ScopeJ: A Multi-threaded Region Calculus

Tian Zhao∗ Jason Baker† Jan Vitek†

∗ EECS, University of Wisconsin – Milwaukee
† Department of Computer Science, Purdue University

ABSTRACT
We introduce ScopeJ, a statically typed, multi-threaded, object calculus in which scopes are first class constructs which reify the notion of allocation context and provide a safe alternative to automatic memory management by ensuring linear time allocation and bulk deallocation. The design of ScopeJ has been driven by the requirement of the Real-time Specification for Java and backwards compatibility with the Java programming language. ScopeJ has an expressive ownership type system which provides strong static guarantees without requiring changes to the Java language.

1. INTRODUCTION
Efforts at integrating manual memory reclamation in high-level programming languages have focused on type-safe region-based allocation schemes. Regions are memory pools used by application code to allocate data structures which can be freed in a single, constant-time, operation. A region-based program is deemed type-safe if no memory location is ever used after it has been freed. Safety can be achieved at runtime [1] or at compile-time [7]. In this paper we propose a new abstraction, referred to as a scope, for safely integrating manual memory management into a garbage collected runtime system. Scope reifies the notion of lexically-scoped region in order to allow multiple threads to operate within the same allocation context and let them communicate through shared variables.

This work is part of a larger effort motivated by requirements from the OpenGroup to design a variant of the Real-time Specification for Java (RTSJ) language suited for use in safety-critical applications [8]. The dynamic region-based memory model of the RTSJ has been singled out as one of the most egregious source of errors in real-time Java programs and statically safe alternative are being considered. Previous work on RavenscarJava [12], the leading high-integrity profile, severely curtails the use of regions without providing strong static guarantees. Our goal is to find a compromise that is more expressive than RavenscarJava, yet statistically type-safe, and abides by the following three pragmatic constraints: (i) no changes to standard Java libraries and non-RT applications, (ii) no changes to the Java syntax and (iii) minimal changes to the RT Java runtime system. The portion of RTSJ programs that require region-based allocation is expected to be small. And by necessity, if the programs are to be certified for safety-critical applications, the scope structures will be simple. Thus we are not looking for a general region-based programming model, but rather a model that is minimally intrusive and sufficiently expressive for typical tasks. We use the catalogue of real-time design patterns presented in this section can be expressed as a type-correct ScopeJ program.

ScopeJ is a dialect of Java in which memory regions are represented by scopes. They are first class values which can be created, entered, shared, and reclaimed under programmer control. In order to enforce memory-safety, ScopeJ programs are structured so as to make the mapping between allocation contexts and objects explicit. This is achieved by introducing the following entities into the language:

- **Scope**: A scope is an object that acts as a region handle, it has a one-to-one mapping to a memory region. Whenever a method is invoked on a scope, the allocation context is set to the associated region. The contents of this region can be released in a single constant-time operation if no threads are currently active in the scope.
- **Scope-Bound**: An instance of a scope-bound class is always allocated in a memory region, and any data allocated while executing one of its methods is allocated in the same region.
- **Permanent**: A permanent object, a special case of scope-bound, lives in a distinguished ‘immortal’ memory region. Permanent objects have a special status as they can be accessed by region- and heap-allocated objects.

The ScopeJ type system is inspired by a long line of previous research on ownership and region types, yet it differs from previous work in that it requires no changes to the Java language. In fact, neither metadata nor generics are required. This paper extends our previous work on SJ [19]. But unlike SJ, we do not rely on the package structure of Java or on existing visibility rules for enforcing confinement. The notion of implicitly parameterized types and borrowed references are new. Unlike our previous work on confined types [20], ScopeJ enforces object-level confinement (i.e. for scope objects).

2. PROGRAMMING WITH SCOPEJ
In this section, we introduce ScopeJ with a series of motivating examples. While we use the full Java syntax, rather than the rarefied language of Section 3, the essence of every one of the examples presented in this section can be expressed as a type-correct ScopeJ program.

The contribution of this paper is a new type system which protects RTSJ programs from dangling pointer errors. The ScopeJ type system ensures that after the contents of a region are reclaimed there can be no pointers into that region. The type system is novel in that it supports implicit region polymorphism, classes can be reused in different regions just as if they were equipped with a region parameter but the parameter is neither declared nor instantiated. Object arrays are naturally supported as a special case of implicit parametric types. The type system allows references to object to be used in different regions with a form of borrowing. Unlike most ownership type systems upcasts are allowed within a scope (even to Object), this allows the use of standard Java collections and methods such as sort (Object []). We have encoded each of the design patterns of [13] and give code snippets for the main ones.
class Unpacker {
    void parse(Data b) {...}
    void send(Data b) {...}
}

class RunLoop implements Runnable {
    Data bytes;
    void setMessage(Data b) { bytes = b; }
    void run() {
        Unpacker p = new Unpacker();
        p.parse(bytes);
        p.send(bytes);
    }
    void runLoop() {
        ScopeMemoryArea mem = new LMemory(...);
        RunLoop runnable = new RunLoop();
        while (true) {
            runnable.setMessage(bytes);
            mem.enter(runnable);
        }
    }
}

(a) Real-time Specification for Java

final class Processor extends Scope {
    static class Unpacker {
        void parse(Data b) {...}
        void send(Data b) {...}
    }
    void processMessage(Data bytes) {
        Unpacker p = new Unpacker();
        p.parse(bytes);
        p.send(bytes);
    }
}

(b) ScopeJ

**Heap**: Heap objects are instances of standard Java classes allocated in the heap. They are subject to the system's garbage collector and have normal Java semantics.

