Implicit Ownership Types for Memory Management

Tian Zhao[a,*], Jason Baker[b], James Hunt[c], James Noble[d], Jan Vitek[b,*]

[a]*University of Wisconsin – Milwaukee, USA*
[b]*Purdue University, West Lafayette, USA*
[c]*aicas GmbH, Karlsruhe, DE*
[d]*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.

Key words: Real-time Java, RTSJ, type systems, memory management, ownership types

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

*Corresponding author
Email addresses: tzhao@uwm.edu (Tian Zhao), jason.baker0@gmail.com (Jason Baker), jjh@aicas.com (James Hunt), kjx@mcs.vuw.ac.nz (James Noble), jv@cs.purdue.edu (Jan Vitek)

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 programs[1]. 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 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

[1]As reported by users and vendors in personal communications.

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].

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.

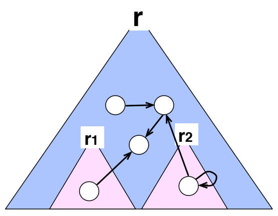


Figure 1: A region hierarchy with safe object references.

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].

```
class Pair⟨r⟩ extends Object ⟨r⟩ {
  Object⟨r⟩ a;
  Object⟨r⟩ b;
  Object⟨r⟩ setA(
       Object⟨r⟩ o)⟨r⟩ {
   Object⟨r⟩ t = a;
   a = o; return t;
  }
}
```

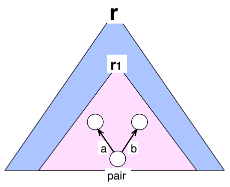


Figure 2: Instances of `Pair` and targets co-located.

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()⟨r⟩`, 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`).

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

```
class Pair⟨r,s⟩ extends Object⟨r⟩ where s ⪰ r {
  Object⟨s⟩ a;
  Object⟨r⟩ b;
  Object⟨s⟩ setA(Object⟨s⟩ o)⟨r⟩ {
     Object⟨s⟩ t = a;
     a = o;
     return t;
  }
}
```

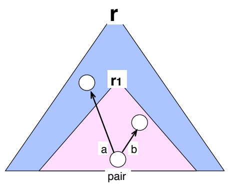


Figure 3: Field `a` can refer to a parent region.

type parameter on the definition of the class as shown by Figure 3. This new region argument, `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.

```
class Pair⟨r,s,t⟩ extends Object⟨r⟩
  where s ⪰ r, t ⪰ r {
  Object⟨s⟩ a;
  Object⟨t⟩ b;
  Pair⟨r′,s,t⟩ clone()⟨r′⟩
   where s ⪰ r′, t ⪰ r′{
   Pair⟨r′,s,t⟩ p =
    new Pair⟨r′,s,t⟩();
   p.a = a; p.b = b; return p;
  }
}
```

Figure 4: Fully general `Pair`.

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 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

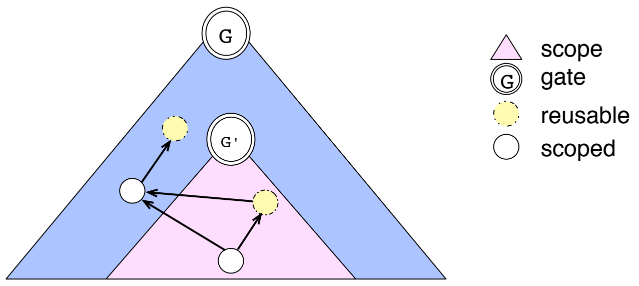


Figure 5: ScopeJ programming model. A scope is associated with every instance of `ScopeGate`. Scoped/reusable classes are allocated within a scope. Scoped/reusable objects can refer to objects allocated in the same or parent scope. Reusable objects are only visible in their allocating scope.

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 even though 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 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 { ...  }
  }
}
```

Figure 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 either scoped or reusable.

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

Figure 7: A scoped `Pair`.

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

Figure 8: A reusable `Pair`.

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.

[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.

```
  class Top extends ScopeGate {
   class Data {...}
   class Processor extends ScopeGate {
    class Parser {
    void send(Data b) {...}
   }
    void getMessage(Data data) {
    Parser q = new Parser(data);
    q.send(data);
   }
  }
 void runLoop() {
    Processor p = new Processor();
   while (true) {
    ...  get data periodically ...
    p.getMessage(data);
 } }
}
```

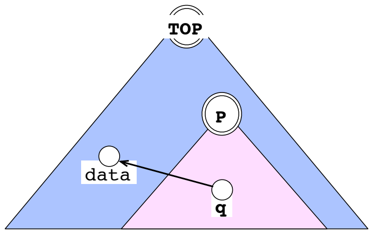


Figure 9: Scoped Run Loop Pattern. The runLoop method periodically acquires data and uses a region to process it.

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 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 run-time 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

```
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());
  }
}
```

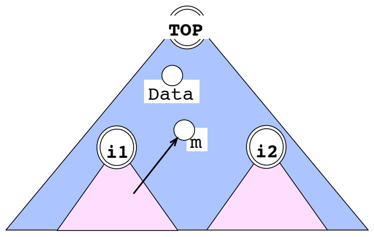


Figure 10: The Multi Scoped Object Design Pattern.

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 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].

```
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);
  }
}
```

Figure 11: Implicit Polymorphism.

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
     }
   }
   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];
}
```

Figure 12: Type-safe casts.

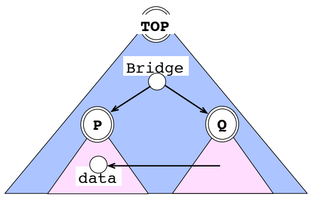
One of the goals of ScopeJ is to allow writing code in a style that is as natural as possible. So, for example, it is desirable to allow programs such as the one in Figure 12 where an array of objects is used to store and then retrieve scoped objects. This example is particularly important as it is the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

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 downcast is safe as we know that anything retrieved from a reusable class has to be locally allocated.

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 lifetimes[3]. 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]

```
class Top extends ScopeGate {
  class Bridge {
   Q q;
   void main(P p, Q q) {
    this.q = q;
      p.enter(this);
   }
   void callback(local Data d){
      q.handoff(d);
   }
  }
  class Q extends ScopeGate {
   void handoff(local Data d) { ...  }
  }
  ...
```



```
class P extends ScopeGate {
 class Data {}
 void enter(Bridge b) {
   b.callback(new Data());
 }
}
}
```

Figure 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] 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.

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 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.

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

$$
\begin{array}{rcl}
\mathtt{L} & ::= & \mathtt{class\ C\ extends\ D}\ \{\ \overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \} \\
\mathtt{M} & ::= & \mathtt{C\ m}\ (\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\ \mathtt{return\ e;}\ \} \\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{v} \mid \mathtt{new\ C()} \mid \mathtt{e.f} \mid \mathtt{e.f} := \mathtt{e} \mid \mathtt{e.m}(\overline{\mathtt{e}}) \mid (\mathtt{C})\ \mathtt{e} \\
\mathtt{v} & ::= & \ell \mid \mathtt{null} \\
\mathtt{S} & ::= & \mathtt{top} \mid \mathtt{S.s} \\
\mathtt{K} & ::= & \mathtt{S.c} \mid \mathtt{c} \\
\mathtt{C,D} & ::= & \mathtt{S} \mid \mathtt{K}
\end{array}
$$

Figure 14: ScopeJ Calculus Syntax.

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 $\mathtt{e.m}(\overline{\mathtt{e}})$, or a cast (C) e. We adopt FJ notation and use an over-bar to represent a finite (possibly empty) sequence. We write $\overline{\mathtt{f}}$ to denote the sequence $\mathtt{f}_1, \ldots, \mathtt{f}_n$ and similarly for $\overline{\mathtt{e}}$ and $\overline{\mathtt{v}}$. We write $\overline{\mathtt{C}}\,\overline{\mathtt{f}}$ to denote $\mathtt{C}_1\,\mathtt{f}_1, \ldots \mathtt{C}_n\,\mathtt{f}_n$ and $\overline{\mathtt{C}} <: \overline{\mathtt{D}}$ to denote $\mathtt{C}_1 <: \mathtt{D}_1, \ldots, \mathtt{C}_n <: \mathtt{D}_n$.

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.

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: $P \equiv \kappa$

$$\frac{\texttt{class C extends D}\ \{\ \overline{\texttt{C}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\ \} \quad \mathit{fields}(\texttt{D}) = (\overline{\texttt{D}}\ \overline{\texttt{g}})}{\mathit{fields}(\texttt{C}) = (\overline{\texttt{C}}\overline{\texttt{D}}\ \overline{\texttt{f}}\overline{\texttt{g}})}$$

$$\frac{\begin{array}{c}\texttt{class C extends D}\ \{\ \ldots;\ \overline{\texttt{M}}\ \} \\ \texttt{m}\ \text{not defined in}\ \overline{\texttt{M}}\end{array}}{\mathit{mtype}(\texttt{m},\texttt{C}) = \mathit{mtype}(\texttt{m},\texttt{D})} \qquad \frac{\begin{array}{c}\texttt{class C extends D}\ \{\ \ldots;\ \overline{\texttt{M}}\ \} \\ \texttt{C}_0\ \texttt{m}\ (\overline{\texttt{C}}\ \overline{\texttt{x}})\ \{\ \texttt{return e};\ \} \in \overline{\texttt{M}}\end{array}}{\mathit{mtype}(\texttt{m},\ \texttt{C}) = \overline{\texttt{C}} \to \texttt{C}_0}$$

$$\frac{\begin{array}{c}\texttt{class C extends D}\ \{\ \ldots;\ \overline{\texttt{M}}\ \} \\ \texttt{m}\ \text{not defined in}\ \overline{\texttt{M}}\end{array}}{\mathit{mbody}(\texttt{m},\texttt{C}) = \mathit{mbody}(\texttt{m},\texttt{D})} \qquad \frac{\begin{array}{c}\texttt{class C extends D}\ \{\ \ldots;\ \overline{\texttt{M}}\ \} \\ \texttt{C}_0\ \texttt{m}\ (\overline{\texttt{C}}\ \overline{\texttt{x}})\ \{\ \texttt{return e};\ \} \in \overline{\texttt{M}}\end{array}}{\mathit{mbody}(\texttt{m},\ \texttt{C}) = (\overline{\texttt{x}},\ \texttt{e})}$$

$$\frac{\begin{array}{c}\texttt{class D extends D}'\ \{\ \ldots;\ \overline{\texttt{M}}\ \} \\ \texttt{m}\ \text{not defined in}\ \overline{\texttt{M}}\end{array}}{\mathit{override}(\texttt{m},\texttt{C},\texttt{D})} \qquad \frac{\mathit{mtype}(\texttt{m},\texttt{C}) = \mathit{mtype}(\texttt{m},\texttt{D})}{\mathit{override}(\texttt{m},\texttt{C},\texttt{D})}$$

Figure 15: Auxiliary definitions

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, $\ell, \texttt{e}$, of a scope and an expression. We use $\overline{\ell\ \texttt{e}}$ to represent a call stack of the form $\epsilon \bullet \ell_0, \texttt{e}_0 \bullet \ldots \bullet \ell_\texttt{n}, \texttt{e}_\texttt{n}$. 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 $\ell.\texttt{m}(\overline{\texttt{v}})$, 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 $\mathit{refcount}(\ell',\ P')$ 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

(R-New)

$$\frac{\begin{array}{c} \mathit{fields}(\mathtt{C}) = (\overline{\mathtt{C}}\ \overline{\mathtt{f}}) \quad \overline{\mathtt{v}} = \overline{\mathtt{null}} \quad |\overline{\mathtt{v}}| = |\overline{\mathtt{f}}| \\ \ell_0 \text{ fresh} \quad \sigma' = \sigma[\ell_0 \mapsto \mathtt{C}^{\ell}(\overline{\mathtt{v}})] \end{array}}{\sigma, \ell, \mathtt{new\ C}() \ \rightarrow\ \sigma', \ell, \ell_0}$$

(R-Field)

$$\frac{\sigma(\ell_0) = \mathtt{C}^{\ell'}(\overline{\mathtt{v}}) \quad \mathit{fields}(\mathtt{C}) = (\overline{\mathtt{C}}\ \overline{\mathtt{f}})}{\sigma, \ell, \ell_0.\mathtt{f_i} \ \rightarrow\ \sigma, \ell, \mathtt{v_i}}$$

(R-Upd)

$$\frac{\begin{array}{c} \sigma(\ell_0) = \mathtt{C}^{\ell'}(\overline{\mathtt{v}}) \quad \mathit{fields}(\mathtt{C}) = (\overline{\mathtt{C}}\ \overline{\mathtt{f}}) \\ \sigma' = \sigma[\ell_0 \mapsto \mathtt{C}^{\ell'}(\mathtt{v}{\downarrow_{\mathtt{i}}}\overline{\mathtt{v}})] \end{array}}{\sigma, \ell, \ell_0.\mathtt{f_i} := \mathtt{v} \ \rightarrow\ \sigma', \ell, \mathtt{v}}$$

(R-Cast)

$$\frac{\mathtt{v} = \mathtt{null} \ \vee\ (\sigma(\mathtt{v}) = \mathtt{C_0}^{\ell'}(\overline{\mathtt{v}}) \ \wedge\ \mathtt{C_0}\ <:\ \mathtt{C})}{\sigma, \ell, (\mathtt{C})\ \mathtt{v} \ \rightarrow\ \sigma, \ell, \mathtt{v}}$$

(R-Invk)

$$\frac{\begin{array}{c} \sigma(\ell_0) = \mathtt{C}^{\ell'_0}(\overline{\mathtt{v}}') \quad \mathit{mbody}(\mathtt{m},\ \mathtt{C}) = (\overline{\mathtt{x}},\ \mathtt{e}') \\ \mathtt{e} = [\overline{\mathtt{v}}/_{\overline{\mathtt{x}}},\ {}^{\ell_0}/_{\mathtt{this}}]\mathtt{e}' \quad \mathit{inscope}_\sigma(\ell_0) = \ell' \end{array}}{\sigma, \ell, \ell_0.\mathtt{m}(\overline{\mathtt{v}}) \ \rightarrow\ \sigma, \ell', \mathtt{e}}$$

(G-Step)

$$\frac{\begin{array}{c} P = P'' \mid \kappa \bullet \ell, E[\mathtt{e}] \\ \mathtt{e} \neq \mathtt{v.m}(\overline{\mathtt{v}}) \quad \sigma, \ell, \mathtt{e} \ \rightarrow\ \sigma', \ell, \mathtt{e}' \\ P' = P'' \mid \kappa \bullet \ell, E[\mathtt{e}'] \end{array}}{\sigma, P \ \rightsquigarrow\ \sigma', P'}$$

