A Type System for Safe Region-Based Memory Management in Real-Time Java

Alexandru Sălcianu, Chandrasekhar Boyapati, William Beebee, Jr., Martin Rinard
MIT Laboratory for Computer Science
200 Technology Square, Cambridge, MA 02139
{salcianu,chandra,wbeebee,rinard}@lcs.mit.edu

ABSTRACT

The Real-Time Specification for Java (RTSJ) allows a program to create real-time threads with hard real-time constraints. Real-time threads use immortal memory and region-based memory management to avoid unbounded pauses caused by interference from the garbage collector. The RTSJ uses runtime checks to ensure that deleting a region does not create dangling references and that real-time threads do not access references to objects allocated in the garbage-collected heap. This paper presents a static type system that guarantees that these runtime checks will never fail for well-typed programs. Our type system therefore 1) provides an important safety guarantee for real-time programs and 2) makes it possible to eliminate the runtime checks and their associated overhead.

Our system also makes several contributions over previous work on region types. For object-oriented programs, it combines region types and ownership types in a unified type system framework. For multithreaded programs, it allows long-lived threads to share objects without using the heap and without having memory leaks. For real-time programs, it ensures that real-time threads do not interfere with the garbage collector.

We have implemented several programs in our system. Our experience indicates that our type system is sufficiently expressive and requires little programming overhead. We also ran these programs on our RTSJ platform. Our experiments show that eliminating the RTSJ runtime checks using a static type system can significantly decrease the execution time of a real-time program.

1. INTRODUCTION

The Real-Time Specification for Java (RTSJ) [6] provides a framework for building real-time systems. The RTSJ allows a program to create real-time threads with hard real-time constraints. These real-time threads cannot use the garbage-collected heap because they cannot afford to be interrupted for unbounded amounts of time by the garbage collector. Instead, the RTSJ allows these threads to use objects allocated in immortal memory (which is never garbage collected) or in regions [36]. Region-based memory management systems structure memory by grouping objects in regions under program control. Memory is reclaimed by deleting regions, freeing all objects stored therein. The RTSJ uses runtime checks to ensure that deleting a region does not create dangling references and that real-time threads do not access heap references.

This paper presents a static type system for writing real-time programs in Java. Our system guarantees that the above-mentioned RTSJ runtime checks will never fail for well-typed programs. Our system can serve as a front-end for the RTSJ platform. It offers two advantages to real-time programmers. First, it provides a safety guarantee that the program will never fail because of a failed access check. Second, if an RTSJ implementation allows disabling the RTSJ runtime checks, then programs written in our system can run more efficiently without risking memory errors.

Our approach is applicable even outside the RTSJ context. In fact, it could be adapted to provide safe region-based memory management for virtually any type-safe real-time language. Given the advantages of using regions and immortal memory in real-time systems, we believe our technology will be relevant in whatever type-safe real-time language eventually emerges as the industry standard.

Our system also makes several important technical contributions over previous work on type systems for region-based memory management. For object-oriented programs, it combines region types [36, 17, 26, 30] and ownership types [14, 13, 7, 8] in a unified type system framework. Region types statically ensure that programs never follow dangling references. Ownership types provide a statically enforceable way of specifying object encapsulation. Ownership types enable modular reasoning about program correctness. Consider, for example, a Stack object s that is implemented using a Vector subobject v. To reason locally about the correctness of the Stack implementation, a programmer must know that v is not directly accessed by objects outside s. With ownership types, a programmer can declare that s *owns* v. The type system then statically ensures that v is encapsulated within s.

In an object-oriented language that only has region types (eg., [30]), the types of s and v would declare that they are allocated in some region r. In an object-oriented language that only has ownership types, the type of v would declare that it is owned by s. Our type system provides a simple unified mechanism to declare *both* properties. The type of s can declare that it is allocated in r and the type of v can declare that it is owned by s. Our system then statically ensures that both objects are allocated in r, that there are no pointers to v and s after r is deleted, and that v is encapsulated within s. We thus combine the benefits of region types and ownership types.

This research was supported by DARPA/AFRL Contract F33615-00-C-1692.

Our system extends region types to multithreaded programs by allowing explicit memory management for objects shared between threads. Our system allows threads to communicate through objects in *shared regions* in addition to the heap. A shared region is deleted when all threads exit the region. However, programs in a system with just shared regions (eg., [25]) will have memory leaks if two long-lived threads communicate by creating objects in a shared region. This is because the objects will not be deleted until both threads exit the shared region. To solve this problem, we introduce the notion of *subregions* within a shared region. A subregion can be deleted more frequently, for example, after each loop iteration in the long-lived threads.

Our system also introduces *typed portal fields* in subregions to serve as a starting point for inter-thread communication. Portals also allow typed communication, so threads do not have to downcast from Object to more specific types. Typed communication prevents more errors statically. Our system introduces user-defined *region kinds* to support subregions and portal fields.

Our system extends region types to real-time programs by using effects clauses [29] to statically check that real-time threads do not interfere with the garbage collector. Our system augments region kind declarations with *region policy* declarations. We support two policies for creating regions as in RTSJ. A region can be an LT (Linear Time) region, or a VT (Variable Time) region. Memory for an LT region is preallocated at region creation time, so allocating an object in an LT region only takes time proportional to the size of the object (because all the bytes have to be zeroed). Memory for a VT region is allocated on demand, so allocating an object in a VT region takes variable time. Our system checks that real-time threads do not use heap references, create new regions, or allocate objects in VT regions.

Most previous region type systems allow programs to create, but not follow, dangling references. Such references can cause a safety problem when used with copying collectors. We therefore prevent a program from creating dangling references in the first place.

Contributions

To summarize, this paper makes the following contributions:

- **Region types for object-oriented programs:** Our system combines region types and ownership types in a unified type system framework that statically enforces object encapsulation as well as enables safe region-based memory management.
- **Region types for multithreaded programs:** Our system introduces *subregions* within a shared region, so long-lived threads can share objects without using the heap and without having memory leaks. Our system also introduces *typed portal fields* to serve as a starting point for typed inter-thread communication. Our system introduces user-defined *region kinds* to support subregions and portals.
- **Region types for real-time programs:** Our system allows programs to create LT (Linear Time) and VT (Variable Time) regions as in RTSJ. It checks that real-time threads do not use heap references, create new regions, or allocate objects in VT regions, so that they do not wait for unbounded amounts of time.
- **Type inference:** Our system uses a combination of intra-procedural type inference and well-chosen defaults to significantly reduce programming overhead. Our approach permits separate compilation.
- **Experience:** We implemented several programs in our system. Our experience indicates that our type system is sufficiently expressive and requires little programming overhead. We also ran these programs on our RTSJ platform [4, 5, 22]. Our experiments show that eliminating the RTSJ dynamic checks using a static type system can significantly speed-up a real-time program.

Outline

The paper is organized as follows. Section 2 describes our type system. Section 3 describes our experimental results. Section 4 presents related work. Section 5 concludes.

2. TYPE SYSTEM

This section presents our type system for safe region-based memory management. Sections 2.1, 2.2, and 2.3 provide an informal sketch of the type system. Section 2.4 presents some of the important rules for typechecking. The full set of rules are in the appendix. Section 2.5 describes type inference techniques that reduce programming overhead. Section 2.6 describes how programs in written our system are translated to run on our RTSJ platform.

2.1 Regions for Object Oriented Programs

This section presents our type system for region-based memory management in object-oriented programs. It combines region types [36, 17, 26, 30] and ownership types [14, 13, 7, 8]. Region types statically ensure that programs using region-based memory management are memory-safe, that is, they never follow dangling references. Ownership types provide a statically enforceable way of specifying object encapsulation. The idea is that an object can *own* subobjects that it depends on, thus preventing them from being accessible outside. (An object a *depends* on subobject b [28] if a calls methods of b and furthermore these calls expose mutable behavior of b in a way that affects the invariants of a.) Object encapsulation enables local reasoning about program correctness. Ownership types have also been used for statically preventing data races [9] and deadlocks [7] in Java programs, for supporting safe lazy upgrades in object-oriented databases [8], and for program understanding [3]. Our type system is based on the type system in [8].

Ownership Relation Objects are allocated in regions. Every object in our system has an owner. An object can be owned by another object, or by a region. We write $o_1 \succeq_o o_2$ if o_1 directly or transitively owns o_2 or if o_1 is the same as o_2. The relation $\succeq_o$ is thus the reflexive transitive closure of the *owns* relation. Our type system statically guarantees the properties in Figure 1. O1 states that our ownership relation has no cycles. O2 states that if an object is owned by a region, then that object and all its subobjects are allocated in that region. O3 states the encapsulation property of our system, that if y is inside the encapsulation boundary of z and x is outside, then x cannot access y.[1] An object x

[1]Our system handles inner class objects specially to support constructs like iterators. Details can be found in [8].

O1. The ownership relation forms a forest of trees.

O2. If region $r \succeq_o$ object x, then x is allocated in r.

O3. If object $z \succeq_o y$ but $z \not\succeq_o x$, then x cannot access y.

Figure 1: Ownership Properties

R1. For any region r, heap $\succeq r$ and immortal $\succeq r$.

R2. $x \succeq_o y \Longrightarrow x \succeq y$.

R3. If region $r_1 \succeq_o$ object o_1, region $r_2 \succeq_o$ object o_2, and $r_1 \succeq r_2$, then o_2 cannot contain a pointer to o_1.

Figure 2: Outlives Properties

accesses an object y if x has a pointer to y, or methods of x obtain a pointer to y. Figure 6 shows an example ownership relation. We draw a solid line from x to y if x owns y. Region r2 owns s1, s1 owns s1.head and s1.head.next, etc.

Outlives Relation Our system allows a program to create regions. Our system also provides two special regions: the garbage collected region heap, and the "immortal" region immortal. The lifetime of a region is the time interval from when the region is created until it is deleted. If the lifetime of region r_1 includes the lifetime of region r_2, we say that r_1 *outlives* r_2, and write $r_1 \succeq r_2$. We extend the *outlives* relation to include objects. The extension is natural: if object o_1 owns object o_2, then o_1 outlives o_2, because o_2 is accessible only through o_1. Also, if region r owns object *obj*, then *obj* is allocated in r, and therefore r outlives *obj*. In all cases, $x \succeq_o y$ implies $x \succeq y$. Our outlives relation has the properties shown in Figure 2. R1 states that heap and immortal outlive all regions. R2 states that the outlives relation includes the ownership relation. R3 implies that there are no dangling references in our system. Figure 6 shows an example outlives relation. We draw a dashed line from region x to region y if x outlives y. In the example, region r1 outlives region r2, and heap and immortal outlive all regions. Using the above definitions, we can prove the following easy lemmas:

LEMMA 1. *If object $o_1 \succeq$ object o_2, then $o_1 \succeq_o o_2$.*

LEMMA 2. *If region $r \succeq$ object o, then there exists a unique region r' such that $r \succeq r'$ and $r' \succeq_o o$.*

Grammar To simplify the presentation of key ideas behind our approach, we describe our type system formally in the context of a core subset of Java known as Classic Java [21]. Our approach, however, extends to the whole of Java and other similar languages. Figure 3 presents the grammar for our core language. A program consists of a series of class declarations followed by an initial expression. A predefined class Object is the root of the class hierarchy.

Owner Polymorphism Every class definition is parameterized with one or more owners. (This is similar to parametric polymorphism [32, 10, 1] except that our parameters are values, not types.) An owner can be an object or a region. Parameterization allows programmers to implement a generic class whose objects can have different owners. The

P	::=	$def^*\ e$
def	::=	`class` $cn\langle formal+\rangle$ `extends` c `where` $constr^*$ { $field^*$ $meth^*$ }
$formal$	::=	$k\ fn$
c	::=	$cn\langle owner+\rangle$ | Object$\langle owner\rangle$
$owner$	::=	fn | r | this | initialRegion
$field$	::=	$t\ fd$
$meth$	::=	$t\ mn\langle formal^*\rangle$(($t\ p$)*) `where` $constr^*$ { e }
$constr$	::=	$owner$ `owns` $owner$ | $owner$ `outlives` $owner$ |
t	::=	c | `int` | `RHandle`$\langle r\rangle$
k	::=	Owner | ObjOwner | $rkind$
$rkind$	::=	Region | GCRegion | NoGCRegion | LocalRegion
e	::=	v | h_heap | h_immortal | `let` v `=` e `in` { e } | $v.fd$ | $v.fd = v$ | $v.mn\langle owner^*\rangle(v^*)$ | `new` c | (`RHandle`$\langle r\rangle$ h) { e }
h	::=	v
cn	$\in$	class names
fd	$\in$	field names
mn	$\in$	method names
fn	$\in$	formal identifiers
v, p	$\in$	variable names
r	$\in$	region identifiers (including heap, immortal)

Figure 3: Grammar for Object Oriented Programs

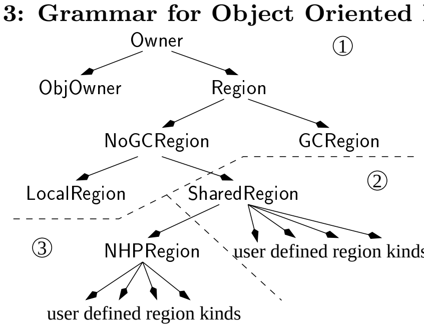

Figure 4: Owner Kind Hierarchy: Section 2.1 uses only Area 1. Sections 2.2 & 2.3 use Areas 2 & 3.

first formal owner is special: it owns the `this` object; the other owners propagate the ownership information. If necessary, methods can declare an additional list of formal owner parameters. Each time new formals are introduced, programmers can specify constraints between them using `where` clauses [18]. The constraints have the form "o_1 `owns` o_2" (i.e., $o_1 \succeq_o o_2$) and "o_1 `outlives` o_2" (i.e., $o_1 \succeq o_2$).

Each formal has an owner kind. There is a subkinding relation between owner kinds, resulting in the kind hierarchy from the upper half of Figure 4. The hierarchy is rooted in Owner, that has two subkinds: ObjOwner (owners that are objects, we avoid using Object because it is already used for the root of the class hierarchy) and Region. Region has two subkinds: GCRegion (the kind of the garbage collected heap) and NoGCRegion (for the normal regions). Finally, NoGCRegion has a single subkind, LocalRegion. (At this point, there is no distinction between NoGCRegion and LocalRegion. We will add new kinds in the next sections.)

Region Creation The expression "(`RHandle`$\langle r\rangle$ h) $\{e\}$" creates a new region and introduces two identifiers r and h that are visible inside the scope of e. r is an owner of kind LocalRegion that is bound to the newly created region. h is a runtime value of type `RHandle`$\langle r\rangle$ that is bound to the handler of the region r. The region name r is only a typechecking entity; it is erased (together with all the ownership and region type annotations) immediately after typechecking. However, the region handle h is required at runtime when we allocate objects in region r; object allocation is explained in the next paragraph. The newly created region is outlived by all regions that existed when it was created; it is destroyed at the end of the scope of e. This corresponds to a "last in first out" region lifetime. As we mentioned before, in addition to the user created regions, we have special regions: the garbage collected region heap (with handle h_heap) and the "immortal" memory immortal (with handle h_immortal). Objects allocated in the immortal region are never deallocated. heap and immortal are never destroyed; hence, they outlive all regions. We also allow methods to allocate objects in initialRegion. The special region initialRegion denotes the most recent region that was created before the method was called. We use runtime support to acquire the handle of initialRegion.

Object Allocation New objects are created using the expression "`new` $cn\langle o_{1..n}\rangle$". o_1 is the owner of the new object. If o_1 is a region, the new object is allocated there; otherwise, it is allocated in the region where the object o_1 is allocated. For the purpose of typechecking, region handles are unnecessary. However, at runtime we need a handle of the region we allocate in. The type rules check that we can obtain such a handle (more details in Section 2.4). If o_1 is a region r, a handle of r must be in the environment. Therefore, if a method has to allocate memory in a specific region that is passed to it as an owner parameter, then it also needs to receive the corresponding region handle as an argument.

We can instantiate a formal owner parameter with an in-scope formal, a region name, or the `this` object. For every type $cn\langle o_{1..n}\rangle$ with multiple owners, our type system statically enforces the constraint that $o_i \succeq o_1$, for all $i \in \{1..n\}$. In addition, if an object of type $cn\langle o_{1..n}\rangle$ has a method mn, and if a formal owner parameter of mn is instantiated with an object obj, then our system ensures that $obj \succeq o_1$. These restriction enable the type system to statically enforce object encapsulation and prevent dangling references.

