

CERIAS Tech Report 2008-32
Implicit ownership types for memory management
by Jan Vitek
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

Implicit Ownership Types for Memory Management

Tian Zhao¹, Jason Baker², James Hunt³, James Noble⁴, and Jan Vitek²

(1) University of Wisconsin – Milwaukee, USA

(2) Purdue University, West Lafayette, USA

(3) aicas GmbH, Karlsruhe, DE

(4) Victoria University, Wellington, NZ

Abstract. The Real-time Specification for Java (RTSJ) introduced a range of language features for explicit memory management. While the RTSJ gives programmers fine control over memory use and allows linear allocation and constant-time deallocation, the RTSJ relies upon dynamic runtime checks for safety making it unsuitable for safety critical applications. We introduce ScopeJ, a statically-typed, multi-threaded, object calculus in which scopes are first class constructs. Scopes reify allocation contexts and provide a safe alternative to automatic memory management. Safety follows from the use of an ownership type system that enforces a topology on run-time patterns of references. ScopeJ’s type system is novel in that ownership annotations are *implicit*. This substantially reduces the burden for developers and increases the likelihood of adoption. The notion of implicit ownership is particularly appealing when combined with pluggable type systems, as one can apply different type constraints to different components of an application depending on the requirements without changing the source language. In related work we have demonstrated the usefulness of our approach in the context of highly-responsive systems and stream processing.

1 Introduction

The Real-Time Specification for Java (RTSJ) [8] was designed to adapt Java for use in real-time applications. Safety critical applications require an exceedingly rigorous validation and certification process. For instance, the aviation industry DO-178B standard levels A and B require stringent guarantees of correctness of both the application software and, in the case of Java, the virtual machine. A tighter and smaller Java standard is needed to support these applications through the validation and certification process. A new specification request (JSR-302) based on the RTSJ, has been initiated within the Java community process to create a specification containing only the minimal features necessary for safety critical systems capable of certification [26].

Our work focuses on the most contentious part of the design of real-time Java in terms of program correctness and certification, namely the memory management subsystem. The dynamically checked region-based memory model of the RTSJ — based on dynamically scoped allocation contexts and runtime tests on assignments and region entry — has been singled out as one of the most egregious source of programmer errors in real-time Java programs¹. Current real-time garbage collectors do not match the performance or latency of region-based memory management [31] and verified garbage collector implementations are unlikely in the medium term. Thus, the best route for

¹ As reported by users and vendors in personal communications.

safety-critical certification may well be a statically-typed region-based memory management programming model. This has the advantage that the runtime infrastructure for region-based allocation is relatively simple, and that the type system is amenable to formal verification.

Our goal is to design a type system for region-based memory management that meets the following pragmatic constraints: (1) ensure that no dynamic memory access error can occur at runtime; (2) require no changes to the Java language, standard libraries or virtual machine, (3) do not modify the semantics of real-time programs. Additionally, the type system needs to be simple and lightweight, the burden imposed by extra type declarations should be minimal.

In previous work, we presented Scoped Types, a type system for RTSJ’s nested memory regions [42]. That type system overloads Java’s static package nesting to model dynamic memory region nesting: instances of scoped classes are allocated within regions corresponding to their Java package. Unfortunately, we have found Scoped Types to have a number of drawbacks. First, they hinder reuse. As a class’s package determines its allocation context, classes must be textually duplicated in different packages if their instances are to be used in more than one region. Another implication of using packages was that most standard library classes are disallowed because casting to or from `Object` is not well-typed. Such casts lose information about the source object’s allocation context. Furthermore, Scoped Types cannot cope with Java features such as primitive array types, which are crucial to practical real-time programming. And finally, the static and dynamic protection models were tightly coupled, so that object references could not be passed as method arguments across region boundaries – even when such temporary references would be perfectly safe.

In this paper, we present ScopeJ, a successor system that is simple, expressive, and, we believe, well suited to be the basis for the upcoming Safety Critical Java standard. ScopeJ includes a range of novel constructs that address the above mentioned drawbacks. First, ScopeJ supports *reusable classes* that can be used across different memory regions, reducing the need for duplication of classes. By treating types such as `Object` as reusable classes, ScopeJ allows many uses of existing library classes that themselves rely on `Object`. Second, ScopeJ directly incorporates primitive types like arrays, by treating them with a generalization of the reusable classes technique. Third, by supporting *reference borrowing*, ScopeJ separates permanent references from temporary references. Finally, the formal model covers all the essential memory and thread-related features used in our implementation, and guarantees that all memory access will be safe in the resulting real-time system.

The key technical insight underlying ScopeJ is a clear treatment of *implicit ownership polymorphism*. Most existing ownership type systems are based on *explicit ownership polymorphism*, where generic parameters of one kind or another carry ownership or region information around the program, and programs must be annotated to declare and initialize those parameters [17, 7, 1, 14, 21, 11, 9, 33]. On the other hand, simpler Confined Type systems (including our earlier Scoped Types) avoid the need for parameterization by confining all instances of a class to the same context — unfortunately these systems generally sacrifice any ownership polymorphism [22, 15, 43]. Implicit ownership polymorphism, as embodied in ScopeJ’s reusable classes, allows classes to be used

polymorphically in different ownership contexts, but without any explicit ownership parameter declaration or instantiation. ScopeJ’s type system is the first to describe implicit ownership polymorphism, and the first to be proved safe and sound.

Earlier work presented an empirical evaluation of Scoped Types on small but realistic applications [42, 2, 3]. STARS [2], notably, implemented Scoped Types with a combination of aspect-oriented programming and pluggable type checking [4] without changing the Java language’s syntax or its tool chain (IDEs, compilers, etc.). We were able to show that a 24 KLoc real-time Java application could be easily refactored to abide by the Scoped Type programming discipline. The data presented in [2] showed that refactoring the application led to an increase of approximately 1000 lines of user code. Furthermore, as [2] did not support reusable classes, it was necessary to duplicate 8 KLoc of collection classes. The refactored application exhibited better performance and predictability than the original program.

We have since developed two other systems that rely on variants of the ScopeJ type system [35, 34, 5] to offer a programming model for highly-responsive real-time systems and stream processing. In each case, the type constraints were defined as a pluggable type system with no user annotations. We evaluated usability by implementing several programs as well as a number of micro-benchmarks. Our conclusion was that implicit ownership type systems are practical and have the potential to be adopted widely in applications that require some form of control over the topology of object graphs.

The remainder of the paper is structured as follows. Section 2 introduces ownership types and their application to memory management. Section 3 illustrates the ScopeJ programming model with a number of examples and design patterns. The main contribution of this paper is in Section 4 where the ScopeJ calculus is given a formal dynamic and static semantics and soundness is proven. Section 5 puts our work in the broader context of other type-based approaches to memory management and Section 6 concludes.

2 Ownership types and Region-based Memory Management

This section serves as a primer on ownership and an introduction of region-based memory management. We start with an explicit ownership type modeled on the work of Chin et al. [11].

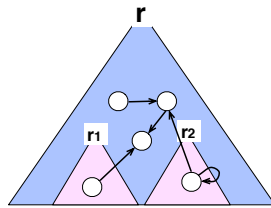


Fig. 1. A region hierarchy with safe object references.

Region-based memory management has been investigated in the context of functional [37], imperative [21] and object-oriented [9] languages. A region is a bounded pool of memory locations that can be used to satisfy allocation requests. Unlike traditional memory management interfaces, allocation does not have to be matched with deallocation directives, instead the entire region can be deallocated in a single, constant-time, step. In order to achieve fine-grained control over the lifetime of region-allocated data, regions can be nested with the semantics that a nested

region’s lifetime is strictly shorter than that of its enclosing region. The combination of

regions and references allows for programming errors where a program attempts to follow a reference to a previously deallocated object (a *dangling pointer*). This typically results in memory corruption and eventually application crashes. Figure 1 illustrates a hierarchy with three regions and ‘safe’ references, i.e. ones that cannot lead to dangling pointer errors.

Ownership type systems can be seen as enforcing a topology on the patterns of references that can occur at run-time such that unsafe references can simply never arise. The invariant they maintain is that an object can never refer to another object which has a possibly shorter lifetime. This is a sufficient condition to ensure correctness. Ownership type systems work by assigning different types to subsets of the heap and perform a kind of static escape analysis that ensures that a reference to a value allocated in one region cannot flow to a region with a shorter lifetime. Superficially most ownership type systems bear a certain resemblance to generic types. Classes, methods and references are parameterized by regions. We shall demonstrate the basics of ownership with a simple example from [11].

Consider a class representing a pair of object references. Assume that, at first, the developer wants to ensure that instances of `Pair` can be used when the pair and the values it refers to are both allocated in the same region. Figure 2 illustrates this example with a class parameterized by a single region r . As the types suggest all objects are allocated in the same region. Notice also the signature of the method, `setA()` $\langle r \rangle$, the parameter after the argument list indicates the region from which this method can be invoked. Ignoring the region subtyping of [11], this class definition allows the reuse of the `Pair` class in any region as long as the fields `a` and `b` refer to co-located objects (i.e. allocated in the same scope as `this`).

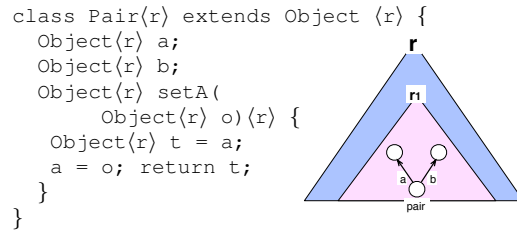


Fig. 2. Instances of `Pair` and targets co-located.

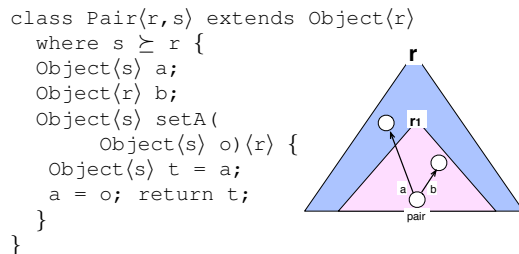


Fig. 3. Field `a` can refer to a parent region.

s , is declared to be either the same region or a parent region of r (by the means of a *where* clause and a region nesting constraint). The method `setA()` must now take an object allocated in s and return the same. Of course, it is still possible to use the class within the same region by simply setting r and s to the same actual region identifier.

Imagine that the programmer now wants to generalize the class to allow its use in contexts where the first field of the pair may refer to an object allocated in a parent region of the region that holds the `Pair` instance. This can be achieved by an additional type parameter on the definition of the class as shown by Figure 3. This new region argument,

The improved definition is not fully general as it still constrains one of the fields to be co-located. We now show the fully general definition (Figure 4) of the class as inferred by [11]. The class must take three region parameters, one for each field and one for `this`. Both field regions must be either the equal or parents of the region holding the `Pair` object. We also give the example of a method that performs allocation as it highlights the expressive power of the inferred typing. Method `clone()` will return a copy of the `Pair` allocated in the *current* region, r' , which need not be nested in the region holding the receiver. But it must be the case that the region given for the fields of the `Pair` are either equal or nested within the region where the targets were allocated. By the definition of `Pair`, it follows that r' must be equal or nested within both s and t . One could go further and define a generic version of `Pair` where the type of the fields would be parameterized both by their object type and region, but the code would become even more awkward.

In practice there are several limitations that may hinder acceptance of an explicit ownership type system such as the one shown here. We will outline the main ones. Readers may have noticed that in our examples all classes did inherit from `Object<r>` – that is to say that every class definition must now take a region parameter. This is rather unfortunate as it entails loss of backwards compatibility. One can envision an implementation of ownership that would use type

```
class Pair<r,s,t> extends Object<r>
  where s  $\succeq$  r, t  $\succeq$  r {
  Object<s> a;
  Object<t> b;
  Pair<r',s,t> clone()<r'>
    where s  $\succeq$  r', t  $\succeq$  r' {
    Pair<r',s,t> p =
      new Pair<r',s,t> ();
    p.a = a; p.b = b; return p;
  }
}
```

Fig. 4. Fully general `Pair`.

erasure and allow interoperability with non-generic libraries. In such a system, `Object<r>` would be a subtype of `Object`. The first problem is that any up-cast from a subtype of `Object<r>` to `Object` irremediably loses all ownership information, unless one is willing to add run-time generics. With type erasure, downcasts between types with different region type parameters are unsafe. Another issue with explicit annotation is the sheer syntactic weight required to make the code usable in different contexts. The added complexity and programmer effort is likely to be a deterrent to adoption. While usability is difficult to evaluate scientifically, we will simply observe that eventhough many ownership type systems have been proposed in the last decade, none is in wide use.

3 Programming with Implicit Ownership Types

We introduce our memory-safe region-based programming model with a series of examples. The code is given in Java syntax rather than the calculus of Section 4, but the essence of the examples can be translated in a straightforward fashion.

ScopeJ supports regions, or *scopes*, that are first-class values that can be entered, shared, and reclaimed. A class called `ScopeGate` identifies classes that delimit scopes — any class that inherits from `ScopeGate` is a *gate* class. Any instance of a gate class defines a new scope, owns that memory region, and is a run-time handle to that scope. Objects created by the gate's methods (or methods invoked transitively) are owned by

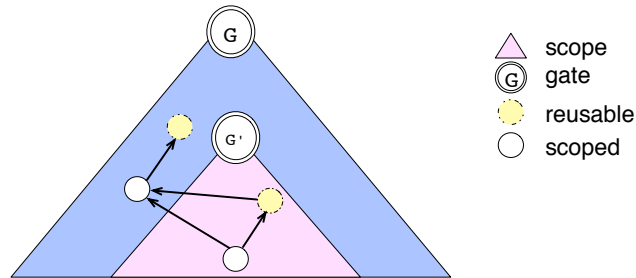


Fig. 5. ScopeJ programming model. A scope is associated with every instance of `ScopeGate`. Scoped/reusable classes are allocated within a scope. Scoped/reusable objects can refer to objects allocated in the same or parent scope. Reusable objects are only visible in their allocating scope.

that gate and are allocated within its scope. When no threads are executing within the gate, its region can be reclaimed.

To enforce memory-safety, ScopeJ programs are structured so as to make the mapping between allocation contexts and objects explicit. This is achieved by introducing the following entities into the language: *scoped classes* and *reusable classes*. Any (non-gate) class nested within the definition of a gate class is a scoped class. All instances of a scoped class are guaranteed to be allocated in the region associated with their directly enclosing gate instance. A class is deemed reusable if it can be used within any region, as opposed to scoped classes which are bound to one particular gate class. Each instance of a reusable class is owned by one particular gate instance, but different instances of a reusable class can belong to different gate instances. That is, reusable classes are ownership polymorphic — but this polymorphism is *implicit* as scope parameters are neither declared or instantiated. Figure 5 illustrates the programming model. It shows two gates G and G' and their corresponding (nested) scopes. Scoped objects are allocated within these scopes and can refer to scoped object allocated in the same or parent scope. Reusable objects are also allocated in scope but they can only be referred to from their allocating scope.

In the source code, a gate is defined by a class extending the distinguished `ScopeGate` class; nested regions are obtained by nesting of `ScopeGate` class definitions. Scoped classes are defined by nesting a Java class within a gate definition,² while reusable classes are defined outside any gate nesting hierarchy. Figure 6 gives an example of two nested gates and a scoped class.

```
class G extends ScopeGate {
  class G' extends ScopeGate {
    class MyScoped { ... }
  }
}
```

Fig. 6. Two gates and a scoped class.

3.1 Pairs revisited

Before presenting an example more relevant to real-time processing, we should revisit class `Pair` and present an implicit ownership solution. With the type system introduced in this paper the programmer has two options for writing class `Pair`, the class can be

```

class G extends ScopeGate {
  class Pair {
    Object a, b;
    Object setA(Object o) {
      Object t = a;
      a = o; return t;
    }
  }
}

```

Fig. 7. A scoped `Pair`.

```

class Pair {
  Object a, b;
  Object setA(Object o) {
    Object t = a;
    a = o; return t;
  }
}
class G extends ScopeGate { }

```

Fig. 8. A reusable `Pair`.

either scoped or reusable.

Figure 7 gives an example where the class is scoped within a gate `G`. Note that no annotation is needed for the scoped class. The visibility of `Pair` references is limited to any scoped class or gate nested within `G`. The use of `Object` is allowed and, as explained later, it implies that the target must be co-located. Thus this version corresponds to definition of Figure 2. In order to refer to a scoped type `S` allocated in an enclosing scope one would have to change the type of the field to `S`, this corresponds to Figure 3. One drawback of using a scoped class is that one may have to duplicate code. Every different use of `Pair` will require its own definition. Reusable classes give a way to circumvent this problem in many practical cases. Figure 8 illustrates a reusable `Pair` class. Observe the absence of type annotations. A reusable class is slightly more restrictive as the targets have to be co-located *and* objects of reusable class can only be accessed by other objects allocated in the same scope and not by objects in nested scopes.

The expressive power of the fully general version of `Pair` remains beyond reach of our type system. But, as we will show in the remaining examples, some `ScopeJ` idioms cannot be captured by previous region ownership type systems.

3.2 The Scoped Run Loop Pattern

Figure 9 illustrates a very common real-time programming pattern, the Scoped Run Loop [30]. In this particular example, the code periodically acquires data and processes it in a scope to ensure that any temporary data structures are reclaimed.

Observe that the memory hierarchy is apparent in the program structure. We assume an enclosing top-level scope, `Top`, and define two classes within it. `Data` is a scoped class holding inputs to the algorithm. `Processor` is a gate class used for processing each input. The `runLoop()` method periodically acquires data and processes it in an instance of `Processor`.

The body of `runLoop()` starts by allocating an instance of `Processor`. The semantics of this operation is to allocate a new object of the class and associate it with a fresh scope. Each time the `getMessage()` method is invoked from the loop, the scope associated with the processor is entered and at each return the scope is cleared. The

² While we find nesting an elegant syntactic device to assign ownership, practical system can use other means, such as annotations (see for example [35]), if the requirement to have all classes in the same file is too constraining.

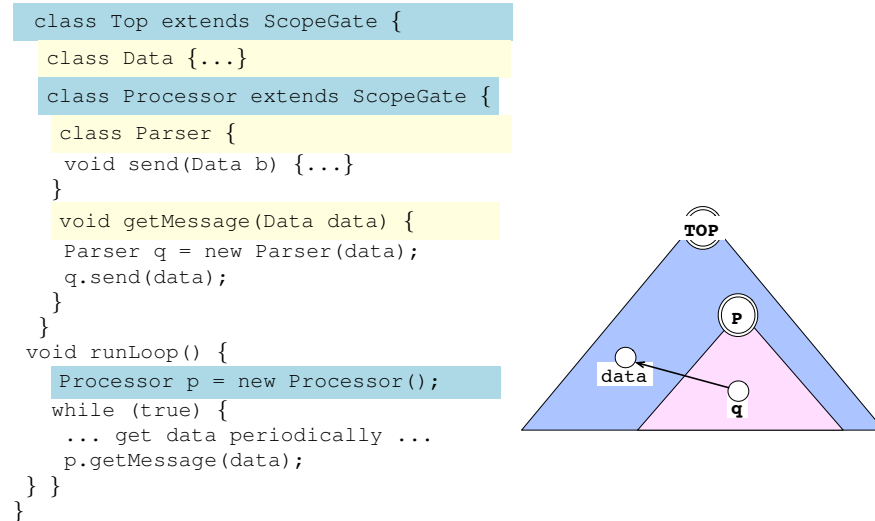


Fig. 9. Scoped Run Loop Pattern. The `runLoop` method periodically acquires data and uses a region to process it.

body of the `getMessage()` method can safely allocate knowing that all temporary data will be reclaimed. In this example, we show a scoped class `Parser` being created within the processor.

