Reference Capabilities for Flexible Memory Management

ELLEN ARVIDSSON, Uppsala University, Sweden
ELIAS CASTEGREN, Uppsala University, Sweden
SYLVAN CLEBSCH, Microsoft, USA
SOPHIA DROSSOPOULOU, Imperial College London, England
JAMES NOBLE, Creative Research & Programming, New Zeeland
MATTHEW J. PARKINSON, Microsoft, UK
TOBIAS WRIGSTAD, Uppsala University, Sweden

Verona is a concurrent object-oriented programming language that organises all the objects in a program into a forest of isolated regions. Memory is managed locally for each region, so programmers can control a program's memory use by adjusting objects' partition into regions, and by setting each region's memory management strategy. A thread can only mutate (allocate, deallocate) objects within one active region — its "window of mutability". Memory management costs are localised to the active region, ensuring overheads can be predicted and controlled. Moving the mutability window between regions is explicit, so code can be executed wherever it is required, yet programs remain in control of memory use. An ownership type system based on reference capabilities enforces region isolation, controlling aliasing within and between regions, yet supporting objects moving between regions and threads. Data accesses never need expensive atomic operations, and are always thread-safe.

CCS Concepts: • **Software and its engineering → Imperative languages**; **Semantics**.

Additional Key Words and Phrases: memory management, type systems, isolation, ownership

ACM Reference Format:
Ellen Arvidsson, Elias Castegren, Sylvan Clebsch, Sophia Drossopoulou, James Noble, Matthew J. Parkinson, and Tobias Wrigstad. 2023. Reference Capabilities for Flexible Memory Management. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 270 (October 2023), 31 pages. https://doi.org/10.1145/3622846

1 INTRODUCTION

Memory management has always been challenging, and programmers and programming language designers have developed a wide range of techniques and patterns to deal with it [Noble and Weir 2000]. Most early languages like FORTRAN and COBOL supported only fixed memory allocation, where memory was allocated before a program began to execute. Algol-60 popularised stack allocation, enabling recursive procedures to be expressed clearly, and then languages like C and Pascal popularised heap allocation, where programmers could manually request memory from the

Authors' addresses: Ellen Arvidsson, Information Technology, Uppsala University, Lägerhyddsvägen 1, Uppsala, 751 05, Sweden, ellen.arvidsson@it.uu.se; Elias Castegren, Information Technology, Uppsala University, Lägerhyddsvägen 1, Uppsala, 751 05, Sweden, elias.castegren@it.uu.se; Sylvan Clebsch, Azure Research, Microsoft, 6200 Turtle Point Drive, Austin, 78746, TX, USA, sylvan.clebsch@microsoft.com; Sophia Drossopoulou, Department of Computing, Imperial College London, 180 Queen's Gate, London, SW7 2BZ, England, s.drossopoulou@imperial.ac.uk; James Noble, Creative Research & Programming, 5 Fernlea Ave, Wellington, 6012, New Zeeland, kjx@acm.org; Matthew J. Parkinson, Azure Research, Microsoft, 21 Station Road, Cambridge, CB1 2FB, UK, mattpark@microsoft.com; Tobias Wrigstad, Information Technology, Uppsala University, Lägerhyddsvägen 1, Uppsala, 751 05, Sweden, tobias.wrigstad@it.uu.se.

2475-1421/2023/10-ART270
https://doi.org/10.1145/3622846

runtime system, and manually return that memory when it was no longer required. LISP introduced the first automatic memory management system—a garbage collector—which relieved programmers from the need to explicitly free memory, rather memory will be automatically reclaimed once it is no longer needed [Jones et al. 2016]. As well as reducing the amount of bookkeeping code programmers have to write, garbage collection typically provides "memory safety" which prevents a number of characteristic errors common to manual memory management, such as failing to free objects that are no longer needed, or accidentally freeing objects that are still in use.

While there is now a 60+ year history of research in garbage collection algorithms and implementations, many programmers seem resistant to using garbage collection, despite the pitfalls of manual memory management. According to the TIOBE index of programming languages [Tiobe 2022], about half out of the top twenty programming languages eschew garbage collection, and rely on various forms of manual memory management. The 1st and 3rd of the top 25 Common Weaknesses in CWE 2020–2022 are writing and reading outside of allocated memory and using memory after freeing it is 7th. Memory leaks come in at 32nd and race conditions 33rd [CWE 2022].

In short, manual memory management (*e.g.*, C/C++) is unsafe and prone to errors but allows programmers to leverage domain knowledge to optimise memory management. Some compile-time memory management (*e.g.*, Rust) and automatic GC (*e.g.*, Java/C#) avoids the memory unsafety, but instead leads programmers to write unsafe code for a variety of reasons. In Rust, programmers must use unsafe code to construct well-known data structures, and object topologies without clear domination. In Java, programmers use unsafe code [Mastrangelo et al. 2015] to leverage domain knowledge to optimise memory management and to make GC performance more predictable. In general, reasoning about the performance of automatic GC is made difficult by the systemic effects of GC on program performance and the heuristics which control when and how GC is run.

Contributions. This paper introduces Verona's region system—Reggio—and its accompanying type system. Reggio gives programmers control over memory management costs by dividing a program's heap into a flexible forest of independent regions. Programmers can pick a suitable strategy for how memory in each region is managed, irrespective of what other regions do. Within each thread, the programmer explicitly moves a single "window of mutability" through the region forest. The single window of mutability makes clear which region each part of a program is working within, and how the program affects that region, in particular with respect to object liveness, and also permits a flexible aliasing. As a region can only be made accessible through a single pointer, programs become free of data-races by design, and cheap ownership transfer to support reconfiguring the region topology is easy. Memory management overheads (*e.g.*, tracing, and reference count manipulations) are likewise localised to just the mutable region.

2 BACKGROUND

The continuing appeal of manual memory management highlights the research problem we aim to solve: how can languages give programmers the level of control offered by manual memory management, while maintaining memory safety? Two broad research streams tackle this problem, one based on *regions* and one based on *ownership*.

Regions. Gay and Aiken [1998] introduced explicit regions for managing memory in C-like languages: objects are allocated in regions; and entire regions of objects are deleted in one operation, rather than deleting objects individually. A later version of this scheme added annotations to indicate that a pointer refers to an object in the same region, in an enclosing region, or is not allocated via the region system [Gay and Aiken 2001]. Utting [1995] had previously shown how regions could help local reasoning, based on the "collections" or "local stores" used to differentiate pointers in Euclid [Lampson et al. 1977].

Rather than using explicit, programmer-visible regions, Tofte and Talpin [2004; 1997] demonstrated how Milner-style inference [Baker 1990] could be extended to implicitly allocate objects to regions, and allocate and deallocate those regions, without either explicit first-class regions, or additional annotations on programs or types. Their MLKit [Tofte et al. 2021] runs ML programs using stack allocation, as the regions are allocated last-in, first-out. Because these inferred regions are implicit, the region structure does not capture a programmer's intent about how and where memory should be allocated and freed. MLKit remains under continuous development, in particular showing how regions can be supported in a straightforward monadic style [Fluet and Morrisett 2006], and integrating generational-style GC within regions [Elsman and Hallenberg 2021].

Safe region allocation was then tested at scale in Cyclone [Grossman et al. 2002] an extension of C with an explicit region construct, rather than using inference. Like the MLKit, Cyclone regions were originally stack based; later versions also adopted support for unique pointers and reference counted objects to permit deallocation of individual objects inside a region, at the cost or introducing memory leaks due to cycles or failure to deallocate a dropped unique pointer [Hicks et al. 2004]. Both MLKit style implicit / inferred regions, and Cyclone explicit / annotated regions can be modelled by a common core calculus based on linear references to explicit, first-class regions [Fluet et al. 2006].

Ownership. Work on object ownership effectively begins with Hogg's Islands [1991] and a general recognition of the need to control topologies of programs [Hogg et al. 1992] in languages where object identity (*i.e.*, dynamic allocation, mutable state), encapsulation, and even "automatic storage management" are taken as essential design principles, rather than accidental optimisations [Ingalls 1981; Lehrmann Madsen et al. 1993; Lieberherr and Holland 1989].

Based on "Flexible Alias Protection" [Noble et al. 1998], *ownership types* [Clarke 2001; Clarke et al. 1998] offer compile-time enforcement of pointer encapsulation, including the property that, considering paths through the object graph, an "owner" object should dominate all other objects that programmers intend to encapsulate "inside" it [Potter et al. 1998]. Leveraging "owners-as-dominators", extensions to ownership types have been applied to encapsulate object invariants [Müller and Poetzsch-Heffter 1999], record conformance to software architecture [Aldrich and Chambers 2004], localise program effects [Clarke and Drossopoulou 2002], scope object cloning [Li et al. 2012], ensure actor isolation [Clarke et al. 2008; Gruber and Boyer 2013; Srinivasan and Mycroft 2008], prevent data races [Boyapati and Rinard 2001; Gonnord et al. 2023], support safe parallelisation [Bocchino 2011; Francis-Landau et al. 2016] or ensure data is only accessed under a mediating lock [Flanagan and Freund 2000]—the first fifteen years of these efforts are surveyed in [Clarke et al. 2013]. Owners-as-dominators leads directly to applications in memory management, as deleting a dominating node from a graph, by definition, must also delete every node it dominates. This was first demonstrated in SafeJava [Boyapati et al. 2003] using ownership types to compile straightforwardly annotated programs to the Real-Time Specification for Java [Bollella et al. 2003], which supports fine-grained control over memory via explicit dynamically-scoped regions.

Distinguishing between the inside and outside of an encapsulated object lets languages generalise traditional pointer-based uniqueness to *external uniqueness* [Clarke and Wrigstad 2003], where an object may have only one pointer from the outside, but any number of pointers from its inside. As with regions for unique objects, an externally unique object can be represented as the sole object in a region; however for external uniqueness, the object's region can also contain one or more enclosed subregions in turn containing the object's insides. External uniqueness is almost as strong as regular uniqueness [Wrigstad 2006], and in particular makes it easier to change objects' ownership, or dually, to transfer objects between actors or independent threads [Clarke et al. 2008; Gordon et al. 2012; Haller and Loiko 2016; Haller and Odersky 2010].

Rust. Regions and ownership have been brought together recently in the design of Rust, combining control of memory use, safe concurrency, and excellent compiler error messages [Hu 2020; Klabnik and Nichols 2019; Krill 2021; Turon 2015]. Rust essentially inherits Cyclone's and MLKit's regions, but strongly integrated with uniqueness. In particular, only values which are uniquely referenced or passed by copy are mutable. Nested uniqueness brings nested ownership: a unique value is owned by its storage location, ensuring that when an owner is deallocated, all the memory owned by that owner can also be deallocated.

To make programming in Rust practical, Rust allows unique values to be borrowed without nullifying their source: a unique mutable reference can be passed up the stack without losing uniqueness [Boyland 2001] or traded for multiple read-only borrowed references [Wadler 1990]. To ensure absence of dangling pointers, Rust tracks lifetimes of borrowed references and ensure that a "longer-lived" (or enclosing) object can never point to a "shorter-lived" (or encapsulated) object. This borrowing semantics is reminiscent of fractional permissions [Boyland 2013]. Through uniqueness, Rust imposes a multiple-reader/single-writer concurrency model [Lea 1998].

The strict rules surrounding ownership and borrowing, and compilers' inability to accept safe programs that they "cannot understand", make Rust hard to learn and to use correctly [Abtahi and Dietz 2020; Blaser 2019; Coblenz et al. 2022; Jung et al. 2020; Qin et al. 2020; Spencer 2020]—to the point where the difficulty of implementing first-year data structures (such as doubly-linked lists) has now become an Internet trope [Beingessner 2019; Cameron 2015; Cohen 2018; ndrewxie 2019]. When faced with these problems in practice, Rust programmers either escape into unsafe Rust or revert to the birthplace of aliasing, using integer array indices, FORTRAN style [Bendersky 2021].

3 REGGIO REGIONS

In this section we describe the central concepts of the Reggio region system: regions, the region topology, operations on regions, the single window of mutability, and the properties of the region system. But first, let us overview the goals of this work.

3.1 Motivation

The design decisions in this paper are motivated by our primary goal:

(G1) *Controllable and Predictable Memory Management Costs.* It should be possible for a programmer to reason about and control the impact of memory management on performance.

Our approach is to divide a program's heap into *isolated regions* and make each region an isolated unit of memory management. Concretely, we set the following five sub-goals:

(G2) *Mix-and-Match Memory Management.* A region is free to manage its own memory however it likes, irrespective of any other regions in the program. Thus, a programmer is free to pick a memory management strategy suited to the needs of particular operations.

(G3) *Incremental Memory Management.* Performance of memory management in one region should not be affected by activities in another region. Thus, fine-grained partitioning gives finer cost-control.

