Ownership Types for Safe Region-Based Memory Management in Real-Time Java

Chandrasekhar Boyapati, Alexandru Sălcianu, William Beebee, Jr., Martin Rinard
MIT Laboratory for Computer Science
200 Technology Square, Cambridge, MA 02139
{chandra,salcianu,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 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 the benefits of 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 memory leaks. For real-time programs, it ensures that real-time threads do not interfere with the garbage collector. Our experience indicates that our type system is sufficiently expressive and requires little programming overhead, and that eliminating the RTSJ runtime checks using a static type system can significantly decrease the execution time of real-time programs.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs;
D.2.4 [**Software Engineering**]: Program Verification;
C.3 [**Special-Purpose Systems**]: Real-time Systems

General Terms

Languages, Verification, Theory

This research was supported by DARPA/AFRL Contract F33615-00-C-1692, NSF Grant CCR00-86154, NSF Grant CCR00-73513, NSF Grant CCR02-09075, and the Singapore-MIT Alliance.

PLDI'03, June 9–11, 2003, San Diego, California, USA.

Keywords

Ownership Types, Regions, Encapsulation, Real-Time

1. Introduction

The Real-Time Specification for Java (RTSJ) [8] 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 [42]. 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 RTSJ runtime checks will never fail for well-typed programs. Our system thus serves as a front-end for the RTSJ platform. It offers two advantages to real-time programmers. First, it provides an important safety guarantee that a program will never fail because of a failed RTSJ runtime check. Second, it allows RTSJ implementations to remove the RTSJ runtime checks and eliminate the associated overhead.

Our approach is applicable even outside the RTSJ context; it could be adapted to provide safe region-based memory management for other real-time languages as well.

Our system makes several important technical contributions over previous type systems for region-based memory management. For object-oriented programs, it combines region types [16, 23, 34, 42] and ownership types [11, 12, 14, 18, 19] in a unified type system framework. Region types statically ensure that programs never follow dangling references. Ownership types statically enforce object encapsulation and enable modular reasoning about program correctness in object-oriented programs.

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 (e.g., [16]), 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`. Our system thus combines the benefits of region types and ownership types.

Our system extends region types to multithreaded programs by allowing explicit memory management for objects shared between threads. It 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 only shared regions (e.g., [33]) 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. Our approach therefore avoids any dynamic type errors associated with these downcasts. Our system introduces user-defined *region kinds* to support subregions and portal fields.

Our system extends region types to real-time programs by statically ensuring that real-time threads do not interfere with the garbage collector. Our system augments region kind declarations with *region policy* declarations. It supports two policies for creating regions as in the 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.

Our system also prevents an RTSJ *priority inversion* problem. In the RTSJ, any thread entering a region waits if there are threads exiting the region. If a regular thread exiting a region is suspended by the garbage collector, then a real-time thread entering the region might have to wait for an unbounded amount of time. Our type system statically ensures that this priority inversion problem cannot happen.

Finally, we note that ownership-based type systems have also been used for preventing data races [14] and deadlocks [11], for supporting modular software upgrades in persistent object stores [13], for modular specification of effects clauses in the presence of subtyping [12, 14] (so they can be used as an alternative to data groups [36]), and for program understanding [3]. We are currently unifying the type system presented in this paper with the above type systems [9]. The unified ownership type system requires little programming overhead, its typechecking is fast and scalable, and it provides several benefits. The unified ownership type system thus offers a promising approach for making object-oriented programs more reliable.

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 1) *subregions* within a shared region, so that long-lived threads can share objects without using the heap and without memory leaks and 2) *typed portal fields* to serve as a starting point for typed inter-thread communication. It also 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 the 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. It also prevents an RTSJ *priority inversion* problem.
- **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 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 the programs on our RTSJ platform [6, 7]. Our experiments show that eliminating the RTSJ runtime checks using a static type system can significantly speed-up programs.

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 describe our type system. Section 2.4 presents some of the important rules for typechecking. The complete set of rules are in the Appendix. Section 2.5 describes type inference techniques. Section 2.6 describes how programs written in our system are translated to run on our RTSJ platform.

2.1 Regions for Object-Oriented Programs

This section presents our type system for safe region-based memory management in single-threaded object-oriented programs. It combines the benefits of region types [16, 23, 34, 42] and ownership types [11, 12, 14, 18, 19]. Region types statically ensure that programs using region-based memory management are memory-safe, that is, they never follow dangling references. Ownership types statically enforce

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 $r_2 \not\succeq r_1$, then o_1 cannot contain a pointer to o_2.

Figure 2: Outlives Properties

object encapsulation. The idea is that an object can *own* subobjects that it depends on, thus preventing them from being accessible outside. (An object x *depends on* [35, 12] subobject y if x calls methods of y and furthermore these calls expose mutable behavior of y in a way that affects the invariants of x.) Object encapsulation enables local reasoning about program correctness in object-oriented programs.

Ownership Relation Objects in our system are allocated in regions. Every object 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 *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 programs to create regions. It 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 a region r_1 includes the lifetime of region r_2, we say that r_1 *outlives* r_2, and write $r_1 \succeq r_2$. The relation $\succeq$ is thus reflexive and transitive. We extend the *outlives* relation to include objects. We define that $x \succeq_o y$ implies $x \succeq y$. 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 o then r outlives o because o is allocated in r. 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 states our memory safety property, that if object o_1 in region r_1 contains a pointer to object o_2 in region r_2, then r_2 outlives r_1. 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`,

[1]Our system handles inner class objects specially to support constructs like iterators. Details can be found in [12].

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` | `heap` | `immortal`
$field$	::=	$t\ fd$
$meth$	::=	$t\ mn\langle formal^*\rangle$`((`$t\ p$`)`*`)` `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 | `let` $v = e$ `in` { e } | $v.fd$ | $v.fd = v$ | `new` c | $v.mn\langle owner^*\rangle(v^*)$ | `(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

Figure 3: Grammar for Object Oriented Programs

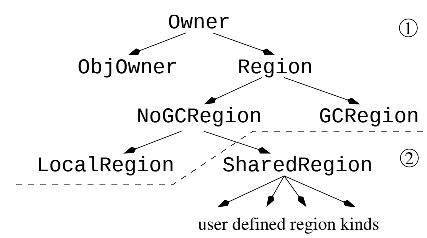

Figure 4: Owner Kind Hierarchy: Section 2.1 uses only Area 1. Sections 2.2 & 2.3 use Areas 1 & 2.

and `heap` and `immortal` outlive all regions. The following lemmas follow trivially from the above definitions:

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 [28]. 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 [1, 15, 38] 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 first formal owner is special: it owns the corresponding object; the other owners propagate the ownership information. Methods can also declare an additional list of formal owner parameters. Each time new formals are introduced, programmers can specify constraints between them using `where` clauses [24]. 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` (the kind of other 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 section.)

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 *handle* of the region r. The region name r is only a compile-time 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 implies a "last in first out" order on region lifetimes. 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" region `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 the special region `initialRegion`, which denotes the most recent region that was created before the method was called. We use runtime support to acquire the handle of `initialRegion`.

Object Creation New objects are created using the expression "new $cn\langle o_{1..n} \rangle$". o_1 is the owner of the new object. (Recall that the first owner parameter always owns the corresponding 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 the handle of the region we allocate in. The typechecker checks that we can obtain such a handle (more details are in Section 2.4). If o_1 is a region r, the 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.

