Safe Region-based Memory Management for Java

Jan Vitek

with Tian Zhao and Jason Baker

PURDUE UNIVERSITY

This talk

- High-level programming languages facilitate software development by abstracting error prone or tedious tasks.
- Memory management is error prone and tedious, but real-time systems sometimes need fine grained control.
- This talk shows how to regain control over memory ... when needed.
- The proposed solution is a hybrid model, a language running mostly on GC but with spurts of manual allocation ... with a new high-level abstraction for Java called *Scopes*.
- Our solution leverages a long line of results from type theory, the key contribution is a statically safe region-based system which is simple and backwards compatible.
- Some related work: MLKit, Cyclone, Ownership types.

Embedded Real-time Systems

- Real-time embedded systems are central to many applications from avionics, to to automotive industry; they are the largest installed base of microprocessors
- The size of embedded systems is growing steadily; up to multi-million line systems (e.g. DD(X) battleship control)
- Very low level languages (e.g. assembly) are not viable; low-level ones (eg. C) are barely tolerable; Ada has unfortunately not found the degree of adoption it deserved...

  ... new programming language abstractions for embedded real-time systems are sorely needed.

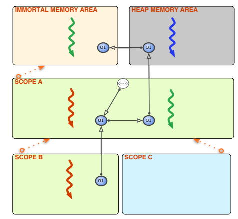
- Reusability is becoming mandatory, even Boeing changing from the old way (recode from scratch) in favor of COTS

Automated Memory Management

- Garbage collection algorithms along with memory-safe languages prevent all memory errors.
- The cost is an increase in memory requirements (or a decrease in performance, take you pick) with 2x over the optimal not uncommon.
- But the main issue is pause times. Stop-the-world collectors require the application to pause for >100ms. In a RT system this can mean missing deadlines.
- Recently, real-time collectors have been able to bound pauses to <20ms, this comes at some cost in throughput
- The state of the art in RT GC seems good enough for many RT applications, but not for some hard real-time subsystems

Region-based allocation

- Region-based allocation is an alternative to GC and manual allocation. It follows from the observation that data exhibits liveness locality (allocation and deallocation times are correlated).
- Idea: create regions which are pools of memory that can be used to store data and bulk deallocated in constant time.
- This simplifies memory management because instead of tracking individual pointers, we deal with entire regions.
- A stack of regions allows to deal with different lifetimes. Parent regions outlive child regions; data is allocated in the region most closely matching its lifetime.
- In itself, region-based allocation is not safe. Programming language techniques must be devised to enforce safety (either run-time or compile-time).
- The Real-time Specification for Java provides region-based MM, but safety is enforced dynamically. This entails runtime overheads and runtime failures.



Motivating example: PRISMj

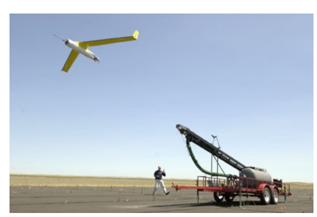
Mission critical avionics DRE

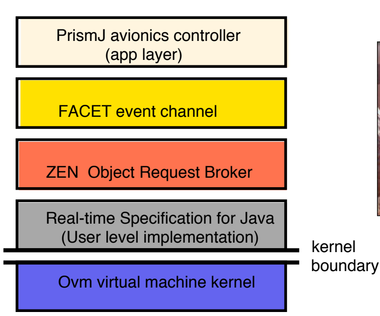
Boeing, Purdue, UCI, WUSTL

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)
several hundred RT threads

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

SCOPES

Design Requirements

- The design of scopes abides by the following requirements:

1. ***Static type safety***: *no runtime errors and no runtime checks.*
2. ***GC-safety***: *no interference with the GC so that a thread within a scope may safely preempt the garbage collector and never need block for GC.*
3. ***Bytecode compatibility***: *the output of the compiler should be valid Java bytecode. The only allowed change to the tool chain is the addition of an extra verification pass.*
4. ***Minimal changes to the VM***: *modifications should be limited to the memory subsystem.*
5. ***Source-level backwards compatibility***: *existing Java class should be reusable from scope code.*

Scope Basics

- Scope objects reify allocation contexts
- Creating a scope allocates a backing store
- Invoking a method on a scope switch allocation context
- Releasing a scope deallocates the backing store