(G4) *Zero-Copy Ownership Transfer.* Ownership transfer between regions must be possible without copying objects. Thus, fine-grained partitioning does not have a hidden expensive cost, and heap topology can be modified cheaply.

(G5) *Concurrent Memory Management.* Timing of memory management in one region should not be contingent by activities in another region. Thus, a programmer can initiate an operation—memory management or not—without having to wait, or forcing a wait upon any other part of the program.

(G6) *Safe Concurrency.* A thread that has access to a datum may access it freely without any need for synchronisation, and with a guarantee of data-race freedom.

Because this paper does not deal explicitly with concurrency, we will refrain from discussing the last two goals until §8.1.

3.2 High-Level Overview

We distinguish between mutable and immutable objects. In this paper, we are mostly concerned with the former. Immutable objects do not live in regions, and can be accessed freely in a program. In contrast, every mutable object belongs to a particular region. In certain circumstances, mutable objects may be made *temporarily immutable*. To avoid confusion, we will sometimes use the phrase *permanently immutable* to denote objects whose mutability is irretrievably lost.

3.2.1 Regions and Region Topology. A region is a set of objects whose memory is managed together. At any moment, one of these objects is designated as the *bridge* object. A region can be *opened* or *closed*. Closed regions are isolated from the rest of the program which means that with the exception of the bridge object, objects in a closed region are only referenced from within the region. Bridge objects are externally unique [Clarke and Wrigstad 2003] so they may have an additional, single external incoming reference.

The only outgoing references from objects in a region are either to immutable objects or to bridge objects of other (nested) regions. Thus, a program's region topology forms a forest, and moving the external reference to the bridge object changes the topology. The topology of references *within* regions is unrestricted: any object can point to any other within the same region.

Fig. 1 illustrates the isolation of the region R (bean-shaped boundary). Object a is the current bridge object of the region, denoted by drawing it on the boundary. Also c and e could serve as the bridge object. References $e \longrightarrow n$ and $m \longrightarrow e$ are not permitted as they break isolation. The reference from $a \longrightarrow i$ is permitted as i is immutable. The reference $m \longrightarrow a$ is permitted because a is the current bridge object. Immutable objects do not live in regions and may only refer to other immutable objects. Therefore, the reference $o \longrightarrow a$ is not allowed. Last, bridge objects may only have one incoming reference from outside the region, so no more references to a are allowed from outside of R, regardless of their origin.

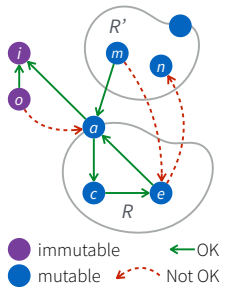

Fig. 1. Region isolation of two closed regions.

3.2.2 Regions and Memory Management. Every region manages the memory of its objects in isolation, and according to a strategy picked by the programmer specifically for that region at the time of its creation. Code inside of a region is agnostic to how memory is managed, meaning that a library can leave such decisions to its users.

When an external reference to a bridge object is dropped, the entire corresponding region can be free'd along with any nested regions. In Fig. 1, dropping $m \longrightarrow a$ makes all objects in region R unreachable as external references to its objects (*e.g.*, $m \longrightarrow e$) are not permitted. Thus, they can be collected immediately.

3.2.3 Single Window of Mutability. A region must be explicitly opened to be accessed, and must be closed before it can be opened again. The open regions form a LIFO stack. The top region is *active* and the remaining regions on the stack are *suspended*. An active region permits allocation, deallocation and mutation of its objects. When a region is suspended, neither allocation, deallocation nor mutation is permitted in it.

Table 1. Allowed actions depending on a region's state. Incoming and outgoing denote references to mutable objects from and to other regions respectively. Bridge means only to bridge objects; any* means to any object of a previously (outgoing) or subsequently (incoming) opened region. Free object denotes ability to free individual objects inside of a region. Free region denotes the ability to free an entire region.

	Encapsulation		Effects		Memory Management			Nested
State	Incoming	Outgoing	Mutate	Read	Alloc object	Free object	Free region	Regions
active	bridge	any*	yes	yes	yes	yes	no	yes
suspended	any*	any*	no	yes	no	no	no	yes
closed	bridge	bridge	no	no	no	no	yes	yes

Making a region active temporarily weakens its *outgoing* encapsulation: its objects are permitted to reference objects in the suspended regions further down the stack. Suspending a region conversely weakens its *incoming* encapsulation: its objects can be referenced by the regions further up the stack. Table 1 overviews the allowed actions depending on a whether a region is active, suspended or closed. When the active region is closed, it is popped from the stack, and the new top region goes from suspended to active. Because closed regions are not permitted outgoing refereces, any references to objects in open regions must be invalidated.

To the active region, the suspended regions appear as a single *temporarily immutable* region whose objects can be referenced as long as the active region remains on the stack. Programmers can thus *trade mutability for access, and any object can be temporarily accessed from anywhere, provided the containing regions are opened on the stack in a permitting order.*

In Fig. 1, to open R we must first open R' to access the reference $m \longrightarrow a$. Opening R through this reference suspends R', making m and n temporarily immutable, a, c, and e mutable, and permitting $e \longrightarrow n$. With the topology of Fig. 1, we cannot open R and R' in a way that permits the creation of $m \longrightarrow e$ as m is immutable when R' is suspended, and e is not accessible when R is closed. To do so, we must change the topology by moving $m \longrightarrow a$ out of R', *e.g.*, to a stack variable or other region.

In combination, the design decision to only permit mutation in one region at a time, the LIFO order of the region stack and the inaccessibility of closed regions facilitates reasoning about side-effects. The main motivation, however, is to control the costs of memory management. As we shall see, direct overheads related to memory management such as maintaining reference counts or tracing object structures are only applicable to active regions.

3.2.4 Navigating Through the Region Forest. Due to the single window of mutability, programs require explicit navigation through the region forest. The left subfigure of Fig. 2 shows Fig. 1, denoting regions' states by colour when R' is active and R is closed. The white box denotes the stack frame of R' with its local variable(s). If we proceed by opening R we arrive at the right subfigure of Fig. 2: a new top frame is created inside R containing its own local variables, with y holding a reference to a; R' is suspended and R active, and the window of mutability is moved from R' to R. The reference $z \longrightarrow m$ shows the weakened isolation allowing outgoing references from R and incoming references to R'. To continue, we may close R or open any reachable region R''. The former will invalidate any references from R to R' since these would violate isolation. The latter gives mutable access to R'' and suspends R, and permits references from R'' to both R and R'.

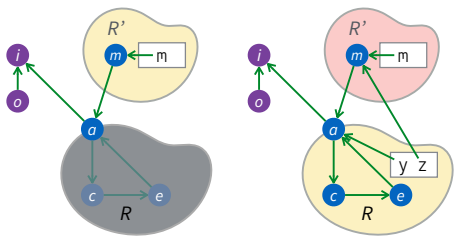

Fig. 2. Left: R' is active and R is closed. Right: we opened R making it active and suspending R'.

3.2.5 Swapping the Bridge Object. Any object in a region can serve as the bridge object. While a region is open, any object can be designated as the new bridge object, and we make this choice visible when the region closes. Thus, it is possible to *e.g.*, create a region with a stack where the bridge object is always the top link. If we wish to create an external iterator to the stack, we can create an iterator inside the region, and make the iterator the bridge object for the duration of the iteration, and then switch back again. Fig. 3 shows the situation pictorially.

3.2.6 Merging and Freezing Regions. A closed region's contents may be merged into another region by dropping the uniqueness of the bridge object. For clarity, we use an explicit merge operation. Fig. 4 shows merging R' into R, which moves the objects in R' into R and creating the (now legal) $a \longrightarrow b$ reference from the variable `x`, after which R' ceases to exist. Merging a region (source) into another (sink) *moves* all regions directly nested in the source to the sink, but does not *merge* those regions into the sink (see R'' in Fig. 4).

Permanently immutable structures are created by constructing a mutable region and then turning its entire contents immutable through an explicit freeze operation that operates on closed regions. In contrast to merging, freezing a region also freezes its nested regions (see Fig. 4). This preserves the property that immutable objects may only reference other immutable objects. Freezing dissolves region boundaries, making the frozen objects freely accessible to objects in all regions.

4 REFERENCE CAPABILITIES FOR STATICALLY ENFORCING REGION ISOLATION

We now introduce a type system that statically enforces region isolation according to the *encapsulation* and *effects* columns of Table 1. A region is opened through an **`enter`** operation that takes a reference to bridge object and a lambda, executes the lambda inside the region passing it the bridge object as argument, and then closes the region. Its companion operation **`explore`** opens a region in a suspended state (while still suspending the former active region).

Our type system uses reference capabilities which decorate all types τ in a program:

- mut τ denotes an intra-regional reference to an object of type τ (`r`, `c`, `e`, and `m` in Listing 1);
- tmp τ is like mut τ, but the lifetime of the object is bound to the **`enter`**/**`explore`** scope;
- imm τ denotes a reference to a permanently immutable object of type τ (`i` and `o`);
- iso τ denotes an externally unique reference to a bridge object of type τ of a closed region (`a`);
- paused τ denotes a reference to an immutable object in a suspended region (`z` in Fig. 2).

On assignment, the LHS capability and the RHS capability must be the same. **`merge`** has the signature iso→mut and **`freeze`** has the signature iso→imm. To **`enter`** or **`explore`**, an iso is needed.

All expressions are typed from the point of view of the currently active region. A field `f` declared with a mut capability will only appear as such when the object o containing `f` is in the active region. If o's containing region is suspended, `f` will also appear as suspended; if o's containing region is closed, `f` is not even visible to the program. This is handled by viewpoint adaptation (*c.f.*, §4.1).

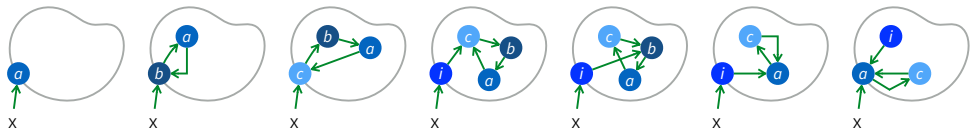

Fig. 3. Examples of bridge swapping. Time moves left; we use different shades of blue for clarity. Subfigures 1–3 construct the cyclic linked list $[a, b, c]$, using the most recently added link as the bridge object. Subfigures 4–6 construct an iterator, for the list, and make it the bridge object. We subsequently use the iterator to iterate to the b link and unlink it, before dropping the iterator and making a the bridge object (and with a garbage iterator whose removal is determined by the region's memory management strategy).

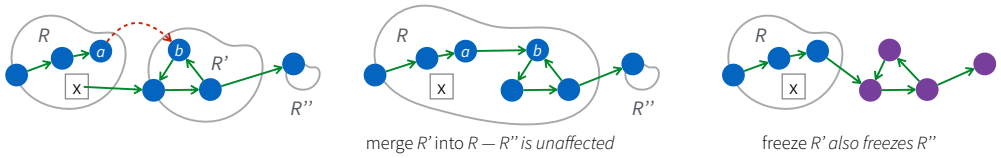

Fig. 4. Left: Three nested regions R, R', and R''. Centre: Merging region R' into R. Right: Freezing R'.

An invariant in our system is that aliases to an object that are *accessible simultaneously* have the same capability. This design is motivated in-part by region isolation and in-part by a desire to keep complexity down in this presentation. For example: If a mut and paused could alias, the immutability of the latter would be weakened to read-only. If paused and imm could alias, it would violate the temporary vs. permanent nature of their immutability. (This restriction could be relaxed to permit some aliasing across the mutable capabilities and aliasing across immutable capabilities.)

Constructing Fig. 1. Listing 1 shows code that creates the regions, objects, and (permitted) references of Fig. 1. Line 4 creates the R region. On Line 2, the immutable object i is constructed by creating a region and freezing it. The object o is created similarly. Its reference to i does not break region isolation as i is immutable. We could get rid of the **freeze** on Line 2 since Line 3 moves i into o's region and then freezes the entire structure.

```
// Freeze creates immutable objects, c.f. §3.2.6          1
let i = freeze new iso Cell(42)                           2
let o = freeze new iso Cell(i)                            3
let a = new iso Link // creates R                         4
// { r ⇒ ... } is a lambda with argument r                5
enter a { r ⇒                                             6
 let c = new mut Link
                 let e = new mut Link                     7
                 r.elem := i                              8
                 r.next := c                              9
                 c.next := e                              10
                 e.next := r }                            11
let m = new mut Cell(a) // buries a                       12
```

Listing 1. Code creating the region R from R' as depicted in Fig. 1.

When a region is created, it is closed and empty, except for its bridge object. To populate R as in Fig. 1, it must first be made active. The **enter** keyword is used to open a region and making it active. It takes a unique reference to a bridge object as its argument. Lines 6–11 of Listing 1 show the use of an **enter** block to open the R region to allocate and mutate its objects. (The code executes with region R active and region R' suspended.)

