On the connection between functional programming languages and real-time Java scoped memory

Delvin C. Defoe [*]
Dept. of Computer Science and Software Engineering
Rose-Hulman Institute of Technology
Terre Haute, Indiana
defoe@rose-hulman.edu

Rob LeGrand Ron K. Cytron
Dept. of Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri
{legrand, cytron}@cse.wustl.edu

Abstract

Java has recently joined C and C++ as a relatively high-level language suitable for developing real-time applications. Java's garbage collection, while generally a useful feature, can be problematic for real-time applications if collection occurs with unpredictable frequency and latency.

The **Real-Time Specification for Java**^{TM} (RTSJ) incorporates a *scoped-memory* model, akin to *regions*, that is not subject to garbage collection. However, applications are subject to strict rules concerning how objects can reference each other in scoped memory. Unfortunately, almost all extant Java code, including Java's vast and useful runtime libraries, will not execute properly in scoped-memory areas without significant modification.

In this paper, we show that programs written in a pure functional programming language can be executed in a provably safe manner using scoped memory in RTSJ. This new connection allows extant implementations of important abstract data types to migrate safely to RTSJ. We also explore the effect of RTSJ's referencing rules on the asymptotic, real-time behavior of some abstract data types.

Categories and Subject Descriptors D.3.4 [*Processors*]: Memory Management; D.3.2 [*Storage Management*]: Garbage collection

General Terms Algorithms, Management, Memory, Real-time, Data structures

Keywords Scoped Memory, Real-time Java, Memory Management, Programming Languages, Performance Analysis, Functional Programming

1. Introduction

Real-time applications require predictable memory-management performance, which includes the cost of storage allocation, deallocation, and (for the relevant languages) the frequency and latency of garbage collection. While real-time applications could be written in languages that do not offer garbage-collection capabilities, applications are often a mixture of hard and soft real-time components. Since its advent in 1995 [1, 11], developers have been exploring ways to use Java for real-time programming, including its garbage collection, to the extent that it does not preclude real-time performance.

To broaden the use of Java as a real-time programming language, the Real-Time for Java Expert Group issued the RTSJ [5], which provides extensions to Java in support of real-time programming without changing the basic structure of the language. Instead, new class libraries were added to realize real-time features (such as periodic threads and timers) and some extensions were introduced to the Java Virtual Machine (JVM) to enforce the memory-management properties of the RTSJ discussed below.

In addition to the heap, where dynamic storage allocation occurs, the RTSJ includes specialized memory areas for dynamic memory allocation: *immortal memory* and *scoped-memory areas* [5]. Objects allocated in those memory areas are never subjected to garbage collection. Objects in immortal memory are collected only when the application terminates, regardless of their liveness. Objects allocated in a scoped-memory area are collected *en masse* when all threads that entered the scope exit it. Scoped-memory areas are best used with **NoHeapRealtimeThread** (NHRT) [5], which are guaranteed not to be preempted by garbage collection.

Use of scoped-memory areas is not without additional cost or burden, as references between objects in the RTSJ's memory areas are subject to certain restrictions. Failure to abide by these restrictions may result in runtime exceptions in the offending thread(s), from which recovery is not possible. While the RTSJ enables Java code that avoids garbage collection to be written, porting or authoring code in the RTSJ can be awkward. In Section 3, we show that there is generally no guarantee that an RTSJ application will execute without scope-referencing errors.

While a space-efficient scoped-memory scheme can be derived for a given run of an RTSJ application [10], that scheme may be unsuitable for other executions of the same application. Static analysis [8] can determine scopes suitable for all executions, but the analysis is necessarily approximate and the resulting scopes may not accommodate sufficient storage reclamation.

Remediation can be found using subsets [14] of RTSJ, annotations attached to RTSJ programs [6], or a real-time garbage collector [4]. Even at its best, each of those approaches has its skeptics, and there are (hard real-time) applications for which developers insist they must avoid automatic garbage collection. Some among these developers use RTSJ in *lieu* of garbage collection.

This paper articulate a correspondence between the functional programming paradigm and the RTSJ that allows extant code in the functional programming paradigm to be moved to RTSJ's scoped-

[*] This author contributed to this work while at Washington University in St. Louis.

memory model without fear of runtime errors. Such correspondence may be insightful to developers who insist on using RTSJ. The rest of the paper is organized as follows. Section 2 presents RTSJ's scoped-memory areas. Section 3 presents our main result: programs written in a pure functional programming language can be executed safely in RTSJ scoped-memory areas. Sections 4, 5, 6, 7 and 8 provide analysis of scoped-memory areas. Section 9 offers concluding remarks and Section 10 acknowledges those who have assisted in preparation of the final draft of the paper.

2. RTSJ's memory management

One of the most interesting features of the RTSJ is its memory management model based on *scoped-memory areas* (or scopes for short) [17]. A scope is a fixed-size, contiguous *region* [18] of memory, with a reference count of the number of threads active in the region. This memory model ensures developers of timely reclamation of objects and predictable memory performance. But this comes at the cost of learning an unfamiliar programming model—a restrictive model that relies on the use of scopes. These new scopes were designed to meet two very important requirements [17]: providing predictable allocation and deallocation performance, and ensuring that real-time threads do not block when memory is reclaimed by the virtual machine.

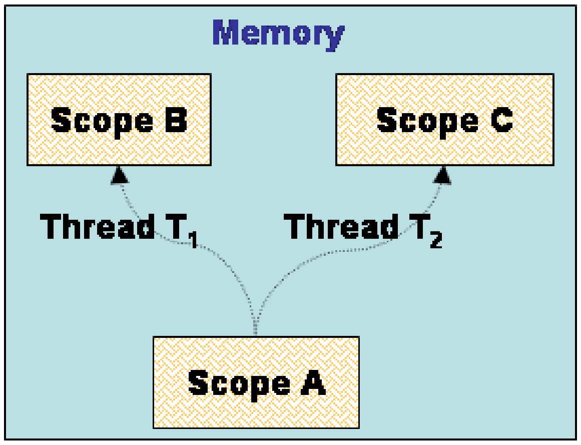

Figure 1. Scoped-memory single-parent rule. *A* is the parent of or outer scope for both *B* and *C*

To meet these requirements, the RTSJ ensures that objects in a scope are not deallocated individually. Instead, the entire scope is collected *en masse* when all threads that entered the scope exit it. Since a scope is a fixed-size region of memory from which objects are allocated, at instantiation the size of a scope is specified in bytes and cannot be changed. Each scope can be entered by multiple threads and each thread can allocate objects within a scope it has entered and communicate with other such threads by shared variables. A new scope can also be instantiated by a thread executing within its current scope. This behavior is known as the nesting of scopes and such nesting is controlled by the order in which threads enter the scopes.

