# Lexically Scoped Regions for Initialization of Pluggable Object Types

A supplement to a course at the ESF Cost Winter School on Verification of OO Programs in Estonia

Christian Haack

January 25, 2009

**Abstract**

Pluggable type systems for object-oriented languages often face the problem that object types are only valid after objects have been initialized. Examples include non-nullness type systems (all object fields are initially `null`) and type systems for object immutability (even immutable objects mutate during initialization). Unfortunately, the technical problems associated with object initialization are often the most complicated aspects of such type systems. The handling of object initialization in some recent systems [FX07, HP09] is based on ideas that are inspired or closely related to lexically scoped regions as used in region-based memory management [TT97]. The aim of this note is to work out the relation between these type systems and lexically scoped regions.

We formalize the type systems discussed in class for a small model language with records and nominal types. We refer to declarations of named record types as classes, and to records as objects. We do not model dynamic dispatch and instead assume that methods are defined separately from classes, essentially like static methods in Java.

Throughout this note, we use the following identifier domains:

| | | | |
|---|---|---|---|
| $C, D \in \mathsf{ClassId}$ | class identifiers | $f, g \in \mathsf{FieldId}$ | field identifiers |
| $m \in \mathsf{MethodId}$ | method identifiers | $x, y, z \in \mathsf{Var}$ | local variables |

We use similar notational conventions as Featherweight Java [IPW01]. For instance, we indicate sequences by an overbar, e.g., $\bar{x}$ represents a sequence of local variables.

## 1 Safe Deallocation with Lexically Scoped Regions

Lexically scoped regions have been used for safe memory deallocation, for instance, in the experimental language Cyclone [GMJ+02] — a memory-safe dialect of C. They have been invented by Tofte and Talpin [TT97], who developed region-based management for ML and emphasized region *inference* in order to *insert safe deallocation instructions at compile time* ("compile-time garbage collection").

The type system presented in this section is an adaptation of a part of Cyclone's type system [GMJ+02][1]. It is designed to guarantee *memory safety*.

> **Safety Property:** Well-typed programs do not attempt to access deallocated memory locations.

[1] Cyclone has more features than lexically scoped regions.

### 1.1 Regions

Regions are named segments of the heap. Regions names are unique and distinct regions are disjoint. So every object is member of exactly one region.

$$n \in \mathsf{RegionName} \quad \text{region names} \qquad \alpha \in \mathsf{RegionVar} \quad \text{region variables}$$
$$p, q, r \ ::= \ \alpha \ \mid \ \texttt{Region(}n\texttt{)} \quad \text{region expressions}$$

The actual names of regions on the heap are of the form `Region(`$n$`)`. *Region variables* refer to these actual names indirectly and are used as class parameters (analogously to type parameters of Java's generic types) and as auxiliary method parameters (analogously to type parameters of Java's generic methods).

### 1.2 Classes and Types

Our model is based on records. We refer to declarations of named record types as classes, and to records as objects.

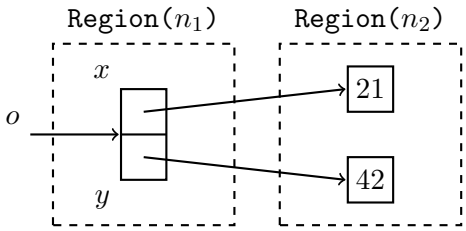
$$\begin{array}{ll} \mathit{class} \ ::= \ \texttt{class}\ C\texttt{<}\bar{\alpha}\texttt{>}\ \texttt{\{}\ \bar{T}\ \bar{f}\ \texttt{\}} & \text{class declarations} \\ T, U, V \in \mathsf{Ty} \ ::= \ r\ C\texttt{<}\bar{r}\texttt{>} \ \mid \ \texttt{int} \ \mid \ \texttt{void} & \text{types} \end{array}$$

*Class declarations* have region parameters $\bar{\alpha}$. These can occur in the field types $\bar{T}$. *Object types* are of the form $q\ C\texttt{<}\bar{r}\texttt{>}$. If an object has type $q\ C\texttt{<}\bar{r}\texttt{>}$, then the region expression $q$ denotes the region that the object belongs to, and region expressions $\bar{r}$ instantiate $C$'s class parameters. Class parameters are ignored at runtime, but the region expression that prefixes the class identifier is not, as it is used for region deallocation. A *class table* is a set of class declarations for distinct class identifiers. Class declarations may be recursive and mutually recursive.

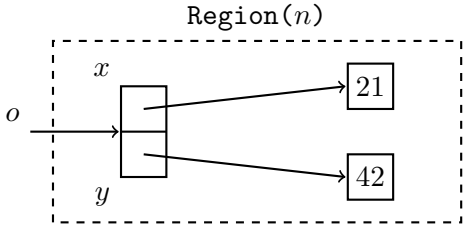
**Example** Here is a simple class table that defines a class for boxed integers, and a region-parametrized class for points whose coordinates are boxed integers.

```
class Integer { int val; }    class Point<r> { r Integer x; r Integer x; }
```

If object $o$ has type `Region(`$n_1$`) Point<Region(`$n_2$`)>`, then $o$ lives in `Region(`$n_1$`)` and the integer objects that represent $o$'s coordinates live in `Region(`$n_2$`)`:




If object $o$ has type `Region(`$n$`) Point<Region(`$n$`)>`, then $o$ and its two coordinate objects all live in the same `Region(`$n$`)`:



### 1.3 Methods

Methods are region-polymorphic, i.e., they have auxiliary region parameters $\bar{\alpha}$ that can occur in the argument and return types, and the method body:

$$\textit{method} \ ::= \ \texttt{<}\bar{\alpha}\texttt{>}\ U\ m\texttt{(}\bar{T}\ \bar{x}\texttt{)\{}e\texttt{\}} \quad \text{method declarations (scope of } \bar{\alpha} \text{ is } U, \bar{T}, e, \text{ and scope of } \bar{x} \text{ is } e\text{)}$$

The region parameters are specification-only: they have no effect on the runtime behaviour of programs. Formally, region parameters are very similar to type parameters of generic methods in Java. The meta-variable $e$ that represents the method body ranges over expressions, which will be defined below. A *method table* is a set of method declarations for distinct method identifiers. Method declarations may be recursive and mutually recursive.