Entering a region moves control inside it and places a mut reference to its bridge object in a variable on the stack (r) that can be used to call methods, or obtain and store references to other objects in the region. Exiting the **enter** block (after Line 11) closes the region, and moves control back to the previous region. While a region is open, the external reference to its bridge object is buried [Boyland 2001], meaning it is not accessible to the program.

Upon exiting the **enter** block, all variables referencing objects in the now-closed region (c, e, and r) are invalidated, save for the reference to the bridge object in a. Any temporarily permitted references to objects in suspended regions, such as $e \longrightarrow n$ in Fig. 1, will be invalidated as well. (We will show how this is enforced statically in §4.3.)

4.1 Controlling Effects through Viewpoint Adaptation

We rely on viewpoint adaptation [Dietl et al. 2007] to capture how a reference's type changes depending on its enclosing region's relation to the active region. Viewpoint adaptation means that the type of an object may appear differently depending on from where it is accessed. For example, when accessed through a variable of type imm, a field declared with type mut appears as imm. (This particular case ensures that turning a unique reference to a bridge object immutable will propagate the immutability to the entire region and nested regions.)

Viewpoint adaptation also changes the types of variables captured by an **enter** or **explore** block to propagate suspension. Captured isos retain their iso-ness rather than become paused. Table 2 shows the viewpoint adaptation rules. The meaning of ⊥/iso is that an iso location is inaccessible through a mut, tmp or paused, unless it is *swapped*, *buried* or *borrowed* (*c.f.*, §4.2).

Table 2. Viewpoint adaptation. If the capabilities of x and f are α and β, then the capability of x.f is $\alpha \odot \beta = \gamma$, which we read as "$\alpha$ sees β as γ." For †, *c.f.*, §4.3. The meaning of ⊥ is inaccessible; For ⊥/iso see text.

Capability on x	Capability on f				
	mut	tmp	imm	iso	paused
mut	mut	⊥†	imm	⊥/iso	⊥†
tmp	mut	tmp	imm	⊥/iso	paused
imm	imm	imm	imm	imm	imm
iso	⊥	⊥	⊥	⊥	⊥
paused	paused	paused	imm	⊥/iso	paused

To illustrate viewpoint adaptation, consider $\mathcal{A}$ **enter** x { y ⇒ $\mathcal{B}$ }. In scope $\mathcal{A}$, let the variables x and v have the types iso τ_1 and k τ_2 respectively. In scope $\mathcal{B}$, x is undefined and y is introduced with type mut τ_1. This reflects the region pointed to by x going from closed (denoted by x being iso) to active (denoted by y being mut). Moving control from $\mathcal{A}$ to $\mathcal{B}$ suspends the region active in $\mathcal{A}$ (denoted there by mut and tmp). Thus, in scope $\mathcal{B}$, the type of v is (paused $\odot$ k) τ_2. Through a paused reference, objects in the same region are paused. paused and imm references stay paused and imm respectively (permanently immutable is stronger than temporarily immutable). iso references stay iso. This permits opening nested regions of a suspended region.

If k = iso, then v.f is not typeable in neither $\mathcal{A}$ nor $\mathcal{B}$ as iso $\odot$ k' = ⊥, regardless of what k' (the capability of f) is. This is as expected, as iso's cannot be dereferenced (they must be **enter**'d).

4.2 Region Isolation and Bridge Object External Uniqueness

Regions which are closed or active only have a single incoming reference, which goes to the bridge object. Thus, when a region is closed, it can be moved in and out of other regions by moving its single incoming reference. When a region is opened, its containing region is suspended, which means that the object containing field holding the reference to the bridge object is paused so the field cannot be reassigned. Thus, regions cannot move while open. Finally, when a region is suspended, incoming references are permitted from the stack and heap of subsequently opened regions (*c.f.*, §4.3). As regions are opened using a lexically scoped construct (**enter** or **explore**), regions are opened and closed in LIFO order. This means that when a suspended region becomes active again, the permitted incoming aliases that could be declared in the block have gone out of scope, and the region's bridge object is again the only incoming alias.

As shown in Table 3 uniqueness of bridge object references is maintained by a combination of swapping, burying, and borrowing.

Table 3. Maintaining uniqueness of bridge object references.

Swap [Harms and Weide 1991]	Reading a mutable variable containing an iso requires that its contents is replaced. For example, `y = x` is not permitted when x is iso. However, `y = x = v` is, which replaces the value of x by the value v and moves the *previous* value of x into y.
Bury [Boyland 2001]	Reading a let-bound variable with an iso invalidates the variable. For example, `foo(x, x)` is not permitted when x is iso as the second use of x cannot be typed.
Borrow [Wadler 1990]	Dereferencing an iso requires opening its region, where aliasing of the bridge object is unrestricted, and region isolation protects aliases to the bridge object from escaping.

Entering a region borrows and/or buries the variable or field referencing the bridge object. In the case of a stack variable, the variable is buried to prevent the region from being multiply opened. In

the case of a field, we instead resort to a dynamic check of the region's state. If the region is closed, it may be opened. If the region is already open, an exception is thrown.

4.3 Temporary Objects Allow References to Suspended Regions on the Heap

As exemplified already, we permit local variables to store references to objects in a suspended region (*e.g.*, z in Fig. 2). This is sound as objects in suspended regions are immutable, and because local variables in the active region are guaranteed to go out of scope before a region opened by an enclosing **enter** or **explore** becomes active (and thus mutable), as explained above.

Because objects with tmp capability are created in, and bounded by, a lexical scope, they have the same lifetimes as local variables declared therein. Therefore, we can grant the same permissions to store references to suspended regions to tmp objects. We permit accessing paused and tmp fields through a tmp, but not through a mut (as shown in Table 2). Permitting a mut object to store a suspended reference could lead to a breach of region isolation (see §4.5 for an example). Thus from $k_1 \odot k_2 = \text{tmp}$ it follows that $k_1 = \text{tmp}$.

In terms of Fig. 1, making e a temporary object allows its fields to store references with tmp capability. This permits the reference $e \longrightarrow n$ when R is active and R' is suspended. However, $c \longrightarrow e$ would no longer be permitted unless c is also tmp, etc.

4.4 Storage Locations, Strong Updates and Bridge Object Swapping

We unify the treatment of mutable variables (denoted **var** as opposed to **let**) and fields through a *storage location* abstraction (similar to a pointer to a variable or field in C).

Storage locations are typed k `Store[`k' τ`]` where k is the capability of the frame or object containing the location and k' is the capability of the value stored at the location.

We add a new capability that we call var for use in mutable local variables. var differs from tmp in that it supports strong updates. Its viewpoint adaptation rule is $\text{var} \odot k = k$ ("var sees k as k").

As shown in Listing 2, a mutable local variable x holding a τ-typed value has the type **var** `Store[`τ`]`. Storage locations are subject to the normal rules for viewpoint adaptation, so opening another region when x is already in scope will change the type of x to **paused** `Store[`τ`]`. We introduce a dereference operator * and an update operator := to access the contents of a storage location. A storage location must be mut, tmp, or var to be updated. We apply viewpoint adaptation to type the result of dereferencing. On Line 7, *x has type (paused $\odot$ mut) `Cell`, *i.e.*, **paused** `Cell`.

We support changing the bridge object of a region—including changing it for an object of a different type—by presenting the borrowed bridge object reference internally in the **enter** block as a var storage location. (Note that it is not possible to update the bridge object in an **explore** as it opens the region as suspended.)

```
var u = new iso Cell(42) // var Store[iso Cell]                1
var x = new Cell(4711) // var Store[mut Cell]                 2
enter u { y ⇒                                                 3
  // y has type var Store[mut Cell]                           4
  // x has type paused Store[mut Cell]                        5
  y := new Foo(*y) // changes bridge object('s type)          6
  y := *x // rejected: *x is paused Cell, not mut Cell        7
  x := ... // rejected: the x storage location is paused      8
} // change to u becomes visible                              9
```

Listing 2. Storage locations example.

Line 6 shows that changing the bridge object to an object of another type is possible by simply assigning to y. Strong updates of fields are not possible, and this is handled by using **tmp** `Store[...]` instead of **var** `Store[...]` to type a bridge object reference borrowed from a field as opposed to a stack variable.

4.5 Types Enforce Region Isolation and the Single Window of Mutability

Region isolation means no references into *active* or *closed* regions from other regions (modulo unique references to bridge objects) or from *immutable* objects, and no outgoing references from *closed* regions to open regions. Let's see how our types enforce this, by looking at R in Listing 1.

The newly created region R (Line 4) is isolated as iso constructors only accept iso's and imm's as arguments. Right after creation, R is *closed*, and its only external reference is iso. Since iso's cannot be the receivers of method calls or field accesses, we cannot read or write internal objects in R, so we cannot create the illegal references $m \longrightarrow e$ or $e \longrightarrow n$.

When R is *active* (Line 6–11), all previously suspended regions stay suspended (none in Listing 1), and are joined by the current region (R'). This is captured by viewpoint adaptation which changes all variables which are mut, tmp, or var in the enclosing region to paused. This prevents these variables from being updated, and reading them yields paused references. Field updates via paused or imm references are not allowed, and method calls on such references require that the method's self type matches the external view, meaning any callable method cannot perform a field update on **`self`** (or call such a method). Thus, we cannot create $m \longrightarrow e$ or $o \longrightarrow a$.

As permitted by our definition of region isolation, we may store paused references in the fields of tmp objects in R while R is active (*e.g.*, $e \longrightarrow n$ if e is created as tmp on Line 7). These references will be invalidated when R closes as tmp references can only be stored in variables local to the **`enter`** block (since mutable variables in the enclosing scope have been suspended), or in other tmp objects (*i.e.*, $a \longrightarrow c \longrightarrow e$ is an impossible path if e is tmp, as the a object is mut by definition).

If we did not invalidate references into suspended regions such as $e \longrightarrow n$, we could circumvent region encapsulation. For example, imagine closing R (without invalidating $e \longrightarrow n$), then closing R' while moving R out of R'. Then reopen R without first opening R'. Now $e \longrightarrow n$ would constitute a reference into the internals of a closed region, thereby breaking region isolation.

Inside an **`enter`** or **`explore`** block the enclosing scope is immutable. Together with region isolation this gives that only one region at a time is mutable, *i.e.*, a "single window of mutability".

Last, region isolation means that reassigning the pointer to the externally unique bridge object effectively changes the region topology of the heap (G4).

5 THE USE OF REGIONS FOR MANAGING LIVENESS

We have sketched how our type system enforces region isolation and the single window of mutability. In this section, we will show how this translates to costs for managing liveness when considering memory management in isolated regions.

Fig. 5 shows a heap consisting of regions R_1 to R_6. Presently, the program has entered R_1, R_2, R_3 and R_4 in that order. Ignoring method call indirections, we can imagine a corresponding program shape starting in R_1: $\mathcal{R}_1$ **`enter`** `x { d` $\Rightarrow$ $\mathcal{R}_2$ **`enter`** `e.f { f` $\Rightarrow$ $\mathcal{R}_3$ **`enter`** `e.g { g` $\Rightarrow$ $\mathcal{R}_4$ `} } }`.

The region stack is thus $[R_4, R_3, R_2, R_1]$ where R_4 is active. Region isolation prevents references from objects in region i to objects in region j if $i < j$ with the exception of the bridge object references: $x \longrightarrow d$, $e \longrightarrow f$, and $e \longrightarrow g$. The first of these is made inaccessible right after the first **`enter`** by making x undefined in $\mathcal{R}_2$ and nested scopes. While R_4 is open, R_2 and R_3 are suspended, meaning the fields holding $e \longrightarrow f$ and $e \longrightarrow g$ cannot be reassigned (static check). Furthermore, they cannot be dereferenced (*i.e.*, opened) since the regions are already open (dynamic check). Thus, while R_4 is open, the incoming references to the bridge objects will remain the same, *i.e.*, the path that holds the region alive is *stable*.[1]

Because of this, as long as R_4 remains active, liveness of objects in regions R_1–R_3 and R_5 is *invariant* as no activity in R_4 can cause objects in these regions to become garbage.[2] This does not hold for R_6, as R_4 could drop $h \longrightarrow i$ to make the entire region R_6 garbage. It does hold for $b \longrightarrow c$ however, since b is in a suspended region (b is temporarily immutable).

[1]Since an **`enter`** block can change the bridge object on exit, the incoming reference does not affect liveness. We can think of the incoming bridge object reference as being *invisibly nullified* during the **`enter`**, and reinstated at the exit.

[2]Every reference in R_4 to an object in R_1–R_3 is a copy of a reference inside one of those (immutable) regions that still exists.

