Scoped Types for Real-time Java

Tian Zhao James Noble[*] Jan Vitek[†]
University of Wisconsin, Milwaukee
[*]Victoria University of Wellington
[†] Purdue University

ABSTRACT

One of the distinctive features of the Real-Time Specification for Java (RTSJ) is the new memory management model based on scoped memory areas. This model allows real-time programmers to ensure timely reclamation of memory and predictable performance for real-time tasks. The price to pay for these benefits is an unfamiliar programming model that is complex, requires runtime checks on all memory accesses, and rewards design-time errors with run-time crashes. In this paper we describe Scoped Types, a statically enforced programming discipline that eschews complexity and run-time exceptions in favour of simplicity and safety. We formalize our programming model in terms of a simple object calculus and prove soundness of its type system.

1. INTRODUCTION

The Real-Time Specification for Java (RTSJ) [4] is designed to allow the construction of large scale real-time systems in type-safe high-level programming languages. The benefits of using Java for mission critical systems are currently being evaluated in a number of companies and research labs such as Boeing [32] and JPL [25]. As of this writing a high-quality commercial implementation of the specification has been released [23], and open source alternatives are in development [2, 15, 26, 9, 36, 17, 33]. While the specification includes many necessary features, the one that is most likely to affect how real-time Java programs are written is the new memory management model based on Scoped Memory.

An obvious concern for meeting hard real-time constraints in Java is the interaction of automatic memory management with real-time tasks. While garbage collection (GC) frees the programmer from the burden of tracking memory usage, it introduces unpredictability because the timing and duration of GC pauses is unknown. To address this problem, the RTSJ provides: (a) regions of memory which are not subject to garbage collection, called scoped memory areas, or *scopes*, (b) threads that are guaranteed to never interact with the heap and thus never interfere with, or be preempted by, the garbage collector. Scoped memory areas provide predictable allocation, and ensure that hard real-time threads will not block while memory is being reclaimed. Their design abides by a number of practical constraints. Firstly, hard real-time tasks must to coexist with soft real-time and non real-time tasks. Secondly, real-time Java programs must remain backward compatible, thus changes to the standard libraries or language syntax were deemed impossible. Lastly, Java being an inherently concurrent language, multi-threading must be supported. The resulting design is a distinctive programming model which differs from existing region-based models such as the MLKit [34].

In principle, scoped memory resembles the familiar notion of cactus-stack allocation [3, 31]. Each scope provides a pool of memory that can be used to allocate objects. Individual objects cannot be deallocated, instead the entire scope is reclaimed when its contents become unreachable. The main difference with stack allocation is that scopes are first-class entities which can be entered by multiple threads. The order in which threads enter scopes induces a runtime structure on scoped memory areas that determines permissible reference patterns. When a real-time thread executing in area $\mathtt{M}_1$ first enters area $\mathtt{M}_2$, area $\mathtt{M}_1$ becomes the parent of $\mathtt{M}_2$. RTSJ semantics guarantee that $\mathtt{M}_2$ will be reclaimed before $\mathtt{M}_1$, the lifetime of a nested area is thus always shorter than its parent area. Threads executing in an area allocate from the same pool and communicate though shared variables. When the last thread exits an area, the objects allocated within it are reclaimed. The last-in first-out natures of scoped memory allows for objects allocated in nested scopes, *e.g.* $\mathtt{M}_2$, to refer objects in their parent, *e.g.* $\mathtt{M}_1$, but not the converse as holding on to a reference into a shorter lived-scoped may lead to dangling references, and jeopardize type safety, when that scope was discarded.

To ensure type safety of real-time Java programs, the following invariants must be maintained at runtime:

1. Because a scope can be reclaimed at any time, an outer scope may not hold a reference to an object within a more deeply nested inner scope.

2. To avoid cycles in the scope parent relation, the nesting structure of scopes is restricted to trees. In other words, a scope may have only a single parent.

3. Because scopes can be shared by multiple threads, objects allocated within a scoped memory area can not be discarded until all threads have finished using that area.

Maintaining these invariants impose burdens on programmers. Any assignment could in principle violate the first invariant, so a RTSJ compliant virtual machine is required to check every store dynamically to ensure it does not create an incoming reference. Thus any assignment statement can potentially throw a runtime error. The second invariant, that scopes have a single parent, must also be checked dynamically, and any operation that attempts to enter a scope may also throw an exception. The third invariant makes it hard to tell when a scope is deallocated, and thus harder to predict a program's memory usage.

We propose to address the above mentioned problems with a new programming mechanism called *Scoped Types*, designed to support safe scoped memory programming in concurrent systems. Our goal is to devise a solution which can be adopted without major changes to the Java language or the tools that are used to write and run Java programs (e.g. compilers, development environment, and preprocessors). The underlying motivation is that we want to keep the cost of adopting our proposed model low. In fact, we have tried to retain the option of translating programs using Scoped Types into plain Java programs and running them on a vanilla Real-time Java virtual machine.

To appeal to practicing Java programmers, our proposal requires minimal syntactic overhead. With Scoped Types, the dynamic scoped memory area hierarchy of the RTSJ is captured by refactoring the program into packages such that objects that are meant to live in the same scope will be defined in the same package and annotated with the keyword `@scoped`. Scopes are reified by instances of portal class, these are Java classes annotated with keyword `@portal`. The definition and behavior of scoped classes is constrained by seven simple rules which taken together ensure that assignment errors will never occur and scope entry operations will always succeed. Furthermore, deallocation of scope will never lead to a dangling pointer.

The contributions of this paper are the following:

1. We present a new programming model for RTSJ which provides static correctness guarantees while remaining simple and eschewing the need to modify Java in a significant way.

2. To gain confidence in the proposed model, we formalize the programming rules in terms of a type system for core object calculus called SJ.

3. We prove soundness of the type system, as well as establish other safety properties.

In related work Boyapati *et.al.* have proposed an ownership based type system for providing similar static guarantees for a RTSJ-like language [6]. While their system is strictly more expressive, it is also much more complex and requires changes to the compiler and entire Java tool chain. We believe that starting with a less powerful but simpler basis is a good way to proceed as it may be easier to convince users to switch if the perceived cost is low. Furthermore, it is still unclear how much expressive power is actually needed to express most interesting RTSJ programs. For example, in Ravenscar-Java (based on the Real-time Ada industry standard) Wellings decided to forbid nested scopes entirely on the grounds that it would render the programming model simpler to use and the programs easier to validate [24].

To the best of our knowledge, the proof of correctness presented in this paper is the first proof addressing safety of scoped allocation in the context of concurrent object calculus. At the time of this writing, Boyapati's system has not yet been shown sound [6]. Work on regions for functional languages did not take concurrency into account [34, 35].

The paper is organized as follows. Section 2 illustrates the use of RTSJ scope memory areas through a motivating example. Section 3 introduces Scoped Types informally using Java syntax and revisits the example of Section 2. Section 4 presents the SJ calculus and the scoped type system. The proof of soundness is given in Section 5. Section 6 discusses related work, and finally Section 7 concludes.

2. AN EXAMPLE: A REAL-TIME COLLISION DETECTOR

To illustrate the usage of scopes, we present the example of an aircraft collision detection algorithm [29]. Collision detection is performed by a single real-time thread which receives a series of *frames* containing aircraft call signs along with positions and direction vectors. The output of the algorithm is a warning each time any pair of aircraft are on a collision course. We have implemented two versions of the algorithm — one in plain Java and one in RTSJ. The Java implementation is 2500 lines of code of which fewer than 200 lines had to be adapted to make the program RTSJ compliant.

Collision detection is performed iteratively. A frame object containing a number of plane objects is received from the sensors once per iteration of the main run loop. The contents of the frame are used to update a state table and then compute collision vectors. The RTSJ implementation of the algorithm uses four scopes, the distinguished `ImmortalMemory` and `HeapMemory`, along with two user-defined instances of `ScopedMemory`. Fig. 2 illustrates the memory structure of

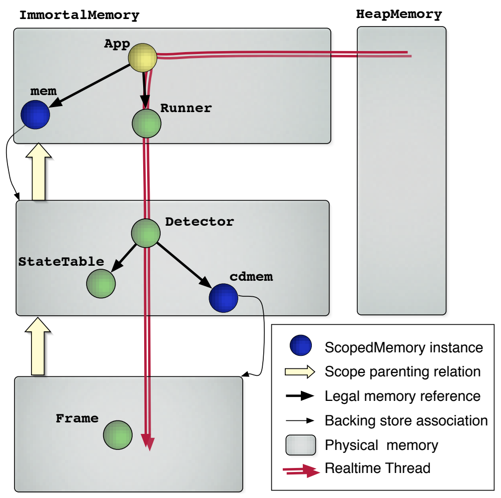


Figure 2: Scopes in the example application. The main `App` object is allocated in immortal memory `imm`. Application stable state is held in the scoped area `mem`, per-iteration objects are allocated in `cdmem`.

```
class App extends NoHeapRealtimeThread {
   static void main() {
      imm = ImmortalMemory.instance();
      app = (App) imm.newInstance( App.class));
      app.start(); }
   void run() {
      LTMemory mem = new LTMemory( ...);
      mem.enter( new Runner() ); } }

class Runner implements Runnable {
   void run() {
      LTMemory cdmem = new LTMemory( ...);
      Detector cd =
         new Detector( new StateTable() );
      while ( true )
         cdmem.enter( cd); } }
```

```
class Detector implements Runnable {
   StateTable state;
   void run() {
      Frame frame = receiveFrame();
      pos_in_table = state.get( frame.getAircraft());
      if (pos_in_table == null) {
         mem = MemoryArea.getMemoryArea( this);
         Aircraft new_plane =
            mem.newInstance( Aircraft.class);
         plane.update( new_plane);
         pos_in_table = mem.newInstance( Position.class);
         state.put( new_plane, pos_in_table); }
      pos_in_table.update( frame.getPosition());
   }
}
```

Figure 1: The main method of the application is used to bootstrap the real-time task. The run method of App is used to set up the application's stable store. The Runner class holds the application's main loop. All methods are public unless stated otherwise. Class LTMemory is a kind of scoped memory which guarantees linear time allocation. Class NoHeapRealtimeThread is the parent class of all hard real-time thread classes.

the program. While the main object, `App`, of the application is created in immortal memory, none of the other objects should be allocated there, as objects in immortal memory are never deleted. Thus, the first action of the `App.run` method is to create and enter a new scope, `mem`, that will be used contain the program's stable storage. A second scope, `cdmem`, is used for temporary storage, so that all the temporary objects are deleted at the end of each iteration.

Fig. 1 illustrates the main points of the algorithm. The `App.main` method is responsible for starting a new real-time thread (`App` extends `NoHeapRealtimeThread`). As real-time threads may not execute within the heap, the first action that is performed by that method is to enter immortal memory. Note the use of reflection (`newInstance`) to create objects in different scopes. Then the `App.run` method creates a new scope to hold the application's stable store (all state that must be preserved between iterations is stored in the instance of class `StateTable`) and starts an instance of `Runner` in the newly created scope.

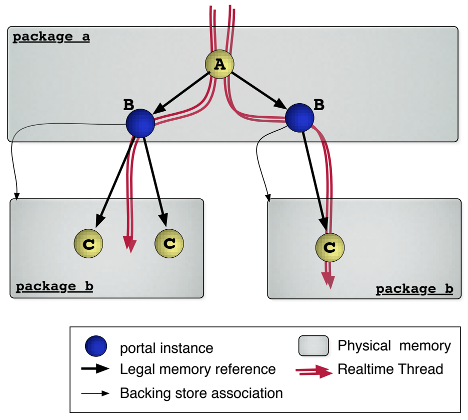
The `Runner.run` method is an example of the *scoped run loop* pattern [29]. The method starts by creating a scope, `cdmem`, to hold temporary objects. Then it repeatedly executes the `Detector.run` method within `cdmem`. Since there is no other thread contending for that scope, after each iteration the scope is cleared. We remark that the `ScopedMemory` object itself remains intact between invocation, as does the `Detector` – both are allocated in the `mem` area which is not reclaimed for the lifetime of the application.

