

Adopting Ownership Types

Jan Vitek

joint work with

James Noble, Jens Palsberg, Alex Pontanin, Tian Zhao

PURDUE
UNIVERSITY

(ζ^3)

“Types errors are tiny fraction of programming errors

they are easy to fix

Static Type systems reduce programmer productivity”

- *anonymous Smalltalk programmer*

Ownership types are just as bad as other static types...
except they are really worse...
because runtime errors are more subtle...
the practical benefits are thus unclear.

can we sell ownership to practitioners?

Motivation

- Historically “ownership type systems” have been studied in the context of statically-typed object-oriented programming languages
 - Noble, Vitek & Potter, Flexible alias protection. ECOOP 1998
 - Inspired by, e.g., Hogg’s Islands and Lea ea Aliasing; but many other important precursors
- We felt that the combination of (1) mutable state, (2) reference semantics, and (3) polymorphism was conducive to software faults
- Hard to reason about a “sea of objects”
- Traditional type systems were not sufficient

All true. But still no (widely) used ownership type system.

- Research agenda:
 - Find applications of ownership and push for adoption
 - Explore the tradeoffs between expressiveness
 - Tool support: inference & refactoring

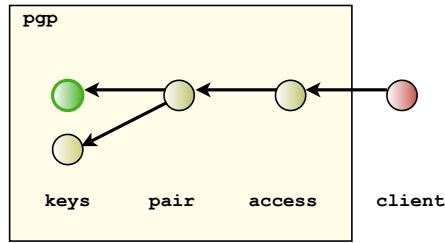
Three Applications

1. Confined Types
2. Place Types
3. Scope Types

Confined Types

[OOPSLA99/SPE01, OOPSLA01, OOPSLA03/JFP05]

Confinement



```
package pgp

public class KeyPair
    private Priv s
    public Pub k

public class Pub

@conf class Priv

public class Access
    public KeyPair getKey_()
        return map.get_()
```

What is confinement?

A machine-checkable programming discipline aimed at preventing the accidental spread of references across module boundaries

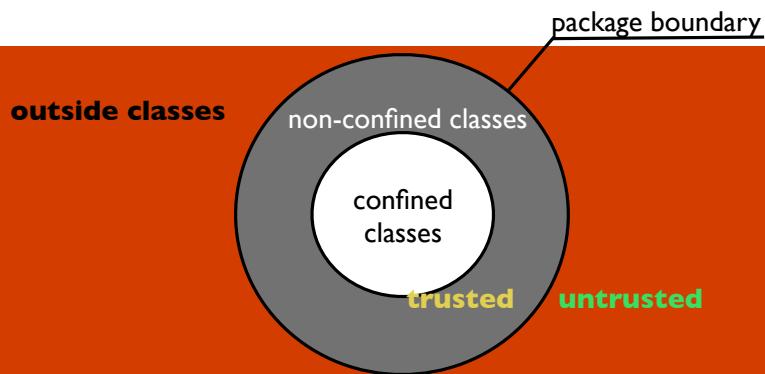
What are applications?

software engineering,
software assurance,

from [BokowskiVitek99]

Confinement for Security

- Confinement can be used to draw a boundary around a security-sensitive software module
- The benefit is that classes that interface with untrusted code can be clearly identified and access control checks can be inserted at that interface
- Very simple, modular & backwards compatible



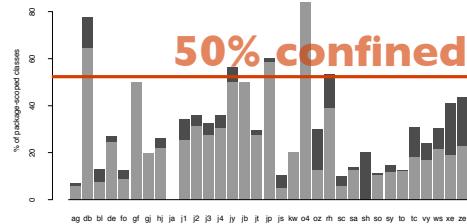
Confinement

Confinement codified for one particular choice of granularity and programming language (Java/package level) in [BV99]

Confinement annotations can be checked by a package-level confinement checker

Confinement annotations can be inferred by fast (10MB/sec) static program analysis

Results from [GPV01] on off-the-shelf libraries



Key Design Choices

- Simplicity
 - CT add: 1 class annotation and 1 method annotation
- Modular checking
 - Whole program verification is not acceptable
 - Only check code that uses CTs
- Backward compatibility
 - Existing code must verify
- No virtual machine support
 - no dynamic information
 - must run on stock virtual machine
- No changes to tool chain
 - e.g. Eclipse, javac, ant, ...
- These choices affect the design of the calculus and the formal treatment, the most significant one is the modularity requirement