(G-Enter)

$$\frac{\begin{array}{c} P = P'' \mid \kappa \bullet \ell, E[\mathtt{e}] \\ \mathtt{e} = \mathtt{v.m}(\overline{\mathtt{v}}) \quad \sigma, \ell, \mathtt{e} \ \rightarrow\ \sigma, \ell', \mathtt{e}' \\ P' = P'' \mid \kappa \bullet \ell, E[\mathtt{e}] \bullet \ell', \mathtt{e}' \end{array}}{\sigma, P \ \rightsquigarrow\ \sigma, P'}$$

(G-Ret)

$$\frac{\begin{array}{c} P = P'' \mid \kappa \bullet \ell, E[\mathtt{e}] \bullet \ell', \mathtt{v} \\ P' = P'' \mid \kappa \bullet \ell, E[\mathtt{v}] \\ \sigma' = \mathit{deallocate}(\sigma, \ell', P') \end{array}}{\sigma, P \ \rightsquigarrow\ \sigma', P'}$$

Evaluation contexts:

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

Active scope:

$$\mathit{inscope}_\sigma(\ell) = \ell \ \ \mathit{if}\ \ \sigma(\ell) = \mathtt{S}^{\ell'}(\overline{\mathtt{v}}) \qquad \mathit{inscope}_\sigma(\ell) = \ell' \ \ \mathit{if}\ \ \sigma(\ell) = \mathtt{K}^{\ell'}(\overline{\mathtt{v}})$$

Release scope memory:

$$\frac{\mathit{refcount}(\ell,\ P) \neq 0}{\mathit{deallocate}(\sigma, \ell, P) = \sigma} \qquad \frac{\begin{array}{c} \mathit{refcount}(\ell,\ P) = 0 \quad \sigma(\ell) = \mathtt{C}^{\ell_0}(\overline{\mathtt{v}}) \quad \{\ell_1..\ell_n\} = \{\ell' \mid \sigma(\ell') = \mathtt{C'}^{\ell}(\overline{\mathtt{v'}})\} \\ \sigma' = \sigma[\ell_1 \mapsto \mathtt{dummy}, .., \ell_n \mapsto \mathtt{dummy}, \ell \mapsto \mathtt{C}^{\ell_0}(\overline{\mathtt{null}})] \end{array}}{\mathit{deallocate}(\sigma, \ell, P) = \sigma'}$$

Reference counts:

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

Figure 16: Dynamic Semantics.

enforce left-to-right evaluation order and call-by-value semantics. Furthermore evaluation contexts are deterministic. For any expression $\mathtt{e}$, there is exactly one evaluation context usable in reduction rules. This can be shown by easy induction on the structure of $\mathtt{e}$ [43].

Expression evaluation is defined by a relation of the form $\sigma, \ell, \mathtt{e} \rightarrow \sigma', \ell', \mathtt{e}'$ 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, $\mathtt{C}^{\ell}(\overline{\mathtt{v}})$, is annotated with the instance ℓ representing the region in which it was allocated. $\mathit{inscope}_{\sigma}(\ell) = \ell'$ gives the scope associated with the object referenced by ℓ. In the case of normal objects this is their allocation scope, and for gates it is the gate itself.

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 $\mathtt{v}' \downarrow_{\mathtt{i}} \overline{\mathtt{v}}$ to denote the sequence $\overline{\mathtt{v}}$ with the ith entry replaced by $\mathtt{v}'$.

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

$$\frac{}{\mathtt{S.c}\ \mathit{viz}\ \mathtt{S}} \qquad \frac{}{\mathtt{S.c}\ \mathit{viz}\ \mathtt{S.c}'} \qquad \frac{}{\mathtt{S.c}\ \mathit{viz}\ \mathtt{S.S}'} \qquad \frac{}{\mathtt{S.c}\ \mathit{viz}\ \mathtt{S.S}'\mathtt{.c}'}$$

$$\frac{}{\mathtt{c}\ \mathit{viz}\ \mathtt{C}} \qquad \frac{}{\mathtt{S.s}\ \mathit{viz}\ \mathtt{S.c}} \qquad \frac{}{\mathtt{S.s}\ \mathit{viz}\ \mathtt{S}}$$

We also define contextual visibility $\mathtt{C}_0 \vdash \mathtt{C}\ \mathit{viz}\ \mathtt{D}$ which is used to prevent reusable types from crossing scope boundaries.

$$\frac{\mathtt{S}\ \mathit{viz}\ \mathtt{D}}{\mathtt{C}_0 \vdash \mathtt{S}\ \mathit{viz}\ \mathtt{D}} \qquad \frac{\mathtt{S.c}\ \mathit{viz}\ \mathtt{D}}{\mathtt{C}_0 \vdash \mathtt{S.c}\ \mathit{viz}\ \mathtt{D}} \qquad \frac{}{\mathtt{c}_0 \vdash \mathtt{c}\ \mathit{viz}\ \mathtt{D}} \qquad \frac{\mathit{scopeof}(\mathtt{C}) = \mathit{scopeof}(\mathtt{D})}{\mathtt{C} \vdash \mathtt{c}\ \mathit{viz}\ \mathtt{D}}$$

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 encloses 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 typable because applying the typing rule for method call to `g.get()` produces a contextual visibility constraint $\texttt{G.G1} \vdash \texttt{Object}\ \mathit{viz}\ \texttt{G}$, 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.

$$\texttt{C} <: \texttt{C} \qquad \frac{\texttt{C} <: \texttt{C}' \quad \texttt{C}' <: \texttt{C}''}{\texttt{C} <: \texttt{C}''} \qquad \frac{\texttt{class K extends K}'\ \{\ \ldots\ \}}{\texttt{K} <: \texttt{K}'}$$

We define a scope-safe subtyping relation between types $\preceq$ to restrict the use of widening. A scoped type $\texttt{S.c}$ may be widened to a reusable type $\texttt{c}'$ in the declaration of $\texttt{C}_0$, if the static scope of $\texttt{C}_0$ is also $\texttt{S}$.

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

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 `this.f = v` is not typable since the typing rule for updates (explained next) has the constraint $\mathtt{G.G1} \vdash \mathtt{G.C} \preceq \mathtt{Object}$, 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`).

4.2.1. Typing Classes and Methods

The type rule for a class `C` requires that the types of all fields be visible in the context of the class definition. A 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`).

$$\frac{\forall \mathtt{M} \in \overline{\mathtt{M}} \,.\, \mathtt{S\ extends\ c} \vdash \mathtt{M} \quad \overline{\mathtt{C}}\ \mathit{viz}\ \mathtt{C}}{\mathtt{class\ S\ extends\ c}\ \{\ \overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \}\ \mathtt{OK}} \qquad \text{(T-CLASS1)}$$

$$\frac{\forall \mathtt{M} \in \overline{\mathtt{M}} \,.\, \mathtt{c\ extends\ c'} \vdash \mathtt{M} \quad \overline{\mathtt{C}}\ \mathit{viz}\ \mathtt{c}}{\mathtt{class\ c\ extends\ c'}\ \{\ \overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \}\ \mathtt{OK}} \qquad \text{(T-CLASS2)}$$

$$\frac{\forall \mathtt{M} \in \overline{\mathtt{M}} \,.\, \mathtt{S.c\ extends\ D} \vdash \mathtt{M} \quad \overline{\mathtt{C}}\ \mathit{viz}\ \mathtt{S.c} \quad (\mathtt{D} = \mathtt{S.c'} \ \vee\ \mathtt{D} = \mathtt{c'})}{\mathtt{class\ S.c\ extends\ D}\ \{\ \overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \}\ \mathtt{OK}} \qquad \text{(T-CLASS3)}$$

A method of class $\mathtt{C}$ 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 $\mathtt{C}$. Finally, the method body is of a type that is a scope-safe subtype of the declared result type.

$$\frac{\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e} : \mathtt{C}'_\mathtt{r} \quad \mathtt{C} \vdash \mathtt{C}'_\mathtt{r} \preceq \mathtt{C}_\mathtt{r} \quad \mathtt{C}_\mathtt{r}\ \mathit{viz}\ \mathtt{C} \quad \overline{\mathtt{C}}\ \mathit{viz}\ \mathtt{C} \quad \mathit{override}(\mathtt{m}, \mathtt{C}, \mathtt{D})}{\mathtt{C}\ \mathtt{extends}\ \mathtt{D} \vdash \mathtt{C}_\mathtt{r}\ \mathtt{m}\ (\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\ \mathtt{return}\ \mathtt{e};\ \}} \tag{T-METH}$$

4.2.2. Typing Expressions

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

$$\frac{\Gamma(\mathtt{x})\ \mathit{viz}\ \Gamma(\mathtt{this})}{\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x})} \tag{T-VAR}$$

In FJ, the type rule of a cast expression $(\mathtt{C}_0)\ \mathtt{e}$ places no constraint on $\mathtt{e}$ 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 $\mathtt{e}$ has a scoped type with static scope $\mathtt{S}$ and the type $\mathtt{C}_0$ is reusable, then the static scope of the context $\mathtt{C}$ is $\mathtt{S}$. Similarly, for down-casts, if $\mathtt{C}_0$ is a scoped type in scope $\mathtt{S}$ and $\mathtt{e}$ has a reusable type, then the static scope of the context $\mathtt{C}$ is $\mathtt{S}$.

$$\frac{\mathtt{C} = \Gamma(\mathtt{this}) \quad \mathtt{C}_0\ \mathit{viz}\ \mathtt{C} \quad \Gamma \vdash \mathtt{e} : \mathtt{K} \quad \mathtt{C} \vdash \mathtt{C}_0 \preceq \mathtt{K} \vee \mathtt{C} \vdash \mathtt{K} \preceq \mathtt{C}_0}{\Gamma \vdash (\mathtt{C}_0)\ \mathtt{e} : \mathtt{C}_0} \tag{T-CAST}$$

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).

$$\frac{\mathtt{C} = \Gamma(\mathtt{this}) \quad \mathtt{C}_0\ \mathit{viz}\ \mathtt{C} \quad (\forall \mathtt{S.c},\ \mathtt{C}_0 = \mathtt{S.c} \Rightarrow \mathtt{S} = \mathit{scopeof}(\mathtt{C}))}{\Gamma \vdash \mathtt{new}\ \mathtt{C}_0() : \mathtt{C}_0} \tag{T-NEW}$$

A field selection expression $\mathtt{e.f_i}$ must abide by the normal FJ typing constraints. The type of the field $\mathtt{C_i}$ must be visible from $\mathtt{C}$, where $\mathtt{C}$ is the type of $\mathtt{this}$, 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 $\mathtt{C_i}$ is reusable, then from the condition $\mathtt{C}_0 \vdash \mathtt{C_i}\ \mathit{viz}\ \mathtt{C}$, either $\mathtt{e}$'s type $\mathtt{C}_0$ is reusable, or the static scopes of $\mathtt{C}_0$ and $\mathtt{C}$ are the same. In both cases, $\mathtt{e}$ and $\mathtt{f_i}$ refer to objects allocated in the current scope. Note that $\mathit{fields}(\mathtt{C})$ returns the field declarations $\overline{\mathtt{C}}\ \overline{\mathtt{f}}$ of the class $\mathtt{C}$. Lastly, the expression $\mathtt{e}$ is either well-typed or $\mathtt{e} = \mathtt{this}$. 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.

$$\frac{\begin{array}{c}\mathtt{C} = \Gamma(\mathtt{this}) \quad \Gamma \vdash \mathtt{e} : \mathtt{C}_0 \ \vee\ (\mathtt{e} = \mathtt{this} \ \wedge\ \mathtt{C}_0 = \mathtt{C}) \\ \mathit{fields}(\mathtt{C}_0) = (\overline{\mathtt{C}}\ \overline{\mathtt{f}}) \quad \mathtt{C}_0 \vdash \mathtt{C_i}\ \mathit{viz}\ \mathtt{C}\end{array}}{\Gamma \vdash \mathtt{e.f_i} : \mathtt{C_i}} \qquad \text{(T-FIELD)}$$

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

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

An invocation expression, $\mathtt{e.m}(\overline{\mathtt{e}})$, is well-typed under the condition that the return type of the method call must be visible from the current context $\mathtt{C} = \Gamma(\mathtt{this})$. 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 $\mathit{mtype}(\mathtt{m}, \mathtt{C})$ returns the parameters types $\overline{\mathtt{C}}$ and the result type $\mathtt{C_r}$ of the method m called on an object of type C.

$$\frac{\begin{array}{c}\mathtt{C} = \Gamma(\mathtt{this}) \quad \Gamma \vdash \mathtt{e} : \mathtt{C}_0 \ \vee\ (\mathtt{e} = \mathtt{this} \ \wedge\ \mathtt{C}_0 = \mathtt{C}) \\ \mathit{mtype}(\mathtt{m},\ \mathtt{C}_0) = \overline{\mathtt{C}} \rightarrow \mathtt{C_r} \quad (\forall \mathtt{S},\ \mathtt{C}_0 = \mathtt{S}\ \Rightarrow\ \mathtt{m}\ \text{defined in}\ \mathtt{C}_0) \\ \Gamma \vdash_{\mathtt{C}} \overline{\mathtt{e}} : \overline{\mathtt{C}'} \quad \mathtt{C} \vdash \overline{\mathtt{C}'} \preceq \overline{\mathtt{C}} \quad \mathtt{C}_0 \vdash \overline{\mathtt{C}}\ \mathit{viz}\ \mathtt{C} \quad \mathtt{C}_0 \vdash \mathtt{C_r}\ \mathit{viz}\ \mathtt{C}\end{array}}{\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{C_r}} \qquad \text{(T-INVK)}$$

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.

4.2.3. 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 borrowed types.

$$\begin{array}{lcl} \mathtt{M} & ::= & \mathtt{C}\,\mathtt{m}\,(\overline{\mathtt{T}}\,\overline{\mathtt{x}})\,\{\,\mathtt{return}\,\mathtt{e};\,\} \\ \mathtt{T} & ::= & \mathtt{C} \mid \mathtt{local}\,\mathtt{K} \end{array}$$

Borrowed types are visible everywhere: `local C` *viz* `D`. We need to supplement the scope-safe subtyping relation with the following rules:

$$\frac{\mathtt{C} <: \mathtt{D}}{\mathtt{C}_0 \vdash \mathtt{C} \preceq \mathtt{local}\,\mathtt{D}} \qquad \frac{\mathtt{C} <: \mathtt{D}}{\mathtt{C}_0 \vdash \mathtt{local}\,\mathtt{C} \preceq \mathtt{local}\,\mathtt{D}}$$

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.

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{local}\,\mathtt{K} \quad \mathit{fields}(\mathtt{K}) = (\overline{\mathtt{C}}\,\overline{\mathtt{f}}) \quad \mathtt{C}_\mathtt{i} \text{ is not a gate type}}{\Gamma \vdash \mathtt{e.f}_\mathtt{i} : \mathtt{local}\,\mathtt{C}_\mathtt{i}}$$

(T-FIELD2)

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

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{local\ K} \quad \mathit{mtype}(\mathtt{m},\ \mathtt{K}) = \overline{\mathtt{T}} \to \mathtt{K}_0 \quad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}'} \\ \Gamma(\mathtt{this}) \vdash \overline{\mathtt{T}'} \preceq \overline{\mathtt{T}} \quad \overline{\mathtt{T}} \text{ are borrowed types}}{\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{local\ K}_0}$$

(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.