- nb: use of syntactic sugar can be replaced by annotations

```
r = new processor;
while (b = nextRequest()) {
   r.getMessage(b,m);
   m.dispatch();
   release r;   }
```


Programming with Scopes

- Scopes have fields, methods, nested classes and scopes.
- Bound classes are always allocated within their enclosing scope.
- Scopes form a tree at runtime, parents always outlive children.
- Invoking a bound class method switches allocation context.

```
scope processor {
   class Unpacker { ...
       void parse(Bytes b) {...}
       void write(Message m) {...}
   }
   void getMessage(Bytes b, Message m) {
       Unpacker p = new Unpacker();
       p.parse(b);
       p.write(m);
}    }
```

A Scope Pool

- Multiple threads can execute in the same scope and communicate via scope fields
- Releasing a scope deallocates the contents only if no thread is active in it

```
scope pool {
  area[] a;  int cnt;
  void create(int i) {
    a=new area[i];
    for(int j=0;j<i;j++) a[j]=new area;
  }
  void run(Message m, Bytes data){
    int i = cnt++ % a.length;
    a[i].run(m, data);
    release a[i];
  }
  scope area {
    void run(Message m, Bytes data){
      ... // service request
} } }
```


Implicit Polymorphism

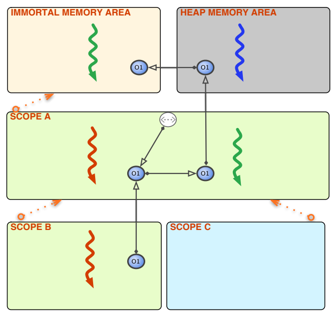
- Classes not lexically enclosed within a scope are **free** classes.
- Free classes are *implicitly* scope-polymorphic.
- Instances of free classes can be allocated within any scope, with the restriction that they not escape their enclosing scope.
- Many library classes can be reused.

```
class List {   List next;   Object value; }

scope user {
    List l;
    void create(int i) {
      List t;
      for(int j=0;j<i;j++)
      {  t = new List(); t.next =l; l = t; }
} }
```

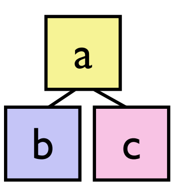
Permanent Classes

- Permanent classes --annotated with a **`top`** qualifier-- exist in immortal memory.
- Instances of a permanent class are never deallocated.
- They are allowed to create top-level scopes.
- Permanent classes can be observed from the heap, but cannot refer to heap objects.
- If a thread is spawned in a permanent object, it is guaranteed not to experience GC-interference.



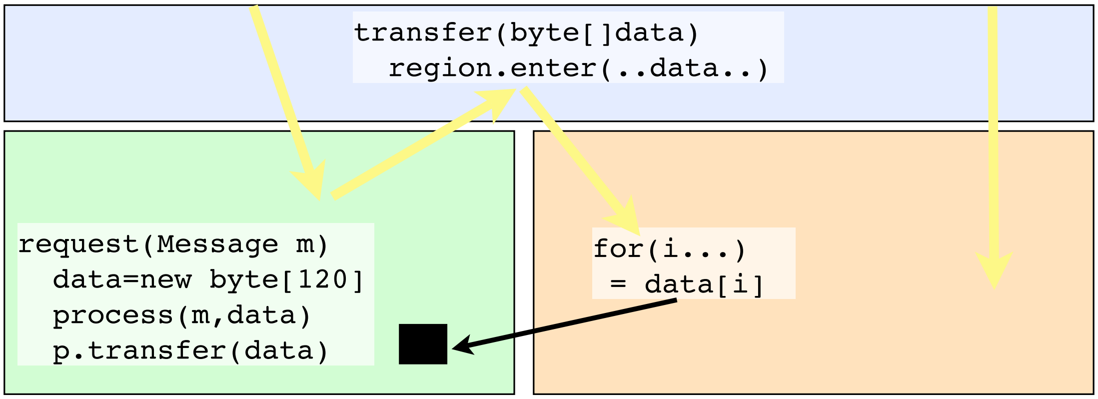
Crossing Scopes

- Bound classes allocated in parent scopes are visible from subscopes.
- Invoking a method of a bound class changes the allocation context to the parent, when the method returns the allocation context is restored.
- Each thread has a stack of scope, but at any given time only one scope is the effective scope.




```
scope a {
  class Box {
    c s2;
    void enter() { s2.mc(); }
  }
  scope b { void mb(Box box) { box.enter(); } }
  scope c { void mc(){...} }
  void ma() {
    b s1 = new b;  c s2 = new c;
    Box box = new Box(); box.s2 = s2;
    s1.mb(box);
} }
```