**Free**: Free objects are instances of classes which can safely be region- or heap-allocated, and will behave accordingly.

In source code, a scope is defined by a final class extending the distinguished `Scope` class; nesting of scopes is obtained by nesting of scope class definitions. Scope-bound classes are defined by nesting a Java class within a scope definition. Permanent classes are declared by implementing a marker interface `Top`.

**The Scoped Run Loop Pattern**: Figure 1 illustrates one of the most common RTSJ patterns, the Scoped Run Loop [13]. In this particular example the code periodically acquires data from, e.g., a sensor and processes that data in a memory region. The application logic can now be expressed as a method of the program is apparent. The scope `Processor` lives in the current region, and can thus be safely referred to from the body of `runLoop`, its fields point into a newly created region. The application logic can now be expressed as a method of the scope. The data is simply passed as an argument (l. 18), there is no need for a dedicated instance field.

In order for the program to type-check, it is necessary to explicite the fact that instances of the `Unpacker` class will be allocated within the `Processor`'s region. This is achieved by nesting the definition of the `Unpacker` class within `Processor`. The `ScopeJ` type system will then be able to prevent memory errors. In particular here, it would be a type-error to return an instance of `Unpacker` from one of `Processor`'s methods, as this would imply returning a scope-bound class out of it’s scope. Discussion of the `release()` method (l. 19) is deferred to the next section.

2.1 Multi-threading

Multiple threads may execute within the same memory region, either at the same time or at different times, by simply invoking method of the scope object. In order for threads to be able to interact, as opposed to simply share an allocation context, they must obtain a reference to shared data. In the RTSJ, each `ScopedMemoryArea` has a `portal`, a distinguished `Object` which can be read /
written by any thread currently active in the region.

Figure 2(a) illustrates an example of a portal idiom [13]. The RTSJ code creates a counter in a dedicated region (l. 5-9). The counter is stored in the portal of the memory area (l. 7). The counter can be incremented by any thread that enters the region (l. 10-14). In the case of the ScopeJ code, Figure 2(b), the scope CounterScope is defined with methods for creating and incrementing a counter. The counter is an instance of a scope-bound class and held in a field of the scope.

Release: The code of Figure 2(a) is not quite right, as the counter is implicitly deleted when the first call ma.enter() (l. 5-8) returns. This is a recognized drawback of the RTSJ API, the solution to keep data within a scope alive is to use a wedge thread [13], unfortunately this entails creating a real-time thread whose sole purpose is to keep a region live. ScopeJ provides an explicit release() method (shown in Figure 1 l. 19) to explicitly free the contents of a region if no threads are currently active in that scope. Depending on the application both blocking and non-blocking variants may be handy. But it should be noted that in either case a race condition exists when multiple threads try to enter and release a scope. This is true for RTSJ as well, and usually solved by an application specific synchronization protocol.

2.2 Nested Scopes

Scopes can be nested to create a runtime cactus stack of scopes. ScopeJ allows objects in an inner scope to refer to data allocated in an enclosing scope, but the converse is not possible as the lifetime of data allocated in an inner scope is strictly shorter than that of data allocated in its enclosing scope. If a reference into the inner scope was allowed this could lead to a dangling pointer.

Figure 3 illustrates an example of scope nesting, as well as an instance of the Multi Scoped Object design pattern [13]. The class Multi is a scope-bound class that will be allocated within the Outer scope. But, instances of Multi can be accessed from a nested scope such as Inner. It is safe to update fields (l. 11) and invoke methods (l. 12-13) on multi-scoped objects. In RTSJ programs it is necessary to classify the methods of a multi-scoped object as either scope-safe or scope-unsafe. For instance, s2() is a scope-unsafe method as it allocates an object and stores the newly allocated object in an instance field. This will fail if the current allocation context is shorter lived than the allocation context of the Multi object. In ScopeJ the invocation of s2() is type safe because of the property of scope-bound objects that any allocation performed within one their methods always uses the allocation context in which the object was created.

2.3 Permanent Classes

We assume the presence of an implicit all-encompassing scope, referred to as Top, which has a lifetime matching that of the application. Permanent classes are scope-bound classes allocated within that global scope. Permanent classes can be safely accessed from heap-allocated objects as well as from scope-allocated ones. To declare that a class belongs to the top scope, we require that it implements the marker interface Top. Permanent objects are the interface between scope code and the heap allocated, garbage collected, portions of the system. They correspond to the RTSJ concept of ImmortalMemoryArea.

2.4 Borrowed References

The strict enforcement of scope visibility constraints is too restrictive in practice. In RTSJ it is possible to establish read-only references that span regions with non-overlapping lifetimes. This is referred to as the Hand Off design pattern in [13]. This pattern is
class Bridge {
    static class Data {}
    void run(P p, Q q) { this.q = q; p.enter(this); }
    void handoff(Data d) { q.useData(d); }
}

class IntList {
    List tail; int value;

    IntList(List t, int i) { tail=t; value=i; }
}

class Outer extends Scope {
    static final class Bridge {}
    void enter(Bridge b) { b.handoff(new Data()); }
    void handoff(Data d) { q.useData(d); }
}

static final class P extends Scope {
    void enter(Bridge b) { b.handoff(new Data()); }
}

static final class Q extends Scope {
    void useData(Data d) {
        // use cross-scope reference
    }
}

Figure 4: Cross scope references with borrowing. Method useData has access to object allocated in a sibling scope.