The type system allows references to the instance of `Data` from within the nested scope, but would flag any attempt to establish a reference from, e.g., `Data` to `Parser` as a compile-time error.

In order for the program to be correct, it is necessary to ensure that instances of `Parser` are allocated within the `Processor`'s region and not accessible outside it. This is achieved by nesting the `Parser` class within `Processor`. It is a type-error to return an instance of `Parser` from one of `Processor`'s methods, as this could leak a reference out of the `Processor`: when the `Processor`'s region is emptied, that would become a dangling reference which could cause memory errors.

3.3 Multi-threading

Scopes can be accessed by multiple threads. As all threads have to enter a scope through its gate, they will naturally be able to communicate by shared variables (the fields of the gate). By default, when the last thread exits a scope, all the objects within the scope are reclaimed and all the reference fields in the gate are nulled out. This is not always convenient, in some cases it would be desirable to keep the scope alive even when no thread is executing within it. An implementation may chose to add a `pin()` operation to keep a scope alive between invocations.

3.4 Limited Nesting

In the RTSJ, scopes can be nested dynamically to create arbitrary tree-like structures. While the same holds in our system, there is one difference: the depth of the tree is fixed statically to the depth of nesting of gate classes. The width of the tree, on the other hand, is not statically bounded as there can be many instances of the same gate class, each with an associated scope. The advantage of our design is that an error that requires runtime checks in the RTSJ simply cannot occur. In the RTSJ, the virtual machine must check explicitly that the programmer does not create a cycle in the scope hierarchy (a so-called `ScopeCycleException`).

Figure 10 illustrates an example of a tree of width two. This is obtained by simply instantiating the `Inner` gate twice. The cycle exceptions are prevented by the visibility rule of the type system that states that a gate type is only visible in enclosing classes. This means that, in Figure 10 `Inner` cannot be used within the body of its only method `callAlloc()`, and similarly `Top` cannot be used anywhere. Since the gate objects are hidden, it is not possible to establish a cyclic reference.

Figure 10 also illustrates an example of the Multi-Scoped Object design pattern [30] where an object is used in multiple scopes. In the RTSJ, this pattern is particularly fragile. Consider the method `alloc()` which allocates an instance of `Data`. The RTSJ semantics would be that the object is to be allocated in whichever scope happens to be current at the time the allocation request is issued. In this example, for instance, as the method is called from within the `Inner` scope, the object would be reclaimed when the inner scope is reclaimed. This makes reasoning about correctness of a multi-scoped object particularly difficult as one has to make sure that the object behaves correctly irrespective of the current allocation context.

In our proposed semantics, `Data` is always allocated in the scope associated to its defining gate, in this example `Top`. This is regardless of the current allocation context. We argue that this is safer – as one need not reason about all possible calling contexts

```
class Top extends ScopeGate {
    class Data {}
    class Multi {
        void alloc() { new Data(); }
    }
}
class Inner extends ScopeGate {
    void callAlloc(Multi m) {
        m.alloc();
    }
}
void main() {
    Inner i1 = new Inner();
    Inner i2 = new Inner();
    i1.callAlloc(new Multi());
}
```

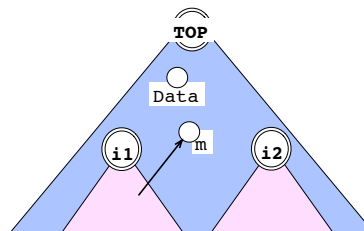


Fig. 10. The Multi Scoped Object Design Pattern.

to show `alloc()` is correct.

To preserve type-safety, the visibility of gate objects is restricted to the scope in which they are allocated. This means, for instance, that it is not possible to use references to `Inner` within its scope. If this was allowed, it would be possible to create references across different instances of the same gate class.

3.5 Reusable Classes

We have already mentioned the need to avoid code duplication. It is clear that if all classes have to appear lexically nested within a gate, classes cannot be reused across memory regions: indeed, in our earlier STARS system (without owner polymorphic classes), we found it necessary to duplicate a significant number of library classes across several scopes [3].

One solution would be to adopt explicit ownership types to provide parametric region polymorphism. ScopeJ's implicit owner polymorphism means that reusable classes can be used in different regions just *as if* they were parameterized with a single region type parameter — but this parameter remains implicit. So that the system remains sound, we must ensure that no reference to an instance of a reusable class be visible from any other scope (even a child) than the one in which it has been allocated. We have the following tradeoff: a scoped class can be allocated in only one context but is visible from all child scopes, whereas a reusable class can be allocated in any context but is visible only within its allocation context. Since a reusable class may be instantiated in any scope, the types in a reusable class must also be reusable to prevent reusable objects from referencing objects allocated in other scopes. Figure 11 illustrates the concept of reusable class with the example of the `IList` class. The class is defined outside of any scope declaration in a straightforward fashion. Instances of `IList` can be created in scope `Outer` and `Inner`. The type system will keep them distinct and ensure that a reference to a list in one scope cannot be leaked to another scope.

If a reference of reusable type crosses scope, then dangling pointers may arise. Consider the following example, where an `IList` object `outerL` is passed from outer scope to inner scope, and an instance of `IList` of inner scope `l` is assigned to the field `outerL.t`. This assignment is not safe since after the call to method `loop` returns, the object `l` is deallocated and `outerL.t` becomes a dangling pointer.

```
class IList { IList t; int value }
class Outer extends ScopeGate {
  class Inner extends ScopeGate {
    void loop(IList outerL) {
      IList l = new IList();
      l.t = outerL; // Ok
      outerL.t = l; // not OK;
      // outerL.t becomes a dangling pointer after this call returns
    }
  }
}
```

```
class IList { IList t; int value; }
class Outer extends ScopeGate {
  IList l = new IList();
  class Inner extends ScopeGate {
    void add(int i){ new IList(); }
  }
  void down() {
    new Inner().add(l.value);
  }
}
```

Fig. 11. Implicit Polymorphism.

```

void main() {
    IList l = new IList();
    (new Inner()).loop(l);
}

```

3.6 Type-safe Casts

Subtyping and type-casts are issues for ownership type systems. As types are used to track the flow of values, the type system must retain enough information to catch a breach of confinement. Operationally it is always safe to widen the type of a reference to `Object`, indeed most Java programs do it frequently, and then down-cast the object to a more precise type. Unfortunately the up-cast erases the static ownership information. There are several solutions: one can disallow widening when it entails losing ownership information (see for example [43]) or add ownership parameters to `Object` (as in [32]). Down-casts into owned types face a different problem, since most systems work by erasure, there is no runtime ownership information to check that the cast is correct.

```

class A extends ScopeGate {
    class B { }

    Object[] obj = { new B() };
    B b = (B) obj[0];
}

```

Fig. 12. Type-safe casts.

In ScopeJ, Figure 12 is well-typed. The intuition is that `Object[]` is treated as a reusable type. As such it cannot be observed from any other scope than `A`. The type system ensures any object stored in it, must have been allocated in `A`. Thus the down-cast is safe as we know that anything retrieved from a reusable class has to be locally allocated.

One of the goals of ScopeJ is to allow writing code in a style that is as natural as possible. So, for example, it is desirable to allow programs such as the one in Figure 12 where an array of objects is used to store and then retrieve scoped objects. This example is particularly important as it is the key to being able to reuse collection classes.

3.7 Borrowed References

Strict enforcement of scope-safe reference patterns is sometimes too restrictive. In RTSJ, for example, it is possible to establish read-only references that span regions with unrelated lifetimes³. This is referred to as the Hand-off design pattern in [30]. This design pattern is useful when data must be transferred between regions while avoiding copies. In our system, the same result can be obtained by relaxing the type constraints to allow temporary references between sibling scopes. Observe that references to scoped objects can be handed safely outside of their defining scope if these references are not retained and the referred scope is not deallocated. We refer to these as *borrowed references*. The annotation `local` is used to mark borrowed parameters.⁴

³ This is obtained by a combination of `executeInArea()` and `enter()` calls to navigate the scope hierarchy. The fact that it is possible to set up such references came as a surprise to the designers of the RTSJ, and complicated its implementation [29].

⁴ In our implementation this is expressed by a meta-data tag `@local` and does not require syntactic extensions.

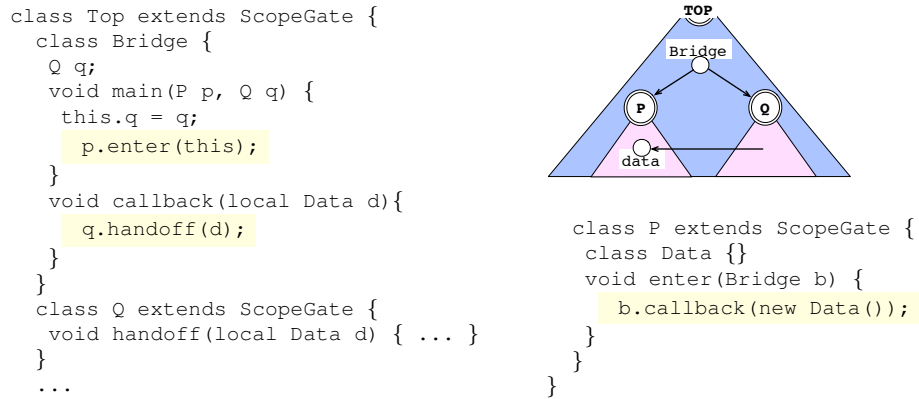


Fig. 13. Cross scope references with borrowing. Method `handoff` has access to object allocated in a sibling scope.

Figure 13 illustrates borrowing. In this scenario two scopes, referred to by instances of gates `P` and `Q` want to communicate via a shared read-only reference to an instance of `Data`. The control flow starts with the `Bridge.main()` method which takes references to the two gates. Control then enters into the scope associated with `p`, which calls back into `Bridge` passing a reference to a new `Data`. At this point, from the instance of `Bridge` we can call into the second scope with the reference to the data.

The key observation is that while method `Q.handoff()` executes there is a reference from a stack frame executing in the scope of `q` to an object allocated in the sibling scope `p`. This would be unsafe, if it was not for the fact that we are still in the *same* thread, so the target scope cannot be reclaimed. As long as we promise to release the reference when we return from `handoff()`, a dangling pointer error cannot occur. The type system ensures that borrowed references cannot be assigned into objects’ fields, and does not allow the `local` annotation to be cast away.

3.8 Comparison with Related Ownership Type Systems

This paper builds on our previous work. In [42], we introduced SJ, a core object calculus for Scoped Types. `ScopeJ` departs from, and improves on SJ in several important ways. SJ used Java packages to express memory regions and extended Java visibility rules. `ScopeJ` does not rely on Java visibility and does not impose structural restrictions. Implicitly parameterized ownership types are novel. Unlike most previous work on confined types [43], `ScopeJ` enforces object-level confinement (i.e. for scoped objects).

`ScopeJ` is also closely related to ownership type systems that use explicit parameterization [16, 13]: a gate object owns instances of classes allocated inside it. The disadvantage of ownership systems — as we’ve described above — is that they generally require new language constructs, and all class declarations and instantiations to be parameterized, imposing a relatively high syntactic overhead. This is a significant drawback and the reason we adopted the implicit approach. Classes are not parameterized explicitly; rather their position in the nested class definition hierarchy models their

instances' position in the dynamic nested regions. The key limitation of our previous attempt at implicit ownership types [42] was that classes could not be reused in different regions. Each class could only be declared in one place, so programmers had to resort to copying code. We mitigate that problem here with reusable classes: effectively they correspond to classes with a single, *implicit* existential owner parameter [27, 40]. The present approach is still limited. Instances of a reusable class are visible in a single scope, and they have only one implicit scope parameter — that is, they are associated with only one region — so that for example the contents of a `Vector` and the `Vector` itself *must* be allocated in the *same* memory space. Similarly, an array of reusable type is also reusable, which means that an array object and its contents must be in the same scope. For general purpose programming, this would be a onerous restriction. In RTSJ programs, however, this is not as difficult as it seems: the general lifetime rules against potential dangling pointers mean that such a collection can only be *shorter-lived* than its elements, so typically collections are placed within the same context.

4 The ScopeJ Calculus

The ScopeJ calculus is a core calculus for modeling region-based programs inspired by Featherweight Java (FJ) [25]. It extends FJ with mutable state, multi-threading, and memory deallocation. We follow our previous work on confined types, and adopt a call-by-value semantics with explicit evaluation contexts [43].

ScopeJ does not have explicit thread creation primitives, rather multi-threading is modeled by configurations in the dynamic semantics. Other features that have been left out include access modifiers, exceptions, and reflection. Modeling these features is interesting but mostly orthogonal to our concerns. ScopeJ does not need nested objects to obtain a reference to their enclosing scope. In Java parlance, ScopeJ classes are static inner classes. Since a gate class is not visible to its enclosed class there would be little point in providing upwards references.

```

L ::= class C extends D {  $\bar{C}$   $\bar{f}$ ;  $\bar{M}$  }
M ::= C m ( $\bar{C}$   $\bar{x}$ ) { return e; }
e ::= x | v | new C() | e.f | e.f := e | e.m( $\bar{e}$ ) | (C) e
v ::=  $\ell$  | null
S ::= top | S.s
K ::= S.c | c
C, D ::= S | K

```

Fig. 14. ScopeJ Calculus Syntax.

The ScopeJ syntax appears in Figure 14. Metavariable L ranges over class declarations. M ranges over method declarations, and f , x , and m range over field, variable

and method names. Metavariables c and s range over disjoint sets of class identifiers and gate identifiers; v is either an object reference ℓ or `null`. Classes contain field and method declarations. The distinguished `Object` class is the root of the class hierarchy. An expression e can be either a variable x (including `this`), a value v , a class or scope instantiation expression `new C()`, a field access $e.f$, a field update $e.f := e$, a method invocation $e.m(\bar{e})$, or a cast $(C) e$. We adopt FJ notation and use an over-bar to represent a finite (possibly empty) sequence. We write \bar{f} to denote the sequence f_1, \dots, f_n and similarly for \bar{e} and \bar{v} . We write $\bar{C}\bar{f}$ to denote $C_1 f_1, \dots, C_n f_n$ and $\bar{C} <: \bar{D}$ to denote $C_1 <: D_1, \dots, C_n <: D_n$.

The nesting classes are modeled with with a naming convention. A class identifier is prefixed by an ordered sequence of enclosing gate identifiers. Thus the following definition

```
class A extends ScopeGate { class B extends C { ... } }
                        desugars to
class top.A extends ScopeGate {
  class top.A.B extends C { ... }
```

Here `top` stands for the name of the top-most gate class. Class names, ranged over by C and D , are either a (possibly empty) sequence of gate identifiers terminated by a class identifier or a gate class name, S , consisting of a sequence of gate identifiers (thus using `ScopeGate` as a marker is not necessary). Reusable classes have an empty sequence of gate identifiers because they are defined outside the nested gate structure.

A number of FJ auxiliary definitions [25] are in Figure 15. Informally, $fields(C)$ returns the field declarations of C , $mtype(m, C)$ returns the type signature of method m in C , $mbody(m, C)$ returns the parameter list/body of m , and $override(m, C, D)$ is true if either m is not defined in D or the signatures of m in C and D are the same.

$$\frac{\text{class } C \text{ extends } D \{ \bar{C} \bar{f}; \bar{M} \} \quad fields(D) = (\bar{D} \bar{g})}{fields(C) = (\bar{C}\bar{D} \bar{f}\bar{g})}$$

$\frac{\text{class } C \text{ extends } D \{ \dots; \bar{M} \} \quad m \text{ not defined in } \bar{M}}{mtype(m, C) = mtype(m, D)}$	$\frac{\text{class } C \text{ extends } D \{ \dots; \bar{M} \} \quad C_0 m(\bar{C} \bar{x}) \{ \text{return } e; \} \in \bar{M}}{mtype(m, C) = \bar{C} \rightarrow C_0}$
$\frac{\text{class } C \text{ extends } D \{ \dots; \bar{M} \} \quad m \text{ not defined in } \bar{M}}{mbody(m, C) = mbody(m, D)}$	$\frac{\text{class } C \text{ extends } D \{ \dots; \bar{M} \} \quad C_0 m(\bar{C} \bar{x}) \{ \text{return } e; \} \in \bar{M}}{mbody(m, C) = (\bar{x}, e)}$
$\frac{\text{class } D \text{ extends } D' \{ \dots; \bar{M} \} \quad m \text{ not defined in } \bar{M}}{override(m, C, D)}$	$\frac{}{mtype(m, C) = mtype(m, D)}$

Fig. 15. Auxiliary definitions

$$\begin{array}{c}
\text{(R-NEW)} \\
\frac{\text{fields}(\mathbf{C}) = (\overline{\mathbf{C}} \ \overline{\mathbf{f}}) \quad \overline{\mathbf{v}} = \overline{\mathbf{null}} \quad |\overline{\mathbf{v}}| = |\overline{\mathbf{f}}|}{\ell_0 \text{ fresh} \quad \sigma' = \sigma[\ell_0 \mapsto \mathbf{C}^{\ell'}(\overline{\mathbf{v}})]} \\
\hline
\sigma, \ell, \mathbf{new} \ \mathbf{C}() \rightarrow \sigma', \ell, \ell_0
\end{array}
\quad
\begin{array}{c}
\text{(G-STEP)} \\
\frac{P = P'' \mid \kappa \bullet \ell, E[\mathbf{e}] \quad \mathbf{e} \neq \mathbf{v.m}(\overline{\mathbf{v}}) \quad \sigma, \ell, \mathbf{e} \rightarrow \sigma', \ell, \mathbf{e}' \quad P' = P'' \mid \kappa \bullet \ell, E[\mathbf{e}']}{\sigma, P \rightsquigarrow \sigma', P'}
\end{array}$$

$$\begin{array}{c}
\text{(R-FIELD)} \\
\frac{\sigma(\ell_0) = \mathbf{C}^{\ell'}(\overline{\mathbf{v}}) \quad \text{fields}(\mathbf{C}) = (\overline{\mathbf{C}} \ \overline{\mathbf{f}})}{\sigma, \ell, \ell_0.\mathbf{f}_i \rightarrow \sigma, \ell, \mathbf{v}_i}
\end{array}$$

$$\begin{array}{c}
\text{(R-UPD)} \\
\frac{\sigma(\ell_0) = \mathbf{C}^{\ell'}(\overline{\mathbf{v}}) \quad \text{fields}(\mathbf{C}) = (\overline{\mathbf{C}} \ \overline{\mathbf{f}}) \quad \sigma' = \sigma[\ell_0 \mapsto \mathbf{C}^{\ell'}(\mathbf{v} \downarrow_i \overline{\mathbf{v}})]}{\sigma, \ell, \ell_0.\mathbf{f}_i := \mathbf{v} \rightarrow \sigma', \ell, \mathbf{v}}
\end{array}
\quad
\begin{array}{c}
\text{(G-ENTER)} \\
\frac{P = P'' \mid \kappa \bullet \ell, E[\mathbf{e}] \quad \mathbf{e} = \mathbf{v.m}(\overline{\mathbf{v}}) \quad \sigma, \ell, \mathbf{e} \rightarrow \sigma, \ell', \mathbf{e}' \quad P' = P'' \mid \kappa \bullet \ell, E[\mathbf{e}] \bullet \ell', \mathbf{e}'}{\sigma, P \rightsquigarrow \sigma, P'}
\end{array}$$

$$\begin{array}{c}
\text{(R-CAST)} \\
\frac{\mathbf{v} = \mathbf{null} \vee (\sigma(\mathbf{v}) = \mathbf{C}_0^{\ell'}(\overline{\mathbf{v}}) \wedge \mathbf{C}_0 <: \mathbf{C})}{\sigma, \ell, (\mathbf{C}) \ \mathbf{v} \rightarrow \sigma, \ell, \mathbf{v}}
\end{array}
\quad
\begin{array}{c}
\text{(G-RET)} \\
\frac{P = P'' \mid \kappa \bullet \ell, E[\mathbf{e}] \bullet \ell', \mathbf{v} \quad P' = P'' \mid \kappa \bullet \ell, E[\mathbf{v}] \quad \sigma' = \text{deallocate}(\sigma, \ell', P')}{\sigma, P \rightsquigarrow \sigma', P'}
\end{array}$$

$$\begin{array}{c}
\text{(R-INVK)} \\
\frac{\sigma(\ell_0) = \mathbf{C}^{\ell_0}(\overline{\mathbf{v}}) \quad \text{mbody}(\mathbf{m}, \mathbf{C}) = (\overline{\mathbf{x}}, \mathbf{e}') \quad \mathbf{e} = [\overline{\mathbf{v}}/\overline{\mathbf{x}}, \ell_0/\text{this}]\mathbf{e}' \quad \text{inscope}_\sigma(\ell_0) = \ell'}{\sigma, \ell, \ell_0.\mathbf{m}(\overline{\mathbf{v}}) \rightarrow \sigma, \ell', \mathbf{e}}
\end{array}$$