Pipelined computations



- LIFO discipline not suited when successive rounds of filtering are needed.
- Pipelined computation can be set up by lending a reference to a sibling region.
- Intuition: a region is pinned while a thread is active within it. If a thread crosses regions, it can safely access its origin scope.
- Safe as long as sibling references are not retained.

Quasi Linear Types

```
scope r {
  class Bridge {  q qscp;
      void run(p pscp, q  qscp) {
          this.qscp=qscp;  pscp.enter(this);
      }
      void handoff(borrow byte[] data) {
          qscp.enter( data);
  }   }
  scope p {
      void enter(Bridge b) {
         byte[] data=new byte[20];
         ... // some computation
         b.handoff( data);
  }   }
  scope q {
      void enter(borrow byte[] data) {
          ... //use cross-scope reference
}  }   }
```

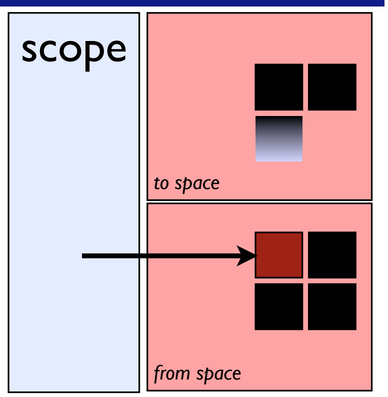
Quasi Linear Types

- **borrow** -- is a type qualifier that can be used on method arguments and locals.
- The type system ensures a borrowed reference doesn't outlive the method invocation.
- It is safe to access a borrowed reference from any scope.
- Borrowed references cannot be assigned to fields of an object or scope.
- It's safe to modify primitive fields of a borrowed object, but not reference fields.
- Any reference retrieved from a borrowed reference is borrowed as well.

```
                                        class BL{BL next;int i;F f;}
method_m( borrow BL bl ) {
   borrow nbl = bl.next;     // ok
   bl.i++;                   // ok
   bl.f = new F();           // Unsafe! Compile time error.
```

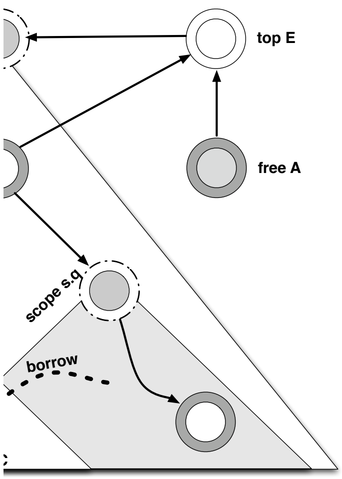

GC-safety

- In a hybrid setting, hard RT code mustn't pause for GC.
- Two scenarios for GC pauses
  - ☑ real-time code allocates in the heap
  - ☑ real-time thread released in midst of GC:
    wait for GC to finish or observe inconsistent data
- Scopes can not refer to the heap thus neither can happen
- Do not prevent transitive priority inversion scenarios
  1) a non-rt thread grabs a lock on a permanent object,
  2) GC is triggered,
  3) a RT thread tries to access the locked object.



The Type System

- The type system enforces the following invariants:



(1) *Scopes* (`s, s.p, s.q`) may refer to objects within their allocation context and to bound classes of ancestor scopes.

(2) *Bound classes* (e.g. `C, D`) may refer to any co-located object, and to bound classes allocated in a parent scope.

(3) *Free class instances allocated in a scope* (e.g. `B`) may refer to any co-located object.

(4) *Free class instances allocated in the heap* (e.g. `A`) may refer to permanent classes.

(5) *Permanent classes* (e.g. E) may refer to other permanent classes and top-level scopes.

(6) *Borrowed reference* may refer to any bound class.