**Example** Here is a method that makes a copy of a point `x` located in region `r`, places this copy in region `q`, and returns the copy. At call sites the two formal region parameters `r` and `q` may either be instantiated by the same actual region, or they may be instantiated by distinct actual regions.

```
<r,q> q Point<q> copy(r Point<r> arg) {
    q Point<q> result = new q Point<q>;
    // result.x = arg.x;     THIS WOULD BE A TYPE ERROR
    result.x = new q Integer;
    result.y = new q Integer;
    result.x.val = arg.x.val;
    result.y.val = arg.y.val;
    return result;
}
```

The commented line would be a type error, because it assigns an object from `Region(r)` to a field that is supposed to refer to objects in `Region(q)`.

It is noteworthy that just by looking at the method signature, we can tell that the method *does not return a shallow copy of its argument.* This is so, because the method is type-correct *for all instantiations of the region parameters* `r` *and* `q`. If `copy()` did return a shallow copy of `arg`, then the returned `Point` would share its coordinate objects with `arg`. This means that these coordinate objects live both in region `r` (because this is where `arg`'s coordinate objects live by its type `r Point<r>`) and in region `q` (because this is where `result`'s coordinate objects live by its type `q Point<q>`). Because the region type system maintains the invariant that distinct regions are disjoint, this would only be possible if `r` and `q` were instantiated by the same region, which contradicts the fact that `copy()` is also type-correct when `r` and `q` are instantiated by distinct regions.

### 1.4 Expressions

Figure 1 displays the expression language. In local variable declarations $T\ x; e$ and region name generation `newName` $n$ `in` $e$ `end`, we treat $x$ and $n$ as binders identifying expressions up to renaming of bound variables and bound names. In method calls `<`$\bar{r}$`>`$m$`(`$\bar{v}$`)`, the actual region parameters $\bar{r}$ instantiate $m$'s formal region parameters. In practice, these actual region parameters would be inferred (similarly to actual type parameters for generic methods in Java). Here are explanations of the two most interesting expressions:

- `new` $q$ $C$`<`$\bar{r}$`>`: Creates a new object of class $C$`<`$\bar{r}$`>` and places it in region $q$.
- `newName` $n$ `in` $e$ `end`: Generates a fresh region name for an initially empty `Region(`$n$`)`, then executes $e$, and finally deallocates all objects in `Region(`$n$`)`.

The region type system enforces the following crucial property: *Objects in* `Region(`$n$`)` *are not accessible outside the lexical scope of* $n$. This is why the region deallocation at the end of $n$'s lexical scope is safe.

| | | |
|---|---|---|
| $e \in \mathsf{Exp}$ ::= | | |
| | `null` | |
| | $i$ | integer |
| | $x$ | local variable |
| | $T\,x; e$ | local variable declaration (scope of $x$ is $e$) |
| | $x = e$ | variable assignment |
| | $e.f = e$ | field assignment |
| | $e.f$ | field selection |
| | $e; e$ | sequential composition |
| | `<`$\bar{r}$`>`$m$`(`$\bar{v}$`)` | method call |
| | `new` $r$ $C$`<`$\bar{r}$`>` | object creation |
| | `newName` $n$ `in` $e$ `end` | region generation (scope of $n$ is $e$) |

Derived form: $e; \triangleq e; \texttt{null}$

Figure 1: Expressions

**Example** The following code snippet generates a fresh region name `n`, allocates a point object in `Region(`$n$`)`, does something with the point, and finally deallocates the point at the end of `n`'s lexical scope:

```
newName n in
    Region(n) Point<Region(n)> p;
    p = new Region(n) Point<Region(n)>;
    p.x = new Region(n) Integer;
    p.y = new Region(n) Integer;
    p.x.val = 21;
    p.y.val = 42;
    ... // Do something with point p.
end // All objects in Region(n), including p, p.x and p.y, get deallocated.
```

We omit a formal operational semantics.[2]

### 1.5 Typing Rules

*Region environments* are sets of region names and region variables. *Type environments* are partial functions mapping local program variables to types:

$$\Delta \subseteq \mathsf{RegionName} \cup \mathsf{RegionVar} \quad \text{region environments} \qquad \Gamma \in \mathsf{Var} \to \mathsf{Ty} \quad \text{type environments}$$

We write $\Delta \vdash r : \mathsf{ok}$ if all region variables and region names that occur in $r$ are members of $\Delta$. We write $\Delta \vdash T : \mathsf{ok}$ (respectively, $\Delta \vdash \Gamma : \mathsf{ok}$), if all classes that occur in $T$ (respectively, $\Gamma$) are declared in the underlying class table and all region variables and names that occur in $T$ (respectively, $\Gamma$) are members of $\Delta$. We sometimes use a comma on the right-hand-side of $\vdash$ to denote a conjunction. For instance, $(\Delta \vdash \Gamma, T : \mathsf{ok})$ abbreviates $(\Delta \vdash \Gamma : \mathsf{ok}$ and $\Delta \vdash T : \mathsf{ok})$.

Typing judgments for expressions have the form $\Delta; \Gamma \vdash e : T$. The typing rules are shown in Figure 2, and those for class and method declarations in Figure 3.

The single most interesting typing rule is the one for region name creation. It is very simple:

$$\frac{n \notin \Delta \quad \Delta, n; \Gamma \vdash e : T \quad \Delta \vdash \Gamma, T : \mathsf{ok}}{\Delta; \Gamma \vdash \texttt{newName}\ n\ \texttt{in}\ e\ \texttt{end} : T}$$

[2] An operational semantics for a similar language can be found in [HP09]. See also the references in [GMJ+02] for a formalization of the core of Cyclone.

Recall that we identify expressions up to renaming of bound names. Hence, the rule premise $n \notin \Delta$ can always be satisfied by renaming $n$. This premise forces the typechecker to do such a renaming before typechecking `newName`'s body $e$. As a consequence, the following unsafe expressions do not typecheck:

```
Region(n) Point<Region(n)> pt0;
pt0 = newName n in
          Region(n) Point<Region(n)> pt1;
          ...
          pt1 // TYPE ERROR: n in pt0's and pt1's type are not the same (by scoping)
      end
pt0.x; // UNSAFE MEMORY ACCESS

Region(n) Point<Region(n)> pt0;
newName n in
    Region(n) Point<Region(n)> pt1;
    ...
    pt0 = pt1; // TYPE ERROR: n in pt0's and pt1's type are not the same (by scoping)
    ...
end
pt0.x; // UNSAFE MEMORY ACCESS
```

What about objects that are reachable through a chain of field references? Why are accesses to deallocated regions prevented, even when following such reference chains? The key to this is the following simple heap invariant:

> *Heap invariant:*
> If object $o$ is reachable from object $p$ of type `Region(`$n_0$`)` $C$`<Region(`$n_1$`),...,Region(`$n_k$`)>`, then $o$'s type has the form `Region(`$m_0$`)` $D$`<Region(`$m_1$`),..,Region(`$m_l$`)>` where $\{m_0, m_1, \ldots, m_l\} \subseteq \{n_1, .., n_k\}$.