Furthermore, because of the absence of references from R_1–R_3 into R_4—with the exceptions of the stable bridge object references that are either buried or cannot be re-opened, objects in R_1–R_3 *cannot affect the liveness of objects in* R_4 (this is also true for R_5 and R_6 as they are closed). Thus, we can safely ignore references to objects in suspended regions when managing liveness. For example, if objects in R_1, R_2 or R_3 are managed by reference counting, we do not need to increment or decrement reference counts when manipulating paused references in R_4. Similarly, if objects in R_1, R_2 or R_3 are managed by a tracing GC, tracing in R_4 does not need to follow paused references. Thus, when managing memory in the active region, we can safely ignore any outgoing references, and so it is possible for R_1–R_3 to use different strategies, unbeknownst to R_4 and irrespective of how R_4 manages its memory (G2). The only references to objects in R_5 and R_6 that are possible (given that the regions are closed) are to the bridge objects c and i. Aliases of c are not possible in R_4 as iso references are unique, and we cannot transfer references out of an immutable b. However, we do need to track liveness of the reference to i, which is possible to do statically *e.g.*, as in Rust.

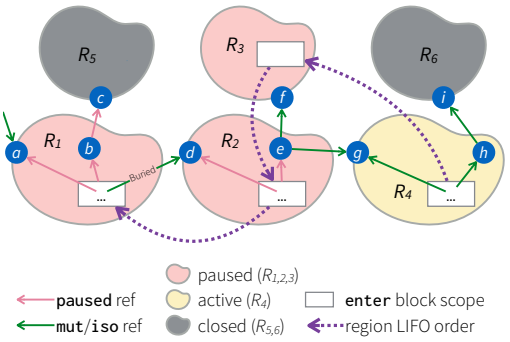

Fig. 5. Program with 4 open and 2 closed regions.

From the reasoning above it follows that liveness of objects in R_4 can be determined by looking at objects in R_4 alone, meaning that the costs of memory management are determined by the contents of and activity in R_4 (G1). This makes it possible to collect garbage in just R_4 (G3).

6 PROGRAMMING WITH REGGIO REGIONS

As an example of how regions enable predictable memory management performance, consider the server application "*Po*" with the following key characteristics (see Listing 3 for skeleton code):

(1) The server serves incoming requests. Tasks that process requests are short-lived and their side-effects are typically in the form of data stored in a database.
(2) Responses to requests are served from data in an in-memory key–value store implemented as a skip list. This storage will shrink and grow continuously during execution.
(3) Values in the store can have a complicated graph-like structure (*e.g.*, they may contain cycles).

We now explain how we can express this, and compare with Cyclone, MLKit, Pony, and Rust.

Processing Requests. To manage allocations necessary to process a request, each request is wrapped in a region (Line 16, Line 49). These regions use *arena allocation*: allocations in the region persist until the region itself is deallocated. This gives cheap bump-pointer allocation and fast deallocation of the entire region once the response has been computed. If a request must be processed by different threads, this can be done cheaply due to the transferability of iso's. If some requests turn out to require considerable processing time, their corresponding regions might switch away from arena allocation at the small cost of changing one annotation at the creation site of the region(s).

Comparison. Cyclone's LIFO regions are perfect for this purpose (as are MLKit's, provided that the inference engine infers according to intentions, or better). While Cyclone does not support switching from arena allocation, it does permit manually managed reference counts or unique objects that can be manually deallocated during the arena's lifetime. Pony only supports memory management on a per-actor level, (using tracing GC). We could make each request an actor, but

```
1  type_alias KV = Skiplist[imm Id, iso Value] // To shorten code horisontally for this presentation
2  type_alias Response = UpdateOK | InsertOK | DeleteOK | Failure
3
4  def start_po(fn : imm String, server_socket : iso ServerSocket) : Unit {
5    let kv : iso KV = new iso<RC> KV // create empty key-value store; <RC>=reference counting, see §8.3
6    enter kv { list => // Populate key-value store from persistent storage
7      let data : tmp File = open(fn, "r")
8      ... list.insert(...) ... // read contents from data and add to list
9    } // data goes out of scope, so get's free'd and closed
10
11   enter server_socket { ss =>
12     while (ss.is_open()) {
13       let socket : mut Socket = ss.accept() // new connection
14       let raw_request : imm String = socket.read_request() // get incoming request
15
16       let r : iso Response = enter new iso<Arena> Unit { _ => // arena-managed region for tmp allocations
17         let work : mut List[mut Tasks] = RequestParser::parse(raw_request) // parse request
18         let response = new mut List[mut Response] // holds all responses to tasks in request
19
20         while (!work.empty()) {
21           response.append(merge match work.pop() { // merge iso result of match into r since append expects mut
22             case mut StopTask => return // stop service, no response to client
23             case mut UpdateTask(id) => explore kv { kv' => update(kv', id) }
24             case mut InsertTask(id, payload) => enter kv { kv' => insert(kv', id, payload) }
25             case mut DeleteTask(id) => enter kv { kv' => delete(kv', id) }
26           })
27         }
28         response.accumulate(new iso Message) // chained through accumuator, moves out of arena
29       } // arena effectively goes out of scope, allocs on line 17, 18 + any tmp objects are freed
30       socket.respond(r); // render response object
31 }}}
32
33 def update(kv : paused KV, id : imm Id) : iso Response // process UpdateTasks, inside suspended kv
34   let value : paused Store[iso Value] = kv.get(id) // reference to a link's reference to a Value
35   enter *value { v => // Requires a dynamic check - because of aliasing cannot rule out v is already open
36     v.add_log_entry() // adds surviving object to v's region
37     v.remove_some_token() // makes object in v's region garbage
38     return new iso UpdateOK(v) // moves out of *value and kv regions and merged into r on line 21
39 }}
40
41 def insert(kv : mut KV, id : imm Id, p : imm Payload) : iso Response // process InsertTasks, inside kv
42   let new_value = Factory::create(id, p) // decides memory management for new_value dynamically
43   enter new_value { v => v.tokens = new mut List; v.log = new mut List }
44   kv.insert(id, new_value) // buries new_value
45   return new iso InsertOK() // moves out of kv region and merged into r on line 21
46 }
47
48 def delete(kv : mut KV, id : imm Id) : iso Response // process DeleteTasks, inside kv
49   enter new iso<Arena> Unit { _ => // create new tmp region, the one created on Line 16 is not accessible here
50     let q : mut SQL_Query = Factory::make_can_delete_query(id) // q cannot refer to kv because it is mut
51     let r : tmp SQL_Result = Backend::execute_query(q) // because it is tmp, r could refer to kv if it needed to
52     if (!r.OK) return new iso Failure(...) // moves out of tmp and kv regions and merged into r on line 21
53   } // arena goes out of scope, allocs on lines 50, 51 are free'd
54   return enter kv.remove(id) { v => new iso DeleteOK(v) } // Creates garbage in kv, drops a Value region
55 }
```

Listing 3. Skeleton code for *Po*. To save space, we permit constructor arguments to **iso** objects to take **mut** arguments (Lines 38, 45, 52, and 54). This is safe and can desugar to an extra **enter**. These lines all create an object that escapes the active region and is merged into the enclosing region, making them **mut** in `r` (`append` expects a **mut** argument). The `ServerSocket` is created elsewhere. It likely does not use arena allocation since it allocates on each turn of the loop (Line 13). If it did, those allocations would only be free'd on Line 31. The default annotation on **new iso** is `<Arena>` (*c.f.*, §8.3). Notice how code is agnostic to how memory is managed.

it would have to communicate asynchronously with all surrounding state. Rust does not support regions, but could *e.g.*, build a unique object holding unique values and thread this object through computation. While more complicated, Rust's values would have individual lifetimes.

Key–Value Store. The key–value store is implemented as a single region containing a skip list (Line 5, could as well be a hash table, B-tree, etc.). As it is a large long-living data structure of objects with different lifetimes, arena allocation is not a suitable strategy. Furthermore, if the resources (values) stored in the skip list are costly, *reference counting* is a good choice as it allows the resources in the list to be recycled immediately when they become garbage. Alternatively (the path we chose) is to make each element a region of its own, with independently managed memory (Line 1).

If the store is large enough to warrant parallel accesses, it can be divided into several smaller regions with one list each. For a compile-time guarantee that no reference count manipulations in the skip list are needed during lookups, lookups can be implemented using **`explore`** rather than **`enter`** as the former opens the skip list region directly as paused (Line 23, and its dynamic extent `foo()`). Note that since the elements in the skip list are regions of their own, these can be entered separately and thus mutated, even if the list structure is immutable (Line 35).

Comparison. Cyclone's dynamic regions are a good fit for the key–value store. The objects that make up the skip list would require manual reference counting, which can be laborious. Failure to properly manage reference counts in Cyclone will also lead to "memory leaks" which will not be reclaimed until the region holding the store is (manually) destroyed. Pony can wrap the skip list inside an actor with an asynchronous interface, and manage its memory using tracing of the entire list leading to more time spent tracing memory and more floating garbage. Skip lists (hash tables, B-trees, etc.) cannot be constructed in Rust without judicious use of `unsafe`. Safe Rust's reference counting does not relax its ownership rules, so mutation of aliased values is not permitted.

Values in the Store. Finally, the elements in the store are suitable for either reference counting or tracing GC because of their graph-like structure (Line 42 delegates this decision to a factory method). GC is especially favoured in the (possible) presence of cycles which are expensive to detect with reference counting [Jones et al. 2016].

Comparison. Later versions of Cyclone and MLKit support a global arena where memory is managed using tracing GC. Thus, all elements in the store contribute to pressure on the same GC, and GC requires tracing through all elements to free garbage objects in one element. MLKit's region propagation requires all elements to be put in a single region. Pony can handle this pattern by making each element an actor, which makes each element aliasable, and use an asynchronous interface. Finally, Rust will not be able to express and manage lifetimes of these structures automatically. A combination of `unsafe` and manual memory management is needed.

Design Thoughts on Explore vs. Enter—And Invariants. The **`explore`** construct is essentially syntactic sugar for two nested **`enter`** blocks, the outermost entering the region to be explored and the innermost entering a fresh region:

`explore` `x { y => ... }` *desugars to* **`enter`** `x { y => {` **`enter`** `(`**`new iso`** `Unit) { _ => ... } }`

The first **`enter`** activates the region, and the second suspends it. The new region (**`new iso`** `Unit`) is independent from the rest of the program. As it is active, it serves all allocations that appear inside the **`explore`** block (as suspended regions do not permit allocation or deallocation, Table 1). What **`explore`** guarantees that unprincipled nesting of **`enter`**s does not, is that the explored region was not mutated before suspended. Conceptually, this is a big difference as we will explain next.

Similar to object invariants, we expect invariants of a closed region to hold at the time of opening. While active, invariants may temporarily be broken and then restablished before the region is closed.

As a nested **enter** can reference any enclosing region, it will be able to observe any invariants broken by mutation following the opening of the enclosing regions. By opening regions directly in a suspended state, **explore** ensures that region invariants continue to hold. We are considering using a separate capability to capture this statically. We are also considering an "eager" explore construct that opens a region along with all its subregions as suspended in one fell swoop. This would avoid the need for explicit opening of subregions thus further simplifying working with immutable objects. The cost is more complexity in the type system. Time will tell whether this complexity is warranted or not.

With respect to memory management, **explore** allows opening a region for reading, and navigating through it, without any memory management overhead as the region's object structure is invariant and there are neither allocations nor deallocations in the region.

Navigating Regions. Listing 4 shows a zip computation involving three unrelated regions. Using **explore**, we open the staff and reviews regions to make them temporarily immutable and their contents accessible. Finally, we open the zip region using enter. This makes the region active which allows allocation of the two iterators on Lines 8 and 9 and any allocation needed by the call to append on Line 11 to extend the list. It also allows the mutation in the `next()` calls to advance the iterators, and the mutation necessary to add the new pair to the zip list. Allocating the iterators inside the same region as their corresponding list is not useful as it would make the iterators immutable on Lines 11 and 12. This would cause the program to not typecheck—as the `next()` method needs to update the iterator, it needs to be called on a mut or tmp receiver. Since the iterators need to store paused references, they must be tmp (*c.f.*, Table 2). This can be handled in `iterator()` by overloading on the self type, letting the implementation with paused self-type return a tmp reference. Elements are immutable so *e.g.*, `next()` returns imm.

```
let staff : iso List[imm Employee] = ...                          1
let reviews : iso List[imm Review] = ...                          2
var zip = new iso List[imm (String, Int)]                         3
                                                                  4
explore staff { s => // open as immutable                         5
  explore reviews { r => // open as immutable                     6
    enter zip { z => // open as mutable                           7
      let si = s.iterator() // si is tmp                          8
      let ri = r.iterator() // ri is tmp                          9
      while (si.has_more() && ri.has_more()) {                    10
        z.append(new imm Pair(si.next().name(),                   11
          ri.next().calculate_salary()))                          12
      }}}}                                                        13
```

Listing 4. Opening two non-nested regions, computing a result in an active region. [ ] is type parameters.

