Scoped Types
A Statically Safe Memory Model for the Real-time Specification for Java

Jan Vitek
&
Tian Zhao & James Noble

[Support: DARPA PCES and NSF/NASA HDCCSR]
Scoped Types for Real-Time Java

Writing functionally correct programs in the RTSJ is harder than in Java, because of the implicit well-formedness relation on references imposed by the RTSJ memory model.

Contributions:

- formalization of the well-formedness relation in a typed object calculus
- proof: well-typed program $\Rightarrow$ no dangling pointers & no memory leaks

Our results can be used as a checked discipline for RTSJ programs on vanilla VM
The Real-Time Specification for Java
The Real-Time Specification for Java

Extend Java and VM Specifications with an API that enables creation, verification, analysis, execution, and management of Java threads whose correctness conditions include timeliness constraints

Timeline:

1999  JSR-001 accepted w. 40 companies involved (IBM, Sun...)
2001  RTSJ v.1.0
2002  TimeSys reference implementation
2003  jRate, Ovm open source; jTime product
       MacKinac project starts @ Sun Grenoble
2004  RTSJ v.1.0.1
Why Java as a Real-Time Platform?

Why switch to Java?

Software-intensive systems require high-level prog. langs. C++ not ideal, Ada struggling
Java = lingua franca in education, well specified, ~ simple
combine real-time and plain Java in the same VM

What about the performance myth?

Folklore: Java 2 times slower than C; true for hand-tuned code, in practice < 2
Component-based apps easier to optimize in Java because code in a common IR
Dynamic compilers getting better than static compilers

Is Java too dynamic?

Classloading need not be used ⇒ off-line whole-system optimization
Garbage collection still a problem (if you allocate)
PRISMj

Mission critical avionics DRE
Boeing, Purdue, UCI, WUSTL
Route computation, Threat deconfliction algorithms
ScanEagle UAV

Middleware stack is 1MLOC Java
⇒ 52KLOC w. Ovm optimizing compiler!

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

<table>
<thead>
<tr>
<th>System</th>
<th>K LOCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRISMj</td>
<td>109K</td>
</tr>
<tr>
<td>FACET event channel</td>
<td>15K</td>
</tr>
<tr>
<td>Zen CORBA ORB</td>
<td>179K</td>
</tr>
<tr>
<td>RTSJ LIBRARIES</td>
<td>60K</td>
</tr>
<tr>
<td>CLASSPATH LIBRARIES</td>
<td>500K</td>
</tr>
<tr>
<td>OVM virtual machine</td>
<td>220K</td>
</tr>
</tbody>
</table>

PrismJ avionics controller (app layer)
FACET event channel
ZEN Object Request Broker
Real-time Specification for Java (User level implementation)
Ovm virtual machine kernel

Embedded Planet PowerPC 8260
Core at 300 MHz
256 Mb SDRAM
32 Mb FLASH
PC/104 mechanical sized
Embedded Linux
The Real-Time Specification for Java

New language in Java clothing

no changes to syntax but idiomatic reinterpretation of existing constructs

- Thread Scheduling & Dispatching
- Synchronization
- Asynchronous Actions
- Memory management
- Time, Clocks and Timers
RTSJ Design Overview

Challenges

- hard/soft/non RT codes must be allowed to operate in the same execution environment (90/9/1 rule)

- RT threads should never wait for the garbage collector (GC)

- prevent undesirable interferences, e.g. RT thread blocks while waiting for a plain thread to release a monitor
Scoped Memory

Object lifetime controlled by reachability, when last thread leaves an area, all objects allocated in it reclaimed.

Eliminate GC latency; allow temporary objects.

NHRT don’t read from heap, thus protected from GC interference.