Consider Figure 1 for example, which shows two threads, each of which has entered Scope A before entering any other scope. When Thread T_1 enters Scope B, A becomes B's parent. Similarly, T_2 causes A to become C's parent. RTSJ allows a scope to have only one parent. Thus, Thread T_1 cannot enter Scope C unless it first exits Scope B, an action that renders Scope B thread-free and thus collectible. Inter-scope references are allowed only from a descendant scope to an ancestor scope. As depicted in Figure 1, an object in Scope B could reference any object in Scope B or Scope A, but references between scopes B and C are not allowed, nor are references from A to either B or C.

To take advantage of scopes, the RTSJ defined a new type of thread called NHRT. NHRTs cannot allocate objects in the garbage-collected heap, nor can they reference objects in the heap. These constraints were added to prevent NHRTs from experiencing delay due to the locking of heap objects during garbage collection [10]. NHRTs have the highest priority among all threads and can preempt even the garbage collector.

Table 1 summarizes the referencing constraints placed on objects in certain memory areas. These constraints do not apply only to objects, but also to threads so that real-time threads do not block when the JVM reclaims objects.

3. Functional programs and RTSJ

Programs written in RTSJ can avoid garbage collection, but they may suffer from exceptions caused by RTSJ-inappropriate memory accesses and single-parent-rule violations. In this section, we articulate a new and interesting relationship between RTSJ programs and *functional* programs. The result of our findings offers RTSJ developers some relief in migrating extant functional implementations of popular data structures to RTSJ.

We say an RTSJ program P is *scope-safe* if no execution of P can issue any `illegalAssignment()` or `memoryCycle()` exceptions. Such exceptions are issued if the program fails to follow the scope-access rules discussed in Section 2.

THEOREM 3.1. *Static determination of the scope-safety of an RTSJ program is undecidable.*

Proof: *By reduction from the halting problem:* Given an encoding of a Turing machine T and its input w, we construct an RTSJ program P as follows:

- P simulates T on w by interpreting T in standard Java: no RTSJ features are used.
- If T should halt on w, then P instantiates two scoped-memory areas, A and B, where A is the parent of B. P next issues a reference from A to B.

Clearly, P generates an `IllegalAssignmentError` if and only if T halts on w. Thus, deciding (statically) that P halts also decides that T halts on input w, which contradicts the undecidability of the halting problem. ■

Theorem 3.1 implies that a compiler cannot generally detect programs that would execute without error in Java but fail due to scope errors in RTSJ. Extant responses to this problem can be summarized as follows:

- A program can be written in a subset of RTSJ that provably avoids scope errors [14], or annotations can be attached to RTSJ programs so that a compiler can reason about scope-safety [6].

  While this approach can be successful, an application must essentially be rewritten to conform with restrictions or to supply annotations. Moreover, a developer must understand the application at a depth sufficient to modify the application correctly.

  Java's extensive libraries offer significant functionality for developers, but they are inherently unsuitable for use in RTSJ's scoped-memory areas. Rewriting the libraries for RTSJ is a daunting task, with no real guarantee of correctness or efficiency.
- Scopes can be avoided by using ordinary Java with a real-time garbage collector [4]. While this approach avoids having to rewrite an application, certain program properties must be asserted or analyzed [12] to configure the automatic garbage

Objects in	Reference to Heap	Reference to Immortal	Reference to Scoped
Heap	Allowed	Allowed	Not allowed
Immortal	Allowed	Allowed	Not allowed
Scoped	Allowed	Allowed	Allowed if same, outer, or shared scope

Table 1. Reference rules for RTSJ memory areas [5]. Objects in the heap are allowed to reference objects in immortal memory, but not objects in scoped memory. Bollella *et. al* [5] discuss outer and shared scopes in detail.

collector so that it sufficiently paces the application's storage needs. Some time efficiency will be lost, as a predictable share of the CPU must be given to the garbage collector. Some space efficiency is also lost, as the heap must be sufficiently over-provisioned to mitigate the collector's share of the CPU.

Even at its best, this approach has its skeptics, and there are (hard real-time) applications for which developers believe they must avoid automatic garbage collection. But this approach works well provided that the pause times (currently in the milliseconds) are acceptable [3] and that certification is possible with the collector.

As an alternative to modifying Java programs to be RTSJ-safe, we consider an apparently different programming paradigm and show that programs written in that paradigm can be easily moved to RTSJ and enjoy scope-safety.

Functional programming languages have emerged as an alternative to the more prevalent style of programming languages (including Java and RTSJ) in which state, and mutation of state, dominate the design and construction of programs. Lisp [13] is perhaps the earliest example of a practical functional programming language still in use today, and Backus's Turing lecture [2] inspired generations of research on functional programming languages.

The property of a pure Lisp program most relevant to our work concerns its mathematical transparency: Lisp expressions can be manipulated mathematically, because the symbols of a symbolic expression cannot change value unexpectedly. Languages like pure Lisp achieve this property by allowing names to be associated with expressions at most once. This "single assignment" rule allows mathematical substitution of a program's names but also implies the following property: in terms of the order of assignment of expressions to names, the expression assigned to a given name can reference only those names that are strictly older than the assigned name. We leverage that property to build scope-safe RTSJ functions.

We use Lisp as an example, but extensions to other pure functional programming languages are straightforward. Memory is allocated in Lisp programs by a `cons` operator, which creates a memory cell containing at most two references to extant storage. We realize a Lisp program's storage allocation in RTSJ as follows:

- The RTSJ program prepares to simulate the Lisp program by creating a NHRT in the usual manner. The details need not be provided here, except to say that the program is subsequently able to create scoped-memory areas.
- Each `cons` operator in the Lisp program is simulated by creating and entering a new scoped-memory area with sufficient storage for two references (which we assume could also accommodate non-reference data such as constants). The references for a `cons` cell must be known in the Lisp program when the `cons` cell is instantiated; we populate the RTSJ scope with precisely those references.

Careful adaptation of functional code is required for code migration to RTSJ to be worthwhile. While the assumption above of one-scope-per`cons`-cell is inefficient in practice, this simplifying assumption allows for easy exposition and analysis. This assumption also allows us to reason about the nature of storage allocated in the corresponding RTSJ program:

- No scoped-memory area will overflow. This follows from the construction of the scoped-memory areas. Each is populated once and for always by the single `cons` cell that prompted creation of the scope.
- All references created in this manner are scope-safe, as proven by the following theorem.