This invariant holds because region expressions occurring in field types can only be class parameters of the enclosing class. Because types of local variables only refer to regions that are in lexical scope, this heap invariant implies that all regions reachable from local variables are in lexical scope, too.

We omit a proper proof of memory safety, which would require the definition of an operational semantics and the proof of a type preservation theorem[3].

## 2 Lexically Scoped Regions and Typestate Transitions

How do lexically scoped regions help with typestate transitions, as for instance encountered in initialization problems for pluggable type systems to transfer from an "Uninitialized" to an "Initialized" object state? The idea is that instead of deallocating a region at the end of its scope, we change the region's type. Intuitively, one can think of this as first deallocating the region and then right away allocating it again, but with a different type. This deallocation/allocation does not really happen at runtime. Instead, the type system *recycles the region with a different type*. Whereas in region-based memory management `newName` is associated with a runtime instruction for memory deallocation, in typestate systems `newName` is associated with a typestate transition that does not have a runtime effect.

## 3 A Type System for Object and Reference Immutability

The type system we present here is a variation of the immutability type system in [HP09]. The difference to [HP09] is that here we tie the typestate change associated with region $n$ to the end of $n$'s lexical scope,

[3] [HP09] contains a type preservation proof for a similar system. See the references in [GMJ+02] for a soundness proof for a core of Cyclone.

$$\text{(Null)}\ \frac{\Delta \vdash \Gamma, T : \mathsf{ok} \quad T \neq \texttt{int}}{\Delta; \Gamma \vdash \texttt{null} : T} \qquad \text{(Int)}\ \frac{\Delta \vdash \Gamma : \mathsf{ok}}{\Delta; \Gamma \vdash i : \texttt{int}} \qquad \text{(Var)}\ \frac{\Delta \vdash \Gamma : \mathsf{ok} \quad \Gamma(x) = T}{\Delta; \Gamma \vdash x : T} \qquad \text{(Dcl)}\ \frac{x \notin \mathsf{dom}(\Gamma) \quad \Delta; \Gamma, x : T \vdash e : U}{\Delta; \Gamma \vdash T\, x; e : U}$$

$$\text{(Assign)}\ \frac{\Delta; \Gamma \vdash e : \Gamma(x)}{\Delta; \Gamma \vdash x = e : \Gamma(x)} \qquad \text{(Set)}\ \frac{\Delta; \Gamma \vdash e : q\ C\texttt{<}\bar{r}\texttt{>} \quad \texttt{class}\ C\texttt{<}\bar{\alpha}\texttt{>}\ \texttt{\{..}\, T\, f\, \texttt{..\}} \quad \Delta; \Gamma \vdash e' : T[\bar{r}/\bar{\alpha}]}{\Delta; \Gamma \vdash e.f = e' : T[\bar{r}/\bar{\alpha}]}$$

$$\text{(Get)}\ \frac{\Delta; \Gamma \vdash e : q\ C\texttt{<}\bar{r}\texttt{>} \quad \texttt{class}\ C\texttt{<}\bar{\alpha}\texttt{>}\ \texttt{\{..}\, T\, f\, \texttt{..\}}}{\Delta; \Gamma \vdash e.f : T[\bar{r}/\bar{\alpha}]} \qquad \text{(Seq)}\ \frac{\Delta; \Gamma \vdash e : T \quad \Delta; \Gamma \vdash e' : U}{\Delta; \Gamma \vdash e; e' : U} \qquad \text{(New)}\ \frac{C\ \text{declared} \quad \Delta \vdash q, \bar{r} : \mathsf{ok}}{\Delta; \Gamma \vdash \texttt{new}\ q\ C\texttt{<}\bar{r}\texttt{>} : q\ C\texttt{<}\bar{r}\texttt{>}}$$

$$\text{(Call)}\ \frac{\Delta \vdash \bar{r} : \mathsf{ok} \quad \texttt{<}\bar{\alpha}\texttt{>}\ U\ m\texttt{(}\bar{T}\ \bar{x}\texttt{)\{}e\texttt{\}} \quad \Delta; \Gamma \vdash \bar{e} : \bar{T}[\bar{r}/\bar{\alpha}]}{\Delta; \Gamma \vdash \texttt{<}\bar{r}\texttt{>}m\texttt{(}\bar{e}\texttt{)} : U[\bar{r}/\bar{\alpha}]} \qquad \text{(New Name)}\ \frac{n \notin \Delta \quad \Delta, n; \Gamma \vdash e : T \quad \Delta \vdash \Gamma, T : \mathsf{ok}}{\Delta; \Gamma \vdash \texttt{newName}\ n\ \texttt{in}\ e\ \texttt{end} : T}$$

Figure 2: Well-typed expressions, $\Delta; \Gamma \vdash e : T$

$$\text{(Class)}\ \frac{\bar{\alpha} \vdash \bar{T} : \mathsf{ok}}{\vdash \texttt{class}\ C\texttt{<}\bar{\alpha}\texttt{>}\ \texttt{\{}\,\bar{T}\ \bar{f}\,\texttt{\}} : \mathsf{ok}} \qquad \text{(Method)}\ \frac{\bar{\alpha}; \bar{x} : \bar{T} \vdash e : U}{\vdash \texttt{<}\bar{\alpha}\texttt{>}\ U\ m\texttt{(}\bar{T}\ \bar{x}\texttt{)\{}e\texttt{\}} : \mathsf{ok}}$$

Figure 3: Well-typed class and method declarations

whereas in [HP09] we have a special specification command that allows to make the typestate change anywhere within $n$'s scope (for instance inside the branch of a conditional) except inside a loop. The system in [HP09] is thus a bit more flexible. The extra flexibility seems useful in practice, for instance, to deal with exceptions. However, here we want to present a system that emphasizes the similarity to lexically scoped regions. We focus on the presentation of the formal type system. For more examples and discussion of immutability types, see [HP09].