Reggio's Borrowing Capabilities. Traditional borrowing as originally introduced by Wadler [1990], explored deeply by *e.g.*, Boyland [2001] and Boyland et al. [2001], and popularised by Rust relaxes uniqueness of a value in a well-defined lexical scope. We can express a similar form of borrowing through the type **paused** `Store[iso T]`, *i.e.*, a reference to a storage location in a suspended region storing a reference to a closed region. Such a reference (*e.g.*, `x`) can be shared freely inside a single thread, allowing it to flow to a place where it can be opened (**enter** `*x`) *with* mutation rights, including swapping the bridge object (as long as the new bridge object is a subtype of `T`).

```
var x = new iso Cell // x : var Store[iso Cell]
enter y { => _ // now x : paused Store[iso Cell]
  enter *x { => z
    ... // Can still mutate z!
  }
}
```

7 FORMALISING REGGIO

We formalise Reggio through two interacting languages: *region* and *command*, and their respective semantics. The former controls all accesses to memory (loads and stores), allocation of objects in regions, creating, merging, freezing—and importantly entering and exiting—regions. The most important properties of the region language is expressed as a *topology invariant* (*c.f.*, §7.5). The

command language is essentially "what the programmer wrote". This separation makes it possible to specify *e.g.*, under what conditions a store is valid, irrespective of what caused the store.

During execution, the command language emits effects which the region language performs. The static semantics of the command language ensures, modulo one dynamic check, that the topology invariant is preserved. We present most of the rules of the region language and key type system rules. Additional details are available in a technical report [Arvidsson et al. 2023].

7.1 Dynamic Semantics of the Region Language

A configuration in the region language has four components: a LIFO region stack RS and the sub-heaps of open regions H_{op}, closed regions H_{cl}, and frozen regions H_{fr}. The latter is an unimportant simplification (conceptually, only mutable objects live in regions). A heap H is a collection of disjoint regions R. Opening a closed region moves it from H_{cl} to H_{op} and pushes a new stack frame on top of RS. Closing the top-most region in RS returns it back to H_{cl}. Freezing a region moves it, and all regions reachable from it, permanently to H_{fr}. As an example, consider the region topology depicted in Fig. 5. We can write down the corresponding configuration as $\langle RS, H_{op}, H_{cl}, H_{fr}\rangle$ where

$$\begin{array}{rcl} RS & = & RF_4 :: RF_3 :: RF_2 :: RF_1 :: \epsilon \\ H_{op} & = & R_1 * R_2 * R_3 * R_4 \\ H_{cl} & = & R_5 * R_6 \\ H_{fr} & = & \epsilon \end{array}$$

$$\begin{array}{rcl} rcfg & ::= & \langle RS; H; H; H\rangle \\ RS & ::= & RF :: RS \mid \epsilon \\ RF & ::= & (r, S, F) \\ H & ::= & R \mid H * H \mid \epsilon \end{array}$$

Fig. 6. Configuration in region language (1/2)

For the region sub-heap R_i, if R_i is open (*i.e.*, part of H_{op}), RF_i is its region frame (depicted in Fig. 5 as a white box) that holds the stack variables created in the scope of the corresponding **enter** block. Opening the closed region R_6 would push a new frame RF_6 above RF_4 in RS, and move R_6 from H_{cl} to H_{op}. Similarly we could freeze (merge) R_6, which would move it from H_{cl} to H_{fr} (remove it from H_{cl} and merge it into R_4).

In this model, an inter-region reference into an open region is permissible iff it points downwards in the region stack (from left to right according to RS), or it is the unique (iso) reference through which the region was opened. The LIFO region stack constitutes a "path" through the region forest that corresponds to the opening order of **enter** blocks (*c.f.*, the region LIFO order in Fig. 5). Thus, in Fig. 5, any reference from R_4 to R_3 is permissible (as long as we do not close R_4), while a reference from R_2 to R_3 must necessarily be the reference from object e to object f. We model **explore** as nested **enter**s.

$$\begin{array}{rcl} S & ::= & \iota \mapsto o, S \mid \epsilon \\ F & ::= & f \mapsto v?, F \mid \epsilon \\ v? & ::= & v \mid \textbf{undef} \\ v & ::= & (k, \iota) \\ R & ::= & (r, S) \\ o & ::= & (\#CL, F) \end{array}$$

Fig. 7. Configuration in region language (2/2)

A region stack frame RF contains a region identifier r, a temporary store S for objects whose lifetimes are bounded by the scope of the region's **enter** block (values with capability tmp), and a map F from variable names f to values v, representing the local variables in that **enter** block (we model destructive reads of a variable x by remapping it to **undef**, at which point reading x again will lead to the program getting stuck). A region R is a tuple of a (unique) region identifier r and a store S containing the objects in that region.

Objects are identified by ι. Values v are object identifiers ι tagged with a capability k. Stores S map object ids ι to objects o which store their class tag $\#CL$ and fields (for simplicity we reuse the same F as for local variables, although a field will never contain **undef**).

The command language communicates with the region language via effects. The relation $rcfg \xrightarrow{Eff} rcfg'$ should be understood as performing the effect Eff in $rcfg$, resulting in $rcfg'$. Effects include entering and exiting a region, loading a value from an object store, writing (swapping) a

value for another in an object store, merging or freezing a region, etc. We now describe a selection of rules for these effects.

REGION-LOAD

$$\frac{\begin{array}{c} x\ \mathit{fresh} \qquad F(y) = (k, \iota) \\ \mathsf{cfg\_load}((r, S, F) :: RS, H_{op} * H_{fr}, \iota) = o[f \mapsto v] \\ F' = F, x \mapsto (k \odot v) \end{array}}{\langle (r, S, F) :: RS; H_{op}; H_{cl}; H_{fr} \rangle \xrightarrow{\mathit{load}\,(x,y.f)} \langle (r, S, F') :: RS; H_{op}; H_{cl}; H_{fr} \rangle}$$

In every effect, the first parameter should be understood as the name of the variable where the results should be stored. For example, the effect $\mathit{load}(x, y.f)$ is handled in rule REGION-LOAD by binding the value of field $y.f$ to variable x. First, the value of y is looked up in the top stack frame as (k, ι). The object id ι is used to find the corresponding object o in the configuration—it may be stored in the subheap of an open or frozen region, or in one of the temporary stores on the region stack. The capability k is used for viewpoint adaptation of the (capability of the) value v of field f in o before it is inserted into the top frame.

REGION-SWAP-TEMP

$$\frac{\begin{array}{c} x\ \mathit{fresh} \qquad \mathbf{get}\,(F, \mathit{use}) = (v, F') \qquad F'(y) = (k, \iota) \qquad \mathsf{load}(S, \iota) = o[f \mapsto v'] \\ o' = o[f \mapsto v] \qquad \mathsf{store}(S, \iota, o') = S' \qquad F'' = F', x \mapsto v' \end{array}}{\langle (r, S, F) :: RS; H_{op}; H_{cl}; H_{fr} \rangle \xrightarrow{\mathit{swap}\,(x,y.f,\mathit{use})} \langle (r, S', F'') :: RS; H_{op}; H_{cl}; H_{fr} \rangle}$$

REGION-SWAP-HEAP

$$\frac{\begin{array}{c} x\ \mathit{fresh} \qquad \mathbf{get}\,(F, \mathit{use}) = (v, F') \qquad F'(y) = (k, \iota) \qquad \mathsf{load}(S', \iota) = o[f \mapsto v'] \\ o' = o[f \mapsto v] \qquad \mathsf{store}(S', \iota, o') = S'' \qquad F'' = F', x \mapsto v' \end{array}}{\langle (r, S, F) :: RS; (r, S') * H_{op}; H_{cl}; H_{fr} \rangle \xrightarrow{\mathit{swap}\,(x,y.f,\mathit{use})} \langle (r, S', F'') :: RS; (r, S'') * H_{op}; H_{cl}; H_{fr} \rangle}$$

Field assignments are caused by the effect $\mathit{swap}(x, y.f, \mathit{use})$, which writes the value of *use* to $y.f$ and binds the *old* value of $y.f$ to x. A *use* is a potentially destructive variable access (z or **drop** z, see Fig. 9). Rules REGION-SWAP-TEMP and REGION-SWAP-HEAP handle the cases where the object being assigned to is in the temporary store or on the heap. In both cases, we perform the *use* (which may make a variable invalid) with the helper function **get** (see Fig. 8). We then proceed just as when loading a field, but finish by updating the object being assigned to and update its containing store S. Note that assigning and loading mutable variables are special cases of the *swap* and *load* effects since we model mutable variables as single-field objects.

$$\begin{array}{l} \mathbf{get}(F[x \mapsto v], \mathbf{drop}\ x) = \\ \quad (v, F[x \mapsto \mathbf{undef}]) \\ \mathbf{get}(F[x \mapsto (k, \iota)], x) = \\ \quad ((k, \iota), F[x \mapsto (k, \iota)]) \\ \quad \text{if } k \neq \mathsf{var} \wedge k \neq \mathsf{iso} \end{array}$$

Fig. 8. (Non-)destructive reads

REGION-ALLOC-HEAP-MUT

$$\frac{\begin{array}{c} x\ \mathit{fresh} \qquad \forall i \in [1, n].\ \mathbf{get}\,(F_i, \mathit{use}_i) = (v_i, F_{i+1}) \qquad \mathbf{fields}\,(C) = f_1, \ldots, f_n \\ o = (\#C, [f_1 \mapsto v_1, \ldots, f_n \mapsto v_n]) \qquad \iota\ \mathit{fresh} \qquad S'' = S', \iota \mapsto o \qquad F' = F_{n+1}, x \mapsto (\mathsf{mut}, \iota) \end{array}}{\langle (r, S, F_1) :: RS; (r, S') * H_{op}; H_{cl}; H_{fr} \rangle \xrightarrow{\mathit{halloc}(x,\mathsf{mut},\#C,\mathit{use}_1\ \ldots\ \mathit{use}_n)} \langle (r, S, F') :: RS; (r, S'') * H_{op}; H_{cl}; H_{fr} \rangle}$$

REGION-ALLOC-HEAP-ISO

$$\frac{\begin{array}{c} x\ \mathit{fresh} \qquad \forall i \in [1, n].\ \mathbf{get}\,(F_i, \mathit{use}_i) = (v_i, F_{i+1}) \qquad \mathbf{fields}\,(C) = f_1, \ldots, f_n \\ o = (\#C, [f_1 \mapsto v_1, \ldots, f_n \mapsto v_n]) \qquad \iota\ \mathit{fresh} \qquad r'\ \mathit{fresh} \qquad F' = F_{n+1}, x \mapsto (\mathsf{iso}, \iota) \end{array}}{\langle (r, S, F_1) :: RS; H_{op}; H_{cl}; H_{fr} \rangle \xrightarrow{\mathit{halloc}(x,\mathsf{iso},\#C,\mathit{use}_1\ \ldots\ \mathit{use}_n)} \langle (r, S, F') :: RS; H_{op}; (r', [\iota \mapsto o]) * H_{cl}; H_{fr} \rangle}$$

Allocation on the heap is caused by the effect $\mathit{halloc}(x, k, C, \mathit{use}_1 \ldots \mathit{use}_n)$, which instructs the region language to **h**eap **alloc**ate a new C object with fields initialized according to $\mathit{use}_1 \ldots \mathit{use}_n$ and bind it

to the name x. The capability k denotes whether to allocate in the current region (REGION-ALLOC-HEAP-MUT) or in a new region (REGION-ALLOC-HEAP-ISO). Since each *use* is a possibly destructive variable access the ordering matters. We begin by performing these one by one with the local variables F_1. Each value v_i is paired up with the corresponding field f_i of the class and put into an object o. We then add o at location ι to the subheap of the currently active region, or add a new region r' in the closed regions containing only the object o at location ι. Finally we bind the object to x in the top frame with capability mut or iso. We omit the rule REGION-ALLOC-TEMP which allocates objects with capabilities tmp or var in the temporary store S of the currently active region frame.

The key rules of the region language govern entering and exiting a region.