THEOREM 3.2. *The RTSJ realization of a Lisp program is scope-safe.*

Proof: *By contradiction:* If a scope-referencing error occurs, one of the following must be its cause:

- A reference is made to scoped memory from an unsuitable memory area (the generic heap). If so, then the program did not launch the NHRT as described above.
- An inappropriate reference is made between scoped-memory areas. There are two cases:
  - The areas are not in an ancestor-descendant relationship. This is a contradiction, since all scopes are created with linear ancestry.
  - The reference is made from an ancestor scope to a descendant scope. This is a contradiction, since the functional program can only have newer cells reference older cells.

■

An important consequence of Theorem 3.2 is that an RTSJ developer can consider migration of extant code written in a pure functional language for deployment under RTSJ, without fear of scoped-memory referencing errors at runtime. As described in Sections 5–8, data structures such as lists and heaps can be implemented in scopes based on their realization in a functional programming language.

Code migrated as described above creates a linear chain of scopes—one for each `cons` cell. However, the thread entering those scopes never retreats, so the resulting program never deallocates storage. In practice, some form of storage reclamation is necessary. Due to space limitations, we summarize our approaches as follows:

- At any moment, the RTSJ application could suspend its primary activity and enter a phase in which it traces program references through the scope chain, copying the resulting objects into a new chain of nested scopes. Any objects not referenced by the program would not be copied. Such a phase essentially emulates a copying garbage collector, but the intent of using RTSJ with scoped-memory areas is to avoid garbage collection.
- The nature of object allocation in pure functional programs guarantees that objects cannot have cyclic references. Therefore, reference-counting can sufficiently identify scopes that are no longer in use. The scope chain can then be compacted by copying as described above.
- Allocating scopes in a chain guarantees that a new object will be able to reference any previously created object, but this ap-

proach is needlessly conservative. Since in a functional program we know which object a new object will reference when it is created (and since references are immutable) we can allocate its scope accordingly: When a new object is created, if it is a non-referencing object or it references nil, it can be placed in a new child scope of a common-root parent scope; if it references another existing object, it can be placed in a new child scope of that object's scope. This approach would create a tree of scopes rooted at one common parent scope. An entire scope chain could then be reclaimed without having to trace or copy a large chain. This approach is closely related to *contaminated garbage collection* [7].

Thus, while the pure-functional implementations can serve as a basis for code migration, more work is necessary to obtain space-efficient RTSJ implementations. In Sections 5–8, we consider liveness issues for each particular data structure, and each is mindful of reclaiming storage where possible. Generally, liveness of a given data structure, in the context of a real application that uses multiple data structures, must be considered to determine a more sophisticated scope structure and to determine when scopes should be exited so that storage can be reclaimed.

In addition to the storage-reclamation problem, a data structure migrated without due consideration may be unsuitable for a real-time application. The rest of this paper provides examples that illustrate the advantages and pitfalls of code migration for real-time applications.

4. Methodology

As an example of migrating functional language implementations to RTSJ, we consider some of the data structure implementations due to Okasaki [15, 16]. Because they were developed for general use, and without regard to real-time requirements, the primary consideration of merit was the normative *average* running time, analyzed over a typical usage pattern.

For some of the data structures (such as the stack), deployment in a real-time context is appropriate. In such cases, the data structure's average running-time complexity is an indication of the *worst-case* running times that might be seen over the life of the data structures. However, for others (such as the queue), implementation in a functional programming language can have reasonable average-case performance with occasionally poor worst-case performance. Because real-time applications must budget for worst-case conditions, it is important to analyze a data structure's migration from the functional programming paradigm to RTSJ with an understanding of the resulting asymptotic worst-case behavior.

In Sections 5 and 6 we suggest particular RTSJ implementations for the stack and queue abstract data types and analyze their time-complexities. RTSJ scopes (and functional programming languages) essentially behave in a stack-like fashion. As such, an RTSJ implementation for stack follows naturally. Our RTSJ queue implementation was inspired by Okasaki's [15] functional implementation and time-complexity analysis. Both have been carefully crafted to have properties desirable for real-time applications. These results were also documented by Defoe *et. al.* [9].

In Sections 7 and 8 we use the transformation described in Section 3 to migrate functional programming implementations of data structures and their time-complexity analyses to RTSJ.

5. Stack analysis

Here we present a scoped-memory implementation of the stack abstract data type and analyze it. A *stack* is an ADT that operates on the *Last-In-First-Out* (LIFO) principle. One end of a stack, called the top-of-stack, is used for each stack operation. The fundamental operations are:

1. IS-EMPTY(S) - an operation that returns the binary value **TRUE** if stack S is empty, **FALSE** otherwise.
2. PUSH(S, x) - an operation that puts element x at the *top* of stack S.
3. POP(S) - an operation that removes the element at the top of stack S and returns it. If the stack is empty the special pointer value $NULL$ is returned. $NULL$ is used to signify that a pointer has no target.

We are most concerned about implementing stack in scoped memory. However, we first present an implementation in heap memory for the sake of comparison.

5.1 Typical implementation of stack

Several data structures, including the singly linked list, can be used to implement a stack in the heap. The IS-EMPTY operation checks whether the top-of-stack points to NULL. The PUSH operation adds a new element to the top-of-stack, and the POP operation updates the top-of-stack and returns the topmost element. Each of these fundamental operations requires $T(n) = O(1)$ time using a singly-linked list.

5.2 Scoped-memory implementation of stack

For a scoped-memory implementation of stack we make the following assumptions:

1. Each application A that manages a stack S is fully compliant with the RTSJ.
2. A has a single thread T_a, which is an instance of NHRT. RTSJ allows multiple threads to share data structures as long as all threads enter scopes in the same order. This simplifying assumption of a single thread is made only for easy exposition and analysis.
3. A executes on an RTSJ-compliant JVM.
4. T_a can legally access stack S and the elements managed by S.
5. Before an element x is pushed on stack S, a new scope s is first instantiated to store x, and T_a enters s.

Assumption 5 is relevant for the purpose of complexity analysis. Although we do not suggest one scope per element in an actual implementation, here we are concerned about worst-case analysis. Storing a single element in a scope simplifies analysis and yields the smallest amount of unnecessarily live storage in scoped memories for this particular data structure. Pseudocode and analysis for the fundamental stack operations follow.

5.2.1 IS-EMPTY

We assume that there is a TOS field in each scope that points to the top-of-stack element. If the TOS field in the current scope points to the stack object S (a sentinel used for indicating the empty stack), then the application thread T_a is executing in the scope containing S. Thus, S contains no elements, so the stack is empty. If c_1 is the time required to execute line 1 of IS-EMPTY then the worst-case running time of IS-EMPTY is $T(n) = c_1 = O(1)$.

```
IS-EMPTY($S$)
1 return $TOS = S$
```

Figure 2. Procedure to test if the stack is empty—scoped-memory implementation.