4.3.1. 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.

$$\mathtt{ScopeStack}(\epsilon) = \epsilon \qquad \frac{\mathtt{ScopeStack}(\kappa) = \gamma}{\mathtt{ScopeStack}(\kappa \bullet \ell, \mathtt{e}) = \gamma \bullet \ell}$$

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

$$\sigma, \gamma \vdash \mathtt{null} \qquad \sigma, \gamma \vdash \mathtt{new\ C()} \qquad \frac{\sigma(\ell) = \mathtt{C}^{\ell'}(\overline{\mathtt{v}}) \quad \ell' \in \gamma}{\sigma, \gamma \vdash \ell} \qquad \frac{\sigma, \gamma \vdash \mathtt{e}}{\sigma, \gamma \vdash (\mathtt{C})\ \mathtt{e}}$$

$$\frac{\sigma, \gamma \vdash \mathtt{e}}{\sigma, \gamma \vdash \mathtt{e.f_i}} \qquad \frac{\sigma, \gamma \vdash \mathtt{e} \quad \sigma, \gamma \vdash \mathtt{e}'}{\sigma, \gamma \vdash \mathtt{e.f_i} := \mathtt{e}'} \qquad \frac{\sigma, \gamma \vdash \mathtt{e} \quad \forall \mathtt{i},\ \sigma, \gamma \vdash \mathtt{e_i}}{\sigma, \gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}})}$$

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.

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

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 $\ell, \mathtt{e}$ of a call stack κ, we can show in addition that for any object ℓ_0 referenced in $\mathtt{e}$, 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 $\mathtt{e}$ could be reduced from a borrowed parameter and be allocated in a scope of a frame below the frame $\ell, \mathtt{e}$ in κ.

4.3.2. Store invariant

We also want to maintain a safety invariant of the store such 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.

$$\frac{\sigma(\ell) = \mathtt{C}^{\ell'}(\overline{\mathtt{v}})}{\mathtt{type}_\sigma(\ell) = \mathtt{C}} \qquad \frac{\sigma(\ell) = \mathtt{C}^{\ell'}(\overline{\mathtt{v}})}{\mathtt{scope}_\sigma(\ell) = \ell'}$$

We also define a binary operator $\preceq_\sigma$ 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'.

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

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

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

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

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

4.3.3. Connection between the store and stack invariants:

A well-typed program needs to maintain both invariants so that deallocation of objects in a scope will not create dangling references.

The store invariant restricts the references held in objects' fields. In subject reduction of expressions (Lemma 1), we use the store invariant and typing rules to prove that field selection will not allow current thread to access objects in scopes that it has not entered. The stack invariant ensures that the objects referenced in a stack frame are allocated in scopes already entered by the current thread. Also in Lemma 1, we use the stack invariant and typing rules to show that field updates preserve the store invariant.

In subject reduction of programs (Lemma 6), both invariants are used in proving that deallocating of a scope does not create dangling pointers. Specifically, stack invariant is used to show that when a scope is cleared, deallocated objects are no longer referenced in the stacks of each thread. Store invariant is used to prove that when an object is removed from a scope, it is not referenced in the fields of any objects that are currently accessible from the stack.

4.3.4. Well-typed store:

The definition of well-typed store 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.

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

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

Also, if $\mathtt{v_i}$ is allocated in $\ell_\mathtt{i}$ and $\mathit{inscope}_\sigma(\ell) = \ell'_0$, then $\mathtt{locate}_{\sigma,\ell'_0}(\mathtt{C_i}) = \ell_\mathtt{i}$. 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 $\mathtt{C}$ and is allocated in ℓ_0, then $\mathtt{locate}_{\sigma,\ell_0}(\mathtt{C}) = \ell_0$. This requirement ensures that if $\mathtt{C}$ is a gate type then its instance is always allocated in an instance of its parent scope; if $\mathtt{C}$ is a scoped type in the scope $\mathtt{S}$, then its instance is always allocated in an instance of $\mathtt{S}$.

$$\frac{\begin{array}{c}\sigma(\ell) = \mathtt{C}^{\ell_0}(\overline{\mathtt{v}}) \quad \mathit{fields}(\mathtt{C}) = (\overline{\mathtt{C}}\ \overline{\mathtt{f}}) \quad \mathtt{locate}_{\sigma,\ell_0}(\mathtt{C}) = \ell_0 \quad \mathit{inscope}_\sigma(\ell) = \ell'_0 \\ \forall \mathtt{i},\ \mathtt{v_i} = \mathtt{null}\ \vee\ \sigma(\mathtt{v_i}) = \mathtt{C'_i}^{\ell_\mathtt{i}}(\overline{\mathtt{v^i}}),\ \mathtt{C'_i}\ <:\ \mathtt{C_i},\ \mathtt{locate}_{\sigma,\ell'_0}(\mathtt{C_i}) = \ell_\mathtt{i}\end{array}}{\sigma \vdash \ell}$$

4.3.5. 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 objects allocated in the scope are removed from the store.

$$\frac{\vdash \sigma \qquad P = \kappa_1 \mid \cdots \mid \kappa_n \qquad \sigma \vdash \kappa_1 \quad \dots \quad \sigma \vdash \kappa_n \qquad \forall \ell.\ \mathit{refcount}(\ell,\, P) = 0 \Rightarrow \nexists \ell' \text{ such that } \mathtt{scope}_\sigma(\ell') = \ell}{\sigma \vdash P}$$

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 $\sigma, \gamma \vdash \mathtt{e}$). This enforces the stack invariants mentioned earlier.

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

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

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

4.3.6. 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.

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

$$\sigma, \ell \vdash_{scope} \texttt{null} : \ell' \qquad \text{(S-Null)}$$

$$\sigma, \ell \vdash_{scope} \texttt{new C()} : \ell \qquad \text{(S-New)}$$

$$\frac{\texttt{scope}_\sigma(\ell') = \ell_0}{\sigma, \ell \vdash_{scope} \ell' : \ell_0} \qquad \text{(S-Var)}$$

$$\frac{\sigma, \ell \vdash_{scope} \texttt{e} : \ell' \quad \texttt{locate}_{\sigma,\ell}(\texttt{C}) = \ell'}{\sigma, \ell \vdash_{scope} (\texttt{C})\ \texttt{e} : \ell'} \qquad \text{(S-Cast)}$$

$$\frac{\sigma, \ell \vdash \texttt{e.f}_\texttt{i} : \texttt{T} \quad \texttt{e} \neq \ell \quad \sigma, \ell \vdash_{scope} \texttt{e} : \ell_0 \quad \texttt{locate}_{\sigma,\ell_0}(\texttt{T}) = \ell'}{\sigma, \ell \vdash_{scope} \texttt{e.f}_\texttt{i} : \ell'} \qquad \text{(S-Field)}$$

$$\frac{\sigma, \ell \vdash \ell.\texttt{f}_\texttt{i} : \texttt{C} \quad \texttt{locate}_{\sigma,\ell}(\texttt{C}) = \ell'}{\sigma, \ell \vdash_{scope} \ell.\texttt{f}_\texttt{i} : \ell'} \qquad \text{(S-Field2)}$$

$$\frac{\sigma, \ell \vdash_{scope} \texttt{e.f}_\texttt{i} : \ell' \quad \sigma, \ell \vdash_{scope} \texttt{e}' : \ell'}{\sigma, \ell \vdash_{scope} \texttt{e.f}_\texttt{i} := \texttt{e}' : \ell'} \qquad \text{(S-Upd)}$$

$$\frac{\sigma, \ell \vdash \texttt{e.m}(\overline{\texttt{e}}) : \texttt{T} \quad \texttt{e} \neq \ell \quad \sigma, \ell \vdash_{scope} \texttt{e} : \ell_0 \quad \texttt{locate}_{\sigma,\ell_0}(\texttt{T}) = \ell'}{\sigma, \ell \vdash_{scope} \texttt{e.m}(\overline{\texttt{e}}) : \ell'} \qquad \text{(S-Invk)}$$

$$\frac{\sigma, \ell \vdash \ell.\texttt{m}(\overline{\texttt{e}}) : \texttt{C} \quad \texttt{locate}_{\sigma,\ell}(\texttt{C}) = \ell'}{\sigma, \ell \vdash_{scope} \ell.\texttt{m}(\overline{\texttt{e}}) : \ell'} \qquad \text{(S-Invk2)}$$

Figure 17: Allocation scope for expressions

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 (see Lemma 6). Therefore, a well-typed program can make progress if it has no exceptions and its threads are not all irreducible values.

The reduction of well-typed program maintains the stack and store invariants, which ensure that deallocation of scoped objects is safe. Intuitively, in order for a scope s to be cleared, there cannot be any thread using s. This implies that no scope stack of a thread can contain s. From the stack invariant, we know that the call stack of a thread can only reference objects allocated in the scopes of the scope stack. Therefore, no thread can reference objects in s. This shows that there will not be any dangling references in the call stacks of threads. To see why there is no dangling references in the store, consider the store invariant that a non-gate object's fields can only reference objects in the same or outer scope; a gate object's fields can in addition reference objects allocated in scope represented by the gate. The reduction rule (G-Ret) resets the fields of the gate object representing s to null. So, any object with fields referencing objects in s has to be in the scope s or inner scopes of s. We had already established that s cannot be on the scope stack of any thread. So if there is an object of a thread with fields referencing objects in s, then the object has to be in an inner scope s' of s and s' has to be in the scope stack of the thread. However, the latter would contradict the stack invariant. The reason is that by stack invariant, the scope stack of a thread must be well-formed, which means that each scope on the scope stack is either the top scope or an outer scope of s' (including s).

We now explain the lemmas that leads to Theorem 1 that says well-typed program does not get stuck. The primary focus is to show that subject reduction of a program preserves the typing of the store and the stacks of the program (Lemma 6). Since well-typed program can make progress (Lemma 8), a well-typed program does not get stuck. To prove Lemma 6, we will first show that the reduction of an expression preserves typing (Lemma 1). We also show that subject reduction preserves the typing of store (Lemma 2). To prove the subject reduction preserves typing of stacks, we first prove that a well-typed expression has a corresponding scope (Lemma 3), this scope is on the scope stack of the current thread (Lemma 4), and the reduction of expressions does not change the scope (Lemma 5).

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

Lemma 1. *If* $\vdash \sigma$, $\sigma, \ell \vdash \mathtt{e} : \mathtt{T}$, $\sigma, \ell, \mathtt{e} \ \rightarrow\ \sigma', \ell', \mathtt{e}'$, *then* $\sigma', \ell' \vdash \mathtt{e}' : \mathtt{T}'$, $\mathtt{type}_{\sigma}(\ell) \vdash \mathtt{T}' \preceq \mathtt{T}$, *and if* $\mathtt{e} = \mathtt{v.m}(\overline{\mathtt{v}})$, *then* $\mathtt{T}'$ *is not a borrowed type.*

Lemma 2 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 2. *If* $\vdash \sigma$, $\sigma, \ell \vdash \mathtt{e} : \mathtt{T}$, $\sigma, \ell \vdash_{scope} \mathtt{e} : \ell'$, *and* $\sigma, \ell, \mathtt{e} \ \rightarrow\ \sigma', \ell, \mathtt{e}'$, *then* $\vdash \sigma'$.

Lemma 3 shows that if an expression $\mathtt{e}$ is well-typed, then we can find a scope to allocate the object reduced from $\mathtt{e}$. Moreover, if the type of $\mathtt{e}$ is not borrowed, then the allocation scope of $\mathtt{e}$ 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 3. *If* $\vdash \sigma$, $\sigma, \ell \vdash \mathtt{e} : \mathtt{T}$, *then* $\exists \ell'$ *such that* $\sigma, \ell \vdash_{scope} \mathtt{e} : \ell'$, *and if* $\mathtt{T}$ *is not borrowed type, then* $\mathtt{locate}_{\sigma,\ell}(\mathtt{T}) = \ell'$.

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

Lemma 4. *If* $\sigma \vdash \gamma$, $\sigma, \gamma \vdash \mathtt{e}$, $\gamma = \gamma' \bullet \ell$, *and* $\sigma, \ell \vdash_{scope} \mathtt{e} : \ell'$, *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 on 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 on the scope stack. Consequently, the stack invariant holds during evaluation steps.

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

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 throws cast exception if it is of the form $(\mathtt{C}')\ \ell$ where ℓ has type `C` but `C` is not a subtype of $\mathtt{C}'$. An expression throws null pointer exception if it is of the forms $\mathtt{null.m}(\overline{\mathtt{v}})$, `null.f`, or $\mathtt{null.f} := \mathtt{v}$. The lemma below proves that a well-typed expression that does not throw cast or null-pointer exceptions can be evaluated.

Lemma 7. *If $\vdash \sigma$, $\sigma, \ell \vdash \mathtt{e} : \mathtt{T}$, `e` has the form of `new C()`, $\mathtt{v.m}(\overline{\mathtt{v}})$, $(\mathtt{C})\ \mathtt{v}$, $\mathtt{v.f} := \mathtt{v}'$, or `v.f`, and it does not raise null pointer or cast exception, then $\exists \ell'$ and $\mathtt{e}'$, such that $\sigma, \ell, \mathtt{e} \rightarrow \sigma', \ell', \mathtt{e}'$.*

Proof. If `e` is a new expression, then it can be evaluated by Rule (R-New).

If `e` is a field selection or field update, since `e` does not cause null pointer exception, from $\vdash \sigma$ and $\sigma, \ell \vdash \mathtt{e} : \mathtt{T}$, it is clear that `v` is in the domain of σ, the field `f` is a member of the class of $\sigma(\mathtt{v})$, and `e` can be reduced by Rule (R-Upd) and (R-Field).

If `e` is a cast expression $(\mathtt{C})\ \mathtt{v}$, then since `e` does not throw cast expression, the type of `v` is a subtype of `C` or `v` is null. In both cases, the expression can be reduced by Rule (R-Cast).

The last case is `e` being a method invocation of the form $\mathtt{v.m}(\overline{\mathtt{v}})$. Since `e` does not throw null pointer exception, `v` is not null and from $\vdash \sigma$ and $\sigma, \ell \vdash \mathtt{e} : \mathtt{T}$, `v` is in the domain of σ and `m` is a method in the class of $\sigma(\mathtt{v})$. Thus, `e` can be reduced by Rule (R-Invk). □

A thread throws exceptions if it is in the form of $\kappa \bullet \ell, E[\mathtt{e}]$, where `e` throws null pointer or cast exceptions, and the thread is alive if it does not throw exceptions and `e` is not a value. 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 $\sigma, P \rightsquigarrow \sigma', P'$.*

The proof follows Lemma 7 and is omitted.