Example We illustrate our type system with the example in Figure 5. A TStack is a stack of T objects. It is implemented using a linked list. The TStack class is parameterized by stackOwner and TOwner. stackOwner owns the new TStack object and TOwner owns the T objects contained in the TStack. The code specifies that the stack object owns the list; therefore the list cannot be accessed from outside the stack object. The program creates two regions `r1` and `r2` such that `r1` outlives `r2`, and declares several stacks: `s1` is a stack allocated in region `r2`, whose elements are allocated in `r2`; `s2` is a stack allocated in `r2`, with elements in `r1`; etc. However, the declaration of `s6` does not typecheck. It is declared as a stack allocated in `r1`, with elements from `r2`. In each instantiation of `TStack`, all owners must outlive the first owner, but `r2` does *not* outlive `r1`; if this declaration were legal, when `r2` is deallocated we would obtain several dangling pointers from the `TNode` objects. Figure 6 presents the ownership and the outlives relations from this example, (assuming the stacks contain two elements each). We use circles for objects, rectangles for regions, solid arrows for ownership and dashed arrows for the outlives relation between regions.

Safety Guarantees The following two theorems summarize our safety guarantees. Theorem 3.1 states the memory

```
class TStack<Owner stackOwner, Owner TOwner> {
  TNode<this, TOwner> head = null;

  void push(T<TOwner> value) {
    TNode<this, TOwner> newNode = new TNode<this, TOwner>;
    newNode.init(value, head);  head = newNode;
  }

  T<TOwner> pop() {
    if(head == null) return null;
    T<TOwner> value = head.value;  head = head.next;
    return value;
  }
}

class TNode<Owner nodeOwner, Owner TOwner> {
  T<TOwner> next;
  TNode<nodeOwner, TOwner> next;

  void init(T<TOwner> v, TNode<nodeOwner, TOwner> n) {
    this.value = v;  this.next = n;
  }
}

(RHandle<r1> h1) {
  (RHandle<r2> h2) {
    TStack<r2,       r2>       s1;
    TStack<r2,       r1>       s2;
    TStack<r1,       immortal> s3;
    TStack<heap,     immortal> s4;
    TStack<immortal, heap>     s5;
 /* TStack<r1,       r2>       s6; illegal! */
 /* TStack<heap,     r1>       s7; illegal! */
} }
```

Figure 5: Stack of T Objects

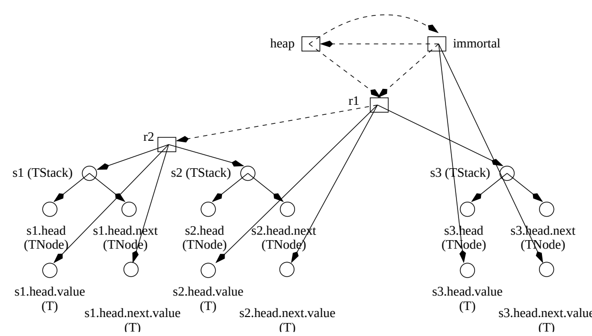

Figure 6: TStack Ownership and Outlives Relations

safety property. Theorems 3.2 and 4 state the object encapsulation property. Note that objects owned by regions are not encapsulated within other objects.

THEOREM 3. *If objects o_1 and o_2 are allocated in regions r_1 and r_2 respectively, and field fd of o_1 points to o_2, then*

1. *r_2 outlives r_1.*

2. *Either owner of $o_2 \succeq_o o_1$, or owner of o_2 is a region.*

PROOF. Suppose *class* $cn\langle f_{1..n}\rangle\{...\ T\langle x_1, ...\rangle\ fd\ ...\}$ is the class of o_1. Field fd of type $T\langle x_1, ...\rangle$ contains a reference to o_2. x_1 must therefore own o_2. x_1 can therefore be either 1) heap, or 2) immortal, or 3) this, or 4) f_i, a class formal. In the first two cases, $r_2 = x_1 \succeq r_1$, and owner of $o_2 = x_1$ is a region. In Case 3, $r_2 = r_1 \succeq r_1$, and owner of $o_2 = o_1 \succeq_o o_1$. In Case 4, we know that $f_i \succeq f_1$, since all owners in a legal type outlive the first owner. Therefore, owner of $o_2 = x_1 = f_i \succeq f_1 \succeq$ this $= o_1$. If owner of o_2 is an object, we know from Lemma 1 that owner of $o_2 \succeq_o o_1$. This also implies that $r_2 = r_1 \succeq r_1$. If the owner of o_2 is a region, we know from Lemma 2 that there exists region r such that owner of $o_2 \succeq r$ and $r \succeq_o o_1$. For the last relation to be true, $r_2 = r \succeq r_1$. □

THEOREM 4. *If a variable v in a method mn of an object o_1 points to an object o_2, then either owner of $o_2 \succeq_o o_1$, or owner of o_2 is a region.*

PROOF. Similar to the proof of Theorem 3, except that now we have a fifth possibility for the owner of o_2: a formal method parameter that is a region or initialRegion (that are not required to outlive o_1). In this case, owner of o_2 is a region. The other four cases are identical. □

2.2 Regions for Multithreaded Programs

In this section, we extend our language to allow multi-threaded programs. Figure 7 presents the language extensions. A `fork` is similar to a method call, except that the invoked method is evaluated in a concurrent thread; the parent thread does not wait for the completion of the new thread, and the result of the method is not used. Our unstructured concurrency model (similar to Java's model) is incompatible to the regions from Section 2.1 whose lifetime is lexically bound. Those regions can still be used for allocating objects that are thread local (hence the name of the associated region kind, LocalRegion), but objects shared by multiple threads require *shared* regions, of kind SharedRegion.

$$
\begin{array}{rcl}
P & ::= & def^*\ srkdef^*\ e \\[6pt]
srkdef & ::= & \texttt{regionKind}\ srkn\langle formal^*\rangle\ \texttt{extends}\ srkind \\
 & & \quad \texttt{where}\ constr^*\ \texttt{\{}\ field^*\ subsreg^*\ \texttt{\}} \\
rkind & ::= & \ldots \text{ as in Figure 3 } \ldots \mid srkind \\
srkind & ::= & srkn\langle owner^*\rangle \mid \mathsf{SharedRegion} \\
subsreg & ::= & srkind\ rsub \\[6pt]
e & ::= & \ldots \text{ as in Figure 3 } \ldots \mid \\
 & & \texttt{fork}\ v.mn\langle owner^*\rangle(v^*) \mid \\
 & & \texttt{(RHandle}\langle rkind\ \ r\rangle\ h\texttt{)}\ \texttt{\{}\ e\ \texttt{\}} \mid \\
 & & \texttt{(RHandle}\langle r\rangle\ h\ \texttt{=}\ [\texttt{new}]_{\text{opt}}\ h.rsub\texttt{)}\ \texttt{\{}\ e\ \texttt{\}} \mid \\
 & & h.fd \mid h.fd\ \texttt{=}\ v \\[6pt]
srkn & \in & \text{shared region kind names} \\
rsub & \in & \text{shared subregion names}
\end{array}
$$

Figure 7: Extensions for Multithreaded Programs

Shared Regions "(RHandle⟨*rkind* *r*⟩ *h*) {*e*}" creates a shared region (ignore *rkind* for the moment); inside expression e, the identifiers r and h are bound to the region, respectively the region handle. Inside e, r and h can be passed to children threads. The objects allocated inside a shared region are not deallocated as long as some thread can still access them. To ensure this, each thread maintains a stack of shared regions it can access, and each shared region maintains a counter of how many such stacks it is an element of. When a new shared region is created, it is pushed on the stack and its counter is initialized to 1. A child thread inherits all the shared regions of its parent thread; the counters of these regions are incremented when the child thread is forked. When the scope of a region name ends (the names of the shared regions are still lexically scoped, even if their lifetimes are not), the corresponding region is popped off the stack and its counter is decremented. When a thread terminates, the counters of all the regions from its stack are decremented. If the counter of a region ever becomes zero, the region is deleted. The typing rule for `fork` checks that no argument passed to the started thread is allocated in a local region; it also checks that no owner passed to the child thread is a local region.

Subregions and Portals Shared regions provide the basis of inter-thread communication. However, in many cases, they are not enough. E.g., imagine two threads, a producer and a consumer, that communicate through a shared region in a repetitive way. In each iteration, the producer allocates some objects in the shared region, that are subsequently used by the consumer. These objects become unreachable after each iteration. However, these objects are not deallocated until the two threads terminate and exit the shared region. To solve this memory leak, we allow shared regions to have subregions. In each iteration, the producer and the consumer enter a subregion of the shared region and use it to allocate/read the object they want to communicate. At the end of the iteration, both threads exit the subregion, its reference count goes to zero, and its objects are deallocated. To allow the two threads to pass the references of the objects they want to communicate, we allow (sub)regions to have *portal* fields. Notice that storing the references in the fields of a "hook" object is not possible: objects allocated outside the subregion cannot point down inside the region and objects allocated in the subregion do not survive between iterations, hence being unusable as "hooks".

Region Kinds In practice, programs can declare several shared region kinds. Each such kind extends another shared region kind and can declare several fields and/or subregions (see grammar rule for *srkdef* in Figure 7). The resulting shared region kind hierarchy has SharedRegion as its root. The new owner kind hierarchy consists of Areas 1 and 2 from Figure 4. Similar to classes, shared region kinds can be generalized with respect to a list of owners that are used in the field types; unlike objects, regions are not owned by anybody so there is no special meaning attached to the first owner. "(RHandle⟨r_2⟩ h_2 = $[\texttt{new}]_{\text{opt}}$ $h_1.rsub$) {e}" evaluates expression e in an environment where r_2 is bound to the subregion *rsub* of the region r_1 that h_1 is a handle of, and h_1 is bound to the handle of r. In addition, if the keyword `new` is present, r is a newly created region, distinct from the previous *rsub* subregion. If h is a handle of a region r, "$h.fd$" reads r's shared field fd, and "$h.fd$ = v" stores a value into that field. The rule for region fields is the same as that for object fields: a shared field of a region r is required to point to an object allocated in r or in a region that outlives r.

Example Figure 8 contains an example that illustrates the use of region fields and subregions. The main thread creates a shared region of kind `CommunicationRegion` and then starts two threads, a producer and a consumer, that communicate through the shared region. In each iteration, the producer enters the `buffer` subregion (of kind `FrameBuffer`), allocates a `Frame` object in it, and stores a reference to the frame in subregion's field `f`. Next, the producer exits the subregion and waits for the consumer. As `f` is non-null, the subregion is not flushed. The consumer enters the subregion, uses the frame object pointed to by its `f` field, sets that field to `null`, and exits the subregion. Now, the subregion is flushed (its counter is zero and all its fields are null) and a new iteration is able to start. In this paper, we

```
regionKind FrameBuffer extends SharedRegion {
  Frame<this> f;
}

regionKind CommunicationRegion extends SharedRegion {
  FrameBuffer buffer;
}

class Producer<CommunicationRegion r> {
  void run(RHandle<r> h) {
    while(true) {
      (RHandle<FrameBuffer buffer> hbuffer = h.buffer) {
        Frame<buffer> frame = new Frame<buffer>;
        grab_image(frame);
        hbuffer.f = frame;
      }
      ... // wake up the consumer
      ... // wait for the consumer
}}}

class Consumer<CommunicationRegion r> {
  void run(RHandle<r> h) {
    while(true) {
      ... // wait for the producer
      (RHandle<FrameBuffer buffer> hbuffer = h.buffer) {
        Frame<buffer> frame = hbuffer.f;
        hbuffer.f = null;
        process_image(frame);
      }
      ... // wake up the producer
}}}

(RHandle<CommunicationRegion r> h) {
  fork (new Producer<r>).run(h);
  fork (new Consumer<r>).run(h);
}
```

Figure 8: Producer Consumer Example

do not discuss synchronization issues; the synchronization primitives in the example are similar to those in Java.

Flushing Subregions When all the objects in a subregion become inaccessible, the subregion is flushed, i.e., all objects allocated inside it are deallocated. We do not flush a subregion if its counter is positive. Furthermore, we do not flush a subregion r if any of its portal fields is non-null (to allow some thread to enter it later and use those objects) or if any of r's subregions has not been flushed yet (because the objects from those subregions might point to objects from r). Recall that subregions are a way of "packaging" some data and sending it to another thread; the receiver thread looks inside the subregion (starting from the portal fields) and uses the data. Therefore, as long as a subregion with non-null portal fields is reachable (i.e., a thread may obtain a handler of it), the objects allocated inside it can be reachable even if no thread is currently in the subregion.

2.3 Regions for Real-Time Programs

A real-time program consists of a set of real-time threads with hard real-time constraints, a set of non real-time threads, and a special garbage collection thread. (This is a conceptual model; actual implementations might differ.) A real-time thread has strict deadlines for completing its tasks. Such a thread cannot afford to be interrupted for an unbounded amount of time by the garbage collector. However, the garbage collector cannot execute in parallel with a thread that creates/destroys heap roots, i.e., pointers to heap objects, because this might cause it to collect reachable objects. Therefore, special care should be taken to ensure that the real-time threads do not hold and do not overwrite any heap reference. (The last restriction is needed to support copying collectors.) The language extensions for real-time programs are in Figure 9.

$$
\begin{array}{rcl}
\mathit{meth} & ::= & t\ \mathit{mn}\langle \mathit{formal}^*\rangle\texttt{(}(t\ p)^*\texttt{)}\ \mathit{effects}\ \texttt{where}\ \mathit{constr}^*\ \{e\} \\
\mathit{effects} & ::= & \texttt{accesses}\ \mathit{owner}^* \\
\mathit{srkind} & ::= & \ldots \text{ as in Figure 7 } \ldots \mid \mathsf{NHPRegion} \\
\mathit{subsreg} & ::= & \mathit{srkind} : \mathit{rpol}\ \mathit{rsub} \\
\mathit{rpol} & ::= & \mathsf{LT}(\mathit{size}) \mid \mathsf{VT} \\
k & ::= & \ldots \text{ as in Figure 3 } \ldots \mid \mathit{rkind} : \mathsf{LT} \\
 & & \\
e & ::= & \ldots \text{ as in Figure 7 } \ldots \mid \\
 & & \texttt{(RHandle}\langle \mathit{rkind} : \mathit{rpol}\ \ r\rangle\ h\texttt{)}\ \{\ e\ \} \mid \\
 & & \texttt{NoGC\_fork}\ v.\mathit{mn}\langle \mathit{owner}^*\rangle\texttt{(}v^*\texttt{)}
\end{array}
$$

Figure 9: Extensions for Real-Time Programs

Effects The expression "`NoGC_fork` $v.mn\langle owner^*\rangle(v^*)$" starts a real-time thread that is guaranteed not to require any interaction with the garbage collector. To statically check this, we introduce method *effects*. The effect clause of a method lists the owners (some of them regions) that the method *accesses*. Accessing a region means allocating an object in that region. Accessing an object means reading/overwriting a reference to that object or allocating another object owned by it; notice that in our system reading/writing a field of an object is not considered an access to that object. If a method's effects clause consists of the owners $o_{1..n}$ then that method and the methods it invokes and the threads that it starts (transitively) are guaranteed to access only objects and regions owned by $o_{1..n}$ (transitively and reflexively). The typing rule for a `NoGC_fork` expression checks all the constraints for a normal `fork` expression: local regions or objects allocated in local regions cannot be passed to the started thread as owner/normal parameters. It also checks that no parameter is a heap object. Finally, it checks that none of the owners mentioned in the effects of the started thread method is heap or an object allocated in heap. If a `NoGC_fork` expression typechecks, the started thread does not receive any heap reference; also, it cannot obtain one by creating an object in the heap or by reading a heap reference from an object field: the type system makes sure that in both cases heap or an object allocated in heap appears in the method effects.

Region Allocation Policies Our system supports two allocation policies for regions. One policy is to allocate memory on demand, as new objects are created in a region. Our runtime system allocates memory at the granularity of memory pages. For efficiency, we can have our own trivial allocator run inside these pages. Pages allocated for a region can be chained into a linked list. We can maintain a pointer to the first free address inside the last page. When we need to allocate a new object, we can simply slide that pointer. If this makes the pointer go outside its page, we can allocate a new page, and allocate the new object at its beginning. Allocating a new object can take unbounded time (or might not even succeed), because of the possible memory page allocation. Flushing the region frees all the memory pages allocated for that region. Following the RTSJ terminology, we call such regions VT (Variable Time) regions.

