Adopting Ownership Types

Jan Vitek

joint work with

James Noble, Jens Palsberg, Alex Pontanin, Tian Zhao

PURDUE
UNIVERSITY

"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

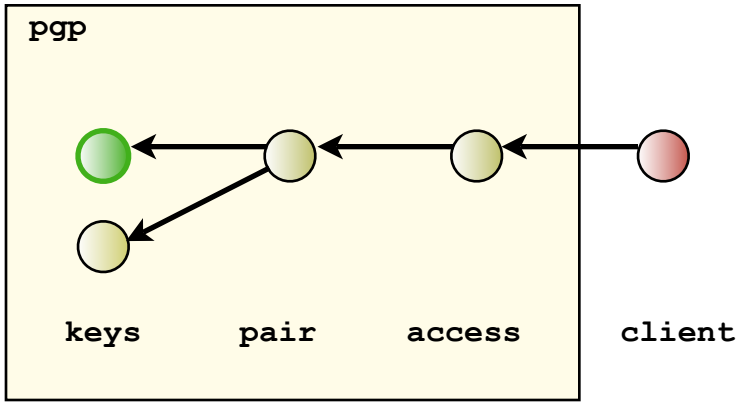
Three Applications

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

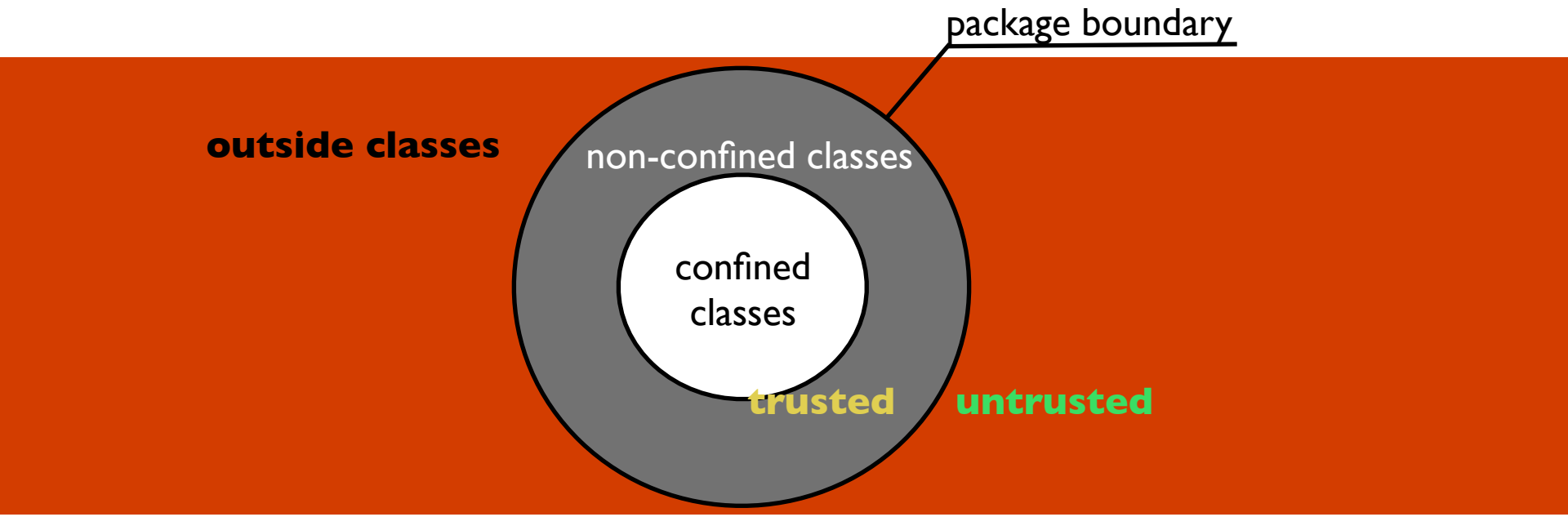
What are applications?