### 3.1 Type Qualifiers

Initialized objects have an access qualifier `RdWr` (object fields can be read and written) or `Rd` (object fields can be read but not written). In addition, there is the access qualifier `Any`, which is the least upper bound of `RdWr` and `Rd`. Objects that `Any`-references refer to may either be `RdWr` or `Rd`, and the type system disallows writing through `Any`-references. Finally, there are access qualifiers `Fresh(`$n$`)`. These are associated with objects that are still in their initialization phase. It is allowed to write to `Fresh(`$n$`)`-objects. One can think of the argument $n$ as the name of a memory region containing a set of related objects that are currently under initialization.

| | | |
|---|---|---|
| $n \in \mathsf{Name}$ | names | $\alpha \in \mathsf{QVar}$ qualifier variables, including `myaccess` |
| $p, q, r \in \mathsf{Qual}$ | ::= | type qualifiers |
| | `RdWr` | read-write access (default) |
| | `Rd` | read-only access |
| | `Any` | least upper bound of `RdWr` and `Rd` |
| | `Fresh(`$n$`)` | fresh object currently under initialization inside region $n$ |
| | $\alpha$ | qualifier variable |

*Subqualifying* is the least partial order such that `Rd` <: `Any` and `RdWr` <: `Any`.

### 3.2 Classes and Types

$$\begin{array}{lll} \mathit{class} & ::= & \texttt{class}\ C\ \texttt{\{}\ \bar{T}\ \bar{f}\ \texttt{\}} \\ T, U, V \in \mathsf{Ty} & ::= & q\ C \mid \texttt{int} \mid \texttt{void} \end{array} \qquad \begin{array}{l} \text{class declarations} \\ \text{types} \end{array}$$

For simplicity, in this system we omit explicit class parameters, although they can be useful (see [HP09] for an example) and would be unproblematic. Crucially, though, classes have one implicit class parameter `myaccess` that, when mentioned in a field type, refers to the access right of the enclosing record[4]. This class parameter allows us to use a single access qualifier for specifying the access rights for an entire object graph. Furthermore, having at least on class parameter is crucial for dealing with simultaneous initialization of multiple related objects, as often needed to initialize cyclic structures.

Object types are of the form $q\ C$. Unlike in region-based memory management, the type qualifier $q$ is ignored at runtime. (In region-based memory management the qualifier is needed for region deallocation at the end of a region's lexical scope.) *Subtyping* is the least partial order such that $p\ C <: q\ C$ for all $p <: q$ and $C$.

If an object has a type of the form `Rd` $C$, we call it a `Rd`-*object*. The immutability type system is designed to satisfy the following safety property:

**Safety Property:** Well-typed programs do not write to fields of `Rd`-objects.

**Example** In the `Square` class below, `myaccess` annotates the type `Point` of its fields. The method `m()` takes in a `Rd`-square `s`. Mutating the `Point`-fields of `s` and mutating the `int`-fields of `s`'s points is forbidden.

```
class Point { int x; int y; }
class Square { myaccess Point upperleft; myaccess Point lowerright; }
void m(Rd Square s) {
    s.upperleft = s.lowerright; // TYPE ERROR
    s.upperleft.x = 42; // TYPE ERROR
}
```

### 3.3 Bounded Qualifier Polymorphism for Methods

Our system has bounded qualifier polymorphism. We introduce the *qualifier bounds* `Writeable` and `Qual`. These can only be used as qualifier bounds, but not as qualifiers. `Any` can be used both as a qualifier and a bound.

$$B \in \mathsf{QualBound} \ ::= \ \texttt{Writeable} \mid \texttt{Any} \mid \texttt{Qual} \qquad \text{qualifier bounds}$$

The order on qualifiers and bounds is formalized by the simple system displayed in Figure 3.3, which also depicts the qualifier hierarchy as a Hasse diagram.

In qualifier-polymorphic methods, programmers may constrain the qualifier parameters by bounds:

$$\mathit{method} \ ::= \ \texttt{<}\bar{\alpha} \triangleleft \bar{B}\texttt{>}\ T\ m\texttt{(}\bar{T}\ \bar{x}\texttt{)\{}e\texttt{\}} \qquad \text{method declaration}$$

**Example** The following `copy`-method may be applied to actual `dst`-parameters of type `RdWr Point` or `Fresh(`$n$`) Point`, but not of type `Rd Point` or `Any Point`.

```
<a ◁ Qual, b ◁ Writeable> void copy(a Point src, b Point dst) {
    dst.x = src.x; dst.y = src.y;
}
```

[4] We could also add an analogous class parameter `myregion` to the region type system.

$$\Delta ::= \cdot \mid \Delta, \alpha \triangleleft B \mid \Delta, n \qquad \text{qualifier environments}$$

$$\frac{}{\Delta, \alpha \triangleleft B, \Delta' \vdash \alpha \triangleleft B} \qquad \frac{q <: \texttt{Any}}{\Delta \vdash q \triangleleft \texttt{Any}} \qquad \frac{\Delta \vdash q \triangleleft B}{\Delta \vdash q \triangleleft \texttt{Qual}} \qquad \frac{}{\Delta \vdash \texttt{RdWr} \triangleleft \texttt{Writeable}} \qquad \frac{n \in \Delta}{\Delta \vdash \texttt{Fresh}(n) \triangleleft \texttt{Writeable}}$$

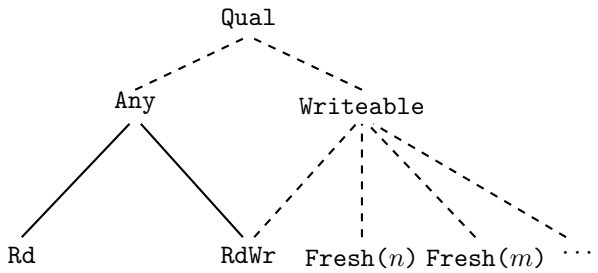



Figure 4: The qualifier hierarchy. `Qual` and `Writable` are qualifier *bounds*, not qualifiers, so they cannot be used as type qualifiers, only as bounds of qualifier parameters.

### 3.4 Expressions and Typing Rules

The `newName` expression now has an additional access qualifier argument $q$ in order to indicate to the type system that after $e$ terminates the qualifier `Fresh(`$n$`)` gets converted to $q$.

- `newName` $n$ `for` $q$ `in` $e$ `end`: This has no effect at runtime. To the typechecker, it introduces a new name $n$ with scope $e$ and tells the typechecker that `Fresh(`$n$`)` becomes $q$ after $e$ terminates.