The other policy is to allocate all the memory for a region at its creation. The programmer must provide an upper bound for the total size of the objects that will be allocated in the region. Allocating an object requires sliding a pointer,

$$\frac{\begin{array}{c} E_2 = E,\ \mathsf{LocalRegion}\ r,\ \mathtt{RHandle}\langle r\rangle\ h,\ (r_e \succeq r)_{\forall r_e \in Regions(E)} \\ P \vdash_{\text{env}} E_2 \quad P; E_2; X, r; r \vdash e : t \quad E \vdash X \succeq_o \mathsf{heap} \end{array}}{P; E; X; r_{cr} \vdash \mathtt{(RHandle}\langle r\rangle\ h\mathtt{)}\ \{e\} : \mathsf{int}}$$

$$\frac{\begin{array}{c} P; E; X; r_{cr} \vdash e : cn\langle o_{1..n}\rangle \quad P \vdash (t\ fd) \in cn\langle (k_i\ fn_i)_{i\in\{1..n\}}\rangle \\ t' = t[o_1/fn_1]..[o_n/fn_n] \quad t' = cn\langle o'_{1..m}\rangle \rightarrow E \vdash X \succeq_o o'_1 \end{array}}{P; E; X; r_{cr} \vdash e.fd : t'}$$

$$\frac{\begin{array}{c} \mathtt{class}\ cn\langle (k_i\ fn_i)_{i\in\{1..n\}}\rangle\ \ldots\ \mathtt{where}\ constr_{1..c}\ \ldots \in P \\ \forall i \in \{1..m\},\ (E \vdash_{\text{k}} o_i : k'_i \quad P \vdash k'_i \leq_{\text{k}} k_i) \quad E \vdash o_i \succeq o_1 \\ \forall i \in \{1..c\},\ E \vdash constr_i[o_1/fn_1]..[o_m/fn_m] \\ E \vdash X \succeq_o o_1 \quad E \vdash_{\text{av}} \mathsf{RH}(o_1) \end{array}}{P; E; X; r_{cr} \vdash \mathtt{new}\ cn\langle o_{1..n}\rangle : cn\langle o_{1..n}\rangle}$$

$$\frac{\begin{array}{c} P; E; X; r_{cr} \vdash v_0 . mn\langle o_{1..n}\rangle(v_{1..k}) : t \\ \forall i \in \{0..k\},\ (E \vdash_{\text{v}} \text{RKind}(v_i) = rkind_i \ \wedge\ rkind_i \neq \mathsf{LocalRegion}) \\ \forall i \in \{1..n\},\ (E \vdash_{\text{o}} \text{RKind}(o_i) = rkind'_i \ \wedge\ rkind'_i \neq \mathsf{LocalRegion}) \\ E \vdash_{\text{o}} \text{RKind}(r_{cr}) = k_{cr} \quad k_{cr} \neq \mathsf{LocalRegion} \end{array}}{P; E; X; r_{cr} \vdash \mathtt{fork}\ v_0 . mn\langle o_{1..n}\rangle(v_{1..k}) : \mathsf{int}}$$

$$\frac{E = E_1,\ \mathtt{RHandle}\langle r\rangle\ h,\ E_2}{E \vdash_{\text{av}} \mathsf{RH}(r)} \qquad \frac{}{E \vdash_{\text{av}} \mathsf{RH(this)}} \qquad \frac{E \vdash o_1 \succeq_o o_2 \quad E \vdash_{\text{av}} \mathsf{RH}(o_2)}{E \vdash_{\text{av}} \mathsf{RH}(o_1)} \qquad \frac{E \vdash o_1 \succeq_o o_2 \quad E \vdash_{\text{av}} \mathsf{RH}(o_1)}{E \vdash_{\text{av}} \mathsf{RH}(o_2)}$$

Figure 10: Some Typechecking Rules. The Complete Set of Rules is in the Appendix.

but if we try to overflow the region, we do not allocate any new memory page. Instead, we throw an exception to signal that the region size was too small. Allocating a new object is linear in its size: sliding the pointer takes constant time, but we also have to set to zero each allocated byte. Flushing the region simply resets a pointer, and, importantly, *does not* free the memory allocated for the region. We call regions that use this allocation policy *LT* (Linear Time) regions. Once we have an LT subregion, we can repeatedly enter it, allocate objects in it, exit it (thus flushing it), and re-enter it without having to allocate a new chunk of memory. This is possible because flushing an LT region does not free its memory. LT regions are ideal for real-time threads: once such a region is created (with a large enough upper bound), all memory allocation will succeed, in linear time; moreover, the region can be flushed and reused without memory allocation.

We allow the user to specify the region allocation policy (LT or VT) when a new region is created. The policy for the subregions is declared in the shared region kind declarations. When we specify an LT policy, we also have to specify the size of the region (in bytes). An expression "$\mathtt{(RHandle}\langle rkind : rpol\ \ r\rangle\ \ h\mathtt{)}\ \ \{e\}$" creates a region with allocation policy *rpol* and allocates memory for all its (transitive) (sub)regions (including itself) that are LT regions. Our type system checks that a region has a finite number of transitive LT subregions. Each time we enter a VT region, or an LT top level region (i.e., a region that is not a subregion), the typechecker requires that heap is in the effects. This is not required when entering an existing LT subregion because no memory is allocated in that case.

No Heap Pointers Regions The garbage collector needs to know all locations that point to heap objects, including those locations that are inside regions. Moreover, a moving collector might have to write to these locations. Suppose we have a real-time thread that uses an LT region that contains several such locations (these heap pointers might have been created by a non real-time thread). The real-time thread can flush the region (by exiting it), re-enter it and allocate objects inside it, thus possibly writing at any location inside the LT region, including those that may be written by the garbage collector! There are several ways of solving this data race: 1) synchronize with the garbage collector, which might introduce unbounded delays, 2) use a conservative non-moving collector, and 3) require that a real-time thread uses only regions that do not contain any heap reference (regardless of whether the thread actually reads those references or not). Our system uses the third solution, which allows it to work with any garbage collector.

We introduce a new shared region kind, NHPRegion (*No Heap Pointers Region*). A shared region of this kind does not contain any object with a field pointing to a heap object. Programmers can define shared region kinds that (transitively) extend NHPRegion. The extended owner kind hierarchy now includes all three areas from Figure 4. A real-time thread is allowed to enter only regions of kind NHPRegion (or a subkind of it). We enforce this by requiring that each time the program enters a region whose kind is not a subkind of NHPRegion, heap is among the effects. (Recall that the typing rule for `NoGC_fork` does not allow the body of a child thread to have heap in its effects.) To enforce that a NHPRegion region does not contain any heap reference, each time we enter such a region we check that the code cannot manipulate any heap reference, similar to the way we check a `NoGC_fork` expression.

2.4 Rules for Typechecking

The previous sections presented the grammar for our core language in Figures 3, 7, and 9. This section presents some of the important type rules. The full set of rules and the complete grammar can be found in the appendix. Figure 10 presents several (simplified) typechecking rules from our system. At the core of our type system lies a set of rules of the form $P; E; X; r_{cr} \vdash e : t$. Such a rule means that in program P, environment E and current region r_{cr}, expression e has type t, and accesses only objects/regions that are owned by the owners from the effect list X. The environment E stores information about the identifiers from the current scope: the type of each variable, the owner kind of each formal owner parameter or region, and the ownership/outlives relation between these owners. Formally, $E ::= \emptyset \mid E,\ t\ v \mid E,\ k\ o \mid E,\ o_2 \succeq_o o_1 \mid E,\ o_2 \succeq o_1$.

A useful auxiliary rule is $E \vdash X_1 \succeq_o X_2$, i.e., the effects X_1 *subsume* the effects X_2: $\forall o \in X_2, \exists g \in X_1$, s.t. $g \succeq_o o$. To prove constraints of the form $g \succeq_o o$, $g \succeq o$ etc. in a specific environment E, the checker uses the constraints from E, and the properties of $\succeq$ and $\succeq_o$: transitivity, reflexivity, $\succeq_o$ implies $\succeq$, and the first owner from the type of an object owns the object.

The expression "$\mathtt{(RHandle}\langle r\rangle\ \ h\mathtt{)}\ \ \{e\}$" creates a local region and evaluates e in an environment where r and h are bound to the new region, respectively to its handle. The associated rule constructs an environment E_2 that extends the original environment E by recording that r has kind LocalRegion and h has type $\mathtt{RHandle}\langle r\rangle$. As r is deleted at the end of e, all existing regions outlive it; E_2 records this too. e should typecheck according to the environment E_2 and the permitted effects X, r (the local region r is a permitted effect inside e). Because creating a region requires memory allocation, X must subsume heap. The expression is evaluated only for its side-effects; this way, we avoid problems with the case when e evaluates to an object from r. Hence, the type of the entire expression is int.

The rule for a field read expression "e.fd" first checks e and finds its type $cn\langle o_{1..n}\rangle$. Next, it verifies that fd is a field of class cn; let t be its declared type. We obtain the type of the entire expression by replacing in t each formal owner parameter fn_i of cn with the corresponding owner o_i. If the expression reads an object, the rule checks that X subsumes the owner of that object.

In the case of "new $cn\langle o_{1..n}\rangle$", we first check that the class cn is defined in P. Next, we check that each formal owner parameter fn_i of cn is instantiated with an owner o_i of appropriate kind, i.e., the kind k'_i of o_i (provided by E) is a subkind of the declared kind of fn_i. We also check that in E, each owner o_i outlives the first owner o_1, and each constraint of cn is satisfied. Allocating a (sub)object means accessing its owner; therefore, we require that X subsumes o_1. The new object is allocated in the region o_1 stands for (if it is a region) or o_1 is allocated in (if it is an object). The rule $E \vdash_{\text{av}} \mathsf{RH}(o_1)$ checks that a handle for this region is available. If $o_1 \in \{\mathsf{heap}, \mathsf{immortal}\}$ this is trivially true; otherwise, we use the bottom four rules from Figure 10. If o_1 is a region, the first rule looks for its handle into the environment. The next two rules use the fact that all objects are allocated in the same region as their owner. Therefore, if $o_1 \succeq_o o_2$ and we have the region handle for one of them, we have the region handle for the other one. Finally, our runtime is able to find the handle of a region where an object (this in particular) is allocated. Notice that these rules do significant reasoning, thus reducing annotation burden.

The case of `fork` is similar to that of a method call. In addition, the rule checks that the started thread receives only regions that are not local (i.e., GCRegion and SharedRegion regions) or objects allocated in such regions. For this, it retrieves the kinds of the regions that the owners stand for or are allocated in and checks that none of these is LocalRegion; similarly for the kinds of the regions where the parameters are allocated. The rules for retrieving these region kinds (not shown here) are similar to those for checking that a region handle is available, rules that we explained in the previous paragraph. The key idea they exploit is that a subobject is allocated in the same region as its owner.

2.5 Type Inference

Although our type system is explicitly typed in principle, it would be onerous to fully annotate every method with the extra type information that our system requires. Instead, we use a combination of inference and well-chosen defaults to significantly reduce the number of annotations needed in practice. We emphasize that our approach to inference is purely intra-procedural and we do not infer method signatures or types of instance variables. Rather, we use a default completion of partial type specifications in those cases. This approach permits separate compilation.

If owners of method local variables are not specified, we use a simple unification-based approach to infer the owners. The approach is similar to the ones in [9, 7]. For parameters unconstrained after unification, we use initialRegion. For unspecified owners in method signatures, we use initialRegion as the default. For unspecified owners in instance variables, we use the owner of this as the default. For static fields, we use immortal as the default. Our default accesses clauses contain all class and method owner parameters and initialRegion.

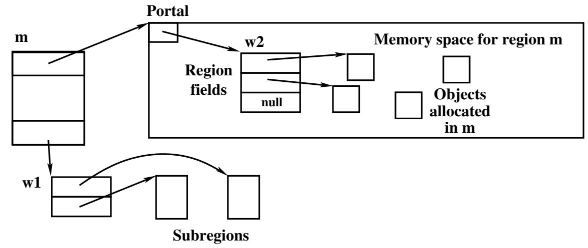

Figure 11: Translation of a region with three fields and two subregions.

2.6 Translation to Real-Time Java

Although our system provides significant improvements over RTSJ (static guarantees, subregions etc.), our language can be translated to RTSJ in a reasonably easy way, by local translation rules. This is mainly due to the fact that being able to do type erasure was a key design goal; e.g., region handles exist specifically for this purpose. Also, RTSJ has mechanisms that are powerful enough to support our features. RTSJ offers `LTMemory` and `VTMemory` regions where it takes linear time, respectively variable time to allocate objects. RTSJ regions are Java objects that point to some memory space. In addition, RTSJ has two special regions: heap and immortal. A thread can allocate in the current region using `new`. A thread can also allocate in any region that it entered using `newInstance`, which requires a region object. RTSJ regions are maintained similarly to our shared regions, by counting the number of threads executing in them. RTSJ regions have one *portal*, which is similar to a region field except that its declared type is `Object`. Most of the translation effort is focused on providing the missing features: subregions and multiple, typed region fields. We discuss the translation of several important features from our type system; the full translation is discussed in [35].

We represent a region r from our system as an RTSJ region `m` plus two auxiliary objects `w1` and `w2` (Figure 11). `m` points to a memory area that is pre-allocated for an LT region, or grown on-demand for a VT region. `m` also points to an object `w1` whose fields point to the representation of r's subregions. (We subclass `LT/VTMemory` to add an extra field.) In addition, `m`'s portal points to an object `w2` that serves as a wrapper for r's fields. `w2` is allocated in the memory space attached to `m`, while `m` and `w1` are allocated in the current region at the moment `m` was created.

The translation of "new $cn\langle o_{1..n}\rangle$" requires a reference to (i.e., a handle of) the region we allocate in. If this is the same as the current region, we use the more efficient `new`. The type rules already proved that we can obtain the necessary handle, i.e., $E \vdash_{\text{av}} \mathsf{RH}(o_1)$, using the four bottom rules from Figure 10. Those rules "pushed" the statement $E \vdash_{\text{av}} \mathsf{RH}(o)$ up and down the ownership relation until we obtained an owner for which we can obtain the handle: immortal, heap, this, or a region for which we have a variable that holds the region handle. RTSJ provides mechanisms for retrieving the handle in the first three cases: `ImmortalArea.instance()`, `HeapArea.instance()`, respectively `MethodArea.getMethodArea(Object)`. In the last case, we simply use the handle variable.

Program	Lines of Code	Lines Changed
Array	56	4
Tree	83	8
Water	1850	31
Barnes	1850	16
ImageRec	567	8
http	603	20
game	97	10
phone	244	24

Figure 12: Programming Overhead

Program	Execution Time (sec)		Overhead
	Static Checks	Dymanic Checks	
Array	2.24	16.2	7.23
Tree	4.78	23.1	4.83
Water	2.06	2.55	1.24
Barnes	19.1	21.6	1.13
ImageRec	6.70	8.10	1.21
load	0.667	0.831	1.25
cross	0.014	0.014	1.0
threshold	0.001	0.001	1
hysteresis	0.005	0.006	1
thinning	0.023	0.026	1.1
save	0.617	0.731	1.18

Figure 13: Dynamic Checking Overhead

3. RESULTS

To gain preliminary experience, we implemented several programs in our system. These include two synthetic programs (Array and Tree), two scientific computations (Water and Barnes), several components of an image recognition pipeline (load, cross, threshold, hysteresis, and thinning), and several simple servers (http, game, and phone, a database-backed information sever). In each case, once we understood how the program worked, adding the extra type annotations was fairly straight forward. Figure 12 presents a measure of the programming overhead involved. The figure shows the lines of code that needed type annotations. In most cases, we only had to change code where regions were created.

We also used our RTSJ implementation to measure the execution times of these programs both with and without the dynamic referencing and assignment checks as specified in the Real-Time Specification for Java. Figure 13 presents the running times of the benchmarks both with and without dynamic checks in the absence of garbage collection. Our synthetic programs (Array and Tree) were written specifically to maximize the check overhead—our development goal was to maximize the ratio of assignments to other computation. The synthetic programs exhibit the largest performance increases—they run approximately 7.2 and 4.8 times faster, respectively, without checks. The performance improvements for the scientific programs and image processing components provides a more realistic picture of the check overhead. These programs have more modest performance improvements, running up to 1.25 times faster without checks. For the servers, the running time is dominated by the network processing overhead and check removal has virtually no effect. We present the overhead of dynamic referencing and assignment checks in this paper. For a detailed analysis of the performance of a full range of RTSJ features, see [15, 16].