5.2.2 PUSH

The PUSH operation depicted in Figure 3 is equivalent to the following sequence of basic operations performed by the application thread T_a. From the current scope T_a instantiates a new scope sm. T_a enters sm then sets the TOS field in sm to point to element x.

```
PUSH($S, x$)
1  $sm \leftarrow new\ ScopedMemory(size)$
2  $enter(sm, T_a)$
3  $TOS \leftarrow x$
```

Figure 3. Procedure to push an element onto the stack—scoped-memory implementation. $size \geq |x| + |TOS|$.

Assuming each line i in PUSH requires c_i time for execution, the worst case execution time for PUSH is given by $T(n) = c_1 + c_2 + c_3 = O(1)$. The correctness of this result is based on the fact that each line is executed once per invocation. Because a scope has a limited lifetime dictated by the reference count of threads executing in it, T_a is not allowed to exit sm. To ensure that T_a keeps sm live, T_a does not return from the $enter()$ method in line 2 of Figure 3. Should T_a return from the $enter()$ method, the thread reference-count of sm would drop to zero, sm would be collected, and the PUSH operation would fail.

5.2.3 POP

The POP operation returns the TOS element if one exists and $NULL$ otherwise. Assuming each line i of the POP operation (Figure 4) requires c_i time to execute, the worst-case execution time for the POP operation is given as $T(n) = O(1)$.

```
POP($S$)
1  **if** IS-EMPTY($S$)
2     **then** $x \leftarrow NULL$
3     **else** $x \leftarrow TOS$
4  **return** $x$
```

Figure 4. Procedure to pop the topmost element off the stack—scoped-memory implementation.

After popping the stack, T_a must return from the $enter()$ method of line 2 of Figure 3. We assume for all practical purposes that returning from the $enter()$ method takes $O(1)$ time. The new top-of-stack becomes the TOS element of the parent scope, *i.e.*, the parent of the scope that contained the popped element.

5.3 Cumulative analysis for stack

Here we consider an intermixed sequence of n PUSH and POP operations on a stack instance. We analyze this sequence of operations for a singly-linked-list implementation in the heap and for a scoped-memory implementation of stack. Let n denote the total number of operations and let m denote the number of PUSH operations. The number of POP operations is thus given by $n - m$ where $n - m \leq m \leq n$. The worst-case running time for the singly-linked-list implementation of the intermixed sequence of operations is computed as

$$\begin{aligned} T(n) &= T_{\text{push}}(m) + T_{\text{pop}}(n-m) \\ &= m * c_1 + (n-m) * c_2 \\ &= mc_1 + nc_2 - mc_2 \\ &= (c_1 - c_2)m + c_2 n \\ &= O(n) \end{aligned}$$

For a scoped-memory implementation the running time for PUSH or POP is $O(1)$. Thus, the running time for the intermixed sequence of operations in the context of a scoped-memory implementation is given by $T(n) = O(n)$.

5.4 Discussion

The linked-list implementation in the heap yields $T(n) = O(1)$ worst-case execution time for each stack operation. The scoped-memory implementation also yields $T(n) = O(1)$ worst-case execution time for each operation. The problem of running an intermixed sequence of n PUSH and POP operations, yields a worst-case running time of $T(n) = O(n)$ for each implementation, as expected. Given a particular program that uses a stack, the programmer can thus choose between these two implementations.

Although a singly-linked-list implementation works well in the heap, pointer manipulation can affect the proportionality constants of the running time for each operation. Garbage collection can also interfere with the running times of stack operations if the application executes in a heap that is subject to garbage collection.

A scoped-memory implementation, while good in real-time environments, comes at the cost of learning a new, restrictive programming model. Real-time programmers, however, can benefit from the timing guarantees promised by the RTSJ. Moreover, the memory used by a scoped-memory implementation of stack at any instant during execution is proportional to the number of elements on the stack at that instant.

6. Queue analysis

Our scoped-memory implementation of the queue abstract data type uses an approach similar to Okasaki's [15] functional-language implementation in that a queue is simulated as a pair of stacks. The *queue* ADT operates on the *First-In-First-Out* (FIFO) principle. The fundamental operations for a queue are:

1. ISQ-EMPTY(Q) - an operation that returns the binary value **TRUE** if queue Q is empty, **FALSE** otherwise.
2. ENQUEUE(Q, x) - an operation that adds element x to the *rear* of queue Q.
3. DEQUEUE(Q) - an operation that removes the element at the *front* of queue Q and returns it. If the queue is empty, $NULL$ is returned.

First, we present an implementation of queue in heap memory.

6.1 Typical implementation of queue

One typical implementation of the queue ADT in the heap uses a singly-linked-list data structure with two special pointers, *front* and *rear*. The ISQ-EMPTY operation checks whether *front* points to *NULL*. The ENQUEUE operation adds a new element to the *rear* end of the linked list and updates the *rear* pointer. The DEQUEUE operation updates the *front* pointer and returns the element that was at the front of the linked list. Each of these fundamental operations takes time $T(n) = O(1)$.

6.2 Scoped-memory implementation of queue

Consider execution of an application A that manages a queue instance in an RTSJ scoped-memory environment. Efficient execution of A depends on proper management of memory, which is a limited resource. Assume A uses a stack of scoped-memory instances to manage the queue. Assume also, for the purpose of worst-case analysis, that a queue element resides in its own scope when added to the queue. A service stack with its own NHRT T_1 is used to facilitate the ENQUEUE operation. See Figure 5 for a representation of a queue instance. If T_0 is the application thread, then T_0 is a NHRT. Detailed analysis of the fundamental queue operations follows.

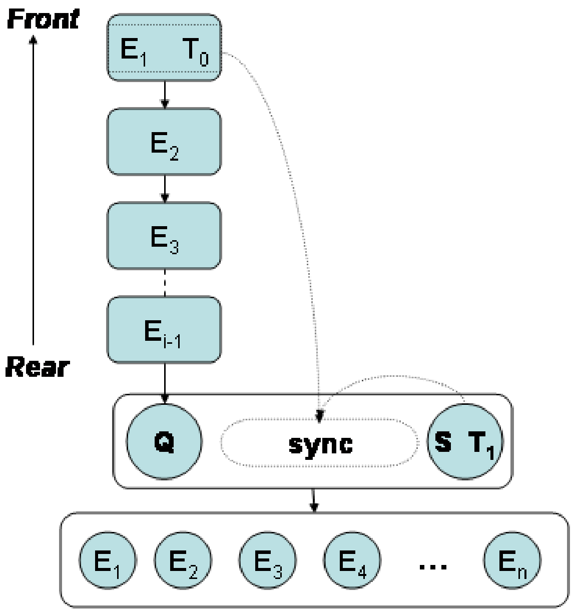

Figure 5. Queue representation in an RTSJ scoped-memory environment. Rounded rectangles represent scoped-memory instances and circles represent object instances. T_0 is the application thread and T_1 services the stack. The arrows pointing downward represent legal scope references. The *sync* field/object is a synchronization point for T_0 and T_1. E_i denotes element i. On the queue, E_i is a reference to element i.