Figures 5 and 6 show the typing rules, highlighting the differences to the region system in bold. The differences to the region system are: firstly, there is now a subsumption rule to capture the order on access qualifiers; secondly, in the rule for setting fields the type system checks that the object is writeable; thirdly, in the rule for method calls the type system checks that qualifier parameters satisfy their bounds; fourthly, in the type for `newName` $n$ `for` $q$ `in` $e$ `end` the type system substitutes substitutes $q$ for `Fresh(`$n$`)` in order to reflect the typestate change.

In the rules, we use the following abbreviations: $\Delta \vdash q : \mathsf{ok}$ means $\Delta \vdash q \triangleleft \texttt{Qual}$; $\Delta \vdash T : \mathsf{ok}$ means $\Delta \vdash q : \mathsf{ok}$ for all qualifiers $q$ that occur in $T$; $\Delta \vdash \Gamma : \mathsf{ok}$ means $\Delta \vdash \Gamma(x) : \mathsf{ok}$ for all $x$ in $\mathsf{dom}(\Gamma)$.

**Example** We show two factory methods for `Point` objects. First, a factory method that takes care of both object creation and object initialization:

```
<a ◁ Qual> a Point factory1(int x, int y) {
  newName n for a in
    Fresh(n) Point p = new Fresh(n) Point; p.x=x; p.y=y; p
  end
}
```

Now a factory method that takes care of object creation and part of object initialization, but leaves the completion of object initialization to the client:

```
<a ◁ Writeable> a Point factory2(int x) {
  a Point p = new a Point; p.x=x; p
}
```

Here is a client of this method:

```
newName n for Rd in
  Fresh(n) Point p = <Fresh(n)>factory2(7); p.y=3
end
```

$$\frac{\Delta \vdash \Gamma, T : \mathsf{ok} \quad T \neq \texttt{int}}{\Delta; \Gamma \vdash \texttt{null} : T}\ (\text{Null}) \qquad \frac{\Delta \vdash \Gamma : \mathsf{ok}}{\Delta; \Gamma \vdash i : \texttt{int}}\ (\text{Int}) \qquad \frac{\Delta \vdash \Gamma : \mathsf{ok} \quad \Gamma(x) = T}{\Delta; \Gamma \vdash x : T}\ (\text{Var}) \qquad \frac{\Delta; \Gamma \vdash e : T \quad T <: U}{\Delta; \Gamma \vdash e : U}\ \textbf{(Sub)}$$

$$\frac{x \notin \mathsf{dom}(\Gamma) \quad \Delta; \Gamma, x : T \vdash e : U}{\Delta; \Gamma \vdash T\, x; e : U}\ (\text{Dcl}) \qquad \frac{\Delta; \Gamma \vdash e : \Gamma(x)}{\Delta; \Gamma \vdash x = e : \Gamma(x)}\ (\text{Assign})$$

$$\frac{\Delta; \Gamma \vdash e : q\ C \quad \mathbf{\Delta} \vdash \mathbf{q} \triangleleft \texttt{Writeable} \quad \texttt{class}\ C\ \texttt{\{..}\ T\ f\ \texttt{..\}} \quad \Delta; \Gamma \vdash e' : T[q/\texttt{myaccess}]}{\Delta; \Gamma \vdash e.f = e' : T[q/\texttt{myaccess}]}\ (\text{Set})$$

$$\frac{\Delta; \Gamma \vdash e : q\ C \quad \texttt{class}\ C\ \texttt{\{..}\ T\ f\ \texttt{..\}}}{\Delta; \Gamma \vdash e.f : T[q/\texttt{myaccess}]}\ (\text{Get}) \qquad \frac{\Delta; \Gamma \vdash e : T \quad \Delta; \Gamma \vdash e' : U}{\Delta; \Gamma \vdash e; e' : U}\ (\text{Seq}) \qquad \frac{C \text{ declared} \quad \Delta \vdash q : \mathsf{ok}}{\Delta; \Gamma \vdash \texttt{new}\ q\ C : q\ C}\ (\text{New})$$

$$\frac{\texttt{<}\bar{\alpha} \triangleleft \bar{B}\texttt{>}\ U\ m\texttt{(}\bar{T}\ \bar{x}\texttt{)\{}e\texttt{\}} \quad \mathbf{\Delta} \vdash \bar{\mathbf{q}} \triangleleft \bar{\mathbf{B}} \quad \Delta; \Gamma \vdash \bar{e} : \bar{T}[\bar{q}/\bar{\alpha}]}{\Delta; \Gamma \vdash \texttt{<}\bar{q}\texttt{>}m\texttt{(}\bar{e}\texttt{)} : U[\bar{q}/\bar{\alpha}]}\ (\text{Call}) \qquad \frac{n \notin \Delta \quad \Delta, n; \Gamma \vdash e : T \quad \Delta \vdash \Gamma, q : \mathsf{ok}}{\Delta; \Gamma \vdash \texttt{newName}\ n\ \texttt{for}\ q\ \texttt{in}\ e\ \texttt{end} : \mathbf{T}[\mathbf{q}/\texttt{Fresh(n)}]}\ (\text{New Name})$$

Figure 5: Well-typed expressions, $\Delta; \Gamma \vdash e : T$

$$\frac{\texttt{myaccess} \triangleleft \texttt{Qual} \vdash \bar{T} : \mathsf{ok}}{\vdash \texttt{class}\ C\ \texttt{\{}\bar{T}\ \bar{f}\texttt{\}} : \mathsf{ok}}\ (\text{Class}) \qquad \frac{\bar{\alpha} \triangleleft \bar{B}; \bar{x} : \bar{T} \vdash e : U}{\vdash \texttt{<}\bar{\alpha} \triangleleft \bar{B}\texttt{>}\ U\ m\texttt{(}\bar{T}\ \bar{x}\texttt{)\{}e\texttt{\}} : \mathsf{ok}}\ (\text{Method})$$

Figure 6: Well-typed class and method declarations

## 4 A Type System for Non-Nullness

A type system for non-nullness is designed to guarantee the following safety property:

> **Safety Property:** Well-typed programs do not attempt to dereference `null`.

The type system we present is essentially Fähndrich and Xia's system of delayed types [FX07], which is under the hood of Spec♯'s non-nullness checker. For better comparison with the region system from Section 1 and the immutability system from Section 3, we change Fähndrich and Xia's notation a bit.