Figure 5: Implicit Region Polymorphism. The IntList class can be allocated in any scope or in the heap.

declaration, in a straightforward fashion. Instance of IntList can be created in scope Outer (l. 6) and Inner (l. 8). The type system will keep them distinct and ensure that a reference to a list in one scope be assigned to a variable of an object in another scope. The constraints imposed by the type system are stronger for free classes than for scope-bound classes, as a variable of free type in one scope cannot be observed in an inner scope.

2.6 Type-safe Casts

Subtyping is an issue that all ownership type systems have to contend with. The tension is that ownership type systems use types to track the flow of values in a program, thus they must to retain enough type information to be able to catch a breach of encapsulation. Operationally it is clearly correct to widen the type of an expression to Object, but once this is done all ownership information (or in our previous work package information [20]) is lost. There are several solution to this: disallow widening when it entails losing ownership information (for example [20]) or add ownership parameters to Object (as in [14]). Downcasts into owned types face a different problem, since most systems work by erasure, there is no runtime ownership information to check that the cast is correct. One of the goals of ScopeJ is to allow writing code in a style that is as natural as possible. So for example, it is desirable to allow programs such as the one of Figure 6 where an array of objects is used to store (l.5) and then retrieve scope-bound objects (l. 6). This example is particularly important as it is the key to being able to reuse collection classes.

Figure 6: Type-safe casts. All of the above casts are well-typed in ScopeJ.

The program in Figure 6 is indeed type safe. The intuition is that Object[] is a free class. As such it cannot be observed from

2.5 Implicit Region Polymorphism

Up to this point, all our examples have featured scope-bound classes, the reader may rightly worry about code duplication. Is it possible to write a class that may be allocated in different scopes, or even in the heap? This is a key question for reusability. The obvious solution is to adopt parametric polymorphism. But this is has the disadvantage that existing classes must be retrofitted with type arguments. ScopeJ takes a slightly different direction. We allow some classes to be reused in different regions just as if they were parameterized with a single region type parameter. But this parameter remains implicit. We refer to such classes as free classes. Since we want to be able to reuse existing library classes we assume a simple static analysis that determines if a class is free. The only constraint that we require for a class to be considered free is that no static reference fields be accessed within the class or a method transitively call from that class.

Figure 5 illustrates the definition of a free class. Here the IntList class is a simple list of integers. It is defined outside of any scope

1 final class Outer extends Scope {
2     static class Bridge {
3         Q q;
4         void run(P p, Q q) { this.q = q; p.enter(this); }
5         void handoff(Data d) { q.useData(d); }
6     }
7 static final class P extends Scope {
8     static class Data {}
9     void enter(Bridge b) { b.handoff(new Data()); }
10    }
11 static final class Q extends Scope {
12    void useData(Data d) {
13        // use cross-scope reference
14    }
15 }
16
17 Figure 4: Cross scope references with borrowing. Method useData has access to object allocated in a sibling scope.

useful when data must be transferred between scope while avoiding copies. The solution is to relax the scope constraint to allow for temporary cross-scope references. ScopeJ allows references to scope-bound objects to be handed out outside of their defining scope provided that these references are not retained (i.e. they are used in a quasi-linear fashion). We refer to those as borrowed references. It is noteworthy that no special annotation is required to define such references.

Figure 4 illustrates borrowing. Scope P allocates a Data object (l. 9) which is an instance of a scope-bound class (l. 8). The Bridge class is used to stage a hand-off of the data to scope Q (l. 5). The control-flow start with run() which enters scope P passing a reference to the Bridge. The enter method of P will call back into the bridge with a new Data object as argument. The Bridge object will finally be able to call into scope Q with, as argument, a borrowed reference to Data allocated in P.

The type system restricts borrowed references so that it is not possible to store a borrowed reference into a field, and it is not allowed to store anything into a reference field of a borrowed object. Borrowed references can be used to read fields, invoke method with borrowed arguments, and update primitive fields.

The program in Figure 6 is indeed type safe. The intuition is that Object[] is a free class. As such it cannot be observed from

...
any other scope than a. The type system ensures any object stored in it, must have been allocated in a. Thus the downcast is safe as we know that anything retrieved from a free class has to be locally allocated.

2.7 Example: Region Pool

Figure 7 illustrates a ScopeJ program that implements the interface to a pool of region that can be used by multiple threads to process messages. The method Pool.create() is used to allocate a user-defined number of regions (l. 5). Then each time a thread needs to process a message, it will invoke Pool.run() which picks a scope and invokes Area.run() on it. The actual processing is performed within the region associated with a [i].

```java
1 final class Pool extends Scope {
2    Area[] a; int cnt;
3    void create(int i) {
4        a=new Area[i];
5        for(int j=0;j<i;j++) a[j]=new Area();
6    }
7    void run(Message m, Data data) {
8        int i=0;
9        synchronized(thin) {i=cnt++%a.length;}
10       synchronized(a[i]) {
11           a[i].run(m, data);
12           a[i].release();
13    }
14 }
15 final static class Area extends Scope {
16    void run(Message m, Data data){
17        ... // service request
18    }
19 }
```

Figure 7: Pool of Scopes.

3. THE SCOPEJ CALCULUS

The ScopeJ calculus is a core calculus for modeling region-based programs inspired by Featherweight Java (FJ) [11]. It extends FJ with features such as scopes, nested declarations, mutable state, and multi-threading. We follow our previous work on confined types, and adopt a call-by-value semantics with explicit evaluation contexts [20].

The calculus uses a special keyword to distinguish scope declarations from classes. For succinctness, the nesting of class and scope declarations is modeled by a naming convention. Any class or scope identifier is prefixed by an ordered sequence of enclosing scope identifiers. Thus the following definition

```
scope A {
    class B extends T {} desugars to
    scope top.A {
        class top.A.B extends T {} desugars to
    }
```

Here `top` stands for the name of the permanent scope. Class names, ranged over by `C`, are a (possibly empty) sequence of scope identifiers terminated by a class identifier. A scope name, `S`, consist of a sequence of scope identifiers. `T` ranges over both scope and class names.