Evaluation contexts:

$$E ::= [] \mid E.\mathbf{f} \mid E.\mathbf{f} := \mathbf{e} \mid \mathbf{v}.\mathbf{f} := E \mid E.\mathbf{m}(\overline{\mathbf{e}}) \mid \mathbf{v}.\mathbf{m}(\overline{\mathbf{v}}, E, \overline{\mathbf{e}}) \mid (\mathbf{C}) E$$

Active scope:

$$\begin{array}{l}
\text{inscope}_\sigma(\ell) = \ell \quad \text{if } \sigma(\ell) = \mathbf{S}^{\ell'}(\overline{\mathbf{v}}) \\
\text{inscope}_\sigma(\ell) = \ell' \quad \text{if } \sigma(\ell) = \mathbf{K}^{\ell'}(\overline{\mathbf{v}})
\end{array}$$

Release scope memory:

$$\frac{\text{refcount}(\ell, P) \neq 0 \quad \text{deallocate}(\sigma, \ell, P) = \sigma \quad \text{refcount}(\ell, P) = 0 \quad \sigma(\ell) = \mathbf{C}^{\ell_0}(\overline{\mathbf{v}}) \quad \{\ell_1.. \ell_n\} = \{\ell' \mid \sigma(\ell') = \mathbf{C}^{\ell'}(\overline{\mathbf{v}})\}}{\sigma' = \sigma[\ell_1 \mapsto \mathbf{dummy}, \dots, \ell_n \mapsto \mathbf{dummy}, \ell \mapsto \mathbf{C}^{\ell_0}(\overline{\mathbf{null}})] \quad \text{deallocate}(\sigma, \ell, P) = \sigma'}$$

Reference counts:

$$\begin{array}{l}
\text{refcount}(\ell, \epsilon) = 0 \\
\text{refcount}(\ell, (P \mid P')) = \text{refcount}(\ell, P) + \text{refcount}(\ell, P') \\
\text{refcount}(\ell, (\kappa \bullet \ell', \mathbf{e})) = \text{refcount}(\ell, \kappa) \quad \text{if } \ell \neq \ell' \\
\text{refcount}(\ell, (\kappa \bullet \ell, \mathbf{e})) = 1 + \text{refcount}(\ell, \kappa)
\end{array}$$

Fig. 16. Dynamic Semantics.

4.1 Dynamic Semantics

The dynamic semantics of the calculus is given in Figure 16 in terms of a two-level small-step operational semantics. A ScopeJ configuration is a pair σ, P where σ is partial map from locations to objects and a program P is a set of threads: $P \equiv \kappa$ or $P \equiv P'|P''$. The global reduction relation has thus the form, $\sigma, P \rightsquigarrow \sigma', P'$, and determines the behavior of a program as a whole. Each thread is modeled by a call stack κ which can be either empty, ϵ , or a sequence of frames. A frame is a pair, ℓ, e , of a scope and an expression. We use $\overline{\ell e}$ to represent a call stack of the form $\epsilon \bullet \ell_0, e_0 \bullet \dots \bullet \ell_n, e_n$. We assume the presence of an implicit class table with definitions for all classes.

In the global reduction rules, we assume commutative and associative thread composition so that $P|P'$ may also be written as $P'|P$. Rule (G-STEP) evaluates the top frame of the call stack. Rules (G-ENTER) and (G-RET) manage the call stack. When a thread executes $\ell.m(\overline{v})$, the thread enters the active scope of ℓ by pushing a frame onto the stack. The active scope of ℓ is itself if ℓ is a gate object, and otherwise, the active scope of ℓ is the allocation scope of ℓ . A thread removes the top of its call stack if the top frame only contains a value. When a method returns and ℓ' represents the scope instance in which the method body is evaluated, Rule (G-RET) uses the predicate $refcount(\ell', P')$ to check whether ℓ' is used by any of the threads in P' . If no thread is using ℓ' , then Rule (G-RET) clears the scope represented by ℓ' by setting all fields of ℓ' to null and replacing all objects allocated in the scope with a dummy value. Dummy values are used to differentiate access to null pointers from access to ‘dangling’ pointers. It is allowable for a computation to get stuck on access to a null reference, but the type system should ensure that dummy is never used in a receiver position.

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

The global evaluation rules rely on the notion of *evaluation contexts*, which are as usual expressions E with a hole. The syntax of method and assignment contexts enforce left-to-right evaluation order and call-by-value semantics. Furthermore evaluation contexts are deterministic. For any expression e , there is exactly one evaluation context usable in reduction rules. This can be shown by easy induction on the structure of e [43].

Expression evaluation is defined by a relation of the form $\sigma, \ell, e \rightarrow \sigma', \ell', e'$ where σ is a store mapping locations ℓ to instances and ℓ is the gate object representing the region of the allocation context. Each object, $C^{\ell}(\overline{v})$, is annotated with the instance ℓ representing the region in which it was allocated. The expression rules are mostly standard. We explain some interesting cases. In (R-NEW), the current allocation context is tagged onto the newly allocated instance. Note that we assume that the location ℓ used in the assignment is globally unique and that location is never reused. In (R-INVK), method invocation incurs locating the current allocation context ℓ' by looking up the active scope of the receiver ℓ . In (R-UPD), we write $v' \downarrow_i \overline{v}$ to denote the sequence \overline{v} with the i th entry replaced by v' .

4.2 The ScopeJ Type System

The ScopeJ type system enforces constraints on programs so that a gate is the owner of the objects allocated in the corresponding region. Rather than using explicit ownership parameters to annotate source programs, we define visibility constraints of types (similar to confined type systems) and place constraints on the use of gates to achieve the same effect of object ownership.

Visibility is critical to ScopeJ's type system. We say that type C' is visible from type C , if values of type C' can be referenced within the declaration of C . This is written $C' \text{ viz } C$. We say that S is the static scope of C if either $C = S$ or $C = S.c$. This is written as $\text{scopeof}(C)$. Reusable classes c are visible in every context. A gate type $S.s$ is visible to classes with static scope S while scoped types $S.c$ are visible to any class with static scope S or a static scope enclosed by S . The relation is defined as follows: (S' is a sequence of gate identifiers excluding `top`).

$$\frac{}{S.c \text{ viz } S} \quad \frac{}{S.c \text{ viz } S.c'} \quad \frac{}{S.c \text{ viz } S.S'} \quad \frac{}{S.c \text{ viz } S.S'.c'}$$

$$\frac{}{c \text{ viz } C} \quad \frac{}{S.s \text{ viz } S.c} \quad \frac{}{S.s \text{ viz } S}$$

We also define contextual visibility $C_0 \vdash C \text{ viz } D$ which is used to prevent reusable types from crossing scope boundaries.

$$\frac{S \text{ viz } D}{C_0 \vdash S \text{ viz } D} \quad \frac{S.c \text{ viz } D}{C_0 \vdash S.c \text{ viz } D} \quad \frac{}{c_0 \vdash c \text{ viz } D} \quad \frac{\text{scopeof}(C) = \text{scopeof}(D)}{C \vdash c \text{ viz } D}$$

The visibility rules make sure that if a thread currently in scope a enters scope b , then the type of the gate representing a must directly enclose the type of the gate representing b . This ensures that each scope has only one parent (single-parent rule) and prevents scope cycles. Consider the example below where the classes G and $G.G1$ are both gate types.

```
class G extends ScopeGate {
    void enter() {
        (new G.G1()).enter2(); // OK to enter inner scope
    }
}

class G.G1 extends ScopeGate {
    void enter2() {
        (new G()).enter(); // not OK to enter outer scope
    }
}
```

We illustrate the purpose of contextual visibility rules with the following example where G and $G.G1$ are gate classes. The method $G.access()$ calls the method $G.G1.get()$ to obtain an object of reusable type. The return value is allocated in a scope s represented by g . Assigning the result of call $g.get()$ to the field $G.f$ will result in dangling pointer if the scope g is deallocated. The call $g.get()$ is not typeable because the typing rule for method call includes a contextual visibility constraint $G.G1 \vdash \text{Object} \text{ viz } G$, which does not hold.

```

class G extends ScopeGate {
  Object f;
  void access() {
    G.G1 g = new G.G1();
    this.f = g.get();
    // not OK to reference reusable object allocated in inner scope
  };
}

class G.G1 extends ScopeGate {
  Object get() {
    return new Object();
  }
}

```

By type visibility rules, type $G.G1$ is visible from G , while type G is not visible from $G.G1$. The typing rules (explained in the next section) require that method bodies must be well-typed and consequently, new expressions in method bodies must have types visible from the enclosing classes. Thus, the expression `new G.G1()` in class G is typable while `new G()` in class $G.G1$ is not.

A type C is a subtype of D iff C is a subclass of D except that gate types may not be widened.

$$C <: C \quad \frac{C <: C' \quad C' <: C''}{C <: C''} \quad \frac{\text{class } K \text{ extends } K' \{ \dots \}}{K <: K'}$$

We define a scope-safe subtyping relation between types \preceq to restrict the use of widening. A scoped type $S.c$ may be widened to a reusable type c' in the declaration of C_0 , if the static scope of C_0 is also S .

$$\frac{}{C_0 \vdash S \preceq S} \quad \frac{S.c <: S.c'}{C_0 \vdash S.c \preceq S.c'} \quad \frac{c <: c'}{C_0 \vdash c \preceq c'} \quad \frac{S.c <: c' \quad S = \text{scopeof}(C_0)}{C_0 \vdash S.c \preceq c'}$$

We need the scope-safe subtyping relation to prevent improper widening of scoped types to reusable types. Consider the following example where G and $G.G1$ are gate types while $G.C$ and $G.G1.C$ are scoped types.

```

class G extends ScopeGate { ... }
class G.C extends Object { ... }
class G.G1 extends ScopeGate {
  Object f;
  void m(G.C v, G.G1.C v1) {
    this.f = v; // not OK to reference scoped objects in outer scope
    this.f = v1; // OK to reference scoped object in current scope
  }
}
class G.G1.C extends Object { ... }

```

The assignment `this.f = v` is not allowed since we want to maintain the invariant that the allocation scope for values of reusable type (in this case `Object`) must be the scope of the current context. In the example, the field `this.f` has reusable type and

it should reference a value allocated in a scope represented by a gate of the type $G.G1$, while $G.C$ is a scoped type with static scope G and its object must be allocated in a scope represented by a gate of the type G . The assignment $\text{this}.f = v$ is not typable since the typing rule for updates (explained next) has the constraint $G.G1 \vdash G.C \preceq \text{Object}$, which does not hold since the static scope of $G.C$ (i.e. G) is different from the static scope of $G.G1$ (i.e. $G.G1$).

Typing Classes and Methods The type rule for a class C requires that the types of all fields be visible in the context of the class definition. A scoped class can inherit from either a reusable class (including Object), or another scoped class with the same static scope. A gate class may not inherit from gate or scoped classes, and a reusable class may only inherit from another reusable class (again including Object).

$$\frac{S \text{ extends } c \vdash \bar{M} \quad \bar{C} \text{ viz } C}{\text{class } S \text{ extends } c \{ \bar{C} \bar{f}; \bar{M} \} \text{ OK}} \quad (\text{T-CLASS1})$$

$$\frac{c \text{ extends } c' \vdash \bar{M} \quad \bar{C} \text{ viz } c}{\text{class } c \text{ extends } c' \{ \bar{C} \bar{f}; \bar{M} \} \text{ OK}} \quad (\text{T-CLASS2})$$

$$\frac{S.c \text{ extends } D \vdash \bar{M} \quad \bar{C} \text{ viz } S.c \quad (D = S.c' \vee D = c')}{\text{class } S.c \text{ extends } D \{ \bar{C} \bar{f}; \bar{M} \} \text{ OK}} \quad (\text{T-CLASS3})$$

A method of class C is well-typed if its body is well-typed. In an overriding method, the signatures must match (definition elided). The argument and result types must be visible from C . Finally, the method body is of a type that is a scope-safe subtype of the declared result type.

$$\frac{\bar{x} : \bar{C}, \text{this} : C \vdash e : C'_r \quad C \vdash C'_r \preceq C_r \quad C_r \text{ viz } C \quad \bar{C} \text{ viz } C \quad \text{override}(m, C, D)}{C \text{ extends } D \vdash C_r.m(\bar{C} \bar{x}) \{ \text{return } e; \}} \quad (\text{T-METH})$$

Typing Expressions A ScopeJ expression e is type-checked in the type environment Γ , written $\Gamma \vdash e : C$. The type of a variable x is given by the environment and its type must be visible from the type of this .

$$\frac{\Gamma(x) \text{ viz } \Gamma(\text{this})}{\Gamma \vdash x : \Gamma(x)} \quad (\text{T-VAR})$$

In FJ, the type rule of a cast expression $(C_0) e$ places no constraint on e leaving it up to the runtime to check for errors. ScopeJ has to be more restrictive. Since we need to avoid casting gates to reusable or scoped types, a gate type may not be used in a cast. For up-casts, if e has a scoped type with static scope S and the type C_0 is reusable, then the static scope of the context C is S . Similarly, for down-casts, if C_0 is a scoped type in scope S and e has a reusable type, then the static scope of the context C is S .

$$\frac{C = \Gamma(\text{this}) \quad C_0 \text{ viz } C \quad \Gamma \vdash e : K \quad C \vdash C_0 \preceq K \vee C \vdash K \preceq C_0}{\Gamma \vdash (C_0) e : C_0}$$

(T-CAST)

The type rules for object creation ensure that a scoped class can only be allocated from classes in its scope, while a gate class can only be allocated from classes in its parent scope (this is enforced by the visibility constraint).

$$\frac{C = \Gamma(\text{this}) \quad C_0 \text{ viz } C \quad (\forall S.c, C_0 = S.c \Rightarrow S = \text{scopeof}(C))}{\Gamma \vdash \text{new } C_0() : C_0} \quad (\text{T-NEW})$$

A field selection expression $e.f_i$ must abide by the normal FJ typing constraints. The type of the field C_i must be visible from C , where C is the type of this , so that the field always references objects allocated in the current scope or its outer scope. Also, we want to maintain the invariant that a variable of reusable type always references objects allocated in the current scope. Thus, if C_i is reusable, then from the condition $C_0 \vdash C_i \text{ viz } C$, either e 's type C_0 is reusable, or the static scopes of C_0 and C are the same. In both cases, e and f_i refer to objects allocated in the current scope. Note that $\text{fields}(C)$ returns the field declarations $\bar{C} \bar{f}$ of the class C . Lastly, the expression e is either well-typed or $e = \text{this}$. The latter case is used when the expression is defined in a gate class to access the field of the gate itself. Since a gate type is not visible from itself, the variable this is not well-typed. So we make this exception.

$$\frac{C = \Gamma(\text{this}) \quad \Gamma \vdash e : C_0 \vee (e = \text{this} \wedge C_0 = C) \quad \text{fields}(C_0) = (\bar{C} \bar{f}) \quad C_0 \vdash C_i \text{ viz } C}{\Gamma \vdash e.f_i : C_i} \quad (\text{T-FIELD})$$

An update expression, $e.f_i := e'$, is well-typed if the type of e' is a scope-safe subtype of the type of f_i . The conditions make sure that if the field f_i has a reusable type, then e and the object referenced in f_i may only be allocated in the current scope.

$$\frac{\Gamma \vdash e.f_i : C_i \quad \Gamma \vdash e' : C' \quad \Gamma(\text{this}) \vdash C' \preceq C_i}{\Gamma \vdash e.f_i := e' : C'} \quad (\text{T-UPD})$$

An invocation expression, $e.m(\bar{e})$, is well-typed under the condition that the return type of the method call must be visible from the current context $C = \Gamma(\text{this})$. This ensures that the call always returns objects with a lifetime at least as long as the current context. Also, the type of each argument must be a scope-safe subtype of the corresponding parameter type, which must be visible from the current context. These conditions make sure that if the parameter has a reusable type, then the argument and the receiver object must be allocated in the current scope. Finally, a method called on a gate object may not be inherited from its super class. This prevents widening this when it refers to a gate object. This restriction can be relaxed if we can make sure that inherited methods only use this for field selection and for method calls that do not themselves breach this restriction. Note that the function $mtype(m, C)$ returns the parameters types \bar{C} and the result type C_r of the method m called on an object of type C .

$$\begin{array}{c}
C = \Gamma(\text{this}) \quad \Gamma \vdash e : C_0 \vee (e = \text{this} \wedge C_0 = C) \\
mtype(m, C_0) = \bar{C} \rightarrow C_x \quad (\forall S, C_0 = S \Rightarrow m \text{ defined in } C_0) \\
\hline
\Gamma \vdash_c \bar{e} : \bar{C}' \quad C \vdash \bar{C}' \preceq \bar{C} \quad C_0 \vdash \bar{C} \text{ viz } C \quad C_0 \vdash C_x \text{ viz } C \\
\hline
\Gamma \vdash e.m(\bar{e}) : C_x
\end{array}
\quad (\text{T-INVK})$$

Taken together, the ScopeJ type rules enforce visibility constraints on types and, restrictions on how fields and methods can be used. Gate objects can only be accessed in their defining context. Scoped objects can be accessed in their defining context and from all nested classes. Reusable objects can only be accessed in the context where they are instantiated. These visibility constraints are sufficient to prevent object references from leaking to regions with potentially longer lifetimes. One of the surprising features of the type system is that a gate object is visible in its defining context but not to classes nested within it. Since a gate is only visible from its parent scope, a thread is forced to enter scopes one at a time, working its way down the scope nesting hierarchy, and so avoiding RTSJ's `ScopeCycleExceptions`.

Borrowed Parameters We include borrowed (scope-local) parameters in method declarations. A parameter declared `local C x` states that the variable `x` is a borrowed reference that may only be used for field selection, method call, and as a method argument to other formal borrowed parameters. A borrowed reference may *not* be assigned to a field [24]. Borrowed parameters can reference objects with shorter or unrelated lifetimes since they are only temporarily accessed in the current context. A gate type cannot be borrowed since calling a method of a gate object means entering the scope represented by the object and temporary access to a gate object by objects of unrelated lifetime is not safe. To see the reason, consider the following example of two gate classes `G` and `G.G1`.

```

class G extends ScopeGate {
    void main() {
        G.G1 g1 = new G.G1();
        G.G1 g2 = new G.G1();
        g1.enter(g2);
    }
}
class G.G1 extends ScopeGate {
    void enter(local G.G1 g) {
        local Object obj = g.get();
        // obj becomes dangling pointer at this point
    }
    Object get() {
        return new Object();
    }
}

```

The `G.G1.enter()` method takes a borrowed parameter `g` of the type `G.G1` and uses the call `g.get()` to obtain a value `obj`. The method `G.main()` creates two scopes represented by `g1` and `g2`, and passes scope `g2` to the scope `g1`. The scope `g2`

is deallocated immediately after the call $g.get()$ returns. Since the object referenced by obj is allocated in the scope $g2$, it becomes a dangling pointer in $g1$.