A formal owner parameter can be instantiated 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 restrictions 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 `TStack` object and `TOwner` owns the `T` objects contained in the `TStack`. The code specifies that the `TStack` object owns

```
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> value;
  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

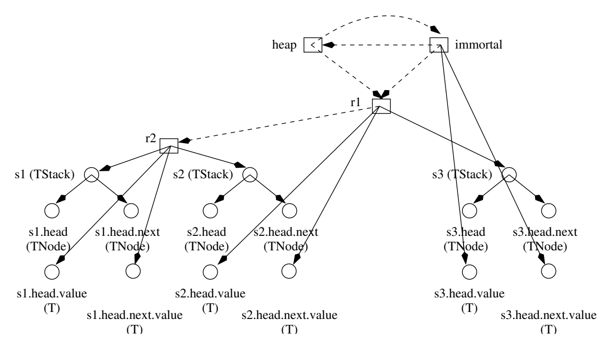

Figure 6: TStack Ownership and Outlives Relations

the nodes in the list; therefore the list nodes cannot be accessed from outside the `TStack` object. The program creates two regions `r1` and `r2` such that `r1` outlives `r2`. The program declares several `TStack` variables: the type of `TStack` `s1` specifies that it is allocated in region `r2` and so are the `T` objects in `s1`; `TStack` `s2` is allocated in region `r2` but the `T` objects in `s2` are allocated in region `r1`; etc. Note that the type of `s6` is illegal. This is because `s6` is declared as TStack$\langle$r1,r2$\rangle$, and r2 $\not\succeq$ r1. (Recall that in any legal type $cn\langle o_{1..n} \rangle$ with multiple owners, $o_i \succeq o_1$ for all $i \in \{1..n\}$.) 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 state our safety guarantees. Part 1 of Theorems 3 and 4 state the object encapsulation property. Note that objects owned by regions are not encapsulated within other objects. Part 2 of Theorem 3 states the memory safety property.

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. *Either owner of $o_2 \succeq_o o_1$, or owner of o_2 is a region.*
2. *Region r_2 outlives region r_1.*

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 be either 1) `heap`, or 2) `immortal`, or 3) `this`, or 4) f_i, a class formal. In the first two cases, (owner of o_2) = x_1 is a region, and $r_2 = x_1 \succeq r_1$. In Case 3, (owner of o_2) = $o_1 \succeq_o o_1$, and $r_2 = r_1 \succeq r_1$. 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 $r \succeq_o o_1$. Therefore $r_2 = r \succeq r_1$. □

THEOREM 4. *If a variable v in a method mn of an object o_1 points to an object o_2, then*

1. *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. □

Most previous region type systems allow programs to create, but not follow, dangling references. Such references can cause a safety problem when used with moving collectors. Our system therefore prevents a program from creating dangling references in the first place. Part 2 of Theorem 3 prevents object fields from containing dangling references. Even though Theorem 4 does not have a similar Part 2, we can prove, using lexical scoping of region names, that local variables cannot contain dangling references either.

2.2 Regions for Multithreaded Programs

This section describes how we support multithreaded programs. Figure 7 presents the language extensions. A `fork` instruction spawns a new thread that evaluates the invoked method. The evaluation is performed only for its effect; the parent thread does not wait for the completion of the new thread and does not use the result of the method call. Our unstructured concurrency model (similar to Java's model) is incompatible with the regions from Section 2.1 whose lifetimes are lexically bound. Those regions can still be used for allocating thread-local objects (hence the name of the associated region kind, `LocalRegion`), but objects shared by multiple threads require *shared* regions, of kind `SharedRegion`.

Shared Regions "(`RHandle`$\langle$*rkind r*$\rangle$ *h*) {*e*}" creates a shared region (*rkind* specifies the region kind of *r*; region kinds are explained later in this section). Inside expression *e*, the identifiers *r* and *h* are bound to the region and the region handle, respectively. Inside *e*, *r* and *h* can be passed to child threads. The objects allocated inside a shared region are not deleted 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 onto the region stack of the current thread and its counter is initialized to one. 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 the lifetimes of the regions 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. When the counter of a region becomes zero, the region is deleted. The typing rule for a `fork` expression checks that objects allocated in local regions are not passed to the child thread as arguments; it also checks that local regions and handles to local regions are not passed to the child thread.

P	::=	*def* srkdef* e*
srkdef	::=	`regionKind` *srkn*⟨*formal**⟩ `extends` *srkind* `where` *constr** { *field* subsreg** }
rkind	::=	... as in Figure 3 ... | *srkind*
srkind	::=	*srkn*⟨*owner**⟩ | `SharedRegion`
subsreg	::=	*srkind rsub*
e	::=	... as in Figure 3 ... | `fork` *v.mn*⟨*owner**⟩(*v**) | (`RHandle`⟨*rkind r*⟩ *h*) { *e* } | (`RHandle`⟨*r*⟩ *h* = [`new`]$_{opt}$ *h.rsub*) { *e* } | *h.fd* | *h.fd* = *v*
srkn	∈	shared region kind names
rsub	∈	shared subregion names

Figure 7: Extensions for Multithreaded Programs

Subregions and Portals Shared regions provide the basis for inter-thread communication. However, in many cases, they are not enough. E.g., consider two long-lived 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 and the consumer subsequently uses the objects. These objects become unreachable after each iteration. However, these objects are not deleted until both threads terminate and exit the shared region. To prevent this memory leak, we allow shared regions to have subregions. In each iteration, the producer and the consumer can enter a subregion of the shared region and use it for communication. At the end of the iteration, both the threads exit the subregion and the reference count of the subregion goes to zero—the objects in the subregion are thus deleted after each iteration.

We must also allow the producer to pass references to objects it allocates in the subregion in each iteration to the consumer. Note that storing the references in the fields of a "hook" object is not possible: objects allocated outside the subregion cannot point to objects in the subregion (otherwise, those references would result in dangling references when objects in the subregion are deleted), and objects allocated in the subregion do not survive between iterations and hence cannot be used as "hooks". To solve this problem, we allow (sub)regions to contain *portal* fields. A thread can store the reference to an object in a portal field; other threads can then read the portal field to obtain the reference.

Region Kinds In practice, programs can declare several shared region kinds. Each such kind extends another shared region kind and can declare several portal fields and subregions (see grammar rule for *srkdef* in Figure 7). The resulting shared region kind hierarchy has `SharedRegion` as its root. The owner kind hierarchy now includes both Areas 1 and 2 from Figure 4. Similar to classes, shared region kinds

```
regionKind BufferRegion extends SharedRegion {
  BufferSubRegion b;
}

regionKind BufferSubRegion extends SharedRegion {
  Frame<this> f;
}

class Producer<BufferRegion r> {
  void run(RHandle<r> h) {
    while(true) {
      (RHandle<BufferSubRegion r2> h2 = h.b) {
        Frame<r2> frame = new Frame<r2>;
        get_image(frame);
        h2.f = frame;
      }
      ... // wake up the consumer
      ... // wait for the consumer
}}}

class Consumer<BufferRegion r> {
  void run(RHandle<r> h) {
    while(true) {
      ... // wait for the producer
      (RHandle<BufferSubRegion r2> h2 = h.b) {
        Frame<r2> frame = h2.f;
        h2.f = null;
        process_image(frame);
      }
      ... // wake up the producer
}}}

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

Figure 8: Producer Consumer Example

can be parameterized with owners; however, unlike objects, regions do not have owners so there is no special meaning attached to the first owner.

Expression "$\texttt{(RHandle}\langle r_2\rangle\ h_2\ \texttt{=}\ [\texttt{new}]_{\mathrm{opt}}\ h_1\texttt{.}\mathit{rsub}\texttt{)}\ \{e\}$" evaluates e in an environment where r_2 is bound to the subregion *rsub* of the region r_1 that h_1 is the handle of, and h_2 is bound to the handle of r_2. In addition, if the keyword `new` is present, r_2 is a newly created subregion, distinct from the previous *rsub* subregion.

If h is the handle of region r, the expression "$h\,.\mathit{fd}$" reads r's portal field fd, and "$h\,.\mathit{fd}\ \texttt{=}\ v$" stores a value into that field. The rule for portal fields is the same as that for object fields: a portal field of a region r is either null or points to an object allocated in r or in a region that outlives r.

Flushing Subregions When all the objects in a subregion become inaccessible, the subregion is flushed, i.e., all objects allocated inside it are deleted. 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 in those subregions might point to objects in 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 its handle), the objects allocated inside it can be reachable even if no thread is currently in the subregion.

Example Figure 8 contains an example that illustrates the use of subregions and portal fields. The main thread creates a shared region of kind `BufferRegion` and then starts

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

Figure 9: Extensions for Real-Time Programs

two threads, a producer and a consumer, that communicate through the shared region. In each iteration, the producer enters subregion `b` (of kind `BufferSubRegion`), allocates a `Frame` object in it, and stores a reference to the frame in subregion's portal field `f`. Next, the producer exits the subregion and waits for the consumer. The subregion is not flushed because the portal field `f` is non-null. The consumer then enters the subregion, uses the frame object pointed to by its portal field `f`, sets `f` to `null`, and exits the subregion. Now, the subregion is flushed (because its counter is zero and all its fields are null) and a new iteration starts. In this paper, we do not discuss synchronization issues; we assume synchronization primitives similar to those in Java.