Heap-allocated classes and free classes have an empty sequence of scope identifiers. To simplify the presentation, we will assume that all heap classes are free (which is trivially true in ScopeJ since we do not model static fields).

The ScopeJ syntax appears in Figure 8. Metavarniables `L` and `R` range over class and scope declarations, respectively. `N` ranges over method declarations, and `f`, `x`, and `n` range over field, variable and method names. Meta variables `a` and `c` range over disjoint sets of scope identifiers and class identifiers. `v` is either an object reference, `ℓ` or `null`. Scopes and classes contain both field and method declarations. The distinguished `Object` class is the root of the class hierarchy. An expression `e` can be either one of a variable `x` (including this), a value `v`, a class or scope instantiation expression `new T[i]`, a field access `e.f`, a field update `e.f := e`, a method invocation `e.m(π)`, or a cast `(T) e`.

We assume that all scope have a special method `release()` which can not be overridden. ScopeJ does not have explicit thread creation primitives, multi-threading is modeled by configurations in the dynamic semantics. Other features that have been left out include access modifiers, exceptions and reflection. Modeling these features is interesting but mostly orthogonal to our concerns.

```
R ::= scope S { T, R }
L ::= class C extends C { T, R }
N ::= T m { return e; }
e ::= x | v | new T | e.f | e.f := e | e.m(π) | (T) e
v ::= ℓ | null
T ::= C | S
C ::= S.c | c
S ::= S.s | top
```

Figure 8: Scope Calculus Syntax.

We adopt FJ notational idiosyncrasies and use an over-bar to represent a finite (possibly empty) sequence. We write `T` to denote the sequence `f_1, ..., f_n` and similarly for `π` and `T`. We write `T[i]` to denote `C_1, f_1, ..., C_n, f_n` and `T <[ ]>; Π` to denote `C_1 <[ ]>; D_1, ..., C_n <[ ]>; D_n`. Due to space consideration a number of standard FJ auxiliary definitions are omitted. We assume the presence of an implicit class table with definition for all classes and scopes. The subtyping relation `<:`, and the definitions of `fields`, `mtype`, `mbody`, and `overrides` are standard.

3.1 The ScopeJ Type System

The notion of visibility is critical to ScopeJ’s type system. We say that a type `T` is visible from another type `T'`, if values of type `T` can be referenced within the declaration of `T'`. This is written `T visible T'`. The relation is formally defined as follows:

```
c visible T  top.c visible T  S.visible S  SUFFER S  S.visible S'  S.visible S'.c'  \forall i, T, visible T
```

Free classes, `c`, as well as permanent classes `top.c` are visible in every context. A scope `S.b` is visible in the enclosing scope `S`, or at `top`. A scope `S.a` is also visible in classes defined in the enclosing scope, e.g. `S.c`. A scope bound class `S.c` is visible in any nested
scope S'. Finally we extend visibility to tuples in a straightforward way.

Function \textit{scopeof} maps a type name to its defining scope, or it if none. Moreover, we use \textit{scopeof}_T to refer to scope of a name occurring within the definition of some type T. The functions only differ in the treatment of free classes which, when used in a scope, take on their enclosing scope. \textit{scopeof}_T is the source of scope polymorphism for these classes.

\[
\text{scopeof}(S.c) = S \quad \text{scopeof}(S) = S \quad \text{scopeof}(c) = \epsilon
\]

\[
\text{scopeof}_T(S.c) = S \quad \text{scopeof}_T(S) = S \quad \text{scopeof}_T(c) = \text{scopeof}(T)
\]

ScopeJ allows the use of reference to types which are not visible in the current context, as long as these reference are consumed before the method returns. The predicate \textit{borrow}_T(T') holds if T' is a scope-bound class name which is not visible in the context of the definition of T.

\[
\text{to illustrate these definition consider the following example:}
\]

\[
\begin{align*}
\text{scope S0} & \quad \text{scope S1} \\
\text{class C}() & \\
\text{void m(D x, C y)} & \\
\text{class D}()
\end{align*}
\]

which, in desugared form, is:

\[
\begin{align*}
\text{scope top.S0} & \quad \text{scope top.S0.SI}() \\
\text{class top.S0.SI.C}() & \\
\text{class D()}
\end{align*}
\]

The scope of class C is top.S0.SI. The scope of free class D is \(\epsilon\), but when the class is used within S0, it assumes that scope. The second argument to m() refers to a that is not visible in S0, it can be used as a borrowed reference (and \textit{borrow}_T(top.S0.SI.C) holds).