Seven commandments

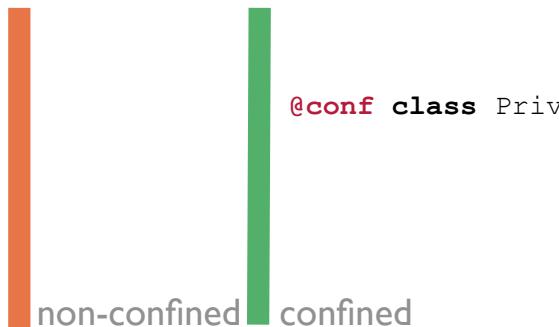
C_1	Confined types cannot be public
C_2	Public (protected) fields and method return types may not be confined.
C_3	Confined types cannot be widened to a non-confined type
C_4	Inherited methods invoked on a confined type must be anonymous or defined in a confined class
C_5	Subtypes of a confined type must be confined
C_6	Overriding must preserve anonymity

A_1	<code>this</code> can only be used to select fields and to invoke anonymous methods
-------	--

C_1	Confined types cannot be public
-------	---------------------------------

... `new Priv()` ...

client code



- the scope of confined classes is limited to their defining package.
Use the Java type system to prevent clients from referring to confined types without having to change the typechecker or bytecode verifier.
(backward compatibility, modularity)

C2

Public (protected) fields and method return types may not be confined.

```
Object obj =           public class Access      @conf class Priv  
    new Access().val;    public Priv val
```

client code

non-confined

confined

- prevents client code from widening confined types to unrestricted types (modularity)

C3

Confined types cannot be widened to a non-confined type

```
Object obj =           public class Access      @conf class Priv  
    new Access().get();   public Object get(){  
        return  
            (Object) new Priv();
```

client code

non-confined

confined

- prevents trusted code from widening confined types to unrestricted types

C4

Inherited methods invoked on a confined type must be anonymous or defined in a confined class

```
public class Bad {      public class Access      @conf class Priv
    public Object get(){    public Object get(){    extends Bad
        return this; }      return priv.get()
    new Access().get(); }
```

client code

non-confined

confined

- prevents client code from widening confined this reference
Averts requiring verification of parent classes
(modularity)

C5

Subtypes of a confined type must be confined

```
np = new NotPriv();  public class Access      @conf class Priv
access.set(np);    private Priv priv;
                    public void
                    get(NotPriv p){  extends Bad
                        priv = notpriv;
                    }

```

client code

non-confined

confined

- enforces “expectation of privacy” in trusted code
i.e. a variable of confined type is expected to **not** be accessible from client code

A1

`this` can only be used to select fields and
to invoke anonymous methods

```
public class Bad {  
    @anon Object get1(){  
        return this; }  
  
    @anon Object get2(){  
        return this.get(); }  
  
client code
```

```
@conf class Priv  
extends Bad
```

non-confined

confined

- allows confined types to use some inherited methods

C6

Overriding must preserve anonymity

```
public class Worse {  
    @anon Object get1(){  
  
public class Bad  
    extends Worse {  
    Object get1(){  
        return this;  
  
client code
```

```
@conf class Priv  
extends Bad
```

non-confined

confined

- allows safe use of inherited methods

Confined Featherweight Java

C ::= p.q

L ::= [conf] class C < D { C f; K M }

K ::= C(C f) { super(f); this.f = f; }

M ::= [anon] C m(C x) { return e; }

e ::= x | e.f | e.m(e) | (C) e | new C(e)

v ::= new C(v)

Observations

- Featherweight Java does not have assignment.
- Confinement does not differentiate between object instances.
- We only need to track types, for this FJ is adequate
- We use a call-by-value semantics. Every expression is fully evaluated in its defining context.
- An alternative has been studied by Grossman et al. and Aldrich, but appears more cumbersome.