REGION-ENTER-OK

$$\frac{\begin{array}{c} w\ fresh \qquad \forall i \in [1,n].\ z_i\ fresh \qquad \forall i \in [1,n].\ \mathbf{get}\,(F_i, use_i) = ((k_i,\iota_i), F_{i+1}) \\ \forall i \in [1,n].\ v_i' = \begin{cases} (k_i,\iota_i) & \text{if } k_i = \text{iso} \\ (\text{paused} \odot k_i, \iota_i) & \text{otherwise} \end{cases} \\ F = [z_i \mapsto v_i' \mid i \in [1,n]] \qquad F_{n+1}(y) = (\_,\iota) \qquad \text{cfg\_load}((r,S,F_1) :: RS, H_{op}, \iota) = o[f \mapsto (\_,\iota')] \\ \iota' \in dom\,(S') \qquad \iota''\ fresh \qquad F' = F, w \mapsto (k,\iota'') \qquad RF = (r', [\iota'' \mapsto (\texttt{\#Cell}, [\texttt{val} \mapsto (\text{mut},\iota')])], F') \end{array}}{\langle (r,S,F_1) :: RS; H_{op}; (r',S') * H_{cl}; H_{fr}\rangle \xrightarrow{enter\,(w,k,y.f,z_1=use_1,\ldots,z_n=use_n)} \langle RF :: (r,S,F_{n+1}) :: RS; (r',S') * H_{op}; H_{cl}; H_{fr}\rangle}$$

REGION-ENTER-FAIL

$$\frac{\begin{array}{c} \forall i \in [1,n].\ \mathbf{get}\,(F_i, use_i) = ((k_i,\iota_i), F_{i+1}) \qquad F_{n+1}(y) = (\_,\iota) \\ \text{cfg\_load}((r,S,F) :: RS, H_{op}, \iota) = o[f \mapsto (\_,\iota')] \qquad \forall R' \in H_{cl}.\ \iota' \notin dom\,(R'.S) \end{array}}{\langle (r,S,F) :: RS; H_{op}; H_{cl}; H_{fr}\rangle \xrightarrow{badenter(y.f)} \langle (r,S,F) :: RS; H_{op}; H_{cl}; H_{fr}\rangle}$$

REGION-ENTER-OK shows successfully entering a region r' through its bridge object ι' stored in the field f of the variable y. (This operation can fail if R' is already opened. This will not change the state in the region language, as seen in REGION-ENTER-FAIL, and it is up to the command language to choose how to handle this: by exception, having a construct like **`if-enter-else`**, etc. For simplicity, the command language steps to a failure state.) The *enter* effect supplies four things: the name w and capability k of the parameter of the **`enter`** block, the field $y.f$ through which we are entering, and a list of bindings $\overline{z = use}$ denoting the block's captured variables. Going back to Listing 1, the **`enter`** block captures `i`, so the corresponding effect would include $z = \texttt{i}$. Note that z is chosen by the command language and due to variable renaming is not necessarily `i`. Considering the rule again, we first use the **get** helper function to perform the *use*s. For each resulting value (k_i, ι_i), we apply the paused viewpoint adaptation when k_i is not iso and create a new mapping F of the captured variables. We then get the value ι of y, load its corresponding object o and extract the value ι' of field f (our bridge object). On the last line of the premises we check that ι' is an object in a closed region r'; this region will be moved into the collection of open regions. We extend F with a mapping from w to a fresh location ι'', and finally install this extended F into a region frame RF where ι'' is the identifier of a ref cell object pointing to our bridge object ι'. We push this region frame onto the region stack.

REGION-EXIT-HEAP

$$\frac{\begin{array}{c} x\,\mathit{fresh} \qquad \mathbf{get}\,(F', \mathit{use}) = (v, F'') \qquad F''(z) = (\_, \iota') \\ \mathsf{load}(S', \iota') = o'[f' \mapsto (\_, \iota'')] \\ F(y) = (\_, \iota) \qquad \mathsf{heap\_load}(H_{op}, \iota) = o[f \mapsto (k, \_)] \\ \mathsf{heap\_store}(H_{op}, \iota, o[f \mapsto (k, \iota'')]) = H'_{op} \qquad F''' = F, x \mapsto v \end{array}}{\langle (r', S', F') :: (r, S, F) :: RS; (r', S_{op}) * H_{op}; H_{cl}; H_{fr} \rangle \xrightarrow{\mathit{exit}\,(x,\mathit{use},y.f,z.f')} \langle (r, S, F''') :: RS; H'_{op}; (r', S_{op}) * H_{cl}; H_{fr} \rangle}$$

REGION-EXIT-TEMP

$$\frac{\begin{array}{c} x\,\mathit{fresh} \qquad \mathbf{get}\,(F', \mathit{use}) = (v, F'') \qquad F''(z) = (\_, \iota') \\ \mathsf{load}(S', \iota') = o'[f' \mapsto (\_, \iota'')] \\ F(y) = (\_, \iota) \qquad \mathsf{stack\_load}((r, S, F) :: RS, \iota) = o[f \mapsto (k, \_)] \\ \mathsf{stack\_store}((r, S, F) :: RS, \iota, o[f \mapsto (k, \iota'')]) = (r, S'', F) :: RS' \qquad F''' = F, x \mapsto v \end{array}}{\langle (r', S', F') :: (r, S, F) :: RS; (r, S_{op}) * H_{op}; H_{cl}; H_{fr} \rangle \xrightarrow{\mathit{exit}\,(x,\mathit{use},y.f,z.f')} \langle (r, S'', F''') :: RS'; H_{op}; (r, S_{op}) * H_{cl}; H_{fr} \rangle}$$

REGION-EXIT-HEAP and REGION-EXIT-TEMP describe exiting the region r', popping its region frame from the top of the region frame stack. After exiting, the region frame corresponding to r will be on the top of the stack and thus active. The *exit* effect provides *use* and x which correspond to the return value and the variable to which this will be bound (in the stack frame of r). $z.f'$ specifies a location in r' where a reference to the new bridge object can be found. Finally, $y.f$ specifies a location where this reference will be written. The only difference between REGION-EXIT-HEAP and REGION-EXIT-TEMP is where the object pointed to by y is located. In the former it is on the heap of some open region, while for the latter it is in a temporary store in the region stack.

For simplicity, we do not implicitly reinstate iso variables captured from the previous region even if they are still valid upon exit from a region (this would be sound, *c.f.*, Listing 4 where variables `reviews` and `zip` are reinstated in the top-level scope after line 13). This is without loss of generality as we can return them in an object together with the result v and reinstate them manually.

REGION-FREEZE

$$\frac{\begin{array}{c} x\,\mathit{fresh} \qquad \mathbf{get}\,(F, \mathit{use}) = ((k, \iota), F') \qquad \iota \in \mathit{dom}\,(R.S) \\ H = \mathsf{reachable\_regions}(R, (H * H_{cl}) * H_{op}) \qquad F'' = F', x \mapsto (\mathsf{imm}, \iota) \end{array}}{\langle (r, S, F) :: RS; H_{op}; R * (H * H_{cl}); H_{fr} \rangle \xrightarrow{\mathit{freeze}(x,\mathit{use})} \langle (r, S, F'') :: RS; H_{op}; H_{cl}; (R * H) * H_{fr} \rangle}$$

REGION-MERGE

$$\frac{\begin{array}{c} x\,\mathit{fresh} \qquad \mathbf{get}\,(F, \mathit{use}) = ((k, \iota), F') \qquad \iota \in \mathit{dom}\,(R.S) \\ R' = (r, S' \uplus R.S) \qquad F'' = F', x \mapsto (\mathsf{mut}, \iota) \end{array}}{\langle (r, S, F) :: RS; (r, S') * H_{op}; R * H_{cl}; H_{fr} \rangle \xrightarrow{\mathit{merge}(x,\mathit{use})} \langle (r, S, F'') :: RS; R' * H_{op}; H_{cl}; H_{fr} \rangle}$$

The rules for merging (REGION-MERGE) and freezing (REGION-FREEZE) are similar. Both perform a *use* to get an object identifier ι and find its containing region R among the closed regions. For merges, the subheap of R is merged with the subheap of the currently active region, and ι is bound to x as mut. For freezes, all the reachable regions of R are moved from the closed to the frozen regions together with R, and ι is bound to x as imm.

In addition to allocation in the temporary store, we have omitted the rules for type casts and rebinding of variables.

$$\begin{array}{rcl} e & ::= & \mathit{use} \mid \mathbf{let}\ x = b\ \mathbf{in}\ e \\ & \mid & \mathbf{if\ typetest}(\mathit{use}, t)\{y \Rightarrow e\}\{y \Rightarrow e\} \\ \mathit{use} & ::= & x \mid \mathbf{drop}\ x \\ b & ::= & *\mathit{lval} \mid \mathit{lval} := \mathit{use} \mid \mathit{fnc}(\overline{\mathit{use}}) \mid \mathbf{var}\ \mathit{use} \\ & \mid & \mathbf{new}\ k\ C(\overline{\mathit{use}}) \mid \mathbf{freeze}\ \mathit{use} \mid \mathbf{merge}\ \mathit{use} \\ & \mid & \mathbf{enter}\ \mathit{lval}\ [\overline{y = \mathit{use}}]\{z \Rightarrow e\} \mid e \\ \mathit{lval} & ::= & x \mid x.f \\ t & ::= & k\ CL \mid t \mid t \end{array}$$

Fig. 9. Syntax of the command language

7.2 Static Semantics of the Command Language

The command language is an imperative language in A-normal form. The syntax is shown in Fig. 9. We encode mutable variables of type t as ref cells. For uniformity we model these as objects of type `Cell[`t`]` with a single field `val`. The **`if typetest`** expression is a dynamic type test similar to Java 16 style pattern matching, **`drop`** x denotes a destructive read, $*lval$ dereferences a field or ref cell, and **`var`** allocates a new ref cell with the capability var and initializes its value from *use*. For simplicity we provide **`enter`** blocks with an explicit capture list, but this could also be inferred from variable use. Types t are unions $t_1 \mid t_2$ or $k\ CL$, where k is a capability and CL is `Cell` or a class name C.

The static semantics is a flow-sensitive type system producing judgements of the form $\Gamma_1 \vdash r : t \dashv \Gamma_2$ ($r \in e \cup b \cup use$). Thus it statically tracks destructive reads and strong updates of unique variables. We discuss a few of the rules below.

CMD-TY-USE-KEEP

$$\frac{\vdash \Gamma_1 \qquad \Gamma_1(x) = k\ CL \qquad k \neq \text{iso} \qquad k \neq \text{var}}{\Gamma_1 \vdash x : k\ CL \dashv \Gamma_1}$$

CMD-TY-USE-DROP

$$\frac{\vdash \Gamma_1 \qquad \Gamma_1 = \Gamma[x : t] \qquad \Gamma_2 = \Gamma[x : \textbf{undef}]}{\Gamma_1 \vdash \textbf{drop}\ x : t \dashv \Gamma_2}$$

CMD-TY-DEREF-FIELD

$$\frac{\Gamma(x) = k\ CL \qquad \textbf{ftype}\ (CL, f) = t \qquad \vdash k \odot t}{\Gamma \vdash *x.f : (k \odot t) \dashv \Gamma}$$

CMD-TY-ASSIGN

$$\frac{\Gamma_1 \vdash use : t \dashv \Gamma_2 \qquad \Gamma_2(x) = k\ CL \qquad \textbf{ftype}\ (CL, f) = t \qquad k \in \{\text{mut}, \text{tmp}\}}{\Gamma_1 \vdash x.f := use : t \dashv \Gamma_2}$$

CMD-TY-ASSIGN-VAR

$$\frac{\Gamma_1 \vdash use : t_1 \dashv \Gamma_2[x : \text{var Cell}[t_2]]}{\Gamma_1 \vdash x := use : t_2 \dashv \Gamma_2[x : \text{var Cell}[t_1]]}$$

Reading a variable x that is not a var or iso is straightforward and introduces an alias (CMD-TY-USE-KEEP). When x is var or iso, CMD-TY-USE-DROP allows reading the variable but undefines it in the environment as a side-effect to ensure its single use. When accessing a field $x.f$ (CMD-TY-DEREF-FIELD), its type is subject to viewpoint adaptation $k \odot t$ where k is the capability of x and t the type of f. Note that viewpoint adaptation disallows reading an iso field unless k is imm, expressing the fact that freezing a region is deep (all nested regions will be frozen as well). A field $x.f$ can be updated through assignment (CMD-TY-ASSIGN) when the capability of x is mut or tmp, *i.e.*, internal references in the currently active region (note that assignment returns the *old* value of the field). Viewpoint adaptation is not needed as we are moving values rather than copying them, allowing swapping of iso references. Local variables allow strong updates (CMD-TY-ASSIGN-VAR). As we model them as ref cells we update the type parameter for x after assignment.