In non-nullness type systems some object fields are annotated as holding non-null values. Unfortunately, initially all object fields hold their default values, which is `null` for reference types. Therefore, the non-nullness annotations on fields are only valid after object initialization. A non-nullness system has to distinguish between uninitialized and initialized objects.

The nature of a non-nullness type system is somewhat different from an immutability type system: Whereas a non-nullness system is primarily concerned with *guaranteeing an object property* (namely that certain fields are non-null), an immutability type system is primarily concerned with *restricting object access permissions* (namely the permission to write). Whereas in a non-nullness type system object initialization is associated with an *obligation to establish a property*, in an immutability type system object initialization is associated with a *temporary permission to write*. Whereas in a non-nullness type system initialized objects have *more properties* than fresh ones, in an immutability type system initialized objects are associated with *less permissions* than fresh ones.

Despite this difference, lexically scoped regions can be employed for initialization in a very similar way as for immutability. However, the fact that non-nullness of fields is an object property poses an additional complication: the system has to ensure that at the end of object initialization this property is indeed established. To this end, Fähndrich and Xia's type system keeps track of a write effect that collects object fields that have been written. Using our terminology, their type system ensures that at the end of a (`newName` $n$ `in` $e$ `end`)-expression the non-null fields of `Fresh(`$n$`)`-objects have been written at least once. Technically, they achieve this by tying object allocation to a `let`-expression (`let` $x$ `=new Fresh(`$n$`)` $C$ `in` $e$ `end`). For an expression of this form, their type system checks that the write effect of $e$ contains all non-null fields of $x$. Because the system forbids assigning `null` to non-null fields (even during initialization), this ensures that upon $e$'s termination $x$'s non-null fields contain non-null values and that this is still true when the end of $n$'s scope is reached.

### 4.1 Type Qualifiers

In the non-nullness system, types have a qualifier that indicates if an object is in an `Initialized` or `Fresh` state. Just like in the immutability system, the `Fresh` qualifier has a name as an argument, so that we can apply lexically scoped regions for a safe state transition from `Fresh` to `Initialized`[5]. Just like in the immutability system, we use a special class parameter that may be mentioned in field types and refers to the type qualifier of the enclosing object. Unlike in the immutability type system, there is no ordering on type qualifiers. Intuitively, the qualifiers `Rd`, `RdWr` and `Any` are collapsed into the single qualifier `Initialized`.

[5] [FX07]'s delay time `Now` corresponds to our qualifier `Initialized`. [FX07]'s time variables correspond to our names. We present a simplified version of [FX07]'s system without an ordering on time variables. The resulting system is slightly weaker than the one presented in [FX07], but Spec♯ does not implement time constraints anyways, see Section 5.5 of [FX07], so the ordering on time variables is not used in Spec♯. [FX07]'s `delay` $t$ `in` $e$ corresponds to our `newName` $n$ `in` $e$ `end`.

$n \in \mathsf{Name}$ names $\quad \alpha \in \mathsf{QVar}$ qualifier variables, including `mystate` $\quad \beta \in \mathsf{FVar}$ freshness variables

$\iota ::= n \mid \beta$ freshness expressions

$p, q, r \in \mathsf{Qual} ::=$ type qualifiers

| | |
|---|---|
| `Initialized` | initialized object |
| `Fresh(`$\iota$`)` | fresh object currently under initialization inside region $\iota$ |
| $\alpha$ | qualifier variable |

### 4.2 Classes and Types

$$\mathit{class} ::= \texttt{class}\ C\ \texttt{\{}\ \bar{T}\ \bar{f}\ \texttt{\}} \quad \text{class declarations}$$

$\tau ::=$ unqualified types

| | |
|---|---|
| $C$ | $C$-object or `null` |
| $C!$ | $C$-object and not `null` |

$$T, U, V \in \mathsf{Ty} ::= q\ \tau \mid \texttt{int} \mid \texttt{void} \quad \text{types}$$

*Subtyping* is the least partial order such that $q\ C! <: q\ C$ for all $q$, $C$. For a class $C$, let $\mathsf{nnfields}(C)$ be the set of fields that have a non-null type, i.e., a type of the form $q\ D!$ for some $q, D$.

### 4.3 Methods

Effects are sets of variable-field pairs $x.f$:

$$E \subseteq \mathsf{Var} \times \mathsf{FieldId} \quad \text{write effect}$$

For an effect $E$, let $\pi_x(E) = \{y.f \in E \mid y = x\}$.

Methods are polymorphic in qualifiers and names[6]. A write-effect $E$ specifies the fields that $m$ writes. In well-typed method tables, elements of $E$ are always of the form $x.f$ where $x$ is a method parameter.

$$\mathit{method} ::= \texttt{<}\bar{\alpha}, \bar{\beta}\texttt{>}\ U\ m\texttt{(}\bar{T}\ \bar{x}\texttt{)}\ E\texttt{\{}e\texttt{\}} \quad \text{method declarations (scope of } \bar{\alpha}, \bar{\beta} \text{ is } U, \bar{T}, e \text{, and scope of } \bar{x} \text{ is } E, e)$$

### 4.4 Expressions and Typing Rules

Typing judgments for expressions have the following form:

$$\Delta; \Gamma \vdash e : T; E$$

where $\Delta$ is a variable-and-name environment:

$$\Delta \subseteq \mathsf{QVar} \cup \mathsf{FVar} \cup \mathsf{Name} \quad \text{variable-and-name environment}$$

As discussed above, object generation is associated with a `let`-expression. The type system checks that all non-null fields of the new object have been written at the end of the `let`-body $e$:

$$\frac{C \text{ declared} \quad x \notin \mathsf{dom}(\Gamma) \quad \Delta; \Gamma, x : \texttt{Fresh(}\iota\texttt{)}\ C \vdash e : T; E \quad \mathsf{nnfields}(C) \subseteq \pi_x(E)}{\Delta; \Gamma \vdash \texttt{let}\ x\texttt{=new Fresh(}\iota\texttt{)}\ C\ \texttt{in}\ e\ \texttt{end} : T; (E \setminus \pi_x(E))}$$

It is important to take tests for non-nullness into account. In the following rule, note that, in the right branch of the conditional, $x$'s type is promoted to a non-null type:

$$\frac{\Delta; \Gamma \vdash x : q\ C; \emptyset \quad \Delta; \Gamma \vdash e : T; E \quad \Delta; \Gamma[x \mapsto q\ C!] \vdash e' : T; E'}{\Delta; \Gamma \vdash \texttt{ifnull}\ x\ \texttt{then}\ e\ \texttt{else}\ e' : T; E \cap E'}$$