We modify the syntax for method declaration and add a meta variable T to represent types that may be scope local.

$$\begin{aligned} M &::= C\ m(\bar{T}\ \bar{x})\ \{\text{return } e;\} \\ T &::= C \mid \text{local } K \end{aligned}$$

Borrowed types are visible everywhere: $\text{local } C\ \text{viz } D$. We need to supplement the scope-safe subtyping relation with the following rules:

$$\frac{C <: D}{C_0 \vdash C \preceq \text{local } D} \quad \frac{C <: D}{C_0 \vdash \text{local } C \preceq \text{local } D}$$

and also need to ensure that a borrowed expression may not be cast to other types, a field of a borrowed object may not be updated, and that fields read from a borrowed object are themselves borrowed.

$$\frac{\Gamma \vdash e : \text{local } C_0 \quad \text{fields}(C_0) = (\bar{C}\ \bar{f}) \quad C_i \text{ is not a gate type}}{\Gamma \vdash e.f_i : \text{local } C_i} \quad (\text{T-FIELD2})$$

Finally, the return type of a method called on a borrowed object is itself borrowed, and the parameter types must be borrowed.

$$\frac{\Gamma \vdash e : \text{local } K \quad \text{mtype}(m, K) = \bar{T} \rightarrow K_0 \quad \Gamma \vdash \bar{e} : \bar{T}' \quad \Gamma(\text{this}) \vdash \bar{T}' \preceq \bar{T} \quad \bar{T} \text{ are scope local}}{\Gamma \vdash e.m(\bar{e}) : \text{local } K_0} \quad (\text{T-INVK2})$$

The above rules make sure that the references transitively reachable through a borrowed reference must be borrowed. This is similar to the read-only references of Javari [39]. The difference is that while a method called on a read-only object should not mutate the object's state, a method called on a borrowed object should only accept arguments that are treated as borrowed within the method (since borrowed arguments may have shorter lifetimes than that of the receiver, and may not be stored).

4.3 Properties

The correctness property that we are after is that a well-typed ScopeJ program should never try to access a field of an object that has been deallocated. The type system enforces a stronger property as it prevents the creation of dangling pointers altogether. In order to prove this property, we introduce a notion of well-typed programs, and show that the evaluation of a program preserves the typing and the safety invariants. We assume that all classes in the class table are well-typed.

Stack invariant Intuitively, we want to maintain the safety invariant that for each call stack κ of a thread in P , the objects referenced in the expression of each frame of κ must be allocated in a scope in the scope stack γ of κ , where the scope stack γ is defined as below.

$$\text{ScopeStack}(\epsilon) = \epsilon \quad \frac{\text{ScopeStack}(\kappa) = \gamma}{\text{ScopeStack}(\kappa \bullet \ell, \mathbf{e}) = \gamma \bullet \ell}$$

Given a store σ and a scope stack γ , a judgment of the form $\sigma, \gamma \vdash \mathbf{e}$ as defined by the rules below says that the objects accessed in the expression \mathbf{e} are allocated in scope in γ .

$$\begin{array}{c} \sigma, \gamma \vdash \text{null} \quad \sigma, \gamma \vdash \text{new } \mathbf{C}() \quad \frac{\sigma(\ell) = \mathbf{C}^{\ell'}(\bar{v}) \quad \ell' \in \gamma \quad \sigma, \gamma \vdash \mathbf{e}}{\sigma, \gamma \vdash \ell} \quad \frac{\sigma, \gamma \vdash \mathbf{e}}{\sigma, \gamma \vdash (\mathbf{C}) \mathbf{e}} \\ \\ \frac{\sigma, \gamma \vdash \mathbf{e}}{\sigma, \gamma \vdash \mathbf{e}.f_i} \quad \frac{\sigma, \gamma \vdash \mathbf{e} \quad \sigma, \gamma \vdash \mathbf{e}'}{\sigma, \gamma \vdash \mathbf{e}.f_i := \mathbf{e}'} \quad \frac{\sigma, \gamma \vdash \mathbf{e} \quad \forall i, \sigma, \gamma \vdash \mathbf{e}_i}{\sigma, \gamma \vdash \mathbf{e}.m(\bar{\mathbf{e}})} \end{array}$$

Moreover, the scope stack γ of each call stack must be well-formed so that a thread can only enter a scope that it has entered before or enter a new scope which is a child of the current scope. This invariant is defined as below.

$$\frac{\ell \text{ is the immortal scope}}{\sigma \vdash \epsilon \bullet \ell} \quad \frac{\sigma \vdash \gamma \quad \ell \in \gamma \vee (\gamma = \gamma' \bullet \ell' \wedge \sigma(\ell) = \mathbf{C}^{\ell'}(\bar{v}))}{\sigma \vdash \gamma \bullet \ell}$$

Note that we assume the existence of an unique object that represents the immortal scope. This particular scope must be at the bottom, the oldest, of each scope stack. The immortal scope only serves the purpose of allocating objects of types in the immortal scope top .

Also note that if we do not allow borrowed parameters, then we can have a stronger safety invariant for call stacks. That is, for each stack frame ℓ, \mathbf{e} of a call stack κ , we can show in addition that for any object ℓ_0 referenced in \mathbf{e} , ℓ_0 is allocated in the scope represented by ℓ or its outer scope. With borrowed parameters, this may not be true since an object in \mathbf{e} could be reduced from a borrowed parameter and be allocated in a scope of a frame below the frame ℓ, \mathbf{e} in κ .

Store invariant Also, we want to maintain the safety invariant of the store that the fields of each non-gate object ℓ can only refer to objects allocated in the allocation scope of ℓ or its outer scopes; the fields of each gate object ℓ can only refer to objects allocated in the scope represented by ℓ or its outer scopes.

We first define some helper functions to retrieve the type and the allocation scope of an object.

$$\frac{\sigma(\ell) = \mathbf{C}^{\ell'}(\bar{v})}{\text{type}_\sigma(\ell) = \mathbf{C}} \quad \frac{\sigma(\ell) = \mathbf{C}^{\ell'}(\bar{v})}{\text{scope}_\sigma(\ell) = \ell'}$$

We also define a binary operator \preceq_σ as below to order the scopes represented by gate objects so that if $\ell \preceq_\sigma \ell'$, then either $\ell = \ell'$ or ℓ represents a scope enclosed by the scope represented by ℓ' .

$$\ell \preceq_\sigma \ell \quad \frac{\text{scope}_\sigma(\ell) = \ell'}{\ell \preceq_\sigma \ell'} \quad \frac{\ell \preceq_\sigma \ell' \quad \ell' \preceq_\sigma \ell''}{\ell \preceq_\sigma \ell''}$$

We also define a function $\text{locate}_{\sigma,\ell}(\mathbf{C})$ to retrieve the scope where an object of type \mathbf{C} should be allocated, provided that the store is σ and the scope of the current allocation context is represented by ℓ .

$$\text{locate}_{\sigma,\ell}(c) = \ell \quad \frac{\text{type}_{\sigma}(\ell) = \mathbf{S}}{\text{locate}_{\sigma,\ell}(\mathbf{S}.s) = \ell} \quad \frac{\ell \preceq_{\sigma} \ell' \quad \text{type}_{\sigma}(\ell') = \mathbf{S}}{\text{locate}_{\sigma,\ell}(\mathbf{S}.c) = \ell'}$$

$$\text{locate}_{\sigma,\ell}(\text{local } \mathbf{C}) = \text{locate}_{\sigma,\ell}(\mathbf{C})$$

In other words, an object of reusable type or gate type is always allocated in the current scope. Also, an object of scoped type \mathbf{C} is allocated in a scope that equals to or encloses the current scope ℓ so that the static scope of \mathbf{C} must be the same as the static scope of the type of ℓ . Note that the function is not defined if its conditions are not met.

Well-typed store: The definition of well-typed store also enforces the store invariant discussed earlier. We assume that a store always contains a location ℓ corresponding to the unique instance of immortal scope and it has the type top . The sole purpose of the immortal scope is to allocate objects with types in the scope top .

A store is well-typed if each of its object is well-typed (written as $\sigma \vdash \ell$).

$$\frac{\forall \ell \in \text{dom}(\sigma). \sigma \vdash \ell}{\vdash \sigma}$$

The immortal scope is always well-typed and we suppose that it has the type top and it is allocated in itself.

$$\frac{\sigma(\ell) = \text{top}^{\ell}() \quad \ell \text{ is the immortal scope}}{\sigma \vdash \ell}$$

An object ℓ is well-typed in σ if each of its field v_i is either null or points to an object defined in σ so that its type is a subtype of the field type \mathbf{C}_i .

Also, if v_i is allocated in ℓ_i and $\text{inscope}_{\sigma}(\ell) = \ell'_0$, then $\text{locate}_{\sigma,\ell'_0}(\mathbf{C}_i) = \ell_i$. This constraint enforces the store invariant mentioned earlier. Note that if ℓ is a gate object then $\ell'_0 = \ell$, else ℓ is allocated in ℓ'_0 . This constraint is stronger than the store invariant since we need it along with the other constraints discussed below to prove that subject reduction preserves store and stack invariants of a program.

If ℓ has type \mathbf{C} and is allocated in ℓ_0 , then $\text{locate}_{\sigma,\ell_0}(\mathbf{C}) = \ell_0$. This requirement ensures that if \mathbf{C} is a gate type then its instance is always allocated in an instance of its parent scope; if \mathbf{C} is a scoped type in the scope \mathbf{S} , then its instance is always allocated in an instance of the scope \mathbf{S} .

$$\frac{\sigma(\ell) = \mathbf{C}^{\ell_0}(\bar{v}) \quad \text{fields}(\mathbf{C}) = (\bar{\mathbf{C}} \bar{\mathbf{f}}) \quad \text{locate}_{\sigma,\ell_0}(\mathbf{C}) = \ell_0 \quad \text{inscope}_{\sigma}(\ell) = \ell'_0}{\forall i, v_i = \text{null} \vee \sigma(v_i) = \mathbf{C}'_i{}^{\ell_i}(\bar{v}^i), \mathbf{C}'_i <: \mathbf{C}_i, \text{locate}_{\sigma,\ell'_0}(\mathbf{C}_i) = \ell_i} \sigma \vdash \ell$$

Well-typed program A program σ, P is well-typed if σ is well-typed, the call stack κ of each thread in P is well-typed (written as $\sigma \vdash \kappa$), and if the reference count of ℓ in P is zero, then no object in σ is allocated in the scope represented by ℓ . The last constraint makes sure that if a scope is not used by any thread of P , then the objects allocated in it are removed from the store.

$$\frac{\begin{array}{c} \vdash \sigma \quad P = \kappa_1 \mid \dots \mid \kappa_n \\ \sigma \vdash \kappa_1 \quad \dots \quad \sigma \vdash \kappa_n \\ \forall \ell. \text{refcount}(\ell, P) = 0 \Rightarrow \nexists \ell' \text{ such that } \text{scope}_\sigma(\ell') = \ell \end{array}}{\sigma \vdash P}$$

If a call stack κ is well-typed, then the expression in each frame of κ must be well-typed, the scope stack γ of κ is well-formed (as specified by $\sigma \vdash \gamma$), and objects referenced in the expression are allocated in scope in γ (as specified by $\sigma, \gamma \vdash e$). This enforces the stack invariants mentioned earlier.

$$\frac{\sigma, \ell \vdash e : C \quad \gamma = \epsilon \bullet \ell \quad \sigma \vdash \gamma \quad \sigma, \gamma \vdash e}{\sigma \vdash \epsilon \bullet \ell, e}$$

$$\frac{\begin{array}{c} \sigma \vdash \kappa \quad \kappa = \kappa' \bullet \ell, E[e] \\ (\sigma, \ell \vdash e : C) \vee (\sigma, \ell \vdash e : \text{local } C) \quad \sigma, \ell' \vdash e' : C' \quad C' <: C \\ \gamma = \text{ScopeStack}(\kappa \bullet \ell', e') \quad \sigma \vdash \gamma \quad \sigma, \gamma \vdash e \\ \sigma, \ell \vdash_{\text{scope}} e : \ell_0 \quad \sigma, \ell' \vdash_{\text{scope}} e' : \ell_0 \end{array}}{\sigma \vdash \kappa \bullet \ell', e'}$$

The typing of an expression on stack is given by the judgment $\sigma, \ell \vdash e : T$, which holds if the expression e has type T given a store σ and a gate object ℓ representing the current scope. The typing rules for expressions during evaluation steps are shown in Figure 18 (in the Appendix) and they are similar to those for expressions in class declarations.

Note that for a call stack with at least two frames, it must have the form of $\kappa \bullet \ell', e'$, where $\kappa = \kappa' \bullet \ell, E[e]$. The expression e is a method call evaluated to e' . The type of expression e' is not scope local and it is the subtype of the type of e . This is used to prove that when the call returns, substituting e with the call result preserves the typing of $E[e]$. Also, the last condition contains judgment of the form $\sigma, \ell \vdash_{\text{scope}} e : \ell_0$ as defined in Figure 17. The judgment is used to derive the allocation scope for the value that can be evaluated from the expression e , where σ is the store and ℓ represents the scope of the current context. Intuitively, since e' is reduced from e , if e should be reduced to an object allocated in ℓ_0 , then so is e' . Again this constraint helps us to prove that subject reduction preserves the store and stack invariants and the typing of expressions.

Well-typed program does not get stuck We now show that a well-typed program can make progress until it either raises an exception due to an illegal cast or null pointer dereference, or all threads in the program have evaluated to irreducible values.

$$\begin{array}{c}
\sigma, \ell \vdash_{scope} \text{null} : \ell' \quad (\text{S-Null}) \\
\sigma, \ell \vdash_{scope} \text{new } \mathbf{C}() : \ell \quad (\text{S-New}) \\
\frac{\text{scope}_\sigma(\ell') = \ell_0}{\sigma, \ell \vdash_{scope} \ell' : \ell_0} \quad (\text{S-Var}) \\
\frac{\sigma, \ell \vdash_{scope} \mathbf{e} : \ell' \quad \text{locate}_{\sigma, \ell}(\mathbf{C}) = \ell'}{\sigma, \ell \vdash_{scope} (\mathbf{C}) \mathbf{e} : \ell'} \quad (\text{S-Cast}) \\
\frac{\sigma, \ell \vdash \mathbf{e.f}_i : \mathbf{T} \quad \mathbf{e} \neq \ell \quad \sigma, \ell \vdash_{scope} \mathbf{e} : \ell_0 \quad \text{locate}_{\sigma, \ell_0}(\mathbf{T}) = \ell'}{\sigma, \ell \vdash_{scope} \mathbf{e.f}_i : \ell'} \quad (\text{S-Field}) \\
\frac{\sigma, \ell \vdash \ell.f_i : \mathbf{C} \quad \text{locate}_{\sigma, \ell}(\mathbf{C}) = \ell'}{\sigma, \ell \vdash_{scope} \ell.f_i : \ell'} \quad (\text{S-Field2}) \\
\frac{\sigma, \ell \vdash_{scope} \mathbf{e.f}_i : \ell' \quad \sigma, \ell \vdash_{scope} \mathbf{e}' : \ell'}{\sigma, \ell \vdash_{scope} \mathbf{e.f}_i := \mathbf{e}' : \ell'} \quad (\text{S-Upd}) \\
\frac{\sigma, \ell \vdash \mathbf{e.m}(\bar{\mathbf{e}}) : \mathbf{T} \quad \mathbf{e} \neq \ell \quad \sigma, \ell \vdash_{scope} \mathbf{e} : \ell_0 \quad \text{locate}_{\sigma, \ell_0}(\mathbf{T}) = \ell'}{\sigma, \ell \vdash_{scope} \mathbf{e.m}(\bar{\mathbf{e}}) : \ell'} \quad (\text{S-Invk}) \\
\frac{\sigma, \ell \vdash \ell.m(\bar{\mathbf{e}}) : \mathbf{C} \quad \text{locate}_{\sigma, \ell}(\mathbf{C}) = \ell'}{\sigma, \ell \vdash_{scope} \ell.m(\bar{\mathbf{e}}) : \ell'} \quad (\text{S-Invk2})
\end{array}$$

Fig. 17. Allocation scope for expressions

The only possible scenario for a program to get stuck is after deallocating objects allocated in a scope (Rule (G-Ret)), there are dangling pointers in the stack or in the store. In this case, dangling pointers are locations used in the stack or in the fields of an object but the store maps these locations to dummy values. To show that there are no dangling pointers after deallocation, we only need to prove that the store and the call stacks of the program are well-typed. Therefore, a well-typed program can make progress if it has no exceptions and its threads are not all irreducible values

Lemma 1 says that the evaluation of an expression \mathbf{e} on top of a stack $\kappa \bullet \ell$, \mathbf{e} in one step preserves typing.

Lemma 1. *If $\vdash \sigma, \sigma, \ell \vdash \mathbf{e} : \mathbf{T}$, $\sigma, \ell, \mathbf{e} \rightarrow \sigma', \ell', \mathbf{e}'$, then $\sigma', \ell' \vdash \mathbf{e}' : \mathbf{T}'$, $\text{type}_\sigma(\ell) \vdash \mathbf{T}' \preceq \mathbf{T}$, and if $\mathbf{e} = \mathbf{v.m}(\bar{\mathbf{v}})$, then $\mathbf{T}' = \mathbf{C}'$.*

Lemma 2 shows that if an expression \mathbf{e} is well-typed, then we can find a scope to allocate the object reduced from \mathbf{e} . Moreover, if the type of \mathbf{e} is not scope local, then the allocation scope of \mathbf{e} can be found by searching the scope hierarchy outwards starting from the scope instance represented by ℓ , which is the scope of the current context.

Lemma 2. *If $\vdash \sigma, \sigma, \ell \vdash \mathbf{e} : \mathbf{T}$, then $\exists \ell'$ such that $\sigma, \ell \vdash_{scope} \mathbf{e} : \ell'$, and if $\mathbf{T} = \mathbf{C}$, then $\text{locate}_{\sigma, \ell}(\mathbf{C}) = \ell'$.*

Lemma 3 proves that the evaluation step of a well-typed expression preserves the typing of the store. Intuitively, we need to show that a new expression creates an object

allocated in a scope suitable for the type of the object; and an update expression only puts an object ℓ in the field of another object ℓ' if the allocation scope of ℓ matches the allocation scope that ℓ' expects for a value in the field.

Lemma 3. *If $\vdash \sigma$, $\sigma, \ell \vdash e : T$, $\sigma, \ell \vdash_{scope} e : \ell'$, and $\sigma, \ell, e \rightarrow \sigma', \ell, e'$, then $\vdash \sigma'$.*

Lemma 4 shows that if a scope stack γ is well-formed and the objects referenced in an expression e are all allocated in scopes of γ , then the allocation scope of e is in γ as well. This lemma is used to prove that evaluation steps preserve the stack invariant.

Lemma 4. *If $\sigma \vdash \gamma$, $\sigma, \gamma \vdash e$, $\gamma = \gamma' \bullet \ell$, and $\sigma, \ell \vdash_{scope} e : \ell'$, then $\ell' \in \gamma$.*

Lemma 5 shows that evaluation steps preserve the allocation scope of well-typed expressions. This lemma is also used to prove that evaluation steps preserve the stack invariant. That is, since well-typed expressions should be allocated in scopes in the scope stack and evaluation of these expressions does not change the allocation scopes, when these expressions are reduced to object values, their allocation scopes are also in the scope stack. Consequently, the stack invariant holds during evaluation steps.