As can be seen from Fig. 1, although perhaps simple in theory, RTSJ Scoped Memory is complex in practice. For example, in each iteration of the run loop a new frame object is created along with an aircraft and a position. These objects are all allocated in the `cdmem` scope, whereas the state table is in the parent `mem` scope. In order to store a newly detected plane in the state table, the program has to reflectively create instances of `Aircraft` and `Position` in the correct (`mem`) scope. For another example, consider that the `Aircraft.update` method (not shown here), takes an aircraft as argument and copies the information out of itself into its argument. We were forced to use this tortured design so that we can copy data allocated in the inner `cdmem` scope into an object in the out stable store `mem` scope.

Discussion. This complex explanation shows that a large amount of important information about this example is implicit in the RTSJ code. The memory scopes within which variables are allocated (and therefore to which they can refer) are not recorded in the text of the program; there is no information about the scope that a particular instance of the `Aircraft` class is stored in, for example. This means that a minor typographical error could go undetected by the compiler, and then cause a runtime failure during rare circumstances as the program is run — such as when the program actually detects a collision. Similarly, the nesting relationship between the `imm`, `mem`, and `cdmem` scopes is implicit in the code: if the `run` method of the `Detector` class attempted to reenter the `mem` scope, the program would suffer a `ScopedCycleException`. Finally, the programmer intends that all the objects contained within the `cdmem` scope will be discarded after each iteration, but this is not in any way obvious from the code of the program.

3. REAL-TIME PROGRAMMING WITH SCOPED TYPES

To make real-time Java programming more reliable and predictable, we have developed a type system that can statically guarantee the absence of reference assignment errors, enforce the single parent rule for scopes, and ensure that



```
package a;
    @scoped class A {
        ...
    }

package a.b;
    @portal class B {
        ...
    }
    @scoped class C {
        ...
    }
```

Figure 3: A small example of a program written with Scoped Types. The program's static structure consists of two packages `a` and `a.b`. At runtime to instance of the portal class `B` are created, thus giving rise to two distinct scopes. Notice that portal objects, like `ScopedMemory` instances in the RTSJ, are allocated in the parent scope. Overall, the code is shorter than the RTSJ version and makes explicit the allocation context of objects.

there can be no references into the contents of a scope when it is scheduled to be discarded. Our proposal, referred to as *Scoped Types*, is intended as a replacement for scoped memory in a real-time Java virtual machine. Scoped Types require the addition of only two modifiers to the language, no compiler changes, and minimal extra support from the underlying virtual machine.

Scoped Types are envisioned primarily for the hard real-time kernels of RTSJ applications. The current state-of-the art in large commercial systems is that the hard real-time component is typically 1% of the overall system, but the code is rewritten from scratch for every release — reuse is still viewed as a risk in most of the real-time community. In this setting we believe that simplicity and clarity are the most important requirements for any programming model. For this reason, all choices in the design of Scoped Types were made in favor of simplicity.

Language extensions. Our model distinguishes between two kinds of classes in a real-time Java program: *scoped classes* which are allocated within a particular memory scope, and *portal classes* which reify memory scopes. Most of the objects in the program are instances of scoped classes, and are allocated in memory scopes. Instances of portal classes turn scopes into first-class entities: threads enter memory scopes by invoking methods of portal objects, and exit scopes when these calls return. The key observation is that an object allocated in a scoped memory area can only be used in that scope and its nested subscopes. Thus, we statically restrict the accessibility of a scoped class to the classes whose instances are allocated in the same or nested scopes.

Integration with Java. To minimize the changes to the language (at least to its syntax and to the tool processing the language) we take advantage of existing concepts, such as visibility rules and access modifiers, to integrate Scoped Tscopeypes with Java. Scoped and portal classes are declared by appending the respective modifiers to class declarations (`@scoped` and `@portal`), no other annotations are needed. These annotations are consistent with the Metadata JSR, and will be recognized by Java compilers. An alternative that does not rely on Metadata is to use an idiom based on marker interfaces [38]. We call packages that contain scoped types *scoped packages*. Scoped packages are the unit of protection and of allocation. Each scoped package is the static representation a family of memory scopes and defines the types of objects that may be allocated in these scopes. We use nested packages to represent potentially nested memory scopes: a memory scope created by some portal class in a scoped package can only contain nested subscopes defined by portal classes in immediate subpackages. Instances of classes defined in top-level package are allocated in immortal memory.

Dynamics. While scoped packages describe the static structure of a runtime application, restricting applications to have one single instance of each scope (and thus matching the static package hierarchy) prevents some useful programming idioms. Thus at runtime every *instance* of a portal class corresponds to a new memory scope. So an application that creates two instances of some portal gets two distinct scopes which can be used independently. The type system guarantees that references across these scopes cannot arise, thus objects allocated within them can safely be reclaimed at different times. A scope's portal is the *only* object from the scoped package that is visible in the parent package. In fact, portals are allocated in their parent scope, just as RTSJ `ScopedMemory` objects are allocated in an enclosing area. The current allocation context is *always* defined by the package in which the current class was defined (where "current class" is taken to mean the class of the receiver of the executing method). Changing allocation context is thus as simple as calling a method of an object living in a different scope. Concurrency comes in quite naturally – multiple threads execute in the same scope if they invoke a method on the same portal. The implementation keeps track of the number of threads in a scope by a simple reference counting scheme. Just as in RTSJ, objects within a scope can be reclaimed when the last thread exits. Fig. 3 illustrates these concepts.

Static guarantees. Our model imposes some static constraints on the accessibility of classes. We require that scoped classes in a package are accessible only to the classes defined that package and its subpackages, while portal classes are only accessible to classes defined in their immediate parent package. In other words, classes are *not* able to access classes in inner nested subpackages (other than the portals of their immediate subpackages). These constraints ensure that a package's portal classes form an encapsulation boundary for classes outside that package: scoped classes, and classes in subpackages are inside that encapsulation boundary [27]. More importantly, they ensure that objects allocated in one scope may never have incoming references to objects allocated in inner scopes, and thus that `IllegalAssignmentError`s can never happen. Threads can only enter the scopes defined in some package (by calling methods on portal classes in that package) from the code in the immediate super-package. This ensures that the hierarchy of memory scopes always follows the same hierarchy as the corresponding packages, enforcing the single parent rule and preventing `ScopedCycleException`s.

Scoped Type Confinement Rules. Scoped Types' static guarantees are enforced by the following syntactic rules that must hold for all scoped and portal classes. Rules $\mathcal{C}1$, $\mathcal{C}2$, and $\mathcal{C}3$ bind scoped classes, while Rules $\mathcal{S}1$ to $\mathcal{S}3$ bind portal classes. Besides the visibility constraints of Rules $\mathcal{C}1$ and $\mathcal{S}1$, we also require that ($\mathcal{C}2$) references of scoped type can not be widened to types in other packages while ($\mathcal{S}2$) the references of portal types cannot be widened to other types. The restrictions on reference widening help us to track references by their static types.

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

These rules are similar in spirit to the confinement rules presented in [38]. The type system presented in the next section formalizes these intuitive rules.

Restrictions. Scoped Types do restrict the set of valid programs, while they do not require changes to the syntax they do introduce significant changes to the programming model. To start with, while an instance of a scoped class may extend an arbitrary class, none of the methods defined outside of the scoped package can be invoked. Furthermore the restrictions on widening mean that the reuse of library classes will be limited. We defend these choices by remarking that most library code has not been designed for being used in a real-time setting, and libraries can be used freely in non-RT parts of the program.

3.1 The Example Revisited

Scoped Types simplify programming within nested memory scopes. We can rewrite the collision detector example to use Scoped Types with very few changes. We first need to define three packages to model the three scopes of the original application. This is because in Scoped Types, the programmers have to choose the package within which each class should be statically *defined*, rather than deciding where instance should be dynamically *allocated*, as in RTSJ.

The scoped version of the program, shown in Fig. 4 and the code in Fig. 5, consists of three packages, `imm`, `imm.mem` and `imm.mem.cdmem` mirroring the dynamic scope hierarchy of the algorithm. The class `Main` is the only class that executes in immortal memory, and its only purpose is to create an instance of the `App` class, which is the portal of the `imm.mem` package. The `App` class, once started, will then allocate an instance of the `Detector` class, which is the portal for the `imm.mem.cdmem` scope. The run loop again boils down to calling the detector's `run` method. The program's sta-

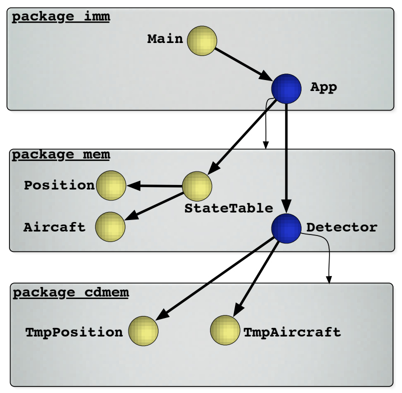


Figure 4: The reference patterns of scoped and portal objects in the Scoped Type version of the example. The only references allowed to go from a parent package to a child are references originating from the portal. The portal object is a dominator for all scoped types in its package and subpackages. Note that although the figure does not show it, references from child packages to their parents are allowed.

```
package imm;

   @scoped class Main {
      static void main( ) { new App().start();}}

package imm.mem;

   @portal final class App
         extends NoHeapRealtimeThread {
      void run() {
         cd = new Detector();
         state = new StateTable();
         key = new Aircraft();
         while ( true ) cd.run( state, key); } }

   @scoped class StateTable ...
   @scoped class Aircraft ...
   @scoped class Position ...
```

```
package imm.mem.cdmem;

   @portal final class Detector {
      void run( StateTable state, Aircraft key) {
         frame = receiveFrame();
         TmpAircraft plane = frame.getAircraft();
         plane.update( key);
         pos_in_table = state.get( key);
         if ( pos_in_table == null )
            state.put( plane.copy(),
                       frame.getPosition().copy());
         else
            frame.getPosition().update(pos_in_table);
   }  }

   @scoped class TmpAircraft ...
   @scoped class TmpPosition ...
   @scoped class Frame ...
```

Figure 5: The collision detector example with Scoped Types. The program is split into three packages representing the different scopes used in that program. All support classes (*e.g.* Aircraft) are defined in the appropriate scope.

ble state is held in the `imm.mem` package, and is composed of instances of the `StateTable`, `Aircraft`, and `Position` classes. Per-iteration temporary objects are stored in the `cdmem` package and consist of `TmpAircraft`, `TmpPosition` and `Frame`.

Notice that with Scoped Types it is impossible to confuse planes in the inner `imm.mem.cdmem` scope with planes in the stable `imm.mem` scope, as they are represented by different types. A `copy` method is implemented in `TmpAircraft` to create a `Aircraft` object that *must*, by definition, be allocated in the parent scope. Similarly, since the state table is allocated in `imm.mem`, the types in `imm.mem.cdmem` are not accessible to it. Thus, we cannot use a `TmpAircraft` object as the `key` to find out whether a plane is already stored in the table, and we use an `Aircraft` object instead. The `update` method of a `TmpAircraft` object refreshes the `key` with the information about the current plane.

In this way, Scoped Types statically maintains the invariants that RTSJ checks dynamically. By statically associating scoped objects to their defining packages, Scoped Types can ensure that incoming references are never created. Similarly, by modeling nested scopes with nested packages, Scoped Types ensure that scopes will never form cycles. Finally, by statically tracking the objects contained within each scope, Scoped Types ensure that it is safe to discard all the objects in a scope once the last thread has left it.

The expressive power of Scoped Types, as presented here, is strictly less than that of approaches based on ownership types [6]. We can easily regain some expressive power by, for example, using generics *à la* Generic CFJ [38], to parameterize classes by their scope. With generics, there would only be a single `Aircraft` class instantiated in different scopes. While we have investigated such an extension, it is unclear if the added expressive power would justify the increase in complexity.

4. THE SJ CALCULUS

To gain confidence in the programming model underlying our proposal, we introduce the SJ calculus, a sparse imperative and concurrent object calculus, modeled after Featherweight Java [22], in which scopes are first-class values. SJ formalizes the type confinement rules of Scoped Type in terms of a type system. Our proof of type soundness gives us the guarantee that confinement can not be breached during execution of a well type program. We can then proceed to prove that the shape of the scope hierarchy is restricted to tree. And, finally, that deallocation of a scope will not result in dangling references.