Unfortunately, there is an additional complication in the non-nullness system, namely, that the obvious rule for setting fields is too restrictive. The obvious rule for setting fields is obtained by analogy with the region and the immutability system:

[6]Because we syntactically distinguish between names and qualifiers, we do not need to introduce constraints of the form `Initialized` $= \alpha$ corresponding to [FX07]'s constraints `Now` $< \alpha$.

$$\frac{\Delta;\Gamma \vdash x : q\ C!;\emptyset \quad \texttt{class}\ C\ \{..\ T\ f\ ..\} \quad \Delta;\Gamma \vdash e : T[q/\texttt{mystate}];E}{\Delta;\Gamma \vdash x.f = e : T[q/\texttt{mystate}];E \cup \{x.f\}}$$

The problem is that this rule disallows assigning `Initialized` objects to `Fresh`-typed fields[7]. This is needed when inserting an element into a mutable circular data structure[8]. See, for instance, the running example in [FX07], which is a doubly linked list with an insertion function. To be able to handle such examples, [FX07] makes the rule for setting fields more liberal:

$$\frac{\Delta;\Gamma \vdash x : q\ C!;\emptyset \quad \texttt{class}\ C\ \{..\ T\ f\ ..\} \quad T[q/\texttt{mystate}] = p\ \tau \quad \Delta;\Gamma \vdash e : p'\ \tau;E \quad p' = \texttt{Initialized} \text{ or } p' = p}{\Delta;\Gamma \vdash x.f = e : \texttt{void};E \cup \{x.f\}}$$

Now, that the rule for writing fields is more liberal, it is important to adjust the rule for reading fields correspondingly. When we read a field, we cannot expect that the field holds a value of the field type, because the rule for writing fields allows writing `Initialized` values into `Fresh`-typed fields. Note, however, that the rule for writing enforces that only `Initialized` values are written into `Initialized`-typed fields. Therefore, one could use the following rule for reading fields:

$$\frac{\Delta;\Gamma \vdash e : q\ C!;E \quad \texttt{class}\ C\ \{..\ T\ f\ ..\} \quad T[q/\texttt{mystate}] = \texttt{Initialized}\ \tau}{\Delta;\Gamma \vdash e.f : T[q/\texttt{mystate}];E}$$

This rule entirely disallows reading `Fresh`-typed fields. Fähndrich and Xia use a more liberal rule that allows reading `Fresh`-typed fields and associates a *qualifier wildcard* with the retrieved value:

$$\frac{\Delta;\Gamma \vdash e : q\ C!;E \quad \texttt{class}\ C\ \{..\ T\ f\ ..\} \quad T[q/\texttt{mystate}] = p\ \tau <: p\ D \quad U = \begin{cases} p\ \tau & \text{if } p = \texttt{Initialized} \\ \texttt{?}\ D & \text{otherwise} \end{cases}}{\Delta;\Gamma \vdash e.f : U;E}$$

In case $p \neq \texttt{Initialized}$, note that we use a qualifier wildcard `?` in place of a proper qualifier in $e.f$'s type. Moreover, we upcast $e.f$'s type to a possibly-null type, because $e$ is possibly uninitialized. This rule allows reading `Fresh`-typed fields. On the other hand, the type system disallows writing to the retrieved object, which could lead to unsoundness.

*Qualifier wildcards* are denoted by `?`. They are analogous to wildcards in Java's generic types and express that the real qualifier is unknown. In order to be able to do anything at all with objects that are associated with qualifier wildcards, Fähndrich and Xia introduce another specification command for unpacking wildcards. This `unpack`-command replaces a qualifier wildcard by a fresh qualifier variable — a process called *wildcard capture* in Java generics[9]:

$$\frac{\Delta;\Gamma \vdash e : \texttt{?}\ \tau;E \quad \alpha \notin \Delta \quad x \notin \mathsf{dom}(\Gamma) \quad \Delta,\alpha;\Gamma,x : \alpha\ \tau \vdash e' : T;E'}{\Delta;\Gamma \vdash \texttt{unpack}\ \alpha, x = e\ \texttt{in}\ e'\texttt{end} : T;E \cup (E' \setminus \pi_x(E'))}$$

Note that this rule makes it almost[10] impossible for $e'$ to write to fields of object $x$, because $x$ is the only value whose qualifier is $\alpha$.

Figures 7 and 8 display (almost) all typing rules for the model language. We have omitted the rules for reading and writing fields of type `int`, because they are not so interesting.

[7] `Fresh`-typed fields can arise through instantiating the `mystate`-parameter in classes of `Fresh` objects.

[8] Our immutability system [HP09] does not run into this problem, because one never inserts into *immutable* circular data structures, and for mutable data structures one can construct objects as `RdWr` rather than as `Fresh`.

[9] In practice, the programmer never has to write `unpack`-commands, but Spec$^\sharp$'s typechecker inserts them automatically on the fly.

[10] The only value that can possibly be assigned to $x.f$ is $x$ itself, i.e., $x.f = x$.

(Null)
$$\frac{\Delta \vdash \Gamma, T : \mathsf{ok} \quad T \neq \mathtt{int}}{\Delta; \Gamma \vdash \mathtt{null} : T; \emptyset}$$

(Int)
$$\frac{\Delta \vdash \Gamma : \mathsf{ok}}{\Delta; \Gamma \vdash i : \mathtt{int}; \emptyset}$$

(Var)
$$\frac{\Delta \vdash \Gamma : \mathsf{ok} \quad \Gamma(x) = T}{\Delta; \Gamma \vdash x : T; \emptyset}$$

(Sub)
$$\frac{\Delta; \Gamma \vdash e : T; E \quad T <: U}{\Delta; \Gamma \vdash e : U; E}$$

(Let)
$$\frac{\Delta; \Gamma \vdash e : T; E \quad x \notin \mathsf{dom}(\Gamma) \quad \Delta; \Gamma, x : T \vdash e' : T'; E'}{\Delta; \Gamma \vdash \mathtt{let}\ x = e\ \mathtt{in}\ e'\ \mathtt{end} : T'; E \cup (E' \setminus \pi_x(E'))}$$

(Seq)
$$\frac{\Delta; \Gamma \vdash e : T; E \quad \Delta; \Gamma \vdash e' : T'; E'}{\Delta; \Gamma \vdash e; e' : T'; E \cup E'}$$