2.3 Regions for Real-Time Programs

A real-time program consists of a set of real-time threads, a set of regular threads, and a special garbage collector thread. (This is a conceptual model; actual implementations might differ.) A real-time thread has strict deadlines for completing its tasks.[2]

Figure 9 presents the language extensions to support real-time programs. The expression "$\texttt{RT\_fork}\ v\texttt{.}\mathit{mn}\langle \mathit{owner}^*\rangle\texttt{(}v^*\texttt{)}$" spawns a new real-time thread to evaluate *mn*. Such a thread cannot afford to be interrupted for an unbounded amount of time by the garbage collector—the rest of this section explains how our type system statically ensures this property.

Effects The garbage collector thread must synchronize with any thread that creates or destroys heap roots, i.e., references to heap objects, otherwise it might end up collecting reachable objects. Therefore, we must ensure that the real-time threads do not read or overwrite references to heap objects. (The last restriction is needed to support moving collectors.) To statically check this, we allow methods to declare *effects* clauses [37]. In our system, the effects 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 or overwriting a reference to that object or allocating another object owned by that object. Note that we do not consider reading or writing a field of an object as accessing that object. If a method's effects clause consists of owners $o_{1..n}$, then any object or region accessed by that method, the methods it invokes, and the threads it spawns (transitively) is guaranteed to be outlived by o_i, for some $i \in \{1..n\}$.

The typing rule for an `RT_fork` expression checks all the constraints of a regular `fork` expression. In addition, it checks that references to heap objects are not passed as arguments to the new thread, and that the effects clause

[2]Our terminology is related, but not identical to the RTSJ terminology. E.g., our real-time threads are similar to (and more restrictive than) the RTSJ `NoHeapRealtimeThread`s.

of the method evaluated in the new thread does not contain the heap region or any object allocated in the heap region. If an `RT_fork` expression typechecks, the new real-time thread cannot receive any heap reference. Furthermore, it cannot create a heap object, or read or overwrite a heap reference in an object field—the type system ensures that in each of the above cases, the heap region or an object allocated in the heap region appears in the method effects.

Region Allocation Policies A real-time thread cannot create an object if this operation requires allocating new memory, because allocating memory requires synchronization with the garbage collector. A real-time thread can, however, create an object in a preallocated memory region.

Our system supports two allocation policies for regions. One policy is to allocate memory on demand (potentially in large chunks), as new objects are created in the region. Allocating a new object can take unbounded time or might not even succeed (if a new chunk is needed and the system runs out of memory). Flushing the region frees all the memory 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 region creation time. 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—if the region is already full, the system throws an exception to signal that the region size was too small. Allocating a new object takes time 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, threads can repeatedly enter it, allocate objects in it, exit it (thus flushing it), and re-enter it without having to allocate new memory. This is possible because flushing an LT region does not free its memory. LT subregions are thus ideal for real-time threads: once such a subregion is created (with a large enough size), all object creations will succeed, in linear time; moreover, the subregion can be flushed and reused without memory allocation.

We allow users to specify the region allocation policy (LT or VT) when a new region is created. The policy for subregions is declared in the shared region kind declarations. When a user specifies an LT policy, the user also has to specify the size of the region (in bytes). An expression "$(\texttt{RHandle}\langle rkind : rpol\ \ r\rangle\ \ h)\ \{e\}$" creates a region with allocation policy *rpol* and allocates memory for all its (transitive) LT (sub)regions (including itself). Our system checks that a region has a finite number of transitive subregions.

If a method enters a VT region or a top level region (i.e., a region that is not a subregion), the typechecker ensures that the method contains the heap region in its effects clause. This is to prevent real-time threads from invoking such methods. However, a method that does not contain the heap region in its effects clause can still enter an existing LT subregion, because no memory is allocated in that case.

Preventing the RTSJ Priority Inversion So far, we presented techniques for checking that real-time threads do not create or destroy heap references, create new regions, or allocate objects in VT regions. However, there are two other subtle ways a thread can interact with the garbage collector.

First, the garbage collector needs to know all locations that refer to heap objects, including locations that are inside regions. Suppose a real-time thread uses an LT region that contains such heap references (created by a non-real-time thread). The real-time thread can flush the region (by exiting it) thus destroying any heap reference that existed in the region. If we use a moving garbage collector, the real-time thread has to interact with the garbage collector to inform it about the destruction of those heap references. Therefore, we should prevent regions that can be flushed by a real-time thread from containing any heap reference (even if the reference is not explicitly read or overwritten by the real-time thread). Note that this restriction is relevant only for subregions: a real-time thread cannot create a top-level region and hence cannot flush a top-level region either.

Second, when a thread enters or exits a subregion, it needs to do some bookkeeping. To preserve the integrity of the runtime region implementation, some synchronization is necessary during this bookkeeping. E.g., when a thread exits a subregion, the test that the subregion can be flushed and the actual flushing have to be executed atomically, without allowing any thread to enter the subregion "in between". If a regular thread exiting a subregion is suspended by the garbage collector, then a real-time thread entering the subregion might have to wait for an unbounded amount of time. This *priority inversion* problem occurs even in the RTSJ.

To prevent these subtle interactions, we impose the restriction that real-time threads and regular threads cannot share subregions. Subregions used by real-time threads thus cannot contain heap references, and real-time threads never have to wait for unbounded amounts of time.

For each subregion, programmers specify in the region kind definitions whether the subregion will be used only by real-time threads (`RT` subregions) or only by regular threads (`NoRT` subregions). Note that real-time and regular threads can still communicate using top-level regions. Any method that enters an `RT` subregion must contain the special effect `RT` in its effects clause. Any method that enters a `NoRT` subregion must contain the heap region in its effects clause. The type system checks that no regular thread can invoke a method that has an `RT` effect, and no real-time thread can invoke a method that has a heap effect.

2.4 Rules for Typechecking

Previous sections presented the grammar for our core language in Figures 3, 7, and 9. This section presents some sample typing rules. The Appendix contains all the rules.

The core of our type system is a set of typing judgments of the form $P; E; X; r_{cr} \vdash e : t$. 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, $E ::= \emptyset \mid E,\ t\ \ v \mid E,\ k\ o \mid E,\ o_2 \succeq_o o_1 \mid E,\ o_2 \succeq o_1$. r_{cr} is the current region. X must subsume the effects of e. t is the type of the expression e.

A useful auxiliary rule is $E \vdash X_1 \succeq X_2$, i.e., the effects X_1 *subsume* the effects X_2: $\forall o \in X_2,\ \exists g \in X_1$, s.t. $g \succeq o$. To prove constraints of the form $g \succeq o$, $g \succeq_o o$ 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 fact that the first owner from the type of an object owns the object.

The expression "$\texttt{(RHandle}\langle r\rangle\ h\texttt{)}\ \{e\}$" creates a local region and evaluates e in an environment where r and h are bound to the new region and its handle respectively. The associated typing rule is presented below:

[EXPR LOCAL REGION]

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

The rule starts by constructing an environment E_2 that extends the original environment E by recording that r has kind `LocalRegion` and h has type $\texttt{RHandle}\langle r\rangle$. As r is deleted at the end of e, all existing regions outlive it; E_2 records this too ($Regions(E)$ denotes the set of all regions from E). e should typecheck in the context of the environment E_2 and the permitted effects are 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 and its result is never used. Hence, the type of the entire expression is `int`.

The rule for a field read expression "$v.fd$" first finds the type $cn\langle o_{1..n}\rangle$ for v. Next, it verifies that fd is a field of class cn; let t be its declared type. The rule obtains the type of the entire expression by substituting in t each formal owner parameter fn_i of cn with the corresponding owner o_i:

[EXPR REF READ]

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

The last line of the rule checks that if the expression reads an object reference (i.e., not an integer), then the list of effects X subsumes the owner of the referenced object.