Lemma 5. *If $\vdash \sigma$, $\sigma, \ell \vdash e : T$, $\sigma, \ell \vdash_{scope} e : \ell_a$, and $\sigma, \ell, e \rightarrow \sigma', \ell', e'$, then $\sigma', \ell' \vdash_{scope} e' : \ell_a$.*

The subject reduction lemma shows program execution preserves typing, the invariants of store and stacks in the program, which means that releasing memory of a scope will not create dangling pointers in the program.

Lemma 6 (Subject Reduction). *If $\sigma \vdash P$ and $\sigma, P \rightsquigarrow \sigma', P'$, then $\sigma' \vdash P'$.*

An expression has cast exception if it contains an expression of the form $(C') \ell$ where ℓ has type C but C is not a subtype of C' . An expression has null pointer exception if it contains an expression of the forms $\text{null.m}(\bar{v})$, null.f , or $\text{null.f} := v$. The lemma below proves that a well-typed expression without cast or null-pointer exceptions can be evaluated.

Lemma 7. *If $\vdash \sigma$, $\sigma, \ell \vdash e : T$, and e is not a value and it does not have null pointer or cast exception, then $\exists \ell'$ and e' , such that $\sigma, \ell, e \rightarrow \sigma', \ell', e'$.*

A thread is alive if it is not in the form of $\epsilon \bullet \ell, v$ and it does not contain a null pointer or cast exception. The lemma below proves that a well-typed program can make progress.

Lemma 8 (Progress). *If $\sigma \vdash P$ and P has at least one live thread, then $\exists \sigma', P'$ such that $\sigma, P \rightsquigarrow \sigma', P'$.*

Let $\sigma, P \uparrow$ represents that the execution of the program σ, P diverges and let \rightsquigarrow^* be the transitive closure of the one-step reduction \rightsquigarrow .

Theorem 1. *If $\sigma \vdash P$, then $\sigma, P \uparrow$ or $\sigma, P \rightsquigarrow^* \sigma', P'$, where either each thread in P' is in the form of $\epsilon \bullet \ell, v$ or P' contains null pointer or cast exceptions.*

This theorem states that if a program is well-typed, then its execution either diverges, reduces to a value, or throws exceptions. The proof of the theorem is by induction using the lemmas of subject reduction and progress.

5 Related Work

Region-based memory management was introduced by Tofte and Talpin [38] and originally implemented for the ML programming language. Region systems organize memory in a stack of regions and use a combination of polymorphism and effects to indicate the allocation context of expressions and the regions they may affect. In the ML family of region-based systems, regions are single threaded and lexically scoped. Furthermore regions are not values, they cannot be stored or shared. Straightforward extensions to Java have been investigated in [12, 41], however language features such as multi-threading do not easily fit in the lexically scoped region model. Our scope calculus could be viewed as making regions first class entities and allowing them to be entered multiple times (by different threads). Each scope can be considered as a wrapper around a `let region ρ` expression such that ρ is only in scope for definition nested within the scope. Each class can then be parameterized by a set of region parameters, one per enclosing scope. Method effects can be approximated by the set of scopes visible to the defining class.

Hallenberg et al. investigated the integration of garbage collection and regions [23], but their goal was different from ours as they wanted to garbage collect regions. Region-based memory can also be managed by reference counting regions to prevent unsafe region deletion [19]. Cyclone is a type-safe C-like language with support for lexically scoped regions [21]. The compiler does not support multi-threading, but plans for a concurrent extension have appeared [20]. In the extension, regions are still lexically scoped, and region sharing is a side-effect of spawning a thread while in a region. In ScopeJ, a thread can join a scope at anytime by invoking one of its methods. Hicks et al. report on an extension with unique pointers and a form of borrowing [36]: the unique pointers can be used to relax regions' LIFO lifetimes.

Research on ownership types dates back to a paper by Noble, Vitek and Potter [28] and was motivated by the desire to impose a structure on the "sea of objects" found in large systems. The term ownership types and the formalization of the underlying ideas is due to Clarke [13]. Since then many papers have extended the basic idea [6, 1]. Generic Universe Types [18] also use type parameters to express ownership to require the modification of an object to be initiated by its owner. Borrowed references go back at least as far as Hogg's Islands [24] and have been studied by Boyland [10]. They can also be viewed as a generalization of the concept of anonymity found in Confined Types [43]. Boyapati et al. have proposed an ownership type system for the RTSJ [9], and Chin et al. later proposed region inference for a similar language [11]. This proposal is comprehensive as it expresses all the varieties of RTSJ regions and real-time threads, however it relies on explicit ownership type parameterization where every type is parameterized by one or more region parameters. This has the drawback that existing Java code cannot be incorporated (Object would require an ownership parameter), that primitive types such as object arrays cannot be used, and that region handles must be passed around explicitly in order to determine where objects are to be allocated. ScopeJ requires no ownership parameters, region handles are implicit, and our implicit ownership polymorphism supports reusable classes and arrays and other primitive types. ScopeJ also supports borrowed references which we have found omnipresent in the RTSJ programs we have experience with. Relationships to some of our earlier work

were discussed in Section 3.8.

6 Conclusions

In this paper, we have presented ScopeJ, a simple multi-threaded object calculus with region-based memory management, supported by a novel type system that ensures safety of object deallocation. The goal of ScopeJ is to offer an alternative to the memory model of the Real-time Specification for Java, and in particular, to be a candidate for the upcoming Safety Critical Java standard. ScopeJ includes novel constructs such as classes that can be reused across memory regions, safe down-casts, and borrowed references.

The key technical insight underlying ScopeJ is a clear treatment of implicit ownership polymorphism which allows classes to be used polymorphically in different ownership contexts, but without any explicit ownership parameter declaration or instantiation. ScopeJ's type system is the first to describe implicit ownership polymorphism, and the first to be proved safe and sound.

We have implemented several variants of the ScopeJ type system with a combination of an extensible, or pluggable, type checker and aspect-oriented techniques. This has let us write, and refactor, over 20K lines of code and gain confidence in the applicability of the approach. We see the combination of pluggable types and implicit ownership as an elegant and non-intrusive way to introduce new aliasing control policies. In future work, we expect to look into making it easier for end-users to define their own ownership type disciplines.

References

1. Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Oslo, Norway, 2004. Springer-Verlag.
2. Chris Andreae, Yvonne Coady, Celina Gibbs, James Noble, Jan Vitek, and Tian Zhao. Scoped Types and Aspects for Real-Time Java. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 124–147, Nantes, France, July 2006.
3. Chris Andreae, Yvonne Coady, Celina Gibbs, James Noble, Jan Vitek, and Tian Zhao. Scoped types and aspects for real-time Java memory management. *Realtime Systems Journal*, 37(1):1–44, October 2007.
4. Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 57–74, 2006.
5. Joshua Auerbach, Jesper Spring, David Bacon, Rachid Guerraoui, and Jan Vitek. A unified restricted thread programming model for java. In *Proceedings of the ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, June 2008.
6. Anindya Banerjee and David A. Naumann. Representation independence, confinement and access control. In *Conference Record of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 166–177, 2002.
7. Boris Bokowski and Jan Vitek. Confined Types. In *Proceedings 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Denver, Colorado, USA, November 1999.
8. Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, June 2000.

9. Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2003.
10. John Boyland, James Noble, and William Retert. Capabilities for Sharing: A Generalization of Uniqueness and Read-Only. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, June 2001.
11. Wei-Ngan Chin, Florin Craciun, Shengchao Qin, and Martin Rinard. Region inference for an object-oriented language. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 243–254, May 2004.
12. Morten V. Christiansen, Fritz Henglein, Henning Niss, and Per Velschow. Safe region-based memory management for objects. Technical report, DIKU, University of Copenhagen, October 1998.
13. Dave Clarke. *Object ownership and containment*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Australia, 2002.
14. Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2002.
15. Dave Clarke, Michael Richmond, and James Noble. Saving the world from bad beans: Deployment-time confinement checking. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, October 2003.
16. David Clarke, James Noble, and John Potter. Simple ownership types for object confinement. In *Proceedings of European Conference for Object-Oriented Programming*. Springer-Verlag, 2001.
17. David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 48–64, October 1998.
18. W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, June 2007.
19. David Gay and Alex Aiken. Language support for regions. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, Snowbird, Utah, June 2001.
20. Dan Grossman. Type-safe multithreading in Cyclone. In *ACM Workshop on Types in Language Design and Implementation*, January 2003.
21. Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 282–293, Berlin, Germany, June 2002.
22. Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. *Transactions on Programming Languages and Systems*, 29(6):32–73, 2007.
23. Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, June 2002. Berlin, Germany.
24. John Hogg. Islands: Aliasing protection in object-oriented languages. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 271–285, November 1991.
25. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.
26. JSR 302. Safety critical Java technology, 2007.
27. Yi Lu and John Potter. Flexible ownership types with owner variance. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2006.
28. James Noble, John Potter, and Jan Vitek. Flexible alias protection. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP)*, Brussels, Belgium, July 1998.

29. Krzysztof Palacz and Jan Vitek. Java subtype tests in real-time. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 378–404, Darmstadt, Germany, July 2003.
30. Filip Pizlo, Jason Fox, David Holmes, and Jan Vitek. Real-time Java scoped memory: design patterns and semantics. In *Proceedings of the IEEE International Symposium on Object-oriented Real-Time Distributed Computing (ISORC)*, Vienna, Austria, May 2004.
31. Filip Pizlo and Jan Vitek. An empirical evaluation of memory management alternatives for Real-time Java. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS)*, December 2006.
32. Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Defaulting Generic Java to Ownership. In *Proceedings of the Workshop on Formal Techniques for Java-like Programs in European Conference on Object-Oriented Programming*, Oslo, Norway, June 2004. Springer-Verlag.
33. Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic ownership for generic Java. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 311–324, 2006.
34. Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. StreamFlex: High-throughput stream programming in Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, October 2007.
35. Jesper Honig Spring, Filip Pizlo, Rachid Guerraoui, and Jan Vitek. Reflexes: Abstractions for highly responsive systems. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE)*, 2007.
36. Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Safe manual memory management in Cyclone. *Science of Computer Programming*, 62:122–144, October 2006. Special issue on memory management.
37. Mads Tofte and Jean-Pierre Talpin. Data region inference for polymorphic functional languages. In *Conference Record of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 188 – 201, 1994.
38. Mads Tofte and Jean-Pierre Talpin. Region-Based Memory Management. *Information and Computation*, 132(2):109–176, 1997.
39. Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 211–230, October 2005.
40. Tobias Wrigstad and Dave Clarke. Existential owners for ownership types. *Journal of Object Technology*, May 2007.
41. Bennett Norton Yates. A type-and-effect system for encapsulating memory in Java. Technical report, MSc.Thesis, University of Oregon, August 1999.
42. Tian Zhao, James Noble, and Jan Vitek. Scoped types for real-time Java. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS)*, Lisbon, Portugal, December 2004.
43. Tian Zhao, Jens Palsberg, and Jan Vitek. Type-based confinement. *Journal of Functional Programming*, 16(1), January 2006.

Appendix

$$\begin{array}{c}
\sigma, \ell \vdash \text{null} : \mathbf{T} \quad (\text{RT-NULL}) \\
\\
\frac{\sigma(\ell_0) = \mathbf{C}^{\ell_0}(\bar{v}) \quad \text{locate}_{\sigma, \ell}(\mathbf{C}) = \ell'}{\sigma, \ell \vdash \ell_0 : \mathbf{C}} \quad (\text{RT-VAR}) \\
\\
\frac{\sigma(\ell_0) = \mathbf{K}^{\ell_0}(\bar{v}) \quad \text{locate}_{\sigma, \ell}(\mathbf{K}) \neq \ell'}{\sigma, \ell \vdash \ell_0 : \text{local } \mathbf{K}} \quad (\text{RT-VAR2}) \\
\\
\frac{\mathbf{C} \text{ viz } \text{type}_{\sigma}(\ell) \quad \sigma, \ell \vdash \mathbf{e} : \mathbf{K} \quad (\mathbf{C} = \mathbf{S}.c \wedge \mathbf{K} = \mathbf{S}'.c') \Rightarrow \mathbf{S} = \mathbf{S}' \quad ((\mathbf{C} = c \wedge \mathbf{K} = \mathbf{S}.c') \vee (\mathbf{K} = c \wedge \mathbf{C} = \mathbf{S}.c')) \Rightarrow \text{type}_{\sigma}(\ell) = \mathbf{S}}{\sigma, \ell \vdash (\mathbf{C}) \mathbf{e} : \mathbf{C}} \quad (\text{RT-CAST}) \\
\\
\frac{\mathbf{C} \text{ viz } \text{type}_{\sigma}(\ell) \quad (\mathbf{C} = \mathbf{S}.c \Rightarrow \text{type}_{\sigma}(\ell) = \mathbf{S})}{\sigma, \ell \vdash \text{new } \mathbf{C}() : \mathbf{C}} \quad (\text{RT-NEW}) \\
\\
\frac{\sigma, \ell \vdash \mathbf{e} : \mathbf{C} \vee (\mathbf{e} = \ell \wedge \mathbf{C} = \text{type}_{\sigma}(\ell)) \quad \text{fields}(\mathbf{C}) = (\bar{\mathbf{C}} \bar{\mathbf{f}}) \quad \mathbf{C} \vdash \mathbf{C}_i \text{ viz } \text{type}_{\sigma}(\ell)}{\sigma, \ell \vdash \mathbf{e}.f_i : \mathbf{C}_i} \quad (\text{RT-FIELD}) \\
\\
\frac{(\sigma, \ell \vdash \mathbf{e} : \mathbf{K}) \vee (\sigma, \ell \vdash \mathbf{e} : \text{local } \mathbf{K}) \quad \text{fields}(\mathbf{K}) = (\bar{\mathbf{C}} \bar{\mathbf{f}}) \quad \mathbf{C}_i \text{ is not a gate type}}{\sigma, \ell \vdash \mathbf{e}.f_i : \text{local } \mathbf{C}_i} \quad (\text{RT-FIELD2}) \\
\\
\frac{\sigma, \ell \vdash \mathbf{e}.f_i : \mathbf{C}_i \quad \sigma, \ell \vdash \mathbf{e}' : \mathbf{C}' \quad \text{type}_{\sigma}(\ell) \vdash \mathbf{C}' \preceq \mathbf{C}_i}{\sigma, \ell \vdash \mathbf{e}.f_i := \mathbf{e}' : \mathbf{C}'} \quad (\text{RT-UPD}) \\
\\
\frac{\sigma, \ell \vdash \mathbf{e} : \mathbf{C}_0 \vee (\mathbf{e} = \ell \wedge \mathbf{C}_0 = \text{type}_{\sigma}(\ell)) \quad \text{mtype}(\mathbf{m}, \mathbf{C}_0) = \bar{\mathbf{T}} \rightarrow \mathbf{C} \quad \sigma, \ell \vdash \bar{\mathbf{e}} : \bar{\mathbf{T}}' \quad \text{type}_{\sigma}(\ell) \vdash \bar{\mathbf{T}}' \preceq \bar{\mathbf{T}} \quad \mathbf{C}_0 \vdash \bar{\mathbf{T}} \text{ viz } \text{type}_{\sigma}(\ell) \quad \mathbf{C}_0 \vdash \mathbf{C} \text{ viz } \text{type}_{\sigma}(\ell) \quad \text{if } \mathbf{C}_0 \text{ is a gate type, then } \mathbf{m} \text{ is defined in } \mathbf{C}_0}{\sigma, \ell \vdash \mathbf{e}.m(\bar{\mathbf{e}}) : \mathbf{C}} \quad (\text{RT-INVK}) \\
\\
\frac{(\sigma, \ell \vdash \mathbf{e} : \mathbf{K}) \vee (\sigma, \ell \vdash \mathbf{e} : \text{local } \mathbf{K}) \quad \text{mtype}(\mathbf{m}, \mathbf{K}) = \bar{\mathbf{T}} \rightarrow \mathbf{K}_0 \quad \sigma, \ell \vdash \bar{\mathbf{e}} : \bar{\mathbf{T}}' \quad \text{type}_{\sigma}(\ell) \vdash \bar{\mathbf{T}}' \preceq \bar{\mathbf{T}} \quad \bar{\mathbf{T}} \text{ are scope local}}{\sigma, \ell \vdash \mathbf{e}.m(\bar{\mathbf{e}}) : \text{local } \mathbf{K}_0} \quad (\text{RT-INVK2})
\end{array}$$

Fig. 18. Runtime type rules. (Unbounded meta-variables are assumed to be universally qualified.)

Recall that in each frame ℓ, \mathbf{e} of a call stack, ℓ is always a gate instance and represents a scope. In the proof, we sometimes refer to ℓ as the scope that it represents.

PROOF OF LEMMA 1

Proof. We prove by case analysis based on the applicable reduction rules.

R-FIELD Let $\mathbf{e} = \ell_0.f_i$, $\sigma(\ell_0) = \mathbf{C}^{\ell_0}(\bar{v})$, $\text{fields}(\mathbf{C}) = (\bar{\mathbf{C}} \bar{\mathbf{f}})$, and $\Sigma \vdash_{\sigma, \ell} \ell_0 : \mathbf{T}_0$. Then, $\mathbf{e}' = v_i$.

We need to show $\sigma, \ell \vdash v_i : T'$ such that $T' \preceq T$, where $T = C_i$ or $T = \text{local } C_i$. If $v_i = \text{null}$, the proof is trivial since it can have any type.

Suppose $v_i = \ell_i$ and $\sigma(\ell_i) = C_i^{\ell'_i(\bar{v})}$. By assumption, we have $\vdash \sigma$, which implies $\sigma \vdash \ell_0$ and $\text{locate}_{\sigma, \ell_0}(C_i) = \ell_i$. Suppose $\text{inscope}_\sigma(\ell_0) = \ell'_0$ and $\sigma(\ell_i) = C_i^{\ell'_i(\bar{v})}$. Then $\text{locate}_{\sigma, \ell'_0}(C_i) = \ell'_i$.

Suppose $\sigma, \ell \vdash \ell_i : T'$. If $\text{locate}_{\sigma, \ell}(C_i) = \ell'_i$ then by Rule (RT-Var) $T' = C'_i$, else by Rule (RT-Var2) $T' = \text{local } C'_i$.

If $T_0 = \text{local } C_0$, then by Rule (RT-Field), C_i is not a gate type and $T = \text{local } C_i$.

From $\sigma \vdash \ell_0$, we have $C'_i <: C_i$. Therefore, $\text{type}_\sigma(\ell) \vdash T' \preceq T$.

If $T_0 = C_0$, then by Rule (RT-Field), $T = C_i$. We need to prove that $\text{locate}_{\sigma, \ell}(C'_i) = \ell'_i$ so that $T' = C'_i$. We also need to show that $\text{type}_\sigma(\ell) \vdash C'_i \preceq C_i$.

From Rule (RT-Var), $\text{locate}_{\sigma, \ell}(C_0) = \ell'_0$. Also from $\vdash \sigma$, we have $\sigma \vdash \ell_0$, which implies $\text{locate}_{\sigma, \ell'_0}(C_i) = \ell'_i$.