6.2.1 ISQ-EMPTY

The current scope contains a *front* field that points to the front of the queue. An empty queue is a queue with no elements. Emptiness, in Figure 6, is illustrated by the *front* field of the current scope pointing to the queue object itself. Assuming that the running time of the lone line of ISQ-EMPTY is c_1, the worst-case running time of ISQ-EMPTY is given as $T(n) = c_1 = O(1)$.

```
ISQ-EMPTY($Q$)
1 return $front = Q$
```

Figure 6. Procedure to test if the queue is empty—scoped-memory implementation.

6.2.2 DEQUEUE

The DEQUEUE operation removes the element at the front of the queue and returns it if one exists. Otherwise, it returns $NULL$. A close examination of the DEQUEUE operation (Figure 7) reveals that it is similar to the POP operation of Figure 4—on a DEQUEUE the thread exits from the innermost $enter()$. Hence, the worst-case running time for DEQUEUE is $T(n) = O(1)$ time.

6.2.3 ENQUEUE

The ENQUEUE operation is a relatively complex operation because of the referencing constraints imposed by RTSJ: Objects in an ancestor scope cannot reference objects in a descendant scope because the descendant scope is reclaimed before the ancestor scope. As a consequence of this and other constraints, the elements already in a queue must first be stored somewhere before a

```
DEQUEUE($Q$)
1 if ISQ-EMPTY(Q)
2     then $x \leftarrow NULL$
3     else $x \leftarrow front$
4 return $x$
```

Figure 7. Procedure to remove an element from the front of the queue—scoped-memory implementation.

new element can be enqueued. After the element is enqueued, all the stored elements are put back on the queue in their correct order. A stack is an ideal structure to store the queue elements because it preserves their order for the queue. As illustrated in Figure 5, two threads are needed to facilitate the ENQUEUE operation: one for the queue and one to service the stack. The thread that services the queue is the application thread and is referred to as T_0; T_1 is the service thread for the stack. These two threads are synchronized by a parameter *sync*, which they use to share data between them—see Figure 8.

```
ENQUEUE($Q, x$)                                   time cost   frequency
1 while !ISQ-EMPTY($Q$)                           $c_1$       $n+1$
2     do $sync \leftarrow$ DEQUEUE($Q$)           $c_2$       $n$
3         PUSH(S, $sync$)                         $c_3$       $n$
4 $S_c \leftarrow new\ ScopedMemory(m)$           $c_4$       1
5 $enter(S_c, T_0)$                               $c_5$       1
6 $front \leftarrow x$                            $c_6$       1
7 while !IS-EMPTY($S$)                            $c_7$       $n+1$
8     do $sync \leftarrow$ POP($S$)               $c_8$       $n$
9         PUSH-Q($Q, sync$)                       $c_9$       $n$
```

Figure 8. Procedure to add an element to the rear of the queue—scoped-memory implementation. Each c_i is a constant and $n = |Q| + |S|$. Initially stack S is empty.

```
PUSH-Q($S, x$)
1 $sm \leftarrow new\ ScopedMemory(size)$
2 $enter(sm, T_a)$
3 $front \leftarrow x$
```

Figure 9. Private helper method that puts an element at the front of the queue in the same manner that an element is pushed onto a stack—scoped-memory implementation. $size \geq |x| + |front|$.

PUSH-Q is a private method that puts a stored element back on the queue in the way that the PUSH operation works for a stack. The worst-case running time for this method is $T(n) = O(1)$ time. This is the same running time for the PUSH operation of Figure 3.

Given the procedure in Figure 8 the worst-case running time for ENQUEUE is linear in the number of elements already in the queue:

$$\begin{aligned} T(n) &= (n+1)c_1 + nc_2 + nc_3 + c_4 + c_5 + c_6 + \\ &\quad (n+1)c_7 + nc_8 + nc_9 \\ &= (c_1 + c_2 + c_3 + c_7 + c_8 + c_9)n + c_1 + c_4 + \\ &\quad c_5 + c_6 + c_7 \\ &= c_b * n + c_a \\ &= O(n) \end{aligned}$$

6.3 Cumulative analysis for queue

We compute the theoretical running time for an intermixed sequence of ENQUEUE and DEQUEUE operations on a queue instance by analyzing the worst-case running time of the sequence. Since we suggested two implementation contexts for the queue ADT, we compute the running time for each implementation. Suppose n denotes the number of operations in the sequence and m denotes the number of ENQUEUE operations, then the number of DEQUEUE operations is given as $n - m$ where $n - m \leq m \leq n$. The worst-case running time for the heap implementation is thus given as:

$$\begin{aligned} T(n) &= T_{\text{enq}}(m) + T_{\text{deq}}(n-m) \\ &= m * c_1 + (n-m) * c_2 \\ &= nc_2 + m(c_1 - c_2) \\ &= O(n) \end{aligned}$$

This is identical to the linked-list analysis of an intermixed sequence of PUSH and POP operations on a stack because the insertion operation an the deletion operation each executes in constant time.

The scoped-memory implementation is more complex for the ENQUEUE operation. The running time for the sequence of operations in that context is also more complex and more costly. We compute the worst-case running time as follows:

$$\begin{aligned} T(n) &= T_{\text{enq}}(m, \vec{s}) + T_{\text{deq}}(n-m) \\ &= T_{\text{enq}}(m, \vec{s}) + (n-m) * c_2 \end{aligned}$$

$\vec{s} = \langle s_1, s_2, \ldots, s_m \rangle$ is included as input to the computation of the running time for the ENQUEUE operations because the running time of each invocation of the ENQUEUE operation depends on the number of elements in the queue. s_i denotes the number of elements on the queue before the ith operation. Given fixed n and m, the worst-case running time for the sequence of operations occurs when no DEQUEUE operations precede an ENQUEUE operation. In this case the values in $\vec{s}$ are monotonically increasing from 0 to $m - 1$, so for the computation of $T(n)$ given below, $s_i = i - 1$. c_a and c_b are derived from the ENQUEUE analysis above and c_d is the time for the constant DEQUEUE operation.