Confined Featherweight Java

Evaluation:

$$\frac{e = \text{new } C(\bar{v}).f_i \quad fields(C) = (\bar{D} \bar{f})}{P . v \in E[e] \rightarrow P . v \in E[v_i]} \quad (\text{R-FIELD})$$

$$\frac{e = (C') \text{ new } C(\bar{v}) \quad C <: C'}{P . v \in E[e] \rightarrow P . v \in E[\text{new } C(\bar{v})]} \quad (\text{R-CAST})$$

$$\frac{e = v'.m'(\bar{v}) \quad v' = \text{new } C(\bar{u}) \quad mbody(m', C) = (\bar{x}, e_0)}{P . v \in E[e] \rightarrow P . v \in E[e] . v' \in [v'/\bar{x}, v'/\text{this}]e_0} \quad (\text{R-INVK})$$

$$\frac{e = v'.m'(\bar{v})}{P . v \in E[e] . v' \in E[v''] \rightarrow P . v \in E[v'']} \quad (\text{R-RET})$$

Evaluation contexts:

$$E[\circ] ::= \circ \mid (C) E[\circ] \mid E[\circ].f_i \mid E[\circ].m(\bar{e}) \mid v.m(\bar{v}, E[\circ], \bar{e}) \mid \text{new } C(\bar{v}, E[\circ], \bar{e})$$



Dagstuhl, June 2005



Type rules

Expression typing:

$$\Gamma \vdash x : \Gamma(x) \quad (\text{T-VAR})$$

$$\frac{\Gamma \vdash e : C \quad fields(C) = (\bar{D} \bar{f})}{\Gamma \vdash e.f_i : C_i} \quad (\text{T-FIELD})$$

$$\frac{\Gamma \vdash e : C_0 \quad \Gamma \vdash \bar{e} : \bar{C} \quad mtype(m, C_0) = \bar{D} \rightarrow C \quad \bar{C} \preceq \bar{D} \quad mdef(m, C_0) = D_0 \quad (C_0 \preceq D_0 \vee \text{anon}(m, D_0))}{\Gamma \vdash e.m(\bar{e}) : C} \quad (\text{T-INVK})$$

$$\frac{fields(C) = (\bar{D} \bar{f}) \quad \Gamma \vdash \bar{e} : \bar{C} \quad \bar{C} \preceq \bar{D}}{\Gamma \vdash \text{new } C(\bar{e}) : C} \quad (\text{T-NEW})$$

$$\frac{\Gamma \vdash e : D \quad D \preceq C}{\Gamma \vdash (C)e : C} \quad (\text{T-UCAST})$$



Dagstuhl, June 2005



Visibility

Confined types, type visibility, and safe subtyping:

$$\frac{CT(C) = \text{conf class } C \triangleleft D \{ \dots \}}{\text{conf}(C)}$$

$$\frac{\neg \text{conf}(C)}{\text{visible}(C, D)} \quad \frac{\text{packof}(C) = \text{packof}(D)}{\text{visible}(C, D)}$$

$$\frac{C <: D \quad \text{conf}(C) \Rightarrow \text{conf}(D)}{C \preceq D}$$

Static expression visibility:

$$\frac{\text{visible}(\Gamma(x), C) \quad \Gamma \vdash e.f_i : C' \quad \text{visible}(C', C) \quad \Gamma \vdash \text{visible}(e, C)}{\Gamma \vdash \text{visible}(x, C)}$$

$$\frac{\text{visible}(C', C) \quad \Gamma \vdash \text{visible}(e, C) \quad \text{visible}(C', C) \quad \forall i, \Gamma \vdash \text{visible}(e_i, C)}{\Gamma \vdash \text{visible}((C') e, C)}$$

$$\frac{\Gamma \vdash e.m(\bar{e}) : C' \quad \text{visible}(C', C) \quad \Gamma \vdash \text{visible}(e, C) \quad \forall i, \Gamma \vdash \text{visible}(e_i, C)}{\Gamma \vdash \text{visible}(e.m(\bar{e}), C)}$$

Anonymity

Anonymous method:

$$\frac{mdef(m, C_0) = C'_0 \quad \text{anon } C \ m (\overline{C \ x}) \ \{ \dots \} \in \text{methods}(C'_0)}{\text{anon}(m, C_0)}$$