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 $\epsilon \bullet \ell, \mathtt{v}$ or P' throws 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 `letregion` ρ expression such that ρ is only in scope for definition nested within the scope. Each class can then be parameterized by a set of region parameters, one per enclosing scope. Method effects can be approximated by the set of scopes visible to the defining class.

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)*, pages 1–25, Oslo, Norway, 2004.

[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. *Real-time 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, Portland, Oregon, USA, October 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)*, Tucson, AZ, USA, 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, Portland, Oregon, USA, January, 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)*, pages 171-193, 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)*, pages 324–337, 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)*, pages 2–27, Budapest, Hungary, 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, Washington, DC, USA, 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)*, pages 292 – 310, Seattle, Washington, USA, November 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)*, pages 374–387, Anaheim, California, USA, 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*. pages 53–76, Budapest, Hungary, June 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, Vancouver, British Columbia, Canada, October 1998.

[18] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 28–53, Berlin, Germany, 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)*, pages 70–80, Snowbird, Utah, USA, June 2001.

[20] Dan Grossman. Type-safe multithreading in Cyclone. In *ACM Workshop on Types in Language Design and Implementation*, pages 13–25, New Orleans, LA, USA, 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)*. pages 141–152, Berlin, Germany, June 2002.

[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, Phoenix, Arizona, USA, 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. On Ownership and Accessibility. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 99–123, Nantes, France, October 2006.

[28] James Noble, John Potter, and Jan Vitek. Flexible alias protection. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP)*, pages 158–166, 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)*, pages 101–110, Vienna, Austria, May 2004.

[31] Filip Pizlo and Jan Vitek. An empirical evalutation of memory management alternatives for Real-time Java. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS)*, pages 35–46, Rio de Janeiro, Brazil, 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.

[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, Portland, Oregon, USA, October 2006.

[34] Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. StreamFlex: High-throughput stream programming in Java. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 211–228, Montreal, Canada, 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)*, pages 191–201, San Diego, California, June 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, Venice, Italy, January 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, San Diego, California, 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)*, pages 241–251, Lisbon, Portugal, December 2004.

[43] Tian Zhao, Jens Palsberg, and Jan Vitek. Type-based confinement. *Journal of Functional Programming*, 16(1):83–128, January 2006.

Appendix

$$\sigma, \ell \vdash \mathtt{null} : \mathtt{T} \qquad \text{(RT-NULL)}$$

$$\frac{\sigma(\ell_0) = \mathtt{C}^{\ell'}(\overline{\mathtt{v}}) \quad \mathtt{locate}_{\sigma,\ell}(\mathtt{C}) = \ell'}{\sigma, \ell \vdash \ell_0 : \mathtt{C}} \qquad \text{(RT-VAR)}$$

$$\frac{\sigma(\ell_0) = \mathtt{K}^{\ell'}(\overline{\mathtt{v}}) \quad \mathtt{locate}_{\sigma,\ell}(\mathtt{K}) \neq \ell'}{\sigma, \ell \vdash \ell_0 : \mathtt{local}\ \mathtt{K}} \qquad \text{(RT-VAR2)}$$

$$\frac{\begin{array}{c}\mathtt{C}\ \mathit{viz}\ \mathtt{type}_\sigma(\ell) \quad \sigma, \ell \vdash \mathtt{e} : \mathtt{K} \quad \forall \mathtt{c}, \mathtt{c}' \,.\, (\mathtt{C} = \mathtt{S.c}\ \wedge\ \mathtt{K} = \mathtt{S}'.\mathtt{c}') \Rightarrow \mathtt{S} = \mathtt{S}' \\ \forall \mathtt{c}, \mathtt{c}' \,.\, ((\mathtt{C} = \mathtt{c}\ \wedge\ \mathtt{K} = \mathtt{S.c}') \vee (\mathtt{K} = \mathtt{c}\ \wedge\ \mathtt{C} = \mathtt{S.c}')) \Rightarrow \mathtt{type}_\sigma(\ell) = \mathtt{S}\end{array}}{\sigma, \ell \vdash (\mathtt{C})\ \mathtt{e} : \mathtt{C}} \qquad \text{(RT-CAST)}$$

$$\frac{\mathtt{C}\ \mathit{viz}\ \mathtt{type}_\sigma(\ell) \quad \forall \mathtt{c} \,.\, (\mathtt{C} = \mathtt{S.c} \Rightarrow \mathtt{type}_\sigma(\ell) = \mathtt{S})}{\sigma, \ell \vdash \mathtt{new}\ \mathtt{C}() : \mathtt{C}} \qquad \text{(RT-NEW)}$$

$$\frac{\sigma, \ell \vdash \mathtt{e} : \mathtt{C} \vee (\mathtt{e} = \ell\ \wedge\ \mathtt{C} = \mathtt{type}_\sigma(\ell)) \quad \mathit{fields}(\mathtt{C}) = (\overline{\mathtt{C}}\ \overline{\mathtt{f}}) \quad \mathtt{C} \vdash \mathtt{C_i}\ \mathit{viz}\ \mathtt{type}_\sigma(\ell)}{\sigma, \ell \vdash \mathtt{e.f_i} : \mathtt{C_i}} \qquad \text{(RT-FIELD)}$$

$$\frac{(\sigma, \ell \vdash \mathtt{e} : \mathtt{K}) \vee (\sigma, \ell \vdash \mathtt{e} : \mathtt{local}\ \mathtt{K}) \quad \mathit{fields}(\mathtt{K}) = (\overline{\mathtt{C}}\ \overline{\mathtt{f}}) \quad \mathtt{C_i} \text{ is not a gate type}}{\sigma, \ell \vdash \mathtt{e.f_i} : \mathtt{local}\ \mathtt{C_i}} \qquad \text{(RT-FIELD2)}$$

$$\frac{\sigma, \ell \vdash \mathtt{e.f_i} : \mathtt{C_i} \quad \sigma, \ell \vdash \mathtt{e}' : \mathtt{C}' \quad \mathtt{type}_\sigma(\ell) \vdash \mathtt{C}' \preceq \mathtt{C_i}}{\sigma, \ell \vdash \mathtt{e.f_i} := \mathtt{e}' : \mathtt{C}'} \qquad \text{(RT-UPD)}$$

$$\frac{\begin{array}{c}\sigma, \ell \vdash \mathtt{e} : \mathtt{C_0} \vee (\mathtt{e} = \ell\ \wedge\ \mathtt{C_0} = \mathtt{type}_\sigma(\ell)) \quad \mathit{mtype}(\mathtt{m},\ \mathtt{C_0}) = \overline{\mathtt{T}} \to \mathtt{C} \\ \sigma, \ell \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}'} \quad \mathtt{type}_\sigma(\ell) \vdash \overline{\mathtt{T}'} \preceq \overline{\mathtt{T}} \quad \mathtt{C_0} \vdash \overline{\mathtt{T}}\ \mathit{viz}\ \mathtt{type}_\sigma(\ell) \\ \mathtt{C_0} \vdash \mathtt{C}\ \mathit{viz}\ \mathtt{type}_\sigma(\ell) \quad \text{if } \mathtt{C_0} \text{ is a gate type, then } \mathtt{m} \text{ is defined in } \mathtt{C_0}\end{array}}{\sigma, \ell \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{C}} \qquad \text{(RT-INVK)}$$

$$\frac{\begin{array}{c}(\sigma, \ell \vdash \mathtt{e} : \mathtt{K}) \vee (\sigma, \ell \vdash \mathtt{e} : \mathtt{local}\ \mathtt{K}) \quad \mathit{mtype}(\mathtt{m},\ \mathtt{K}) = \overline{\mathtt{T}} \to \mathtt{K_0} \\ \sigma, \ell \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}'} \quad \mathtt{type}_\sigma(\ell) \vdash \overline{\mathtt{T}'} \preceq \overline{\mathtt{T}} \quad \overline{\mathtt{T}} \text{ are borrowed types}\end{array}}{\sigma, \ell \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{local}\ \mathtt{K_0}} \qquad \text{(RT-INVK2)}$$

Figure 18: Runtime type rules.

7. Subject Reduction of Programs

In this section, we prove Lemma 6 and its supporting lemmas.

Recall that in each frame $\ell, \mathtt{e}$ of a call stack, ℓ is always a gate instance and represents a scope. In the proofs, we sometimes refer to ℓ as the scope that it represents. Also, when there is no confusion, we refer to the static scope of a class simply as scope.

7.1. Subject reduction of expressions

We prove that subject reduction preserves the typing of expressions. The proof is by case analysis on the reduction rule used. The interesting cases are field selection and method call. In the former case, we show that if expression $\mathtt{v.f}$ in scope ℓ reduces to a value $\mathtt{v}'$, then $\mathtt{v}'$ can be safely accessed in ℓ. That is either $\mathtt{v}'$ has a borrowed type, or it is null, or $\mathtt{locate}_{\sigma,\ell}(\mathtt{C}') = \ell'$, where $\mathtt{C}'$ is the type of $\mathtt{v}'$ and ℓ' is its allocation scope.

The case of (R-Invk) is more complex and we factor out a portion of the proof to Lemma 9, which in turn depends on Lemma 10. In this case, we show that if an expression $\mathtt{v.m}(\overline{\mathtt{v}})$ of type $\mathtt{T}$ in scope ℓ evaluates to $\mathtt{e}$ of type $\mathtt{T}'$ in scope ℓ', then $\overline{\mathtt{v}}$ can be safely accessed in ℓ'. Also, $\mathtt{T}'$ is a safe subtype of that $\mathtt{T}$ so that if $\mathtt{T}$ is a reusable type and $\mathtt{T}'$ is a scoped type in static scope $\mathtt{S}$, then the type of ℓ is $\mathtt{S}$. This result is used later to show that return value of a method call can be safely accessed in the scope of the caller.

Lemma 1. *If* $\vdash \sigma$, $\sigma, \ell \vdash \mathtt{e} : \mathtt{T}$, $\quad \sigma, \ell, \mathtt{e} \rightarrow \sigma', \ell', \mathtt{e}'$, $\quad$ *then* $\sigma', \ell' \vdash \mathtt{e}' : \mathtt{T}'$, $\mathtt{type}_\sigma(\ell) \vdash \mathtt{T}' \preceq \mathtt{T}$*, and if* $\mathtt{e} = \mathtt{v.m}(\overline{\mathtt{v}})$*, then* $\mathtt{T}'$ *is not a borrowed type.*

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

R-FIELD Let $\mathtt{e} = \ell_0.\mathtt{f_i}$, $\sigma(\ell_0) = \mathtt{C_0}^{\ell'_0}(\overline{\mathtt{v}})$, *fields*$(\mathtt{C_0}) = (\overline{\mathtt{C}}\,\overline{\mathtt{f}})$, $\sigma, \ell \vdash \ell_0 : \mathtt{T_0}$, and $\mathtt{e}' = \mathtt{v_i}$. We need to show $\sigma, \ell \vdash \mathtt{v_i} : \mathtt{T}'$ such that $\mathtt{C} \vdash \mathtt{T}' \preceq \mathtt{T}$, where $\mathtt{C} = \mathtt{type}_\sigma(\ell)$, $\mathtt{T} = \mathtt{C_i}$ or $\mathtt{T} = \mathtt{local\ C_i}$.

In case $\mathtt{v_i}$ is null, the proof is trivial since it can have any type. Now let $\mathtt{v_i}$ be a reference $\ell_\mathtt{i}$ and let its type be $\mathtt{C}'_\mathtt{i}$.

If $\mathtt{T}$ is a borrowed type, then by Rule (RT-Field2), $\mathtt{T} = \mathtt{local\ C_i}$. By Rule (RT-Var2), $\sigma, \ell \vdash \ell_\mathtt{i} : \mathtt{local\ C}'_\mathtt{i}$. From $\vdash \sigma$, we have $\sigma \vdash \ell_0$, which implies $\mathtt{C}'_\mathtt{i} <: \mathtt{C_i}$. Thus, we have $\mathtt{C} \vdash \mathtt{T}' \preceq \mathtt{T}$, where $\mathtt{T}' = \mathtt{local\ C}'_\mathtt{i}$.

In case that $\mathtt{T}$ is not a borrowed type, then by Rule (RT-Field), $\mathtt{T} = \mathtt{C_i}$ and $\mathtt{T_0}$ must not be a borrowed type either. Let $\mathtt{T}' = \mathtt{C}'_\mathtt{i}$. By Rule (RT-Var), $\sigma, \ell \vdash \ell_\mathtt{i} : \mathtt{T}'$ if $\mathtt{locate}_{\sigma,\ell}(\mathtt{T}') = \ell'_\mathtt{i}$, where $\ell'_\mathtt{i}$ is the allocation scope of $\ell_\mathtt{i}$.

From $\sigma \vdash \ell_0$, we have $\mathtt{locate}_{\sigma,\ell''_0}(\mathtt{T}) = \ell'_\mathtt{i}$, where *inscope*$_\sigma(\ell_0) = \ell''_0$. Since ℓ_0 is well-typed, by Rule (RT-Var), we have $\mathtt{locate}_{\sigma,\ell}(\mathtt{T_0}) = \ell'_0$, where ℓ'_0 is the allocation scope of ℓ_0.

If $\mathtt{T_0}$ is a scoped or reusable type, then $\ell''_0 = \ell'_0$. If $\mathtt{T}$ is a scoped type, then it must be in the scope $\mathtt{C}$ or its outer scopes. If $\mathtt{T}$ is a gate type, then it must be a direct inner scope of $\mathtt{C}$. Rule (RT-Field) has the condition that $\mathtt{T_0} \vdash \mathtt{T}\ \mathit{viz}\ \mathtt{C}$. If $\mathtt{T}$ is a reusable type, then $\mathtt{T_0}$ is either a reusable type or be in the scope $\mathtt{C}$. In all cases, we have $\mathtt{locate}_{\sigma,\ell}(\mathtt{T}) = \ell'_\mathtt{i}$.

If $\ell = \ell_0$, then from $\sigma \vdash \ell_0$, we have $\mathtt{locate}_{\sigma,\ell}(\mathtt{T}) = \ell'_\mathtt{i}$.

If $\mathtt{T_0}$ is a gate type but $\ell \neq \ell_0$, then $\ell''_0 = \ell_0$ and $\mathtt{locate}_{\sigma,\ell_0}(\mathtt{T}) = \ell'_\mathtt{i}$. Also, ℓ_0 must be a direct inner scope of ℓ. In this case, $\mathtt{T}$ must be a scoped type defined in the scope $\mathtt{C}$ or its outer scopes. So, we also have $\mathtt{locate}_{\sigma,\ell}(\mathtt{T}) = \ell'_\mathtt{i}$.

From $\sigma \vdash \ell_\mathtt{i}$, $\mathtt{locate}_{\sigma,\ell'_\mathtt{i}}(\mathtt{T}') = \ell'_\mathtt{i}$. Together with $\mathtt{T}' <: \mathtt{T}$, we get $\mathtt{locate}_{\sigma,\ell}(\mathtt{T}') =$

$\ell'_{\mathtt{i}}$. Thus, by Rule (RT-Var2), $\sigma, \ell \vdash \ell_{\mathtt{i}} : \mathtt{T}'$ and $\mathtt{C} \vdash \mathtt{T}' \preceq \mathtt{T}$.