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

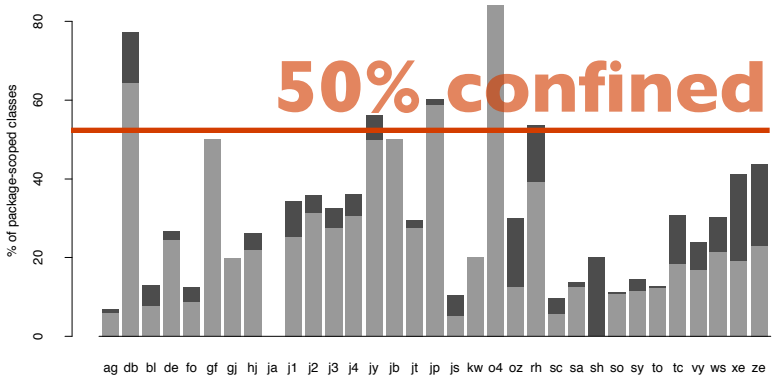


Confinement

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

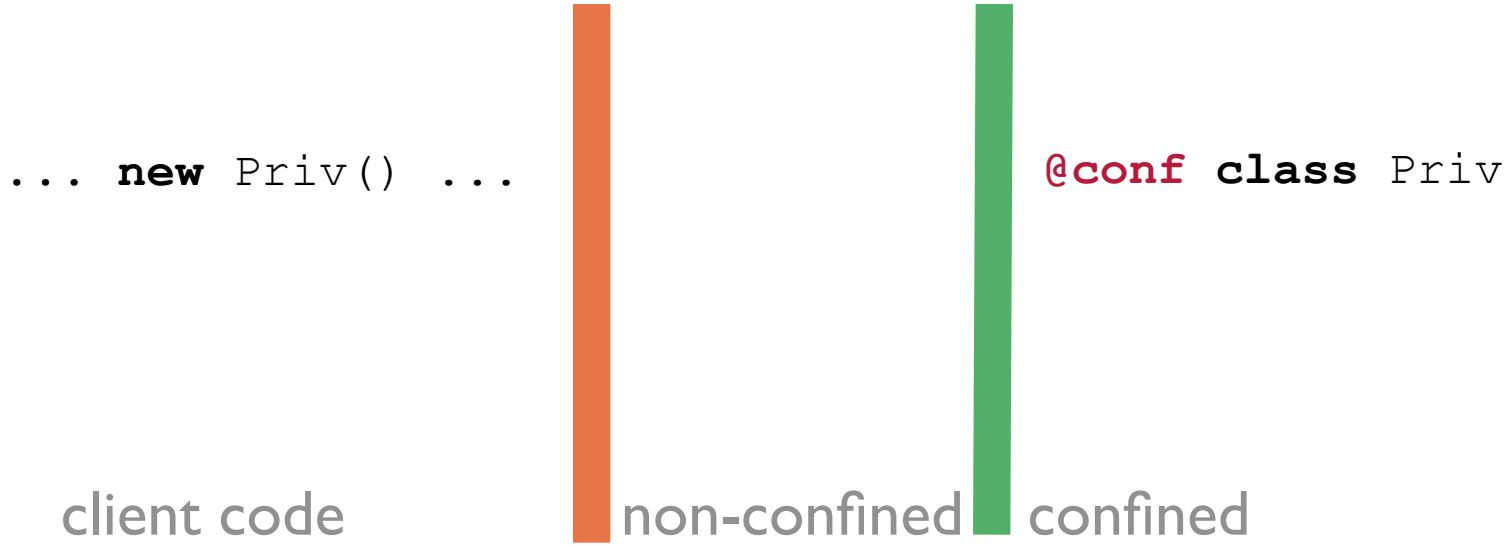
- Simplicity
  - CT add: 1 class annotation and 1 method annotation
- Modular checking
  - Whole program verification is not acceptable
  - Only check code that uses CTs
- Backward compatibility
  - Existing code must verify
- No virtual machine support
  - no dynamic information
  - must run on stock 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

C1	Confined types cannot be public
C2	Public (protected) fields and method return types may not be confined.
C3	Confined types cannot be widened to a non-confined type
C4	Inherited methods invoked on a confined type must be anonymous or defined in a confined class
C5	Subtypes of a confined type must be confined
C6	Overriding must preserve anonymity

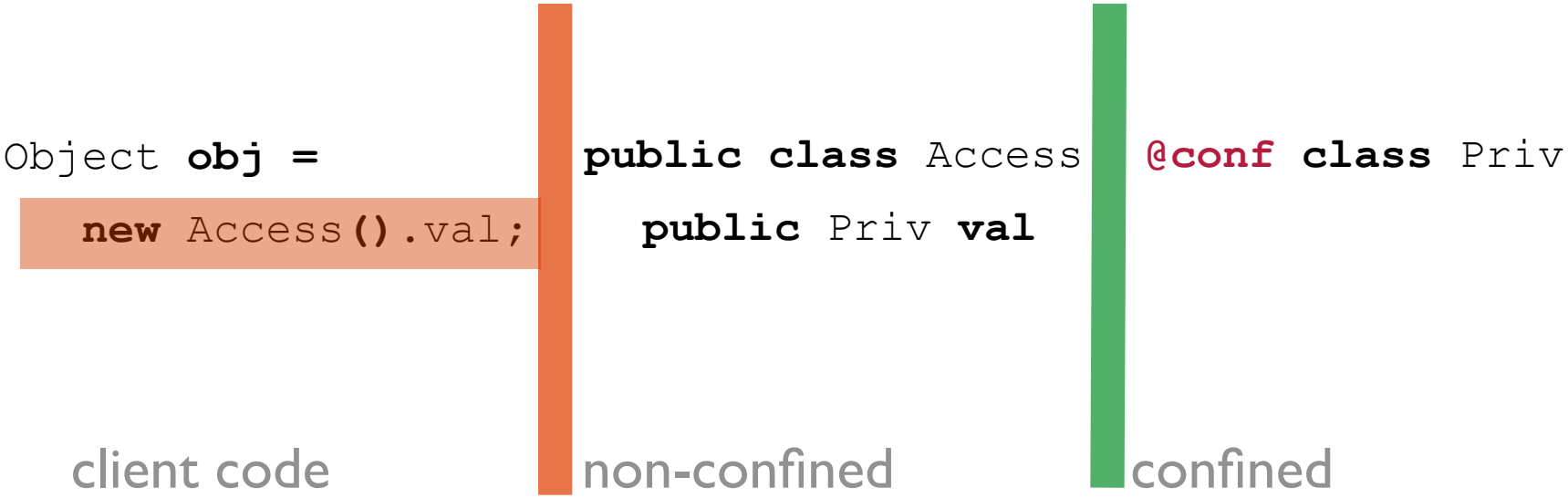
A1	**`this`** can only be used to select fields and to invoke anonymous methods

C1	Confined types cannot be public



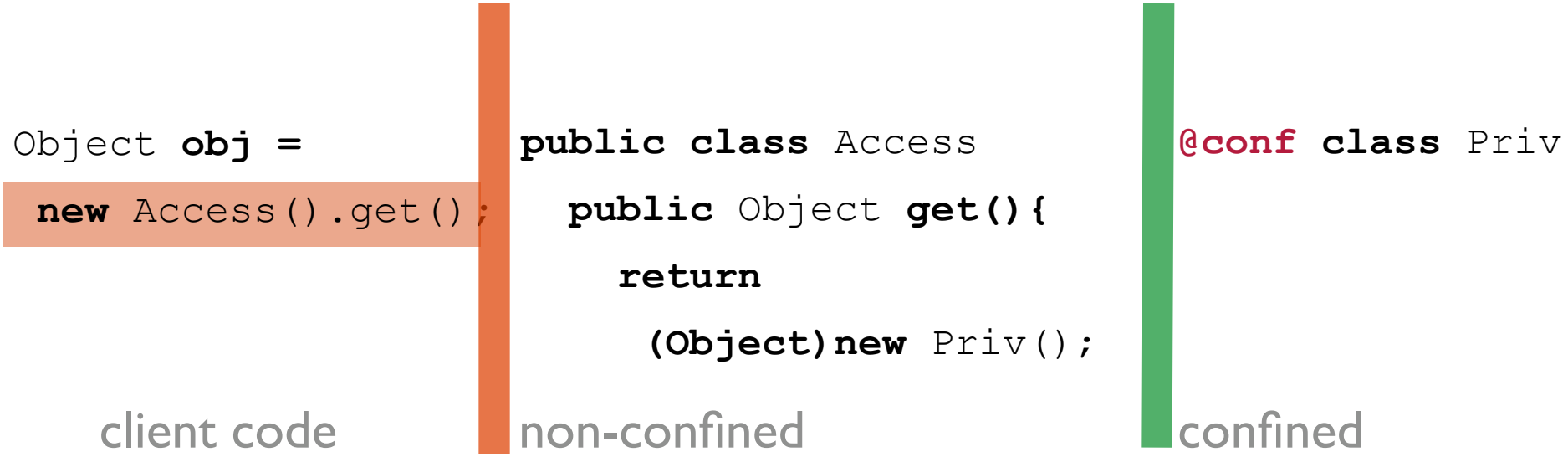
- the scope of confined classes is limited to their defining package.
  Use the Java type system to prevent clients from referring to confined types without having to change the typechecker or bytecode verifier.
  (backward compatibility, modularity)

C2	Public (protected) fields and method return types may not be confined.