SJ is a simple object calculus, to keep the semantics concise we have omitted some features that are not essential to the main results. These features include static methods (we could model classes by singleton objects allocated in the topmost scope), synchronization, access modifiers and down-cast expressions. Unlike other systems such as [6], down-cast are not problem in SJ because the restrictions imposed by the Scoped Type Confinement rules ensure that any variable of a scoped type always refer to an instance of that type or of a subtype defined in the same scoped package. While there is no explicit up-cast expression in SJ, up-casts arise due to the usual implicit widening occurring in assignment and method invocation.

4.1 Syntax and Types

The syntax of the SJ calculus, Figure 6, draws on our previous work [38] which, in turn, was based on Featherweight Java (FJ). SJ has two kinds of class declarations, *scoped classes* and *portal classes*, the former annotated with a `scoped` and the latter with a `portal`. Classes belong to packages, which can be nested in an arbitrary package hierarchy. Each package may contain a mixture of scoped and portal classes. We add an assignment expression and an expression for creating a new thread of control. Finally, we add a *reset* expression, which clears the fields of a portal object if the objects is not used by any threads. Reseting a portal corresponds to deallocating a scope in RTSJ, the operation was added to model GC, but is interesting in its own right as we observed in [29].

We take metavariables $\texttt{C}, \texttt{D}$ to range over classes, $\texttt{M}$ to range over methods, $\texttt{K}$ over constructors, and $\texttt{f}$ and $\texttt{x}$ to range over fields and parameters, respectively. We also use $\texttt{P}$ for package names, $\texttt{e}$ for expressions and ℓ for memory references. We use over-bar to represent a finite ordered sequence, for instance, $\overline{\texttt{f}}$ represents $\texttt{f}_1\,\texttt{f}_2 \ldots \texttt{f}_\texttt{n}$. The term $\overline{\texttt{l}}\,.\,\overline{\texttt{l}'}$ denotes sequence concatenation. The calculus has a call-by-value semantics. The expression $[^{\texttt{v}}/_{\texttt{v}_\texttt{i}}]\overline{\texttt{v}}$ yields a sequence identical to $\overline{\texttt{v}}$ except in the ith field which is set to $\texttt{v}$. We use the usual dot notation to represent nested packages. That is, the package $\texttt{p.q}$ is a subpackage of $\texttt{p}$. The presentation of the calculus inherits some of the syntactic oddities of FJ, so $\overline{\texttt{e}}\,\texttt{e}$ is a short hand for $\texttt{e}_1 \ldots \texttt{e}_n\,\texttt{e}$, and $\texttt{m}(\overline{\texttt{C}}\,\overline{\texttt{x}})$ stands for $\texttt{m}(\texttt{C}_1\,\texttt{x}_1, \ldots, \texttt{C}_n\,\texttt{x}_n)$.

We assumes the existence of a class table CT containing the definitions of all classes. For simplicity, we assume that a class name $\texttt{C}$ uniquely identifies the entry $CT(\texttt{C})$ in the class table (this is not a real restriction, the motivation for this choice was to avoid cluttering rules with package names). We can find out the name of the package that contains $\texttt{C}$ by looking up the defintion in $CT(\texttt{C})$.

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

Figure 6: Syntax of the SJ calculus.

The subtyping rules of Figure 7 are standard. We define the partial order $\preceq$ to limit the variables that can refer to scoped objects and portals; $\texttt{C} \preceq \texttt{C}'$ is defined if either $\texttt{C}$ is a portal type and $\texttt{C} = \texttt{C}'$, or $\texttt{C}$, $\texttt{C}'$ are scoped and belong to the same package.

4.2 Semantics

In SJ, each portal object represents a distinct scoped memory area and whenever a portal is reset all of the objects that were allocated within the associated scope are reclaimed. While the package hierarchy imposes a static structure on scopes, portal objects allow multiple scope instances to be created at runtime. The main restriction imposed by SJ is that a portal can only allocate objects of scoped classes belonging to the same package and portals defined in immediate subpackages. When this restriction is combined with confinement invariants that prevents portal objects leaking from their parent package, we obtain the key property for scoped memory management, namely the restriction that threads enter scopes in the same order as the nesting relation of the packages containing the portal classes.

Subtyping:

$$
\texttt{C} <: \texttt{C} \qquad \frac{\texttt{C} <: \texttt{C}' \quad \texttt{C}' <: \texttt{C}''}{\texttt{C} <: \texttt{C}''}
$$

$$
\frac{CT(\texttt{C}) = \circ\ \texttt{class P.C} \lhd \texttt{C}'\ \{\ \ldots\ \}}{\texttt{C} <: \texttt{C}'}
$$

Scope-safe subtyping:

$$
\texttt{C} \preceq \texttt{C}' \text{ iff } \texttt{C} <: \texttt{C}' \text{ and } \begin{cases} \text{either } \texttt{C}, \texttt{C}' \text{ are scoped types} \\ \quad \text{and in the same package} \\ \text{or } \texttt{C} \text{ is a portal type and } \texttt{C} = \texttt{C}' \end{cases}
$$

Allocation:

$$
\begin{array}{l}
\text{Suppose } \sigma(\ell) = \texttt{C}_0^{\ell'}(\overline{\texttt{v}}) \\
\mathit{allocScope}_\sigma(\texttt{C}, \ell) = \ell \text{ if } \texttt{C}_0 \text{ is a portal type} \\
\quad \text{and } \begin{cases} \text{either } \texttt{C} \text{ is a scoped type and} \\ \quad \texttt{C}, \texttt{C}_0 \text{ in the same package} \\ \text{or } \texttt{C} \text{ is a portal type and} \\ \quad \texttt{C} \text{ in the immediate subpackage of } \texttt{C}_0 \end{cases} \\
\mathit{allocScope}_\sigma(\texttt{C}, \ell) = \mathit{allocScope}_\sigma(\texttt{C}, \ell') \quad \text{otherwise}
\end{array}
$$

Evaluation context:

$$
\begin{array}{lcl}
E[\circ] & ::= & \circ \mid E[\circ].\texttt{m}(\overline{\texttt{e}}) \mid \texttt{v.m}(\ldots, \texttt{v}_{\texttt{i}-1}, E[\circ], \texttt{e}_{\texttt{i}+1} \ldots) \\
 & \mid & E[\circ].\texttt{f}_\texttt{i} \mid E[\circ].\texttt{f}_\texttt{i} := \texttt{e} \mid \texttt{v.f}_\texttt{i} := E[\circ] \mid \texttt{reset}\,E[\circ]
\end{array}
$$

Scope reference counts:

$$
\mathit{refcount}(\ell, t[\,\overline{\ell\,\texttt{e}}\,] \mid P') = \mathit{count}_\ell(\overline{\ell}) + \mathit{refcount}(\ell, P')
$$

$$
\mathit{refcount}(\ell, \emptyset) = 0 \qquad \mathit{count}_\ell(\emptyset) = 0
$$

$$
\mathit{count}_\ell(\overline{\ell}\,.\,\ell) = 1 + \mathit{count}_\ell(\overline{\ell}) \qquad \mathit{count}_\ell(\overline{\ell}\,.\,\ell') = \mathit{count}_\ell(\overline{\ell})
$$

Figure 7: Auxiliary definitions.

As in Featherweight Java, the semantics assumes the existence of a class table containing the definitions of all classes. We had to add a store σ and a collection of threads $\overline{P}$ labeled by distinct identifiers $\overline{t}$. Objects are of the form $\texttt{C}^\ell(\overline{\texttt{v}})$, where $\texttt{C}$ is a class, $\overline{\texttt{v}}$ the value of the fields, and ℓ the portal of the scope in which it was allocated. The store σ is a sequence, $\overline{\texttt{C}^\ell(\overline{\texttt{v}})}$, of objects, each denoted by a distinct label ℓ_i. Fig. 7 defines a number of auxiliaries relations.

The partial function $\mathit{allocScope}_\sigma(\texttt{C}, \ell)$ retrieves the allocation scope for an object of the type $\texttt{C}$ when the current receiver object is ℓ. Our type system ensures that all the scopes form a tree. Intuitively, $\mathit{allocScope}_\sigma(\texttt{C}, \ell)$ searches the scope tree upward starting from ℓ or the scope of ℓ until it finds a scope ℓ' of the type $\texttt{C}'$, which is in the same package as $\texttt{C}$ if $\texttt{C}$ is scoped and is in the immediate super-package of $\texttt{C}$ if $\texttt{C}$ is a portal. For some $\texttt{C}$ and ℓ, $\mathit{allocScope}_\sigma(\texttt{C}, \ell)$ is not defined.

An evaluation context, Fig. 7, is an expression $E[\circ]$ with a hole that can be filled in with another expression of proper

$$\frac{\sigma(\ell) = \mathtt{C}^{\ell'}(\overline{\mathtt{v}}) \quad \mathit{fields}(\mathtt{C}) = (\overline{\mathtt{C}}\ \overline{\mathtt{f}})}{\sigma, \ell_0\ \ell.\mathtt{f}_\mathtt{i} \ \rightarrow\ \sigma, \ell_0\ \mathtt{v}_\mathtt{i}} \qquad \text{(R-FIELD)}$$

$$\frac{\begin{array}{c}\sigma(\ell) = \mathtt{C}^{\ell'}(\overline{\mathtt{v}}) \quad \mathit{fields}(\mathtt{C}) = (\overline{\mathtt{C}}\ \overline{\mathtt{f}}) \\ \sigma' = \sigma[\ell \rightarrow \mathtt{C}^{\ell'}([^{\mathtt{v}}/_{\mathtt{v}_\mathtt{i}}\overline{\mathtt{v}}])]\end{array}}{\sigma, \ell_0\ \ell.\mathtt{f}_\mathtt{i} := \mathtt{v} \ \rightarrow\ \sigma', \ell_0\ \mathtt{v}} \qquad \text{(R-UPDATE)}$$

$$\frac{\begin{array}{c}\mathit{allocScope}_\sigma(\mathtt{C}, \ell_0) = \ell' \quad \ell \text{ fresh} \\ \mathit{init}(\mathtt{C}) = \overline{\mathtt{new}\ \mathtt{D}()} \quad \sigma'' = \sigma[\ell \rightarrow \mathtt{C}^{\ell'}(\overline{\mathtt{null}})] \\ \sigma'', \ell\ \mathtt{new}\ \mathtt{D}_1() \ \rightarrow\ \sigma_1, \ell\ \mathtt{v}_1 \\ \ldots \\ \sigma_{\mathtt{n}-1}, \ell\ \mathtt{new}\ \mathtt{D}_\mathtt{n}() \ \rightarrow\ \sigma_\mathtt{n}, \ell\ \mathtt{v}_\mathtt{n} \\ \sigma' = \sigma_\mathtt{n}[\ell \rightarrow \mathtt{C}^{\ell'}(\overline{\mathtt{v}})]\end{array}}{\sigma, \ell_0\ \mathtt{new}\ \mathtt{C}() \ \rightarrow\ \sigma', \ell_0\ \ell} \qquad \text{(R-NEW)}$$

$$\frac{\mathtt{e} \neq \mathtt{reset}\ \ell,\ \ell.\mathtt{m}(\overline{\mathtt{v}}) \quad \sigma, \ell_0\ \mathtt{e} \ \rightarrow\ \sigma', \ell_0\ \mathtt{e}'}{\sigma, \ell_0\ E[\mathtt{e}] \ \rightarrow\ \sigma', \ell_0\ E[\mathtt{e}']} \qquad \text{(R-CONG)}$$

$$\frac{\begin{array}{c}\sigma(\ell) = \mathtt{C}^{\ell'}(\overline{\mathtt{v}}) \quad \mathit{init}(\mathtt{C}) = \overline{\mathtt{new}\ \mathtt{D}()} \\ \sigma, \ell\ \mathtt{new}\ \mathtt{D}_1() \ \rightarrow\ \sigma_1, \ell\ \mathtt{v}'_1 \ldots \\ \sigma_{n-1}, \ell\ \mathtt{new}\ \mathtt{D}_\mathtt{n}() \ \rightarrow\ \sigma_\mathtt{n}, \ell\ \mathtt{v}'_\mathtt{n} \\ \sigma' = \sigma_\mathtt{n}[\ell \rightarrow \mathtt{C}^{\ell'}(\overline{\mathtt{v}'})]\end{array}}{\sigma, \ell_0\ \mathtt{reset}\ \ell \ \rightarrow\ \sigma', \ell_0\ \ell} \qquad \text{(R-RESET)}$$

$$\frac{\sigma(\ell) = \mathtt{C}^{\ell'}(\overline{\mathtt{v}'}) \quad \mathit{mbody}(\mathtt{m},\ \mathtt{C}) = (\overline{\mathtt{x}},\ \mathtt{e})}{\sigma, \ell_0\ \ell.\mathtt{m}(\overline{\mathtt{v}}) \ \rightarrow\ \sigma, \ell\ [^{\overline{\mathtt{v}}}/_{\overline{\mathtt{x}}},\ ^{\ell}/_{\mathtt{this}}]\mathtt{e}} \qquad \text{(R-INVK)}$$