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

R-NEW Let $\mathtt{e} = \mathtt{new}\ \mathtt{C}()$. By Rule (R-New), $\mathtt{e}' = \ell'$ where $\sigma' = \sigma[\ell' \mapsto \mathtt{C}^{\ell}(\overline{\mathtt{null}})]$. Suppose $\mathtt{type}_\sigma(\ell) = \mathtt{S}$. Then by Rule (RT-New), $\mathtt{C}\ \textit{viz}\ \mathtt{S}$ and if $\mathtt{C} = \mathtt{S}'.\mathtt{c}$, then $\mathtt{S}' = \mathtt{S}$. Thus, $\mathtt{C}$ has the forms of $\mathtt{c}$, $\mathtt{S.s}$, or $\mathtt{S.c}$. Consequently, $\mathtt{locate}_{\sigma,\ell}(\mathtt{C}) = \ell$. By Rule (RT-Var), we have $\sigma, \ell \vdash \ell' : \mathtt{C}$.

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

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

Let $\textit{inscope}_\sigma(\ell_0) = \ell'$ and $\sigma, \ell \vdash \mathtt{e} : \mathtt{T}$, where $\mathtt{T} = \mathtt{local}\ \mathtt{C}$ or $\mathtt{T} = \mathtt{C}$ for some $\mathtt{C}$. By Lemma 9, $\exists \mathtt{C}'$ such that $\sigma, \ell' \vdash \mathtt{e}' : \mathtt{C}'$ and $\mathtt{type}_\sigma(\ell') \vdash \mathtt{C}' \preceq \mathtt{C}$.

If $\mathtt{T}$ is a borrowed type, then it follows $\mathtt{type}_\sigma(\ell) \vdash \mathtt{C}' \preceq \mathtt{T}$.

If $\mathtt{T}$ is not a borrowed type, then $\mathtt{T} = \mathtt{C}$, and by Rule (RT-Invk), $\mathtt{T_0}$ is not a borrowed type and $\mathtt{T_0} \vdash \mathtt{C}\ \textit{viz}\ \mathtt{type}_\sigma(\ell)$. From $\sigma, \ell \vdash \ell_0 : \mathtt{T_0}$, we have $\mathtt{locate}_{\sigma,\ell}(\mathtt{T_0}) = \ell'_0$. We need to show $\mathtt{type}_\sigma(\ell) \vdash \mathtt{C}' \preceq \mathtt{C}$.

The only interesting case is when $\mathtt{C}$ is a reusable type while $\mathtt{C}'$ is a scoped type.

If $\mathtt{T_0}$ is a gate type, then ℓ_0 cannot be an inner scope of ℓ because it would contradict $\mathtt{T_0} \vdash \mathtt{C}\ \textit{viz}\ \mathtt{type}_\sigma(\ell)$. Thus, $\ell_0 = \ell = \ell'$. If $\mathtt{T_0}$ is a reusable type, $\ell' = \ell$. If $\mathtt{T_0}$ is a scoped type, from $\mathtt{T_0} \vdash \mathtt{C}\ \textit{viz}\ \mathtt{type}_\sigma(\ell)$, the scope of $\mathtt{T_0}$ is $\mathtt{type}_\sigma(\ell)$. From $\mathtt{locate}_{\sigma,\ell}(\mathtt{T_0}) = \ell'_0$, we have $\ell' = \ell'_0 = \ell$.

In all cases, we have $\mathtt{type}_\sigma(\ell) \vdash \mathtt{C}' \preceq \mathtt{C}$.

□

The following lemma is used to prove the previous lemma's method invocation case. The reduction of a method call in scope ℓ can move evaluation to a different scope ℓ'. So we prove separately that the resulting expression of a method call has a type in the scope ℓ' and the type is a safe subtype of the method return type.

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

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

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

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

If $\mathtt{T_i} = \mathtt{C_i}$, then $\mathtt{T}'_\mathtt{i} = \mathtt{C}'_\mathtt{i}$. From $\sigma, \ell \vdash \ell_0 : \mathtt{C_0}$, $\mathtt{locate}_{\sigma,\ell}(\mathtt{C_0}) = \ell'_0$, where

$\texttt{scope}_\sigma(\ell_0) = \ell_0'$. From Rule (T-Meth), $\texttt{C}_\texttt{i}$ *viz* $\texttt{C}_0'$ and from class typing rules, we have $\texttt{C}_\texttt{i}$ *viz* $\texttt{C}_0$. Together with $\texttt{type}_\sigma(\ell) \vdash \texttt{C}_\texttt{i}' \preceq \texttt{C}_\texttt{i}$ and $\texttt{C}_0 \vdash \texttt{C}_\texttt{i}$ *viz* $\texttt{type}_\sigma(\ell)$, we can conclude that $\texttt{locate}_{\sigma,\ell'}(\texttt{C}_\texttt{i}') = \texttt{locate}_{\sigma,\ell}(\texttt{C}_\texttt{i}')$. Thus, $\sigma, \ell' \vdash \texttt{v}_\texttt{i} : \texttt{C}_\texttt{i}'$ and $\texttt{type}_\sigma(\ell') \vdash \texttt{C}_\texttt{i}' \preceq \texttt{C}_\texttt{i}$..

Suppose $\texttt{T} = \texttt{local C}$. Then $\sigma, \ell, \texttt{e} \rightarrow \sigma, \ell, \texttt{e}'$ and $\sigma, \ell \vdash \ell_0 : \texttt{local C}_0$. By Rule (RT-Invk2), $\mathit{mtype}(\texttt{C}_0, \texttt{m}) = \overline{\texttt{T}} \rightarrow \texttt{C}$, $\forall \texttt{i}$, $\texttt{T}_\texttt{i} = \texttt{local C}_\texttt{i}$, $\sigma, \ell \vdash \texttt{v}_\texttt{i} : \texttt{T}_\texttt{i}'$, where $\texttt{T}_\texttt{i}' = \texttt{local C}_\texttt{i}'$ or $\texttt{C}_\texttt{i}'$, and $\texttt{type}_\sigma(\ell) \vdash \texttt{T}_\texttt{i}' \preceq \texttt{T}_\texttt{i}$. It is clear that $\sigma, \ell' \vdash \texttt{v}_\texttt{i} : \texttt{T}_\texttt{i}''$, where $\texttt{T}_\texttt{i}'' = \texttt{local C}_\texttt{i}'$ or $\texttt{C}_\texttt{i}'$. Thus, $\texttt{type}_\sigma(\ell) \vdash \texttt{T}_\texttt{i}'' \preceq \texttt{T}_\texttt{i}$.

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

The following lemma shows that substituting formal parameters in a method body with actuals preserves typing and the type of the resulting expression is a safe subtype of the method return type.

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

Proof. We prove by induction on the structure of $\texttt{e}_0$.

- If $\texttt{e}_0 = \texttt{x}_\texttt{i} \in \overline{\texttt{x}}$, then, $\texttt{e} = \texttt{v}_\texttt{i}$. By assumption, $\sigma, \ell \vdash \texttt{v}_\texttt{i} : \texttt{T}_\texttt{i}'$, where $\texttt{type}_\sigma(\ell) \vdash \texttt{T}_\texttt{i}' \preceq \texttt{T}_\texttt{i}$.

  If $\texttt{e}_0 = \texttt{this}$, then $\texttt{e} = \ell_0$ and by Rule (T-Var), we have $\texttt{C}_0'$ *viz* $\texttt{C}_0'$, which means that $\texttt{C}_0'$ is not a gate type. From $\texttt{C}_0' <: \texttt{C}_0$, $\texttt{C}_0$ is not a gate type either. From $\mathit{inscope}_\sigma(\ell_0) = \ell$, we have $\texttt{locate}_{\sigma,\ell_0}(\texttt{C}_0) = \ell$. By Rule (RT-Var), we have $\sigma, \ell \vdash \texttt{e} : \texttt{C}_0$. Since ℓ is the allocation scope of ℓ_0, it is clear that $\texttt{type}_\sigma(\ell) \vdash \texttt{C}_0 \preceq \texttt{C}_0'$.

- If $\texttt{e}_0 = \texttt{new C}()$, then $\texttt{e} = \texttt{e}_0$. By assumption $\mathit{inscope}_\sigma(\ell_0) = \ell$. Thus, $\texttt{C}$ *viz* $\texttt{C}_0$ implies $\texttt{C}$ *viz* $\texttt{type}_\sigma(\ell)$. Also, $\mathit{scopeof}(\texttt{C}_0) = \mathit{scopeof}(\texttt{type}_\texttt{S}(\ell))$. Therefore, by Rule (RT-New) $\sigma, \ell \vdash \texttt{e} : \texttt{C}$.

- Suppose $\texttt{e}_0 = (\texttt{C})\ \texttt{e}_0'$. Then $\texttt{e} = (\texttt{C})\ \texttt{e}'$, where $\texttt{e}' = [\overline{\texttt{v}}/\overline{\texttt{x}}, {}^{\ell_0}/_{\texttt{this}}]\texttt{e}_0'$. By hypothesis, $\exists \texttt{K}$ such that $\Gamma \vdash \texttt{e}_0 : \texttt{K}$ where $\texttt{C}_0' \vdash \texttt{C} \preceq \texttt{K}$ or $\texttt{C}_0 \vdash \texttt{K} \preceq \texttt{C}$. By induction hypothesis, $\exists \texttt{K}'$ such that $\sigma, \ell \vdash \texttt{e}_0' : \texttt{K}'$ where $\texttt{type}_\sigma(\ell) \vdash \texttt{K}' \preceq \texttt{K}$.

  Thus, if $\texttt{C}_0' \vdash \texttt{K} \preceq \texttt{C}$, then $\texttt{type}_\sigma(\ell) \vdash \texttt{K}' \preceq \texttt{C}$. If $\texttt{C}_0' \vdash \texttt{C} \preceq \texttt{K}$, then either $\texttt{K} = \texttt{S.c}'$ and $\texttt{C} = \texttt{S.c}$, or $\texttt{K} = \texttt{c}'$, $\texttt{C} = \texttt{S.c}$, and $\mathit{scopeof}(\texttt{C}_0') = \texttt{S}$, or $\texttt{C} = \texttt{c}$ and $\texttt{K} = \texttt{c}'$. From $\texttt{type}_\sigma(\ell) \vdash \texttt{K}' \preceq \texttt{K}$, either $\texttt{K} = \texttt{S.c}'$ and $\texttt{K}' = \texttt{S.c}''$, or $\texttt{K} = \texttt{c}'$, $\texttt{K}' = \texttt{S.c}''$, and $\texttt{type}_\sigma(\ell) = \texttt{S}$. Therefore, we have either $\texttt{C} = \texttt{S.c}$ and $\texttt{K}' = \texttt{S.c}''$, or $\texttt{C} = \texttt{c}$, $\texttt{K}' = \texttt{S.c}''$, and $\texttt{type}_\sigma(\ell) = \texttt{S}$. Hence by Rule (RT-Cast), we have $\sigma, \ell \vdash \texttt{e}' : \texttt{C}$.

- Suppose $\texttt{e}_0 = \texttt{e}_\texttt{r}.\texttt{f}_\texttt{i}$. Then $\texttt{e} = \texttt{e}_\texttt{r}'.\texttt{f}_\texttt{i}$, where $\texttt{e}_\texttt{r}' = [\overline{\texttt{v}}/\overline{\texttt{x}}, {}^{\ell_0}/_{\texttt{this}}]\texttt{e}_\texttt{r}{}'$.

  There are three cases:

  1. $\texttt{e}_\texttt{r} = \texttt{this}$ and $\Gamma(\texttt{this})$ is a gate type.

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

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

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

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

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

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

  There are three cases.

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

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

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

  3. $\Gamma \vdash \mathtt{e_r} : \mathtt{local}\ \mathtt{C_r}$. By induction hypothesis, $\exists \mathtt{T_r}$ such that $\sigma, \ell \vdash \mathtt{e_r}' : \mathtt{T_r}$ and $\mathtt{type}_\sigma(\ell) \vdash \mathtt{T_r} \preceq \mathtt{local}\ \mathtt{C_r}$. By Rule (T-Invk2), $\overline{\mathtt{T}}$ are borrowed types. Thus, by Rule (RT-Invk2), we have $\sigma, \ell \vdash \mathtt{e} : \mathtt{local}\ \mathtt{C}$.

□

7.2. Subject reduction preserves store invariant

We prove that the reduction of a well-typed expression preserves the typing of the store. The interesting cases are new expression and field update. A new expression $\mathtt{new}\ \mathtt{C}()$ reduces to an object allocated in the scope of the current context. The reduction step adds an object to the store and the store invariant holds because Rule (RT-New) ensures that $\mathtt{C}$ can only be either a reusable type, or a scoped or gate type within the static scope of the current context. A field update $\ell_0.\mathtt{f} := \ell$ modifies the store by changing the field of an object ℓ_0 to reference ℓ. The store invariant holds for the new store if allocation scope of ℓ matches the one that ℓ_0 expects for the field type. This is true because Rule (RT-Upd) ensures that the type of ℓ is a safe subtype of the field type, which is also visible in the current context. Together with the fact that both ℓ_0 and ℓ are accessible in the current context, we can conclude that it is safe for ℓ_0's field to reference ℓ.

Lemma 2. *If* $\vdash \sigma,\ \ \sigma, \ell \vdash \mathtt{e} : \mathtt{T},\ \ \sigma, \ell \vdash_{scope} \mathtt{e} : \ell'$*, and* $\sigma, \ell, \mathtt{e} \ \rightarrow\ \sigma', \ell, \mathtt{e}'$*, then* $\vdash \sigma'$*.*

Proof. The only cases that $\sigma' \neq \sigma$ are when $\mathtt{e}$ has the forms of $\mathtt{new}\ \mathtt{C}()$ or $\ell_0.\mathtt{f_i} := \mathtt{v}$. In both cases, $\mathtt{T}$ cannot be borrowed type. So let $\mathtt{T} = \mathtt{C}$ for some $\mathtt{C}$.

Consider the case of $\mathtt{e} = \mathtt{new}\ \mathtt{C}()$. By Rule (S-New), we have $\sigma, \ell \vdash_{scope} \mathtt{e} : \ell$. By Rule (RT-New), if $\mathtt{C}$ is a scoped type, then it must be in the scope $\mathtt{S}$ and it must be visible from the $\mathtt{S}$, where $\mathtt{S}$ is the type of ℓ. Therefore, we have $\mathtt{locate}_{\sigma,\ell}(\mathtt{C}) = \ell$. By Rule (R-New), $\sigma, \ell, \mathtt{new}\ \mathtt{C}() \ \rightarrow\ \sigma', \ell, \ell_0$, where $\sigma'(\ell_0) = \mathtt{C}^{\ell}(\overline{\mathtt{null}})$. Since $\sigma' = \sigma[\ell_0 \mapsto \mathtt{C}^{\ell}(\overline{\mathtt{null}})]$ and $\vdash \sigma$, we have $\sigma' \vdash \ell_0$ and consequently $\vdash \sigma'$.