CMD-TY-ENTER

$$\frac{\begin{array}{c}\forall i \in [1, n].\ \Gamma_i \vdash use_i : t_i \dashv \Gamma_{i+1} \qquad \Gamma_{n+1}(x) = k\ CL \\ \text{open}(k) \qquad \textbf{ftype}\ (CL, f) = t \qquad \text{cap}(\text{iso}, t) \\ \Gamma' = y_1 : t'_1, \ldots, y_n : t'_n \text{ where } t'_i = \begin{cases} t_i & \text{if cap}(\text{iso}, t_i) \\ \text{paused} \odot t_i & \text{otherwise} \end{cases} \\ t' = make\_mut(t) \qquad \Gamma', z : \text{tmp Cell}[t'] \vdash e : t'' \dashv \Gamma'', z : \text{tmp Cell}[t'] \qquad \text{cap}(\{\text{iso}, \text{imm}\}, t'')\end{array}}{\Gamma_1 \vdash \textbf{enter}\ x.f\ [y_1 = use_1, \ldots, y_n = use_n]\ \{z \Rightarrow e\} : t'' \dashv \Gamma_{n+1}}$$

CMD-TY-ENTER-VAR

$$\frac{\begin{array}{c}\forall i \in [1, n].\ \Gamma_i \vdash use_i : t_i \dashv \Gamma_{i+1} \qquad \Gamma[x : \text{var Cell}[t]] = \Gamma_{n+1} \qquad \text{cap}(\text{iso}, t) \\ \Gamma' = y_1 : t'_1, \ldots, y_n : t'_n \text{ where } t'_i = \begin{cases} t_i & \text{if cap}(\text{iso}, t_i) \\ \text{paused} \odot t_i & \text{otherwise} \end{cases} \\ \Gamma', z : \text{var Cell}[make\_mut(t)] \vdash e : t'' \dashv \Gamma'', z : \text{var Cell}[t'] \\ \text{cap}(\text{mut}, t') \qquad \text{cap}(\{\text{iso}, \text{imm}\}, t'')\end{array}}{\Gamma_1 \vdash \textbf{enter}\ x\ [y_1 = use_1, \ldots, y_n = use_n]\ \{z \Rightarrow e\} : t'' \dashv \Gamma[x : \text{var Cell}[make\_iso(t')]]}$$

The predicate $\mathsf{cap}(k, t)$ asserts that the type t has capability k; the predicate $\mathsf{open}(k)$ is true if the capability denotes an open region, *i.e.*, k is mut, tmp, var or paused. Finally, $make\_mut(t)$ and $make\_iso(t)$ return a t whose iso capabilities have been replaced by mut and vice versa.

Opening a region through a field $x.f$ (CMD-TY-ENTER) requires that x's capability is open, and f's capability is iso. We create a new environment Γ' with the captured variables $y_1, ..., y_n$, using viewpoint adaptation to suspend the types of all non-iso variables, as well as a tmp ref cell holding the bridge object. We use $make\_mut(t)$ to change the type of the bridge from iso to mut as control is moving *inside* the opened region. Finally, the **enter** block may only return iso's and imm's. (Note that entering a region through a field incurs a dynamic check to see if the region is already open.)

Opening a region through a var ref cell (CMD-TY-ENTER-VAR) is similar to a field (CMD-TY-ENTER), but allows strong updates of the ref cell holding the bridge object by retaining its var capability. This allows changing the bridge object's type from within the **enter** block.

$$\textsc{cmd-ty-merge}\ \frac{\Gamma_1 \vdash use : \mathsf{iso}\ CL \dashv \Gamma_2}{\Gamma_1 \vdash \textbf{merge}\ use : \mathsf{mut}\ CL \dashv \Gamma_2} \qquad \textsc{cmd-ty-freeze}\ \frac{\Gamma_1 \vdash use : \mathsf{iso}\ CL \dashv \Gamma_2}{\Gamma_1 \vdash \textbf{freeze}\ use : \mathsf{imm}\ CL \dashv \Gamma_2}$$

The rules for merging and freezing a region (CMD-TY-MERGE and CMD-TY-FREEZE) are straightforward. Both demand that the value that we operate on is an iso reference (*i.e.*, bridge object to a closed region), and produce either a mut or imm depending on the operation.

7.3 Dynamic Semantics of the Command Language

A configuration $\{de\}$ in the command language is a dynamic expression de, which is an extension of e by "entered blocks" that propagates syntactically the nesting structure of enters and exits, and thus dynamically tracks the nesting of open regions, and **Failure**, used to report failed dynamic checks when entering an already open region. The dynamic semantics steps a configuration and produces an effect of the same kind consumed by the region language. For example, the expression **let** $x = *y.f$ **in** e produces the effect $load(x, y.f)$, which tells the region language to load the field f from the object stored in y and store it in x.

7.4 Interaction Between the Region and Command Languages

A complete configuration is a product of the configurations of the region and command languages. It steps if there is an effect that steps both of them in tandem:

$$\textsc{tandem-step}\ \frac{de \xrightarrow{Eff} de' \qquad rcfg \xrightarrow{Eff} rcfg'}{\langle \{\, de \,\}\ rcfg \rangle \rightarrow \langle \{\, de' \,\}\ rcfg' \rangle}$$

A dynamic expression is typed under a stack of typing contexts $\overline{\Gamma}$, corresponding to the nesting of entered blocks. We lift the static semantics of the command language to define well-formed effects: the relation $\overline{\Gamma} \vdash Eff \dashv \overline{\Gamma}'$ statically describes the effect Eff and how it changes the typing context. For example, the static description of the effect $load(x, y.f)$ states that the type of y in the top-most entered block is $k\ CL$, that CL has a field f of type t, and that the viewpoint adapted type $k \odot t$ is well-formed (*c.f.*, CMD-TY-DEREF-FIELD).

In order to reason about soundness, we define well-formedness of a configuration in the region language as the relation $\overline{\Gamma} \vdash \langle RS; H_{op}; H_{cl}; H_{fr} \rangle$. A well-formed configuration ensures four things. First, the stack of environments $\overline{\Gamma}$ mirrors the region stack RS so that each environment describes the local variables of a region frame in RS. Second, each field of every object in the configuration contains a value that corresponds to its static type. Third, we have invariants about the reference

capabilities: var references are unique, objects in frozen regions only refer to other references in frozen regions, mut references point within the same region and to the heap, tmp and var point within the same region and to the temporary store, paused references point downwards in the region stack, etc. Finally, we have the invariant that the object graph and its regions have the expected topology. We describe this invariant in detail in the following section.

7.5 The Topology Invariant

The most important properties of the object (and region) graph are captured in a single invariant that we call the topology invariant (Fig. 10). We express this as a property that holds for any pair of references ref_1 and ref_2 in a well-formed configuration. The helper functions src() and dst() denote the storage location and referee of a reference respectively; reg() denotes the region of an object (or variable); regions() projects the region identifiers out of a set of regions H.

For all references ref_1 and ref_2, either: they are the same reference, *e.g.*, both are stored in the same $\iota.f$ or variable x (1); both refer to objects in different regions (2); or at least one of them is an intra-region reference (3); refers to a permanently immutable object (4); or is a reference outwards in the nesting hierarchy, downwards in the region stack (5). The relation $RS \vdash \mathsf{dst}(ref) \preceq \mathsf{src}(ref)$ holds if $\mathsf{dst}(ref)$ is higher up in RS than $\mathsf{src}(ref)$ and ref originates from the temporary store. In other words, we allow temporary references into suspended regions from open regions.

The topology invariant has several important implications: The *object graph* inside a region is unconstrained (3). The object graph of the permanently immutable objects is unconstrained (4). Temporary objects in an open region R are allowed to refer to objects in an open region R' as long as R' was opened before R (5). Finally, considering the whole invariant, if we have two external references ((3) does not hold) pointing into the same non-frozen region ((2) and (4) do not hold), and neither of them points downwards in the region stack ((5) does not hold), then they must be the same reference ((1) holds). In particular, this means that there is at most one external reference into any closed region, implying that the region graph of closed regions forms a forest.

$$
\begin{aligned}
&\forall ref_1, ref_2 \in \mathsf{references}(\langle RS; H_{op}; H_{cl}; H_{fr}\rangle).\\
&\bigvee \begin{cases}
ref_1 = ref_2 & (1)\\
\mathsf{reg}(\mathsf{dst}(ref_1)) \neq \mathsf{reg}(\mathsf{dst}(ref_2)) & (2)\\
\mathsf{reg}(\mathsf{src}(ref_1)) = \mathsf{reg}(\mathsf{dst}(ref_1)) \vee \\
\quad \mathsf{reg}(\mathsf{src}(ref_2)) = \mathsf{reg}(\mathsf{dst}(ref_2)) & (3)\\
\mathsf{reg}(\mathsf{dst}(ref_1)) \in \mathsf{regions}(H_{fr}) \vee \\
\quad \mathsf{reg}(\mathsf{dst}(ref_2)) \in \mathsf{regions}(H_{fr}) & (4)\\
RS \vdash \mathsf{dst}(ref_1) \preceq \mathsf{src}(ref_1) \vee \\
\quad RS \vdash \mathsf{dst}(ref_2) \preceq \mathsf{src}(ref_2) & (5)
\end{cases}
\end{aligned}
$$

Fig. 10. The topology invariant

The Topology Invariant and Fig. 1. Applying the topology invariant to all pairs of references in Fig. 1, assuming R was opened after R', the reference $e \to n$ is allowed to co-exist with any other alias of n since $RS \vdash n \preceq e$ (5). The reference $m \to e$ is *not* allowed to co-exist with $m \to a$ since there would be two external references into the same region (1–5). However, $e \to a$ can co-exist with $m \to a$ since the former stays within its region (3). The references $a \to i$ and $o \to i$ are allowed to co-exist because i is in a frozen region (4). Finally, $o \to a$ is illegal both because it cannot co-exist with $m \to a$, and since references in frozen regions cannot point to non-frozen regions.

7.6 Reggio is Sound

We prove soundness of our system by proving variants of progress and preservation for the respective language. (The full proofs are available in a technical report [Arvidsson et al. 2023].)

Lemma 7.1. ***Command Language Progress*** *A well-formed command configuration is done, has failed or can step:* $\overline{\Gamma} \vdash \{de\} \implies de = use \vee de = \textbf{\textit{Failed}} \vee \exists Eff, de'.\ \{de\} \xrightarrow{Eff} \{de'\}$.

Lemma 7.2. ***Command Language Preservation*** *The command language preserves well-formedness and produces well-formed effects:* $\overline{\Gamma} \vdash \{de\} \wedge \{de\} \xrightarrow{Eff} \{de'\} \implies \exists \overline{\Gamma}'.\, \overline{\Gamma}' \vdash \{de'\} \wedge \overline{\Gamma} \vdash Eff \dashv \overline{\Gamma}'$.

The command language is more permissive than the region language, since it has no way of inspecting the state of the global configuration. For example, an enter can *always* both fail and succeed in the command language, whereas the region language always permits exactly one of the behaviours. This affects the formulation of progress:

Lemma 7.3. ***Region Language Progress*** *In a well-formed configuration where the command configuration can step, there is some effect which steps both configurations:* $\vdash \langle \{de\}\ rcfg \rangle \wedge \{de\} \xrightarrow{Eff} \{de'\} \implies \exists Eff', de'', rcfg'.\{de\} \xrightarrow{Eff'} \{de''\} \wedge rcfg \xrightarrow{Eff'} rcfg'$.

Lemma 7.4. ***Region Language Preservation*** *The region language preserves well-formedness for well-formed effects:* $\overline{\Gamma} \vdash rcfg \wedge rcfg \xrightarrow{Eff} rcfg' \wedge \overline{\Gamma} \vdash Eff \dashv \overline{\Gamma}' \implies \overline{\Gamma}' \vdash rcfg'$.

Note that Lemma 7.4 includes preservation of the topology invariant. Together, these lemmas prove the final soundness theorem:

Theorem 7.5. ***Soundness*** *A program never gets stuck and it preserves well-formedness:* $\vdash \langle \{de\}\ rcfg \rangle \implies de = use \vee de = \textbf{\textit{Failed}} \vee \exists de', rcfg'.\langle \{de\}\ rcfg \rangle \rightarrow \langle \{de'\}\ rcfg' \rangle \wedge \vdash \langle \{de'\}\ rcfg' \rangle$.

8 REGGIO IN VERONA

While Reggio regions are a stand-alone language design component, they were developed specifically for the Verona programming language, from where the overarching goal (G1) stems. In this section, we describe Verona-specific aspects and revisit concurrency-related goals (G5) and (G6).

8.1 Safe Concurrency

While regions and isolation can form the backbone of a "safe concurrency" story for a language, concurrency is an orthogonal aspect to our region design. Reggio regions can be integrated with different concurrency models. The necessary feature missing from this paper is a way to share regions across threads of control.

Verona uses a concurrency model based on *behaviours* (tasks that do not join or have a return value) that operate on *cowns*, short for concurrent owners. A cown is a wrapper around an iso that permits regions to be indirectly shared across multiple threads of control, but importantly does not permit direct access to its contents. Cowns and iso's are similar in that an explicit operation is needed to access their contents. In the case of iso's, access is immediate and synchronous as exclusivity is already established. In the case of cowns, access is asynchronous and will only commence after exclusive access has been established dynamically. This check requires region isolation for soundness [Cheeseman et al. 2023] and as a result, any mutable reference accessible to a thread of control is safe to access synchronously (G6).

For a complete introduction to Verona's concurrency model, see work by Cheeseman et al. [2023].