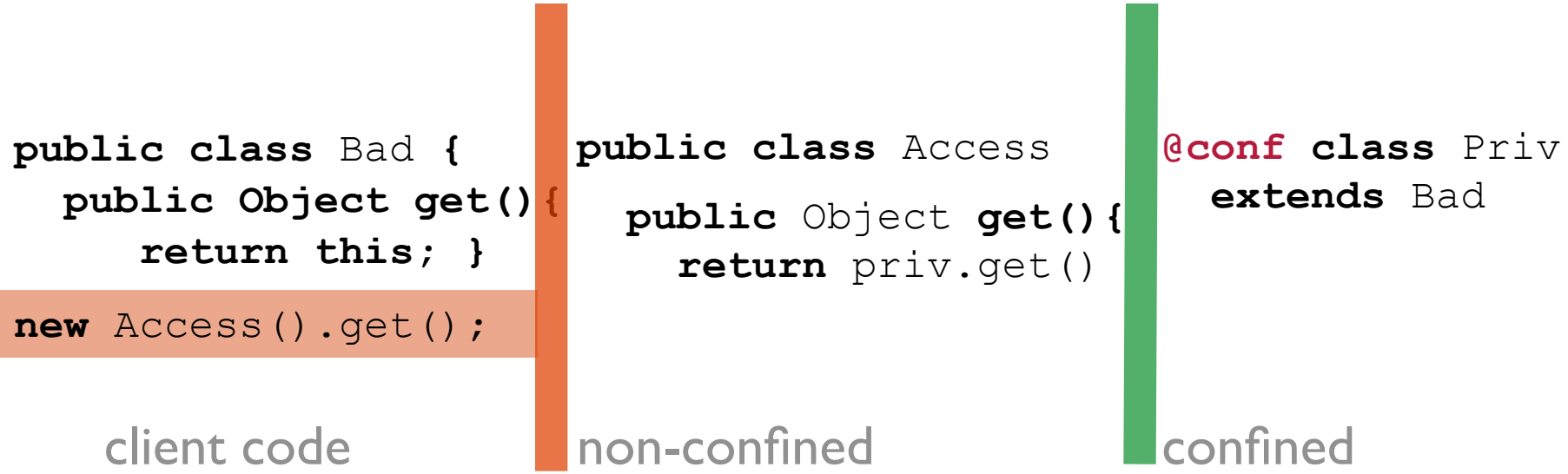
- prevents client code from widening confined types to unrestricted types (modularity)

C3	Confined types cannot be widened to a non-confined type



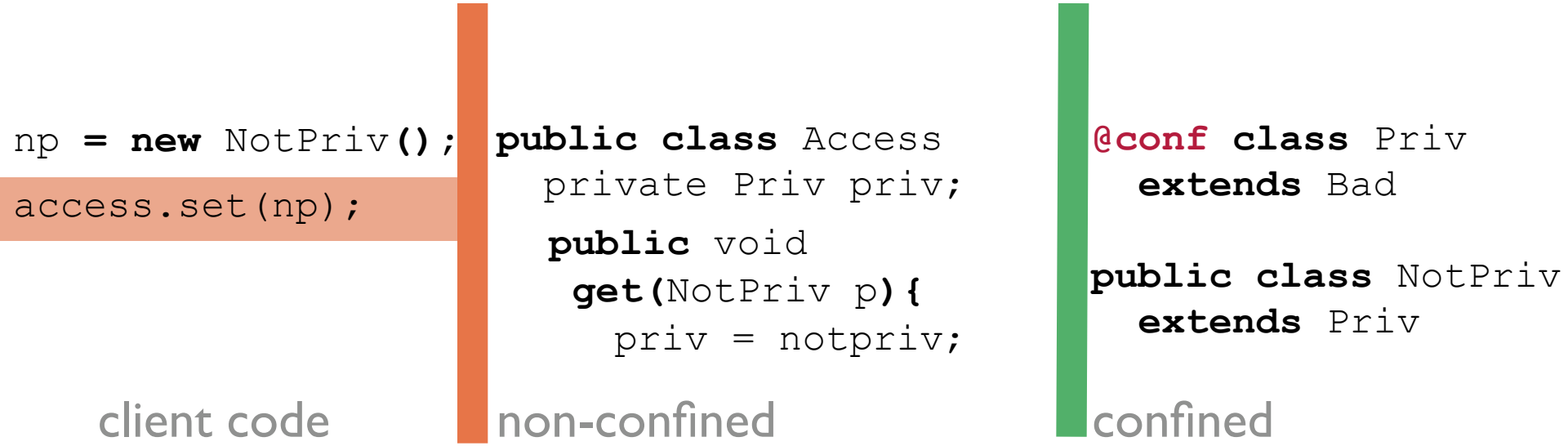
- prevents trusted code from widening confined types to unrestricted types

C4	Inherited methods invoked on a confined type must be anonymous or defined in a confined class



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

C5	Subtypes of a confined type must be confined



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

$\mathcal{A}1$	**`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

non-confined

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

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

non-confined

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

confined

- allows safe use of inherited methods

Confined Featherweight Java

$$
\begin{array}{lcl}
\texttt{C} & ::= & \texttt{p.q} \\
\texttt{L} & ::= & \texttt{[conf] class C} \lhd \texttt{D} \{ \overline{\texttt{C f}}; \texttt{K M} \} \\
\texttt{K} & ::= & \texttt{C}(\overline{\texttt{C f}}) \{ \texttt{super}(\overline{\texttt{f}}); \texttt{this.}\overline{\texttt{f}} = \overline{\texttt{f}}; \} \\
\texttt{M} & ::= & \texttt{[anon] C m}(\overline{\texttt{C x}}) \{ \texttt{return e}; \} \\
\texttt{e} & ::= & \texttt{x} \mid \texttt{e.f} \mid \texttt{e.m}(\overline{\texttt{e}}) \mid (\texttt{C})\ \texttt{e} \mid \texttt{new C}(\overline{\texttt{e}}) \\
\texttt{v} & ::= & \texttt{new C}(\overline{\texttt{v}})
\end{array}
$$