For an object allocation expression "$\texttt{new}\ cn\langle o_{1..n}\rangle$", the rule first checks that class cn is defined in P:

[EXPR NEW]

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

Next, it checks 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 is a subkind of the declared kind k_i of fn_i. It also checks that in E, each owner o_i outlives the first owner o_1, and each constraint of cn is satisfied. Allocating an object means accessing its owner; therefore, X must subsume o_1. The new object is allocated in the region o_1 (if o_1 is a region) or in the region that o_1 is allocated in (if o_1 is an object). The last part of the precondition, $E \vdash_{\text{av}} \texttt{RH}(o_1)$, checks that the handle for this region is available. To prove facts of this kind, the type system uses the following rules:

[AV HANDLE]

$$\frac{E = E_1,\ \texttt{RHandle}\langle r\rangle\ h,\ E_2}{E \vdash_{\text{av}} \texttt{RH}(r)}$$

[AV THIS]

$$\frac{}{E \vdash_{\text{av}} \texttt{RH(this)}}$$

[AV TRANS1]

$$\frac{E \vdash o_1 \succeq_o o_2 \quad E \vdash_{\text{av}} \texttt{RH}(o_2)}{E \vdash_{\text{av}} \texttt{RH}(o_1)}$$

[AV TRANS2]

$$\frac{E \vdash o_1 \succeq_o o_2 \quad E \vdash_{\text{av}} \texttt{RH}(o_1)}{E \vdash_{\text{av}} \texttt{RH}(o_2)}$$

The rule [AV HANDLE] looks for a region handle in the environment. The environment always contains handles for `heap` and `immortal`; in addition, it contains all handle identifiers that are in scope. The rule [AV THIS] reflects the fact that our runtime is able to find the handle of the region where an object (`this` in particular) is allocated. The last 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 the region handle for one of them is available, then the region handle for the other one is also available. Note that these rules do significant reasoning, thus reducing annotation burden; e.g., if a method allocates only objects (transitively) owned by `this`, it does not need an explicit region handle argument.

We end this section with the typing rule for `fork`. The rule first checks that the method call is well-typed. (see rule [EXPR INVOKE] in Appendix B). Note that mn cannot have the `RT` effect: a non-real-time thread cannot enter a subregion that is reserved only for real-time threads.

[EXPR FORK]

$$\frac{\begin{array}{c} P; E; X \setminus \{\texttt{RT}\}; r_{cr} \vdash v_0 . mn\langle o_{(n+1)..m}\rangle(v_{1..u}) : t \\ NonLocal(k) \stackrel{def}{=} (P \vdash k \leq_{\text{k}} \texttt{SharedRegion}) \ \vee\ (P \vdash k \leq_{\text{k}} \texttt{GCRegion}) \\ E \vdash \text{RKind}(r_{cr}) = k_{cr} \quad NonLocal(k_{cr}) \\ P; E; X; r_{cr} \vdash v_0 : cn\langle o_{1..n}\rangle \\ \forall i = 1..m,\ (E \vdash \text{RKind}(o_i) = k_i \ \wedge\ NonLocal(k_i)) \end{array}}{P; E; X; r_{cr} \vdash \texttt{fork}\ v_0 . mn\langle o_{(n+1)..m}\rangle(v_{1..u}) : \texttt{int}}$$

The rule checks that the new thread does not receive any local region or objects allocated in a local region. It uses the following observation: the only owners that appear in the types of the method arguments are: `initialRegion`, `this`, the formals for the method and the formals for the class the method belongs to. Therefore, the arguments passed to the method mn from the `fork` instruction may be owned only by the current region at the point of the `fork`, by the owners $o_{1..n}$ that appear in the type of the object v_0 points to, or by the owners $o_{(n+1)..m}$ that appear explicitly in the `fork` instruction. For each such owner o, our system uses the rule $E \vdash \text{RKind}(o) = k$ to extract the kind k of the region it stands for (if it is a region), or of the region it is allocated in (if it is an object). The rule next checks that k is a subkind of `SharedRegion` or `GCRegion`. The rules for inferring statements of the form $E \vdash \text{RKind}(o_i) = k_i$ (see Appendix B) are similar to the previously explained rules for checking that a region handle is available. 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 type inference and well-chosen defaults to significantly reduce the number of annotations needed in practice. Our system also supports user-defined defaults to cover specific patterns that might occur in user code. 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.

The following are some defaults currently provided by our system. 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 [14,

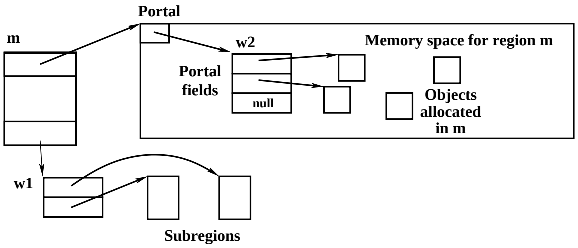

Figure 10: Translation of a Region With Three Fields and Two Subregions.

11]. 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`.

2.6 Translation to Real-Time Java

Although our system provides significant improvements over the RTSJ, programs in our language can be translated to RTSJ reasonably easily, by local translation rules. This is mainly because we designed our system so that it can be implemented using type erasure (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 and variable time (respectively) 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 the corresponding 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 portal field except that its declared type is `Object`. Most of the translation effort is focused on providing the missing features: subregions and multiple, typed portal fields. We discuss the translation of several important features from our type system; the full translation is discussed in [40].

We represent a region r from our system as an RTSJ region `m` plus two auxiliary objects `w1` and `w2` (see Figure 10). `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 portal fields. `w2` is allocated in the memory space attached to `m`, while `m` and `w1` are allocated in the region that was current at the time `m` was created.

The translation of "`new` $cn\langle o_{1..n}\rangle$" requires a reference to (i.e., the 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., $E \vdash_{\mathrm{av}} \mathtt{RH}(o_1)$; we presented the relevant type rules in Section 2.4. Those rules "pushed" the judgment $E \vdash_{\mathrm{av}} \mathtt{RH}(o)$ up and down the ownership relation until we obtained an owner whose region handle was available: `immortal`, `heap`, `this`, or a region whose region handle was available in a local variable. RTSJ provides mechanisms for retrieving the handle in the first three cases: `ImmortalArea.instance()`, `HeapArea.instance()`, and `MethodArea.getMethodArea(Object)`, respectively. In the last case, we simply use the handle from the local 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 11: Programming Overhead

Program	Execution Time (sec)		Overhead
	Static Checks	Dynamic 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 12: Dynamic Checking Overhead

3. Experience

To gain preliminary experience, we implemented several programs in our system. These include two micro benchmarks (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 our implementations, the primary data structures are allocated in regions (i.e., not in the garbage collected heap). In each case, once we understood how the program worked and decided on the memory management policy to use, adding the extra type annotations was fairly straightforward. Figure 11 presents a measure of the programming overhead involved. It shows the number of 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 checks specified in the Real-Time Specification for Java. Figure 12 presents the running times of the benchmarks both with and without dynamic checks. Note that there is no garbage collection overhead in any of these running times because the garbage collector never executes. Our micro benchmarks (Array and Tree) were written specifically to maximize the checking overhead—our development goal was to maximize the ratio of assignments to other computation. These 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 provide a more realistic picture of the dynamic checking overhead. These programs have more modest performance improvements, running up to 1.25 times faster without the 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 [21, 22].

4. Related Work

The seminal work in [43, 42] 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.

One disadvantage with most region-based management systems is that they enforce a lexical nesting on region lifetimes; so objects allocated in a given region may become inaccessible long before the region is deleted. [2] presents an analysis that enables some regions to be deleted early, as soon as all of the objects in the region are unreachable. Other approaches include the use of linear types to control when regions are deleted [23, 25]. None of these approaches currently support object-oriented programs and the consequent subtyping, multithreaded programs with shared regions, or real-time programs with real-time threads (although it should be possible to extend them to do so). Conversely, it should also be possible to apply these techniques to our system. In fact, existing systems already combine ownership-based type systems and unique pointers [20, 14, 3].

RegJava [16] has a region type system for object-oriented programs that supports subtyping and method overriding. Cyclone [34] is a dialect of C with a region type system. Our work improves on these two systems by combining the benefits of ownership types and region types in a unified framework. An extension to Cyclone handles multithreaded programs and provides shared regions [33]. 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 [29, 30] 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 for memory management in real-time programs. [26, 27] automatically translates 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. We previously used escape analysis [39] to remove RTSJ runtime checks [41]. However, the analysis is effective only for programs in which no object escapes the computation that allocated it. Our type system is more flexible: we allow a computation to allocate objects in regions that may outlive the computation.