Consider the case that $\mathtt{e}$ equals to $\ell_0.\mathtt{f_i} := \mathtt{v}$. By Rule (R-Upd), $\sigma, \ell, \mathtt{e} \ \rightarrow\ \sigma', \ell, \mathtt{v}$, where $\sigma(\ell_0) = \mathtt{C_0}^{\ell'_0}(\overline{\mathtt{v}})$ and $\sigma' = \sigma[\ell_0 \mapsto \mathtt{C_0}^{\ell'_0}(\mathtt{v} \downarrow_{\mathtt{i}} \overline{\mathtt{v}})]$. Let $\mathtt{v} = \ell_{\mathtt{i}}$ and $\sigma(\ell_{\mathtt{i}}) = \mathtt{C'}^{\ell'}(\overline{\mathtt{v}'})$.

To provide $\sigma' \vdash \ell_0$, we need to show $\mathtt{locate}_{\sigma,\ell''_0}(\mathtt{C}') = \ell'$, where $\mathit{inscope}_{\sigma}(\ell_0) = \ell''_0$.

From Rule (RT-Upd) and (RT-Var), we have $\mathtt{locate}_{\sigma,\ell}(\mathtt{C}') = \ell'$ and $\mathtt{type}_{\sigma}(\ell) \vdash \mathtt{C}' \ \preceq\ \mathtt{C}$.

If $\mathtt{C_0}$ is not a gate type, then $\mathit{inscope}_{\sigma}(\ell_0) = \ell'_0$ and $\ell''_0 = \ell'_0$. From Rule (RT-Field), we have $\mathtt{locate}_{\sigma,\ell}(\mathtt{C_0}) = \ell'_0$. From Rule (RT-Upd) and (RT-Var), we have $\mathtt{locate}_{\sigma,\ell}(\mathtt{C}') = \ell'$. From Rule (T-Class), $\mathtt{C}$ is visible from $\mathtt{C_0}$. Together with $\mathtt{type}_{\sigma}(\ell) \vdash \mathtt{C}' \ \preceq\ \mathtt{C}$. we have $\mathtt{locate}_{\sigma,\ell''_0}(\mathtt{C}') = \ell'$.

If $\mathtt{C_0}$ is a gate type, then $\mathit{inscope}_{\sigma}(\ell_0) = \ell_0$ and $\ell_0 = \ell''_0$. If $\ell_0 = \ell$, then $\ell''_0 = \ell$ and it follows $\mathtt{locate}_{\sigma,\ell''_0}(\mathtt{C}') = \ell'$. Consider the case $\ell_0 \neq \ell$. From Rule (RT-Field), we have $\mathtt{locate}_{\sigma,\ell}(\mathtt{C_0}) = \ell'_0$. Let $\mathtt{S} = \mathtt{type}_{\sigma}(\ell)$. Since $\mathtt{C_0}$ is a gate type, $\ell'_0 = \ell$ and $\mathtt{C_0}$ has the form of $\mathtt{S.s}$. From Rule (RT-Field), $\mathtt{C_0} \vdash \mathtt{C}\ \mathit{viz}\ \mathtt{S}$. Thus, $\mathtt{C}$ may only have the form of $\mathtt{S'.c}$, where $\mathtt{S}' = \mathtt{S}$ or $\mathtt{S}'$ encloses $\mathtt{S}$. Therefore, $\mathtt{locate}_{\sigma,\ell_0}(\mathtt{C}) = \mathtt{locate}_{\sigma,\ell'_0}(\mathtt{C}) = \ell'$.

It follows that $\sigma' \vdash \ell_0$ and $\vdash \sigma'$. □

7.3. A well-type expression has a corresponding scope

Objects have their scopes but we also need to find scopes for expressions to prove that subject reduction preserves stack invariant. Lemma 3 says that for an expression of type $\mathtt{T}$ in scope ℓ, we can find a corresponding scope ℓ' by applying the rules in Figure 17. Moreover, if the expression does not have a borrowed type, then it must satisfy the restriction $\texttt{locate}_{\sigma,\ell}(\texttt{T}) = \ell'$. The proof is by induction on the structure of an expression and it depends on Lemma 11 that says if a type $\texttt{C}$ is visible from $\texttt{S}$, then we can find the scope of an object with type $\texttt{C}$ by searching the scope hierarchy upwards starting from a scope of type $\texttt{S}$. The proof is straightforward and omitted.

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

We also need Lemma 12 that says if an expression's scope has the type $\texttt{S}$, then the expression must have a reusable type, a gate or scoped type inside $\texttt{S}$. The proof is simple and also omitted.

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

Lemma 3. *If* $\vdash \sigma$, $\sigma, \ell \vdash \texttt{e} : \texttt{T}$*, then* $\exists \ell'$ *such that* $\sigma, \ell \vdash_{scope} \texttt{e} : \ell'$*, and if* $\texttt{T}$ *is not borrowed type, then* $\texttt{locate}_{\sigma,\ell}(\texttt{T}) = \ell'$.

Proof. We prove by induction on the structure of $\texttt{e}$.

- $\texttt{e} = \texttt{null}$. $\sigma, \ell \vdash \texttt{null} : \texttt{C}$ for any $\texttt{C}$ and $\sigma, \ell \vdash_{scope} \texttt{null} : \ell'$ for any ℓ'.

- $\texttt{e} = \ell_0$. If $\sigma(\ell_0) = \texttt{C}^{\ell'_0}(\overline{\texttt{v}})$, then $\sigma, \ell \vdash_{scope} \ell_0 : \ell'_0$. If $\texttt{T} = \texttt{C}$, then by Rule (RT-Var), $\texttt{locate}_{\sigma,\ell}(\texttt{C}) = \ell'_0$.

- $\texttt{e} = \texttt{new C()}$. By Rule (RT-New), $\texttt{C}$ *viz* $\texttt{type}_\sigma(\ell)$. By Lemma 11, $\exists \ell'$ such that $\sigma, \ell \vdash_{scope} \texttt{e} : \ell'$. Let $\texttt{S} = \texttt{type}_\sigma(\ell)$. Since $\texttt{C}$ *viz* $\texttt{S}$, $\texttt{C} = \texttt{c}$ or $\texttt{C} = \texttt{S.s}$ or $\texttt{C} = \texttt{S}'\texttt{.c}$ and $\texttt{S}'$ is $\texttt{S}$ or encloses $\texttt{S}$. If $\texttt{C} = \texttt{c}$ or $\texttt{S.s}$, then $\ell' = \ell$ and $\texttt{locate}_{\sigma,\ell}(\texttt{C}) = \ell$. If $\texttt{C} = \texttt{S}'\texttt{.c}$, then by Rule (RT-New), $\texttt{S}' = \texttt{S}$. Thus, we have $\ell' = \ell$ and $\texttt{locate}_{\sigma,\ell}(\texttt{C}) = \ell$ as well.

- $\texttt{e} = (\texttt{C})\,\texttt{e}$. By induction hypothesis, $\exists \ell''$ such that $\sigma, \ell \vdash_{scope} \texttt{e} : \ell''$. By Rule (RT-Cast), we have $\texttt{C}$ *viz* $\texttt{type}_\sigma(\ell)$. By Lemma 11, $\exists \ell'$ such that $\texttt{locate}_{\sigma,\ell}(\texttt{C}) = \ell'$. Let $\sigma, \ell \vdash \texttt{e} : \texttt{K}$. By induction, $\texttt{locate}_{\sigma,\ell}(\texttt{K}) = \ell''$. Also by Rule (RT-Cast), $(\texttt{C} = \texttt{S.c} \wedge \texttt{K} = \texttt{S}'\texttt{.c}') \Rightarrow \texttt{S} = \texttt{S}'$ and $((\texttt{C} = \texttt{c} \wedge \texttt{K} = \texttt{S.c}') \vee (\texttt{K} = \texttt{c} \wedge \texttt{C} = \texttt{S.c}')) \Rightarrow \texttt{type}_\sigma(\ell) = \texttt{S}$. Thus, $\ell'' = \ell'$ and $\sigma, \ell \vdash_{scope} (\texttt{C})\,\texttt{e} : \ell'$.

- $\texttt{e} = \texttt{e}_0\texttt{.f}_\texttt{i}$ and $\texttt{e}_0 \neq \ell$.

  By induction hypothesis, $\sigma, \ell \vdash_{scope} \texttt{e}_0 : \ell_0$. Suppose $\sigma, \ell \vdash \texttt{e}_0\texttt{.f}_\texttt{i} : \texttt{T}$ and $\sigma, \ell \vdash \texttt{e}_0 : \texttt{T}_0$. If $\texttt{T}_0 = \texttt{local C}_0$, then $\texttt{C}_0$ is not a gate type and $\exists \texttt{C}$ such that $\texttt{T} = \texttt{local C}$.

  By class typing rules, $\texttt{C}$ *viz* $\texttt{C}_0$. Let $\texttt{type}_\sigma(\ell_0) = \texttt{S}$. By Lemma 12, $\texttt{C}_0 = \texttt{c}$ or $\texttt{C}_0 = \texttt{S.s}$ or $\texttt{C}_0 = \texttt{S.c}$. If $\texttt{C}_0$ is not a gate type, then we have $\texttt{C}$ *viz* $\texttt{S}$).

  If $\texttt{T}_0 = \texttt{C}_0$, then $\exists \texttt{C}$ such that $\texttt{T} = \texttt{C}$. By Rule (RT-Select), we have $\texttt{C}_0 \vdash \texttt{C}$ *viz* $\texttt{type}_\sigma(\ell)$. If $\texttt{C}_0 = \texttt{S.s}$, then $\texttt{type}_\sigma(\ell) = \texttt{S}$ and $\texttt{C}$ is not of the form $\texttt{c}$ or $\texttt{S.s.c}$ or $\texttt{S.s.s}'$. Thus, we have $\texttt{C}$ *viz* $\texttt{S}$.

  Hence, by Lemma 11, $\exists \ell'$ such that $\texttt{locate}_{\sigma,\ell_0}(\texttt{T}) = \ell'$.

  Moreover, if $\texttt{T} = \texttt{C}$, then $\exists \texttt{C}_0$ such that $\texttt{T}_0 = \texttt{C}_0$ and $\texttt{C}_0 \vdash \texttt{C}$ *viz* $\texttt{type}_\sigma(\ell)$. Together with $\texttt{C}$ *viz* $\texttt{C}_0$, if $\texttt{C} = \texttt{S}'\texttt{.c}$, then $\texttt{S}'$ equals to or encloses the static scopes of $\mathtt{C}_0$ and $\mathtt{type}_\sigma(\ell)$; if $\mathtt{C} = \mathtt{c}$ or $\mathtt{S}'$, then the static scopes of $\mathtt{C}_0$ and $\mathtt{type}_\sigma(\ell)$ are the same. By induction hypothesis, $\exists \ell_0$ such that $\sigma, \ell \vdash_{scope} \mathtt{e}_0 : \ell_0$ and $\mathtt{locate}_{\sigma,\ell}(\mathtt{C}_0) = \ell_0$. Since $\mathtt{locate}_{\sigma,\ell_0}(\mathtt{C}) = \ell'$, we have $\mathtt{locate}_{\sigma,\ell}(\mathtt{C}) = \ell'$.

- $\mathtt{e} = \ell.\mathtt{f_i}$. If $\sigma, \ell \vdash \mathtt{e} : \mathtt{C}$, then by class typing rule, $\mathtt{C}\ \mathit{viz}\ \mathtt{type}_\sigma(\ell)$. Thus, by Lemma 11, $\exists \ell'$ such that $\mathtt{locate}_{\sigma,\ell}(\mathtt{C}) = \ell'$. By Rule (S-Field2), $\sigma, \ell \vdash_{scope} \mathtt{e} : \ell'$.

- $\mathtt{e} = \mathtt{e}_0.\mathtt{f_i} := \mathtt{e}'$. By induction hypothesis, $\exists \ell', \ell''$ such that $\sigma, \ell \vdash_{scope} \mathtt{e}_0.\mathtt{f_i} : \ell'$ and $\sigma, \ell \vdash_{scope} \mathtt{e}' : \ell''$. Let $\sigma, \ell \vdash \mathtt{e}_0 : \mathtt{C}$, $\sigma, \ell \vdash \mathtt{e}' : \mathtt{C}'$ and $\mathit{fields}(\mathtt{C}) = (\overline{\mathtt{C}}\ \overline{\mathtt{f}})$.

  By induction hypothesis, $\mathtt{locate}_{\sigma,\ell}(\mathtt{C_i}) = \ell'$ and $\mathtt{locate}_{\sigma,\ell}(\mathtt{C}') = \ell''$. Thus, $\mathtt{locate}_{\sigma,\ell}(\mathtt{C}'') = \ell''$. Since $\mathtt{type}_\sigma(\ell) \vdash \mathtt{C}' <: \mathtt{C_i}$, if $\mathtt{C_i} = \mathtt{S}$, then $\mathtt{C}' = \mathtt{S}$, else if $\mathtt{C_i} = \mathtt{S.c}$, then $\mathtt{C}' = \mathtt{S.c}'$, else if $\mathtt{C_i} = \mathtt{c}$, then either $\mathtt{C}' = \mathtt{c}'$ or $\mathtt{C}' = \mathtt{S.c}'$, where $\mathtt{S} = \mathtt{type}_\sigma(\ell)$. Thus, $\ell' = \ell''$ by the definition of $\mathtt{locate}_{\sigma,\ell}(\mathtt{C}')$ and $\mathtt{locate}_{\sigma,\ell}(\mathtt{C_i})$.

- $\mathtt{e} = \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})$ and $\mathtt{e}_0 \neq \ell$. Let $\mathtt{type}_\sigma(\ell) = \mathtt{S}$. By induction hypothesis, $\exists \ell_0$ such that $\sigma, \ell \vdash_{scope} \mathtt{e}_0 : \ell_0$. Let $\mathtt{type}_\sigma(\ell_0) = \mathtt{S}_0$.

  Let $\sigma, \ell \vdash \mathtt{e}_0 : \mathtt{T}_0$ and $\mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{T}} \to \mathtt{C}$, where $\mathtt{T}_0 = \mathtt{local}\ \mathtt{C}_0$ or $\mathtt{C}_0$. If $\mathtt{T}_0 = \mathtt{local}\ \mathtt{C}_0$, then $\mathtt{C}_0$ is not a gate type. Then $\mathtt{C}_0$ either is $\mathtt{c}$ or $\mathtt{S}_0.\mathtt{c}$. By method typing rule, we have $\mathtt{C}\ \mathit{viz}\ \mathtt{C}_0$. Thus, $\mathtt{C}\ \mathit{viz}\ \mathtt{S}_0$. By Lemma 11, $\exists \ell'$ such that $\mathtt{locate}_{\sigma,\ell_0}(\mathtt{C}) = \ell'$.