4. RELATED WORK

The seminal work in [37, 36] introduces a static type system for region-based memory management for ML. Our system extends this to object-oriented programs by combining the benefits of region types and ownership types in a unified type system framework. Our system extends region types to multithreaded programs by allowing long-lived threads to share objects without using the heap and without having memory leaks. Our system extends region types to real-time programs by ensuring that real-time threads do not interfere with the garbage collector.

[2] provides a static analysis to free some regions early to avoid the constraints of LIFO region lifetimes in [37, 36]. Capability Calculus [17] proposes a general type system for freeing regions based on linear types. Vault's type system [19] allows a region to be freed before it leaves scope. These systems are more expressive than our framework. However, these systems do not handle object-oriented programs and the consequent subtyping, multithreaded programs with shared regions, or real-time programs with real-time threads. Moreover, we can augment our system with linear types and other techniques used in the above systems. In fact, other systems have already combined ownership-based type systems and unique pointers [11, 9, 3].

RegJava [30] describes a region type system for object-oriented programs that handles subtyping and method overriding. We improve on this by combining the benefits of ownership types and region types in a unified framework. Cyclone [26] is a dialect of C with a region type system. An extension to Cyclone handles multithreaded programs and provides shared regions [25]. Our work improves on this by providing subregions in shared regions and portal fields in subregions, so that long-lived threads can share objects without using the heap and without having memory leaks. Other systems for regions [23, 24] use runtime checks to ensure memory safety. These systems are more flexible, but they do not statically ensure safety.

To our knowledge, ours is the first static type system that handles memory management in real-time programs. [31, 20] automatically translate Java code into RTSJ code using off-line dynamic analysis to determine the lifetime of an object. Unlike our system, this system does not require type annotations. It does, however, impose a runtime overhead and it is not safe because the dynamic analysis might miss some execution paths. Programmers can use this dynamic analysis to obtain suggestions for region type annotations. If a program typechecks, the suggested annotations were sound. We expect such tools to be of great value in helping programmers write precise type annotations. We previously used escape analysis [34] to remove RTSJ runtime checks [33]. However, the analysis works only for some programs. Our type system is more flexible.

5. CONCLUSIONS

The Real-Time Specification for Java (RTSJ) allows a program to create real-time threads with hard real-time constraints. The RTSJ uses runtime checks to ensure memory safety. This paper presents a static type system that guarantees that these runtime checks will never fail for well-typed programs. Our type system therefore 1) provides an important safety guarantee for real-time programs and 2) makes it possible to eliminate the runtime checks and their associated overhead. Our system also makes several contributions over previous work on region types. For object-oriented programs, it combines region types and ownership types in a unified type system framework. For multithreaded programs, it allows long-lived threads to share objects without using the heap and without having memory leaks. For real-time programs, it ensures that real-time threads do not interfere with the garbage collector. We have implemented several programs in our system. Our experience indicates that our type system is sufficiently expressive and requires little programming overhead. We also ran these programs on our RTSJ platform. Our experiments show that eliminating the RTSJ runtime checks using a static type system can significantly speed-up a real-time program.

6. REFERENCES

[1] O. Agesen, S. N. Freund, and J. C. Mitchell. Adding type parameterization to the Java language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1997.

[2] A. Aiken, M. Fahndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Programming Language Design and Implementation (PLDI)*, June 1995.

[3] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[4] W. Beebee, Jr. Region-based memory management for Real-Time Java. MEng thesis, Massachusetts Institute of Technology, September 2001.

[5] W. Beebee, Jr. and M. Rinard. An implementation of scoped memory for Real-Time Java. In *First International Workshop on Embedded Software (EMSOFT)*, October 2001.

[6] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000. Latest version available from http://www.rtj.org.

[7] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[8] C. Boyapati, B. Liskov, and L. Shrira. Ownership types and safe lazy upgrades in object-oriented databases. Technical Report TR-858, MIT Laboratory for Computer Science, July 2002.

[9] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2001.

[10] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

[11] D. Clarke and T. Wrigstad. External uniqueness. September 2002.

[12] D. G. Clarke. Ownership and containment. PhD thesis, University of New South Wales, Australia, July 2001.

[13] D. G. Clarke and S. Drossopoulou. Ownership, encapsulation and disjointness of type and effect. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[14] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

[15] A. Corsaro and D. Schmidt. The design and performance of the jRate Real-Time Java implementation. In *International Symposium on Distributed Objects and Applications (DOA)*, October 2002.

[16] A. Corsaro and D. Schmidt. Evaluating Real-Time Java features and performance for real-time embedded systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, September 2002.

[17] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Principles of Programming Languages (POPL)*, January 1999.

[18] M. Day, R. Gruber, B. Liskov, and A. C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1995.

[19] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[20] M. Deters and R. Cytron. Automated discovery of scoped memory regions for Real-Time Java. In *International Symposium on Memory Management (ISMM)*, June 2002.

[21] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Principles of Programming Languages (POPL)*, January 1998.

[22] C. A. Frâncu. Real-time scheduling for Java. MEng thesis, Massachusetts Institute of Technology, June 2002.

[23] D. Gay and A. Aiken. Memory management with explicit regions. In *Programming Language Design and Implementation (PLDI)*, June 1998.

[24] D. Gay and A. Aiken. Language support for regions. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[25] D. Grossman. Type-safe multithreading in Cyclone. In *Workshop on Types in Language Design and Implementation (TLDI)*, January 2003.

[26] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[27] D. Lea. JSR166: Concurrency utilities.

[28] K. R. M. Leino and G. Nelson. Data abstraction and information hiding. Research Report 160, Compaq Systems Research Center, November 2000.

[29] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Principles of Programming Languages (POPL)*, January 1988.

[30] H. N. M V Christiansen, F Henglein and P. Velschow. Safe region-based memory management for objects. Technical Report D-397, DIKU, University of Copenhagen, October 1998.

[31] N. L. Morgan Deters and R. Cytron. Translation of Java to Real-Time Java using aspects. In *International Workshop on Aspect-Oriented Programming and Separation of Concerns*, August 2001.

[32] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *Principles of Programming Languages (POPL)*, January 1997.

[33] A. Sălcianu. Pointer and escape analysis for multithreaded programs. In *Principles and Practice of Parallel Programming (PPoPP)*, June 2001.

[34] A. Sălcianu. Pointer analysis and its applications for Java programs. MEng thesis, Massachusetts Institute of Technology, September 2001.

[35] A. Sălcianu, C. Boyapati, W. Beebee, Jr., and M. Rinard. A type system for safe region-based memory management in Real-Time Java. Technical Report TR-869, MIT Laboratory for Computer Science, November 2002.

[36] M. Tofte and J. Talpin. Region-based memory management. In *Information and Computation 132(2)*, February 1997.

[37] M. Tofte and J. Talpin. Implementing the call-by-value λ-calculus using a stack of regions. In *Principles of Programming Languages (POPL)*, January 1994.

APPENDIX

A. COMPLETE GRAMMAR

$$
\begin{array}{rcl}
P & ::= & \mathit{def}^* \ \mathit{srkdef}^* \ e \\
\mathit{def} & ::= & \texttt{class}\ \mathit{cn}\langle \mathit{formal}{+}\rangle\ \texttt{extends}\ c\ \texttt{where}\ \mathit{constr}^*\ \{\mathit{field}^*\ \mathit{meth}^*\} \\
\mathit{formal} & ::= & k\ \mathit{fn} \\
c & ::= & \mathit{cn}\langle \mathit{owner}{+}\rangle \mid \mathsf{Object}\langle \mathit{owner}\rangle \\
\mathit{owner} & ::= & \mathit{fn} \mid r \mid \mathsf{this} \mid \mathsf{initialRegion} \\
\mathit{field} & ::= & t\ \mathit{fd} \\
\mathit{meth} & ::= & t\ \mathit{mn}\langle \mathit{formal}^*\rangle \texttt{((}t\ p\texttt{)}^*\texttt{)}\ \mathit{effects}\ \texttt{where}\ \mathit{constr}^*\ \{e\} \\
\mathit{effects} & ::= & \texttt{accesses}\ \mathit{owner}^* \\
\mathit{constr} & ::= & \mathit{owner}\ \mathsf{owns}\ \mathit{owner} \mid \mathit{owner}\ \mathsf{outlives}\ \mathit{owner} \\
t & ::= & c \mid \texttt{int} \mid \texttt{RHandle}\langle r\rangle \\
 & & \\
\mathit{srkdef} & ::= & \texttt{regionKind}\ \mathit{srkn}\langle \mathit{formal}^*\rangle\ \texttt{extends}\ \mathit{srkind}\ \texttt{where}\ \mathit{constr}^*\ \{\mathit{field}^*\ \mathit{subsreg}^*\} \\
\mathit{subsreg} & ::= & \mathit{srkind} : \mathit{rpol}\ \ \mathit{rsub} \\
\mathit{srkind} & ::= & \mathit{srkn}\langle \mathit{owner}^*\rangle \mid \mathsf{SharedRegion} \mid \mathsf{NHPRegion} \\
\mathit{rkind} & ::= & \mathit{srkind} \mid \mathsf{Region} \mid \mathsf{NoGCRegion} \mid \mathsf{GCRegion} \mid \mathsf{LocalRegion} \\
\mathit{rpol} & ::= & \mathsf{LT}(\mathit{size}) \mid \mathsf{VT} \\
k & ::= & \mathsf{Owner} \mid \mathsf{ObjOwner} \mid \mathit{rkind} \mid \mathit{rkind} : \mathsf{LT} \\
 & & \\
e & ::= & v \mid \mathsf{h\_heap} \mid \mathsf{h\_immortal} \mid \texttt{let}\ v\ \texttt{=}\ e\ \texttt{in}\ \{\ e\ \} \mid \\
 & & v.\mathit{fd} \mid v.\mathit{fd} = v \mid v\,.\,\mathit{mn}\langle \mathit{owner}^*\rangle \texttt{(}v^*\texttt{)} \mid \texttt{new}\ c \mid \\
 & & \texttt{fork}\ v\,.\,\mathit{mn}\langle \mathit{owner}^*\rangle\texttt{(}v^*\texttt{)} \mid \texttt{NoGC\_fork}\ v\,.\,\mathit{mn}\langle \mathit{owner}^*\rangle\texttt{(}v^*\texttt{)} \mid \\
 & & \texttt{(RHandle}\langle r\rangle\ h\texttt{)}\ \{\ e\ \} \mid \\
 & & \texttt{(RHandle}\langle \mathit{rkind} : \mathit{rpol}\ \ r\rangle\ h\texttt{)}\ \{\ e\ \} \mid \\
 & & \texttt{(RHandle}\langle r\rangle\ h\ \texttt{=}\ [\texttt{new}]_{\mathrm{opt}}\ h\,.\,\mathit{rsub}\texttt{)}\ \{\ e\ \} \mid \\
 & & h\,.\,\mathit{fd} \mid h\,.\,\mathit{fd}\ \texttt{=}\ v \\
 & & \\
h & ::= & v \\
 & & \\
\mathit{cn} & \in & \text{class names} \\
\mathit{fd} & \in & \text{field names} \\
\mathit{mn} & \in & \text{method names} \\
\mathit{fn} & \in & \text{formal identifiers} \\
v, p & \in & \text{variable names} \\
r & \in & \text{region identifiers (including } \mathsf{heap} \text{ and } \mathsf{immortal}) \\
\mathit{srkn} & \in & \text{shared region kind names} \\
\mathit{rsub} & \in & \text{shared subregion names}
\end{array}
$$

B. TYPE RULES

In this section, we formally describe our type system. To simplify the presentation of key ideas, we describe our type system in the context of the previously introduced small language. However, our approach extends to the whole of Java and other similar languages. Throughout this section, we try to use the same notations as in the grammar. To save space, we use o instead of *owner* and f instead of *formal*. We also use g and a as alternate variables ranging over the set of owners. We also suppose the program source has been preprocessed by replacing each constraint "o_1 owns o_2" with non-ASCII, but shorter form "$o_1 \succeq_o o_2$" and each constraint "o_1 outlives o_2" with "$o_1 \succeq o_2$."

The core of our type system is a set of typing judgments of the form $P; E; X; r_{cr} \vdash e : t$. The meaning of such a rule is that in the context of the program P, environment E, e yields type t and accesses only objects (transitively) owned by the owners from X (X is simply a list of owners: $X ::= o^*$). P, the program being checked, is included to provide information about class definitions. The typing environment E provides information about the type of the free variables of e ($t\ v$, i.e., variable v has type t), the kind of the owners currently in scope ($k\ o$. i.e., owner o has kind k), and the two relations between owners: the "ownership" relation ($o_2 \succeq_o o_1$, i.e., o_2 owns o_1) and the "outlives" relation ($o_2 \succeq o_1$, i.e., o_2 outlives o_1). More formally, $E ::= \emptyset \mid E,\ t\ v \mid E,\ k\ o \mid E,\ o_2 \succeq_o o_1 \mid E,\ o_2 \succeq o_1$.

The table below presents the format and the meaning of all our typing rules:

Typing rule	Meaning
$\vdash P : t$	Program P has type t.
$P \vdash_{\text{def}} def$	*def* is a well formed class definition from program P.
$P \vdash_{\text{srkdef}} srkdef_j$	$srkdef_j$ is a well formed shared region kind definition from program P.
$P; E; X; r_{cr} \vdash e : t$	In program P, environment E, and current region r_{cr}, expression e has type t. Its evaluation accesses only objects (transitively) outlived by owners listed in the effects X.
$P \vdash_{\text{env}} E$	E is a well formed environment with respect to program P.
$P; E \vdash_{\text{meth}} meth$	Method definition *meth* is well defined with respect to program P and environment E.
$P \vdash mbr \in c$	Class c defines or inherits "member" definition *mbr*. "Member" refers to a field or a method: $mbr = field \mid meth$.
$P \vdash rmbr \in rkind$	Shared region kind *rkind* defines or inherits "region member" definition *rmbr*. "Region member" refers to a field or a subregion: $rmbr = field \mid subsreg$.
$P; E \vdash_{\text{type}} t$	t is a well formed type with respect to program P and environment E.
$P \vdash t_1 \leq t_2$	t_1 is a subtype of t_2, with respect to program P.
$P; E \vdash_{\text{okind}} k$	k is a well formed owner kind, with respect to program P and environment E.
$P \vdash k_1 \leq_{\text{k}} k_2$	k_1 is a subkind of k_2, with respect to program P and environment E.
$E \vdash X_2 \succeq_o X_1$	Effect X_1 is subsumed by effect X_2, with respect to environment E.
$E \vdash constr$	In environment E, *constr* is well formed (i.e., the owners involved in it are well formed) and satisfied.
$E \vdash_{\text{k}} o : k$	In environment E, o is a well formed owner with kind k.
$E \vdash_{\text{o}} \text{RKind}(o) = k$	Owner o is allocated in a region of kind k.
$E \vdash_{\text{v}} \text{RKind}(v) = k$	v points to an object allocated in a region of kind k.
$E \vdash_{\text{av}} \mathsf{RH}(o)$	The handle of the region o is allocated in (if o is an object) or o stands for (if it is a region) is available in the environment E.
$E \vdash o_2 \succeq_o o_1$	In environment E, owner o_2 (an object or a region) owns owner o_1 (which must be an object).
$E \vdash o_2 \succeq o_1$	In environment E, owner o_2 outlives owner o_1.

We use several auxiliary predicates, almost all of them very straightforward. The only exception is *InheritanceOK*(P) which will be defined formally later. The table below presents a description of these auxiliary predicates:

Predicate	Meaning
$\mathit{WFClasses}(P)$	No class is defined twice and there is no cycle in the class hierarchy.
$\mathit{WFRegionKinds}(P)$	No region kind is defined twice and there is no cycle in the region kind hierarchy.
$\mathit{FieldsOnce}(P)$	No class or region kind contains two fields with the same name, either declared or inherited.
$\mathit{MethodsOnce}(P)$	No class declares two methods with the same name.
$\mathit{InheritanceOK}(P)$	The requirements (i.e., constraints) of a sub-class/kind are included in the super-class/kind requirements. For overriding methods, in addition to the usual subtyping relations between the return/parameter types, the requirements of the overrider are included in the overridden method requirements; similarly for the effects. This predicate will be formally defined later.

