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 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; ... }
}

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
Universes for Race Safety

D. Cunningham, S. Drosopoulou, S. Eisenbach

Abstract

Race conditions occur when two concurrently executing threads simultaneously access the same object. Race-tree systems have been suggested to prevent them. Typically, they are constructed 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. Our understanding is that a fine-grained or a Universe type is a simple form of mutex type. We explore the use of universe types for matter of sophistication of race conditions. We use a model that uses language types universe types and concurrency.

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

    void m2(){
        peer B bpl = bp;
        sync bpl.owner{
            while (bpl){
                bpl = bpl.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

```java
class B {
    peer B next;
}

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

    void m3(){
        any B bal = ba;
        lock bal.owner{
            while (bal)
                bal = bal.next;
            // 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

---

Object invariants - originally

```java
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

---

**Framework @ FOOL’08 and ECOOP’08**

A Unified Framework for Verification Techniques for Object Invariants

S. Drossopoulou(1), A. Frantzalas(2), P. Müller(3), and A. J. Samelson(4)

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

Abstract. Object invariants define the consistency of objects. They
serve to inhibit premature termination of call backs, enable object invariants and
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’s 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@MPCS04, Husing&Kuiper@FASE00, Leaves&Mueller@TCSF07, Middlecoop&Huizing&Kuiper&Luit@SMRBP7, Poetzsch-HeffterHabil97, Mueller&Petzsch-Heffter&Leavens@SCP06, Lu&Potter@ECOOP07.

- 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

```java
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

```java
void withdraw(int amount)
{
    ...

    this.balance -= amount;
    poss break V

    ...
    this.card.pay(amount);
    poss break V
    poss break V
    poss break V
    poss break V
    poss break V
    ...
}
```
• Invariants
  • Expected at the start of a method: \( X \)
  • Vulnerable to a method: \( V \)

```java
void withdraw(int amount)
{
  ...
  poss break V
  this.balance -= amount;
  poss break V
  ...
  poss break V
  this.card.pay(amount);
  poss break V
  ..
}
```

• Proof obligations
  • Before a method call: \( B \)
  • At the end of a method body: \( E \)

```java
void withdraw(int amount)
{
  ...
  poss break V
  this.balance -= amount;
  poss break V
  ...
  poss break V
  this.card.pay(amount);
  poss break V
  ..
}
```

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

```java
void withdraw(int amount)
{
  ...
  check U
  this.balance -= amount;
  check U
  ...
  check C; prove B'
  this.card.pay(amount);
  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

\[ r \subseteq C_{c,m,c',m'} \Rightarrow (r \triangleright X_{c',m'}) \setminus (X_{c,m} \setminus V_{c,m}) \subseteq B_{c,m,r} \]

If method \( m \) may legally call method \( m' \), then any invariants expected by \( m' \)

Soundness Conditions - 1a

\[ r \subseteq C_{c,m,c',m'} \Rightarrow (r \triangleright X_{c',m'}) \setminus (X_{c,m} \setminus V_{c,m}) \subseteq B_{c,m,r} \]

If method \( m \) may legally call method \( m' \)

Soundness Conditions - 1b

\[ r \subseteq C_{c,m,c',m'} \Rightarrow (r \triangleright X_{c',m'}) \setminus (X_{c,m} \setminus V_{c,m}) \subseteq B_{c,m,r} \]

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

Soundness Conditions - 1c

\[ r \subseteq C_{c,m,c',m'} \Rightarrow (r \triangleright X_{c',m'}) \setminus (X_{c,m} \setminus V_{c,m}) \subseteq B_{c,m,r} \]
Soundness Conditions - 1d

\[
r \subseteq C_{c,m,c',m'} \Rightarrow (r \cup X_{c',m'})(X_{c,m} \setminus V_{c,m}) \subseteq B_{c,m,r}
\]

*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**

\[
r \subseteq C_{c,m,c',m'} \Rightarrow (r \cup X_{c',m'})(X_{c,m} \setminus V_{c,m}) \subseteq B_{c,m,r}
\]

\[
V_{c,m} \cap X_{c,m} \subseteq E_{c,m}
\]

\[
C_{c,m,c',m'} \cup (V_{c',m'} \setminus E_{c',m'}) \subseteq V_{c,m}
\]

\[
U_{c,m,c'} \cup D_{c'} \subseteq V_{c,m}
\]

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

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 hierarchical/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?