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:

$$\frac{\mathtt{e} = \mathtt{new\ C}(\overline{\mathtt{v}}).\mathtt{f_i} \quad \mathit{fields}(\mathtt{C}) = (\overline{\mathtt{D}}\ \overline{\mathtt{f}})}{P\ .\ \mathtt{v\ m}\ E[\mathtt{e}] \quad \rightarrow \quad P\ .\ \mathtt{v\ m}\ E[\mathtt{v}_i]} \qquad \text{(R-Field)}$$

$$\frac{\mathtt{e} = (\mathtt{C}')\ \mathtt{new\ C}(\overline{\mathtt{v}}) \quad \mathtt{C} <: \mathtt{C}'}{P\ .\ \mathtt{v\ m}\ E[\mathtt{e}] \quad \rightarrow \quad P\ .\ \mathtt{v\ m}\ E[\mathtt{new\ C}(\overline{\mathtt{v}})]} \qquad \text{(R-Cast)}$$

$$\frac{\mathtt{e} = \mathtt{v}'.\mathtt{m}'(\overline{\mathtt{v}}) \quad \mathtt{v}' = \mathtt{new\ C}(\overline{\mathtt{u}}) \quad \mathit{mbody}(\mathtt{m}',\ \mathtt{C}) = (\overline{\mathtt{x}},\ \mathtt{e}_0)}{P\ .\ \mathtt{v\ m}\ E[\mathtt{e}] \quad \rightarrow \quad P\ .\ \mathtt{v\ m}\ E[\mathtt{e}]\ .\ \mathtt{v}'\ \mathtt{m}'\ [\overline{\mathtt{v}}/\overline{\mathtt{x}},\ \mathtt{v}'/\mathtt{this}]\mathtt{e}_0} \qquad \text{(R-Invk)}$$

$$\frac{\mathtt{e} = \mathtt{v}'.\mathtt{m}'(\overline{\mathtt{v}})}{P\ .\ \mathtt{v\ m}\ E[\mathtt{e}]\ .\ \mathtt{v}'\ \mathtt{m}'\ \mathtt{v}'' \quad \rightarrow \quad P\ .\ \mathtt{v\ m}\ E[\mathtt{v}'']} \qquad \text{(R-Ret)}$$

Evaluation contexts:

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

Type rules

Expression typing:

$$\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \qquad \text{(T-Var)}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{C} \quad \mathit{fields}(\mathtt{C}) = (\overline{\mathtt{C}\ \mathtt{f}})}{\Gamma \vdash \mathtt{e.f_i} : \mathtt{C_i}} \qquad \text{(T-Field)}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{C_0} \quad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}} \quad \mathit{mtype}(\mathtt{m},\ \mathtt{C_0}) = \overline{\mathtt{D}} \to \mathtt{C} \quad \overline{\mathtt{C}} \preceq \overline{\mathtt{D}} \quad \mathit{mdef}(\mathtt{m},\ \mathtt{C_0}) = \mathtt{D_0} \quad (\mathtt{C_0} \preceq \mathtt{D_0} \vee \mathit{anon}(\mathtt{m}, \mathtt{D_0}))}{\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{C}} \qquad \text{(T-Invk)}$$

$$\frac{\mathit{fields}(\mathtt{C}) = (\overline{\mathtt{D}\ \mathtt{f}}) \quad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}} \quad \overline{\mathtt{C}} \preceq \overline{\mathtt{D}}}{\Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{e}}) : \mathtt{C}} \qquad \text{(T-New)}$$

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

Visibility

Confined types, type visibility, and safe subtyping:

$$\frac{CT(\mathtt{C}) = \mathtt{conf\ class\ C} \lhd \mathtt{D}\ \{\ldots\}}{conf(\mathtt{C})}$$

$$\frac{\neg conf(\mathtt{C})}{visible(\mathtt{C},\mathtt{D})} \qquad \frac{packof(\mathtt{C}) = packof(\mathtt{D})}{visible(\mathtt{C},\mathtt{D})}$$

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

Static expression visibility:

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

$$\frac{visible(\mathtt{C}',\mathtt{C}) \quad \Gamma \vdash visible(\mathtt{e},\mathtt{C})}{\Gamma \vdash visible((\mathtt{C}')\ \mathtt{e},\mathtt{C})} \qquad \frac{visible(\mathtt{C}',\mathtt{C}) \quad \forall \mathtt{i},\ \Gamma \vdash visible(\mathtt{e_i},\mathtt{C})}{\Gamma \vdash visible(\mathtt{new\ C}'(\overline{\mathtt{e}}),\mathtt{C})}$$

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

Anonymity

Anonymous method:

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

Anonymity constraints:

$$\frac{anon(\texttt{e}, \texttt{C})}{anon((\texttt{C}')\ \texttt{e}, \texttt{C})} \qquad \frac{anon(\overline{\texttt{e}}, \texttt{C})}{anon(\texttt{new C}'(\overline{\texttt{e}}), \texttt{C})} \qquad \frac{\texttt{x} \neq \texttt{this}}{anon(\texttt{x}, \texttt{C})}$$

$$\frac{anon(\texttt{e}, \texttt{C})}{anon(\texttt{e.f}, \texttt{C})} \qquad \frac{anon(\texttt{e}, \texttt{C}) \quad anon(\overline{\texttt{e}}, \texttt{C})}{anon(\texttt{e.m}(\overline{\texttt{e}}), \texttt{C})}$$

$$\frac{}{anon(\texttt{this.f}, \texttt{C})} \qquad \frac{anon(\texttt{m}, \texttt{C}) \quad anon(\overline{\texttt{e}}, \texttt{C})}{anon(\texttt{this.m}(\overline{\texttt{e}}), \texttt{C})}$$