Figure 8: Expression evaluation.

$$\frac{\begin{array}{c}P = P'' \mid t[\,\overline{\ell\,\mathtt{e}}\,.\,\ell\,\mathtt{e}\,] \\ P' = P'' \mid t[\,\overline{\ell\,\mathtt{e}}\,.\,\ell\,\mathtt{e}'\,] \\ \mathtt{e} \neq \mathtt{reset}\ \ell',\ \ell'.\mathtt{m}(\overline{\mathtt{v}}) \qquad \sigma, \ell\ \mathtt{e} \ \rightarrow\ \sigma', \ell\ \mathtt{e}'\end{array}}{\sigma, P \Rightarrow \sigma', P'} \qquad \text{(G-STEP)}$$

$$\frac{\begin{array}{c}P = P'' \mid t[\,\overline{\ell\,\mathtt{e}}\,.\,\ell\,\mathtt{e}\,] \\ P' = P'' \mid t[\,\overline{\ell\,\mathtt{e}}\,.\,\ell\,E[\ell^{\mathtt{ret}}]\,.\,\ell'\,\mathtt{e}'\,] \\ \mathtt{e} = E[\mathtt{e}_0] \qquad \mathtt{e}_0 = \ell'.\mathtt{m}(\overline{\mathtt{v}}) \\ \sigma, \ell\ \mathtt{e}_0 \ \rightarrow\ \sigma, \ell'\ \mathtt{e}'\end{array}}{\sigma, P \Rightarrow \sigma, P'} \qquad \text{(G-ENTER)}$$

$$\frac{\begin{array}{c}P = P'' \mid t[\,\overline{\ell\,\mathtt{e}}\,.\,\ell\,\mathtt{e}\,.\,\ell'\,\mathtt{v}\,] \\ P' = P'' \mid t[\,\overline{\ell\,\mathtt{e}}\,.\,\ell\,E[\mathtt{v}]\,] \quad \mathtt{e} = E[\ell^{\mathtt{ret}}]\end{array}}{\sigma, P \Rightarrow \sigma, P'} \qquad \text{(G-RETURN)}$$

$$\frac{\begin{array}{c}P = P'' \mid t[\,\overline{\ell\,\mathtt{e}}\,.\,\ell\,\mathtt{e}\,] \\ P' = P'' \mid t[\,\overline{\ell\,\mathtt{e}}\,.\,\ell\,E[\ell^{\mathtt{th}}]\,] \ \mid\ t'[\,\overline{\ell\,\ell^{\mathtt{ret}}}\,.\,\ell\,\mathtt{e}_0\,] \\ \mathtt{e} = E[\mathtt{spawn}\,\mathtt{e}_0] \qquad t' \text{ fresh}\end{array}}{\sigma, P \Rightarrow \sigma, P'} \qquad \text{(G-SPAWN)}$$

$$\frac{\begin{array}{c}P = P'' \mid t[\,\overline{\ell\,\mathtt{e}}\,.\,\ell\,E[\mathtt{reset}\ \ell']\,] \\ P' = P'' \mid t[\,\overline{\ell\,\mathtt{e}}\,.\,\ell\,E[\ell']\,] \\ \text{if } \mathit{refcount}(\ell', P) \neq 0 \text{ then } \sigma' = \sigma \\ \text{else } \sigma, \ell\ \mathtt{reset}\ \ell' \ \rightarrow\ \sigma', \ell\ \ell'\end{array}}{\sigma, P \Rightarrow \sigma', P'} \qquad \text{(G-RESET)}$$

Figure 9: Computation rules.

type. Evaluation contexts do not include the body of `spawn`.

The dynamics semantics of SJ is split in two: expression evaluation rules given in Fig. 8 and the computation rules in Fig. 9.

Expression rules. These evaluation rule consider only operations performed within a single thread. The evaluation relation has the form $\sigma, \ell\ \mathtt{e} \ \rightarrow\ \sigma', \ell'\ \mathtt{e}'$ where σ is the initial store, ℓ is the reference to object currently executing, and `e` the expression to evaluate. The reduction rules field select (R-FIELD), field update (R-UPDATE), and method invocation (R-INVK) are not surprising. The rule (R-CONG) allows for the reduction of a subexpression within an evaluation context.

The instantiation rule (R-NEW) must ensure that the class of the object about to be created, `C`, can be instantiated in the current scope (as defined by $\mathit{allocScope}_\sigma(\mathtt{C}, \ell_0) = \ell'$). The fields of `C` must also be initialized, though not in the current scope but rather in the scope of the newly allocated object (this only matters if `C` is a portal). Finally the store is updated with a fresh reference ℓ bound to the newly allocated object. The helper function $\mathit{init}(\mathtt{C})$ is defined in Figure 12.

The reset rule (R-RESET) clears all fields of the target object. This has the effect to ensure that all objects previously in the scope are now unreachable. The target object fields are set to newly allocated default values. In practice, the real-time Java programmers don't have to explicitly write such reset expressions as a Java virtual machine should be able to maintain reference count to a portal and implicitly perform reset operations.

Computation rules. The computation rules of the form $\sigma, P \Rightarrow \sigma', P'$ where σ is a store and P a set of threads. Each thread $t[\,\overline{\ell\,\mathtt{e}}\,]$ in P has a distinct label t and a runtime call stack which is a list of receiver-expression pairs $\ell, \mathtt{e}$.

Rule (G-STEP) is simple, it picks one thread for execution and evaluates the expression `e` on the top of the thread's stack. Note that this rule applies when `e` is not a method invocation or reset expression.

Rule (G-ENTER) evaluates a thread $t[\,\overline{\ell\,\mathtt{e}}\,.\,\ell\,\mathtt{e}\,]$ containing a method call $\mathtt{e} = E[\mathtt{e}_0]$ and $\mathtt{e}_0 = \ell'.\mathtt{m}(\overline{\mathtt{v}})$. It creates a new stack frame for the body of the method, $\mathtt{e}'$, and introduces a place holder, $\ell^{\mathtt{ret}}$ for the return value of the call in the original expression. Thus the result is a frame $\overline{\ell\,\mathtt{e}}\,.\,\ell\, E[\ell^{\mathtt{ret}}]\,.\,\ell'\,\mathtt{e}'$. Note that $\ell^{\mathtt{ret}}$ does not correspond to any actual object, there can only be one occurrence of $\ell^{\mathtt{ret}}$ per frame, and that (G-Return) ensure that $\ell^{\mathtt{ret}}$ will never be manipulated during expression evaluation.

If the expression on the top of a thread's stack is reduced to a value $\mathtt{v}$, then by Rule (G-Return), the thread can pop the stack frame and continue execution with $\mathtt{v}$ as the resulted value of a method call.

Rule (G-Spawn) evaluates a thread $t[\overline{\ell\,\mathtt{e}}\,.\,\ell\,\mathtt{e}]$ containing a spawn expression $\mathtt{e} = E[\mathtt{spawn}\,\mathtt{e}_0]$. The value of the spawn expression in $\mathtt{e}$ is the distinguished $\ell^{\mathtt{th}}$ which is a unique, global reference to an object of class `Thread` and we assume that $\ell^{\mathtt{th}}$ is allocated in immortal memory. A new thread t' is created to evaluate $\ell\,\mathtt{e}_0$. The new thread is started with a call stack $\overline{\ell\,\ell^{\mathtt{ret}}}$ that matches the call stack of the original thread t to ensure that scope reference counts are accurate.

Rule (G-Reset) clears the fields of a portal ℓ' when no thread is using that portal (i.e. when $\mathit{refcount}(\ell', P) = 0$). For simplicity, the fields of a portal are reset to default value explicitly by a *reset* expression of the form `reset` ℓ' and if the reference count of ℓ' is not zero, then the fields of ℓ' are not cleared (this makes *reset* nonblocking to avoid deadlock).

4.3 Type Rules

The typing rules are shown in Figure 10 and 11 and the related auxiliary functions are defined in Figure 12. The type judgments are of the form $\Gamma, \Sigma \vdash \mathtt{e} : \mathtt{C}$, where Γ is the type environment of variables and Σ is the type environment of object labels.

Recall that we defined a partial order $\preceq$ on types such that $\mathtt{C} \preceq \mathtt{C}'$ is true iff $\mathtt{C} <: \mathtt{C}'$, $\mathtt{C}'$ must be defined in the same package as $\mathtt{C}$ and if $\mathtt{C}$ is a portal type, then $\mathtt{C}'$ must be the same type. If $\mathtt{C} \preceq \mathtt{C}'$, then we say that $\mathtt{C}$ is a *scope-safe* subtype of $\mathtt{C}'$ and the widening of a reference from the type $\mathtt{C}$ to $\mathtt{C}'$ is *scope-safe*. By Rules (T-Update) and (T-Invk), the reference widening in the field assignments and parameters passing is *scope-safe*.

Rule (T-Store) of the form $\Sigma \vdash \sigma$ says that object store σ is well typed, if the type environment Σ has the same domain as σ and for each object label ℓ in the domain of σ, $\Sigma(\ell)$ is equal to the type of $\sigma(\ell)$ and $\sigma(\ell)$ must also be well-typed. If $\sigma(\ell) = \mathtt{C}^{\ell'}(\overline{\mathtt{v}})$, then by Rule (T-StoreLoc), an object $\mathtt{C}^{\ell'}(\overline{\mathtt{v}})$ is well-typed, if the types of $\overline{\mathtt{v}}$ are *scope-safe* subtypes of the field types.

In the typing rule for class (T-Class), we require that in a class $\mathtt{C}$, the base class can not be a portal type and the types of the fields and the base class must be visible in $\mathtt{C}$. Also, all methods in a class must be well-typed by Rule (T-Method). If a method is well-typed, then the method body is well-typed by the expression typing rules, the type of the return expression is not widened, and the method expression must be visible in the class of the method by the expression visibility rules in Figure 13. Note that in (T-Class) we abuse notation by writing $\mathit{visible}(\overline{\mathtt{C}}, \mathtt{C})$ to assert that all types in the $\overline{\mathtt{C}}$ are visible in $\mathtt{C}$.

Type visibility:

$$\mathit{visible}(\mathtt{C}, \mathtt{C}_0) \text{ iff } \begin{cases} \text{either } \mathtt{C} \text{ is a scoped type and in the same} \\ \quad \text{or the super-package of } \mathtt{C}_0 \\ \text{or } \mathtt{C} \text{ is a portal type and} \\ \quad \text{in the immediate subpackage of } \mathtt{C}_0 \end{cases}$$

Static expression visibility:

$$\Gamma \vdash \mathit{visible}(\mathtt{this}, \mathtt{C}_0) \qquad \frac{\Gamma, \emptyset \vdash \mathtt{e} : \mathtt{C} \quad \mathit{visible}(\mathtt{C}, \mathtt{C}_0) \qquad \forall \mathtt{e}' \in \mathit{subexp}(\mathtt{e})\ .\ \Gamma \vdash \mathit{visible}(\mathtt{e}', \mathtt{C}_0)}{\Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C}_0)}$$

$$\mathit{subexp}(\mathtt{e}) = \begin{cases} \emptyset & \text{if } \mathtt{e} = \mathtt{x} \mid \mathtt{v} \mid \mathtt{new}\ \mathtt{C}() \\ \{\mathtt{e}_0, \overline{\mathtt{e}}\} & \text{if } \mathtt{e} = \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) \\ \{\mathtt{e}_0, \mathtt{e}_1\} & \text{if } \mathtt{e} = \mathtt{e}_0.\mathtt{f}_\mathtt{i} := \mathtt{e}_1 \\ \{\mathtt{e}_0\} & \text{if } \mathtt{e} = \mathtt{e}_0.\mathtt{f}_\mathtt{i} \mid \mathtt{spawn}\,\mathtt{e}_0 \mid \mathtt{reset}\ \mathtt{e}_0 \end{cases}$$

Figure 13: Type and expression visibility.