Anonymity constraints:

$$\frac{\text{anon}(e, C)}{\text{anon}((C') e, C)} \quad \frac{\text{anon}(\bar{e}, C)}{\text{anon}(\text{new } C'(\bar{e}), C)} \quad \frac{x \neq \text{this}}{\text{anon}(x, C)}$$

$$\frac{\text{anon}(e, C)}{\text{anon}(e.f, C)} \quad \frac{\text{anon}(e, C) \quad \text{anon}(\bar{e}, C)}{\text{anon}(e.m(\bar{e}), C)}$$

$$\frac{}{\text{anon}(\text{this}.f, C)} \quad \frac{\text{anon}(m, C) \quad \text{anon}(\bar{e}, C)}{\text{anon}(\text{this}.m(\bar{e}), C)}$$

Type rules

Method typing:

$$\frac{\bar{x} : \bar{C}, \text{this} : C_0 \vdash e : D \quad D \preceq C \quad \text{override}(m, C_0, D_0)}{\bar{x} : \bar{C}, \text{this} : C_0 \vdash \text{visible}(e, C_0) \quad (\text{anon}(m, C_0) \Rightarrow \text{anon}(e, C_0))} \quad [\text{anon}] \quad \text{C } m(\bar{C} \bar{x}) \{ \text{return } e; \} \text{ OK IN } C_0 \triangleleft D_0 \quad (\text{T-METHOD})$$

Class typing:

$$\frac{\text{fields}(D) = (\bar{D} \bar{g}) \quad K = C(\bar{D} \bar{g}, \bar{C} \bar{f}) \{ \text{super}(\bar{g}); \text{this.}\bar{f} = \bar{f}; \} \quad \text{visible}(D, C) \quad (\text{conf}(D) \Rightarrow \text{conf}(C)) \quad \bar{M} \text{ OK IN } C \triangleleft D}{[\text{conf}] \text{ class } C \triangleleft D \{ \bar{C} \bar{f}; K \bar{M} \} \text{ OK}} \quad (\text{T-CLASS})$$

Confinement

Theorem 2 (Confinement)

If P is well-typed and satisfies confinement, and $P \rightarrow^* P'$ then P' satisfies confinement.

Definition 2 (Confinement Satisfaction)

A program $P = v_1 \ m_1 \ e_1 \dots v_n \ m_n \ e_n$ satisfies confinement iff for all $i \in [1, n]$ we have $\text{visible}_C(e_i, C')$, where $v_i = \text{new } C(\bar{v})$, $mdef(m_i, C) = C'$.

$$\frac{\emptyset \vdash e.f_i : C' \quad \text{visible}(C', C) \quad (e = \text{new } C_0(\bar{u}) \vee \text{visible}_{C_0}(e, C))}{\text{visible}_{C_0}(e.f_i, C)}$$

$$\frac{\text{visible}(C', C) \quad \text{visible}_{C_0}(e, C) \quad \text{visible}(C', C) \quad \forall i, \text{visible}_{C_0}(e_i, C)}{\text{visible}_{C_0}((C') e, C) \quad \text{visible}_{C_0}(\text{new } C'(\bar{v} \bar{e}), C)}$$

$$\frac{\emptyset \vdash e.m(\bar{e}) : C' \quad \text{visible}(C', C) \quad (e = \text{new } C_0(\bar{u}) \vee \text{visible}_{C_0}(e, C)) \quad \forall i, \emptyset \vdash \text{visible}(e_i, C)}{\text{visible}_{C_0}(e.m(\bar{e}), C)}$$

Generics

Collection classes, e.g. **List**, breach confinement due to widening of contents to **Object**

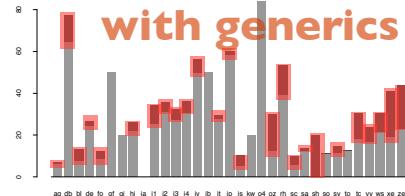
Add support for a Confinable annotation in type variable (or in plain Java)

`class List <X implements Confined>`

```
public class A.Pair<X,Y:>Confined>
    public X x
    public Y y
```

```
public class A.Client
    public Pair<Client,Client> p
```

```
public class C.Priv <: Confined
    public Pair<Client, Priv>
```



Generic Commandments

C7	A variable of a generic type cannot be widened to a type containing a different set of confined type variables
C8	A method invoked on an expression of type T must either be defined in a type with the same set of type variables as that in T or be an anonymous method.