(Set)
$$\frac{\Delta; \Gamma \vdash x : q\ C!; \emptyset \quad \mathtt{class}\ C\ \{..\ T\ f\ ..\} \quad T[q/\mathtt{mystate}] = p\ \tau \quad \Delta; \Gamma \vdash e : p'\ \tau; E \quad p' = \mathtt{Initialized} \text{ or } p' = p}{\Delta; \Gamma \vdash x.f = e : \mathtt{void}; E \cup \{x.f\}}$$

(Get)
$$\frac{\Delta; \Gamma \vdash e : q\ C!; E \quad \mathtt{class}\ C\ \{..\ T\ f\ ..\} \quad T[q/\mathtt{mystate}] = p\ \tau <: p\ D \quad U = \begin{cases} p\ \tau & \text{if } p = \mathtt{Initialized} \\ ?\ D & \text{otherwise} \end{cases}}{\Delta; \Gamma \vdash e.f : U; E}$$

(New)
$$\frac{C \text{ declared} \quad x \notin \mathsf{dom}(\Gamma) \quad \Delta; \Gamma, x : \mathtt{Fresh}(\iota)\ C \vdash e : T; E \quad \mathsf{nnfields}(C) \subseteq \pi_x(E)}{\Delta; \Gamma \vdash \mathtt{let}\ x = \mathtt{new}\ \mathtt{Fresh}(\iota)\ C\ \mathtt{in}\ e\ \mathtt{end} : T; (E \setminus \pi_x(E))}$$

(If Null)
$$\frac{\Delta; \Gamma \vdash x : q\ C; \emptyset \quad \Delta; \Gamma \vdash e : T; E \quad \Delta; \Gamma[x \mapsto q\ C!] \vdash e' : T; E'}{\Delta; \Gamma \vdash \mathtt{ifnull}\ x\ \mathtt{then}\ e\ \mathtt{else}\ e' : T; E \cap E'}$$

(Call)
$$\frac{\mathtt{<}\bar{\alpha}, \bar{\beta}\mathtt{>}\ U\ m\mathtt{(}\bar{T}\ \bar{x}\mathtt{)}E\mathtt{\{}e\mathtt{\}} \quad |\bar{q}| = |\bar{\alpha}| \quad |\bar{\iota}| = |\bar{\beta}| \quad \Delta; \Gamma \vdash \bar{y} : \bar{T}[\bar{q}, \bar{\iota}/\bar{\alpha}, \bar{\beta}]; \emptyset}{\Delta; \Gamma \vdash \mathtt{<}\bar{q}, \bar{\iota}\mathtt{>}m\mathtt{(}\bar{y}\mathtt{)} : U[\bar{q}, \bar{\iota}/\bar{\alpha}, \bar{\beta}]; E[\bar{y}/\bar{x}]}$$

(New Name)
$$\frac{n \notin \Delta \quad \Delta, n; \Gamma \vdash e : T; E \quad \Delta \vdash \Gamma : \mathsf{ok}}{\Delta; \Gamma \vdash \mathtt{newName}\ n\ \mathtt{in}\ e\ \mathtt{end} : T[\mathtt{Initialized}/\mathtt{Fresh}(n)]; E}$$

(Unpack)
$$\frac{\Delta; \Gamma \vdash e : ?\ \tau; E \quad \alpha \notin \Delta \quad x \notin \mathsf{dom}(\Gamma) \quad \Delta, \alpha; \Gamma, x : \alpha\ \tau \vdash e' : T; E'}{\Delta; \Gamma \vdash \mathtt{unpack}\ \alpha, x = e\ \mathtt{in}\ e'\mathtt{end} : T; E \cup (E' \setminus \pi_x(E'))}$$

Figure 7: Well-typed expressions, $\Delta; \Gamma \vdash e : T; E$

(Class)
$$\frac{\mathtt{mystate} \vdash \bar{T} : \mathsf{ok}}{\vdash \mathtt{class}\ C\ \mathtt{\{}\ \bar{T}\ \bar{f}\ \mathtt{\}} : \mathsf{ok}}$$

(Method)
$$\frac{\bar{\alpha}, \bar{\beta}; \bar{x} : \bar{T} \vdash e : U; E}{\vdash \mathtt{<}\bar{\alpha}, \bar{\beta}\mathtt{>}\ U\ m\mathtt{(}\bar{T}\ \bar{x}\mathtt{)}E\mathtt{\{}e\mathtt{\}} : \mathsf{ok}}$$

Figure 8: Well-typed class and method declarations

### 4.5 Spec♯'s Surface Syntax for Delayed Types

Spec♯'s surface syntax hides the type qualifiers and qualifier parameters behind the single type qualifier `Delayed`. Here is how Spec♯ desugars to the core language:

- Every instance field is implicitly qualified by the variable `mystate`.
- Unqualified reference types in method and constructor signatures are implicitly qualified by `Initialized`.
- Every method is implicitly parameterized by a single qualifier parameter $\alpha$. Within the scope of $\alpha$, type qualifiers `Delayed` are translated to $\alpha$. A method annotation `Delayed` is translated to an $\alpha$-annotation on the method's receiver parameter.
- Every constructor is implicitly parametrized by a single freshness parameter $\beta$. Within the scope of $\beta$:
  - The type of `this` is implicitly annotated by `Fresh(`$\beta$`)`.
  - `Delayed` annotations on types of constructors parameters are translated to `Fresh(`$\beta$`)`.
  - Constructor calls `new` $C$`[Delayed](...)` inside the constructor body are translated to:

    `let tmp = new Fresh(`$\beta$`)` $C$ `in tmp.<Fresh(`$\beta$`)>ctor(...); end; tmp`

- Constructor calls `new` $C$`(...)` are translated to:

  `newName` $n$ `in let tmp = new Fresh(`$n$`)` $C$ `in tmp.<Fresh(`$n$`)>ctor(...); end; tmp end`

## References


[FX07] M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *Object-Oriented Programming Systems, Languages, and Applications*, pages 337–350. ACM, 2007.

[GMJ+02] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Programming Languages Design and Implementation*, pages 282–293, 2002.

[HP09] C. Haack and E. Poll. Type-based object immutability with flexible initialization. Technical Report ICIS-R09001, Radboud University, Nijmegen, January 2009.

[IPW01] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[TT97] M. Tofte and J-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.