B.1 Well Typed Programs: $\vdash P : t$

[PROG]

$$\frac{\begin{array}{c}\mathit{WFClasses}(P) \quad \mathit{WFRegionKinds}(P) \quad \mathit{FieldsOnce}(P) \quad \mathit{MethodsOnce}(P) \quad \mathit{InheritanceOK}(P) \\ P = \mathit{def}_{1..n}\ \ \mathit{srkdef}_{1..r}\ \ e \quad \forall i \in \{1..n\},\ P \vdash_{\text{def}} \mathit{def}_i \quad \forall i \in \{1..r\},\ P \vdash_{\text{srkdef}} \mathit{srkdef}_j \\ P; \emptyset; \mathsf{world}; \mathsf{heap} \vdash e : t\end{array}}{\vdash P : t}$$

B.2 Well Formed Environments: $P \vdash_{\text{env}} E$

[ENV $\emptyset$]

$$\frac{}{P \vdash_{\text{env}} \emptyset}$$

[ENV v]

$$\frac{\begin{array}{c}P \vdash_{\text{env}} E \\ v \notin \mathit{Dom}(E) \\ P; E \vdash_{\text{type}} t\end{array}}{P \vdash_{\text{env}} E,\ t\ v}$$

[ENV OWNER]

$$\frac{\begin{array}{c}P \vdash_{\text{env}} E \\ o \notin \mathit{Dom}(E) \\ P; E \vdash_{\text{okind}} k\end{array}}{P \vdash_{\text{env}} E,\ k\ o}$$

[ENV $\succeq_o$]

$$\frac{\begin{array}{c}P \vdash_{\text{env}} E \\ E \vdash_{\text{k}} o_1 : \mathsf{ObjOwner} \\ E \vdash_{\text{k}} o_2 : k\end{array}}{P \vdash_{\text{env}} E,\ o_1 \succeq_o o_2}$$

[ENV $\preceq$]

$$\frac{\begin{array}{c}P \vdash_{\text{env}} E \\ E \vdash_{\text{k}} o_1 : k_1 \\ E \vdash_{\text{k}} o_2 : k_2\end{array}}{P \vdash_{\text{env}} E,\ o_1 \succeq o_2}$$

where
$$\begin{array}{lcl}\mathit{Dom}(\emptyset) & = & \emptyset \\ \mathit{Dom}(E,\ t\ v) & = & \{v\} \cup \mathit{Dom}(E) \\ \mathit{Dom}(E,\ k\ o) & = & \{o\} \cup \mathit{Dom}(E) \\ \mathit{Dom}(E,\ \_) & = & \mathit{Dom}(E), \text{ otherwise}\end{array}$$

B.3 Well Formed Class Definitions: $P \vdash_{\text{def}} \mathit{def}$

[CLASS DEF]

$$\frac{\begin{array}{c}\mathit{def} = \texttt{class}\ \mathit{cn}\langle (k_i\ \mathit{fn}_i)_{i\in\{1..n\}}\rangle\ \texttt{extends}\ c\ \texttt{where}\ \mathit{constr}_{1..c}\ \{\mathit{field}_{1..m}\ \mathit{meth}_{1..p}\} \quad P = \ldots\ \mathit{def}\ \ldots \\ E = \emptyset,\ (k_i\ \mathit{fn}_i)_{i\in\{1..n\}},\ \mathit{constr}_{1..c},\ \mathit{cn}\langle \mathit{fn}_{1..n}\rangle\ \mathsf{this},\ (\mathit{fn}_i \succeq \mathit{fn}_1)_{i\in\{2..n\}} \quad P \vdash_{\text{env}} E \quad P; E \vdash_{\text{type}} c \\ \forall j \in \{1..m\}, (\mathit{field}_j = t_j\ \mathit{fd}_j \quad P; E \vdash_{\text{type}} t_j) \quad \forall k \in \{1..p\},\ P; E \vdash_{\text{meth}} \mathit{meth}_k\end{array}}{P \vdash_{\text{def}} \mathit{def}}$$

B.4 Well Formed Shared Region Kind Definitions: $P \vdash_{\text{srkdef}} \mathit{srkdef}$

[REGION KIND DEF]

$$\frac{\begin{array}{c}\mathit{srkdef} = \texttt{regionKind}\ \mathit{srkn}\langle (k_i\ \ \mathit{fn}_i)_{i\in\{1..n\}}\rangle\ \texttt{extends}\ r\ \texttt{where}\ \mathit{constr}_{1..c}\ \{\ \mathit{field}_{1..m}\ \ \mathit{subsreg}_{1..s}\ \} \\ P = \ldots\ \mathit{srkdef}\ \ldots \\ E = \emptyset,\ (k_i\ \mathit{fn}_i)_{i\in\{1..n\}},\ \mathit{constr}_{1..c},\ \mathit{srkn}\langle \mathit{fn}_{1..n}\rangle\ \mathsf{this},\ (\mathit{fn}_i \succeq \mathsf{this})_{i\in\{1..n\}} \quad P \vdash_{\text{env}} E \quad P; E \vdash_{\text{okind}} r \\ \forall j \in \{1..m\}, (\mathit{field}_j = t_j\ \mathit{fd}_j \quad P; E \vdash_{\text{type}} t_j) \\ \forall k \in \{1..s\}, (\mathit{subsreg}_k = \mathit{srkind}_k\ \mathit{rsub}_k \quad P; E \vdash_{\text{srkind}} \mathit{srkind}_k)\end{array}}{P \vdash_{\text{srkdef}} \mathit{srkdef}}$$

B.5 Well Formed Types: $P; E \vdash_{\text{type}} t$

[TYPE INT]

$$\frac{}{P; E \vdash_{\text{type}} \mathsf{int}}$$

[TYPE OBJECT]

$$\frac{E \vdash_{\text{k}} o : k}{P; E \vdash_{\text{type}} \mathsf{Object}\langle o \rangle}$$

[TYPE REGION HANDLE]

$$\frac{E \vdash_{\text{k}} r : k \quad P \vdash k \leq_{\text{k}} \mathsf{Region}}{P; E \vdash_{\text{type}} \mathtt{RHandle}\langle r \rangle}$$

[TYPE C]

$$\frac{\begin{array}{c} P = \ldots\ \mathit{def}\ \ldots \qquad \mathit{def} = \mathtt{class}\ \mathit{cn}\langle (k_i\ \mathit{fn}_i)_{i \in \{1..n\}} \rangle\ \ldots\ \mathtt{where}\ \mathit{constr}_{1..c}\ \ldots \\ \forall i \in \{1..n\},\ (E \vdash_{\text{k}} o_i : k'_i\ \wedge\ P \vdash k'_i \leq_{\text{k}} k_i\ \wedge\ E \vdash o_i \succeq o_1) \\ \forall i \in \{1..c\},\ E \vdash \mathit{constr}_i[o_1/\mathit{fn}_1]..[o_n/\mathit{fn}_n] \end{array}}{P; E \vdash_{\text{type}} \mathit{cn}\langle o_{1..n} \rangle}$$

B.6 Subtyping Rules: $P \vdash t_1 \leq t_2$

[SUBTYPE REFL]

$$\frac{}{P \vdash t \leq t}$$

[SUBTYPE TRANS]

$$\frac{\begin{array}{c} P \vdash t_1 \leq t_2 \\ P \vdash t_2 \leq t_3 \end{array}}{P \vdash t_1 \leq t_3}$$

[SUBTYPE CLASS]

$$\frac{\begin{array}{l} P \vdash_{\text{def}}\ \mathtt{class}\ \mathit{cn}\langle (k_i\ \mathit{fn}_i)_{i \in \{1..n\}} \rangle \\ \qquad\qquad \mathtt{extends}\ \mathit{cn}_2\langle \mathit{fn}_1\ o^* \rangle\ \ldots \end{array}}{P \vdash \mathit{cn}\langle o_{1..n} \rangle \leq \mathit{cn}_2\langle \mathit{fn}_1\ o^* \rangle[o_1/\mathit{fn}_1]..[o_n/\mathit{fn}_n]}$$

B.7 Well Formed Owner Kinds: $P; E \vdash_{\text{okind}} k$

[STANDARD OWNERS]

$$\frac{\begin{array}{c} k \in \{\mathsf{Owner}, \mathsf{ObjOwner}, \mathsf{Region}, \mathsf{GCRegion}, \\ \mathsf{NoGCRegion}, \mathsf{NHPRegion}, \mathsf{LocalRegion}, \mathsf{SharedRegion}\} \end{array}}{P; E \vdash_{\text{okind}} k}$$

[LT REGIONS]

$$\frac{P; E \vdash_{\text{okind}} \mathit{rkind}}{P; E \vdash_{\text{okind}} \mathit{rkind} : \mathsf{LT}}$$

[USER DECLARED SHARED REGION]

$$\frac{\begin{array}{c} P \vdash_{\text{srkdef}} \mathtt{regionKind}\ \mathit{srkn}\langle (k_i\ \mathit{fn}_i)_{i \in \{1..n\}} \rangle\ \ldots\ \mathtt{where}\ \mathit{constr}_{1..c}\ \ldots \\ \forall i \in \{1..n\},\ (E \vdash_{\text{k}} o_i : k'_i \quad P \vdash k'_i \leq_{\text{k}} k_i) \\ \forall i \in \{1..c\},\ E \vdash \mathit{constr}_i[o_1/\mathit{fn}_1]..[o_n/\mathit{fn}_n] \end{array}}{P; E \vdash_{\text{okind}} \mathit{srkn}\langle o_{1..n} \rangle}$$

B.8 Owner Subkinding Rules: $P \vdash k_1 \leq_{\text{k}} k_2$

[SUBKIND OWNER]

$$\frac{k \in \{\mathsf{ObjOwner}, \mathsf{Region}\}}{P \vdash k \leq_{\text{k}} \mathsf{Owner}}$$

[SUBKIND REGION]

$$\frac{k \in \{\mathsf{GCRegion}, \mathsf{NoGCRegion}\}}{P \vdash k \leq_{\text{k}} \mathsf{Region}}$$

[SUBKIND NOGCREGION]

$$\frac{k \in \{\mathsf{LocalRegion}, \mathsf{SharedRegion}\}}{P \vdash k \leq_{\text{k}} \mathsf{NoGCRegion}}$$

[SUBKIND NHPREGION]

$$\frac{}{P \vdash \mathsf{NHPRegion} \leq_{\text{k}} \mathsf{SharedRegion}}$$

[SUBKIND SHARED REGION KIND]

$$\frac{P \vdash_{\text{srkdef}} \mathtt{regionKind}\ \mathit{srkn}\langle (k_i\ \mathit{fn}_i)_{i \in \{1..n\}} \rangle\ \mathtt{extends}\ r\ \ldots}{P \vdash \mathit{srkn}\langle o_1..o_n \rangle \leq_{\text{k}} r[o_1/\mathit{fn}_1]..[o_n/\mathit{fn}_n]}$$

[DELETE LT]

$$\frac{}{P \vdash \mathit{rkind} : \mathsf{LT} \leq_{\text{k}} \mathit{rkind}}$$

[ADD LT]

$$\frac{P \vdash \mathit{rkind}_1 \leq_{\text{k}} \mathit{rkind}_2}{P \vdash \mathit{rkind}_1 : \mathsf{LT} \leq_{\text{k}} \mathit{rkind}_2 : \mathsf{LT}}$$

[SUBKIND REFL]

$$\frac{}{P \vdash k \leq_{\text{k}} k}$$

[SUBKIND TRANS]

$$\frac{P \vdash k_1 \leq_{\text{k}} k_2 \quad P \vdash k_2 \leq_{\text{k}} k_3}{P \vdash k_1 \leq_{\text{k}} k_3}$$

[SUBKIND VALUE]

$$\frac{}{P \vdash \mathsf{Value} \leq_{\text{k}} k}$$

B.9 Well Formed Methods: $P; E \vdash_{\text{meth}} \mathit{meth}$

[METHOD]

$$\frac{\begin{array}{c} E' = E,\ f_{1..n},\ \mathit{constr}_{1..c},\ (t_j\ p_j)_{j \in \{1..p\}},\ \textsf{Region initialRegion},\ \texttt{RHandle}\langle\textsf{initialRegion}\rangle\ h_{\mathit{fresh}} \\ P \vdash_{\text{env}} E' \quad P; E'; a_{1..q}; \textsf{initialRegion} \vdash e : t \end{array}}{P; E \vdash_{\text{meth}} t\ \ mn\langle f_{1..n}\rangle\texttt{(}(t_j\ \ p_j)_{j \in \{1..p\}}\texttt{)}\ \texttt{accesses}\ a_{1..q}\ \texttt{where}\ \mathit{constr}_{1..c}\ \{e\}}$$

B.10 Methods/Fields Defined in a Class: $P \vdash \mathit{field} \in c$ and $P \vdash \mathit{meth} \in c$

In the next two rules, let $\mathit{mbr} = \mathit{field} \mid \mathit{meth}$ be a class member (i.e., a field / method definition).

[DECLARED CLASS MEMBER]

$$\frac{P \vdash_{\text{def}} \texttt{class}\ cn\langle(k_i\ fn_i)_{i \in \{1..n\}}\rangle\ ...\ \{...\ \mathit{mbr}\ ...\ \}}{P \vdash \mathit{mbr} \in cn\langle fn_{1..n}\rangle}$$

[INHERITED CLASS MEMBER]

$$\frac{\begin{array}{c} P \vdash \mathit{mbr} \in cn\langle fn_{1..n}\rangle \\ P \vdash_{\text{def}} \texttt{class}\ \ cn_2\langle(k_i\ \ fn'_i)_{i \in \{1..m\}}\rangle\ \ \texttt{extends}\ \ cn\langle o_{1..n}\rangle ... \end{array}}{P \vdash \mathit{mbr}[o_1/fn_1]..[o_n/fn_n] \in cn_2\langle fn'_{1..m}\rangle}$$

B.11 Subregions/Fields Defined in a Region Kind: $P \vdash \mathit{field} \in \mathit{rkind}$ and $P \vdash \mathit{subsreg} \in \mathit{rkind}$

In the next two rules, let $\mathit{rmbr} = \mathit{field} \mid \mathit{subsreg}$ be a region member (i.e., a field / subregion definition).

[DECLARED REGION MEMBER]

$$\frac{P \vdash_{\text{srkdef}} \texttt{regionKind}\ \ \mathit{srkn}\langle(k_i\ \ fn_i)_{i \in \{1..n\}}\rangle\ \ ...\ \ \{\ \ ...\ \ \mathit{rmbr}\ \ ...\ \ \}}{P \vdash \mathit{rmbr} \in \mathit{srkn}\langle fn_{1..n}\rangle}$$

[INHERITED REGION MEMBER]

$$\frac{\begin{array}{c} P \vdash \mathit{rmbr} \in \mathit{srkn}\langle fn_{1..n}\rangle \\ P \vdash_{\text{srkdef}} \texttt{regionKind}\ \ \mathit{srkn}\langle(k_i\ \ fn'_i)_{i \in \{1..m\}}\rangle\ \ \texttt{extends}\ \ \mathit{srkn}\langle o_{1..n}\rangle\ \ ... \end{array}}{P \vdash \mathit{rmbr}[o_1/fn_1]..[o_n/fn_n] \in \mathit{srkn}_2\langle fn'_{1..m}\rangle}$$

B.12 Effect subsumption: $E \vdash X_1 \succeq_o X_2$

[$X_1 \succeq_o X_2$]

$$\frac{\begin{array}{c} X_1 = o_{1..n} \quad X_2 = g_{1..m} \\ \forall i \in \{1..n\},\ \exists j \in \{1..m\},\ E \vdash o_j \succeq_o g_i \end{array}}{E \vdash X_1 \succeq_o X_2}$$

We frequently use a special case of this rule, $E \vdash X \succeq_o o$, where the second effect, i.e., list of owners, consists of a single owner.