The Type System

- Some of the constraints are:
  - ☑ Scopes can not extend other classes
  - ☑ Bound classes can only extend bound classes in the same scope (or free classes)
  - ☑ Free classes can only extend free classes.
  - ☑ Borrowed types can not be stored in fields, or have reference types stored into.
  - ☑ Widening can not cast off the borrowed annotation.
  - ☑ Reference types returned by a borrowed object must be borrowed.
  - ☑ Free types are globally visible
  - ☑ Bound types are visible in the defining scope and subscopes
  - ☑ Scope types are only visible in the enclosing scope
  - ☑ The public interface (field/methods) of scopes and bound types can not contain free classes
  - ☑ Widening is restricted in a number of ways ...

Proof of Soundness

- The scope type system has been formalized as an extension of Igarashi e.a.'s Featherweight Java, and proven sound. The main theorem ensures that after a release, there cannot be any reference into the scope.

$$
\begin{array}{lcl}
\mathtt{R} & ::= & \mathtt{scope\ S}\ \{\ \overline{\mathtt{K\ f}};\ \overline{\mathtt{M}}\ \} \\
\mathtt{L} & ::= & \mathtt{class\ C} \triangleleft \mathtt{C}\ \{\ \overline{\mathtt{K\ f}};\ \overline{\mathtt{M}}\ \} \\
\mathtt{M} & ::= & \mathtt{K\ m}\ (\overline{\mathtt{T\ x}})\ \{\, \mathtt{return\ e};\ \} \\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{v} \mid \mathtt{new\ K()} \mid \mathtt{e.f} \mid \mathtt{e.f} := \mathtt{e} \mid \mathtt{e.m}(\overline{\mathtt{e}}) \\
 & & \mid (\mathtt{K})\ \mathtt{e} \mid \mathtt{release\ e} \\
\mathtt{v} & ::= & \ell \mid \mathtt{null} \\
\mathtt{T} & ::= & \mathtt{K} \mid \mathtt{borrow\ C} \\
\mathtt{K} & ::= & \mathtt{C} \mid \mathtt{S} \\
\mathtt{C} & ::= & \mathtt{S.c} \mid \mathtt{c} \\
\mathtt{S} & ::= & \mathtt{S.s} \mid \mathtt{top}
\end{array}
$$

$$\frac{\Gamma \vdash_{K_t} \mathtt{e} : \mathtt{T} \quad \mathit{fields}(\mathtt{T}) = (\overline{\mathtt{K}}\ \overline{\mathtt{f}}) \quad \mathtt{T} \uparrow \mathtt{K_i} = \mathtt{T}' \quad (\mathtt{T}, \mathtt{K_i}) <: (\mathtt{K}_t, \mathtt{T}') \quad \mathtt{T}'\ \mathit{viz}\ \mathtt{K}_t}{\Gamma \vdash_{K_t} \mathtt{e.f_i} : \mathtt{T}'} \quad \text{(T-FIELD)}$$

$$\frac{\Gamma \vdash_{K_t} \mathtt{e} : \mathtt{K} \quad \mathit{fields}(\mathtt{K}) = (\overline{\mathtt{K}}\ \overline{\mathtt{f}}) \quad \Gamma \vdash_{K_t} \mathtt{e}' : \mathtt{K}' \quad (\mathtt{K}_t, \mathtt{K}') <: (\mathtt{K}, \mathtt{K_i})}{\Gamma, \Sigma \vdash_{K_t} \mathtt{e.f_i} = \mathtt{e}' : \mathtt{K}'} \quad \text{(T-UPD)}$$