Type rules

Method typing:

$$\frac{\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C}_0 \vdash \mathtt{e} : \mathtt{D} \quad \mathtt{D} \preceq \mathtt{C} \quad \mathit{override}(\mathtt{m}, \mathtt{C}_0, \mathtt{D}_0) \quad \overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C}_0 \vdash \mathit{visible}(\mathtt{e}, \mathtt{C}_0) \quad (\mathit{anon}(\mathtt{m}, \mathtt{C}_0) \Rightarrow \mathit{anon}(\mathtt{e}, \mathtt{C}_0))}{[\mathtt{anon}]\ \mathtt{C}\ \mathtt{m}(\overline{\mathtt{C}\ \mathtt{x}})\ \{\, \mathtt{return}\ \mathtt{e};\ \}\ \mathtt{OK\ IN}\ \mathtt{C}_0 \triangleleft \mathtt{D}_0} \quad \text{(T-Method)}$$

Class typing:

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

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 $P = \mathtt{v_1\ m_1\ e_1} \ldots \mathtt{v_n\ m_n\ e_n}$ satisfies confinement iff for all $\mathtt{i} \in [\mathtt{1}, \mathtt{n}]$ we have $visible_{\mathtt{C}}(\mathtt{e_i}, \mathtt{C}')$, where $\mathtt{v_i} = \mathtt{new\ C}(\overline{\mathtt{v}})$, $mdef(\mathtt{m_i},\ \mathtt{C}) = \mathtt{C}'$.

$$\frac{\emptyset \vdash \mathtt{e.f_i} : \mathtt{C}' \quad visible(\mathtt{C}', \mathtt{C}) \quad (\mathtt{e} = \mathtt{new\ C_0}(\overline{\mathtt{u}}) \ \vee \ visible_{\mathtt{C_0}}(\mathtt{e}, \mathtt{C}))}{visible_{\mathtt{C_0}}(\mathtt{e.f_i}, \mathtt{C})}$$

$$\frac{visible(\mathtt{C}', \mathtt{C}) \quad visible_{\mathtt{C_0}}(\mathtt{e}, \mathtt{C})}{visible_{\mathtt{C_0}}((\mathtt{C}')\ \mathtt{e}, \mathtt{C})} \qquad \frac{visible(\mathtt{C}', \mathtt{C}) \quad \forall \mathtt{i},\ visible_{\mathtt{C_0}}(\mathtt{e_i}, \mathtt{C})}{visible_{\mathtt{C_0}}(\mathtt{new\ C}'(\overline{\mathtt{v}}\,\overline{\mathtt{e}}), \mathtt{C})}$$

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

Generics

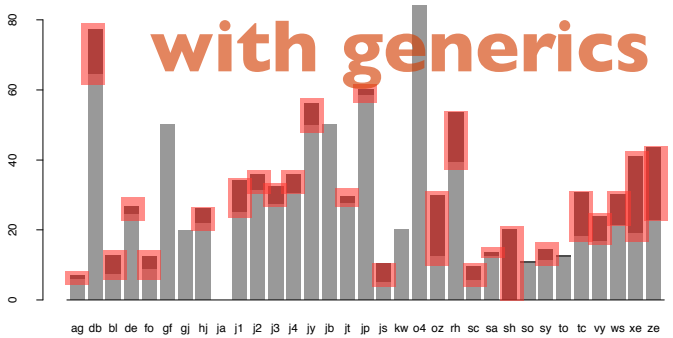
Collection classes, e.g. **`List`**, *breach confinement due to widening of contents to* **`Object`**

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

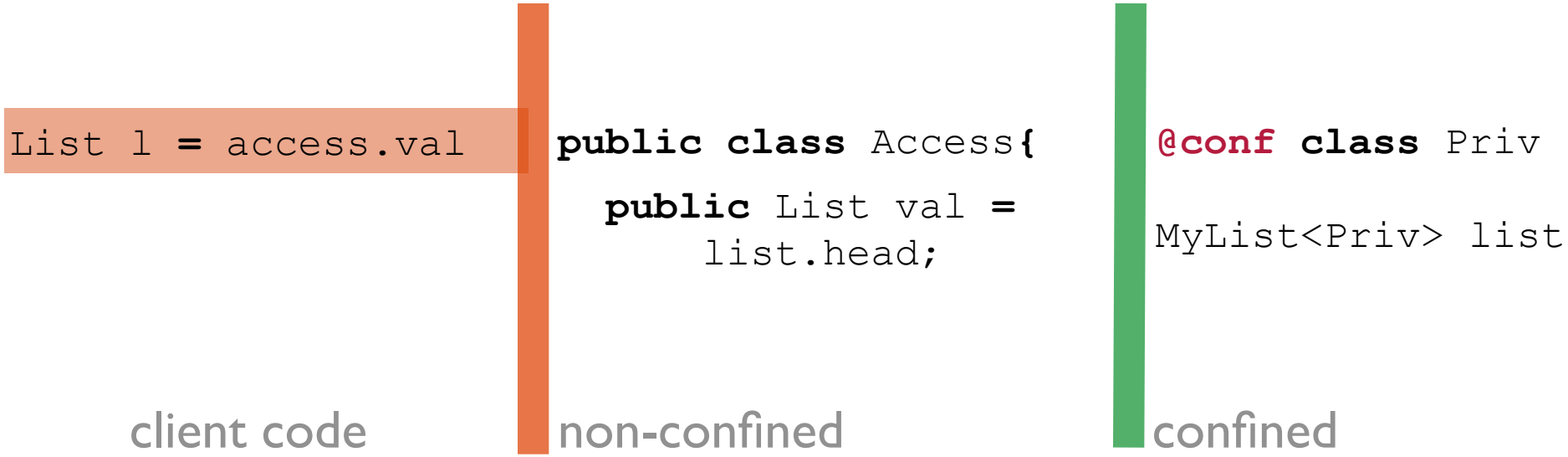
non-confined

confined

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; }
```



C6	Overriding must preserve anonymity