Visibility of types and expressions. The static constraints in our model are mostly to restrict widening of references, and also to limit the accessibility of expressions by their types. For example, an expression of scoped type $\mathtt{C}$ is only visible in the defining package of $\mathtt{C}$ and its subpackages.

In Figure 13, we define a relation on types, $\mathit{visible}(\mathtt{C}, \mathtt{C}')$ (type $\mathtt{C}$ is *visible from* type $\mathtt{C}'$), which encodes the SJ access control rules: a scoped type defined in package `P` is visible to classes defined in `P` and its subpackages; a portal class is only visible from classes defined in the immediate parent package. One slightly surprising implication of this definition is that a portal type is not visible in its own class definition. Thus a portal class $\mathtt{C}$ does not contain code that refers to itself with the exception, as we shall see later, of the pseudo variable `this` which may indeed be used to access fields and methods from within the portal class.

We illustrate the visibility relation of types with the table in Figure 14, which shows when the types in the leftmost column is visible in the classes in the first row.

	`p.Port`	`p.Scop`	`p.q.Port`	`p.q.Scop`
`p.Port`				
`p.Scop`	visible	visible	visible	visible
`p.q.Port`	visible	visible		
`p.q.Scop`			visible	visible

Figure 14: The types `p.Port`, `p.Scop` are the portal and scoped type in the package `p` and `p.q.Port`, `p.q.Scop` are the portal and scoped type in the package `p.q`. The package `p.q` is a subpackage of `p`. The table entries indicate whether the types in the leftmost column is visible in the classes in first row.

We check the method body to determine whether type visibility constraints are violated in a class. The checking rules

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

$$\Gamma, \Sigma \vdash \ell : \Sigma(\ell) \qquad \text{(T-Loc)}$$

$$\Gamma, \Sigma \vdash \ell^{\mathtt{th}} : \mathtt{Thread} \qquad \text{(T-Thread)}$$

$$\frac{\Gamma, \Sigma \vdash \mathtt{e}_0 : \mathtt{C} \quad \mathit{fields}(\mathtt{C}) = (\overline{\mathtt{C}}\ \overline{\mathtt{f}})}{\Gamma, \Sigma \vdash \mathtt{e}_0.\mathtt{f}_\mathtt{i} : \mathtt{C}_\mathtt{i}} \qquad \text{(T-Field)}$$

$$\frac{\begin{array}{l}\Gamma, \Sigma \vdash \mathtt{e}_0 : \mathtt{C}' \quad \mathit{mdef}(\mathtt{m},\ \mathtt{C}') = \mathtt{C}'' \\ \mathit{mtype}(\mathtt{m},\ \mathtt{C}'') = \overline{\mathtt{C}} \to \mathtt{C} \\ \Gamma, \Sigma \vdash \overline{\mathtt{e}} : \overline{\mathtt{D}} \quad \overline{\mathtt{D}} \preceq \overline{\mathtt{C}} \quad \mathtt{C}' \preceq \mathtt{C}''\end{array}}{\Gamma, \Sigma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{C}} \qquad \text{(T-Invk)}$$

$$\Gamma, \Sigma \vdash \mathtt{new\ C}() : \mathtt{C} \qquad \text{(T-New)}$$

$$\frac{\begin{array}{l}\Gamma, \Sigma \vdash \mathtt{e} : \mathtt{C} \quad \mathit{fields}(\mathtt{C}) = (\overline{\mathtt{C}}\ \overline{\mathtt{f}}) \\ \Gamma, \Sigma \vdash \mathtt{e}_1 : \mathtt{C}_1 \quad \mathtt{C}_1 \preceq \mathtt{C}_\mathtt{i}\end{array}}{\Gamma, \Sigma \vdash \mathtt{e}.\mathtt{f}_\mathtt{i} = \mathtt{e}_1 : \mathtt{C}_\mathtt{i}} \qquad \text{(T-Update)}$$

$$\frac{\Gamma, \Sigma \vdash \mathtt{e} : \mathtt{C}}{\Gamma, \Sigma \vdash \mathtt{spawn\,e} : \mathtt{Thread}} \qquad \text{(T-Spawn)}$$

$$\frac{\Gamma, \Sigma \vdash \mathtt{e} : \mathtt{C} \quad \mathtt{C} \text{ is a portal}}{\Gamma, \Sigma \vdash \mathtt{reset\ e} : \mathtt{C}} \qquad \text{(T-Reset)}$$

Figure 10: Expression typing.

Store Typing:

$$\frac{\begin{array}{c}\mathrm{dom}(\Sigma) = \mathrm{dom}(\sigma) \quad \forall \ell \in \mathrm{dom}(\sigma)\ . \\ \Sigma \vdash \sigma(\ell)\ \wedge\ \Sigma(\ell) = \mathtt{C} \text{ if } \sigma(\ell) = \mathtt{C}^{\ell_0}(\overline{\mathtt{v}})\end{array}}{\Sigma \vdash \sigma} \qquad \text{(T-Store)}$$

$$\frac{\mathit{fields}(\mathtt{C}) = (\overline{\mathtt{C}}\ \overline{\mathtt{f}}) \quad \emptyset, \Sigma \vdash \overline{\mathtt{v}} : \overline{\mathtt{D}} \quad \overline{\mathtt{D}} \preceq \overline{\mathtt{C}}}{\Sigma \vdash \mathtt{C}^{\ell}\ (\overline{\mathtt{v}})} \qquad \text{(T-StoreLoc)}$$

Method typing:

$$\frac{\begin{array}{l}\Gamma = \overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C} \quad \Gamma, \emptyset \vdash \mathtt{e} : \mathtt{C}'' \quad \mathtt{C}'' \preceq \mathtt{C}' \\ \mathit{override}(\mathtt{m}, \mathtt{D}, \overline{\mathtt{C}} \to \mathtt{C}') \quad \Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C})\end{array}}{\mathtt{C}'\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\,\{\,\mathtt{return\,e};\ \}\ \mathtt{OK\ IN\ C} \triangleleft \mathtt{D}} \qquad \text{(T-Method)}$$

Class typing:

$$\frac{\begin{array}{l}\mathtt{K} = \mathtt{C}()\ \{\mathtt{super}();\ \mathtt{this}.\overline{\mathtt{f}} := \overline{\mathtt{new\ D}()};\} \\ \overline{\mathtt{M}}\ \mathtt{OK\ IN\ C} \triangleleft \mathtt{D} \quad \overline{\mathtt{D}} \preceq \overline{\mathtt{C}} \quad \mathit{visible}(\overline{\mathtt{C}}\overline{\mathtt{D}}\mathtt{D}, \mathtt{C})\end{array}}{\circ\ \mathtt{class\ P.C}\ \triangleleft\ \mathtt{D}\ \{\,\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\,\}\ \mathtt{OK}} \qquad \text{(T-Class)}$$

Figure 11: Type rules of store, method, and class.

Initializer look-up:

$$\mathit{init}(\mathtt{Object}) = ()$$

$$\frac{\begin{array}{l}\mathit{CT}(\mathtt{C}) = \circ\ \mathtt{class\ P.C}\ \triangleleft\ \mathtt{D}\ \{\,\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\,\} \quad \mathit{init}(\mathtt{D}) = \overline{\mathtt{new\ D}'()} \\ \mathtt{K} = \mathtt{C}()\ \{\mathtt{super}();\ \mathtt{this}.\overline{\mathtt{f}} := \overline{\mathtt{new\ C}'()};\}\end{array}}{\mathit{init}(\mathtt{C}) = \overline{\mathtt{new\ D}'()},\ \overline{\mathtt{new\ C}'()}}$$

Field look-up:

$$\mathit{fields}(\mathtt{Object}) = ()$$

$$\frac{\mathit{CT}(\mathtt{C}) = \circ\ \mathtt{class\ P.C}\ \triangleleft\ \mathtt{C}'\ \{\,\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\,\} \quad \mathit{fields}(\mathtt{C}') = (\overline{\mathtt{C}'}\ \overline{\mathtt{g}})}{\mathit{fields}(\mathtt{C}) = (\overline{\mathtt{C}'}\ \overline{\mathtt{g}}, \overline{\mathtt{C}}\ \overline{\mathtt{f}})}$$

Method definition lookup:

$$\frac{\mathit{CT}(\mathtt{C}) = \circ\ \mathtt{class\ P.C}\ \triangleleft\ \mathtt{C}'\ \{\,\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\,\} \quad \mathtt{m} \text{ is defined in } \overline{\mathtt{M}}}{\mathit{mdef}(\mathtt{m},\ \mathtt{C}) = \mathtt{C}}$$

$$\frac{\mathit{CT}(\mathtt{C}) = \circ\ \mathtt{class\ P.C}\ \triangleleft\ \mathtt{C}'\ \{\,\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\,\} \quad \mathtt{m} \text{ is not defined in } \overline{\mathtt{M}}}{\mathit{mdef}(\mathtt{m},\ \mathtt{C}) = \mathit{mdef}(\mathtt{m},\ \mathtt{C}')}$$

Method type lookup:

$$\frac{\begin{array}{c}\mathit{mdef}(\mathtt{m},\ \mathtt{C}) = \mathtt{C}' \quad \mathit{CT}(\mathtt{C}') = \circ\ \mathtt{class\ P.C}'\ \triangleleft\ \mathtt{C}''\ \{\,\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\,\} \\ \mathtt{D\ m}(\overline{\mathtt{D}}\ \overline{\mathtt{x}})\,\{\,\mathtt{return\,e};\ \} \in \overline{\mathtt{M}}\end{array}}{\mathit{mtype}(\mathtt{m},\ \mathtt{C}) = \overline{\mathtt{D}} \to \mathtt{D}}$$

Method body look-up:

$$\frac{\begin{array}{c}\mathit{mdef}(\mathtt{m},\ \mathtt{C}) = \mathtt{C}' \quad \mathit{CT}(\mathtt{C}') = \circ\ \mathtt{class\ P.C}'\ \triangleleft\ \mathtt{C}''\ \{\,\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\,\} \\ \mathtt{D\ m}(\overline{\mathtt{D}}\ \overline{\mathtt{x}})\,\{\,\mathtt{return\,e};\ \} \in \overline{\mathtt{M}}\end{array}}{\mathit{mbody}(\mathtt{m},\ \mathtt{C}) = (\overline{\mathtt{x}},\ \mathtt{e})}$$

Valid method overriding

$$\frac{\mathit{mtype}(\mathtt{m},\ \mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{D} \quad \mathtt{C}, \overline{\mathtt{C}} = (\mathtt{D}, \overline{\mathtt{D}})}{\mathit{override}(\mathtt{m}, \mathtt{C}_0, \overline{\mathtt{C}} \to \mathtt{C})}$$

Figure 12: Auxiliary functions.

are in Figure 13 and they state that if an expression $\mathtt{e}$ of type $\mathtt{C}$ is visible in a class $\mathtt{C}_0$, then either $\mathtt{e} = \mathtt{this}$ or the type $\mathtt{C}$ must be visible in the class $\mathtt{C}_0$ (i.e. $\mathit{visible}(\mathtt{C}, \mathtt{C}_0)$) and all the subexpressions of $\mathtt{e}$ to be visible in $\mathtt{C}_0$. This fact is represented by the judgment $\Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C}_0)$. We make an exception for `this` because even though a portal type is visible only to the classes of its immediate super-package, a portal object must be able to use the variable `this` for accessing its fields and calling its methods. For any scoped class, the type of the variable `this` are always visible in its class.