### 3.1.1 Typing Classes and Methods

The type rule for a class C requires that the types of all fields be visible in the context of the class definition. A class can inherit from, either, a free class (including \textit{object}) or a scope-bound class defined in the same scope.

\[
\begin{align*}
C \text{ extends } C' & \quad T \text{ vis } C \\
C' & = c \lor (C = S.c \land C' = S.c') \\
\text{class } C \text{ extends } C' \{TT; R\} & \text{ OK}
\end{align*}
\]

(T-CLASS)

The typing rule for a scope is straightforward. All of its methods must be well-typed and fields must be visible.

\[
\begin{align*}
S & \vdash R \\
T & \text{ vis } S \\
\text{scope } S \{TT; R\} & \text{ OK}
\end{align*}
\]

(T-SCOPE)

A method of class C is well-typed if its body is well-typed within the context of C, and it is a subtype of the declared result type. In case of an override, the signatures must match (definition elided). The return type must be visible from C and all argument types must either be visible or borrowed. Finally, if the method body is a scope-bound type being cast to a free type then it must be the case that the method body is of a type declared in the same scope as C.

\[
\begin{align*}
\pi : T, \text{this : } C \vdash e : T & \quad T' \vdash : T \\
\text{override}(m, C, C') & \quad \text{viz} C \quad \forall i, \text{borrow}_T(T_i) \lor T, \text{viz} C \\
(T = c \land T' \neq c') & \Rightarrow \text{scopeof}(T') = \text{scopeof}(C)
\end{align*}
\]

(T-METHC)

The type rules for methods of scopes are almost identical, with the difference that scopes cannot override methods.

\[
\begin{align*}
\pi : T, \text{this : } S \vdash e : T & \quad T' \vdash : T \\
\text{override}(m, S, C) & \quad \forall i, \text{borrow}_S(T_i) \lor T, \text{viz} S \\
(T = c \land T' \neq c') & \Rightarrow \text{scopeof}(T') = \text{scopeof}(S)
\end{align*}
\]

(T-METHS)

### 3.1.2 Typing Expressions

A ScopeJ expression e is type checked in a type environment \(\Gamma\) and in the context of a type declaration T, written \(\Gamma \vdash_T e : T\). Variables and null value have the obvious types.

\[
\begin{align*}
\Gamma \vdash x : \Gamma(x) & \quad (T-VAR) \\
\emptyset \vdash \text{null} : T & \quad (T-NULL)
\end{align*}
\]

In FJ, the type rule of a cast expression (T) e places no constraint on e leaving it up to the runtime to check for errors. ScopeJ has to be more restrictive as runtime checks are not practical. The interesting case is when the expression e has a scope-bound (S.c) or scope (S) type and the target type is free. Consider for instance (\textit{object} new top.S0.SI.C), this expression will turn an obviously scope-bound value into a free object. In ownership type systems, this expression would be an error. ScopeJ allows it, as long as it occurs within a class that is in the same scope as top.S0.SI.C.

\[
\begin{align*}
\Gamma \vdash_T e : T & \quad (T-CAST)
\end{align*}
\]

The type rules for object creation ensure that scopes and scope-bound classes can only be allocated from code within their enclosing scope. There are no restriction on free classes.

\[
\begin{align*}
(T = S.c \lor T = S.a) & \Rightarrow \text{scopeof}(T_i) = \text{scopeof}(S) \\
\Gamma \vdash_T \text{new } T() : T & \quad (T-NEW)
\end{align*}
\]

A field selection expression e.f, occurring within the context of some type T, must abide by the normal FJ typing constraints and is subject to casting restrictions described for (T-Cast). Furthermore, if the type of e is visible in T then the type of the field must be as well. Without this, it would be possible access fields of scope
object from classes in the same scope. This would be an error, as the field of a scope object are not allocated in the same region. If the type of the receiver \( e \) is borrowed, then the field being extracted cannot be of free type or visible bound type (it must be borrowed as well).

\[
\Gamma \vdash \tau, e : \tau_r \quad \text{fields}(\tau_r) = (\mathbf{T} \mathbf{T}) \\
(T, \mathbf{v}z \mathbf{T}_r) \Rightarrow \tau, (\mathbf{v}z \mathbf{T}_r) \Rightarrow \mathbf{borrow}_\tau(\mathbf{T}_r) \\
(T_i = c_i \land \tau_r \neq c_i) \Rightarrow \text{scopeof}(\mathbf{T}_r) = \text{scopeof}(\mathbf{T}_i)
\]

\[
\Gamma \vdash \tau, e.f_i : \tau_i \\
(T-FIELD)
\]

Consider, the following example:

```
scope \text{SO} { 
    \text{class A} { 
        \text{void} m(SI x, y) \{ \ldots x.f \ldots y.b \ldots y.a \} 
    } 
    \text{class SI} { 
        \text{class B} \{ \text{A a; B b;} \} 
    }
}
```

Focusing on the method \( m() \), which has three field accesses. The first one is invalid because the type of \( x \) is \( x \) which is not visible in scope \( \text{SO} \), even if \( x \) is. The access to field \( y.b \) is legal, as the type of \( x \) is borrowed and so is the type of \( x.b \). Finally the expression \( y.a \) is invalid as it tries to access a type that is visible in \( A \) off a borrowed variable.

An update expression, \( e.f_i := e' \), is well-typed if neither the type of the receiver \( e \) or the assigned value \( e' \) are borrowed. Moreover if the type of the field \( f_i \) is free then the scopes of \( e \) and \( e' \) must match.

\[
\Gamma \vdash \tau, e : \tau_r \quad \text{fields}(\tau_r) = (\mathbf{T} \mathbf{T}) \\
\quad \text{scopeof}(\mathbf{T}_r) = \text{scopeof}(\mathbf{T}_i) \\
\quad \Gamma \vdash \tau, e.f_i := e' : \tau_i \\
(T-UPD)
\]

An invocation expression, \( e.m(\mathbf{e}) \), is well-typed under the following condition. If the type or the receiver \( e \) is visible in the current context, then so must the return type of \( m \). If the receiver is borrowed, then so must be the return type. The casting restriction of T-Case applies. A receiver of borrowed type can only have parameters of borrowed type, while argument of borrowed type can only be passed to parameter of borrowed type. If a parameter is of free type, then the corresponding argument must either be in the same scope as the receiver or be of free type.