Suppose C_0 is not a gate type. Then $\text{inscope}_\sigma(\ell_0) = \ell'_0 = \ell''_0$. From $\text{locate}_{\sigma, \ell''_0}(C_i) = \ell'_i$, we have $\text{locate}_{\sigma, \ell'_0}(C_i) = \ell'_i$ and $\text{type}_\sigma(\ell'_0) \vdash C'_i \preceq C_i$. Also from Rule (RT-Field), $C_0 \vdash C_i \text{ viz } \text{type}_\sigma(\ell)$. Since ℓ'_0 is the allocation scope of ℓ_0 , $C_0 = c$ or $\text{type}_\sigma(\ell'_0) = \text{scopeof}(C_0)$. From $C_0 \vdash C_i \text{ viz } \text{type}_\sigma(\ell)$, if $C_i = c$ and $\text{scopeof}(C_0) = S$, then $S = \text{type}_\sigma(\ell)$. From $\text{type}_\sigma(\ell'_0) \vdash C'_i \preceq C_i$, if $C_i = c$ and $\text{scopeof}(C'_i) = S$, then $S = \text{type}_\sigma(\ell'_0)$. If $C_0 = c$, then $\ell'_0 = \ell$. Thus, in all cases, if $C_i = c$ and $\text{scopeof}(C'_i) = S$, then $S = \text{type}_\sigma(\ell)$. Consequently, we have $\text{type}_\sigma(\ell) \vdash C'_i \preceq C_i$. From $\text{locate}_{\sigma, \ell}(C_0) = \ell'_0$, $\text{locate}_{\sigma, \ell'_0}(C_i) = \ell'_i$, and $C_0 \vdash C_i \text{ viz } \text{type}_\sigma(\ell)$, we can conclude that $\text{locate}_{\sigma, \ell}(C_i) = \ell'_i$. Therefore, in this case, we have $\sigma, \ell \vdash \ell_i : C'_i$, where $\text{type}_\sigma(\ell) \vdash C'_i \preceq C_i$.

Suppose C_0 is a gate type and $\ell_0 \neq \ell$. Then $\text{inscope}_\sigma(\ell_0) = \ell_0 = \ell'_0$. From $\text{locate}_{\sigma, \ell'_0}(C_i) = \ell'_i$, we have $\text{locate}_{\sigma, \ell_0}(C_i) = \ell'_i$ and $C_0 \vdash C'_i \preceq C_i$. Also from Rule (RT-Field), $C_0 \vdash C_i \text{ viz } \text{type}_\sigma(\ell)$. Thus, if $C_i = c$ and $C'_i = S.c'$, then $S = C_0 = \text{type}_\sigma(\ell)$. Consequently, we have $\text{type}_\sigma(\ell) \vdash C'_i \preceq C_i$. Let $S = \text{type}_\sigma(\ell)$. Since C_0 is a gate type, we have $\text{locate}_{\sigma, \ell}(C_0) = \ell = \ell'_0$ and $C_0 = S.s$. By class typing rules, $\text{visible}(C_i, C_0)$. Together with $C_0 \vdash C_i \text{ viz } S$, C_i can only have the form of $S'.c$, where $S' = S$ or S' contains S . Thus, from $\text{locate}_{\sigma, \ell_0}(C_i) = \ell'_i$, we have $\text{locate}_{\sigma, \ell'_0}(C_i) = \ell'_i$. Since $\ell = \ell'_0$, $\text{locate}_{\sigma, \ell}(C_i) = \ell'_i$.

R-UPD Let $e = \ell_0.f_i := v$. By Rule (R-Upd), $e' = v$ and by Rule (RT-Upd), $\sigma, \ell \vdash v : T$. Thus, $T' = T$ in this case.

R-NEW Let $e = \text{new } C()$. By Rule (R-New), $e' = \ell'$ where $\sigma' = \sigma[\ell' \mapsto C^{\ell(\overline{\text{null}})]}$.

Suppose $\text{type}_\sigma(\ell) = S$. Then by Rule (RT-New), $C \text{ viz } S$ and if $C = S'.c$, then $S' = S$. Thus, C has the forms of c , $S.s$, or $S.c$. Consequently, $\text{locate}_{\sigma, \ell}(C) = \ell$.

By Rule (RT-Var), we have $\sigma, \ell \vdash \ell' : C$.

R-CAST Let $e = (C) \ell_0$. By Rule (R-Cast), $e' = \ell_0$ and if $\sigma(\ell_0) = C_0^{\ell'_0(\bar{v})}$, then $C_0 <: C$. By Rule (RT-Cast), $\Sigma \vdash_{\sigma, \ell} \ell_0 : K$ and if $C' = S.c'$ and $C = c$, then $\text{type}_\sigma(\ell) = S$. Thus, we have $\text{type}_\sigma(\ell) \vdash C' \preceq C$.

R-INVK Let $e = \ell_0.m(\bar{v})$. By (R-Invk), if $\sigma(\ell_0) = C_0^{\ell'_0(\bar{v})}$, $mbody(C_0, m) = (\bar{x}, e_0)$, then $e' = [\bar{v}/\bar{x}, \ell'_0/\text{this}]e_0$.

Suppose $\sigma, \ell \vdash e : T$, $T = \text{local } C$ or C , $\text{inscope}_\sigma(\ell_0) = \ell'$. By Lemma 9, $\exists C'$ such that $\sigma, \ell' \vdash e' : C'$ and $\text{type}_\sigma(\ell') \vdash C' \preceq C$.

If $T = \text{local } C$, then we have $\text{type}_\sigma(\ell) \vdash C' \preceq T$.

If $T = C$, then by Rule (RT-Invk), $T_0 = C_0$ and $C_0 \vdash C \text{ viz } \text{type}_\sigma(\ell)$. Thus, if

$C = c$ and $\text{scopeof}(C_0) = S$, then $\text{type}_\sigma(\ell) = S$. Also from $\text{type}_\sigma(\ell') \vdash C' \preceq C$, if $C = c$ and $\text{scopeof}(C') = S'$, then $\text{type}_\sigma(\ell') = S'$. From $\sigma, \ell \vdash \ell_0 : C_0$, $\text{locate}_{\sigma, \ell}(C_0) = \ell'_0$. Therefore, if $C_0 = c_0$, then $\ell = \ell'_0 = \ell'$; otherwise $\text{type}_\sigma(\ell') = \text{scopeof}(C_0)$ and if $C = c$, $\text{scopeof}(C') = S'$, $\text{scopeof}(C_0) = S$, then $\text{type}_\sigma(\ell) = \text{scopeof}(C_0) = \text{type}_\sigma(\ell') = \text{scopeof}(C')$. Therefore, $\text{type}_\sigma(\ell) \vdash C' \preceq C$.

Lemma 9. *If $\vdash \sigma$, $e = \ell_0.m(\bar{v})$, $\sigma, \ell \vdash e : T$, $T = \text{local } C$ or C , and $\sigma, \ell, e \rightarrow \sigma, \ell', e'$, then $\exists C'$ such that $\sigma, \ell' \vdash e' : C'$ where $\text{type}_\sigma(\ell') \vdash C' \preceq C$.*

Proof. Suppose $T = C$. By Rule (RT-Invk), $\exists C_0, \bar{T}, \bar{T}'$ such that $m\text{type}(C_0, m) = \bar{T} \rightarrow C$, $\sigma, \ell \vdash \ell_0 : C_0, \forall i, \sigma, \ell \vdash v_i : T'_i, \text{type}_\sigma(\ell) \vdash T'_i \preceq T_i, C_0 \vdash T_i \text{ viz } \text{type}_\sigma(\ell)$.

Suppose $m\text{body}(C_0, m) = (\bar{x}, e)$. It can be shown from Rule (T-Meth) and class typing rules that $\exists C'_0$ such that $C_0 <: C'_0$ and $\bar{x} : \bar{T}$, $\text{this} : C'_0 \vdash e_0 : C'$.

Let $\text{inscope}_\sigma(\ell_0) = \ell'$. $\forall i$, if $T_i = \text{local } C_i$ and $T'_i = \text{local } C'_i$ or C''_i , then $\sigma, \ell' \vdash v_i : T''_i$, where $T''_i = C'_i$ or C''_i . Thus, $\text{type}_\sigma(\ell') \vdash T''_i \preceq T_i$.

If $T_i = C_i$, then $T'_i = C'_i$. From $\sigma, \ell \vdash \ell_0 : C_0$, $\text{locate}_{\sigma, \ell}(C_0) = \ell'_0$, where $\text{scope}_\sigma(\ell_0) = \ell'_0$. From Rule (T-Meth), $C_i \text{ viz } C'_0$ and from class typing rules, we have $C_i \text{ viz } C_0$. Together with $\text{type}_\sigma(\ell) \vdash C'_i \preceq C_i$ and $C_0 \vdash C_i \text{ viz } \text{type}_\sigma(\ell)$, we can conclude that $\text{locate}_{\sigma, \ell'}(C'_i) = \text{locate}_{\sigma, \ell}(C'_i)$. Thus, $\sigma, \ell' \vdash v_i : C'_i$ and $\text{type}_\sigma(\ell') \vdash C'_i \preceq C_i$.

Suppose $T = \text{local } C$. Then $\sigma, \ell, e \rightarrow \sigma, \ell, e'$ and $\sigma, \ell \vdash \ell_0 : \text{local } C_0$. By Rule (RT-Invk2), $m\text{type}(C_0, m) = \bar{T} \rightarrow C, \forall i, T_i = \text{local } C_i, \sigma, \ell \vdash v_i : T'_i, T'_i = \text{local } C'_i$ or C''_i , and $\text{type}_\sigma(\ell) \vdash T'_i \preceq T_i$. It is clear that $\sigma, \ell' \vdash v_i : T''_i$, where $T''_i = \text{local } C'_i$ or C''_i . Thus, $\text{type}_\sigma(\ell) \vdash T''_i \preceq T_i$.

Therefore, by Lemma 10, $\sigma, \ell' \vdash e' : C'$ and $\text{type}_\sigma(\ell') \vdash C' \preceq C$.

Lemma 10. *If $\vdash \sigma$, $\text{inscope}_\sigma(\ell_0) = \ell$, $\text{type}_\sigma(\ell_0) = C_0, C_0 <: C'_0, \sigma, \ell \vdash \bar{v} : \bar{T}'$, where $\text{type}_\sigma(\ell) \vdash \bar{T}' \preceq \bar{T}$, and $\Gamma \vdash e_0 : T$, where, $\Gamma = \bar{x} : \bar{T}, \text{this} : C'_0$, then $\sigma, \ell \vdash e : T'$, where $e = [\bar{v}/\bar{x}, \ell'_0/\text{this}]e_0$ and $\text{type}_\sigma(\ell) \vdash T' \preceq T$.*

Proof. We prove by induction on the structure of e_0 .

- If $e_0 = x_i \in \bar{x}$, then, $e = v_i$. By assumption, $\sigma, \ell \vdash v_i : T'_i$, where $\text{type}_\sigma(\ell) \vdash T'_i \preceq T_i$.
If $e_0 = \text{this}$, then $e = \ell_0$ and by Rule (T-Var), we have $C'_0 \text{ viz } C'_0$, which means that C'_0 is not a gate type. From $C'_0 <: C_0$, C_0 is not a gate type either. From $\text{inscope}_\sigma(\ell_0) = \ell$, we have $\text{locate}_{\sigma, \ell_0}(C_0) = \ell$. By Rule (RT-Var), we have $\sigma, \ell \vdash e : C_0$. Since ℓ is the allocation scope of ℓ_0 , it is clear that $\text{type}_\sigma(\ell) \vdash C_0 \preceq C'_0$.
- If $e_0 = \text{new } C()$, then $e = e_0$. By assumption $\text{inscope}_\sigma(\ell_0) = \ell$. Thus, $C \text{ viz } C_0$ implies $C \text{ viz } \text{type}_\sigma(\ell)$. Also, $\text{scopeof}(C_0) = \text{scopeof}(\text{type}_\sigma(\ell))$. Therefore, by Rule (RT-New) $\sigma, \ell \vdash e : C$.
- Suppose $e_0 = (C) e'_0$. Then $e = (C) e'$, where $e' = [\bar{v}/\bar{x}, \ell'_0/\text{this}]e'_0$. By hypothesis, $\exists K$ such that $\Gamma \vdash e_0 : K$ where $C'_0 \vdash C \preceq K$ or $C_0 \vdash K \preceq C$. By induction hypothesis, $\exists K'$ such that $\sigma, \ell \vdash e'_0 : K'$ where $\text{type}_\sigma(\ell) \vdash K' \preceq K$.
Thus, if $C'_0 \vdash K \preceq C$, then $\text{type}_\sigma(\ell) \vdash K' \preceq C$. If $C'_0 \vdash C \preceq K$, then either $K = S.c'$ and $C = S.c$, or $K = c'$, $C = S.c$, and $\text{scopeof}(C'_0) = S$, or $C = c$ and $K = c'$.

From $\text{type}_\sigma(\ell) \vdash K' \preceq K$, either $K = S.c'$ and $K' = S.c''$, or $K = c'$, $K' = S.c''$, and $\text{type}_\sigma(\ell) = S$. Therefore, we have either $C = S.c$ and $K' = S.c''$, or $C = c$, $K' = S.c''$, and $\text{type}_\sigma(\ell) = S$. Hence by Rule (RT-Cast), we have $\sigma, \ell \vdash e' : C$.

- Suppose $e_0 = e_r.f_i$. Then $e = e'_r.f_i$, where $e'_r = [\bar{v}/\bar{x}, \ell_0/\text{this}]e_r'$.

There are three cases:

1. $e_r = \text{this}$ and $\Gamma(\text{this})$ is a gate type.

In this case, $e_r' = \ell_0 = \ell$ and $C_0 = C'_0$. Thus by Rule (T-Field), $C_0 \vdash C_i \text{ viz } C_0$. Since $\text{type}_\sigma(\ell) = C_0$, by Rule (RT-Select), we have $\sigma, \ell \vdash e' : C_i$.

2. $\Gamma \vdash e_r : C_r$, or $e_r = \text{this}$ and $\Gamma(\text{this})$ is not a gate type.

Since a non-gate type is visible from itself, by Rules (T-Field) and (T-Var), $\Gamma \vdash e_r : C_r$. By induction hypothesis, $\exists C'_r$ such that $\sigma, \ell \vdash e_r' : C'_r$ and $\text{type}_\sigma(\ell) \vdash C'_r \preceq C_r$. By the definition of *fields*, the type of f_i is still C_i . By Rule (T-Field), $C_r \vdash C_i \text{ viz } C'_0$. Thus, if C_i is a reusable type, then either C_r is reusable or C_r and C'_0 have the same static scope. From $\text{type}_\sigma(\ell) \vdash C'_r \preceq C_r$, if C_r is a reusable type, then either C'_r is reusable or C'_r and $\text{type}_\sigma(\ell)$ have the same static scope. Also, if C_r is not a reusable type, then C_r and C'_r have the same static scope. Moreover, if C'_0 has a static scope, then it is the same as that of $\text{type}_\sigma(\ell)$. Thus, if C_i is reusable, then either C'_r is reusable or C'_r and $\text{type}_\sigma(\ell)$ have the same static scope. This implies $C'_r \vdash C_i \text{ viz } \text{type}_\sigma(\ell)$. By Rule (RT-Field), we have $\sigma, \ell \vdash e' : C_i$.

3. $\Gamma \vdash e_r : \text{local } C_r = T_r$. By Rule (T-Field2), $\Gamma \vdash e_0 : \text{local } C_i$. By induction hypothesis, $\exists T'_r$ such that $\sigma, \ell \vdash e'_r : T'_r$ and $\text{type}_\sigma(\ell) \vdash T'_r \preceq T_r$. By definition of *fields*, the type of the field f_i is still C_i . Thus, $\sigma, \ell \vdash e : \text{local } C_i$ by Rule (RT-Field).

- Suppose $e_0 = e_r.f_i := e'$. Then $e = e'_r.f_i := e''$, where $e'_r = [\bar{v}/\bar{x}, \ell_0/\text{this}]e_r'$ and $e'' = [\bar{v}/\bar{x}, \ell_0/\text{this}]e'$. Also suppose $\Gamma \vdash e_r.f_i : C_i$ and $\Gamma \vdash e' : C'$. Then by induction hypothesis, $\sigma, \ell \vdash e'_r.f_i : C_i$ and $\sigma, \ell \vdash e'' : C''$, where $\text{type}_\sigma(\ell) \vdash C'' \preceq C'$. By Rule (T-Upd), $C'_0 \vdash C'' \preceq C_i$. Thus, $\text{type}_\sigma(\ell) \vdash C'' \preceq C_i$. By Rule (RT-Upd), $\sigma, \ell \vdash e : C''$.

- Suppose $e_0 = e_r.m(\bar{e})$. Then $e = e'_r.m(\bar{e}')$, where $e'_r = [\bar{v}/\bar{x}, \ell_0/\text{this}]e_r'$ and $\forall i, e_i' = [\bar{v}/\bar{x}, \ell_0/\text{this}]e_i$. Also suppose $\Gamma \vdash e_0 : T$, $\Gamma \vdash e_r : C_r$ or $\Gamma \vdash e_r : \text{local } C_r$, $mtype(m, C_r) = \bar{T} \rightarrow C$, and $\Gamma \vdash \bar{e} : \bar{T}'$. Then, $T = C$ or $T = \text{local } C$. By Rule (T-Invk), $C'_0 \vdash \bar{T}' \preceq \bar{T}$. By induction hypothesis, $\exists \bar{T}''$ such that $\sigma, \ell \vdash \bar{e}' : \bar{T}''$ and $\text{type}_\sigma(\ell) \vdash \bar{T}'' \preceq \bar{T}'$. Thus, we have $\text{type}_\sigma(\ell) \vdash \bar{T}'' \preceq \bar{T}$.

There are three cases.

1. $e_r = \text{this}$ and $\Gamma(\text{this})$ is a gate type. In this case, $e = \ell.m(\bar{e}')$ and $C_r = C_0 = C'_0 = \text{type}_\sigma(\ell)$. By Rule (T-Invk), we have $C_r \vdash \bar{T} \text{ viz } C'_0$. $C_r \vdash C \text{ viz } C'_0$. This is the same as $C_r \vdash \bar{T} \text{ viz } \text{type}_\sigma(\ell)$. $C_r \vdash C \text{ viz } \text{type}_\sigma(\ell)$. Thus by Rule (RT-Invk), we have $\sigma, \ell \vdash e : C$.

2. $\Gamma \vdash e_r : C_r$, or $e_r = \text{this}$ and $\Gamma(\text{this})$ is not a gate type.

We have $\Gamma \vdash e_r : C_r$ in either case. By induction hypothesis, $\exists C'_r$ such that $\sigma, \ell \vdash e_r' : C'_r$ and $\text{type}_\sigma(\ell) \vdash C'_r \preceq C_r$. By Rule (T-Meth), $mtype(m, C'_r) = \bar{T} \rightarrow C$. By Rule (T-Invk), we have $C_r \vdash \bar{T} \text{ viz } C'_0$. $C_r \vdash C \text{ viz } C'_0$. Together with $\text{type}_\sigma(\ell) \vdash C'_r \preceq C_r$, we have $C'_r \vdash \bar{T} \text{ viz } \text{type}_\sigma(\ell)$. $C'_r \vdash C \text{ viz } \text{type}_\sigma(\ell)$. Thus, by Rule (RT-Invk), we have $\sigma, \ell \vdash e : C$.

3. $\Gamma \vdash e_r : \text{local } C_r$. By induction hypothesis, $\exists T_r$ such that $\sigma, \ell \vdash e_r' : T_r$ and $\text{type}_\sigma(\ell) \vdash T_r \preceq \text{local } C_r$. By Rule (T-Invk2), \bar{T} are scope local. Thus, by Rule

(RT-Invk2), we have $\sigma, \ell \vdash e : \text{local } C$.