$$\begin{aligned} T(n) &= T_{\text{enq}}(m, \vec{s}) + T_{\text{deq}}(n-m) \\ &= \sum_{i=1}^{m}(c_a + s_i c_b) + T_{\text{deq}}(n-m) \\ &= \sum_{i=1}^{m}(c_a + (i-1)c_b) + T_{\text{deq}}(n-m) \\ &= mc_a + \left(\sum_{i=1}^{m} c_b i\right) - mc_b + T_{deq}(n-m) \\ &= mc_a - mc_b + c_b\left(\sum_{i=1}^{m} i\right) + T_{deq}(n-m) \\ &= m(c_a - c_b) + c_b\frac{m(m+1)}{2} + T_{\text{deq}}(n-m) \\ &= m(c_a - c_b) + \frac{m^2 c_b}{2} + \frac{mc_b}{2} + T_{\text{deq}}(n-m) \\ &= \frac{c_b}{2}m^2 + \frac{2c_a - c_b}{2}m + (n-m)c_d \\ &= \frac{c_b}{2}m^2 + \frac{2c_a - c_b - 2c_d}{2}m + c_d n \end{aligned}$$

Since $m \leq n$ it follows that $T(n) = O(n^2)$. Thus, for a scoped-memory queue implementation the worst-case running time for an intermixed sequence of n ENQUEUE and DEQUEUE operations is $T(n) = O(n^2)$.

6.4 Discussion

Two possible implementations for the queue ADT were suggested: a singly-linked-list implementation and an RTSJ scoped-memory implementation. The singly-linked-list implementation yields $T(n) = O(1)$ worst-case execution time for each queue operation. The scoped-memory implementation yields $T(n) = O(1)$ worst-case execution time for the ISQ-EMPTY and DEQUEUE operations, but $O(n)$ time for the ENQUEUE operation. The reason the worst-case execution time for ENQUEUE is linear instead of constant is based on the referencing constraints imposed by RTSJ's scoping rules. Scopes are usually instantiated in a stack-like fashion. Thus, to enqueue an element the scope stack must be popped and the element in each scope must be stored on a stack or some other data structure. A new scope to enqueue the element must then be instantiated from the base of the scope stack and be placed on the queue. The elements stored away for the ENQUEUE operation must then be restored on the queue in a LIFO manner.

In addition to performing analysis for each operation, we performed analysis for the problem of running a sequence of n ENQUEUE and DEQUEUE operations on a queue instance. The singly-linked-list implementation gives a worst-case running time of $O(n)$ and the scoped-memory implementation gives a worst-case running time of $O(n^2)$. Thus, the scoped-memory implementation gives a running time that is an order of magnitude larger than the running time given by the singly-linked-list implementation. This is rather expensive for an environment that governs its own memory and gives NHRTs higher priorities than any garbage collector.

6.5 Improved scoped-memory implementation of queue

We presented thus far an implementation of a queue in an RTSJ scoped-memory environment that turned out to have a worst-case running time of $O(n^2)$ for n consecutive ENQUEUE operations. Here, we present a modified queue implementation that has better worst-case time performance on the problem of running an intermixed sequence of n ENQUEUE and DEQUEUE operations on a queue instance (see Figure 10).

```
ENQUEUE($Q, x$)
1  $n_{\text{enq}} \leftarrow n_{\text{enq}} + 1$
2  **if** $n_{\text{enq}}$ is some power of 2
3     **then while** !ISQ-EMPTY(Q)
4              **do** $sync \leftarrow$ DEQUEUE(Q)
5                   PUSH(S, $sync$)
6           **for** $i = 1$ **to** $n_{\text{enq}}$
7              **do** $S_c \leftarrow new\ ScopedMemory(m)$
8                   $enter(S_c, T_0)$
9                   $front \leftarrow getOuterScope()$
10          $thread[next] \leftarrow front$
11          $front \leftarrow x$
12          **while** !IS-EMPTY(S)
13             **do** $sync \leftarrow$ POP(S)
14                  PUSH-Q(Q, $sync$)
15    **else** $temp \leftarrow thread[next][front]$
16          $thread[next][front] \leftarrow x$
17          $thread[next] \leftarrow temp$
```

Figure 10. Procedure to add an element to the rear of the queue—scoped-memory implementation.

As with the previous implementation, we use a service stack with its own NHRT T_1 to manage the queue. We also limit each scope to holding at most one queue element, and the ISQ-EMPTY and DEQUEUE operations remain the same as those presented above. Whereas before we copied the entire queue over to the service stack for each ENQUEUE operation, now we do so only for the ith ENQUEUE operation when i is a power of 2. After the queue elements are copied to the service stack, and before they are copied back to the queue in their previous order, we create not one but i new scopes at the rear of the queue. The new element is enqueued in the deepest scope—the one nearest to the front of the queue. The other scopes remain empty until they are filled on subsequent ENQUEUE operations.

Suppose, for example, we start with an empty queue and perform 15 consecutive ENQUEUE operations. The queue now has 15 elements, each in its own scope. Then another ENQUEUE operation is to be performed. First, the elements already in the queue are copied over to the service stack. Then, not one but 16 nested scopes, each capable of holding one queue element, are created. The element being enqueued is placed in the most deeply nested scope, *i.e.*, the one closest to the front of the queue. Then the 15 elements on the service stack are copied back over to the queue in their correct order. Now, the next 15 ENQUEUE operations will fill the empty scopes without having to use the service stack. A field n_{enq} in the synchronized shared memory (in the scope containing both the queue and service stack) will keep track of the number of times ENQUEUE has been called.

line	*time cost*	*freq. when* $n_{\mathrm{enq}} = 2^x$	*freq. otherwise*
1	c_1	1	1
2	c_2	1	1
3	c_3	$n+1$	0
4	c_4	n	0
5	c_5	n	0
6	c_6	$n_{\mathrm{enq}}+1$	0
7	c_7	n_{enq}	0
8	c_8	n_{enq}	0
9	c_9	n_{enq}	0
10	c_{10}	1	0
11	c_{11}	1	0
12	c_{12}	$n+1$	0
13	c_{13}	n	0
14	c_{14}	n	0
15	c_{15}	0	1
16	c_{16}	0	1
17	c_{17}	0	1

Figure 11. Statistics for procedure in Figure 10. Each c_i is a constant and $n = |Q| + |S|$. Initially stack S is empty and so $n = |Q|$.