C7

A variable of a generic type cannot be widened to a type containing a different set of confined type variables

```
public class List<C>
```

```
Object o = access.val    public class Access{  
                           public Object val =  
                               (Object) list;  
                           }  
                           @conf class Priv  
                           List<Priv> list
```

client code

non-confined

confined



Dagstuhl, June 2005

(§³)

C7

A method invoked on an expression of type T must either be defined in a type with the same set of type variables as that in T or be an anonymous method.

```
public class MyList<C> extends List
```

```
public class List {  
    List head() { return this; }
```

```
List l = access.val    public class Access{  
                           public List val =  
                               list.head;  
                           }  
                           @conf class Priv  
                           MyList<Priv> list
```

client code

non-confined

confined



Dagstuhl, June 2005

(§³)

```

class Naive<X extends A>{
    X x;
    Object reveal() {
        return x.m();
    }

class A {
    anon Object m() {
        return new Object();
    }

class B extends A {
    Object m() {
        return this;
    }
}

```

client code | non-confined | confined

@conf class Priv
extends B
Naive<Priv> naif

Confinement markers

How to use the same confined class in different contexts?

```
public class A.PrivKey<X>
    ...

```

```
conf public class B.Marker
```

```
public class B.Client
    private PrivKey<Marker> pk
```

```
conf class C.Marker
```

```
public class C.ClientToo
    private PrivKey<Marker> pk
```

Conclusions

- Confined types based on an embarrassingly simple set of rules; thus easy to understand and (hopefully) easy to use
- Confinement rules have been proven sound
- A whole program inference tool (Kacheck) available
- Confinement extends straightforwardly to generics



Place Types

X10

- X10 is a new parallel programming language
 - developed within the DARPA funded PERCS program at IBM
(with some University collaborators, Purdue ...)
- Increase programmer productivity
 - targets scientific domains, not general purpose (not Java)
 - based on state-of-the-art O.O. language (ie. Java) to decrease entry costs
 - support common parallel programming idioms (data, control, divide-conquer...)
- Deliver scalable performance
 - large degrees of parallelism
 - large uniformities in data access



(based on slides by Vijay Saraswat)

Places

- Places abstract non-uniform memory
 - transparency hides latencies and results an opaque performance model
- Places consist of
 - data - updateable memory locations
 - activities - executing code that updates data
- Global address space
 - place named by datum that resides there
- Atomicity
 - multiple mutable datum can be accessed atomically only if there are co-located
- Async
 - `async (D) {S}`
 - runs S at place D in parallel with other activities at that place
 - S may contain atomic operations at that location or asyncs to other places

Place Types

- Variant of ownership types

Goal is to identify statically co-located data items, this allows the compiler to ensure that remote operations are always asynced

No provision for encapsulation

- Place-qualified types

`String@Home name;`

Place is a type qualifier

Qualifiers must agree

`String@Q other = name;`

- Place parameters

`<place P> void process(String@P n)`

like generics: `class Holder<place P> { Object@P value; }`

- here

current place

Thread Locals

- Values that can only be used within the current thread are annotated

`List@thread ls = new List@thread();`

can only be used in method scope (illegal `class X{Obj@thread o;}`)

can be passed as a parameter:

`class Y <place P> { Object@P o; ...`

`...`

`void m() { Y@thread y = new Y@thread(); ...`

Relaxing the Type System

- anywhere types

```
Holder@? value;  
we don't know where value is  
  
Holder@?[] arr  
heterogeneous collections
```

```
H@P x = ( H@P ) y;  
cast y to type H at P  
  
if ( x instanceof Place@P )  
check that x is at P  
  
locate H@P x = y  
introduces the place name P
```

Conclusions

- Place types have a different purpose than ownership types
- They control locality and do not enforce encapsulation, and has dynamic information
- They are crucial to the programming model of X10, and are thus going to be as widely used as X10

Scoped Types

PURDUE
UNIVERSITY

(Σ^3)

Scoped Types

- A variant of ownership types.
- Related to region-types and separation logics.
- Formalized in a simple extension of Featherweight Java with threads & state.