\[
\Gamma \vdash \tau, e : \tau_r \quad \text{mtype}(\mathbf{a}, \tau_r) = \mathbf{T} \rightarrow \mathbf{T} \\
\quad \Gamma \vdash \mathbf{a} : \mathbf{T} \\
\quad (\mathbf{T}, \mathbf{v}z \mathbf{T}_r) \Rightarrow \mathbf{T}, (\mathbf{v}z \mathbf{T}_r) \Rightarrow \mathbf{borrow}_\mathbf{T}(\mathbf{T}_r) \\
\quad (\mathbf{T} = c \land \tau_r \neq c_r) \Rightarrow \text{scopeof}(\mathbf{T}_r) = \text{scopeof}(\mathbf{T}_i) \\
\quad \forall \mathbf{i}, (\mathbf{borrow}_\mathbf{T}(\mathbf{T}_i)) \Rightarrow \text{scopeof}(\mathbf{T}_r) = \text{scopeof}(\mathbf{T}_i) \\
\quad \forall \mathbf{i}, (\mathbf{T}_i = c_i) \Rightarrow \text{scopeof}(\mathbf{T}_i) = \text{scopeof}(\mathbf{T}_r) \\
\quad \Gamma \vdash \tau, e.m(\mathbf{e}) : \mathbf{T} \\
(T-INVK)
\]

Taken together the ScopeJ type rules enforce visibility constraints on types and restrictions on how fields and methods can be used. Scopes can only be accessed in their defining context. Scope-bound classes can be access in their defining context and from all nested classes and scopes. These visibility constraints are sufficient to prevent object reference from leaking to regions with potentially longer lifetimes. One of the surprising features of the type system is that a scope is visible in its defining context but not from classes nested within it. Since a scope type is only visible from its parent scope, a thread is forced to enter the scopes following the hierarchy of scope definition. This is sufficient to prevent RTSJ’s ScopeCycleExceptions.

### 3.2 Dynamic Semantics

The dynamic semantics of the scope calculus is given in Figure 9 in terms of a two-level small-step operational semantics. A ScopeJ configuration is a pair \( \sigma, P \) where \( \sigma \) is partial map from location to objects and a program \( P \) defined as the parallel composition of a set of threads: \( P = \kappa \parallel P' \). The main reduction relation has thus the form, \( \sigma, P \xrightarrow{\kappa} \sigma', P' \), and determines the behavior of a program as a whole. Each thread is modeled by a call stack \( \kappa \) which can be either empty, \( \epsilon \), or a sequence of frames. A frame is a pair, \( \ell, e \), of a scope instance and an expression. We use \( \ell \mathbf{e} \) to represent a call stack of the form \( \epsilon \bullet \ell_0 \mathbf{e}_0 \bullet \ldots \bullet \ell_n \mathbf{e}_n \).

Rule (G-STEP) evaluates the top frame of the call stack. Rules (G-ENTER) and (G-RET) manage the call stack. When a thread executes \( \ell.m(\mathbf{e}) \), the thread enters the active scope of \( \ell \) by pushing a frame onto the stack. A thread removes the top of its call stack if the top frame only contains a value. The rule used to clear a scope, (G-REL), sets all field of the scope instance to \( \text{null} \) and removes all objects allocated in the scope and its inner scopes from the store. It use the predicate \( \text{refcount}(\ell, P) \) to check whether a scope instance \( \ell \) is entered by any of the threads in \( P \). We adopt a non-blocking semantics for release to simplify the proof (with a blocking semantics it is necessary to account for the potential of deadlocks).

The dynamic semantics abstracts some of the low-level details of region-based memory management. In particular, we do not model the way memory is reused after a region is freed. This is not necessary for our purposes as the property we are interested in is the absence of dangling pointers.

The global evaluation rules rely on the notion of evaluation contexts, which are as usual expressions \( E[\mathbf{e}] \) with a hole. The syntax of method and assignment contexts enforce left-to-right evaluation order and call-by-value semantics. Furthermore evaluation contexts are deterministic. For any expression \( e \), there is exactly one evaluation context. This can be shown by easy induction on the structure of \( e \) [20]. Determinacy is central to our treatment of calls and returns.

Expression evaluation is defined by a relation of the form \( \sigma, \ell_1, e \rightarrow \sigma', \ell_2, e' \) where \( \sigma \) is a store mapping locations \( \ell \) to instances and \( \ell \) is the scope of the allocation context. Each class or scope instance, \( \mathbf{T}(\mathbf{v}) \), is annotated with the scope instance \( \ell \) in which it was allocated. The expression rules are mostly standard. In (R-NEW), the current allocation context is tagged onto the newly allocated instance. To apply Rule (R-NEW), a reduction step must satisfy the invariant \( \text{StoreInv}_{\mathbf{T}}(\ell) \) such that a bound class instance may
only be allocated in an instance of its scope, while a scope instance may only be allocated in an instance of its parent scope. Note that we assume that the location \( \ell \) used in the assignment is globally unique and that location are never reused. In (R-UPD), field update must maintain invariant \( \text{StoreInv_2}_a(\ell_i) \), which means that if \( \ell \) is assigned to a field of \( \sigma(\ell_i) \), then \( \ell_i \prec_\sigma \ell' \) where \( \ell_i' \) and \( \ell' \) are the allocation scope of \( \ell_i \) and \( \ell \) respectively. In other words, \( \ell \) must be in a scope of longer lifetime than that of \( \ell_i \). This invariant represents the absence of IllegalAssignmentError in RTSJ. In (R-INVK), method invocation incurs locating the current allocation context \( \ell_a \) by looking up the active scope of the receiver \( \ell \) (so that \( \text{inscope}_\sigma(\ell_i) = \ell_a \)). That is, \( \ell_a \) is \( \ell \) if \( \ell \) is a scope instance, and \( \ell_a \) is the allocation scope of \( \ell \) otherwise.