5. PROPERTIES OF THE CALCULUS

The purpose of our model is to simplify the allocation of objects in scoped memory areas. Thus, we would like to statically guarantee the properties that:

> During the evaluation of a real-time program,
> **(1)** the nesting structure of scopes remain a tree,
> **(2)** deallocated objects in scopes are no longer accessible.

In RTSJ, the nesting structure of scopes is determined by how the threads enter the scopes. In our model, the scope structure is fixed by how the portal objects representing the scopes are created. That is, if a scope a is represented by a portal object created in the scope b, then a must be directly contained in b; moreover, the portal object representing a is defined in the immediate subpackage of the portal object representing b. In Theorem 7, we prove that if a program is well-typed, then it will maintain some runtime invariants (explained below) during computation and it won't get stuck. The runtime invariants guarantee that the scopes represented by the portal objects can form a tree. They also ensure that the threads in a program will preserve such a scope tree such that each thread either enters the scopes already entered by the thread or enters a new scope directly contained in the current scope of the thread. We prove this claim in Lemma 8. Thus, even though a scope stack of a thread may grow indefinitely (e.g. the thread reenters the scopes already on stack), the nesting structure of scopes resembles the nesting structure of the scoped packages and always remains a tree.

We prove the second property in Theorem 10. Since we do not remove any object from the store, there will never be any dangling references in our calculus. However, we can still model the absence of dangling references in actual languages by ensuring that the deallocated objects are never used by any threads. We model deallocation using the explicit *reset* expression, which clears the fields of a portal if the portal is not used by any threads. We prove in Theorem 10 that the objects allocated in the portal before the reset are no longer accessible afterward.

The proofs of the theorems use two runtime invariants specified by the predicates $\mathit{safe}(\sigma)$ and $\mathit{safe}(\sigma,\, \ell_0,\, \mathtt{e})$ defined in Figure 15. If the predicate $\mathit{safe}(\sigma)$ is true, then any object in σ can safely access the objects referenced in its fields. The predicate $\mathit{safe}(\sigma,\, \ell_0,\, \mathtt{e})$ is true if any object referenced in the expression $\mathtt{e}$ can be safely accessed by ℓ_0.

Before we define the safe access of an object by another, first recall that an object of the label ℓ in the store σ is of the form $\mathtt{C}^{\ell'}(\overline{\mathtt{v}})$, where $\mathtt{C}$ is the type, $\overline{\mathtt{v}}$ are the fields, and ℓ' represents the memory area where the object is allocated.

$$\begin{array}{rcl} \text{if } \sigma(\ell) = \mathtt{C}^{\ell_0}(\overline{\mathtt{v}}) & \text{then} & \mathit{field}_\sigma(\ell) = \overline{\mathtt{v}},\ \mathit{scopeof}_\sigma(\ell) = \ell_0, \\ & & \text{and } \mathit{type}_\sigma(\ell) = \mathtt{C} \\ \ell \prec_\sigma \ell' & \text{if} & \mathit{scopeof}_\sigma(\ell) = \ell' \\ & \text{or} & \ell \prec_\sigma \ell'' \ \wedge\ \ell'' \prec_\sigma \ell' \end{array}$$

$$\frac{\forall \ell \in \mathtt{e}\ .\ \ell_0 \prec_\sigma \mathit{scopeof}_\sigma(\ell)\ \vee\ \mathit{scopeof}_\sigma(\ell) = \ell_0}{\mathit{safe}(\sigma,\, \ell_0,\, \mathtt{e})}$$

$$\frac{\begin{array}{c}\forall \ell \in \mathtt{dom}(\sigma) \quad \sigma(\ell) = \mathtt{C}^{\ell_0}(\overline{\mathtt{v}}) \quad \mathit{allocScope}_\sigma(\mathtt{C}, \ell_0) = \ell_0 \\ \forall \ell' \in \overline{\mathtt{v}}\ .\ \ell \prec_\sigma \mathit{scopeof}_\sigma(\ell')\ \vee\ \mathit{scopeof}_\sigma(\ell') = \ell\end{array}}{\mathit{safe}(\sigma)}$$

Figure 15: Safe stores and expressions.

The helper functions $\mathit{type}_\sigma(\ell) = \mathtt{C}$, $\mathit{scopeof}_\sigma(\ell) = \ell'$, and $\mathit{field}_\sigma(\ell) = \overline{\mathtt{v}}$, retrieve types, scope, and fields of an object.

We define a relation $\prec_\sigma$ on labels such that $\ell \prec_\sigma \ell'$ holds if the object ℓ belongs to the scope represented by portal ℓ'. Note that $\prec_\sigma$ is transitive but not reflexive, a portal object is not allocated within the scope that it represents.

We can think of the objects in σ as a tree so that if $\ell \prec_\sigma \ell'$, then $\sigma(\ell')$ is an ancestor of $\sigma(\ell)$. If $\sigma(\ell)$, $\sigma(\ell')$ are of portal types and $\ell \prec_\sigma \ell'$, then $\sigma(\ell')$ represents a memory area containing the area represented by $\sigma(\ell)$. Thus, objects in σ can form a tree that has the same hierarchy as that of the memory areas. In this tree, the root is the object representing the immortal memory area, the other inner nodes are portals, and scoped objects can only be the leaves. We assume that the scope of the root object is the immortal memory area represented by itself.

Safety invariants during computation. The safety conditions of store and expressions ensure safe access of objects allocated in scoped memory areas.

An object ℓ can safely access ℓ' if either $\ell \prec_\sigma \mathit{scopeof}_\sigma(\ell')$ or $\mathit{scopeof}_\sigma(\ell') = \ell$. The first case means that ℓ' must be allocated in the scope of ℓ or one of its parents. The latter case means that ℓ' is allocated in the scope represented by ℓ (provided that $\sigma(\ell)$ is a portal).

The condition $\mathit{safe}(\sigma)$ says that every object in the store σ is safe. An object $\sigma(\ell) = \mathtt{C}^{\ell_0}\ (\overline{\mathtt{v}})$ is safe if ℓ can be allocated in ℓ_0 (as specified by the condition $\mathit{allocScope}_\sigma(\mathtt{C}, \ell_0) = \ell_0$) and ℓ can safely access any object pointed by $\ell' \in \overline{\mathtt{v}}$. This invariant is easy to maintain since the allocation area of an object remains the same during the lifetime of the object. The condition $\mathit{safe}(\sigma,\, \ell_0,\, \mathtt{e})$ says that the ℓ_0 can safely access the objects referenced in the expression $\mathtt{e}$. The place holder $\ell^{\mathtt{ret}}$ may be a part of $\mathtt{e}$ for the condition $\mathit{safe}(\sigma,\, \ell_0,\, \mathtt{e})$ to be true since $\ell^{\mathtt{ret}}$ will not be manipulated during computation.

Runtime expression visibility. The definition of runtime expression visibility is technical and only used in the proofs. It describes the visibility constraints of runtime expressions corresponding to the static expression visibility constraints. The judgment $\Sigma \vdash \mathit{visible}(\mathtt{e}, \ell)$ is true if the expression $\mathtt{e}$ is visible in the object ℓ. That is, either $\mathtt{e} = \ell$ (when ℓ is a portal), or $\mathtt{e} = \ell^{\mathtt{ret}}$, or if $\emptyset, \Sigma \vdash \mathtt{e} : \mathtt{C}$, then the type $\mathtt{C}$ is visible in the type $\Sigma(\ell)$ and the subexpressions of

$\mathtt{e}$ are also visible in ℓ.

$$\Sigma \vdash \mathit{visible}(\ell, \ell) \qquad \Sigma \vdash \mathit{visible}(\ell^{\mathtt{ret}}, \ell)$$

$$\frac{\emptyset, \Sigma \vdash \mathtt{e} : \mathtt{C} \quad \mathit{visible}(\mathtt{C}, \Sigma(\ell)) \qquad \forall \mathtt{e}' \in \mathit{subexp}(\mathtt{e}) \;.\; \Sigma \vdash \mathit{visible}(\mathtt{e}', \ell)}{\Sigma \vdash \mathit{visible}(\mathtt{e}, \ell)}$$

The properties of expressions. In our model, a program consists of a set of threads. Each thread $t[\overline{\ell\,\mathtt{e}}]$ has a unique label t and $\overline{\ell\,\mathtt{e}}$ is a stack of 2-tuples, each of which includes an object label ℓ_0 and an expression $\mathtt{e}$. Intuitively, each of the 2-tuple corresponds to a frame in the call stack of the thread and the label ℓ_0 refers to the receiver object of the method call that is reduced to $\mathtt{e}$. We want to show that at each step of the computation, we have $\mathit{safe}(\sigma)$ and $\mathit{safe}(\sigma,\ \ell_0,\ \mathtt{e})$. The second condition means that each object referenced in the current expression $\mathtt{e}$ can be safely accessed by ℓ_0. Note that this is a requirement stronger than necessary to prove Theorem 10. We choose this condition because it is easier to prove. If we change our model to accommodate more flexible programming style, then we might have to relax this condition.

For rest of the section, we assume that all classes in the class table CT are well-typed. The following lemma proves that subject reduction preserves typing and safety conditions.

Lemma 1 *Given a store σ, an object label ℓ_0 and an expression $\mathtt{e}$, if*

1. *$\mathit{safe}(\sigma)$, $\mathit{safe}(\sigma,\ \ell_0,\ \mathtt{e})$, and*
2. *$\Sigma \vdash \sigma$, $\emptyset, \Sigma \vdash \mathtt{e} : \mathtt{C}$, $\Sigma \vdash \mathit{visible}(\mathtt{e}, \ell_0)$,*

and $\sigma, \ell_0\ \mathtt{e} \;\rightarrow\; \sigma', \ell_0'\ \mathtt{e}'$, then

1. *$\mathit{safe}(\sigma')$, $\mathit{safe}(\sigma',\ \ell_0',\ \mathtt{e}')$, and*
2. *$\exists \Sigma', \mathtt{C}'$ such that $\Sigma' \vdash \mathit{visible}(\mathtt{e}', \ell_0')$, $\Sigma' \vdash \sigma'$, $\emptyset, \Sigma' \vdash \mathtt{e}' : \mathtt{C}'$, where $\mathtt{C}' \preceq \mathtt{C}$.*

The proof is by induction on the type derivation of the expression $\mathtt{e}$. In the case $\mathtt{e}$ is a method call, we uses Lemma 2.

Lemma 2 *If $\overline{\mathtt{x}} : \overline{\mathtt{B}}, \Sigma \vdash \mathtt{e} : \mathtt{D}$, $\emptyset, \Sigma \vdash \overline{\mathtt{v}} : \overline{\mathtt{A}}$, $\overline{\mathtt{A}} \preceq \overline{\mathtt{B}}$, then $\exists \mathtt{C}$ such that $\emptyset, \Sigma \vdash [{}^{\overline{\mathtt{v}}}/_{\overline{\mathtt{x}}}]\mathtt{e} : \mathtt{C}$, where $\mathtt{C} \preceq \mathtt{D}$.*

The following lemmas show that a well-typed and safe expression and store can make progress.