Lemma 11. *If $C \text{ viz } C_0$ and $\text{type}_\sigma(\ell) = C_0$, then $\exists \ell'$ such that $\text{locate}_{\sigma, \ell}(C) = \ell'$.*

Lemma 12. *If $\vdash \sigma$, $\sigma, \ell \vdash e : T$, $\sigma, \ell \vdash_{\text{scope}} e : \ell'$, $T = \text{local } C$ or $T = C$, and $\text{type}_\sigma(\ell') = S$, then either $C = c$, $C = S.s$, or $C = S.c$.*

PROOF OF LEMMA 2

Proof. We prove by induction on the structure of e .

- $e = \text{null}$. $\sigma, \ell \vdash \text{null} : C$ for any C and $\sigma, \ell \vdash_{\text{scope}} \text{null} : \ell'$ for any ℓ' .
- $e = \ell_0$. If $\sigma(\ell_0) = C^{\ell_0}(\bar{v})$, then $\sigma, \ell \vdash_{\text{scope}} \ell_0 : \ell'_0$. If $T = C$, then by Rule (RT-Var), $\text{locate}_{\sigma, \ell}(C) = \ell'_0$.
- $e = \text{new } C()$. By Rule (RT-New), $C \text{ viz } \text{type}_\sigma(\ell)$. By Lemma 11, $\exists \ell'$ such that $\sigma, \ell \vdash_{\text{scope}} e : \ell'$. Let $S = \text{type}_\sigma(\ell)$. Since $C \text{ viz } S$, $C = c$ or $C = S.s$ or $C = S'.c$ and S' is S or encloses S . If $C = c$ or $S.s$, then $\ell' = \ell$ and $\text{locate}_{\sigma, \ell}(C) = \ell$. If $C = S'.c$, then by Rule (RT-New), $S' = S$. Thus, we have $\ell' = \ell$ and $\text{locate}_{\sigma, \ell}(C) = \ell$ as well.
- $e = (C) e$. By induction hypothesis, $\exists \ell''$ such that $\sigma, \ell \vdash_{\text{scope}} e : \ell''$. By Rule (RT-Cast), we have $C \text{ viz } \text{type}_\sigma(\ell)$. By Lemma 11, $\exists \ell'$ such that $\text{locate}_{\sigma, \ell}(C) = \ell'$. Let $\sigma, \ell \vdash e : K$. By induction, $\text{locate}_{\sigma, \ell}(K) = \ell''$. Also by Rule (RT-Cast), $(C = S.c \wedge K = S'.c') \Rightarrow S = S'$ and $((C = c \wedge K = S.c') \vee (K = c \wedge C = S.c')) \Rightarrow \text{type}_\sigma(\ell) = S$. Thus, $\ell'' = \ell'$ and $\sigma, \ell \vdash_{\text{scope}} (C) e : \ell'$.
- $e = e_0.f_i$ and $e_0 \neq \ell$.
By induction hypothesis, $\sigma, \ell \vdash_{\text{scope}} e_0 : \ell_0$. Suppose $\sigma, \ell \vdash e_0.f_i : T$ and $\sigma, \ell \vdash e_0 : T_0$. If $T_0 = \text{local } C_0$, then C_0 is not a gate type and $\exists C$ such that $T = \text{local } C$. By class typing rules, $C \text{ viz } C_0$. Let $\text{type}_\sigma(\ell_0) = S$. By Lemma 12, $C_0 = c$ or $C_0 = S.s$ or $C_0 = S.c$. If C_0 is not a gate type, then we have $C \text{ viz } S$.
If $T_0 = C_0$, then $\exists C$ such that $T = C$. By Rule (RT-Select), we have $C_0 \vdash C \text{ viz } \text{type}_\sigma(\ell)$.
If $C_0 = S.s$, then $\text{type}_\sigma(\ell) = S$ and C is not of the form c or $S.s.c$ or $S.s.s'$. Thus, we have $C \text{ viz } S$.
Hence, by Lemma 11, $\exists \ell'$ such that $\text{locate}_{\sigma, \ell_0}(T) = \ell'$.
Moreover, if $T = C$, then $\exists C_0$ such that $T_0 = C_0$ and $C_0 \vdash C \text{ viz } \text{type}_\sigma(\ell)$. Together with $C \text{ viz } C_0$, if $C = S'.c$, then S' equals to or encloses the static scopes of C_0 and $\text{type}_\sigma(\ell)$; if $C = c$ or S' , then the static scopes of C_0 and $\text{type}_\sigma(\ell)$ are the same.
By induction hypothesis, $\exists \ell_0$ such that $\sigma, \ell \vdash_{\text{scope}} e_0 : \ell_0$ and $\text{locate}_{\sigma, \ell}(C_0) = \ell_0$. Since $\text{locate}_{\sigma, \ell_0}(C) = \ell'$, we have $\text{locate}_{\sigma, \ell}(C) = \ell'$.
- $e = \ell.f_i$. If $\sigma, \ell \vdash e : C$, then by class typing rule, $C \text{ viz } \text{type}_\sigma(\ell)$. Thus, by Lemma 11, $\exists \ell'$ such that $\text{locate}_{\sigma, \ell}(C) = \ell'$. By Rule (S-Field2), $\sigma, \ell \vdash_{\text{scope}} e : \ell'$.
- $e = e_0.f_i := e'$. By induction hypothesis, $\exists \ell', \ell''$ such that $\sigma, \ell \vdash_{\text{scope}} e_0.f_i : \ell'$ and $\sigma, \ell \vdash_{\text{scope}} e' : \ell''$. Let $\sigma, \ell \vdash e_0 : C$, $\sigma, \ell \vdash e' : C'$ and $\text{fields}(C) = (\bar{C} \bar{f})$.
By induction hypothesis, $\text{locate}_{\sigma, \ell}(C_i) = \ell'$ and $\text{locate}_{\sigma, \ell}(C') = \ell''$. Thus, $\text{locate}_{\sigma, \ell}(C'') = \ell''$. Since $\text{type}_\sigma(\ell) \vdash C' <: C_i$, if $C_i = S$, then $C' = S$, else if $C_i = S.c$, then $C' = S.c'$, else if $C_i = c$, then either $C' = c'$ or $C' = S.c'$, where $S = \text{type}_\sigma(\ell)$. Thus, $\ell' = \ell''$ by the definition of $\text{locate}_{\sigma, \ell}(C')$ and $\text{locate}_{\sigma, \ell}(C_i)$.

- $e = e_0.m(\bar{e})$ and $e_0 \neq \ell$. Let $\text{type}_\sigma(\ell) = S$. By induction hypothesis, $\exists \ell_0$ such that $\sigma, \ell \vdash_{\text{scope}} e_0 : \ell_0$. Let $\text{type}_\sigma(\ell_0) = S_0$.
Let $\sigma, \ell \vdash e_0 : T_0$ and $m\text{type}(m, C_0) = \bar{T} \rightarrow C$, where $T_0 = \text{local } C_0$ or C_0 . If $T_0 = \text{local } C_0$, then C_0 is not a gate type. Then C_0 either is c or $S_0.c$. By method typing rule, we have $C \text{ viz } C_0$. Thus, $C \text{ viz } S_0$. By Lemma 11, $\exists \ell'$ such that $\text{locate}_{\sigma, \ell_0}(C) = \ell'$.
If $T = C_0$, then by Rule (RT-Invk), we have $C_0 \vdash C \text{ viz } S$. By method typing rule, we have $C \text{ viz } C_0$. From $\sigma, \ell \vdash_{\text{scope}} e_0 : \ell_0$ and induction hypothesis, we have $\text{locate}_{\sigma, \ell}(C_0) = \ell_0$. Thus, by Lemma 12, either $C_0 = c_0$, or $C_0 = S_0.c_0$ where S_0 equals to or encloses the static scope of S , or $C_0 = S_0.s_0$, where $S_0 = S$. From $C_0 \vdash C \text{ viz } S$, either $C = c$, where the static scope of C_0 must be S , or $C = S'.c$, where S' equals to or encloses the static scope of S , or $C = S.s$, where $C_0 = S.c_0$. If $C_0 = S_0.s_0$, then C cannot have the forms of c or $S_0.s_0.c$ or $S_0.s_0.s$ since it would violate $C_0 \vdash C \text{ viz } S$. Thus, from $\text{visible}(C, C_0)$, we have $C \text{ viz } S_0$. By Lemma 11, $\exists \ell'$ such that $\text{locate}_{\sigma, \ell_0}(C) = \ell'$.
Recall that $\text{locate}_{\sigma, \ell}(C_0) = \ell_0$. From $C'_0 \vdash C \text{ viz } S$, if $C = c$ or $S.s$, then $S = S_0$ and if $C = S'.c$, then S' equals to or encloses the static scope of S_0 , which equals to or encloses the static scope of S . Thus, we can conclude that $\text{locate}_{\sigma, \ell}(C) = \ell'$.
- $e = \ell.m(\bar{e})$. If $\text{type}_\sigma(\ell) = C_0$ and $\sigma, \ell \vdash e : C$, then from Rule (RT-Invk2), $C \text{ viz } C_0$. By Lemma 11, $\exists \ell'$ such that $\text{locate}_{\sigma, \ell}(C) = \ell'$. By Rule (S-Invk2), $\sigma, \ell \vdash e : \ell'$.

PROOF OF LEMMA 3

Proof. The only cases that $\sigma' \neq \sigma$ are when $e = \text{new } C()$ or $e = \ell_0.f_i := v$. Let $T = C$ since in both cases, T may not be scope local.

Suppose $e = \text{new } C()$. Then from $\sigma, \ell \vdash_{\text{scope}} e : \ell'$ and by Rule (S-New), $\ell' = \ell$. By Lemma 2, $\text{locate}_{\sigma, \ell}(C) = \ell$. By Rule (R-New), $\sigma, \ell, \text{new } C() \rightarrow \sigma', \ell, \ell_0$, where $\sigma'(\ell_0) = C^\ell(\overline{\text{null}})$. Since $\sigma' = \sigma[\ell_0 \mapsto C^\ell(\overline{\text{null}})]$ and $\vdash \sigma$, we have $\sigma' \vdash \ell_0$ and consequently $\vdash \sigma'$.

Suppose $e = \ell_0.f_i := v$ and $\ell_0 \neq \ell$. By Rule (R-Upd), $\sigma, \ell, e \rightarrow \sigma', \ell, v$, where $\sigma(\ell_0) = C^{\ell_0}(\bar{v})$ and $\sigma' = \sigma[\ell_0 \mapsto C^{\ell_0}(\bar{v} \downarrow_i \bar{v})]$. Suppose $v = \ell_i$ and $\sigma(\ell_i) = C^{\ell'}(\bar{v})$. By Rule (RT-Upd), we have $C' <: C$.

From $\sigma, \ell \vdash e : C$ and Lemma 2, $\sigma, \ell \vdash_{\text{scope}} \ell_i : \ell'$ and $\sigma, \ell \vdash_{\text{scope}} \ell_0.f_i : \ell'$. By Rule (S-Var), $\sigma, \ell \vdash_{\text{scope}} \ell_0 : \ell'_0$. By Rule (S-Field), $\text{locate}_{\sigma, \ell'_0}(C) = \ell'$.

If C_0 is not a gate type, then $\text{inscope}_\sigma(\ell_0) = \ell'_0$, $\sigma' \vdash \ell_0$ and $\vdash \sigma'$.

If C_0 is a gate type, then $\text{inscope}_\sigma(\ell_0) = \ell_0$ and we need to show that $\text{locate}_{\sigma, \ell_0}(C) = \ell'$. From $\sigma, \ell \vdash_{\text{scope}} \ell_0 : \ell'_0$ and Lemma 2, $\text{locate}_{\sigma, \ell}(C_0) = \ell'_0$. Let $S = \text{type}_\sigma(\ell)$. Since C_0 is a gate type, $\ell'_0 = \ell$ and C_0 has the form of $S.s$. From Rule (RT-Field), $C_0 \vdash C \text{ viz } S$. Thus, C may only have the form of $S'.c$, where $S' = S$ or S' encloses S . Therefore, $\text{locate}_{\sigma, \ell_0}(C) = \text{locate}_{\sigma, \ell'_0}(C) = \ell'$. Consequently, $\sigma' \vdash \ell_0$ and $\vdash \sigma'$.

Suppose $e = \ell.f_i := v$ and $v = \ell_i$. By Rule (S-Upd), $\sigma, \ell \vdash \ell.f_i : \ell'$ and $\sigma, \ell \vdash v : \ell'$. By Lemma 2, $\text{locate}_{\sigma, \ell}(C) = \ell'$. Thus, $\sigma' \vdash \ell_0$ and $\vdash \sigma'$.

Lemma 13. *If $\sigma \vdash \gamma$, $\ell \in \gamma$, and $\ell \preceq_\sigma \ell'$, then $\ell' \in \gamma$.*

Proof. We prove by induction. Consider the case of $\gamma = \epsilon \bullet \ell_0$. Then by $\sigma \vdash \gamma$, ℓ_0 is the immortal scope. Since $\ell \in \gamma$, $\ell = \ell_0$. Since $\ell \preceq_\sigma \ell'$, $\ell' = \ell_0$ as well. Thus, $\ell' \in \gamma$.

Suppose that $\gamma' \neq \epsilon$ and the lemma holds for γ' . Let $\gamma = \gamma' \bullet \ell_0$. By assumption, $\ell \in \gamma$. Either $\ell = \ell_0$ or $\ell \in \gamma'$. If $\ell \in \gamma'$, then the lemma holds by induction hypothesis.

Consider the case of $\ell = \ell_0$. By assumption, $\ell \preceq_\sigma \ell'$. In case of $\ell = \ell'$, the lemma holds. The other case is $\ell \neq \ell'$. Suppose $\sigma(\ell) = \mathcal{C}^{\ell'_0(\bar{v})}$. Then $\ell'_0 \preceq_\sigma \ell'$. By definition of $\sigma \vdash \gamma$, either $\ell_0 \in \gamma'$ or $\gamma' = \gamma'' \bullet \ell'_0$. If $\ell_0 \in \gamma'$, then from $\ell = \ell_0$, the lemma holds by induction. Otherwise, $\ell'_0 \in \gamma'$ and $\ell'_0 \preceq_\sigma \ell'$, the lemma also holds by induction hypothesis.

PROOF OF LEMMA 4

Proof. We prove by induction on the structure of e .

- $e = \ell_0$. By assumption, $\sigma, \gamma \vdash e$, we have $\ell_0 \in \gamma$.
- $e = \text{new } \mathcal{C}()$. By Rule (S-New), $\ell' = \ell$ and $\ell' \in \gamma$.
- $e = (\mathcal{C}) e'$. By Rule (S-Cast), $\text{locate}_{\sigma, \ell}(\mathcal{C}) = \ell'$. Thus, $\ell \preceq_\sigma \ell'$. By Lemma 13 and $\sigma \vdash \gamma$, $\ell' \in \gamma$.
- $e = e_0.f_i := e'$. By Rule (S-Upd), $\sigma, \ell \vdash_{\text{scope}} e' : \ell'$. By induction hypothesis, $\ell' \in \gamma$.
- $e = e_0.f_i$ and $e_0 \neq \ell$. By Rule (S-Field), $\exists T$ such that $\sigma, \ell \vdash e : T$, and $\exists \ell_0$ such that $\sigma, \ell \vdash_{\text{scope}} e_0 : \ell_0$ and $\text{locate}_{\sigma, \ell_0}(T) = \ell'$. Again, $\ell_0 \preceq_\sigma \ell'$ and by induction hypothesis, $\ell_0 \in \gamma$. By Lemma 13 and $\sigma \vdash \gamma$, $\ell' \in \gamma$.
- $e = \ell.f_i$. By Rule (S-Field2), $\exists T$ such that $\sigma, \ell \vdash e : T$, and $\text{locate}_{\sigma, \ell}(T) = \ell'$, which implies $\ell \preceq_\sigma \ell'$. We know $\ell \in \gamma$. By Lemma 13 and $\sigma \vdash \gamma$, $\ell' \in \gamma$.
- $e = e_0.m(\bar{e})$ and $e_0 \neq \ell$. The proof is the same as the case when $e = e_0.f_i$ and $e_0 \neq \ell$.
- $e = \ell.m(\bar{e})$. The proof is the same as the case when $e = \ell.f_i$.

PROOF OF LEMMA 5

Proof. We prove by induction on the structure of e .

- $e = \text{new } \mathcal{C}()$.
By Rule (S-New), $\sigma, \ell \vdash_{\text{scope}} e : \ell$. By Rule (R-New), $\sigma, \ell, e \rightarrow \sigma', \ell, e'$ and $e' = \ell_0$, where $\sigma'(\ell_0) = \mathcal{C}^{\ell(\overline{\text{null}})}$. Thus, $\sigma', \ell \vdash_{\text{scope}} e' : \ell$ by Rule (S-Var).
- $e = (\mathcal{C}) v$.
By Rule (S-Cast), $\sigma, \ell \vdash_{\text{scope}} e : \ell_a$ and $\sigma, \ell \vdash_{\text{scope}} v : \ell_a$.
- $e = \ell'.f_i$ and $\ell' \neq \ell$.
By Rule (S-Field), $\sigma, \ell \vdash_{\text{scope}} e : \ell_a$, where $\sigma, \ell \vdash_{\text{scope}} \ell' : \ell_0$, $\text{locate}_{\sigma, \ell_0}(T) = \ell_a$, $\sigma, \ell \vdash e : T$.
By Rule (R-Field), $\sigma, \ell, e \rightarrow \sigma, \ell, e'$ and $e' = v_i$, where $\sigma(\ell') = \mathcal{C}_0^{\ell_0(\bar{v})}$. The case is trivial if $v_i = \text{null}$. Let $v_i = \ell_i$.
Suppose $\sigma, \ell \vdash \ell' : T_0$. If $T = \text{local } \mathcal{C}$, then $T_0 = \text{local } \mathcal{C}_0$. In this case, \mathcal{C}_0 cannot be a gate type and hence, $\text{inscope}_\sigma(\ell') = \ell_0$. From $\vdash \sigma$, we have $\sigma \vdash \ell'$,

which implies $\text{locate}_{\sigma, \ell_0}(\mathbf{C}) = \ell'_i$, where $\text{scope}_{\sigma}(\ell_i) = \ell'_i$. In this case, $\ell'_i = \text{locate}_{\sigma, \ell_0}(\mathbf{T}) = \ell_a$. Since $\sigma, \ell \vdash_{\text{scope}} e' : \ell'_i$ by Rule (S-Var), we have $\sigma, \ell \vdash_{\text{scope}} e' : \ell_a$.

Consider the case $\mathbf{T} = \mathbf{C}$. By Lemma 2, $\text{locate}_{\sigma, \ell}(\mathbf{C}) = \ell_a$. By Lemma 1, $\sigma, \ell \vdash e' : \mathbf{C}'$ such that $\text{type}_{\sigma}(\ell) \vdash \mathbf{C}' \preceq \mathbf{C}$. Again, by Lemma 2, if $\sigma, \ell \vdash_{\text{scope}} e' : \ell'_a$, then $\text{locate}_{\sigma, \ell}(\mathbf{C}') = \ell'_a$. From $\text{type}_{\sigma}(\ell) \vdash \mathbf{C}' \preceq \mathbf{C}$, if $\mathbf{C} = \mathbf{S}$, then $\mathbf{C}' = \mathbf{S}$; if $\mathbf{C} = \mathbf{S}.c$, then $\mathbf{C}' = \mathbf{S}.c'$; if $\mathbf{C} = c$, then either $\mathbf{C}' = c'$ or $\mathbf{C}' = \mathbf{S}.c'$, where $\text{type}_{\sigma}(\ell) = \mathbf{S}$. It is clear that $\ell_a = \ell'_a$ for all cases.

– $e = \ell.f_i$.