  If $\mathtt{T} = \mathtt{C}_0$, then by Rule (RT-Invk), we have $\mathtt{C}_0 \vdash \mathtt{C}\ \mathit{viz}\ \mathtt{S}$. By method typing rule, we have $\mathtt{C}\ \mathit{viz}\ \mathtt{C}_0$. From $\sigma, \ell \vdash_{scope} \mathtt{e}_0 : \ell_0$ and induction hypothesis, we have $\mathtt{locate}_{\sigma,\ell}(\mathtt{C}_0) = \ell_0$. Thus, by Lemma 12, either $\mathtt{C}_0 = \mathtt{c}_0$, or $\mathtt{C}_0 = \mathtt{S}_0.\mathtt{c}_0$ where $\mathtt{S}_0$ equals to or encloses the static scope of $\mathtt{S}$, or $\mathtt{C}_0 = \mathtt{S}_0.\mathtt{s}_0$, where $\mathtt{S}_0 = \mathtt{S}$. From $\mathtt{C}_0 \vdash \mathtt{C}\ \mathit{viz}\ \mathtt{S}$, either $\mathtt{C} = \mathtt{c}$, where the static scope of $\mathtt{C}_0$ must be $\mathtt{S}$, or $\mathtt{C} = \mathtt{S}'.\mathtt{c}$, where $\mathtt{S}'$ equals to or encloses the static scope of $\mathtt{S}$, or $\mathtt{C} = \mathtt{S.s}$, where $\mathtt{C}_0 = \mathtt{S.c}_0$. If $\mathtt{C}_0 = \mathtt{S}_0.\mathtt{s}_0$, then $\mathtt{C}$ cannot have the forms of $\mathtt{c}$ or $\mathtt{S}_0\mathtt{s}_0.\mathtt{c}$ or $\mathtt{S}_0\mathtt{s}_0.\mathtt{s}$ since it would violate $\mathtt{C}_0 \vdash \mathtt{C}\ \mathit{viz}\ \mathtt{S}$. Thus, from $\mathit{visible}(\mathtt{C}, \mathtt{C}_0)$, we have $\mathtt{C}\ \mathit{viz}\ \mathtt{S}_0$. By Lemma 11, $\exists \ell'$ such that $\mathtt{locate}_{\sigma,\ell_0}(\mathtt{C}) = \ell'$.

  Recall that $\mathtt{locate}_{\sigma,\ell}(\mathtt{C}_0) = \ell_0$. From $\mathtt{C}_0' \vdash \mathtt{C}\ \mathit{viz}\ \mathtt{S}$, if $\mathtt{C} = \mathtt{c}$ or $\mathtt{S.s}$, then $\mathtt{S} = \mathtt{S}_0$ and if $\mathtt{C} = \mathtt{S}'.\mathtt{c}$, then $\mathtt{S}'$ equals to or encloses the static scope of $\mathtt{S}_0$, which equals to or encloses the static scope of $\mathtt{S}$. Thus, we can conclude that $\mathtt{locate}_{\sigma,\ell}(\mathtt{C}) = \ell'$.

- $\mathtt{e} = \ell.\mathtt{m}(\overline{\mathtt{e}})$. If $\mathtt{type}_\sigma(\ell) = \mathtt{C}_0$ and $\sigma, \ell \vdash \mathtt{e} : \mathtt{C}$, then from Rule (RT-Invk2), $\mathtt{C}\ \mathit{viz}\ \mathtt{C}_0$. By Lemma 11, $\exists \ell'$ such that $\mathtt{locate}_{\sigma,\ell}(\mathtt{C}) = \ell'$. By Rule (S-Invk2), $\sigma, \ell \vdash \mathtt{e} : \ell'$.

□

7.4. *An expression's scope is on the scope stack*

Earlier, we showed that well-typed expression has a corresponding scope. In Lemma 4, we show that if a thread's scope stack γ and its frame $\ell, \mathtt{e}$ are well-formed, then the scope of $\mathtt{e}$ must be in γ. The proof is based on Lemma 13 that says if a scope stack γ is well-formed, then for each scope in γ, its outer scope is also in γ.

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 $\sigma(\ell) = \mathtt{C'}^{\ell'_0}(\overline{\mathtt{v}})$. 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. □

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

Proof. We prove by induction on the structure of $\mathtt{e}$.

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

□

7.5. Subject reduction preserves the scope of expression

We have shown that a well-typed expression in a thread t has a corresponding scope on the scope stack of t. Now we show that reduction of the expression does not change the scope. The proof is by straightforward induction on the structure of an expression.

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

Proof. We prove by induction on the structure of $\mathtt{e}$.

- $\mathtt{e} = \mathtt{new}\ \mathtt{C}()$.

  By Rule (S-New), $\sigma, \ell \vdash_{scope} \mathtt{e} : \ell$. By Rule (R-New), $\sigma, \ell, \mathtt{e} \rightarrow \sigma', \ell, \mathtt{e}'$ and $\mathtt{e}' = \ell_0$, where $\sigma'(\ell_0) = \mathtt{C}^{\ell}(\overline{\mathtt{null}})$. Thus, $\sigma', \ell \vdash_{scope} \mathtt{e}' : \ell$ by Rule (S-Var).

- $\mathtt{e} = (\mathtt{C})\ \mathtt{v}$.

  By Rule (S-Cast), $\sigma, \ell \vdash_{scope} \mathtt{e} : \ell_{\mathtt{a}}$ and $\sigma, \ell \vdash_{scope} \mathtt{v} : \ell_{\mathtt{a}}$.

- $\mathtt{e} = \ell'.\mathtt{f_i}$ and $\ell' \neq \ell$.

  By Rule (S-Field), $\sigma, \ell \vdash_{scope} \mathtt{e} : \ell_{\mathtt{a}}$, where $\sigma, \ell \vdash_{scope} \ell' : \ell_0$, $\mathtt{locate}_{\sigma,\ell_0}(\mathtt{T}) = \ell_{\mathtt{a}}$, $\sigma, \ell \vdash \mathtt{e} : \mathtt{T}$.

  By Rule (R-Field), $\sigma, \ell, \mathtt{e} \rightarrow \sigma, \ell, \mathtt{e}'$ and $\mathtt{e}' = \mathtt{v_i}$, where $\sigma(\ell') = \mathtt{C_0}^{\ell_0}(\overline{\mathtt{v}})$. The case is trivial if $\mathtt{v_i} = \mathtt{null}$. Let $\mathtt{v_i} = \ell_{\mathtt{i}}$.

  Suppose $\sigma, \ell \vdash \ell' : \mathtt{T_0}$. If $\mathtt{T} = \mathtt{local}\ \mathtt{C}$, then $\mathtt{T_0} = \mathtt{local}\ \mathtt{C_0}$. In this case, $\mathtt{C_0}$ cannot be a gate type and hence, $\mathit{inscope}_{\sigma}(\ell') = \ell_0$. From $\vdash \sigma$, we have $\sigma \vdash \ell'$, which implies $\mathtt{locate}_{\sigma,\ell_0}(\mathtt{C}) = \ell'_{\mathtt{i}}$, where $\mathtt{scope}_{\sigma}(\ell_{\mathtt{i}}) = \ell'_{\mathtt{i}}$. In this case, $\ell'_{\mathtt{i}} = \mathtt{locate}_{\sigma,\ell_0}(\mathtt{T}) = \ell_{\mathtt{a}}$. Since $\sigma, \ell \vdash_{scope} \mathtt{e}' : \ell'_{\mathtt{i}}$ by Rule (S-Var), we have $\sigma, \ell \vdash_{scope} \mathtt{e}' : \ell_{\mathtt{a}}$.

  Consider the case $\mathtt{T} = \mathtt{C}$. By Lemma 3, $\mathtt{locate}_{\sigma,\ell}(\mathtt{C}) = \ell_{\mathtt{a}}$. By Lemma 1, $\sigma, \ell \vdash \mathtt{e}' : \mathtt{C}'$ such that $\mathtt{type}_{\sigma}(\ell) \vdash \mathtt{C}' \preceq \mathtt{C}$. Again, by Lemma 3, if $\sigma, \ell \vdash_{scope} \mathtt{e}' : \ell'_{\mathtt{a}}$, then $\mathtt{locate}_{\sigma,\ell}(\mathtt{C}') = \ell'_{\mathtt{a}}$. From $\mathtt{type}_{\sigma}(\ell) \vdash \mathtt{C}' \preceq \mathtt{C}$, if $\mathtt{C} = \mathtt{S}$, then $\mathtt{C}' = \mathtt{S}$; if $\mathtt{C} = \mathtt{S.c}$, then $\mathtt{C}' = \mathtt{S.c}'$; if $\mathtt{C} = \mathtt{c}$, then either $\mathtt{C}' = \mathtt{c}'$ or $\mathtt{C}' = \mathtt{S.c}'$, where $\mathtt{type}_{\sigma}(\ell) = \mathtt{S}$. It is clear that $\ell_{\mathtt{a}} = \ell'_{\mathtt{a}}$ for all cases.

- $\mathtt{e} = \ell.\mathtt{f_i}$.

  In this case, if $\sigma(\ell) = \mathtt{S}^{\ell'}(\overline{\mathtt{v}})$, then by Rule (R-Field), $\mathtt{e}' = \mathtt{v_i}$. Suppose $\mathtt{v_i} = \ell_{\mathtt{i}}$. Suppose $\sigma, \ell \vdash \mathtt{e} : \mathtt{C}$. By Lemma 3, $\sigma, \ell \vdash_{scope} \mathtt{e} : \ell_{\mathtt{a}}$ and $\mathtt{locate}_{\sigma,\ell}(\mathtt{C}) = \ell_{\mathtt{a}}$. From $\vdash \sigma$, $\mathtt{locate}_{\sigma,\ell}(\mathtt{C}) = \ell'_{\mathtt{i}}$, where $\mathtt{scope}_{\sigma}(\ell_{\mathtt{i}}) = \ell'_{\mathtt{i}}$. Thus, $\ell'_{\mathtt{i}} = \ell_{\mathtt{a}}$ and $\sigma, \ell \vdash_{scope} \mathtt{e}' : \ell_{\mathtt{a}}$.

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

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

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

  Suppose $\mathtt{C_0}$ is not a gate type. Then, $\ell' = \ell'_0$. By Lemma 9, $\exists \mathtt{C}'$ such that $\sigma, \ell' \vdash \mathtt{e}' : \mathtt{C}'$, where $\mathtt{type}_{\sigma}(\ell') \vdash \mathtt{C}' \preceq \mathtt{C}$. Thus, by Lemma 3, $\exists \ell'_{\mathtt{a}}$ such that $\sigma, \ell' \vdash_{scope} \mathtt{e}' : \ell'_{\mathtt{a}}$ and $\mathtt{locate}_{\sigma,\ell'}(\mathtt{C}') = \ell'_{\mathtt{a}}$. From $\mathtt{type}_{\sigma}(\ell') \vdash \mathtt{C}' \preceq \mathtt{C}$, we have $\mathtt{locate}_{\sigma,\ell'}(\mathtt{C}) = \ell'_{\mathtt{a}}$. Since $\ell' = \ell'_0$, $\mathtt{locate}_{\sigma,\ell'_0}(\mathtt{T}) = \mathtt{locate}_{\sigma,\ell'_0}(\mathtt{C}) = \ell_{\mathtt{a}}$. We have $\ell'_{\mathtt{a}} = \ell_{\mathtt{a}}$ and $\sigma, \ell' \vdash_{scope} \mathtt{e}' : \ell_{\mathtt{a}}$.

  Suppose $\mathtt{C_0}$ is a gate type. Then $\mathtt{T} = \mathtt{C}$ and $\mathtt{T_0} = \mathtt{C_0}$ since a gate type cannot be borrowed type. Then $\mathit{inscope}_{\sigma}(\ell_0) = \ell' = \ell_0$. By Rule (RT-Var), $\mathtt{locate}_{\sigma,\ell}(\mathtt{C_0}) = \ell'_0$. Let $\mathtt{type}_{\sigma}(\ell) = \mathtt{S}$. Then $\mathtt{C_0}$ must have the form of

$\mathtt{S.s_0}$. By Lemma 1, $\sigma, \ell' \vdash \mathtt{e'} : \mathtt{C'}$ where $\mathtt{S} \vdash \mathtt{C'} \preceq \mathtt{C}$ and $\mathtt{C_0} \vdash \mathtt{C}\,\mathit{viz}\,\mathtt{S}$. Therefore, $\mathtt{C}$ cannot have the form of $\mathtt{c}$ since this would imply $\mathit{scopeof}(\mathtt{C_0}) = \mathtt{S}$. $\mathtt{C}$ cannot be a gate type either. Suppose $\mathtt{C}$ is a gate type. By Lemma 3, $\mathtt{locate}_{\sigma,\ell}(\mathtt{C}) = \ell_{\mathtt{a}}$, which means that $\mathtt{C}$ must have the form of $\mathtt{S.s}$. Also by Lemma 3, $\mathtt{locate}_{\sigma,\ell'}(\mathtt{C'}) = \ell'_{\mathtt{a}}$. In this case, $\mathtt{C'}$ must have the form of $\mathtt{S.s_0.s'}$. But this would violate $\mathtt{C'} <: \mathtt{C}$. Thus, the only possible case left is that $\mathtt{C} = \mathtt{S'.c}$ and $\mathtt{C} = \mathtt{S'.c'}$, where $\mathtt{S'} = \mathtt{S}$ or $\mathtt{S'}$ encloses $\mathtt{S}$. Recall that $\ell' = \ell_0$ in this case. From $\mathtt{locate}_{\sigma,\ell'}(\mathtt{C'}) = \ell'_{\mathtt{a}}$, either $\ell' = \ell'_{\mathtt{a}}$ or $\ell'_0 \preceq_\sigma \ell'_{\mathtt{a}}$. However, ℓ' cannot be equal to $\ell'_{\mathtt{a}}$ since it would mean that $\mathtt{S'} = \mathtt{S.s_0}$. Thus, $\ell'_0 \preceq_\sigma \ell'_{\mathtt{a}}$ and this implies $\mathtt{locate}_{\sigma,\ell'_0}(\mathtt{C'}) = \ell'_{\mathtt{a}}$. Since $\mathtt{locate}_{\sigma,\ell'_0}(\mathtt{C}) = \ell_{\mathtt{a}}$, we have $\ell_{\mathtt{a}} = \ell'_{\mathtt{a}}$.