Lemma 3 *If $\Sigma \vdash \sigma$, $\emptyset, \Sigma \vdash \mathtt{e} : \mathtt{C}$, and $\mathtt{e} = \ell.\mathtt{m}(\overline{\mathtt{v}})$, then $\exists \mathtt{e}'$ such that $\sigma, \ell_0\ \mathtt{e} \;\rightarrow\; \sigma, \ell\ \mathtt{e}'$.*

Lemma 4 *If $\Sigma \vdash \sigma$, $\emptyset, \Sigma \vdash \mathtt{e} : \mathtt{C}$, $\mathit{safe}(\sigma)$, $\mathit{safe}(\sigma,\ \ell_0,\ \mathtt{e})$, and $\Sigma \vdash \mathit{visible}(\mathtt{e}, \ell_0)$, then*

1. *either $\mathtt{e}$ is an irreducible value, or contains expressions of the forms $\ell.\mathtt{m}(\overline{\mathtt{v}})$, $\mathtt{spawn}\,\mathtt{e}$, $\mathtt{reset}\ \ell$,*
2. *or $\exists \sigma', \mathtt{e}'$ such that $\sigma, \ell_0\ \mathtt{e} \;\rightarrow\; \sigma', \ell_0\ \mathtt{e}'$.*

The proof is by induction on the type derivation of the expression $\mathtt{e}$. The only non-trivial case is when $\mathtt{e} = \mathtt{new}\ \mathtt{C}()$, we need to show that the partial function $\mathit{allocScope}_\sigma(\mathtt{C}, \ell_0)$ can return an allocation scope for $\mathtt{C}$. Suppose that the type of $\sigma(\ell_0)$ is $\mathtt{C}_0$. From $\Sigma \vdash \mathit{visible}(\mathtt{e}, \ell_0)$, we know that $\mathtt{C}$ is visible from $\mathtt{C}_0$. Thus, if $\mathtt{C}$ is a portal type, then it is defined in the immediate subpackage of $\mathtt{C}_0$ and $\mathit{allocScope}_\sigma(\mathtt{C}, \ell_0) = \ell_0$. If $\mathtt{C}$ is scoped, then it is defined in the same or the super-package of $\mathtt{C}_0$. From $\mathit{safe}(\sigma)$, we know that the nesting relation of scopes corresponds to the nesting relation of the packages (where the portal objects that represent the scopes are defined). Thus, we can always find an allocation scope of $\mathtt{C}$ by searching the scope stack upward starting from the scope of $\sigma(\ell_0)$ or from the scope represented by $\sigma(\ell_0)$ if $\sigma(\ell_0)$ is a portal object.

The properties of computation. We first give the definitions of well-typed and safe programs, then define the conditions in which a thread can get stuck, and last we prove that a well-typed program will not be stuck and its reduction preserves safety properties for the expressions of all threads.

A program σ, P is well-typed if $\exists \Sigma$ such that $\Sigma \vdash \sigma$, and for each thread $t[\overline{\ell\,\mathtt{e}}]$ in P, we have

1. $\forall\ \ell\ \mathtt{e} \in \overline{\ell\,\mathtt{e}}\ .\ \Sigma \vdash \mathit{visible}(\mathtt{e}, \ell)$,
2. if $\overline{\ell\,\mathtt{e}} = \ldots \ell\ \mathtt{e}\ .\ \ell'\ \mathtt{e}' \ldots$, then $\Sigma \vdash \mathit{visible}(\ell', \ell)$, and
3. if $\overline{\ell\,\mathtt{e}} = \ell_1\ \mathtt{e}_1 \ldots \ell_\mathtt{n}\ \mathtt{e}_\mathtt{n}$ and
   $\mathtt{e}_\mathtt{n}' = \mathtt{e}_\mathtt{n}, \ldots, \mathtt{e}_\mathtt{k}' = [{}^{\mathtt{e}_{\mathtt{k}+1}'}/_{\ell^{\mathtt{ret}}}]\mathtt{e}_\mathtt{k}, \ldots, \mathtt{e}_1' = [{}^{\mathtt{e}_2'}/_{\ell^{\mathtt{ret}}}]\mathtt{e}_1$,
   then $\exists \mathtt{C}$ such that $\emptyset, \Sigma \vdash \mathtt{e}_1' : \mathtt{C}$.

The definition of a well-typed program is technical and it basically says that the expressions in each thread of a well-typed program must satisfy some visibility constraints and if we substitute each place holder $\ell^{\mathtt{ret}}$ in a thread with the expression that it replaced, then the final expression is well-typed.

A program σ, P is safe if $\mathit{safe}(\sigma)$ and for each thread $t[\overline{\ell\,\mathtt{e}}]$ in P, we have

1. $\forall \ell\ \mathtt{e} \in \overline{\ell\,\mathtt{e}}\ .\ \mathit{safe}(\sigma,\ \ell,\ \mathtt{e})$, and
2. if $\overline{\ell\,\mathtt{e}} = \ldots \ell\ \mathtt{e}\ .\ \ell'\ \mathtt{e}' \ldots$, then $\mathit{safe}(\sigma,\ \ell,\ \ell')$.

The following two lemmas show that subject reduction preserves typing and safety conditions of a program and if a program is well-typed and safe, then it can make progress.

Lemma 5 *If σ, P is well-typed and safe, and $\sigma, P \Rightarrow \sigma', P'$, then σ', P' is well-typed and safe.*

The proof is straightforward and it applies Lemma 1.

We say that a thread of the form $t[\,\ell_0\,\mathtt{v}\,]$ in P is terminated.

Lemma 6 *If σ, P is well-typed and safe, then either all threads in P are terminated or there exists σ', P' such that $\sigma, P \Rightarrow \sigma', P'$.*

The proof is also straightforward and it applies Lemmas 3 and 4.

We say that an irreducible program σ, P is stuck if P contains a non-terminated thread. Also, let $\Rightarrow^*$ be the reflexive and transitive closure of $\Rightarrow$.

Theorem 7 *If σ, P is well-typed and safe, and $\sigma, P \Rightarrow^* \sigma', P'$, then σ', P' is not stuck and it is well-typed and safe.*

The proof is immediate from Lemma 5 and 6.

Nesting structure of scopes. Suppose that the program σ, P is safe. From $\mathit{safe}(\sigma)$, we have that any portal object $\sigma(\ell)$ in σ, (where $\sigma(\ell) = \mathtt{C}^{\ell_0}(\overline{\mathtt{v}})$), is allocated in the scope represented by the portal $\sigma(\ell_0)$ and the type of $\sigma(\ell_0)$ is defined in the immediate super-package of C. Let the scope represented by $\sigma(\ell_0)$ be the parent of the scope represented by $\sigma(\ell)$. Then, the scopes represented by the portal objects in σ form a tree with immortal memory as the root.

We will show that a thread will enter scopes according to the tree structure of scopes. That is, either a thread t enters the scopes already entered by t or enter a new scope that is directly contained in the current scope.

In our model, a thread enters a scope each time that a method call $\ell.\mathtt{m}(\overline{\mathtt{v}})$ is evaluated and a new stack frame $\ell\ \mathtt{e}$ is pushed onto the stack. If $\sigma(\ell)$ is a scoped object, then the entered scope is represented by the portal $\mathit{scopeof}_\sigma(\ell)$ and we show in Lemma 8 that such a scope is already entered by t. If $\sigma(\ell)$ is a portal, then t enters the scope represented by $\sigma(\ell)$, and in Lemma 8, we show that such a scope is directly contained in the previous scope entered by t.

Lemma 8 *If σ, P is well-typed, safe, $t[\,\overline{\ell\,\mathtt{e}}\ .\ \ell\ \mathtt{e}\,] \in P$, and $\overline{\ell\,\mathtt{e}} = \ldots \ell_0\ \mathtt{e}_0$, then*

1. *if $\sigma(\ell)$ is a scoped object, then $\mathit{scopeof}_\sigma(\ell) \in \overline{\ell}$,*
2. *if $\sigma(\ell)$ is a portal object, then either $\mathit{scopeof}_\sigma(\ell) = \ell_0$ if $\sigma(\ell_0)$ is a portal or $\mathit{scopeof}_\sigma(\ell) = \mathit{scopeof}_\sigma(\ell_0)$ otherwise.*

Proof. Since σ, P is well-typed and safe, $\exists\Sigma$ such that $\Sigma \vdash \mathit{visible}(\ell, \ell_0)$ and $\mathit{safe}(\sigma, \ell_0, \ell)$. The safety condition implies that $\mathit{scopeof}_\sigma(\ell_0) \prec_\sigma \mathit{scopeof}_\sigma(\ell)$ or $\ell_0 = \mathit{scopeof}_\sigma(\ell)$. Moreover, if the types of $\sigma(\ell_0)$ and $\sigma(\ell)$ are $\mathtt{C}_0$ and C, then C is visible from $\mathtt{C}_0$ which means that if C is a portal type, then C is defined in the immediate subpackage of $\mathtt{C}_0$. It is easy to show that if C is a portal type, then either $\mathit{scopeof}_\sigma(\ell) = \ell_0$ or $\mathit{scopeof}_\sigma(\ell) = \mathit{scopeof}_\sigma(\ell_0)$. If C is scoped, then we can show by induction that $\mathit{scopeof}_\sigma(\ell) \in \overline{\ell}$. $\square$

Safe deallocation. We will show that if a scope is not used by any thread, then the objects allocated in the scope can be safely deallocated. We use the reset expression to clear the fields of a portal object that represents a scope and the reset operation will succeed if the scope is not used by any thread. We prove in Theorem 10 that if the fields of a portal is cleared, then no object allocated in the corresponding scope is reachable in the program.

We assume that the first thread of a program always starts from the immortal memory and by Rule (G-Spawn), a new thread inherits the portals of its parent. Thus, if $P = P' \mid t[\,\ell\,\mathtt{e}\ldots]$, then ℓ corresponds to immortal memory, which should be present in the store during the entire computation of the program. We also assume that the object that corresponds to the immortal memory is allocated in itself.

Theorem 10 shows that in a safe program σ, P, if a portal $\sigma(\ell_0)$ is reset successfully, then the objects allocated in the scope represented by $\sigma(\ell_0)$ are not reachable in σ, P. Recall that if a scope represented by $\sigma(\ell_0)$ is not used by any thread, then $\mathit{refcount}(\sigma, \ell_0) = 0$ and if the object $\sigma(\ell)$ is allocated in that scope, then $\mathit{scopeof}_\sigma(\ell) = \ell_0$.

We say that ℓ is reachable in σ, P if either it is referenced in a thread of P or it is in the field of $\sigma(\ell')$, where ℓ' is reachable in σ, P.

Lemma 9 *If σ, P is safe, $P = P' \mid t[\,\overline{\ell\,\mathtt{e}}\,.\,\ell\,\mathtt{e}\,]$, and $\ell \prec_\sigma \ell'$, then $\ell' \in \overline{\ell}$.*

The proof is straightforward by Lemma 8.

Theorem 10 *If σ, P is safe, $P = P'' \mid t[\,\overline{\ell\,\mathtt{e}}\,.\,\ell\,E[\mathtt{reset}\ \ell_0]\,]$, $\mathit{refcount}(\sigma, \ell_0) = 0$, $\sigma, P \Rightarrow \sigma', P'$, and $P' = P'' \mid t[\,\overline{\ell\,\mathtt{e}}\,.\,\ell\,E[\ell_0]\,]$, then the objects of σ that are allocated in the scope represented by the portal $\sigma(\ell_0)$ are not reachable in σ', P'.*

Proof. Since $\mathit{refcount}(\sigma, \ell_0) = 0$, we have $\ell_0 \notin \overline{\ell}$, $\forall t[\,\overline{\ell\,\mathtt{e}}\,] \in P$. We will show by induction that if $\ell' \in \mathit{dom}(\sigma)$ is reachable in σ', P', then $\ell' \not\prec_{\sigma'} \ell$, that is, ℓ' is not allocated in the scope represented by $\sigma'(\ell_0)$ or its subscopes. Suppose $t[\,\overline{\ell\,\mathtt{e}}\,] \in P'$, $\ell\ \mathtt{e} \in \overline{\ell\,\mathtt{e}}$, and $\ell' \in \mathtt{e}$. Since σ', P' is safe, we have $\mathit{safe}(\sigma')$ and $\mathit{safe}(\sigma', \ell, \mathtt{e})$. From $\mathit{safe}(\sigma', \ell, \mathtt{e})$, we have $\ell \prec_{\sigma'} \mathit{scopeof}_{\sigma'}(\ell')$ or $\mathit{scopeof}_{\sigma'}(\ell') = \ell$. Thus, $\ell' \not\prec_{\sigma'} \ell_0$, because otherwise, either $\ell = \ell_0$ or $\ell \prec_{\sigma'} \ell_0$, and from Lemma 9, we have $\ell_0 \in \overline{\ell}$, which is a contradiction. Suppose $\ell'' \in \mathit{field}_{\sigma'}(\ell')$, where $\ell' \in \mathit{dom}(\sigma)$ is reachable in σ', P' but $\ell' \not\prec_{\sigma'} \ell_0$. From $\mathit{safe}(\sigma')$, we have $\ell' \prec_{\sigma'} \mathit{scopeof}_{\sigma'}(\ell'')$ or $\mathit{scopeof}_{\sigma'}(\ell'') = \ell'$. If $\ell' \prec_{\sigma'} \mathit{scopeof}_{\sigma'}(\ell'')$, then by assumption $\ell' \not\prec_{\sigma'} \ell_0$, we have $\ell'' \not\prec_{\sigma'} \ell_0$. If $\mathit{scopeof}_{\sigma'}(\ell'') = \ell'$, then by assumption, we have $\mathit{scopeof}_{\sigma'}(\ell'') \not\prec_{\sigma'} \ell_0$. Furthermore, if $\ell_0 \neq \ell'$, which implies $\ell_0 \neq scopeof_{\sigma'}(\ell'')$, then $\ell'' \not\prec_{\sigma'} \ell_0$. If $\ell_0 = \ell'$, then $\ell'' \in field_{\sigma'}(\ell_0)$ and by Rule (R-Reset), ℓ'' is a new object in the field of $\sigma'(\ell_0)$ and thus $\ell'' \notin dom(\sigma)$. $\square$

6. RELATED WORK

Boyapati et al. [6] combine region-based memory management with ownership types to statically guarantee that realtime threads do not interfere with GC. This approach uses lexically-scoped memory regions to allocate objects for realtime threads. The static scopes of the regions dictate the regions hierarchy and region-allocated objects are parameterized by owners that are either other objects or regions. The type ownership information and their static constraints ensure that objects of inner regions can only reference objects of the same or outer regions. Since the lexically nested regions only support a single thread, they define shared regions (which may contain subregions) to allocate objects used by multiple realtime threads. While more flexible than Scoped Types, this approach is more invasive, requiring more program annotations, and more complex overall.