Real-time garbage collection [5, 4] provides an alternative to region-based memory management for real-time programs. It has the advantage that programmers do not have to explicitly deal with memory management. The basic idea is to perform a fixed amount of garbage collection activity for a given amount of allocation. With fixed-size allocation blocks and in the absence of cycles, reference counting can deliver a real-time garbage collector that imposes no space overhead as compared with manual memory management. Copying and mark and sweep collectors, on the other hand, pay space to get bounded-time allocation. The amount of extra space depends on the maximum live heap size, the maximum allocation rate, and other memory management parameters. The additional space allows the collector to successfully perform allocations while it processes the heap to reclaim memory. To obtain the real-time allocation guarantee, the programmer must calculate the required memory management parameters, then use those values to provide the collector with the required amount of extra space. In contrast, region-based memory management provides an explicit mechanism that programmers can use to structure code based on their understanding of the memory usage behavior of a program; this mechanism may enable programmers to obtain a smaller space overhead. The additional development burden consists of grouping objects into regions and determining the maximum size of LT regions [31, 32].

5. Conclusions

The Real-Time Specification for Java (RTSJ) allows programs to create real-time threads and use region-based memory management. 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 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 the benefits of 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. Our experience indicates that our type system is sufficiently expressive and requires little programming overhead, and that eliminating the RTSJ runtime checks using a static type system can significantly decrease the execution time of real-time programs.

Acknowledgments

We are grateful to Viktor Kuncak, Darko Marinov, Karen Zee, and the anonymous referees for their comments.

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. Fähndrich, 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] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Principles of Programming Languages (POPL)*, January 2003.

[5] H. G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4), 1978.

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

[7] 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.

[8] 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.

[9] C. Boyapati. SafeJava: A unified type system for safe programming. PhD thesis, Massachusetts Institute of Technology. In preparation.

[10] C. Boyapati, R. Lee, and M. Rinard. Safe runtime downcasts with ownership types. In *ECOOP International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*, July 2003.

[11] 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.

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

[13] C. Boyapati, B. Liskov, L. Shrira, C. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2003.

[14] 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.

[15] 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.

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

[17] D. G. Clarke. Object ownership and containment. PhD thesis, University of New South Wales, Australia, July 2001.

[18] 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.

[19] 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.

[20] D. G. Clarke and T. Wrigstad. External uniqueness is unique enough. In *European Conference for Object-Oriented Programming (ECOOP)*, July 2003.

[21] 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.

[22] 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.

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

[24] 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.

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

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

[27] M. Deters, N. Leidenfrost, 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.

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

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

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

[31] O. Gheorghioiu. Statically determining memory consumption of real-time Java threads. MEng thesis, Massachusetts Institute of Technology, June 2002.

[32] O. Gheorghioiu, A. Sălcianu, and M. C. Rinard. Interprocedural compatibility analysis for static object preallocation. In *Principles of Programming Languages (POPL)*, January 2003.

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

[34] 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 2002.

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

[36] K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Programming Language Design and Implementation (PLDI)*, June 2002.

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

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

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

[40] 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.

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

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

[43] 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.

$$
\begin{array}{rcl}
P & ::= & \mathit{def}^*\ \mathit{srkdef}^*\ e \\
\mathit{def} & ::= & \texttt{class}\ \mathit{cn}\langle \mathit{formal}+\rangle\ \texttt{extends}\ c \\
 & & \quad \texttt{where}\ \mathit{constr}^*\ \texttt{\{}\ \mathit{field}^*\ \mathit{meth}^*\ \texttt{\}} \\
\mathit{formal} & ::= & k\ \mathit{fn} \\
c & ::= & \mathit{cn}\langle \mathit{owner}+\rangle \mid \texttt{Object}\langle \mathit{owner}\rangle \\
\mathit{owner} & ::= & \mathit{fn} \mid r \mid \texttt{this} \mid \texttt{initialRegion} \mid \texttt{heap} \mid \texttt{immortal} \mid \texttt{RT} \\
\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}^*\ \texttt{\{}\ e\ \texttt{\}} \\
\mathit{effects} & ::= & \texttt{accesses}\ \mathit{owner}^* \\
\mathit{constr} & ::= & \mathit{owner}\ \texttt{owns}\ \mathit{owner} \mid \mathit{owner}\ \texttt{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} \\
 & & \quad \texttt{where}\ \mathit{constr}^*\ \texttt{\{}\ \mathit{field}^*\ \mathit{subsreg}^*\ \texttt{\}} \\
\mathit{subsreg} & ::= & \mathit{srkind}:\mathit{rpol}\ \mathit{tt}\ \mathit{rsub} \\
\mathit{srkind} & ::= & \texttt{SharedRegion} \mid \mathit{srkn}\langle \mathit{owner}^*\rangle \\
\mathit{rkind} & ::= & \texttt{Region} \mid \texttt{NoGCRegion} \mid \texttt{GCRegion} \mid \texttt{LocalRegion} \mid \\
 & & \mathit{srkind} \\
k & ::= & \texttt{Owner} \mid \texttt{ObjOwner} \mid \mathit{rkind} \mid \mathit{rkind}:\texttt{LT} \\
 & & \\
\mathit{rpol} & ::= & \texttt{LT}(\mathit{size}) \mid \texttt{VT} \\
\mathit{tt} & ::= & \texttt{NoRT} \mid \texttt{RT} \\
 & & \\
e & ::= & v \mid \texttt{let}\ v\ \texttt{=}\ e\ \texttt{in}\ \texttt{\{}\ e\ \texttt{\}} \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{RT\_fork}\ v.\mathit{mn}\langle \mathit{owner}^*\rangle\texttt{(}v^*\texttt{)} \mid \\
 & & \texttt{(RHandle}\langle r\rangle\ h\texttt{)}\ \texttt{\{}\ e\ \texttt{\}} \mid \\
 & & \texttt{(RHandle}\langle \mathit{rkind}:\mathit{rpol}\ r\rangle\ h\texttt{)}\ \texttt{\{}\ e\ \texttt{\}} \mid \\
 & & \texttt{(RHandle}\langle r\rangle\ h\ \texttt{=}\ [\texttt{new}]_{\mathrm{opt}}\ h.\mathit{rsub}\texttt{)}\ \texttt{\{}\ e\ \texttt{\}} \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} \\
\mathit{srkn} & \in & \text{shared region kind names} \\
\mathit{rsub} & \in & \text{shared subregion names}
\end{array}
$$

Figure 13: Grammar

Typing rule	Meaning
$\vdash P : t$	Program P has type t.
$P \vdash_{\text{def}} \mathit{def}$	def is a well formed class definition from program P.
$P \vdash_{\text{srkdef}} \mathit{srkdef}_j$	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.
$P; E \vdash_{\text{meth}} \mathit{meth}$	Method definition meth is well defined with respect to program P and environment E.
$P \vdash \mathit{mbr} \in c$	Class c defines or inherits "member" definition mbr. "Member" refers to a field or a method: $\mathit{mbr} = \mathit{field} \mid \mathit{meth}$.
$P \vdash \mathit{rmbr} \in \mathit{rkind}$	Shared region kind rkind defines or inherits its "region member" definition rmbr. "Region member" refers to a field or a subregion: $\mathit{rmbr} = \mathit{field} \mid \mathit{subsreg}$.
$P; E \vdash_{\text{type}} t$	t is a well formed type with respect to program P and environment E.
$P \vdash t_1 \leq t_2$	In program P, t_1 is a subtype of t_2.
$P; E \vdash_{\text{okind}} k$	k is a well formed owner kind, with respect to program P and environment E.
$P \vdash k_1 \leq_{\text{k}} k_2$	In program P, k_1 is a subkind of k_2.
$E \vdash X_2 \succeq X_1$	Effect X_1 is subsumed by effect X_2, with respect to environment E.
$E \vdash \mathit{constr}$	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.
$E \vdash \text{RKind}(o) = k$	Owner o is either a region of kind k or an object allocated in such a region.
$E \vdash_{\text{av}} \texttt{RH}(o)$	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.