Allocation time 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.
Scoped Memory Usage

positions = new float[size];
s= new ScopedMemoryArea(min, max);
r=new Runnable() {
    void run() {
        ...read sensor output...
        tmp = new float[4*size];
        for(i=0;i<size;i++)
            positions[i]=...tmp...
    }
}
s.enter(r);
done();
Scoped Memory Usage

positions = new float[size];
s = new ScopedMemoryArea(min, max);
r = new Runnable() {
    void run() {
        ...read sensor output...
        tmp = new float[4*size];
        for(i=0; i<size; i++)
            positions[i] = ...tmp...
    }
};
s.enter(r);
done();
Scoped Types
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]
Scoped Types programming model

(set of) scopes $\equiv$ Java package

nested scope $\equiv$ nested package

ScopedMemory object $\equiv$ ScopedGate object

enter() $\equiv$ method invocation

IllegalAssignment $\equiv$ compile-time error

scope cycle error $\equiv$ compile-time error
### Validity constraints for ST programs

<table>
<thead>
<tr>
<th>C1</th>
<th>A scoped type is visible only to classes in the same package or subpackages.</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>A scoped type can only be widened to other scoped types in the same package.</td>
</tr>
<tr>
<td>C3</td>
<td>The methods invoked on a scoped type must be defined in the same package.</td>
</tr>
<tr>
<td>S1</td>
<td>A gate type is only visible to the classes in the immediate super-package.</td>
</tr>
<tr>
<td>S2</td>
<td>A gate type cannot be widened to other types.</td>
</tr>
<tr>
<td>S3</td>
<td>The methods invoked on a gate type must be defined in the same class.</td>
</tr>
</tbody>
</table>
A Scoped Type Program

package corba.orb

public class ORB
    extends ScopedGate {
    POA[] poas = new POA[10];
    Message msg = new Message();

    public void handleRequest(Buffer b) {
        msg.init(b);
        POA poa = findPoap(msg);
        poa = (pos==null) ?
            addPoap(new Poap(msg)) : poa;
        poa.handleRequest(msg);
    }
    ...
}

public class Message {
    ...
    public Message duplicate() {
        return new Message(...); }
}

package corba.orb.poa

public class POA
    extends ScopeGate {
    Scratch scope = new Scratch();
    void handleRequest(Message msg) {
        Message message = msg;
        if (...)
            message = msg.clone();
        scratch.dispatch(message);
        scratch.reset();
    }
}

package corba.orb.poa.scratch

public class Scratch
    extends ScopeGate {
    void dispatch(Message m) {
        ...
    }
    ...
}

CASSIS, Nice, March 05
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
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 ::= ○ class P.C ≪ D { C f; K M }
K ::= C() { super(); this.f := new D(); }
M ::= C m(C x) { return e; }
e ::= x | e.f | e.m(e) | new C() | e.f := e
    | spawn e | reset e | v
○ ::= gate | scoped
    v ::= ℓ
    P ::= p | p.P

Fig. 7. Syntax of the SJ calculus.
The Scoped Type System

\[
\Gamma, \Sigma \vdash x : \Gamma(x) \\
\Gamma, \Sigma \vdash \ell : \Sigma(\ell) \\
\frac{\Gamma, \Sigma \vdash e_0 : C}{\Gamma, \Sigma \vdash e_0.f_i : C_i}
\]

fields(C) = (\overline{C} \overline{f})

\[
\Gamma, \Sigma \vdash e_0 : C_0 \quad mdef(m, C_0) = C'_0 \\
\frac{mtype(m, C'_0) = \overline{C} \rightarrow C}{\Gamma, \Sigma \vdash e : \overline{D} \\ \overline{D} \leq \overline{C} \\ C_0 \leq C'_0}
\]

\[
\Gamma, \Sigma \vdash e_0.m(\overline{e}) : C
\]

\[
\Gamma, \Sigma \vdash \text{new } C() : C
\]

\[
\Gamma, \Sigma \vdash e_0 : C_0 \quad fields(C_0) = (\overline{C} \overline{f}) \\
\Gamma, \Sigma \vdash e : C \quad C \leq C_i
\]

\[
\Gamma, \Sigma \vdash e_0.f_i = e : C_i
\]

\[
\Gamma, \Sigma \vdash \text{spawn } e : \text{Thread}
\]

\[
\Gamma, \Sigma \vdash \text{reset } e : C
\]

\[
\Gamma, \Sigma \vdash e : \text{Thread} \\
\Gamma, \Sigma \vdash \text{spawn } e : \text{Thread}
\]

\[
\Gamma, \Sigma \vdash e : C \quad C \text{ is a gate}
\]

\[
\Gamma, \Sigma \vdash \text{reset } e : C
\]
Correctness

- If it is the case that
  \[ P \equiv P'' | t \left[ \ldots l E \left[ \text{reset } l_0 \right] \right] \]
  \( \sigma P \) is well typed
  \( \sigma P \Rightarrow \sigma' P' \) where \( P' \equiv P'' | t[ \ldots l E[ l_0 ]] \)