6.5.1 Cumulative analysis for queue revisited

The worst-case running time for a single call of the ENQUEUE operation is $O(n)$, where n is the number of elements already on the queue, so the worst-case running time for n consecutive ENQUEUE calls, starting with an empty queue, might reasonably be expected to be $O(n^2)$. Fortunately, that turns out not to be the case. Consider beginning with an empty queue and performing a series of n ENQUEUE operations with no DEQUEUEs. During the ith ENQUEUE call, $n = i - 1$ (since n is the number of elements already on the queue) and $n_{\mathrm{enq}} = i$ (after the shared-memory field n_{enq} is incremented as the first step of the ENQUEUE algorithm). It can be seen from Figure 11 that the ith ENQUEUE call takes $c_a + c_b i$ time if $i = 2^x$ for some integer x, where

$$c_a = c_1 + c_2 - c_4 - c_5 + c_6 + c_{10} + c_{11} - c_{13} - c_{14}$$
$$c_b = c_3 + c_4 + c_5 + c_6 + c_7 + c_8 + c_9 + c_{12} + c_{13} + c_{14}$$

and c_c time otherwise, where

$$c_c = c_1 + c_2 + c_{15} + c_{16} + c_{17}$$

Assuming $n = 2^x$ for some integer x (which is a worst case, since the last ENQUEUE will be a linear-time and not a constant-time operation), the total running time for all n ENQUEUEs is given as

$$\begin{aligned}
T(n) &= \sum_{j=0}^{x}(c_a + c_b 2^x) + (2^x - x - 1)c_c \\
&= (x+1)c_a + \left(\sum_{j=0}^{x} 2^x\right) c_b + (2^x - x - 1)c_c \\
&= (x+1)c_a + (2^{x+1} - 1)c_b + (2^x - x - 1)c_c \\
&= (2c_b + c_c)2^x + (c_a - c_c)x + c_a - c_b - c_c \\
&= (2c_b + c_c)n + (c_a - c_c)\log_2 n + c_a - c_b \\
&\quad -c_c \\
&= O(n)
\end{aligned}$$

Thus, the improved ENQUEUE operation has a worst-case running time of $O(n)$ on the sequence of operations. However, because it overallocates when resizing, ENQUEUE relies at some point on having twice the number of cells allocated as are actually in use. Interestingly, a space-time trade-off of this nature is also endemic to the real-time collectors [4]. Still, the memory required is bounded and proportional to the maximum number of elements in the queue at any given time.

7. List analysis

In this section and Section 8 we use the transformation described in Section 3 to migrate functional programming implementations of data structures and their time-complexity analyses to RTSJ. We begin with the *list* ADT

The *list* ADT is an ADT that formalizes the notion of an ordered collection of entities or items. The fundamental operations of list are:

1. ISLIST-EMPTY(L) - an operation that returns the binary value **TRUE** if list L is empty, **FALSE** otherwise.
2. SIZE(L) - an operation that returns the number of elements in list L.
3. CREATE(L) - an operation that creates an empty list L.
4. INSERT(L, x) - an operation that inserts an item at the head (front) of list L.
5. HEAD(L) - an operation that returns the item at the head of list L.
6. DELETE-ITEM(L) - an operation that removes the item located at head of list L and returns a list containing one fewer item. If L is empty, an error condition is reported.
7. LOOKUP(L, i) - an operation that returns the item located at index i of list L. If L contains fewer than i items, an error condition is reported.
8. UPDATE(L, i, x) - an operation that replaces the item at index i in list L with item x. If L contains fewer than i items, an error condition is reported.

7.1 Typical implementation of list

In the heap, the singly-linked-list or the doubly-linked-list data structure can be used to implement the list ADT. In either case the ISLIST-EMPTY, SIZE, CREATE, HEAD, and DELETE-ITEM operations each executes in $O(1)$ time. The LOOKUP and UPDATE operations each requires $O(n)$ time since the list has to be searched to find the requested index.

7.2 Scoped-memory implementation of list

We do not present a particular scoped-memory implementation as we did in Sections 5 and 6. Instead, we migrate to this section cost analysis for list from the functional programming language community [15, 16]. In his Ph.D. dissertation, Okasaki [16] implemented the list ADT in Standard ML, a functional programming language. The declaration of each operation is similar to those given above. He also analyzed the running time of each operation. Using his analysis and the results in Section 3 we give the following time complexity for each list operation when implemented with RTSJ scoped-memory areas. The ISLIST-EMPTY, SIZE, CREATE, HEAD, and DELETE-ITEM operations each executes in $O(1)$ time while the LOOKUP and UPDATE operations requires $O(\log n)$ time each.

7.3 Cumulative analysis for list

Suppose there exists a list L with n items. We consider computing the running time of executing an intermixed sequence of m LOOKUP and UPDATE operations (the most expensive operations) on list L. In a heap implementation, the running time for the sequence of operations is $O(mn)$. In a scoped-memory implementation, the running time is $O(m \log n)$. While it appears that a scoped-memory implementation is more efficient than a heap implementation, the scoped-memory implementation can leak an unbounded amount of memory if some method of explicitly reclaiming scopes is not utilized. In Section 3 we noted three approaches that can be used to reclaim scopes.

8. Heap analysis

The *heap* or *priority queue* is the second ADT with which we use the transformation outlined in Section 3. This ADT, at a minimum, allows the following operations: INSERT, which inserts an element in the heap; and DELETE-MIN, which finds, returns, and deletes the minimum element from the heap. The heap is generally implemented as a tree-based data structure that satisfies the *structure property* and the *heap order property*. The structure property says that a heap is implemented as a binary tree that is completely filled, with the possible exception being the leaves level, which is filled from left to right [19]. The heap order property requires that data in the heap be an ordered set. Since the minimum element needs to be found quickly, the heap order property requires that the smallest element be at the root of the heap. If it is required that every subtree be a heap, then any node in the heap should be smaller than its descendants. This implementation of the heap ADT is called the binary heap.

Another implementation of the heap ADT is the binomial heap. A binomial heap is similar to a binary heap except that the operation that merges two heaps runs rather quickly. Since the operations for both implementations of the heap are the same, we consider the binomial heap in our analysis. Thus, we define the fundamental operations for heap as follows:

1. CREATE(H) - an operation that creates an empty heap H.
2. ISHEAP-EMPTY(H) - an operation that returns the binary value **TRUE** if heap H is empty, **FALSE** otherwise.
3. INSERT(H, x) - an operation that inserts item x in heap H.
4. FIND-MIN(H) - an operation that finds and returns the minimum item in heap H.
5. DELETE-MIN(H) - an operation that removes the minimum item in heap H and returns a new heap with one fewer item. If H is empty, an error condition is reported.
6. MERGE(H_1, H_2) - an operation that merges heap H_1 with heap H_2 to form a new heap containing as many items as the sum of the number of items in H_1 and H_2.

8.1 Typical implementation of heap

In the heap (not the heap data structure) where dynamic memory management occurs, several options are available for implementing the heap ADT. An array can be used to store the heap; a binary tree can be used to implement the heap; a binomial tree can also be used to implement the heap. We consider the binomial tree implementation, more specifically the binomial heap data structure, in our analysis for reasons given above. Consequently, the cost associated with each operation is given as follows. The CREATE and ISHEAP-EMPTY operations each takes $O(1)$ time to execute. The other operations each requires $O(\log n)$ time. This is not surprising since the height of the tree used to store the heap is $O(\log n)$, where n is the number of nodes (items) in the tree.