B.13 Provable Constraints: $E \vdash \mathit{constr}$

[ENV CONSTR]

$$\frac{E = E_1,\ \mathit{constr},\ E_2}{E \vdash \mathit{constr}}$$

[$\succeq_o$ world] $$\frac{}{E \vdash \mathsf{world} \succeq o}$$

[$\succeq_o$ OWNER] $$\frac{E = E_1,\ cn\langle o_{1..n}\rangle\ \mathsf{this},\ E_2}{E \vdash o_1 \succeq_o \mathsf{this}}$$

[$\succeq_o$ REFL] $$\frac{}{E \vdash o \succeq_o o}$$

[$\succeq_o$ TRANS] $$\frac{E \vdash o_1 \succeq_o o_2 \quad E \vdash o_2 \succeq_o o_3}{E \vdash o_1 \succeq_o o_3}$$

[$\succeq_o\ \rightarrow\ \succeq$] $$\frac{E \vdash o_1 \succeq_o o_2}{E \vdash o_1 \succeq o_2}$$

[$\succeq$ heap/immortal] $$\frac{o_1 \in \{\mathsf{heap}, \mathsf{immortal}\}}{E \vdash o_1 \succeq o_2}$$

[$\succeq$ REFL] $$\frac{}{E \vdash o \succeq o}$$

[$\succeq$ TRANS] $$\frac{E \vdash o_1 \succeq o_2 \quad E \vdash o_2 \succeq o_3}{E \vdash o_1 \succeq o_3}$$

B.14 Well Formed Owners: $E \vdash_{\mathrm{k}} o : k$

[OWNER THIS] $$\frac{E = E_1,\ cn\langle ...\rangle\ \mathsf{this},\ E_2}{E \vdash_{\mathrm{k}} \mathsf{this} : \mathsf{Owner}}$$

[OWNER FORMAL] $$\frac{E = E_1,\ k\ o,\ E_2}{E \vdash_{\mathrm{k}} o : k}$$

[OWNER HEAP] $$\frac{}{E \vdash_{\mathrm{k}} \mathsf{heap} : \mathsf{GCRegion}}$$

[OWNER IMMORTAL] $$\frac{}{E \vdash_{\mathrm{k}} \mathsf{immortal} : \mathsf{SharedRegion}}$$

B.15 Region Kind: $E \vdash_{\mathrm{o}} \mathrm{RKind}(o) = k$ and $E \vdash_{\mathrm{v}} \mathrm{RKind}(v) = k$

[RKIND THIS] $$\frac{E \vdash_{\mathrm{v}} \mathrm{RKind}(\mathsf{this}) = k}{E \vdash_{\mathrm{o}} \mathrm{RKind}(\mathsf{this}) = k}$$

[RKIND FN1] $$\frac{\begin{array}{c} E \vdash_{\mathrm{k}} o : k \\ k \notin \{\mathsf{Owner}, \mathsf{ObjOwner}\} \end{array}}{E \vdash_{\mathrm{o}} \mathrm{RKind}(o) = k}$$

[RKIND FN2] $$\frac{\begin{array}{c} E \vdash_{\mathrm{k}} o : k \\ k \in \{\mathsf{Owner}, \mathsf{ObjOwner}\} \\ E \vdash o_2 \succeq_o o \quad E \vdash_{\mathrm{o}} \mathrm{RKind}(o_2) = k_2 \end{array}}{E \vdash_{\mathrm{o}} \mathrm{RKind}(\mathit{fn}) = k_2}$$

[RKIND INT] $$\frac{E = E_1,\ \mathsf{int}\ v,\ E_2}{E \vdash_{\mathrm{v}} \mathrm{RKind}(v) = \mathsf{Value}}$$

[RKIND CLASS] $$\frac{\begin{array}{c} E = E_1,\ cn\langle o_{1..n}\rangle\ v,\ E_2 \\ E \vdash_{\mathrm{o}} \mathrm{RKind}(o_1) = k \end{array}}{E \vdash_{\mathrm{v}} \mathrm{RKind}(v) = k}$$

B.16 Available Region Handlers: $E \vdash_{\mathrm{av}} \mathsf{RH}(o)$

[AV HEAP] $$\frac{}{E \vdash_{\mathrm{av}} \mathsf{RH}(\mathsf{heap})}$$

[AV IMMORTAL] $$\frac{}{E \vdash_{\mathrm{av}} \mathsf{RH}(\mathsf{immortal})}$$

[AV HANDLE] $$\frac{E = E_1,\ \mathtt{RHandle}\langle r\rangle\ h,\ E_2}{E \vdash_{\mathrm{av}} \mathsf{RH}(r)}$$

[AV THIS] $$\frac{E = E_1,\ cn\langle o_{1..n}\rangle\ \mathsf{this},\ E_2}{E \vdash_{\mathrm{av}} \mathsf{RH}(\mathsf{this})}$$

[AV TRANS1] $$\frac{E \vdash o_1 \succeq_o o_2 \quad E \vdash_{\mathrm{av}} \mathsf{RH}(o_2)}{E \vdash_{\mathrm{av}} \mathsf{RH}(o_1)}$$

[AV TRANS2] $$\frac{E \vdash o_1 \succeq_o o_2 \quad E \vdash_{\mathrm{av}} \mathsf{RH}(o_1)}{E \vdash_{\mathrm{av}} \mathsf{RH}(o_2)}$$

B.17 Well Typed Expressions: $P; E; X; r_{cr} \vdash e : t$

[EXPR VAR] $$\frac{E = E_1,\ t\ v,\ E_2}{P; E; X; r_{cr} \vdash v : t}$$

[EXPR LET] $$\frac{\begin{array}{c} P; E; X; r_{cr} \vdash e_1 : t_1 \quad E' = E, t_1\ v \\ P \vdash_{\mathrm{env}} E' \quad P; E'; X; r_{cr} \vdash e_2 : t_2 \end{array}}{P; E; X; r_{cr} \vdash \mathtt{let}\ v\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 : t_2}$$

[EXPR NEW] $$\frac{\begin{array}{c} P; E \vdash_{\mathrm{type}} c \quad c = cn\langle o_{1..n}\rangle \\ E \vdash_{\mathrm{av}} \mathsf{RH}(o_1) \quad E \vdash X \succeq_o o_1 \end{array}}{P; E; X; r_{cr} \vdash \mathtt{new}\ c : c}$$

[EXPR H_HEAP]

$$\frac{}{P;E;X;r_{cr} \vdash \mathsf{h\_heap} : \mathtt{RHandle}\langle \mathsf{heap}\rangle}$$

[EXPR H_IMMORTAL]

$$\frac{}{P;E;X;r_{cr} \vdash \mathsf{h\_immortal} : \mathtt{RHandle}\langle \mathsf{immortal}\rangle}$$

[EXPR REF READ]

$$\frac{\begin{array}{c} P;E;X;r_{cr} \vdash v : cn\langle o_{1..n}\rangle \\ P \vdash (t\ fd) \in cn\langle fn_{1..n}\rangle \\ t' = t[o_1/fn_1]..[o_n/fn_n] \\ t' = cn\langle o'_{1..m}\rangle \to E \vdash X \succeq_o o'_1 \end{array}}{P;E;X;r_{cr} \vdash v.fd : t'}$$

[EXPR REF WRITE]

$$\frac{\begin{array}{c} P;E;X;r_{cr} \vdash v_1 : cn_1\langle o_{1..n}\rangle \\ P;E;X;r_{cr} \vdash v_2 : t_2 \\ P \vdash (t\ fd) \in cn_1\langle fn_{1..n}\rangle \\ t' = t[o_1/fn_1]..[o_n/fn_n] \quad P \vdash t_2 \leq t' \\ t' = cn'\langle g_{1..m}\rangle \to E \vdash X \succeq_o g_1 \end{array}}{P;E;X;r_{cr} \vdash v_1.fd = v_2 : t'}$$

[EXPR INVOKE]

$$\frac{\begin{array}{c} P;E;X;r_{cr} \vdash v : cn\langle o_{1..n}\rangle \quad \forall i \in \{(n+1)..m\},\ E \vdash o_i \succeq o_1 \\ P \vdash t\ \ mn\langle (k_i\ \ fn_i)_{i\in\{(n+1)..m\}}\rangle \mathtt{(}(t_j\ \ p_j)_{j\in\{1..k\}}\mathtt{)}\ \mathtt{accesses}\ X_m\ \mathtt{where}\ constr_{1..c}\ \ \{e\} \in cn\langle fn_{1..n}\rangle \\ Rename(\alpha) \stackrel{def}{=} \alpha[o_1/fn_1]..[o_m/fn_m][r_{cr}/\mathsf{initialRegion}] \\ \forall i \in \{1..k\},\ (P;E;X;r_{cr} \vdash v_i : t'_i\ \wedge\ P \vdash t'_i \leq Rename(t_i)) \\ \forall i \in \{(n+1)..m\},\ (E \vdash_{\mathrm{k}} o_i : k'_i\ \wedge\ P \vdash k'_i \leq_{\mathrm{k}} Rename(k_i)) \\ E \vdash X \succeq_o Rename(X_m) \quad \forall i \in \{1..c\},\ E \vdash Rename(constr_i) \end{array}}{P;E;X;r_{cr} \vdash v\mathtt{.}mn\langle o_{(n+1)..m}\rangle \mathtt{(}v_{1..k}\mathtt{)} : Rename(t)}$$

[EXPR REGION]

$$\frac{\begin{array}{c} P;E \vdash_{\mathrm{okind}} rkind \quad rkind = srkn\langle\rangle \quad P \vdash rkind \leq_{\mathrm{k}} \mathsf{Region} \\ k_r = \left\{\begin{array}{ll} rkind\!:\!\mathsf{LT} & \text{if } rpol = \mathsf{LT}(size) \\ rkind & \text{otherwise} \end{array}\right. \quad E_2 = E,\ k_r\ r,\ \mathtt{RHandle}\langle r\rangle\ h \\ E_3 = \left\{\begin{array}{ll} E_2,\ (r_e \succeq r)_{\forall r_e \in Regions(E)} & \text{if } P \vdash rkind \leq_{\mathrm{k}} \mathsf{LocalRegion} \\ E_2 & \text{otherwise} \end{array}\right. \quad P \vdash_{\mathrm{env}} E_3 \\ X_2 = \left\{\begin{array}{l} \{o \in X \mid E \vdash_{\mathrm{o}} \mathrm{RKind}(o) = k_i\ \wedge\ P \vdash k_i \leq_{\mathrm{k}} \mathsf{NoGCRegion}\} \\ \quad \text{if } P \vdash rkind \leq_{\mathrm{k}} \mathsf{NHPRegion} \\ X \quad \text{otherwise} \end{array}\right. \\ P;E_3;X_2,r;r \vdash e : t \quad E \vdash X \succeq_o \mathsf{heap} \\ P \vdash rkind \leq_{\mathrm{k}} \mathsf{NHPRegion}\ \to\ \forall v \in FreeVars(e),\ (E \vdash_{\mathrm{v}} \mathrm{RKind}(v) = k\ \wedge\ P \vdash k \leq_{\mathrm{k}} \mathsf{NoGCRegion}) \end{array}}{P;E;X;r_{cr} \vdash \mathtt{(RHandle}\langle rkind\!:\!rpol\ \ r\rangle\ \ h\mathtt{)}\ \ \{e\} : \mathsf{int}}$$

where $FreeVars(e)$ is the set of free variables from e and $Regions(E)$ is the set of all regions mentioned in E:

$$\begin{array}{lcl} Regions(\emptyset) & = & \emptyset \\ Regions(E, rkind\ \ r) & = & Regions(E) \cup \{r\} \\ Regions(E, \_) & = & Regions(E), \text{ otherwise} \end{array}$$

[EXPR SUBREGION]

$$\frac{\begin{array}{c} P;E;X;r_{cr} \vdash h_2 : \mathtt{RHandle}\langle r_2\rangle \quad E \vdash_{\mathrm{k}} r_2 : srkn_2\langle o_{1..n}\rangle \\ P \vdash rkind_3\!:\!rpol\ \ rsub \in srkn_2\langle fn_{1..n}\rangle \quad rkind = rkind_3[o_1/fn_1]..[o_n/fn_n][r_2/\mathsf{this}] \\ k_r = \left\{\begin{array}{ll} rkind\!:\!\mathsf{LT} & \text{if } rpol = \mathsf{LT}(size) \\ rkind & \text{otherwise} \end{array}\right. \quad E_2 = E,\ k_r\ r,\ \mathtt{RHandle}\langle r\rangle\ h,\ r_2 \succeq r \quad P \vdash_{\mathrm{env}} E_2 \\ X_2 = \left\{\begin{array}{l} \{o \in X \mid E \vdash_{\mathrm{o}} \mathrm{RKind}(o) = k_i\ \wedge\ P \vdash k_i \leq_{\mathrm{k}} \mathsf{NoGCRegion}\} \\ \quad \text{if } P \vdash rkind \leq_{\mathrm{k}} \mathsf{NHPRegion} \\ X \quad \text{otherwise} \end{array}\right. \\ P;E_2;X_2,r;r \vdash e : t \\ \mathtt{new}\ \vee\ \neg(P \vdash rkind \leq_{\mathrm{k}} \mathsf{NHPRegion})\ \vee\ (rpol = VT) \to E \vdash X \succeq_o \mathsf{heap} \\ P \vdash rkind \leq_{\mathrm{k}} \mathsf{NHPRegion}\ \to\ \forall v \in FreeVars(e),\ (E \vdash_{\mathrm{v}} \mathrm{RKind}(v) = k\ \wedge\ P \vdash k \leq_{\mathrm{k}} \mathsf{NoGCRegion}) \end{array}}{P;E;X;r_{cr} \vdash \mathtt{(RHandle}\langle r\rangle\ \ h_1\ \mathtt{=}\ [\mathtt{new}]_{\mathrm{opt}}\ \ h_2\mathtt{.}rsub\mathtt{)}\ \ \{e\} : \mathsf{int}}$$

[EXPR LOCALREGION]

$$\frac{P; E; X; r_{cr} \vdash \texttt{(RHandle}\langle \textsf{LocalRegion}:\textsf{VT}\ \ r\rangle\ \ h\texttt{)}\ \ \{e\} : \textsf{int}}{P; E; X; r_{cr} \vdash \texttt{(RHandle}\langle r\rangle\ \ h\texttt{)}\ \ \{e\} : \textsf{int}}$$

[EXPR FORK]

$$\frac{\begin{array}{c} P; E; X; r_{cr} \vdash v_0\,\texttt{.}\,mn\langle o_{1..n}\rangle\texttt{(}v_{1..k}\texttt{)} : t \\ \mathit{NonLocal}(k) \stackrel{def}{=} (P \vdash k \leq_{\mathrm{k}} \textsf{SharedRegion}) \ \vee\ (P \vdash k \leq_{\mathrm{k}} \textsf{GCRegion}) \\ \forall i \in \{0..k\},\ (E \vdash_{\mathrm{v}} \mathrm{RKind}(v_i) = k_i \ \wedge\ \mathit{NonLocal}(k_i)) \\ \forall i \in \{1..n\},\ (E \vdash_{\mathrm{o}} \mathrm{RKind}(o_i) = k'_i \ \wedge\ \mathit{NonLocal}(k'_i)) \\ E \vdash_{\mathrm{o}} \mathrm{RKind}(r_{cr}) = k_{cr} \quad \mathit{NonLocal}(k_{cr}) \end{array}}{P; E; X; r_{cr} \vdash \texttt{fork}\ \ v_0\,\texttt{.}\,mn\langle o_{1..n}\rangle\texttt{(}v_{1..k}\texttt{)} : \textsf{int}}$$

[EXPR NOGCFORK]

$$\frac{\begin{array}{c} \forall i \in \{0..m\},\ (E \vdash_{\mathrm{v}} \mathrm{RKind}(v_i) = k_i \ \wedge\ P \vdash k_i \leq_{\mathrm{k}} \textsf{SharedRegion}:\textsf{LT}) \\ \forall i \in \{1..n\},\ (E \vdash_{\mathrm{o}} \mathrm{RKind}(o_i) = k'_i \ \wedge\ P \vdash k'_i \leq_{\mathrm{k}} \textsf{SharedRegion}:\textsf{LT}) \\ E \vdash_{\mathrm{o}} \mathrm{RKind}(r_{cr}) = k_{cr} \quad P \vdash k_{cr} \leq_{\mathrm{k}} \textsf{SharedRegion}:\textsf{LT} \\ X' = \{o \in X \mid E \vdash_{\mathrm{o}} \mathrm{RKind}(o) = k_i \ \wedge\ P \vdash k_i \leq_{\mathrm{k}} \textsf{SharedRegion}\} \\ P; E; X'; r_{cr} \vdash v_0\,\texttt{.}\,mn\langle o_{1..n}\rangle\texttt{(}v_{1..m}\texttt{)} : t \end{array}}{P; E; X; r_{cr} \vdash \texttt{NoGC\_fork}\ \ v_0\,\texttt{.}\,mn\langle o_{1..n}\rangle\texttt{(}v_{1..m}\texttt{)} : \textsf{int}}$$