$$\frac{\mathtt{K} \neq \mathtt{c} \Rightarrow \quad (\mathit{scopeof}(\mathtt{K}_t) = \mathtt{S} \quad \mathtt{K} = \mathtt{S.c} \ \vee\ \mathtt{K} = \mathtt{S.s})}{\Gamma \vdash_{K_t} \mathtt{new\ K()} : \mathtt{K}} \quad \text{(T-NEW)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{K_t} \mathtt{e} : \mathtt{T}_r \quad \Gamma \vdash_{K_t} \overline{\mathtt{e}} : \overline{\mathtt{T}'} \quad \mathit{mtype}(\mathtt{m},\ \mathtt{T}_r) = \overline{\mathtt{T}} \rightarrow \mathtt{K} \quad \mathtt{T}_r \uparrow \mathtt{K} = \mathtt{T} \quad \mathtt{T}\ \mathit{viz}\ \mathtt{K}_t \\ \forall \mathtt{i}, (\mathtt{K}_t, \mathtt{T}'_\mathtt{i}) <: (\mathtt{T}_r, \mathtt{T_i}) \quad (\mathtt{T}_r, \mathtt{K}) <: (\mathtt{K}_t, \mathtt{T})\end{array}}{\Gamma \vdash_{K_t} \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{T}} \quad \text{(T-INVK)}$$

$$\frac{\Gamma \vdash_{K_t} \mathtt{e} : \mathtt{K}' \quad (\mathtt{K} = \mathtt{c} \ \Rightarrow \quad \mathtt{K}' = \mathtt{c}' \ \vee\ \mathit{scopeof}(\mathtt{K}') = \mathit{scopeof}(\mathtt{K}_t))}{\Gamma \vdash_{K_t} (\mathtt{K})\ \mathtt{e} : \mathtt{K}} \quad \text{(T-CAST)}$$

$$\Gamma \vdash_{K_t} \mathtt{x} : \Gamma(\mathtt{x}) \quad \text{(T-VAR)} \qquad \frac{\Gamma \vdash_{K_t} \mathtt{e} : \mathtt{S}}{\Gamma \vdash_{K_t} \mathtt{release\ e} : \mathtt{S}} \quad \text{(T-REL)}$$

Method typing:

$$\frac{\begin{array}{c}\overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C},\ \emptyset \vdash_\mathtt{C} \mathtt{e} : \mathtt{K}' \quad \mathtt{K}, \overline{\mathtt{T}}\ \mathit{viz}\ \mathtt{C} \\ \mathit{override}(\mathtt{m}, \mathtt{C}, \mathtt{C}') \quad (\mathtt{C}, \mathtt{K}') <: (\mathtt{C}, \mathtt{K})\end{array}}{\mathtt{C} \triangleleft \mathtt{C}' \vdash \mathtt{K\ m}\ (\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \{\, \mathtt{return\ e};\ \}} \quad \text{(T-METHC)} \qquad \frac{\begin{array}{c}\overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{S},\ \emptyset \vdash_\mathtt{S} \mathtt{e} : \mathtt{K}' \quad \mathtt{K}, \overline{\mathtt{T}}\ \mathit{viz}\ \mathtt{S} \\ (\mathtt{S}, \mathtt{K}') <: (\mathtt{S}, \mathtt{K})\end{array}}{\mathtt{S} \vdash \mathtt{K\ m}\ (\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \{\, \mathtt{return\ e};\ \}} \quad \text{(T-METHS)}$$

Class typing:

$$\frac{\mathtt{C} \triangleleft \mathtt{C}' \vdash \overline{\mathtt{M}} \quad \overline{\mathtt{K}}\ \mathit{viz}\ \mathtt{C} \quad \mathtt{C}' = \mathtt{c} \ \vee\ (\mathtt{C} = \mathtt{S.c} \ \wedge\ \mathtt{C}' = \mathtt{S.c}')}{\mathtt{class\ C} \triangleleft \mathtt{C}'\ \{\ \overline{\mathtt{K\ f}};\ \overline{\mathtt{M}}\ \}\ \mathtt{OK}} \quad \text{(T-CLASS)} \qquad \frac{\mathtt{S} \vdash \overline{\mathtt{M}} \quad \overline{\mathtt{K}}\ \mathit{viz}\ \mathtt{S}}{\mathtt{scope\ S}\ \{\ \overline{\mathtt{K\ f}};\ \overline{\mathtt{M}}\ \}\ \mathtt{OK}} \quad \text{(T-SCOPE)}$$

Conclusions

- High-level languages for real-time systems must support memory-safe, and GC-safe, manual memory management techniques.
- Region-based allocation provides linear time allocation and bulk deallocation.
- Type systems can prevent all memory violations at compile time.
- The scopes abstraction introduced in this talk provides static safety without requiring drastic changes to Java.
- Scopes are simple to use and allow the reuse of existing code.