● see also:

Flexible alias protection. Noble, Vitek, Potter [ECOOP'98]

Scoped Types for Real-time Java. Zhao, Noble, Vitek [RTSS04]

Ownership Types for Safe Region-Based Memory Management in Real-Time Java, Boyapati, Salcianu, Beebee, Rinard [PLDI03]

PURDUE
UNIVERSITY

Dagstuhl, June 2005

(Σ^3)

RealTime Specification for Java

Object lifetime controlled by reachability, when last thread leaves area, all objects are reclaimed

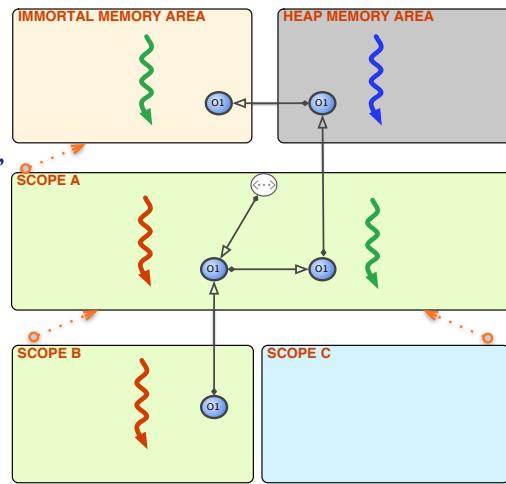
Eliminate GC latency; allow temporary objects

NoHeapRealtimeThreads don't read from heap, protected from GC interference

Allocation linear in size, deallocation $O(1)$ (modulo finalizers)

Multiple threads communicate through portal

Nesting is dynamic, established by entry order; can change for the same scope object



Dagstuhl, June 2005

(S³)

PURDUE
UNIVERSITY



PRISMj

Mission critical avionics DRE

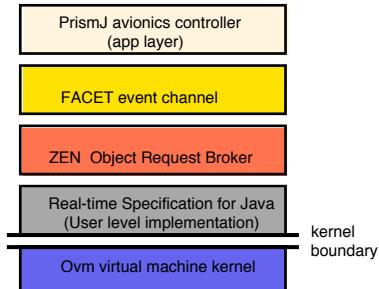
Boeing, Purdue, UCI, WUSTL

Route computation, Threat deconfliction algorithms

ScanEagle UAV



System	K LOCs
PRISMj	109K
FACET EVENT CHANNEL	15K
ZEN CORBA ORB	179K
RTSJ LIBRARIES	60K
CLASSPATH LIBRARIES	500K
OVM VIRTUAL MACHINE	220K



3 rate groups (20, 5, 1Hz)
performance 2x jTime,
 \approx Sun product VM



Embedded Planet PowerPC 8260
Core at 300 MHz
256 Mb SDRAM
32 Mb FLASH
PC/104 mechanical sized
Embedded Linux

PURDUE
UNIVERSITY

Dagstuhl, June 2005

(S³)

Scoped Types programming model

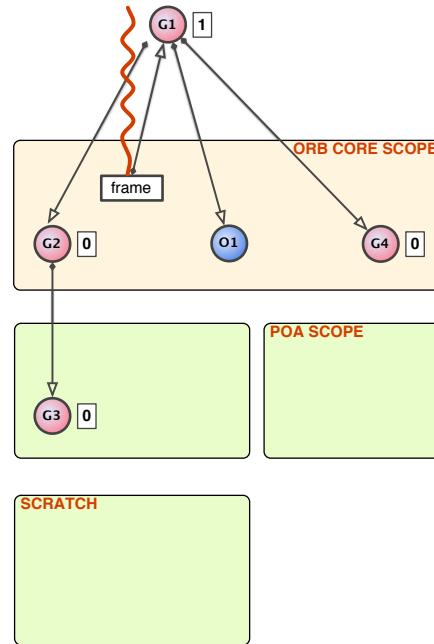
(set of) scopes	≡	Java package
nested scope	≡	nested package
ScopedMemory object	≡	ScopedGate object
enter()	≡	method invocation
IllegalAssignment	≡	compile-time error
scope cycle error	≡	compile-time error

Validity constraints for ST programs

C1	A scoped type is visible only to classes in the same package or subpackages.
C2	A scoped type can only be widened to other scoped types in the same package.
C3	The methods invoked on a scoped type must be defined in the same package.
S1	A gate type is only visible to the classes in the immediate super-package.
S2	A gate type cannot be widened to other types.
S3	The methods invoked on a gate type must be defined in the same class.