Scoped Types also enforce a property similar to deep ownership [10]. A portal object encapsulates the scoped objects allocated in the scope of the portal so that only objects of the same scope or nested scopes are able to reference these scoped objects. Also, with Scoped Types, the shared and unshared scopes are treated uniformly.

Figure 6 shows how all this work fits together into a wider context of ownership and confined types. Like Confined Types [37, 20], this work takes an *implicit* approach, enforcing encapsulation primarily by a set of confinement rules that restrict language constructs, having a minimal syntactic overhead. The original work on Confined Types encapsulated objects within (statically declared) packages. ConfinedBeans recently extended the implicit approach so that individual (dynamically created) objects could be the unit of confinement [11]: Scoped Types further extend this work to support nested (thus "scoped") encapsulation domains. In contrast, the *explicit* approach, pioneered by Ownership Types [13], explicitly annotates types with extra information denoting objects' ownership: these annotations impose a significant syntactic burden on the programmer. While the earliest Ownership Types proposals supported nested objects as the unit of encapsulation ("deep ownership" [12, 10]), more recent work has supported individual un-nested objects ("shallow ownership" [1, 6]) and also per-package ownership [38, 30].

	Per Package	Per Object	Nested Objects
Implicit	Confined Types [37, 20]	Confined Beans [11]	Scoped Types **[this work]**
Explicit	Lightweight Confinement [38, 30]	Shallow Ownership Types [1, 6, 5]	Deep Ownership Types [13, 12, 10]

Cyclone [19] is a type-safe language derived from C and it supports region-based memory management. Cyclone includes dynamic regions with lexically scoped lifetimes, stack regions and a heap region. To prevent dereferencing dangling pointers, Cyclone uses types parameterized by region names to track pointers to regions. Since the pointers to a region may escape the scope of the region via some typing constructs, Cyclone also annotates function types with an effect that records the set of regions the function may access, and a function may be called only if the regions in the effect are alive. The regions in Cyclone are limited to single threaded execution model. Also, the use of effects of functions may not work with realtime Java, since the Java's type safety requirement does not allow objects to hold invalid references even if never used. Grossman recently extended Cyclone with a type system for preventing data races [18].

The MLKit is an implementation of ML which uses regions and region-inference [34, 35]. One of the main difference with the model presented here is that ML is a functional language without built-in support for concurrency. Hallenberg *et al.* [21] compared the performance of region-based memory management with or without copying garbage collection and indicated no visible effect in memory usage when using a *weakened* version of region inference that prevents dangling reference.

Cilk [31] is a C-based language for parallel programming and it provides stack-like allocation in a so-called *cactus-stack* [3] to handle simple allocations for multiple threads. A thread in Cilk can access its own stack allocations and the stacks allocated by its ancestors but not the stacks allocated by its siblings; neither can it return a pointer to an object allocated in its own cactus-stack back to its parent. The sharing of cactus-stacks in Cilk is created by spawning child threads, while with Scoped Types, threads share objects by explicitly entering a shared scope.

The dangers involved in the RTSJ programming model have motivated Kwon *et.al.* to propose a restricted programming model called Ravenscar-Java [24], based on earlier work for Ada [8], in which memory areas can not be nested and are single threaded. We believe that on one hand Ravenscar does not go far enough, and on the other it throws away too many features of the RTSJ. Ravenscar falls short in that it does not offer a model that can be statically checked, and it is too restrictive because we have found concurrency and nested scopes to be essential features for expressing certain algorithms naturally in real-time Java.

BeeBee and Rinard [2] implemented the memory management extensions in the RTSJ and tested some benchmarks on the implementation. They found it "close to impossible" to develop error-free realtime Java programs without some help from debugging tools or static analysis. Their benchmark results showed significant runtime overhead caused by dynamic access checks. Palacz and Vitek [28], as well as Corsaro and Cytron [14] presented constant-time algorithms for checking single parent rules and memory reference checks. Other efficient implementations of checks include [16].

Finally, a more detailed description of the example in this paper can be found in [29], which studies the semantics and design patterns for programming in real-time Java. The addition of the `reset` in the calculus is directly inspired by the wedge thread pattern.

7. CONCLUSION

In this paper we have introduced Scoped Types, a static programming discipline to support the kind of scoped memory management found in the Real-Time Specification for Java. The key contribution of Scoped Types is that it statically maintains the invariants that the RTSJ checks dynamically, yet imposes minimal syntactic overheads upon programmers. In particular, by statically associating scoped objects to their defining packages, Scoped Types ensure that incoming references are never created, eliminating the potential for run-time errors caused by illegal assignment operations. By modeling nested scopes with nested packages, Scoped Types ensure that scopes will never form cycles, again eliminating the potential for runtime exceptions to signal this error. By statically tracking the objects contained within each scope, Scoped Types ensure that it is safe to discard all the objects in a scope once the last thread has left it. We have formalized Scoped Types within the SJ-calculus (a variant of Featherweight Java) and demonstrated that it avoids dangling references (from either incoming references or object discarding) and cycles of scopes. We hope the techniques embodied within Scoped Types may be useful in many RTSJ applications, making real-time Java programming more practical, more convenient, and more reliable.

Admittedly, Scoped Types are restrictive with respect to reuse of existing Java classes. Library classes cannot be directly instantiated in a scopes and, even though scoped classes can extend existing classes, inherited methods can not be invoked. To lift some of these restrictions, we could use concepts such as anonymous methods [38] to allow an object to invoke some inherited methods defined in other packages. Also, we could use genericity to instantiate library classes in different scoped packages. Finally we could use ideas such as borrowed parameters [7] to enable passing such temporary references to support some of the patterns of [29]. While all of these approaches are promising, it is not clear how much expressive power is really needed in practice. Our first goal for future work is thus to gain first-hand experience using Scoped Types on real-time codes.

Acknowledgments. The authors are grateful to David Clarke, Jason Fox, David Holmes, Doug Lea, Filip Pizlo for assistance, comments and inspiration; the ECOOP reviewers provided insightful comments. This work was supported under grant HDCCSR #0341304.

8. REFERENCES

[1] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Appplications (OOPSLA)*, November 2002.

[2] William S. Beebee, Jr. and Martin Rinard. An implementation of scoped memory for real-time Java. In *Proceedings of the First International Workshop on Embedded Software (EMSOFT)*, 2001.

[3] Daniel G. Bobrow and Ben Wegbreit. A model and stack implementation of multiple environments. *Communications of the ACM*, 16:591–602, 1973.

[4] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, June 2000.

[5] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Principles of Programming Languages (POPL*, January 2003.

[6] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, Jr., and Martin Rinard. Ownership types for safe region-based memory management in Real-Time Java. In *Proceedings of Conference on Programming Languages Design and Implementation*. ACM Press, 2003.

[7] John Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 2000. In this issue.

[8] Alan Burns. The Ravenscar Profile. *ACM SIGADA Ada Letters*, 19(4):49–52, 1999.

[9] Dries Buytaert, Frans Arickx, and Johan Vos. A profiler and compiler for the Wonka Virtual Machine. In *USENIX JVM'02 Work in Progress*, San Francisco, CA, August 2002.

[10] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications (OOPSLA-02)*, volume 37, 11 of *ACM SIGPLAN Notices*, pages 292–310, New York, November 4–8 2002. ACM Press.

[11] Dave Clarke, Michael Richmond, and James Noble. Saving the world from bad Beans: Deployment-time confinement checking. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Appplications (OOPSLA)*, Anaheim, CA, November 2003.

[12] David Clarke. *Object Ownership and Containment*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia, 2001.

[13] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 48–64. ACM, October 1998.

[14] Angelo Corsaro and Ron K. Cytron. Efficient memory reference checks for real-time Java. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, 2003.

[15] Angelo Corsaro and Doug Schmidt. The design and performace of the jRate Real-Time Java implementation. In *The 4th International Symposium on Distributed Objects and Applications (DOA'02)*, 2002.

[16] Jason M Fox and Adam Welc. Implementation of Real-Time Java scope access checks for JikesRVM. Tech. report, Purdue, may 2003.

[17] Urs Gleim. JaRTS: A portable implementation of real-time core extensions for Java. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '02)*, Berkeley, CA, USA, 2002. USENIX.

[18] Dan Grossman. Type-safe multithreading in Cyclone. In *ACM Workshop on Types in Language Design and Implementation*, Now Orleans, LA, January 2003.

[19] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 282–293, Berlin, June 2002. ACM Press.

[20] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. *ACM SIGPLAN Notices*, 36(11):241–253, November 2001. Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01).

[21] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02).* ACM Press, June 2002. Berlin, Germany.

[22] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

[23] Timesys Inc. jTime. 2003. `http://www.timesys.com`.

[24] Jagun Kwon, Andy Wellings, and Steve King. Ravenscar-Java: A high integrity profile for real-time Java. In *Joint ACM Java Grande / ISCOPE Conference*, Seattle, Washington, November 2002.

[25] NASA/JPL and Sun. Golden gate. 2003. `http://research.sun.com/projects/goldengate`.

[26] Kelvin Nilsen. Adding real-time capabilities to Java. *Communications of the ACM*, 41(6):49–56, June 1998.

[27] James Noble, Robert Biddle, Ewan Tempero, Alex Potanin, and Dave Clarke. Towards a model of encapsulation. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2003.

[28] Krzysztof Palacz and Jan Vitek. Java subtype test in real-time. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP03)*, Darmstadt, Germany, 2003.

[29] Filip Pizlo, Jason Fox, David Holmes, and Jan Vitek. Real-time java scoped memory: design patterns and semantics. In *Proceedings of the IEEE International Symposium on Object-oriented Real-Time Distributed Computing (ISORC'04)*, Vienna, Austria, May 2004.

[30] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Featherweight generic confinement. In *Informal Proceedings of FOOL 2004*, 2004.

[31] Keith Harold Randall. *Cilk: efficient multithreaded computing.* PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1998.

[32] David Sharp. Real-time distributed object computing: Ready for mission-critical embedded system applications. In *Proceeding of the Third International Symposium on Distribtued-Objects and Applications (DOA'01)*, 2001.

[33] Fridtjof Siebert. Hard real-time garbage collection in the Jamaica Virtual Machine. In *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, Hong Kong, 1999.

[34] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):724–767, July 1998.

[35] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, February 1997.

[36] Jörgen Tryggvesson, Torbjörn Mattsson, and Hansruedi Heeb. Jbed: Java for real-time systems. *Dr. Dobb's Journal of Software Tools*, 24(11), November 1999.

[37] Jan Vitek and Boris Bokowski. Confined types in Java. *Software — Practice and Experience*, 31(6):507–532, 2001.

[38] Tian Zhao, Jens Palsberg, and Jan Vitek. Lightweight confinement for Featherweight Java. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03)*, pages 135–148. ACM Press, October 2003.