Figure 14: Typing Judgments

Predicate	Meaning
WFClasses(P)	No class is defined twice and there is no cycle in the class hierarchy.
WFRegionKinds(P)	No region kind is defined twice and there is no cycle in the region kind hierarchy. In addition, no region kind has an infinite number of (transitive) subregions.
MembersOnce(P)	No class or region kind contains two fields with the same name, either declared or inherited. Similarly, no class declares two methods with the same name.
InheritanceOK(P)	The constraints of a sub-class/kind are included among the super-class/kind constraints. For overriding methods, in addition to the usual subtyping relations between the return/parameter types, the constraints of the overrider are included among the constraints of the overridden method; a similar relation holds for the effects.

Figure 15: Predicates Used in Type Checking Rules

Appendix

A. The Type System

This section formally describes our type system. To simplify the presentation of our key ideas, we describe our type system in the context of a core subset of Java known as Classic Java [28]. Our approach, however, extends to the whole of Java and other similar languages. Figure 13 presents the grammar for our core language.

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 to range over the set of owners. We assume the program source has been preprocessed by replacing each constraint "o_1 `owns` o_2" with the 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 $P; E; X; r_{cr} \vdash e : t$. 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, $E ::= \emptyset \mid E, t\ v \mid E, k\ o \mid E, o_2 \succeq_o o_1 \mid E, o_2 \succeq o_1$. r_{cr} is the current region. X must subsume the effects of e. t is the type of e.

Figure 14 presents our typing judgments. Appendix B presents the rules for these judgments. Figure 15 presents several auxiliary predicates that we use in our rules. Most of these predicates are straightforward and we omit their definitions for brevity; the only exception is *InheritanceOK*(P), which we define formally in the set of rules.

B. Rules for Type Checking

$\boxed{\vdash P : t}$

[PROG]

$$\frac{\begin{array}{c} \mathit{WFClasses}(P) \quad \mathit{WFRegionKinds}(P) \\ \mathit{MembersOnce}(P) \quad \mathit{InheritanceOK}(P) \\ P = \mathit{def}_{1..n}\ \mathit{srkdef}_{1..r}\ e \\ \forall i = 1..n,\ P \vdash_{\mathrm{def}} \mathit{def}_i \quad \forall i = 1..r,\ P \vdash_{\mathrm{srkdef}} \mathit{srkdef}_j \\ E = \emptyset, \texttt{GCRegion heap}, \texttt{SharedRegion:LT immortal}, \\ \texttt{RHandle}\langle\texttt{heap}\rangle\ \texttt{h\_heap}, \texttt{RHandle}\langle\texttt{immortal}\rangle\ \texttt{h\_immortal} \\ P; E; \texttt{world}; \texttt{heap} \vdash e : t \end{array}}{\vdash P : t}$$

$\boxed{P \vdash_{\mathrm{def}} \mathit{def}}$

[CLASS DEF]

$$\frac{\begin{array}{c} \mathit{def} = \texttt{class}\ cn\langle (k_i\ fn_i)_{i=1..n}\rangle\ \texttt{extends}\ c \\ \texttt{where}\ \mathit{constr}_{1..p}\ \{\ \mathit{field}_{1..m}\ \mathit{meth}_{1..q}\ \} \\ P = \ldots \mathit{def} \ldots \\ E = \emptyset, \texttt{GCRegion heap}, \texttt{SharedRegion:LT immortal}, \\ \texttt{RHandle}\langle\texttt{heap}\rangle\ \texttt{h\_heap}, \texttt{RHandle}\langle\texttt{immortal}\rangle\ \texttt{h\_immortal}, \\ (k_i\ fn_i)_{i=1..n}, \mathit{constr}_{1..p}, cn\langle fn_{1..n}\rangle\ \texttt{this}, (fn_i \succeq fn_1)_{i=2..n} \\ P \vdash_{\mathrm{env}} E \quad P; E \vdash_{\mathrm{type}} c \\ \forall j = 1..m,\ (\mathit{field}_j = t_j\ fd_j\ \wedge\ P; E \vdash_{\mathrm{type}} t_j) \quad \forall k = 1..q,\ P; E \vdash_{\mathrm{meth}} \mathit{meth}_k \end{array}}{P \vdash_{\mathrm{def}} \mathit{def}}$$

$\boxed{P \vdash_{\mathrm{srkdef}} \mathit{srkdef}}$

[REGION KIND DEF]

$$\frac{\begin{array}{c} \mathit{srkdef} = \texttt{regionKind}\ \mathit{srkn}\langle (k_i\ fn_i)_{i=1..n}\rangle\ \texttt{extends}\ r\ \texttt{where}\ \mathit{constr}_{1..c}\ \{\ \mathit{field}_{1..m}\ \mathit{subsreg}_{1..s}\ \} \quad P = \ldots\ \mathit{srkdef}\ \ldots \\ E = \emptyset, (k_i\ fn_i)_{i=1..n}, \mathit{constr}_{1..c}, \mathit{srkn}\langle fn_{1..n}\rangle\ \texttt{this}, (fn_i \succeq \texttt{this})_{i=1..n} \quad P \vdash_{\mathrm{env}} E \quad P; E \vdash_{\mathrm{okind}} r \\ \forall j = 1..m,\ (\mathit{field}_j = t_j\ fd_j\ \wedge\ P; E \vdash_{\mathrm{type}} t_j) \quad \forall k = 1..s,\ (\mathit{subsreg}_k = \mathit{srkind}_k{:}\mathit{rpol}_k\ tt\ \mathit{rsub}_k\ \wedge\ P; E \vdash_{\mathrm{srkind}} \mathit{srkind}_k) \end{array}}{P \vdash_{\mathrm{srkdef}} \mathit{srkdef}}$$

$\boxed{P \vdash_{\mathrm{env}} E}$

[ENV $\emptyset$]

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

[ENV v]

$$\frac{P \vdash_{\mathrm{env}} E \quad v \notin \mathit{Dom}(E) \quad P; E \vdash_{\mathrm{type}} t}{P \vdash_{\mathrm{env}} E,\ t\ v}$$

[ENV OWNER]

$$\frac{P \vdash_{\mathrm{env}} E \quad o \notin \mathit{Dom}(E) \quad P; E \vdash_{\mathrm{okind}} k}{P \vdash_{\mathrm{env}} E,\ k\ o}$$

[ENV $\succeq_o$]

$$\frac{P \vdash_{\mathrm{env}} E \quad E \vdash_{\mathrm{k}} o_1 : \texttt{ObjOwner} \quad E \vdash_{\mathrm{k}} o_2 : k}{P \vdash_{\mathrm{env}} E,\ o_1 \succeq_o o_2}$$

[ENV $\preceq$]

$$\frac{P \vdash_{\mathrm{env}} E \quad E \vdash_{\mathrm{k}} o_1 : k_1 \quad E \vdash_{\mathrm{k}} o_2 : k_2}{P \vdash_{\mathrm{env}} E,\ o_1 \succeq o_2}$$

$\boxed{P; E \vdash_{\mathrm{type}} t}$

[TYPE INT]

$$\frac{}{P; E \vdash_{\mathrm{type}} \texttt{int}}$$

[TYPE OBJECT]

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

[TYPE REGION HANDLE]

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

[TYPE C]

$$\frac{\begin{array}{c} P = \ldots\ \mathit{def}\ \ldots \\ \mathit{def} = \texttt{class}\ cn\langle (k_i\ fn_i)_{i=1..n}\rangle\ \ldots\ \texttt{where}\ \mathit{constr}_{1..c}\ \ldots \\ \forall i = 1..n,\ (E \vdash_{\mathrm{k}} o_i : k_i'\ \wedge\ P \vdash k_i' \leq_{\mathrm{k}} k_i\ \wedge\ E \vdash o_i \succeq o_1) \\ \forall i = 1..c,\ E \vdash \mathit{constr}_i[o_1/fn_1]..[o_n/fn_n] \end{array}}{P; E \vdash_{\mathrm{type}} cn\langle o_{1..n}\rangle}$$

$\boxed{P \vdash t_1 \leq t_2}$

[SUBTYPE REFL]