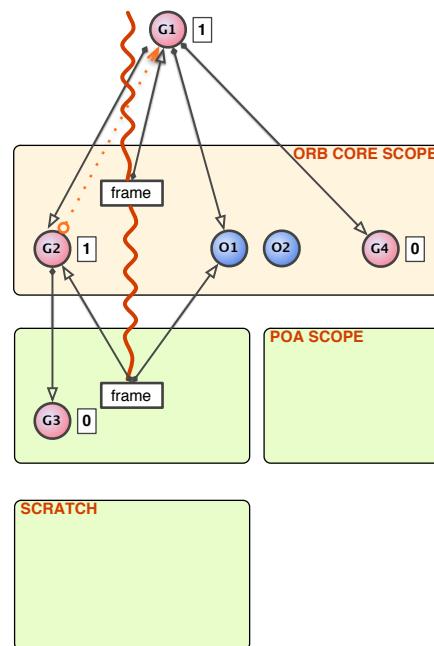
Scoped Program

- Every dynamic scope is implemented by a package in the source represented by a gate at runtime
- Several scope of the same kind can be instantiated
- Gates are normal Java objects with fields pointing into the scope



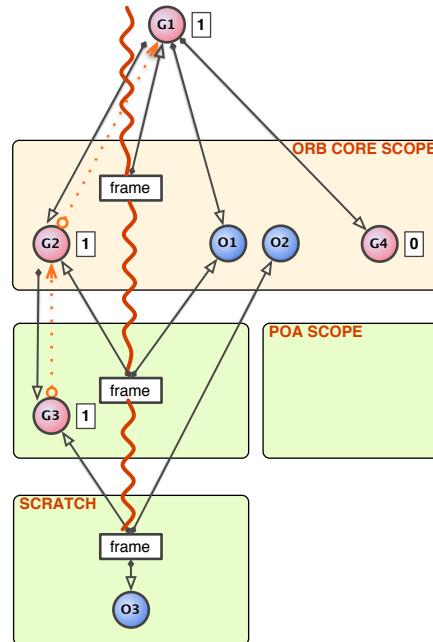
Scoped Program

- Every call to a gate switches the allocation context
- scoped classes can refer to objects in the parent package
- Gates have an associated reference count



Scoped Program

- Scopes are cleared by calling `reset()` on a gate with $RC=0$.
- Code duplication may arise if the same class must be used in different scopes.



Scoped Java Calculus

```

L ::= o class P.C < D {  $\bar{C}$   $\bar{f}$ ; K  $\bar{M}$  }

K ::= C() {super(); this. $\bar{f}$  := new D(); }

M ::= C m( $\bar{C}$   $\bar{x}$ ) { return e; }

e ::= x | e.f | e.m( $\bar{e}$ ) | new C() | e.f := e
     | spawn e | reset e | v

o ::= gate | scoped           v ::=  $\ell$            P ::= p | p.P
  
```

Fig. 7. Syntax of the SJ calculus.

The Scoped Type System

$$\Gamma, \Sigma \vdash x : \Gamma(x)$$

$$\Gamma, \Sigma \vdash \ell : \Sigma(\ell)$$

$$\frac{\Gamma, \Sigma \vdash e_0 : C \quad fields(C) = (\bar{C} \bar{f})}{\Gamma, \Sigma \vdash e_0.f_i : C_i} \quad \frac{\Gamma, \Sigma \vdash e_0 : C_0 \quad fields(C_0) = (\bar{C} \bar{f}) \quad \Gamma, \Sigma \vdash e : C \quad C \preceq C_i}{\Gamma, \Sigma \vdash e_0.f_i = e : C_i}$$

$$\frac{\Gamma, \Sigma \vdash e_0 : C_0 \quad mdef(m, C_0) = C'_0 \quad mtype(m, C'_0) = \bar{C} \rightarrow C \quad \Gamma, \Sigma \vdash \bar{e} : \bar{D} \quad \bar{D} \preceq \bar{C} \quad C_0 \preceq C'_0}{\Gamma, \Sigma \vdash e_0.m(\bar{e}) : C} \quad \frac{\Gamma, \Sigma \vdash e : Thread}{\Gamma, \Sigma \vdash spawn e : Thread}$$