```
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;
```

```
@conf class Priv
   extends B

Naive<Priv> naif
```

client code

non-confined

confined

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 to 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

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]

RealTime Specification for Java

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

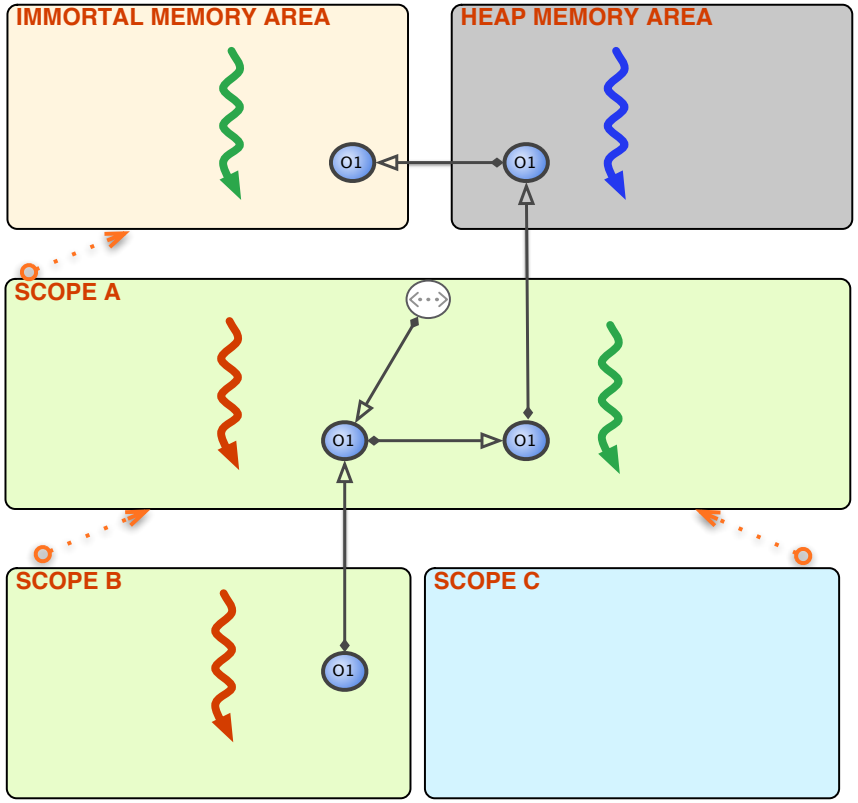
Eliminate GC latency; allow temporary objects

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

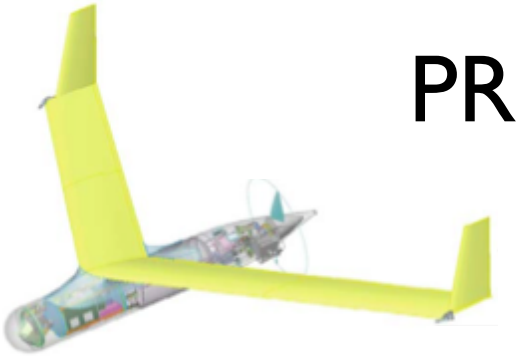
Allocation linear in size, deallocation O(1) (modulo finalizers)

Multiple threads communicate through portal

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



PRISMj



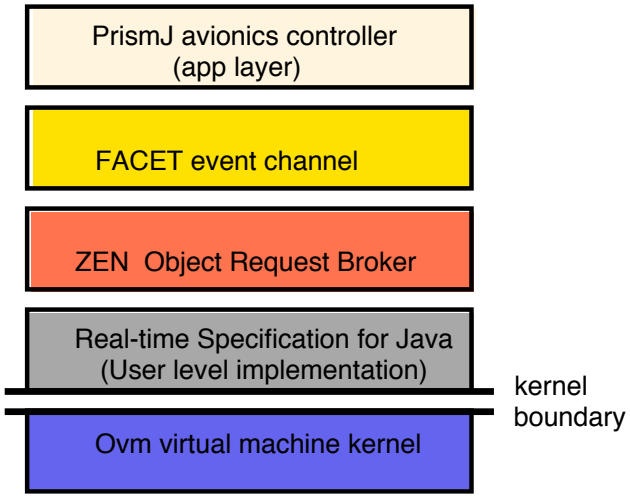
Mission critical avionics DRE

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,
≈ Sun product VM

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

Scoped Types programming model

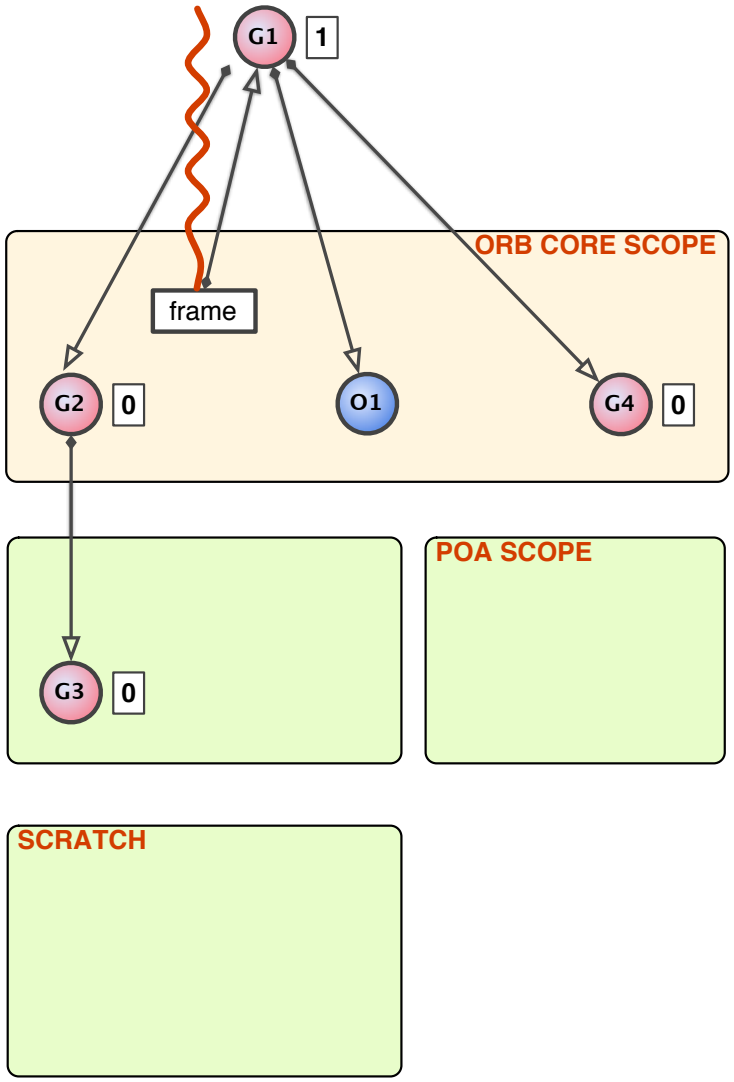
(set of) scopes	≡	Java package
nested scope	≡	nested package
`ScopedMemory` object	≡	`ScopedGate` object
`enter()`	≡	method invocation
`IllegalAssignment`	≡	compile-time error
scope cycle error	≡	compile-time error

Validity constraints for ST programs

$\mathcal{C}1$	A scoped type is visible only to classes in the same package or subpackages.
$\mathcal{C}2$	A scoped type can only be widened to other scoped types in the same package.
$\mathcal{C}3$	The methods invoked on a scoped type must be defined in the same package.
$\mathcal{S}1$	A gate type is only visible to the classes in the immediate super-package.