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

[SUBTYPE TRANS]

$$\frac{P \vdash t_1 \leq t_2 \quad P \vdash t_2 \leq t_3}{P \vdash t_1 \leq t_3}$$

[SUBTYPE CLASS]

$$\frac{\begin{array}{c} P \vdash_{\mathrm{def}} \texttt{class}\ cn\langle (k_i\ fn_i)_{i=1..n}\rangle \\ \texttt{extends}\ cn_2\langle fn_1\ o^*\rangle\ \ldots \end{array}}{P \vdash cn\langle o_{1..n}\rangle \leq cn_2\langle fn_1\ o^*\rangle[o_1/fn_1]..[o_n/fn_n]}$$

$\boxed{P; E \vdash_{\mathrm{okind}} k}$

[STANDARD OWNERS]

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

[LT REGIONS]

$$\frac{P; E \vdash_{\mathrm{okind}} \mathit{rkind}}{P; E \vdash_{\mathrm{okind}} \mathit{rkind}{:}\texttt{LT}}$$

[USER DECLARED SHARED REGION]

$$\frac{\begin{array}{c} P \vdash_{\mathrm{srkdef}} \texttt{regionKind}\ \mathit{srkn}\langle (k_i\ fn_i)_{i=1..n}\rangle\ \ldots\ \texttt{where}\ \mathit{constr}_{1..c}\ \ldots \\ \forall i = 1..n,\ (E \vdash_{\mathrm{k}} o_i : k_i' \quad P \vdash k_i' \leq_{\mathrm{k}} k_i) \\ \forall i = 1..c,\ E \vdash \mathit{constr}_i[o_1/fn_1]..[o_n/fn_n] \end{array}}{P; E \vdash_{\mathrm{okind}} \mathit{srkn}\langle o_{1..n}\rangle}$$

$\boxed{P \vdash k_1 \leq_{\mathrm{k}} k_2}$

[SUBKIND REFL]

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

[SUBKIND TRANS]

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

[SUBKIND OWNER]

$$\frac{k \in \{\texttt{ObjOwner}, \texttt{Region}\}}{P \vdash k \leq_{\mathrm{k}} \texttt{Owner}}$$

[SUBKIND REGION]

$$\frac{k \in \{\texttt{GCRegion}, \texttt{NoGCRegion}\}}{P \vdash k \leq_{\mathrm{k}} \texttt{Region}}$$

[SUBKIND NOGCREGION]

$$\frac{k \in \{\texttt{LocalRegion}, \texttt{SharedRegion}\}}{P \vdash k \leq_{\mathrm{k}} \texttt{NoGCRegion}}$$

[SUBKIND VALUE]

$$\frac{}{P \vdash \texttt{Value} \leq_{\mathrm{k}} k}$$

[SUBKIND SHARED REGION KIND]

$$\frac{P \vdash_{\mathrm{srkdef}} \texttt{regionKind}\ \mathit{srkn}\langle (k_i\ fn_i)_{i=1..n}\rangle\ \texttt{extends}\ r\ \ldots}{P \vdash \mathit{srkn}\langle o_1..o_n\rangle \leq_{\mathrm{k}} r[o_1/fn_1]..[o_n/fn_n]}$$

[DELETE LT]

$$\frac{}{P \vdash \mathit{rkind}{:}\texttt{LT} \leq_{\mathrm{k}} \mathit{rkind}}$$

[ADD LT]

$$\frac{P \vdash \mathit{rkind}_1 \leq_{\mathrm{k}} \mathit{rkind}_2}{P \vdash \mathit{rkind}_1{:}\texttt{LT} \leq_{\mathrm{k}} \mathit{rkind}_2{:}\texttt{LT}}$$

$\boxed{P; E \vdash_{\mathrm{meth}} \mathit{meth}}$

[METHOD]

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

$\boxed{P \vdash \mathit{mbr} \in c, \text{ where } \mathit{mbr} = \mathit{field} \mid \mathit{meth}}$

[DECLARED CLASS MEMBER]

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

[INHERITED CLASS MEMBER]

$$\frac{P \vdash_{\mathrm{def}} \texttt{class}\ cn_2\langle (k_i\ fn_i')_{i=1..m}\rangle\ \texttt{extends}\ cn\langle o_{1..n}\rangle \ldots \quad P \vdash \mathit{mbr} \in cn\langle fn_{1..n}\rangle}{P \vdash \mathit{mbr}[o_1/fn_1]..[o_n/fn_n] \in cn_2\langle fn'_{1..m}\rangle}$$

$\boxed{P \vdash \mathit{rmbr} \in \mathit{rkind}, \text{ where } \mathit{rmbr} = \mathit{field} \mid \mathit{subsreg}}$

[DECLARED REGION MEMBER]

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

[INHERITED REGION MEMBER]

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

$\boxed{E \vdash \mathit{constr}}$

[ENV CONSTR]

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

[$\succeq_o$ world]

$$\frac{o \neq \texttt{RT}}{E \vdash \texttt{world} \succeq o}$$

[$\succeq_o$ OWNER]

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

[$\succeq_o$ 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]

$$\frac{o_1 \in \{\texttt{heap}, \texttt{immortal}\}}{E \vdash o_1 \succeq o_2}$$

[$\succeq$ REFL]

$$\frac{}{E \vdash o \succeq o}$$

[$\succeq$ TRANS]

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

$\boxed{E \vdash X_1 \succeq X_2}$

[$X_1 \succeq X_2$]

$$\frac{\forall o \in X_1,\ \exists g \in X_2,\ E \vdash o \succeq g}{E \vdash X_1 \succeq X_2}$$

$\boxed{E \vdash_{\mathrm{k}} o : k}$

[OWNER THIS]

$$\frac{E = E_1, cn\langle\ldots\rangle\ \texttt{this}, E_2}{E \vdash_{\mathrm{k}} \texttt{this} : \texttt{Owner}}$$

[OWNER FORMAL]

$$\frac{E = E_1, k\ o, E_2}{E \vdash_{\mathrm{k}} o : k}$$

$\boxed{E \vdash_{\text{av}} \texttt{RH}(o)}$

[AV HANDLE]

$$\frac{E = E_1, \texttt{RHandle}\langle r\rangle\ h, E_2}{E \vdash_{\text{av}} \texttt{RH}(r)}$$

[AV THIS]

$$\frac{E = E_1, cn\langle o_{1..n}\rangle\ \texttt{this}, E_2}{E \vdash_{\text{av}} \texttt{RH}(\texttt{this})}$$

[AV TRANS1]

$$\frac{E \vdash o_1 \succeq_o o_2 \quad E \vdash_{\text{av}} \texttt{RH}(o_2)}{E \vdash_{\text{av}} \texttt{RH}(o_1)}$$

[AV TRANS2]

$$\frac{E \vdash o_2 \succeq_o o_1 \quad E \vdash_{\text{av}} \texttt{RH}(o_2)}{E \vdash_{\text{av}} \texttt{RH}(o_1)}$$

$\boxed{E \vdash \text{RKind}(o) = k}$

[RKIND THIS]

$$\frac{\begin{array}{c} E = E_1, cn\langle o_{1..n}\rangle\ \texttt{this}, E_2 \\ E \vdash \text{RKind}(o_1) = k \end{array}}{E \vdash \text{RKind}(\texttt{this}) = k}$$

[RKIND FN1]

$$\frac{\begin{array}{c} E \vdash_{\text{k}} fn : k \\ k \notin \{\texttt{Owner}, \texttt{ObjOwner}\} \end{array}}{E \vdash \text{RKind}(fn) = k}$$

[RKIND FN2]

$$\frac{\begin{array}{c} E \vdash_{\text{k}} fn : k \quad k \in \{\texttt{Owner}, \texttt{ObjOwner}\} \\ E \vdash o \succeq_o fn \quad E \vdash \text{RKind}(o) = k_2 \end{array}}{E \vdash \text{RKind}(fn) = k}$$

$\boxed{\mathit{InheritanceOK}(P)}$

[INHERITANCEOK PROG]

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

[INHERITANCEOK REGION KIND]

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

[INHERITANCEOK CLASS]

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

[OVERRIDESOK METHOD]

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

$\boxed{P; E; X; r_{cr} \vdash e : t}$

[EXPR VAR]