- $\mathtt{e} = \ell.\mathtt{m}(\overline{\mathtt{v}})$

  By Rule (R-Invk), $\sigma, \ell, \mathtt{e} \rightarrow \sigma, \ell, \mathtt{e'}$ and since ℓ is a gate object, it cannot have scope-local type. Thus, by Rule (RT-Invk), $\exists \mathtt{C}$ such that $\sigma, \ell \vdash \mathtt{e} : \mathtt{C}$. By Lemma 1, $\exists \mathtt{C'}$ such that $\sigma, \ell \vdash \mathtt{e'} : \mathtt{C'}$ and $\mathtt{type}_\sigma(\ell) \vdash \mathtt{C'} \preceq \mathtt{C}$. By Lemma 3, $\sigma, \ell \vdash_{\mathit{scope}} \mathtt{e} : \ell_{\mathtt{a}}$, $\mathtt{locate}_{\sigma,\ell}(\mathtt{C}) = \ell_{\mathtt{a}}$, $\sigma, \ell \vdash_{\mathit{scope}} \mathtt{e'} : \ell'_{\mathtt{a}}$, and $\mathtt{locate}_{\sigma,\ell}(\mathtt{C'}) = \ell'_{\mathtt{a}}$. From $\mathtt{type}_\sigma(\ell) \vdash \mathtt{C'} \preceq \mathtt{C}$, if $\mathtt{C} = \mathtt{c}$, then either $\mathtt{C'} = \mathtt{c'}$, or $\mathtt{C'} = \mathtt{S.c'}$ and $\mathtt{type}_\sigma(\ell) = \mathtt{S}$; if $\mathtt{C} = \mathtt{S.s}$, then $\mathtt{C'} = \mathtt{S.s}$. In both cases, $\ell_{\mathtt{a}} = \ell_{\mathtt{a}} = \ell$. If $\mathtt{C} = \mathtt{S'.c}$, then $\mathtt{C'} = \mathtt{S'.c'}$ and $\mathtt{S'} = \mathtt{S}$ or $\mathtt{S'}$ encloses $\mathtt{S}$. Since $\mathtt{C}$ and $\mathtt{C'}$ have the same static scope, we have $\ell_{\mathtt{a}} = \ell'_{\mathtt{a}}$ as well.

- $\mathtt{e} = \ell.\mathtt{f_i} := \mathtt{v}$.

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

□

7.6. Subject reduction of programs preserves stack and store invariants

Finally we prove the subject reduction lemma. The proof is to show that the reduction of a program by Rule (G-Step), (G-Enter), and (G-Ret), preserves stack and store invariants. In particular, reduction by Rule (G-Ret) may result in setting references to objects in a scope to dummy values and setting the fields of the scope to null so that these objects are no longer accessible to the program. After clearing a scope, the stack invariant holds because each expression in each thread has a corresponding scope which must be in its scope stack of the thread. To clear a scope s, it is required that s not be on the scope stack of any thread. Also, the store invariant holds because each object in the store can only have field references to objects in the same or outer scopes. This means only objects in the inner scope of s can have field references to objects in s. But the stack invariant ensures that inner scopes of s cannot be in any of the scope stack in the program because if they do, s must be on the scope stack as well. So reference count on s and its inner scopes are zero. By stack invariant, the store may not contain these objects. Therefore, no object in the store can have fields referencing objects in s.

The previous lemmas about scopes of expressions and preservation of scopes are used in proving that reduction steps by Rule (G-Step) and (G-Enter) preserves stack invariant. But first, we prove a helper lemma that says if an expression context $E[\mathtt{e}]$ is well-typed, and we replace $\mathtt{e}$ with another expression $\mathtt{e}'$ whose type is a safe subtype of that of $\mathtt{e}$, then $E[\mathtt{e}']$ is well-typed and its type is a safe subtype of that of $E[\mathtt{e}']$.

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

Proof. We prove by a case analysis on the structure of $E[\mathtt{e}]$.

1. $E[\mathtt{e}]$ is $E'[\mathtt{e}].\mathtt{f}$.
   By Rule (RT-Field), if $E[\mathtt{e}]$ has type $\mathtt{C_i}$ and $E'[\mathtt{e}]$ has type $\mathtt{C}$, then $\mathtt{C} \vdash \mathtt{C_i}$ *viz* $\mathtt{type}_\sigma(\ell)$. By induction hypothesis, $E'[\mathtt{e}']$ has a type $\mathtt{C}'$ and $\mathtt{type}_\sigma(\ell) \vdash \mathtt{C}' \preceq \mathtt{C}$. If $\mathtt{C}$ is a reusable type and $\mathtt{C}'$ is scoped, then $\mathtt{C}'$ and $\mathtt{type}_\sigma(\ell)$ are in the same scope. It follows that $\mathtt{C}' \vdash \mathtt{C_i}$ *viz* $\mathtt{type}_\sigma(\ell)$ and $E[\mathtt{e}']$ has type $\mathtt{C_i}$.
   By Rule (RT-Field2), if $E[\mathtt{e}]$ has type $\mathtt{T}_0$ and $E'[\mathtt{e}]$ has type $\mathtt{T}$, then the type of $E'[\mathtt{e}']$ is $\mathtt{T}'$, where $\mathtt{type}_\sigma(\ell) \vdash \mathtt{T}' \preceq \mathtt{T}$. It is clear that $E[\mathtt{e}']$ also has type $\mathtt{T}_0$.
2. $E[\mathtt{e}]$ is $E'[\mathtt{e}].\mathtt{f} := \mathtt{e}_0$. The proof is similar to previous case.
3. $E[\mathtt{e}]$ is $\mathtt{v.f} := E'[\mathtt{e}]$.
   By Rule (RT-Upd), if $E[\mathtt{e}]$ has type $\mathtt{C}'$, then $E'[\mathtt{e}]$ has type $\mathtt{C}'$ and $\mathtt{type}_\sigma(\ell) \vdash \mathtt{C}' \preceq \mathtt{C_i}$, where $\mathtt{C_i}$ is the type of the field $\mathtt{f}$. By induction hypothesis, $E'[\mathtt{e}']$ has a type $\mathtt{C}''$ and $\mathtt{type}_\sigma(\ell) \vdash \mathtt{C}'' \preceq \mathtt{C}'$. It is clear that $\mathtt{type}_\sigma(\ell) \vdash \mathtt{C}'' \preceq \mathtt{C_i}$. Therefore, $E[\mathtt{e}']$ has the type $\mathtt{C}''$.
4. $E[\mathtt{e}]$ is $E'[\mathtt{e}].\mathtt{m}(\overline{\mathtt{e}})$. The proof is similar to case 1.
5. $E[\mathtt{e}]$ is $\mathtt{v.m}(\overline{\mathtt{v}}, E'[\mathtt{e}], \overline{\mathtt{e}})$. The proof is similar to case 3.
6. $E[\mathtt{e}]$ is $(\mathtt{C})E'[\mathtt{e}]$.
   By Rule (RT-Cast), if $\mathtt{e}$ has type $\mathtt{K}$, then either both $\mathtt{K}$ and $\mathtt{C}$ are scoped types in the same scope, or both of them are reusable type, or one of them is a reusable type while the other is a scoped type in the scope of $\mathtt{type}_\sigma(\ell)$. By induction hypothesis, $\mathtt{e}'$ has a type $\mathtt{K}'$ such that $\mathtt{type}_\sigma(\ell) \vdash \mathtt{K}' \preceq \mathtt{K}$. This means that if $\mathtt{K}$ is scoped type, then $\mathtt{K}'$ must be a scoped type in the same scope. If $\mathtt{K}$ is a reusable type, then $\mathtt{K}'$ must be either a reusable type or a scoped type in the same of $\mathtt{type}_\sigma(\ell)$. In all cases, $E[\mathtt{e}']$ has the type $\mathtt{C}$ by Rule (RT-Cast).

□

Lemma 6. *If* $\sigma \vdash P$ *and* $\sigma, P \rightsquigarrow \sigma', P'$, *then* $\sigma' \vdash P'$.

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

G-STEP Let $P = P'' \mid \kappa \bullet \ell, E[\mathtt{e}]$, $\mathtt{e}$ is not a method call, and $\sigma, \ell, \mathtt{e} \rightarrow \sigma', \ell, \mathtt{e}'$. By Rule (G-STEP), $P' = P'' \mid \kappa'$, where $\kappa' = \kappa \bullet \ell, E[\mathtt{e}']$. By assumption, $\vdash \sigma$, $\sigma \vdash \kappa \bullet \ell, E[\mathtt{e}]$, We need to show $\sigma' \vdash P'$, which is to show $\sigma' \vdash P''$, $\vdash \sigma'$, and $\sigma' \vdash \kappa'$,

By Rule (G-Step), a reduction step from P to P' involves the expression reduction rule (R-New), (R-Upd), (R-Field), or (R-Cast). Only the first two rules can change the store σ to σ'. For the first case, σ' is σ with an additional object while for the latter case, σ' is σ with an field of one object in σ changed. For both cases, the typing of expressions in the call stacks of P'' does not change since the types of references do not change. Also since the allocation scope of each object remains the same in σ', the stack invariant and store invariant hold for σ'. Thus, $\sigma' \vdash P''$.

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

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

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

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

Let $\gamma = \mathtt{ScopeStack}(\kappa \bullet \ell, E[\mathtt{e}] \bullet \ell', \mathtt{v})$ and $\gamma' = \mathtt{ScopeStack}(\kappa')$. Then $\gamma = \gamma' \bullet \ell'$. It is clear that $\sigma \vdash \gamma'$. Suppose $\mathit{refcount}(\ell',\ P) = 0$. 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 a valid reference in σ'. Therefore, we have $\sigma' \vdash \gamma'$. Also, from $\sigma \vdash \kappa \bullet \ell, E[\mathtt{e}] \bullet \ell, E[\mathtt{v}]$, we have $\sigma, \gamma' \vdash E[\mathtt{e}]$. Each object in $E[\mathtt{e}]$ is allocated in one of the scopes in γ'. Since $\ell' \notin \gamma'$, we have $\sigma', \gamma' \vdash E[\mathtt{e}]$. Also, $\sigma', \ell \vdash E[\mathtt{e}] : \mathtt{T}_0$ if $\sigma, \ell \vdash E[\mathtt{e}] : \mathtt{T}_0$. By induction on each frame in κ, we can show $\sigma' \vdash \kappa \bullet E[\mathtt{e}]$ Similarly, for each call stack κ'' in P, we have $\sigma' \vdash \kappa''$.

To show $\vdash \sigma'$, we need to prove that for each valid reference $\ell_{\mathtt{r}}$ in σ', the fields of $\ell_{\mathtt{r}}$ are either null or valid references in σ'. From $\sigma \vdash P$, for any $\ell_{\mathtt{a}}$, if $\mathit{refcount}(\ell_{\mathtt{a}},\ P) = 0$, then no object in σ is allocated in $\ell_{\mathtt{a}}$. Since σ' is σ without references to objects in ℓ' and $\mathit{refcount}(\ell',\ P') = 0$, we have that for any $\ell_{\mathtt{a}}$, $\mathit{refcount}(\ell_{\mathtt{a}},\ P') = 0$ implies no object in σ' is allocated in $\ell_{\mathtt{a}}$. Conversely, if an object in σ' is allocated in $\ell_{\mathtt{a}}$, then $\mathit{refcount}(\ell_{\mathtt{a}},\, P') \neq 0$.

Suppose $\ell_{\mathtt{r}}$ is allocated in $\ell'_{\mathtt{r}}$ and it contains a field $\mathtt{v_i} = \ell_{\mathtt{i}}$. We will show that if $\sigma(\ell_{\mathtt{i}})$ is allocated in ℓ', then it will lead to contradiction to the conditions of Rule (G-Ret). From $\vdash \sigma$, if $\mathit{inscope}_\sigma(\ell_{\mathtt{r}}) = \ell''_{\mathtt{r}}$, then $\mathtt{locate}_{\sigma,\ell''_{\mathtt{r}}}(\mathtt{C}') = \ell'$, which means that $\ell''_{\mathtt{r}} \preceq_\sigma \ell'$. Also, if $\ell_{\mathtt{r}}$ is a valid reference in σ', then $\mathit{refcount}(\ell'_{\mathtt{r}},\, P') \neq 0$. Thus, $\exists\kappa''$ such that $P' = P'' \mid \kappa''$, $\gamma'' = \mathtt{ScopeStack}(\kappa'')$ and $\ell'_{\mathtt{r}} \in \gamma''$.

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

Suppose that $\ell_{\mathtt{r}}$ is a gate object. Then $\ell''_{\mathtt{r}} = \ell_{\mathtt{r}}$. If $\ell_{\mathtt{i}}$ is not allocated in $\ell_{\mathtt{r}}$, then $\ell'_{\mathtt{r}} \preceq_\sigma \ell'$ and we can reach the same contradiction to $\mathit{refcount}(\ell',\, P') = 0$.

If $\ell_{\mathtt{i}}$ is allocated in $\ell_{\mathtt{r}}$, then $\ell' = \ell_{\mathtt{r}}$. However, by Rule (G-Ret), the fields of $\sigma'(\ell')$ are all set to null so that $\ell_{\mathtt{i}}$ 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 $\sigma' \vdash \kappa \bullet \ell, E[\mathtt{e}]$, which implies $\mathtt{C}' <: \mathtt{C}$, where $\sigma', \ell \vdash \mathtt{e} : \mathtt{T}$, $\mathtt{T} = \mathtt{local\ C}$ or $\mathtt{T} = \mathtt{C}$, and $\sigma, \ell' \vdash \mathtt{v} : \mathtt{C}'$. If $\sigma', \ell \vdash_{\mathit{scope}} \mathtt{e} : \ell_{\mathtt{a}}$, then from $\sigma' \vdash \kappa \bullet \ell, E[\mathtt{e}] \bullet \ell', \mathtt{v}$, $\sigma, \ell' \vdash_{\mathit{scope}} \mathtt{v} : \ell_{\mathtt{a}}$, which means that $\mathtt{v}$ is either null or allocated in $\ell_{\mathtt{a}}$. Suppose $\mathtt{v}$ is not null. If $\mathtt{T} = \mathtt{local\ C}$, then we have $\mathtt{type}_\sigma(\ell) \vdash \mathtt{C}' \preceq \mathtt{T}$. Otherwise $\mathtt{T} = \mathtt{C}$. Then $\mathtt{locate}_{\sigma',\ell}(\mathtt{C}) = \ell_{\mathtt{a}}$. Thus, if $\mathtt{C} = \mathtt{c}$ and $\mathtt{C}' = \mathtt{S.c}$, then $\ell_{\mathtt{a}} = \ell$ and $\mathtt{S} = \mathtt{type}_{\sigma'}(\ell)$. Therefore, $\mathtt{type}_{\sigma'}(\ell) \vdash \mathtt{C}' \preceq \mathtt{T}$. Thus, by Lemma 14, $\exists \mathtt{C}'_0$ such that $\sigma', \ell \vdash E[\mathtt{v}] : \mathtt{C}'_0$ and $\mathtt{C}'_0 <: \mathtt{C}_0$, where $\sigma', \ell \vdash E[\mathtt{e}] : \mathtt{C}_0$.

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

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

□