$\mathcal{S}2$	A gate type cannot be widened to other types.
$\mathcal{S}3$	The methods invoked on a gate type must be defined in the same class.

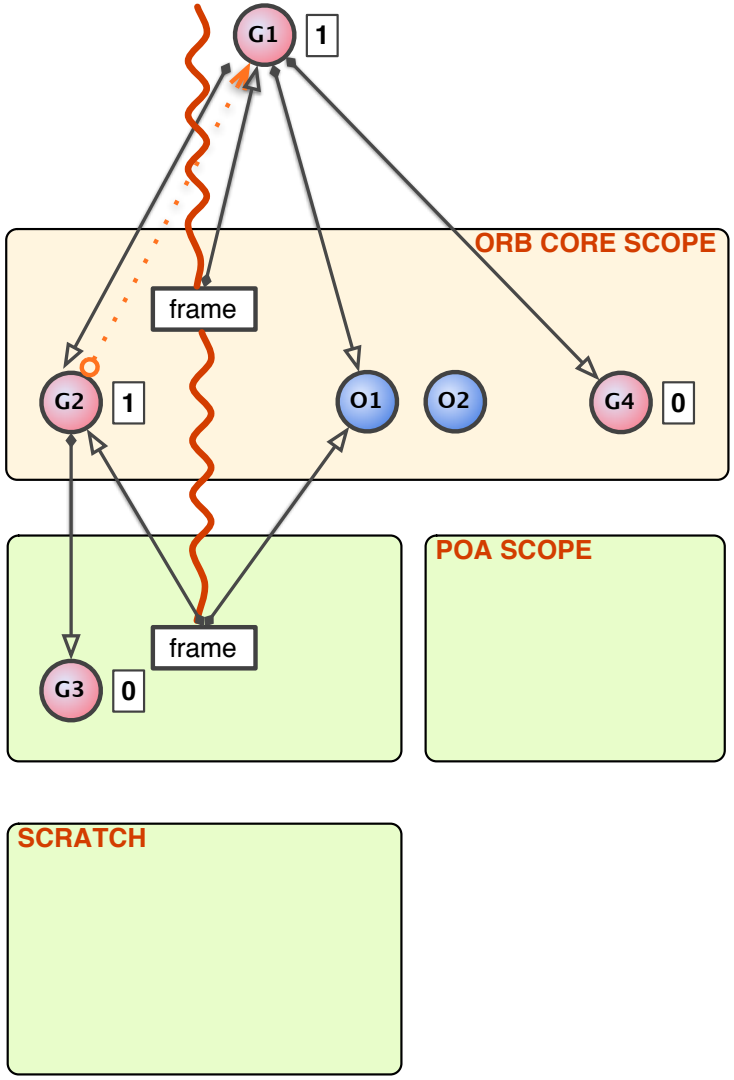
Scoped Program

- 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 norma Java objects with fields pointing into the scope



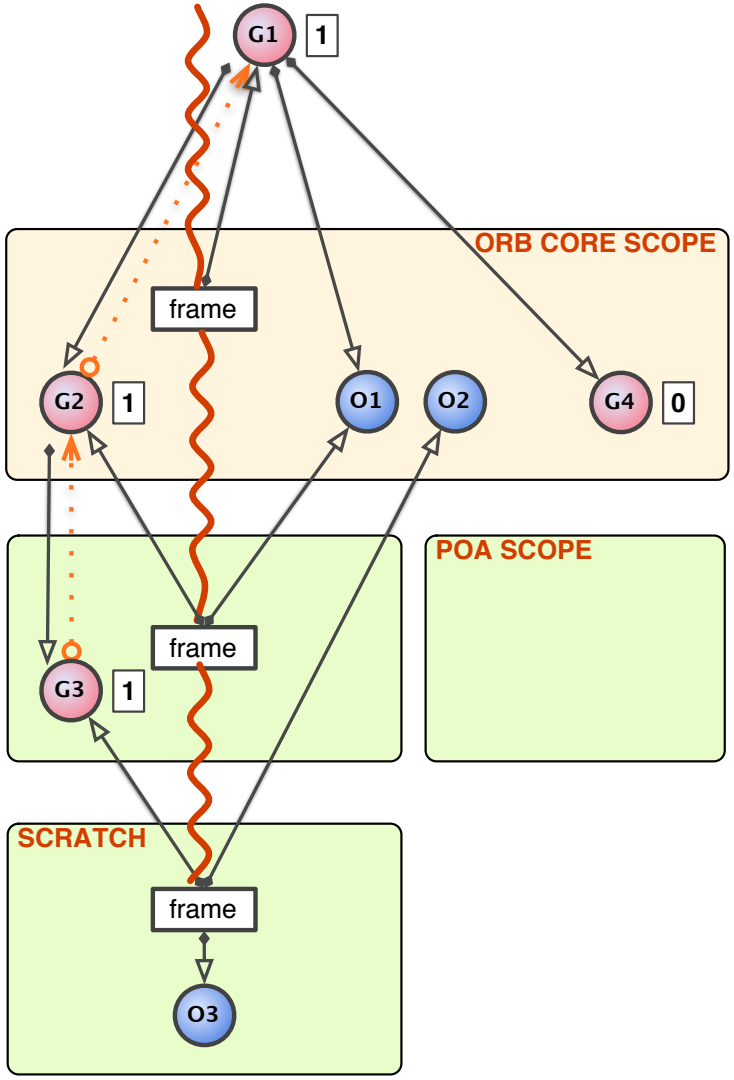
Scoped Program

- 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

$$
\begin{array}{lcl}
\mathtt{L} & ::= & \circ\ \mathtt{class\ P.C} \lhd \mathtt{D}\ \{\, \overline{\mathtt{C}}\ \overline{\mathtt{f}}; \mathtt{K}\ \overline{\mathtt{M}}\,\} \\
\mathtt{K} & ::= & \mathtt{C}()\ \{\mathtt{super}();\ \mathtt{this}.\overline{\mathtt{f}} := \overline{\mathtt{new\ D}()};\} \\
\mathtt{M} & ::= & \mathtt{C\ m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\, \mathtt{return\ e};\ \} \\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m}(\overline{\mathtt{e}}) \mid \mathtt{new\ C}() \mid \mathtt{e.f} := \mathtt{e} \\
 & & \mid \mathtt{spawn\ e} \mid \mathtt{reset\ e} \mid \mathtt{v} \\
\circ & ::= & \mathtt{gate} \mid \mathtt{scoped} \qquad \mathtt{v} ::= \ell \qquad \mathtt{P} ::= \mathtt{p} \mid \mathtt{p.P}
\end{array}
$$

Fig. 7. Syntax of the SJ calculus.

The Scoped Type System

$$\Gamma, \Sigma \vdash \mathtt{x} : \Gamma(\mathtt{x})$$

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

$$\frac{\Gamma, \Sigma \vdash \mathtt{e}_0 : \mathtt{C} \quad \mathit{fields}(\mathtt{C}) = (\overline{\mathtt{C}}\ \overline{\mathtt{f}})}{\Gamma, \Sigma \vdash \mathtt{e}_0.\mathtt{f}_i : \mathtt{C}_i}$$