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

[EXPR LET]

$$\frac{\begin{array}{c} P; E; X; r_{cr} \vdash e_1 : t_1 \quad E' = E, t_1\ v \\ P \vdash_{\text{env}} E' \quad P; E'; X; r_{cr} \vdash e_2 : t_2 \end{array}}{P; E; X; r_{cr} \vdash \texttt{let}\ v\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : t_2}$$

[EXPR NEW]

$$\frac{\begin{array}{c} P; E \vdash_{\text{type}} c \quad c = cn\langle o_{1..n}\rangle \\ E \vdash_{\text{av}} \texttt{RH}(o_1) \quad E \vdash X \succeq o_1 \end{array}}{P; E; X; r_{cr} \vdash \texttt{new}\ c : c}$$

[EXPR REF READ]

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

[EXPR REF WRITE]

$$\frac{\begin{array}{c} P; E; X; r_{cr} \vdash v_1 : cn\langle o_{1..n}\rangle \\ P \vdash (t\ fd) \in cn\langle fn_{1..n}\rangle \\ t' = t[o_1/fn_1]..[o_n/fn_n] \\ P; E; X; r_{cr} \vdash v_2 : t_2 \quad P \vdash t_2 \leq t' \\ (t' = \texttt{int})\ \vee \\ (t' = cn'\langle o'_{1..m}\rangle\ \wedge\ E \vdash X \succeq o'_1) \end{array}}{P; E; X; r_{cr} \vdash v_1.fd = v_2 : t'}$$

[EXPR INVOKE]

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

[EXPR REGION]

$$\frac{\begin{array}{c} P; E \vdash_{\text{okind}} \mathit{rkind} \quad \mathit{rkind} = \mathit{srkn}\langle\rangle \\ k_r = \begin{cases} \mathit{rkind}:\texttt{LT}\ \text{if}\ \mathit{rpol} = \texttt{LT}(\mathit{size}) \\ \mathit{rkind} \quad \text{otherwise} \end{cases} \\ E_2 = E, k_r\ r, \texttt{RHandle}\langle r\rangle\ h \\ E_3 = \begin{cases} E_2, (r_e \succeq r)_{\forall r_e \in \mathit{Regions}(E)} \\ \quad \text{if} P \vdash \mathit{rkind} \leq_{\text{k}} \texttt{LocalRegion} \\ E_2\ \text{otherwise} \end{cases} \\ P \vdash_{\text{env}} E_3 \quad P; E_3; X, r; r \vdash e : t \quad E \vdash X \succeq \texttt{heap} \end{array}}{P; E; X; r_{cr} \vdash \texttt{(RHandle}\langle \mathit{rkind}:\mathit{rpol}\ \ r\rangle\ \ h\texttt{)}\ \{e\} : \texttt{int}}$$

[EXPR SUBREGION]

$$\frac{\begin{array}{c} P; E; X; r_{cr} \vdash h_2 : \texttt{RHandle}\langle r_2\rangle \quad E \vdash_{\text{k}} r_2 : \mathit{srkn}_2\langle o_{1..n}\rangle \\ P \vdash \mathit{rkind}_3 : \mathit{rpol}\ \ \mathit{tt}\ \ \mathit{rsub} \in \mathit{srkn}_2\langle fn_{1..n}\rangle \\ \mathit{rkind} = \mathit{rkind}_3[o_1/fn_1]..[o_n/fn_n][r_2/\texttt{this}] \\ k_r = \begin{cases} \mathit{rkind}:\texttt{LT}\ \text{if}\ \mathit{rpol} = \texttt{LT}(\mathit{size}) \\ \mathit{rkind} \quad \text{otherwise} \end{cases} \\ E_2 = E, k_r\ r, \texttt{RHandle}\langle r\rangle\ h, r_2 \succeq r \quad P \vdash_{\text{env}} E_2 \\ P; E_2; X, r; r \vdash e : t \\ (\texttt{new}\ \vee\ (\mathit{rpol} = \mathit{VT})\ \vee\ (\mathit{tt} = \texttt{NoRT}))\ \rightarrow\ E \vdash X \succeq \texttt{heap} \\ (\mathit{tt} = \texttt{RT})\ \rightarrow\ E \vdash X \succeq \texttt{RT} \end{array}}{P; E; X; r_{cr} \vdash \texttt{(RHandle}\langle r\rangle\ \ h_1\ \texttt{=}\ [\texttt{new}]_{\text{opt}}\ \ h_2.\mathit{rsub}\texttt{)}\ \{e\} : \texttt{int}}$$

$\mathit{Regions}(E)$ is the set of all regions mentioned in E, defined as follows:

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

[EXPR LOCALREGION]

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

[EXPR FORK]

$$\frac{\begin{array}{c} P; E; X \setminus \{\texttt{RT}\}; r_{cr} \vdash v_0.mn\langle o_{(n+1)..m}\rangle(v_{1..u}) : t \\ \mathit{NonLocal}(k) \stackrel{def}{=} (P \vdash k \leq_{\text{k}} \texttt{SharedRegion})\ \vee\ (P \vdash k \leq_{\text{k}} \texttt{GCRegion}) \\ E \vdash \text{RKind}(r_{cr}) = k_{cr} \quad \mathit{NonLocal}(k_{cr}) \\ P; E; X; r_{cr} \vdash v_0 : cn\langle o_{1..n}\rangle \\ \forall i = 1..m,\ (E \vdash \text{RKind}(o_i) = k_i\ \wedge\ \mathit{NonLocal}(k_i)) \end{array}}{P; E; X; r_{cr} \vdash \texttt{fork}\ \ v_0.mn\langle o_{(n+1)..m}\rangle(v_{1..u}) : \texttt{int}}$$

[EXPR RTFORK]

$$\frac{\begin{array}{c} X' = \{o \in X \mid E \vdash \text{RKind}(o) = k\ \wedge\ P \vdash k \leq_{\text{k}} \texttt{SharedRegion}:\texttt{LT}\} \\ P; E; X', \texttt{RT}; r_{cr} \vdash v_0.mn\langle o_{(n+1)..m}\rangle(v_{1..u}) : t \\ P; E; X; r_{cr} \vdash v_0 : cn\langle o_{1..n}\rangle \\ \forall i = 1..m,\ (E \vdash \text{RKind}(o_i) = k_i\ \wedge\ P \vdash k_i \leq_{\text{k}} \texttt{SharedRegion}) \\ E \vdash \text{RKind}(r_{cr}) = k_{cr} \quad P \vdash k_{cr} \leq_{\text{k}} \texttt{SharedRegion} \end{array}}{P; E; X; r_{cr} \vdash \texttt{RT\_fork}\ \ v_0.mn\langle o_{(n+1)..m}\rangle(v_{1..u}) : \texttt{int}}$$

[EXPR GET REGION FIELD]

$$\frac{\begin{array}{c} P; E; X; r_{cr} \vdash h : \texttt{RHandle}\langle r\rangle \quad E \vdash_{\text{k}} r : \mathit{srkn}\langle o_{1..n}\rangle \\ P \vdash t\ fd \in \mathit{srkn}\langle fn_{1..n}\rangle \quad t' = t[o_1/fn_1]..[o_n/fn_n][r/\texttt{this}] \\ ((t' = \texttt{int})\ \vee\ (t' = cn\langle o'_{1..m}\rangle\ \wedge\ E \vdash X \succeq o'_1)) \end{array}}{P; E; X; r_{cr} \vdash h.fd : t'}$$

[EXPR SET REGION FIELD]

$$\frac{\begin{array}{c} P; E; X; r_{cr} \vdash h : \texttt{RHandle}\langle r\rangle \quad E \vdash_{\text{k}} r : \mathit{srkn}\langle o_{1..n}\rangle \\ P \vdash t\ fd \in \mathit{srkn}\langle fn_{1..n}\rangle \quad t' = t[o_1/fn_1]..[o_n/fn_n][r/\texttt{this}] \\ P; E; X; r_{cr} \vdash v : t_1 \quad P \vdash t_1 \leq t' \\ ((t' = \texttt{int})\ \vee\ (t' = cn\langle o'_{1..m}\rangle\ \wedge\ E \vdash X \succeq o'_1)) \end{array}}{P; E; X; r_{cr} \vdash h.fd\ \texttt{=}\ v : t'}$$