$$\frac{\Gamma, \Sigma \vdash e : C \quad C \text{ is a gate}}{\Gamma, \Sigma \vdash reset e : C}$$

$$\Gamma, \Sigma \vdash new C() : C$$

Correctness

- If it is the case that

$$P \equiv P'' | t[\dots] E [\mathbf{reset} l_o]]$$

σP is well typed

$$\sigma P \Rightarrow \sigma' P' \quad \text{where } P' \equiv P'' | t[\dots] E[l_o]]$$

- then

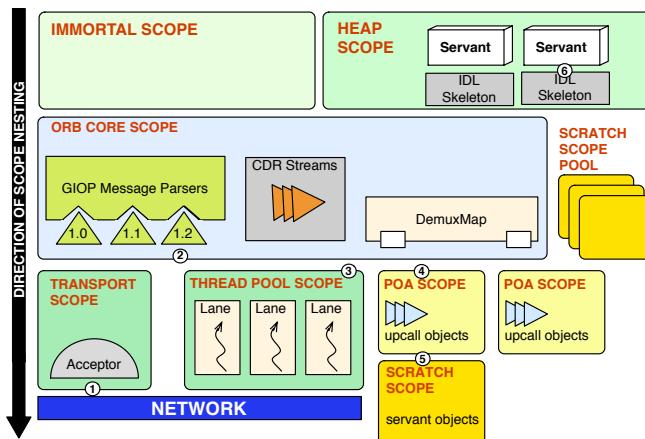
objects allocated in the scope represented by gate σl_o are not reachable in $\sigma' P'$

(i.e. no dangling pointers)

[Zhao, Noble, Vitek. RTSS'04]

Empiric Validation

- Zen is a RT-CORBA object request broker
 - ~ 100 KLoc written in RTSJ at UCI
 - with a rich (ie. complex) memory scope structure
 - scopes protect the ORB core from interference
 - use RTSJ design patterns
Bridge, Wedge Thread,
ScopedRunLoop, EIR
- Real-Time Java scoped memory:
design patterns and semantics.
Pizlo, Fox, Holmes, Vitek.
[ISORC04]*



Refactoring ZEN

Successfully refactored Zen

Eliminated ~30 classes out of 186,
little code duplication

Software structure became easier to
understand

Several bugs were discovered

Faster execution times

zen.orb	38	zen.orb	16
zen.orb.any	2	—	
zen.orb.any.monolithic	1	—	
zen.orb.dynany	11	—	
zen.orb.giop	6	zen.org.giop	4
zen.orb.giop.IOP	3	zen.orb.giop.IOP	3
zen.orb.giop.type	5	zen.orb.giop.type	5
zen.orb.giop.v1.0	9	zen.orb.giop.v1.0	9
zen.orb.giop.v1.1	5	zen.orb.giop.v1.1	5
zen.orb.giop.v1.2	4	zen.orb.giop.v1.2	4
zen.orb.policies	13	zen.orb.policies	9
zen.orb.resolvers	2	—	
zen.orb.transport	11	zen.orb.transport	3
zen.orb.transport.iip	4	zen.orb.transport.iip	1
zen.poa	16	zen.poa	3
zen.poa.mechanism	27	zen.poa.mechanism	19
zen.poa.policy	7	zen.poa.policy	7
zen.util	21	zen.util	11
		scope.orb	45
		scope.orb.connection	7
		scope.orb.requestprocessor	10
		scope.requestwaiter	3

... 2nd round of refactoring in progress

Borrowing

```
public class Parent {  
    void get(@borrow Priv p){  
        ... p.f ... p.m() ...
```

parent package

```
class Priv  
...  
parent.get( new Priv())
```

confined

- Some RTSJ idioms require controlled relaxation of ownership, aka John Boyland's borrowing

Conclusions

- A clear notion of failure with strong incentive to strive for correctness
- Scoped types offer a lightweight variant of object based protection
- Protection is not about encapsulation but about lifetimes
- Do not support dynamics a la X10
- Requires borrowing