$$\frac{\Gamma, \Sigma \vdash \mathtt{e}_0 : \mathtt{C}_0 \quad \mathit{fields}(\mathtt{C}_0) = (\overline{\mathtt{C}}\ \overline{\mathtt{f}}) \quad \Gamma, \Sigma \vdash \mathtt{e} : \mathtt{C} \quad \mathtt{C} \preceq \mathtt{C}_i}{\Gamma, \Sigma \vdash \mathtt{e}_0.\mathtt{f}_i = \mathtt{e} : \mathtt{C}_i}$$

$$\frac{\Gamma, \Sigma \vdash \mathtt{e}_0 : \mathtt{C}_0 \quad \mathit{mdef}(\mathtt{m},\ \mathtt{C}_0) = \mathtt{C}_0' \quad \mathit{mtype}(\mathtt{m},\ \mathtt{C}_0') = \overline{\mathtt{C}} \to \mathtt{C} \quad \Gamma, \Sigma \vdash \overline{\mathtt{e}} : \overline{\mathtt{D}} \quad \overline{\mathtt{D}} \preceq \overline{\mathtt{C}} \quad \mathtt{C}_0 \preceq \mathtt{C}_0'}{\Gamma, \Sigma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{C}}$$

$$\frac{\Gamma, \Sigma \vdash \mathtt{e} : \mathtt{Thread}}{\Gamma, \Sigma \vdash \mathtt{spawn}\ \mathtt{e} : \mathtt{Thread}}$$

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

$$\Gamma, \Sigma \vdash \mathtt{new}\ \mathtt{C}() : \mathtt{C}$$

Correctness

- If it is the case that

  $P \equiv P'' \mid t\,[\,\ldots l\,E\,[\,\textbf{reset}\ l_0\,]\,]$

  $\sigma\ P$ *is well typed*

  $\sigma\ P \Rightarrow \sigma'\ P' \qquad$ *where* $P' \equiv P'' \mid t[\,\ldots l\,E[\,l_0\,]]$

- then

  objects allocated in the scope represented by gate $\sigma\ l_0$ are not reachable in $\sigma'\ P'$

  (i.e. no dangling pointers)

[*Zhao, Noble, Vitek. RTSS'04*]

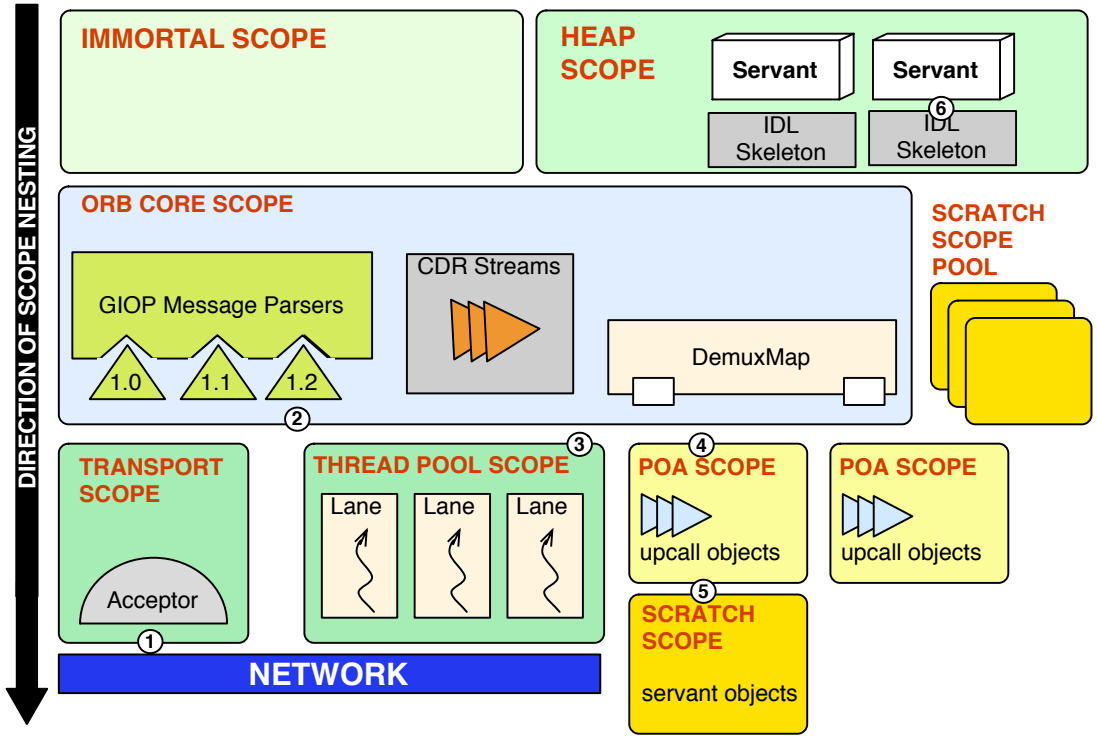
Empiric Validation

- 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.iiop	4	zen.orb.transport.iiop	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() ...
```

```
class Priv

...

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

parent package

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