### 3.3 Properties

The correctness property we are interested in is that ScopeJ programs never access a dangling pointer. The type system enforces a stronger property as it prevents dangling pointers altogether. To prove this, we need a notion of well-typed store, and must show that a well-typed program preserves well-typedness of the store. We say that a configuration \( \sigma, P \) is well-typed if the store \( \sigma \) is well-typed (written \( \vdash \sigma \)) and the threads are well-typed under \( \sigma \) (written \( \vdash \sigma, P \)).

A well-typed store satisfies the invariants that if \( \ell \) refers to an object, then the field of \( \sigma(\ell) \) refer to null or other objects allocated in either the same memory region or a region associated with an ancestor scope. The fields of the object are well-typed and if a field is of a free type, then the target object must have been allocated in the same region.

**Definition 1 (Store and Object).** A store is well-typed if each of its objects is well-typed.

\[
\forall \ell \in \text{dom}(\sigma), \quad \sigma \vdash \ell
\]

\[
\sigma(\ell) = T^\ell(\tau) \quad (T = \text{top}) \lor \text{StoreInv}_1(\ell) \lor \text{StoreInv}_2(\ell)
\]

\[
\text{inscope}_\sigma(\ell_i) = \ell' \quad \text{fields}(T) = (TT) \quad \forall i, \nu_i = \text{null} \lor (\nu_i(T_i) = T_i^\ell(\tau') \land T_i' \prec_\sigma < T_i \land (T_i = c_i \Rightarrow \ell' = \ell_i))
\]

\[
\sigma \vdash \ell
\]

The invariant \( \text{StackInv}_\sigma(\kappa) \) checks that each reference in the top frame \( \ell_i, e \) of a call stack \( \kappa \) refers to an object allocated in either \( \ell_i \) or the allocation scope of a previous frame. That is, objects referenced in this frame are allocated in scopes that the thread has already entered. In addition, \( \sigma \vdash \ell_i \), \( e \) checks that each reference in \( e \) must either be of a type considered borrowed in the scope \( \ell_i \), or be of a type visible in the scope \( \ell_i \) and in which case, be allocated in the same or outer scope of \( \ell_i \). Moreover, free objects in \( e \) are always allocated in \( \ell_i \). We will show that the stack invariant holds after each reduction step so that there is no dangling pointers in any thread after clearing the scope.

**Definition 2 (Stack Invariants).**

\[
\sigma(\ell_i) = T_i^\ell(\tau) \quad \text{for all } \ell \in e, \quad \sigma(\ell) = T^\ell(\tau)
\]

\[
(\ell = \ell_i) \lor \text{borrow}_\tau(T_i) \lor (T \text{ inv } T_i \land \ell_i \prec_\sigma \ell')
\]

\[
(T = c) \Rightarrow \ell_i = \ell'
\]

\[
\sigma \vdash \ell_i, e
\]

\[
\text{StackInv}_\sigma(\kappa \cdot \ell_i, e)
\]

A thread is well-typed if its call stack \( \kappa \) is well-typed with respect to a store \( \sigma(i.e., \sigma \vdash \kappa) \). The condition recursively checks whether the stack invariant holds for each frame in \( \kappa \) so that each object referenced in that thread is allocated in a scope that the thread has already entered. In addition, if \( \ell_i, e \) is not the last frame in \( \kappa \), then the \( e \) must correspond to the return expression of a method call in the previous frame \( \ell_i, E[\ell.m(\tau)] \), while \( \ell_i \) is the active scope of \( \ell \).

**Definition 3 (Threads).** A set of threads \( P \) is well-typed with store \( \sigma \), if each of the thread is well-typed.

\[
P ::= P' \mid \kappa \quad \sigma \vdash P \quad \sigma \vdash \kappa
\]

\[
\sigma \vdash \kappa \quad \text{StackInv}_\sigma(\kappa \cdot \ell_i, e)
\]

A thread containing call stacks \( \kappa \) is well-typed if the expression in each of its stack frame is well-typed.

\[
\kappa ::= \kappa' \cdot \ell_i, e
\]

\[
\kappa \vdash e \quad \text{StackInv}_\sigma(\kappa \cdot \ell_i, e) \\
\sigma(\ell_i) = T_i, e(\tau)\quad \sigma \vdash \ell_i, e: T_i
\]

\[
\sigma = \epsilon \lor (\kappa = \kappa' \cdot \ell_i, e', E[\ell.m(\tau')]) \quad \quad \text{inscope}_\sigma(\ell_i) = \ell_i,
\]

\[
\sigma(\ell_i) = T_i, e(\tau'), \sigma \vdash \ell_i, e: T_i', T_i' \prec_\sigma, (T' = c' \land T' \neq c) \Rightarrow \text{scope}(T) = T_i
\]

\[
\sigma \vdash \kappa \cdot \ell_i, e
\]

In a well-typed call stack, the expression \( e \) in each stack frame is also well-typed. The type judgment for well-typed runtime expression has the form \( \sigma \vdash e : T \), where \( \sigma \) is used to retrieve types of object references so that \( \sigma(\ell) = T^\ell(\tau) \). The type rules for runtime expressions are identical to those for static expressions with type environment \( \Gamma' \) replaced by \( \sigma \).

We assume that all classes in the class table are well-typed. The below lemma proves that the evaluation of an expression in one step preserves typing and a free type expression may only be reduced to other expression of free type or bound type defined in the scope of the current execution context.