8.2 Scoped-memory implementation of heap

As we did for the list ADT, we migrate to this section cost analysis of the running times of heap operations from the functional programming language community. In particular, we migrate analyses from Okasaki [16]. Okasaki used a binomial heap implementation for the heap ADT, which he developed in Standard ML. He analyzed the running time of each operation and obtained a complexity of $O(\log n)$ for each operation, except CREATE and ISHEAP-EMPTY, which each executes in $O(1)$ time. Adopting Okasaki's results, we conclude that for an RTSJ scoped-memory implementation of the heap ADT, the operations CREATE and ISHEAP-EMPTY execute in $O(1)$ time. Every other operation, namely INSERT, FIND-MIN, DELETE-MIN, and MERGE, each requites $O(\log n)$ time.

8.3 Cumulative analysis for heap

Here we consider executing an intermixed sequence of m INSERT, FIND-MIN, DELETE-MIN, and MERGE operations. Interestingly, these operations have the same running time for both a heap implementation and a scoped-memory implementation. Since each operation has a running time of $O(\log n)$ and there are m operations in the sequence, the running time for the sequence of operations is $O(m \log n)$.

Although the results are the same for both implementations, the heap implementation is simple and exists in most data structure texts. Further, scoped-memory implementation of heap is not commonplace. Theorem 3.2 allows us to migrate a functional programming language implementation of heap to RTSJ; however, such implementation can consume an unnecessary amount of memory if extreme care is not taken in reclaiming scopes. Section 3 offers a few suggestions.

9. Conclusions

There are many implementations of RTSJ including TimeSys, JRate, and an implementation from Sun Microsystems. In the real-time and academic communities many are experimenting with RTSJ's scoped-memory areas and NHRTs. Until now, there has not been objective time-complexity analysis of scoped-memory areas and NHRTs.

In this paper we presented asymptotic time-complexity analysis for RTSJ scoped-memory areas and NHRTs. One approach was to suggest implementations in RTSJ for abstract data types like stack and queue and to determine asymptotic bounds for their execution. The results allow us to compare scoped memory with other memory models and to reason more thoroughly about the differences among those models.

One of our assumptions for the above approach was to consider one element per scope. We do not recommend this restriction in practice; however, the analysis holds if multiple elements are allowed per scope. Consider, for example, $4k$ elements per scope. If $4k$ elements are enqueued on a queue and a $(4k+1)$th element is to be enqueued (with no intervening dequeue operation), a new scope would have to be instantiated to accommodate that element. That enqueue operation suffers the cost discussed in Section 6.5.1.

An additional approach to providing asymptotic time-complexity analysis for RTSJ scoped-memory areas and NHRTs is to migrate extant functional programming language implementations of important data structures to RTSJ. This migrations is a direct consequence of our proof that programs written in a pure functional programming language can be executed in a provably safe manner in RTSJ when scoped-memory areas and NHRTs are used. This migration also allows us to migrate time-complexity analysis for data structures implemented in a functional programming language to RTSJ. We used that approach to perform time-complexity analysis for RTSJ scoped-memory areas and NHRTs using the list and heap abstract data types. We discovered that the time-complexity results for RTSJ are comparable to the results for heap implementations. Moreover, using RTSJ forces the developer to think more carefully about memory management since RTSJ scopes can leak an unbounded amount of memory. Ours is the first work to point this out.

10. Acknowledgments

We thank Morgan Deters for his insight, feedback, and support. We also thank other colleagues in the Department of Computer Science and Engineering at Washington University who gave us editorial feedback in preparation of the final draft of this paper.

References

[1] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, Boston, MA, 2000.

[2] John Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978.

[3] David F. Bacon, Perry Cheng, and V. T. Rajan. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 81–92, San Diego, California, June 2003.

[4] David F. Bacon, Perry Cheng, and V. T. Rajan. The Metronome: A simpler approach to garbage collection in real-time systems. In R. Meersman and Z. Tari, editors, *Proceedings of the OTM Workshops: Workshop on Java Technologies for Real-Time and Embedded Systems*, volume 2889 of *Lecture Notes in Computer Science*, pages 466–478, Catania, Sicily, November 2003. Springer-Verlag.

[5] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[6] Chandrasekhar Boyapati, Alexandru Salcianu, Jr. William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time java. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 324–337, New York, NY, USA, 2003. ACM Press.

[7] Dante J. Cannarozzi, Michael P. Plezbert, and Ron K. Cytron. Contaminated garbage collection. In *Proceedings of the ACM SIGPLAN '00 conference on Programming Language Design and Implementation (PLDI 2000)*, pages 264–273, 2000.

[8] Sigmund Cherem and Radu Rugina. Region analysis and transformation for java programs. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 85–96, New York, NY, USA, 2004. ACM Press.

[9] Delvin Defoe, Rob LeGrand, and Ron K. Cytron. Cost analysis for real-time java scoped-memory areas. *Journal of Systemics, Cybernetics and Informatics*, 5, 2007. forthcoming.

[10] Morgan Deters and Ron K. Cytron. Automated discovery of scoped memory regions for real-time java. In *ISMM '02: Proceedings of the 3rd international symposium on Memory management*, pages 25–35, New York, NY, USA, 2002. ACM Press.

[11] Max Goff. Celebrating 10 years of Java and our technological productivity: A look back on the last 10 years of the network age. `http://www.javaworld.com`, May 2005.

[12] Tobias Mann, Morgan Deters, Rob LeGrand, and Ron K. Cytron. Static determination of allocation rates to support real-time garbage collection. In *LCTES'05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 193–202, New York, NY, USA, 2005. ACM Press.

[13] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960.

[14] Kelvin Nilsen. A type system to assure scope safety within safety-critical java modules. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 97–106, New York, NY, USA, 2006. ACM Press.

[15] Chris Okasaki. Functional Data Structures. In *Advanced Functional Programming, LNCS 1129*, pages 131–158. Springer, August 1996.

[16] Chris Okasaki. *Purely Functional Data Structures*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, September 1996. This research was sponsored by the Advanced Research Projects Agency (ARPA) under Contract No. F19628-95-C-0050.

[17] F. Pizlo, J. M. Fox, D. Holmes, and J. Vitek. Real-time java scoped memory: Design patterns and semantics. In *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 101–110, Vienna, Austria, May 2004. IEEE Computer Society.

[18] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4):724–767, 1998.

[19] Mark A. Weiss. *Data Structures and Algorithm Analysis in C*. Addison-Wesley Longman, Inc., Menlo Park, CA, second edition, 1997.