[EXPR GET REGION FIELD]

$$\frac{\begin{array}{c} P; E; X; r_{cr} \vdash h : \texttt{RHandle}\langle r\rangle \quad E \vdash_{\mathrm{k}} r : \mathit{srkn}\langle o_{1..n}\rangle \\ P \vdash t\ \mathit{fd} \in \mathit{srkn}\langle \mathit{fn}_{1..n}\rangle \quad t' = t[o_1/\mathit{fn}_1]..[o_n/\mathit{fn}_n][r/\textsf{this}] \\ E \vdash X \succeq_o r \quad ((t' = \textsf{int}) \ \vee\ (t' = \mathit{cn}\langle o'_{1..m}\rangle \ \wedge\ E \vdash X \succeq_o o'_1)) \end{array}}{P; E; X; r_{cr} \vdash h\,\texttt{.}\,\mathit{fd} : t'}$$

[EXPR SET REGION FIELD]

$$\frac{\begin{array}{c} P; E; X; r_{cr} \vdash h : \texttt{RHandle}\langle r\rangle \quad E \vdash_{\mathrm{k}} r : \mathit{srkn}\langle o_{1..n}\rangle \\ P \vdash t\ \mathit{fd} \in \mathit{srkn}\langle \mathit{fn}_{1..n}\rangle \quad t_1 = t[o_1/\mathit{fn}_1]..[o_n/\mathit{fn}_n][r/\textsf{this}] \\ P; E; X; r_{cr} \vdash v : t' \quad P \vdash t' \leq t_1 \quad E \vdash X \succeq_o r \end{array}}{P; E; X; r_{cr} \vdash h\,\texttt{.}\,\mathit{fd}\ \texttt{=}\ v : t_1}$$

B.18 Correct Method Overriding: $\mathit{InheritanceOK}(P)$

[INHERITANCEOK PROG]

$$\frac{\begin{array}{c} P = \mathit{def}_{1..n}\ \ \mathit{srkdef}_{1..r}\ \ e \quad \forall i \in \{1..n\},\ P \vdash \mathit{InheritanceOK}(\mathit{def}_i) \\ \forall i \in \{1..r\},\ P \vdash \mathit{InheritanceOK}(\mathit{srkdef}_i) \end{array}}{\mathit{InheritanceOK}(P)}$$

[INHERITANCEOK CLASS]

$$\frac{\begin{array}{l} \mathit{def} = \texttt{class}\ \ \mathit{cn}\langle(k_i\ \ \mathit{fn}_i)_{i\in\{1..n\}}\rangle\ \ \texttt{extends}\ \ \mathit{cn}\langle o_{1..m}\rangle\ \ \texttt{where}\ \ \mathit{constr}_{1..q}\ \ \ldots \\ \mathit{def}' = \texttt{class}\ \ \mathit{cn}'\langle(k'_i\ \ \mathit{fn}'_i)_{i\in\{1..m\}}\rangle\ \ \texttt{extends}\ \ c\ \ \texttt{where}\ \ \mathit{constr}'_{1..u}\ \ \ldots \\ \qquad\qquad P \vdash_{\mathrm{def}} \mathit{def}' \quad \mathit{constr}_{1..u}[o_1/\mathit{fn}'_1]..[o_m/\mathit{fn}'_m] \subseteq \mathit{constr}'_{1..q} \\ \quad \forall mn, \quad (\ \ P \vdash \mathit{meth} \in \mathit{def} \quad \wedge \quad \mathit{meth} = t_r\ \ mn\langle\ldots\rangle\texttt{(}\ldots\texttt{)}\ \ \ldots\ \ \wedge \\ \qquad\qquad\qquad\ \ P \vdash \mathit{meth}' \in \mathit{def}' \quad \wedge \quad \mathit{meth} = t'_r\ \ mn\langle\ldots\rangle\texttt{(}\ldots\texttt{)}\ \ \ldots\ \ ) \\ \qquad\qquad \rightarrow \quad P \vdash \mathit{OverridesOK}(\mathit{meth}, \mathit{meth}') \end{array}}{P \vdash \mathit{InheritanceOK}(\mathit{def})}$$

[OVERRIDESOK METHOD]

$$\frac{\begin{array}{c} meth = t_r \;\; mn\langle f_{1..n}\rangle\texttt{(}(t_i \;\; p_i)_{i\in\{1..m\}}\texttt{)} \;\texttt{accesses}\; a_{1..q} \;\texttt{where}\; constr_{1..r} \;\ldots \\ meth' = t'_r \;\; mn\langle f_{1..n}\rangle\texttt{(}(t'_i \;\; p'_i)_{i\in\{1..m\}}\texttt{)} \;\texttt{accesses}\; a'_{1..s} \;\texttt{where}\; constr'_{1..u} \;\ldots \\ P \vdash t'_r \leq t_r \quad \forall i \in \{1..m\},\; P \vdash t_i \leq t'_i \\ a'_{1..q} \subseteq a_{1..s} \quad constr'_{1..r} \subseteq constr_{1..u} \end{array}}{P \vdash \mathit{OverridesOK}(meth, meth')}$$

[INHERITANCEOK REGION KIND]

$$\frac{\begin{array}{c} srkdef = \texttt{regionKind}\; srkn\langle f_{1..n}\rangle \;\texttt{extends}\; srkn'\langle o_{1..m}\rangle \;\texttt{where}\; constr_{1..q} \;\ldots \\ srkdef' = \texttt{regionKind}\; srkn'\langle (k'_i \;\; fn'_i)_{i\in\{1..m\}}\rangle \;\texttt{extends}\; srkind \;\texttt{where}\; constr'_{1..s} \;\ldots \\ P \vdash_{\text{srkdef}} srkdef' \quad constr_{1..s}[o_1/fn'_1]..[o_m/fn'_m] \subseteq constr'_{1..q} \end{array}}{P \vdash \mathit{InheritanceOK}(srkdef)}$$

C. TRANSLATION TO RTSJ

In this section, we present details on how we translate programs written in our system to RTSJ programs using local translation rules. The following subsections show the translation for expressions that create or enter regions, access region fields, allocate objects, or start threads. The translation for the other language constructs is trivial—we do a simple type-erasure based translation. In this section, we use **bold font** for the generated code and *italic font* (default font for mathematical notation in LaTeX) for the expressions that are evaluated at translation time.

C.1 Region Representation

Similar to our system, RTSJ allows the programmers to create memory regions ("memory areas" in RTSJ terminology). It also has two special memory areas: a garbage-collected heap and an immortal memory area. A memory area is a normal Java objects that also point to a piece of memory for the objects allocated in that memory area. Figure 14 presents the RTSJ hierarchy of classes for memory management. The root of this hierarchy, `MemoryArea`, is subclassed by `HeapMemory`, `ImmortalMemory` and `ScopedMemory`. `HeapMemory` and `ImmortalMemory` represent the heap, respectively the immortal memory area; there exists only one instance of each of them. `ScopedMemory` is the class for normal regions; it has two subclasses: `LTMemory` (for LT regions) and `VTMemory` (for VT regions). The RTSJ regions are maintained similarly to our shared regions: each thread has a stack of regions it can currently access, and each region has counter of how many threads can access it.

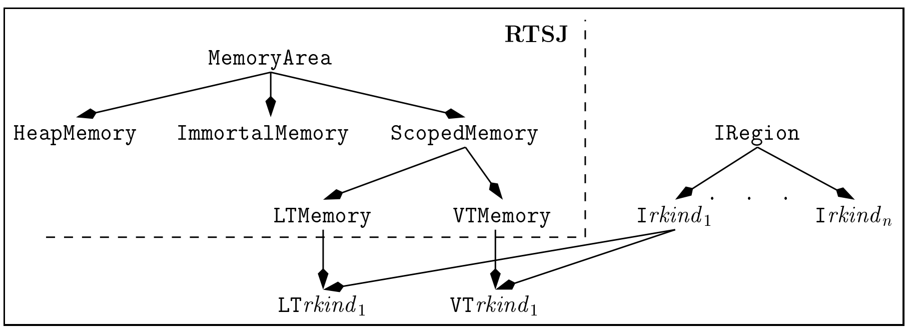

Figure 14: RTSJ hierarchy of memory management related classes and our extensions to it.

We translate each region from our system into a memory area and some additional objects. Figure 15 presents the translation into RTSJ of a region r with three fields and two subregions. We represent r as an RTSJ memory area `m` plus two auxiliary objects `w1` and `w2`. `m` points to a piece of memory that is pre-allocated for an LT region, or grown on-demand for a VT region. `m` also points to an object `w1` whose fields point to the representation of r's subregions (we subclass `LT/VTMemory` to add an extra field); these subregions are represented in the same way as r. (Note that we only allow a region to have a bounded number of subregions.) In addition, `m`'s portal points to an object `w2` that serves as a wrapper for r's fields. `w2` is allocated in the memory space attached to `m`, while `m`, `w1`, and all the similar objects for the

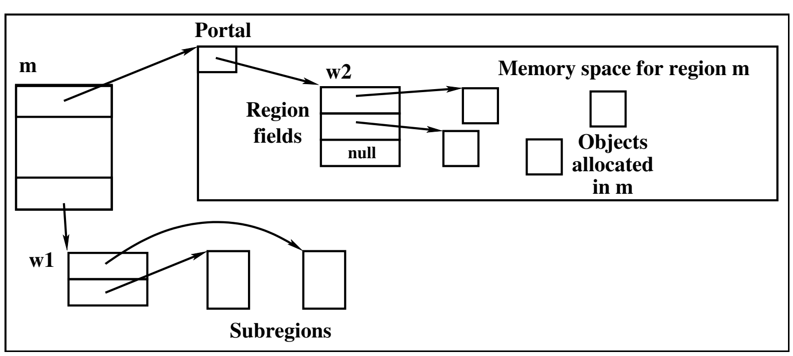

Figure 15: Translation of a region with three fields and two subregions

```
public interface Irkind extends IRegion {
  rkindSubs getSubs();
  rkindFields getFields();
}

public class rkindFields {
  ∀ field (t fd) ∈ rkind
    public t fd;
}

public class rkindSubs {
  ∀ subregion (srkind_s : rpol_s rsub) ∈ rkind
    public Isrkind_s rsub;
}
```

Figure 16: Declaration of I*rkind* for representing regions of kind *rkind*

```
public interface IRegion {
  boolean isFlushed();
  void    setIsFlushed(boolean value);

  void    tryToFlush();

  boolean isASubregion();
  void    setIsASubregion(boolean value);

  AtomicInteger getNInside();
  AtomicInteger getNExiting();
}
```

Figure 17: Declaration of IRegion, the common super type of I*rkind*s

representation of r's transitive subregions are allocated in the current region at the time `m` was created.

There are several differences between RTSJ memory areas and our regions. The following list presents them, together with our translation for each case; Figure 16 presents the classes and interfaces we introduce along the way.

1. To ensure proper nesting of memory area enter/exit operations, RTSJ uses the following mechanism for executing code inside a memory area: the program calls the `enter` method of the memory area object and passes it a `Runnable` object; the `run()` method of this object is executed inside the memory area. To translate an expression of the form "`(RHandle`$\langle rkind : rpol \;\; r \rangle \;\; h$`)` $\{e\}$" into this pattern, we have to create `Runnable` objects.
2. RTSJ does not have thread-local memory areas. Therefore, we have to translate the local regions into memory areas that are maintained through thread counting; even if we know that only one thread uses them. This is less efficient than a genuine implementation of local regions but is still correct.
3. An RTSJ memory area has one portal field, similar to our region fields, except that it is untyped (i.e., has type `Object`). To allow multiple and typed region fields, for each region kind *rkind* from our system, we introduce a *field wrapper* class *rkind*`Fields` (Figure 16) that contains a field for each original field. For each region of kind *rkind*, the portal of the corresponding memory area points to an object of class *rkind*`Fields` allocated in that memory area.
4. RTSJ does not have anything equivalent to our subregions. To simulate them, for each region kind *rkind*, we introduce a *subregion wrapper* class *rkind*`Subs` (Figure 16) that has one field for each

```
import javax.realtime.*;
import java.util.concurrent.atomic.*;

public class LTrkind extends LTMemory implements Irkind {
  private boolean isFlushed;
  public  boolean isFlushed() { return isFlushed; }
  public  boolean setIsFlushed(boolean value) { isFlushed = value; }
  private boolean isASubregion;
  public  boolean isASubregion() { return isASubregion; }
  private rkindSubs subs;
  public  rkindSubs   getSubs()   { return  subs; }
  public  rkindFields getFields() { return (rkindFields) getPortal(); }
  private AtomicInteger n_inside  = new AtomicInteger(0);   // number of threads inside
  public  AtomicInteger getNInside()   return n_inside;
  public  AtomicInteger n_exiting = new AtomicInteger(0);   // number of threads exiting
  public  AtomicInteger getNExiting()  return n_exiting;    // See JSR 166

  public LTrkind(boolean isASubregion, int size) {
    super(size);
    this.flushed = true;  this.isASubregion = isASubregion;
    this.subs = new rkindSubs();
    ∀  subregion (srkind_s:rpol_s rsub) ∈ rkind, generate
      subs.rsub =
        if rpol_s = LT(size)  new LTsrkind_s(true, size);
        else new VTsrkind_s(true);
  }

  public void tryToFlush() {
    if(canFlush()) {
      setIsFlushed(true);  setPortal(null);
      rkindSubs subs = getSubs();
      if(!isASubregion()) {
        ∀ subregion (rkind_s:rpol_s rsub) ∈ rkind, generate
          IRegion sr = subs.rsub;
          if(!sr.isFlushed()) sr.tryToFlush();
  }}}

  private boolean canFlush() {
    if(getReferenceCount() != 1) return false;
    if(!isASubregion()) return true;
    rkindFields fields = this.getFields();
    if(fields != null) {
      ∀ field (t fd) ∈ rkind, generate
        if(fields.fd != null) return false;
    }
    rkindSubs subs = this.getSubs();
    ∀ subregion (srkind_s:rpol_s rsub)  ∈ rkind, generate
      if(!subs.rsub.isFlushed()) return false;
    return true;
}}
```

Figure 18: Declaration of class LT*rkind*. LT*rkind* represents regions of kind *rkind*, with allocation policy LT. The declaration of VT*rkind*, the VT version, is almost identical, except that it subclasses VTMemory and its constructor does not take any size parameter.

subregion. Subregions are represented similar to their parent region. When we create a region, we automatically create all its transitive subregions. All the memory area objects and the field wrappers are allocated in the current region at the creation time. Each memory area that represents a region of kind *rkind* implements the interface `I`*rkind* (Figure 16); this interface has special methods for retrieving the field wrapper object and the subregion wrapper. In addition `I`*rkind* extends the interface `IRegion` (Figure 17) that has some region maintenance methods.

5. Our RTSJ platform flushes a memory area when its counter changes from one to zero and its portal is null. While designing our language, our goal was as follows. We wanted to provide semantics where flushing or deleting of regions is transparent to programs (so one cannot detect under program control if a region has been flushed or deleted). However, we also wanted to reclaim memory space as soon as possible. We therefore came up with the following rules for flushing a region: we flush a subregion if the counter is zero, all fields are null and all subregions have been flushed; we flush a region as soon as its counter is zero (because the region won't be used afterward). To enforce our rules, at each place where we might exit a region we call its method `tryToFlush()`. This method acts as follows: if the counter is about to become zero and our other conditions are satisfied, we set the portal field to null such that RTSJ flushes that region; otherwise, we let it be non-null to prevent the region from being flushed.

In RTSJ, the programmer chooses the desired allocation policy by allocating a `LTMemory` or a `VTMemory` object. Hence, for each region kind *rkind*, we define two implementations of `I`*rkind*: a class `LT`*rkind* that extends `LTMemory`, and a class `VT`*rkind* that extends `VTMemory`. This creates the class hierarchy from Figure 14. Figure 18 presents the declaration of `LT`*rkind*. `VT`*rkind* is almost identical, except that it inherits from `VTMemory`, and its constructor does not take any size argument. As Java does not have multiple class inheritance, there is significant code duplication between `LT`*rkind* and `VT`*rkind*. This can be improved by factoring out most of the code as static methods in one of the two classes.

In our system, we have both region names and region handles. The region names are for typechecking purposes only that are removed by the type erasure. The region handles are runtime values; a region handle that had the type `RHandle`$\langle r \rangle$ in our system is translated into a reference to an object `I`*rkind*, where *rkind* is the kind of the region r.