- 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

<table>
<thead>
<tr>
<th>zen.orb</th>
<th>38</th>
</tr>
</thead>
<tbody>
<tr>
<td>zen.orb.any</td>
<td>2</td>
</tr>
<tr>
<td>zen.orb.any.monolithic</td>
<td>1</td>
</tr>
<tr>
<td>zen.orb.dynany</td>
<td>11</td>
</tr>
<tr>
<td>zen.orb.giop</td>
<td>6</td>
</tr>
<tr>
<td>zen.orb.giop.IOP</td>
<td>3</td>
</tr>
<tr>
<td>zen.orb.giop.type</td>
<td>5</td>
</tr>
<tr>
<td>zen.orb.giop.v1_0</td>
<td>9</td>
</tr>
<tr>
<td>zen.orb.giop.v1_1</td>
<td>5</td>
</tr>
<tr>
<td>zen.orb.giop.v1_2</td>
<td>4</td>
</tr>
<tr>
<td>zen.orb.policies</td>
<td>13</td>
</tr>
<tr>
<td>zen.orb.resolvers</td>
<td>2</td>
</tr>
<tr>
<td>zen.orb.transport</td>
<td>11</td>
</tr>
<tr>
<td>zen.orb.transport.IIOP</td>
<td>4</td>
</tr>
<tr>
<td>zen.poa</td>
<td>16</td>
</tr>
<tr>
<td>zen.poa.mechanism</td>
<td>27</td>
</tr>
<tr>
<td>zen.poa.policy</td>
<td>7</td>
</tr>
<tr>
<td>zen.util</td>
<td>21</td>
</tr>
<tr>
<td>zen.org.giop</td>
<td>4</td>
</tr>
<tr>
<td>zen.org.giop.IOP</td>
<td>3</td>
</tr>
<tr>
<td>zen.org.giop.type</td>
<td>5</td>
</tr>
<tr>
<td>zen.org.giop.v1_0</td>
<td>9</td>
</tr>
<tr>
<td>zen.org.giop.v1_1</td>
<td>5</td>
</tr>
<tr>
<td>zen.org.giop.v1_2</td>
<td>4</td>
</tr>
<tr>
<td>zen.org.policies</td>
<td>9</td>
</tr>
<tr>
<td>zen.org.transport</td>
<td>3</td>
</tr>
<tr>
<td>zen.org.transport.IIOP</td>
<td>1</td>
</tr>
<tr>
<td>zen.poa</td>
<td>3</td>
</tr>
<tr>
<td>zen.poa.mechanism</td>
<td>19</td>
</tr>
<tr>
<td>zen.poa.policy</td>
<td>7</td>
</tr>
<tr>
<td>zen.util</td>
<td>11</td>
</tr>
<tr>
<td>scope.orb</td>
<td>45</td>
</tr>
<tr>
<td>scope.orb.connection</td>
<td>7</td>
</tr>
<tr>
<td>scope.orb.requestprocessor</td>
<td>10</td>
</tr>
<tr>
<td>scope.requestwaiter</td>
<td>3</td>
</tr>
</tbody>
</table>

... 2nd round of refactoring in progress
Conclusions

✓ Java is safer than C++ because
  VM guarantees memory errors do not occur ⇒ increased productivity
  comes at cost in performance/predictability
  GC = a system-managed memory leak

✓ RTSJ is memory-safe but
  harder to use because of extra dimension (locality)
  errors are reported at run-time ⇒ decreased reliability
  memory leaks are reported eagerly

✓ Scoped Types
  prevent dynamic access violation and
  structure code so as to reflect a program’s memory layout
Credits

- **Ovm Core:**
  - **SUN:** Krzysztof Palacz
  - **DLTech (Brisbane):** David Holmes
  - **Purdue:** Jason Baker, Chap Flack, Filip Pizlo, Hiroshi Yamauchi
- **... and also**
  - Christian Grothoff, Marek Prochazka, Andrei Madan (Medtronic), Jacques Thomas, James Liang (JPL), Antonio Cunei
- **Scoped Types**
  - Tian Zhao (UWisc) and James Noble, Alex Pontanin (U Vic, NZ)