**Lemma 1.** If \( e \not\equiv v.m(\tau), \sigma \vdash e, \sigma \vdash \ell_i, \epsilon \), then \( \sigma(\ell_i) = T_i(\tau), \sigma \vdash \ell_i, e: T_i, \sigma \vdash \ell_i, e \Rightarrow \sigma', \sigma' \vdash \ell_i, e': T_i', T_i' \prec_\sigma, (T' = c \land T' \neq c), \text{ then scope}(T') = T_i \).

The subject reduction lemma shows program execution preserves typing and releasing memory of a scope will not create dangling pointers in the program.

**Lemma 2 (Subject Reduction).** If \( \sigma, P \sim \sigma', P', \sigma \vdash \sigma, \sigma \vdash P, \text{ then } \sigma' \vdash e \text{ and } \sigma' \vdash P' \).

The following predicates define cast exception and null pointer exceptions in an expression.

\[
e = (T \ell) \quad \sigma(e(\tau)) = T'(\tau) \quad \text{T' } \not\subseteq T
\]

\[
\text{CastException}_\sigma(\epsilon)
\]
null pointer exception or cast exception.

Release scope memory:

\[ (\sigma(\ell) = T^{\ell}(\tau) \wedge \text{inscope}_{\sigma}(\ell) = \ell') \Rightarrow (\forall \ell' \mid v_i = \ell_i \wedge \sigma(\ell_i) = T^{\ell_i}(\tau) \Rightarrow \ell' <_{\sigma} \ell_i) \]

Store invariants:

\[ (\sigma(\ell) = T^{\ell}(\tau) \wedge \sigma(\ell_a) = S^{\ell}(\tau')) \Rightarrow (T = S.c \lor T = S.s \lor T = c) \]

\[ \text{StoreInv}_1(\ell) \]

Reference counts:

\[ \text{refcount}(\ell, e) = 0 \]
\[ \text{refcount}(\ell, (P \mid P')) = \text{refcount}(\ell, P) + \text{refcount}(\ell, P') \]
\[ \text{refcount}(\ell, (\kappa \bullet \ell, e)) = \text{refcount}(\ell, \kappa) \quad \text{if } \ell \neq \ell' \]
\[ \text{refcount}(\ell, (\kappa \bullet \ell, e)) = 1 + \text{refcount}(\ell, \kappa) \]

Reference counts:}

\[ e = \text{null}\mathbf{m}(\tau) \mid \text{null}\mathbf{f} \mid \text{null}\mathbf{f} := v \]

\section{Related Work}

Region-based memory management was introduced by Tofte and Talpin [17] and originally implemented for the ML programming
language. Region systems organize memory in a stack of regions and use a combination of polymorphism and effects to indicate the allocation context of expressions and the regions they may affect. In the ML family of region-based systems, regions are single threaded and lexically scoped. Furthermore regions are not values, they cannot be stored or shared. Straightforward extension to Java have been investigated in [5, 18]. Language features such as multi-threading do not easily fit in the lexically scoped region model. Our scope calculus could be viewed as making region first class entities and allowing them to be entered multiple times (by different threads). Each scope can be considered as a wrapper around a letregion ρ expression such that ρ is only in scope for definition nested within the scope. Each class can then be parameterized by a set of region parameters, one per enclosing scope. Method effects can be approximated by the set of scopes visible to the defining class. To the best of our understanding there is no obvious way to encode borrowed types in a region-based system. Hallenberg et.al. investigated the integration of garbage collection and regions [9], but their goal was different from ours as they wanted to garbage collect regions. Cyclone is a type-safe C-like language with support for lexically scoped regions [7]. The compiler does not support multi-threading, but plans for a concurrent extension have appeared in [6]. In that version regions are still lexically scoped, region sharing is a side-effect of spawning a thread while in a region. In the scope calculus, a thread can join a scope at anytime by invoking one of its methods. Hicks et.al. report on an extension with unique pointers and a form of borrowing [10]. Unique pointers can be used to, for example, relax the LIFO lifetimes of regions.

Borrowed references have been studied by, e.g., Boyland [3] and can also be viewed as a generalization of the concept of anonymity found in [20]. Boyapati et al. have proposed an ownership type system for the RTSJ [2], later Chik et al. proposed a region inference for a similar language [4]. The proposal is comprehensive as it allows to express all flavors of RTSJ regions and different variants of real-time threads (heap, noheap). Their approach relies on explicit ownership type parameterization. Every type is parameterized by one or more region parameters. The drawback of the system is that existing Java code cannot be incorporated (the parent class Object would require a region parameter) and also that region handles must be passed around explicitly in order to determine where an object is to be allocated. In ScopeJ no parameters are required and region handles are implicit (the scope objects). Their system did not allow borrowed references which we have found omnipresent in the RTSJ programs we have experience with. And importantly, object arrays cannot be used. In Java an array X[] cannot be parameterized and, this even with Java Generics, it is not possible to write new X<R> [10]. ScopeJ allows the use of reference arrays thanks to its implicit polymorphism.

5. CONCLUSIONS
This paper has introduced scopes, a new programming abstraction for highly responsive systems. We have presented a type system that ensures no memory error can occur at runtime due to manual scope deallocation. Scopes are GC safe, the type system ensures that they cannot refer to data allocated on the heap, or trigger garbage collection. The design of the ScopeJ has taken great care to be non-intrusive, requiring no changes to the Java language. ScopeJ can be implemented as an additional bytecode processing pass on top of Java 1.4 real-time virtual machine without any changes to the runtime system.

6. REFERENCES