C.2 Creating a Default Local Region

Instead of presenting directly the translation for general region creation expression of the form "`(RHandle`$\langle rkind : rpol \;\; r \rangle$ h`)` $\{e\}$", we first look at a simplified case. An expression of the form "`(RHandle`$\langle r \rangle$ h`)` $\{e\}$" creates a local region (i.e., a region of kind LocalRegion) using the default allocation policy VT. Figure 19 presents the translation for "`(RHandle`$\langle r \rangle$ h`)` $\{e\}$". The code from Figure 19 works as follows:

1. Create region h `= new VTLocalRegion(false)`
2. Declare a class RE that implements the `Runnable` interface. Its `run()` method serve as a wrapper for the expression e. We introduce several fields in RE for dealing with the free variables of e. For each such variable $v \neq$ this, RE has a field v of appropriate type. If this appears in e, we create a field with a fresh name _this (this refers to another object in the methods of RE). The body of the `run()` method consists of e, with each free occurrence of this substituted by _this.
3. We create an instance re of RE in the current region and initialize its fields with the free variables of e.
4. Execute e: h`.enter(`re`);`
5. Retrieve the (possibly changed) values of the free variables of e from re's fields.

C.3 Creating a Shared Region

The translation for "`(RHandle`$\langle rkind : rpol \;\; r \rangle$ h`)` $\{e\}$" is very similar to that for a local region, with the following differences:

1. We replace line 1 with the following code:

```
{
   // create a new RTSJ region
1: ILocalRegion h = new VTLocalRegion(false);

   // create a Runnable object to wrap the code of e
   class RE implements Runnable {
     // one field for h and for each free variable of e
2:   ILocalRegion h;
     ∀ v ∈ FreeVars(e) \ {this}; let t be its type in the environment
       public t v;
     if this ∈ FreeVars(e); let t be its type in the environment
       public t _this;

     public void run() {
3:     translation of e[_this/this]
     }
   }
   RE re = new RE();

   // store h and all free variables of e in re's fields
   re.h = h;
   ∀ v ∈ FreeVars(e) \ {h, this}
     re.v = v;
   if this ∈ FreeVars(e)
     re._this = this;

   // evaluate e
   ((MemoryArea) h).enter(re);

   // restore the values of e's free variables
   ∀ v ∈ FreeVars(e) \ {h, this}
     v = re.v;
}
```

where RE, re, and _this are fresh identifiers.

Figure 19: Translation for "(RHandle⟨r⟩ h) $\{e\}$"

```
1': Irkind h =
      if(rpol = LT(size)) new LTrkind(true, size);
      else new VTrkind(true);
```

Accordingly, field h of the class RE (line 2) has now type `I`*rkind*.

2. The body of the `run()` method of class RE (line 3) is replaced with

```
public void run() {

  int x = h.getNInside().add(1);
  while (h.getNExiting().get() > 0) sleep(0,1000);   // Wait until exiting threads finish exiting
  if (x == 1) {
    h.setIsFlushed(false);
    if (h.getPortal() == null) h.setPortal(new rkindFields());
  } else {
    while (h.getPortal() == null) sleep(0,1000);
  }


  try {

    translation of e[id/this]

  } finally {
    h.getNExiting().add(1); // Begin exiting the region
    int x = h.getNInside().add(-1);
    if (x == 0) h.tryToFlush();
    h.getNExiting().add(-1); // Finish exiting the region
  }
}
```

At the beginning of the `run()` method, if this is the first thread entering the region, we mark the region h as unflushed and make sure it has a non-null portal. By using the `try-finally` block, we ensure that no matter how e terminates, we execute some code right before `run()` is exited and the counter of memory area h is decremented. The method h`.tryToFlush()` enforces our region flushing policy.

In the above code, both the beginning and the end of `run()` access the `isFlushed` field from the region object (indirectly via `tryToFlush`) and its portal. We avoid race conditions with a tricky synchronization mechanism that uses atomic operations defined in JSR 166 [27]. Our synchronization ensures safety: when a thread is using a region, no other thread can flush the same region; and when a thread is using a region, the region has a non-null portal. Our synchronization also ensures that the last thread exiting a region flushes the region if the conditions for flushing are met.

Note that the above code has a priority inversion problem. In the above code, a thread entering a region waits if there are threads exiting (and perhaps flushing) the region. The process of exiting the region only takes a bounded amount of time, so normally, the wait should be for a bounded amount of time. However, when a regular thread is exiting the region, it might be suspended for an unbounded amount of time by the garbage collector. If a real-time thread then wants to enter the region, it might have to wait for an unbounded amount of time for the regular thread to finish exiting the region.

The above priority inversion problem occurs even in the Real-Time Specification for Java, so we do no worse than the RTSJ. However, it is possible to modify our type system slightly to avoid the priority inversion problem. We describe the modifications in Section C.9.

C.4 Entering a Subregion

The translation for "(RHandle⟨r⟩ h = h_2.$rsub$) {e}" is very similar to the one from Section C.3. The only difference is that now, instead of creating a region, we simply read one and use it. The beginning of the translation (line 1') becomes:

```
1'': Isrkind h = h2.getSubs().h;
```

where *rkind* is the kind of the subregion *rsub*.

C.5 Creating a Subregion

The translation for "(RHandle⟨r⟩ h = new h_2.$rsub$) {e}" is very similar to the one from Section C.3. Only the beginning of the translation changes as follows:

```
1''': class RE implements Runnable {
        VTMemory h;
        VTMemory h2;
        public void run() {
          h =
            if rpol_s = LT(size) new LTrkind_s(true, size);
            else new VTrkind_s(true);
          h2.subs.rsub = h;
        }
      }
      MemoryArea ma = MemoryArea.getMemoryArea(h2);
      h2.subs.rsub.setIsASubregion(false);
      RE re = new RE();
      re.h = h; re.h2 = h2;
      ma.enter(re);
      h = re.h; h2 = re.h2;
```

where $rkind_s$ is the kind of the subregion *rsub*, $rpol_s$ is its allocation policy, and *RE*, *re*, and *ma* are fresh identifiers. The above code works as follows:

1. To be consistent with our representation of regions (see Section C.1), we allocate the memory area objects for the new subregion (and its subregions) in the same memory area where the previous subregion was allocated. Most of the above code deals with technical details related to this operation.
2. The previous subregion is "detached" from its parent: "h_2.subs.$rsub$.setIsASubregion(false)" to record the fact that it cannot be entered from its parent. The first time its counter becomes zero, it will be flushed (and subsequently deleted along with its subregions).

C.6 Manipulating Region Fields

We translate "$h.fd$" as follows:

```
((rkindFields) h.getPortal()).fd
```

where *rkind* is the kind of the region r that h is a handle of, i.e., in the type environment, h has type RHandle⟨r⟩. The translation for "$h.fd = v$" is similar:

```
((rkindFields) h.getPortal()).fd = v
```

C.7 Allocating an Object

There are no constructors in the language we presented so far. However, they are trivial to add: an expression of the form "`new` $cn\langle o_{1..n}\rangle$`(`$e_{1..m}$`)`" desugars into a "`new` $cn\langle o_{1..n}\rangle$" followed by a call to the appropriate constructor. We translate "`new` $cn\langle o_{1..n}\rangle$`(`$e_{1..m}$`)`" as follows:

1. First, we generate Java code to retrieve the memory area where the new object is allocated. The type rule for `new` already checked that $E \vdash_{\text{av}} \mathsf{RH}(o_1)$, i.e., a handle for this region is available at runtime, even after type-erasure (see the rules from Section B.16). We use the typechecker judgments to retrieve that region. Notice that each of the rules that prove a statement of the form $E \vdash_{\text{av}} \mathsf{RH}(o)$ has at most one such statement among its preconditions. Therefore, if we consider the part of the proof tree for $E \vdash_{\text{av}} \mathsf{RH}(o_1)$ that corresponds only to this kind of rules, we obtain a chain. We can retrieve the relevant memory area in all cases:

[AV HANDLE]	The typing environment contains a handle h of type `RHandle`$\langle r\rangle$ for the relevant region r. In the translated code h is a local variable that points to the region we allocate in; we directly use `(MemoryArea) h`.
[AV HEAP]	Allocation in heap; call `HeapMemory.instance()`.
[AV IMMORTAL]	Allocation in immortal; call `ImmortalMemory.instance()`.
[AV THIS]	Allocation in the region this is allocated in; call `MemoryArea.getMemoryArea(this)`

2. Next, we generate a call to `newInstance`, to allocate a new object in the memory area that the code generated at 1 evaluates to. We also recursively translate $e_1, \ldots, e_m$ (the arguments of the constructor).

Optimization: `newInstance` allows us to allocate an object in any region. However, it currently uses reflection, e.g., for passing the class of the allocated object. Hence, it is less efficient than `new`, that allocates only in the current region. The typechecker knows the current region r_{cr} for the `new` expression that we translate. For [AV HEAP], [AV IMMORTAL] and [AV HANDLE], if r_{cr} is identical to the region we allocate in, we use `new`. We can apply this optimization even in the case of [AV THIS], if the typechecker can prove that $r_{cr} \succeq_o \mathsf{this}$.

C.8 Forking a Thread

Figure 20 presents the translation for an expression of the form "`fork` v_0`.`$mn\langle o_{1..n}\rangle$`(`$v_{1..m}$`)`". The resulting code works as follows:

1. In Java, threads are objects whose class is a subclass of `java.lang.Thread`. Programmers create thread objects using `new` and start them by invoking their `start()` method. `start()` starts a thread whose body is the `run()` method of the thread object. In RTSJ, threads that want to use regions have to subclass `javax.realtime.RealtimeThread`, which itself subclasses `java.lang.Thread`. Accordingly, we define a class T for our thread. T has one field `v`i to store the value of each variable v_i.
2. The `run()` method of T invokes mn with the right receiver and parameters. When the thread terminates, each region that is still on its stack of regions is exited. Therefore, for each such region, if our conditions for flushing it are fulfilled, we need to make sure that it meets the conditions for being flushed by RTSJ. Fortunately, RTSJ offers methods that allow us to examine the stack of memory areas associated with a thread.
3. We create a instance of class T, generate code to store the result of each variable v_i in the appropriate field and next start the thread.

The only difference in the translation for "`NoGC_fork` v_0`.`$mn\langle o_{1..n}\rangle$`(`$v_{1..m}$`)`" is that we subclass T from `NoHeapRealtimeThread`, instead of `RealtimeThread`. In RTSJ, a `NoHeapRealtimeThread` is not interrupted by the garbage collector because it cannot manipulate heap references. RTSJ ensures this using dynamic checks. Our system ensures this statically, so we can remove these dynamic checks if an RTSJ platform allows us to do so.

C.9 Avoiding Priority Inversion Using Static Type Checking

Recall the priority inversion problem from Section C.3. A real-time thread entering a region waits for threads exiting (and perhaps flushing) the region. If a regular thread exiting the region is suspended (for

```
{
  class T extends javax.realtime.RealtimeThread {
    ∀ i ∈ {0,...,m}, one field to store the value of v_i (of type t_i):
      t_i vi;
    public void run() {
      try {
        v0.mn(v1,..., vm);
      } finally {
        for(int i = 0; i < getMemoryAreaStackDepth(); i++) {
          IRegion isr = (IRegion) getOuterMemoryArea(i);
          synchronized(isr) { isr.tryToFlush(); }
        }
      }
    } // end of run()
  }
  T t = new T();
  ∀ i ∈ {0,...,m} generate one line of the form:
    t.vi = v_i;
  t.start();
}
```

where T and t are fresh identifiers.

Figure 20: Translation for "fork $v_0 . mn\langle o_{1..n}\rangle(v_{1..m})$"

an unbounded amount of time) by the garbage collector, the real-time thread might have to wait for an unbounded amount of time before being able to enter the region.

We can avoid this problem by modifying our type system slightly. Note that the problem occurs only when a regular thread and a real-time thread share a region. Note also that the problem occurs only when a real-time thread is trying to enter a region. But since a real-time thread cannot create new regions, it can only enter sub-regions of shared regions. Therefore, if we prevent regular threads and real-time threads from sharing the same sub-regions, we can avoid the priority inversion problem.

Here are the necessary extensions to the type system to achieve this:

1. For space reasons, we use the names GC, NoGC threads for normal, respectively real-time threads. For each subregion declaration, we ask the programmer to specify whether that subregion is 1) a GC subregion (i.e., only GC threads can access it) or 2) a NoGC region (i.e., only NoGC threads can access it). We introduce a grammar rule for the thread types tt and we update the grammar rule for subregion declarations:

$$\begin{array}{rcl} tt & ::= & \mathsf{GC} \mid \mathsf{NoGC} \\ subsreg & ::= & srkind : rpol\ \ tt\ \ rsub \end{array}$$

2. When we enter a subregion, we can find out from its declaration whether it is a GC or a NoGC subregion. We need to check that the current thread has the appropriate *thread type* (GC or NoGC). For this, we modify the type judgments for expressions $P; E; X; r_{cr} \vdash e : t$ to keep track of the possible types of threads that can execute e. The new type judgments have the form $P; E; X; r_{cr}; T \vdash e : t$, where $T \subseteq \{\mathsf{GC}, \mathsf{NoGC}\}$ is the set of types of the threads that may execute e. Most of the rules just propagate T without using it. The rule for entering a subregion is one of the exceptions:

[EXPR SUBREGION]

$$\frac{\begin{array}{c} E; X; r_{cr}; h_2; P \vdash \mathtt{RHandle}\langle r_2\rangle : \quad E \vdash_{\mathrm{k}} r_2 : srkn_2\langle o_{1..n}\rangle \\ P \vdash rkind_3 : rpol\ \ tt\ \ rsub \in srkn_2\langle (k_i\ fn_i)_{i\in\{1..n\}}\rangle \quad T = \{tt\} \\ \text{... the other antecedents are unchanged ...} \end{array}}{P; E; X; r_{cr}; T \vdash (\mathtt{RHandle}\langle r\rangle\ \ h_1\ \ \mathtt{=}\ \ [\mathtt{new}]_{\mathrm{opt}}\ \ h_2 . rsub)\ \ \{e\} : \mathsf{int}}$$

This way, only a normal (i.e., GC) thread can execute an expression that enters a GC subregion. Similarly, only a real-time (i.e., NoGC) thread can execute an expression that enters a NoGC subregion. Expressions that do not enter any subregion can be executed by any thread.

3. Each method declaration should specify the types of the threads that may execute it:

$$meth \;\; ::= \;\; t \;\; mn\langle formal^*\rangle\texttt{(}(t \;\; p)^*\texttt{)} \;\; effects \;\; \texttt{where} \;\; constr^* \;\; \texttt{callableFrom} \;\; tt^* \;\; \{e\}$$

   By default, if a method declaration does not have any `callableFrom` clause, we assume the method is declared as callable from any thread, i.e. "`callableFrom GC,NoGC`". The list of thread types from the method declaration is used while typechecking the expression e, the body of the method:

   [METHOD]

$$\frac{\begin{array}{c} E' = E,\; f_{1..n},\; constr_{1..c},\; (t_j \;\; p_j)_{j\in\{1..p\}},\; \mathsf{Region\ initialRegion},\; \texttt{RHandle}\langle\mathsf{initialRegion}\rangle\; h_{fresh} \\ P \vdash_{\text{env}} E' \quad P; E'; a_{1..q}; \mathsf{initialRegion}; T \vdash e : t \end{array}}{P; E \vdash_{\text{meth}} t \;\; mn\langle f_{1..n}\rangle\texttt{(}(t_j \;\; p_j)_{j\in\{1..p\}}\texttt{)} \;\; \texttt{accesses} \;\; a_{1..q} \;\; \texttt{where} \;\; constr_{1..c} \;\; \texttt{callableFrom} \;\; T \;\; \{e\}}$$

   Therefore, if a method declares that a normal (i.e., GC) thread may call it, then that method cannot be called from a NoGC thread.

4. The rule for a `fork` expression checks that the initial method invoked in the child thread may be called from a GC thread. Analogously, the rule for a `NoGC_fork` expression checks that the corresponding method may be called from a NoGC thread.

5. We update the rule for method invocation to check that the set of types of the threads that may execute the call expression is included in the set of types of the threads that may execute the invoked method:

   [EXPR INVOKE]

$$\frac{\begin{array}{c} P; E; X; r_{cr}; T \vdash v : cn\langle o_{1..n}\rangle \\ P \vdash t \;\; mn\langle f_{(n+1)..m}\rangle\texttt{(}(t_j \;\; p_j)_{j\in\{1..k\}}\texttt{)} \; ... \; \texttt{callableFrom} \;\; T_{mn} \;\; \{e\} \in cn\langle f_{1..n}\rangle \\ T \subseteq T_{mn} \\ \text{... the other antecedents are unchanged ...} \end{array}}{P; E; X; r_{cr}; T \vdash v\,\texttt{.}\,mn\langle o_{(n+1)..m}\rangle\texttt{(}v_{1..k}\texttt{)} : Rename(t)}$$