In this case, if $\sigma(\ell) = S^{\ell'}(\bar{v})$, then by Rule (R-Field), $e' = v_i$. Suppose $v_i = \ell_i$. Suppose $\sigma, \ell \vdash e : \mathbf{C}$. By Lemma 2, $\sigma, \ell \vdash_{\text{scope}} e : \ell_a$ and $\text{locate}_{\sigma, \ell}(\mathbf{C}) = \ell_a$. From $\vdash \sigma, \text{locate}_{\sigma, \ell}(\mathbf{C}) = \ell'_i$, where $\text{scope}_{\sigma}(\ell_i) = \ell'_i$. Thus, $\ell'_i = \ell_a$ and $\sigma, \ell \vdash_{\text{scope}} e' : \ell_a$.

– $e = \ell_0.m(\bar{v})$ and $\ell_0 \neq \ell$. By Rule (R-Invk), $\sigma, \ell, e \rightarrow \sigma, \ell', e'$, where $\text{inscope}_{\sigma}(\ell_0) = \ell'$. If $\sigma, \ell \vdash e : \mathbf{T}$, then by Lemma 1, $\sigma, \ell' \vdash e' : \mathbf{T}'$ and $\text{type}_{\sigma}(\ell) \vdash \mathbf{T}' \preceq \mathbf{T}$.

Suppose $\sigma(\ell_0) = C_0^{\ell'_0}(\bar{v}')$. Then by Rule (S-Var), $\sigma, \ell \vdash_{\text{scope}} \ell_0 : \ell'_0$ and by Rule (S-Invk) $\text{locate}_{\sigma, \ell'_0}(\mathbf{T}) = \ell_a$.

Suppose $\sigma, \ell \vdash \ell_0 : T_0$. By Rules (RT-Invk) and (RT-Inv2), either $\mathbf{T} = \text{local } \mathbf{C}$ and $T_0 = \text{local } C_0$, or $\mathbf{T} = \mathbf{C}$ and $T_0 = C_0$.

Suppose C_0 is not a gate type. Then, $\ell' = \ell'_0$. By Lemma 9, $\exists \mathbf{C}'$ such that $\sigma, \ell' \vdash e' : \mathbf{C}'$, where $\text{type}_{\sigma}(\ell') \vdash \mathbf{C}' \preceq \mathbf{C}$. Thus, by Lemma 2, $\exists \ell'_a$ such that $\sigma, \ell' \vdash_{\text{scope}} e' : \ell'_a$ and $\text{locate}_{\sigma, \ell'}(\mathbf{C}') = \ell'_a$. From $\text{type}_{\sigma}(\ell') \vdash \mathbf{C}' \preceq \mathbf{C}$, we have $\text{locate}_{\sigma, \ell'}(\mathbf{C}) = \ell'_a$. Since $\ell' = \ell'_0$, $\text{locate}_{\sigma, \ell'_0}(\mathbf{T}) = \text{locate}_{\sigma, \ell'_0}(\mathbf{C}) = \ell_a$. We have $\ell'_a = \ell_a$ and $\sigma, \ell' \vdash_{\text{scope}} e' : \ell_a$.

Suppose C_0 is a gate type. Then $\mathbf{T} = \mathbf{C}$ and $T_0 = C_0$ since a gate type cannot be scope local. Then $\text{inscope}_{\sigma}(\ell_0) = \ell' = \ell_0$. By Rule (RT-Var), $\text{locate}_{\sigma, \ell}(C_0) = \ell'_0$. Let $\text{type}_{\sigma}(\ell) = \mathbf{S}$. Then C_0 must have the form of $\mathbf{S}.s_0$. By Lemma 1, $\sigma, \ell' \vdash e' : \mathbf{C}'$ where $\mathbf{S} \vdash \mathbf{C}' \preceq \mathbf{C}$ and $C_0 \vdash \mathbf{C}$ *viz* \mathbf{S} . Therefore, \mathbf{C} cannot have the form of c since this would imply $\text{scopeof}(C_0) = \mathbf{S}$. \mathbf{C} cannot be a gate type either. Suppose \mathbf{C} is a gate type. By Lemma 2, $\text{locate}_{\sigma, \ell}(\mathbf{C}) = \ell_a$, which means that \mathbf{C} must have the form of $\mathbf{S}.s$. Also by Lemma 2, $\text{locate}_{\sigma, \ell'}(\mathbf{C}') = \ell'_a$. In this case, \mathbf{C}' must have the form of $\mathbf{S}.s_0.s'$. But this would violate $\mathbf{C}' <: \mathbf{C}$. Thus, the only possible case left is that $\mathbf{C} = \mathbf{S}'.c$ and $\mathbf{C} = \mathbf{S}'.c'$, where $\mathbf{S}' = \mathbf{S}$ or \mathbf{S}' encloses \mathbf{S} . Recall that $\ell' = \ell_0$ in this case. From $\text{locate}_{\sigma, \ell'}(\mathbf{C}') = \ell'_a$, either $\ell' = \ell'_a$ or $\ell'_0 \preceq_{\sigma} \ell'_a$. However, ℓ' cannot be equal to ℓ'_a since it would mean that $\mathbf{S}' = \mathbf{S}.s_0$. Thus, $\ell'_0 \preceq_{\sigma} \ell'_a$ and this implies $\text{locate}_{\sigma, \ell'_0}(\mathbf{C}') = \ell'_a$. Since $\text{locate}_{\sigma, \ell'_0}(\mathbf{C}) = \ell_a$, we have $\ell_a = \ell'_a$.

– $e = \ell.m(\bar{v})$

By Rule (R-Invk), $\sigma, \ell, e \rightarrow \sigma, \ell, e'$ and since ℓ is a gate object, it cannot have scope-local type. Thus, by Rule (RT-Invk), $\exists \mathbf{C}$ such that $\sigma, \ell \vdash e : \mathbf{C}$. By Lemma 1, $\exists \mathbf{C}'$ such that $\sigma, \ell \vdash e' : \mathbf{C}'$ and $\text{type}_{\sigma}(\ell) \vdash \mathbf{C}' \preceq \mathbf{C}$. By Lemma 2, $\sigma, \ell \vdash_{\text{scope}} e : \ell_a$, $\text{locate}_{\sigma, \ell}(\mathbf{C}) = \ell_a$, $\sigma, \ell \vdash_{\text{scope}} e' : \ell'_a$, and $\text{locate}_{\sigma, \ell}(\mathbf{C}') = \ell'_a$. From $\text{type}_{\sigma}(\ell) \vdash \mathbf{C}' \preceq \mathbf{C}$, if $\mathbf{C} = c$, then either $\mathbf{C}' = c'$, or $\mathbf{C}' = \mathbf{S}.c'$ and $\text{type}_{\sigma}(\ell) = \mathbf{S}$; if $\mathbf{C} = \mathbf{S}.s$, then $\mathbf{C}' = \mathbf{S}.s$. In both cases, $\ell_a = \ell'_a = \ell$. If $\mathbf{C} = \mathbf{S}'.c$, then $\mathbf{C}' = \mathbf{S}'.c'$ and $\mathbf{S}' = \mathbf{S}$ or \mathbf{S}' encloses \mathbf{S} . Since \mathbf{C} and \mathbf{C}' have the same static scope, we have $\ell_a = \ell'_a$ as well.

– $e = \ell.f_i := v$.

By Rule (R-Upd), $\sigma, \ell, e \rightarrow \sigma, \ell, e'$ and $e' = v$. By Rule (S-Upd), if $\sigma, \ell \vdash_{scope} e : \ell_a$, then $\sigma, \ell \vdash_{scope} v : \ell_a$, which means that $\sigma, \ell \vdash_{scope} e' : \ell_a$.

Lemma 14. *If $\vdash \sigma, \sigma, \ell \vdash E[e] : T_0$, $\sigma, \ell \vdash e : T$, $\sigma, \ell \vdash e' : T'$, and $\text{type}_\sigma(\ell) \vdash T' \preceq T$, then $\sigma, \ell \vdash E[e'] : T'_0$, $\text{type}_\sigma(\ell) \vdash T'_0 \preceq T_0$.*

PROOF OF LEMMA 6

Proof. The proof is by case analysis of the reduction rule used.

G-STEP Let $P = P'' \mid \kappa \bullet \ell, E[e]$, e is not a method call, and $\sigma, \ell, e \rightarrow \sigma', \ell, e'$.

By Rule (G-STEP), $P' = P'' \mid \kappa'$, where $\kappa' = \kappa \bullet \ell, E[e']$. By assumption, $\vdash \sigma, \sigma \vdash \kappa \bullet \ell, E[e]$, We need to show $\sigma' \vdash P'$, which is to show $\vdash \sigma', \sigma' \vdash \kappa'$,

From Lemma 3, we have $\vdash \sigma'$. To show $\sigma \vdash \kappa'$, we need to show $\exists T'$, such that $\sigma', \ell \vdash E[e'] : T'$, $\sigma', \gamma \vdash E[e']$, and $\sigma', \ell \vdash_{scope} e' : \ell_a$, where $\gamma = \text{ScopeStack}(\kappa')$ and $\sigma, \ell \vdash_{scope} e : \ell_a$. From Lemma 1, $\exists T'$ such that $\sigma', \ell \vdash e' : T'$ where $\text{type}_\sigma(\ell) \vdash T' \preceq T$ and $\sigma, \ell \vdash e : T$. By Lemma 2, $\exists \ell'_a$ such that $\sigma', \ell \vdash_{scope} e' : \ell'_a$. By Lemma 5, $\ell'_a = \ell_a$. By Lemma 4, $\ell_a \in \gamma$. Thus, if $e' = \ell'$, then $\sigma'^{\ell'}(=)\ell_a \in \gamma$. Therefore, $\sigma', \gamma \vdash e'$ holds. It is clear that $\sigma', \gamma \vdash E[e']$ holds since E remains the same. Also, if $\sigma, \ell \vdash_{scope} E[e] : \ell_0$, it is clear that $\sigma', \ell \vdash_{scope} E[e'] : \ell_0$. By Lemma 14, $\exists C'_0$ such that $\sigma', \ell \vdash E[e'] : C'_0$ and $C'_0 <: C_0$, where $\sigma, \ell \vdash E[e] : C_0$.

G-ENTER If $P = P'' \mid \kappa \bullet \ell, E[e]$ and $\sigma, \ell, e \rightarrow \sigma, \ell', e'$, then by (G-ENTER), $P' = P'' \mid \kappa'$, where $\kappa' = \kappa \bullet \ell, E[e] \bullet \ell', e'$, $e = \ell_0.m(\bar{v})$, $\text{inscope}_\sigma(\ell_0) = \ell'$, $\sigma(\ell_0) = C_0^{\ell'_0}(\bar{v}')$, $\text{mbody}(C_0, m) = (\bar{x}, e_0)$, $e' = [\bar{v}/\bar{x}, \ell_0/\text{this}]e_0$.

By Lemma 1, $\exists C'$ such that $\sigma, \ell' \vdash e' : C'$ and $C' <: T$, where $\sigma, \ell \vdash e : T$. By Lemma 5, $\sigma, \ell' \vdash e' : \ell_a$, where $\sigma, \ell \vdash e : \ell_a$. Also, from $\sigma, \gamma \vdash e$, where $\gamma = \text{ScopeStack}(\kappa \bullet \ell, E[e])$, we have $\forall i, \sigma, \gamma \vdash v_i$. It is clear that $\sigma, \gamma' \vdash e'$ where $\gamma' = \text{ScopeStack}(\kappa')$ since $\gamma' = \gamma \bullet \ell'$. We also need to show $\sigma \vdash \gamma'$. From $\sigma, \gamma \vdash e$, we have $\sigma, \gamma \vdash \ell_0$. Then $\ell'_0 \in \gamma$. If ℓ_0 is not a gate object, then $\ell' = \ell_0$. If ℓ_0 is a gate object, then $\ell' = \ell_0$ but from $\sigma, \ell \vdash \ell_0 : C_0$, we have $\text{locate}_{\sigma, \ell}(C_0)$, which means that $\ell'_0 = \ell$. Therefore, $\sigma \vdash \gamma$ holds. Also, for all ℓ , if $\text{refcount}(\ell, P') = 0$, then $\text{refcount}(\ell, P) = 0$ and σ remains the same. Thus by definition, we have $\sigma \vdash \kappa'$ and $\sigma \vdash P'$.

G-RET Let $P = P'' \mid \kappa \bullet \ell, E[e] \bullet \ell', v$. By Rule (G-RET), $P' = P'' \mid \kappa'$ where $\kappa' = \kappa \bullet \ell, E[v]$; also, $\sigma' = \sigma$ if $\text{refcount}(\ell', P) \neq 0$ and otherwise, σ' is σ with all objects allocated in ℓ' removed and the fields of ℓ' reset to null.

Let $\gamma = \text{ScopeStack}(\kappa \bullet \ell, E[e] \bullet \ell', v)$ and $\gamma' = \text{ScopeStack}(\kappa')$. Then $\gamma = \gamma' \bullet \ell'$. It is clear that $\sigma \vdash \gamma'$. Suppose $\text{refcount}(\ell', P) = 0$. Then $\ell' \notin \gamma'$. We need to show $\sigma' \vdash \gamma'$ and $\vdash \sigma'$.

From $\sigma \vdash \gamma'$, if ℓ_0 is allocated in ℓ' , then $\ell' \in \gamma'$. Since $\ell' \notin \gamma'$, no object in γ' can be allocated in ℓ' . Thus, each $\ell_0 \in \gamma'$ is in the domain of σ' . Therefore, we have $\sigma' \vdash \gamma'$. Also, from $\sigma \vdash \kappa \bullet \ell, E[e] \bullet \ell', E[v]$, we have $\sigma, \gamma' \vdash E[e]$. Each object in $E[e]$ is allocated in one of the scopes in γ' . Since $\ell' \notin \gamma'$, we have $\sigma', \gamma' \vdash E[e]$. Also, $\sigma', \ell \vdash E[e] : T_0$ if $\sigma, \ell \vdash E[e] : T_0$. By induction on each frame in κ , we can show $\sigma' \vdash \kappa \bullet E[e]$ Similarly, for each call stack κ'' in P , we have $\sigma' \vdash \kappa''$.

To show $\vdash \sigma'$, we need to prove that $\forall \ell_x \in \text{dom}(\sigma')$, if the field of a ℓ_x points to another object ℓ'_x , then $\ell'_x \in \text{dom}(\sigma')$. From $\sigma \vdash P$, if $\text{refcount}(\ell_a, P) = 0$, then no object in σ is allocated in ℓ_a . Since no object in σ' is allocated in ℓ' and $\text{refcount}(\ell', P') = 0$, we conclude that $\text{refcount}(\ell_a, P') = 0$ implies no object in σ' is allocated in ℓ_a . Conversely, if an object in σ' is allocated in ℓ_x , then $\text{refcount}(\ell_a, P') \neq 0$.

Suppose ℓ_x is allocated in ℓ_x and it contains a field $v_i = \ell_i$. We will show that if $\sigma(\ell_i)$ is allocated in ℓ' , then it will lead to contradiction to the conditions of Rule (R-Ret). Then, ℓ_i would be undefined in σ' . From $\vdash \sigma$, if $\text{inscope}_\sigma(\ell_x) = \ell''_x$, then $\text{locate}_{\sigma, \ell''_x}(\mathcal{C}') = \ell'$, which means that $\ell''_x \preceq_\sigma \ell'$. Also, if ℓ_x is defined in σ' , then $\text{refcount}(\ell''_x, P') \neq 0$. Thus, $\exists \kappa''$ such that $P' = P'' \mid \kappa'', \gamma'' = \text{ScopeStack}(\kappa'')$ and $\ell'_x \in \gamma''$.

Suppose ℓ_x is not a gate object. Then from, $\ell''_x \preceq_\sigma \ell'$, $\ell'_x = \ell''_x \preceq_\sigma \ell'$. Thus, from $\sigma' \vdash \gamma''$, $\ell'_x \in \gamma''$, and Lemma 13, we have $\ell' \in \gamma''$. But this is a contradiction to $\text{refcount}(\ell', P') = 0$.

Suppose that ℓ_x is a gate object. Then $\ell''_x = \ell_x$. If ℓ_i is not allocated in ℓ_x , then $\ell'_x \preceq_\sigma \ell'$ and we can reach the same contradiction to $\text{refcount}(\ell', P') = 0$.

If ℓ_i is allocated in ℓ_x , then $\ell' = \ell_x$. However, by Rule (G-Ret), the fields of $\sigma'(\ell')$ are all set to null so that ℓ_i should not exist, which is also a contradiction.

Therefore, no object in σ' contains field objects allocated in ℓ' . From $\vdash \sigma$, we have $\vdash \sigma'$

From $\sigma' \vdash \kappa \bullet \ell, E[e]$, which implies $\mathcal{C}' <: \mathcal{C}$, where $\sigma', \ell \vdash e : \mathbb{T}$, $\mathbb{T} = \text{local } \mathcal{C}$ or $\mathbb{T} = \mathcal{C}$, and $\sigma, \ell' \vdash v : \mathcal{C}'$. If $\sigma', \ell \vdash_{\text{scope}} e : \ell_a$, then from $\sigma' \vdash \kappa \bullet \ell, E[e] \bullet \ell', v$, $\sigma, \ell' \vdash_{\text{scope}} v : \ell_a$, which means that v is either null or allocated in ℓ_a . Suppose v is not null. If $\mathbb{T} = \text{local } \mathcal{C}$, then we have $\text{type}_\sigma(\ell) \vdash \mathcal{C}' \preceq \mathbb{T}$. Otherwise $\mathbb{T} = \mathcal{C}$. Then $\text{locate}_{\sigma', \ell}(\mathcal{C}) = \ell_a$. Thus, if $\mathcal{C} = \mathcal{c}$ and $\mathcal{C}' = \mathcal{S.c}$, then $\ell_a = \ell$ and $\mathcal{S} = \text{type}_{\sigma'}(\ell)$. Therefore, $\text{type}_{\sigma'}(\ell) \vdash \mathcal{C}' \preceq \mathbb{T}$. Thus, by Lemma 14, $\exists \mathcal{C}'_0$ such that $\sigma', \ell \vdash E[v] : \mathcal{C}'_0$ and $\mathcal{C}'_0 <: \mathcal{C}_0$, where $\sigma', \ell \vdash E[e] : \mathcal{C}_0$.

By Lemma 4 and the fact that $\gamma' = \text{ScopeStack}(\kappa \bullet \ell, E[e])$, if $\sigma', \ell \vdash_{\text{scope}} e : \ell_a$, then $\ell_a \in \gamma'$. Also, from $\sigma \vdash \kappa \bullet \ell, E[e] \bullet \ell', v$, $\sigma, \ell' \vdash_{\text{scope}} v : \ell_a$, which implies $\sigma', \ell \vdash_{\text{scope}} v : \ell_a$ by Rules (S-Null) and (S-Var). Together with $\sigma', \gamma' \vdash E[e]$, we conclude that $\sigma', \gamma' \vdash E[v]$. Also, if $\sigma', \ell \vdash_{\text{scope}} E[e] : \ell_0$, it is clear that $\sigma', \ell \vdash_{\text{scope}} E[v] : \ell_0$.

By assumption, for all ℓ_0 , if $\text{refcount}(\ell_0, P) = 0$, then no object in σ is allocated in ℓ_0 . By Rule (G-Ret), if $\text{refcount}(\ell', P') = 0$, then no object in σ' is allocated in ℓ' . For all $\ell_0 \neq \ell'$, $\text{refcount}(\ell_0, P) = \text{refcount}(\ell_0, P')$. Thus, for all ℓ_0 , $\text{refcount}(\ell_0, P') = 0$ implies no object in σ' is allocated in ℓ_0 . Therefore, we can conclude that $\sigma' \vdash P'$.