8.2 Concurrent Memory Management

As memory management must only consider objects inside the active region (G3) when determining liveness (*c.f.*, §5), and regions are always exclusive to one thread, reference count manipulations do not need atomic instructions, tracing GC does not need barriers, and there is no need to momentarily stop all threads as in concurrent GC's [Click et al. 2005; Flood et al. 2016; Lidén and Karlsson 2018].

By extension, a thread in Verona is free to mediate between program work or memory management work without informing or synchronising with other threads. Thus, we achieve concurrent memory management (G5).

8.3 Support for Different Memory Management Strategies

Verona currently supports three different memory management strategies for regions: *arena allocation*, *reference counting*, and *tracing GC*. How a region manages its memory is decided at use-site at creation time using a qualifier on the **new** keyword: **new iso**<Arena>, **new iso**<RC> and **new iso**<GC>. As liveness is a local property, different regions' memory management does not interact, so we do not need to *e.g.*, propagate this information further in the program.

Selecting memory management at use-site is desirable since it lets a programmer implement a data structure or library without having to commit to decisions that could limit its future use. Such a design also allows straightforward support for libraries that consist of multiple nested regions whose memory management can be controlled when the library is instantiated by programmatic means, *e.g.*, through a strategy pattern or equivalent.

8.4 Memory Management of Immutable Objects

Note that the memory management offered by regions does not extend to immutable objects, at least not conceptually. One option for implementation is going the way of Erlang and let a region have a copy of each immutable object it references. This may facilitate fast reclamation, but increases memory pressure. Furthermore, it introduces a $O(n)$ copying overhead for transferring immutable objects across region boundaries, in sharp contrast with (G4).

In the Verona run-time, we permit immutable objects to be shared between regions. Thus, when a region is collected, we must detect its implications for liveness of immutable objects. Immutable objects must also consider roots across multiple threads. On the other hand, tracing immutable objects is easy and efficient as the structures are guaranteed not to change underfoot [Clebsch et al. 2017]. How Verona manages immutable objects is out of scope of this paper.

8.5 Propagating Capabilities Through Self Typing

Verona is a class-based programming language. As an instance's capability is determined at use-site, methods declare an explicit self-capability, *e.g.*, `self : mut` to propagate the external view of the instance into the instance. A class may provide several different implementations of a method overloaded on the self-capability.

```
class Cell {                                                    1
  var value : I64 // mut Store[mut I64]                         2
  def set_value(self:mut, v:I64) {                              3
    // value : (mut ⊙ mut) Store[mut I64]                       4
    value := v                                                  5
  }                                                             6
  def set_value(self:paused, v:I64) {                           7
    // value : (paused ⊙ mut) Store[mut I64]                    8
    value := v // does not typecheck!                           9
  }                                                             10
  def get_value(self:mut) = value                               11
  def get_value(self:paused) = value                            12
}                                                               13
```

Listing 5. Self typing in `Cell`.

A method can only be called on a receiver if its self-capability matches the receiver's static type, which means that the object's treatment of itself internally will match the external view, both in terms of restrictions and abilities. For example, a method whose self-capability is paused can only be called when the receiver's region is paused. Notably, it is *not* permitted to call a paused method on a mut receiver because this would lead to aliasing between mut and paused (which would weaken paused from temporarily immutable to read-only).

Methods that are polymorphic in their self capability can be used to avoid multiple near-identical versions of a single method like in Listing 5. For brevity, we refrain from discussing this further.

9 RELATED WORK

We started out by describing related work leading up to Rust. We now extend this picture by going beyond Rust and also relating our work to garbage collection work before revisiting novelty.

Beyond Rust. In addition to what we have already covered, there is continuing research into Rust to alleviate its restrictions, including incorporating a garbage collector [Coblenz et al. 2022], careful library design [Beingessner 2015], phantom types [Yanovski et al. 2021], or proving unsafe Rust code correct [Jung et al. 2019, 2017; Noble et al. 2022].

Recent research has focused on techniques for "post-Rust languages", building on Rust's use of ownership types, but supporting more flexible program topologies (and hopefully more efficient execution), typically by increasing the complexity of the type system. This remains an active research area: the tradeoffs between regions, ownership, types, capabilities, effects, topologies, restrictions etc are complex and multifaceted [Brachthäuser et al. 2022; Gordon 2020].

Pony [Clebsch et al. 2017; Franco et al. 2018] employs implicit regions, external uniqueness and ownership to offer high performance for actor programs by concurrent execution on multicore CPUs, while maintaining data-race and memory safety. Building on capabilities used to describe what programs can do with particular references [Boyland et al. 2001] Pony offers at least six "reference capabilities": unique, thread-local, read-only, write-only, and identity-only, (globally) immutable, plus type modifiers for ephemeral (Hogg's "free") and aliased references. Reading or writing a field depends on the capabilities of both the reference to the object, and of the field within the object: there are 43 valid cases from 72 possible combinations of capabilities.

Fernandez-Reyes et al. [2021] design Dala as a simplified alternative to Pony, based on three different kinds of objects—immutable, unique (aka isolated), and thread-local—rather than six different kinds of references. Dala programs are also data-race free, however this guarantee may be provided by a race detector at runtime, or by an optional/gradual type system.

Milano et al.'s [2022] Gallifrey aims to be more flexible than Rust, by relying at least as much on MLKit style region inference as on ownership annotations. Rather than an explicit global ownership model, Gallifrey programmers have to annotate unique (aka isolated) object fields, and identify parameters that will be consumed by a method invocation or that should be in the same region as other parameters or the method result. A dynamic "if disconnected" predicate searches the program's heap at runtime to determine of two references are mutually disjoint.

Cogent [O'Connor et al. 2021] is a derivative of Haskell for systems programming. It adopts a Rust-like discipline, permitting either multiple read-only references to objects, or a single read-write reference. Cogent uses annotations to support both a formally defined operational semantics, generation of executable C source code, and a proof certificate proving that the generated code accurately implements the semantics.

Garbage Collection. Of the GC design goals, (G3) and (G5) can be met to some extent *without* region isolation. Here, Reggio's contribution is trading additional work to manage regions (at development time) for reduced overheads of managing memory (at run-time) due to avoiding GC techniques like remembered sets, barrier synchronisation, and stop-the-world pauses. Region isolation guarantees that a region's remembered set will always be empty. Thus, there is no cost associated with additional region partitioning due to tracking of inter-region references.

With respect to (G3), generational GC's (*e.g.*, G1 [Detlefs et al. 2004]) and thread-local GC's (*e.g.*, [Domani et al. 2002]) support collecting only a portion of the heap (*e.g.*, just the young generation or one particular thread), but the shape and size of this heap is beyond programmer control (*e.g.*, all young or thread-local objects will take part of GC, not just particular data structures). Furthermore, in the absence of (something like) region isolation, inter-region aliases must be tracked dynamically to be able to correctly compute liveness. Actor GC's that rely on actor isolation using types (*e.g.*, Pony [Clebsch et al. 2017; Franco et al. 2018]) or copying (*e.g.*, Erlang [Armstrong 2007]) are close as they allow individual actor-local heaps to be collected. This is similar to Reggio's regions, but Reggio supports partitioning of the heap without an imposed asynchronous indirection.

With respect to (G2), while it may be possible to run different GC's in different generations (or threads, actors, etc.), GC's typically use the same algorithm for the entire heap, with minor tweaks (*e.g.*, to account for different object characteristics due to age) as do actor GC's. HRTGC is a real-time GC for mixed-criticality work-loads [Pizlo et al. 2007] that hierarchically decomposes the heap into regions that each run a different tracing GC, tuned differently and with different collection frequency. HRTGC permits inter-region references and tracks them dynamically. In the actor world, Isolde [Yang and Wrigstad 2017] permits actors implemented using type-enforced actor isolation [Castegren and Tobias Wrigstad 2016; Castegren and Wrigstad 2017] to manage their memory concurrent with their execution, using a reference-counting based scheme.

With respect to (G5), garbage collectors like C4 [Tene et al. 2011], Shenandoah [Flood et al. 2016] and ZGC [Lidén and Karlsson 2018] provide concurrent collection with brief stop-the-world pauses to coordinate phase changes, with pause times invariant of heap sizes. Their heaps have unrestricted references, and instead rely on dynamic checks in read and write barriers. The aforementioned actor GC's [Armstrong 2007; Clebsch et al. 2017; Franco et al. 2018] support "fully concurrent" collection: an actor can choose to collect its local garbage without synchronising with any other concurrent activity. Reggio's regions additionally allow us to statically detect when an entire region is invalidated, without the need for a specific actor collector to eventually detect the floating garbage (*e.g.*, [Clebsch and Drossopoulou 2013]). Explicitly killing an actor to instantly free its heap is a common pattern in Erlang where it is made possible by copying objects on transfer, giving up (G4).

10 DISCUSSION

First, we place Verona's design concepts into the context of all related work. Almost any ownership system paired with external uniqueness will support region isolation and dynamic reconfiguration. Verona's key contribution here is a negative contribution, but important nonetheless. Almost all other systems provide one or more top, global, or shared heap region, and in various ways permit references from inner/encapsulated/shorter-lived regions back to outer/enclosing/longer-lived regions. (Generational garbage collection works on a very similar principle [Jones et al. 2016]). Verona does not, and this enhanced decoupling of regions is critical to achieving many of our goals, especially about concurrency, and independent GC.

Verona's dynamic mutability and relaxed isolation is novel and differs from other ownership and region systems. Inasmuch as region systems like the MLKit are based on inference, and are sound for all legal programs, questions of mutability and isolation don't apply—if the program is type-correct, the inference system can always place objects into regions such that no region errors will arise at runtime. Programming with more explicit regions, or with ownership and capability annotations, either lack polymorphism (*e.g.*, Dala) or require complex resolution or viewpoint adaptation rules, or asynchronous indirections (*e.g.*, Pony). In a way, Verona's region system trades precision for simplicity. It cannot construct data structures that consist of several morally overlapping regions, as is possible in *e.g.*, C++ or Rust. We believe all programs written in *e.g.*, C++ or Rust pay the price for that precision—even though most programs do not need it.

Verona's "single window of mutability" is probably its most novel concept. In pretty much every language, from FORTRAN to LISP to ML to Haskell to C++ to Pony to Gallifrey, if some code can finagle a mutable reference to an object, the program can always update the object through that reference. In Rust for example, programs can collect up "mutable borrows" (`&mut`) of any number of objects, pass them around as method arguments to anywhere in the program, and then mutate all the borrowed objects. Rust's "interior mutability" (aka C++'s const-cast) just increases the scope for potential mutation. This kind of indiscriminate mutation is exactly what the "single window of mutability" in Verona prevents. Once a program departs—even temporarily—from the scope of an opened region, (*e.g.*, by opening some *other* closed region) the program can no longer modify

anything in that first opened region, no matter what kind of objects are in the region, nor what kind of capability or reference the program has to those objects. Perhaps a single window of mutability will prove too restrictive in practice, which may be why no other system has yet adopted it. Verona demonstrates that it is possible to build a system with a single mutability window as a core design concept; exploring that concept further must necessarily be further work.

The single window of mutability is key to simplicity, both with respect to the type system that enforces region isolation and our invariants for memory management. We need only distinguish objects in the active region, from objects in suspended regions and objects in closed regions. As suspended regions are immutable and closed regions inaccessible, we do not need to distinguish objects belonging to different regions as nothing can be done to them that affects or is affected by their region membership. By having only a single mutable region at a time, non-local operations cannot effect object liveness in the active region, or the liveness of an entire active region. This permits optimisations at the implementation level and simplifies the task of the programmer wishing to reason about—and control—memory management performance.

In contrast to other works which use types to enforce (or impose) a structure on the heap, we let the path of the program through the heap dictate the permissible pointer structure—not the other way around. For example, Verona allows a single point in the program to access the contents of two mutually isolated regions A and B, simply by virtue of opening them in a nested fashion. The key insight is the decoupling of accessing from mutating, implemented through the movable window of mutability. Rather than alternating between accessing A and B, we can gain access to first A and then B without giving up access to A, just the rights to mutate it. Opening B after A allows references to objects in A from B to be created freely, but these references may only persist as long as B remains open. When B is closed, the references to A are invalidated. This retains the flexibility of navigating heap structures in any order, *e.g.*, we could close B then A and open them immediately in the opposite order, allowing pointers from A to B. It also ensures that an object is either mutable or immutable at any moment in time.

11 CONCLUSION

We have presented Reggio, a region system enforced by reference capabilities that partitions a program's heap into a forest of isolated regions. Memory in different regions can be managed differently (G2), incrementally (G3) and concurrently (G5). The single external reference to each region plus their full encapsulation enable cheap ownership transfer (G4) and guarantees freedom from data races (G6). The ability to temporarily trade mutability for access on the region stack allows any region to (temporarily) reference any other region, and also allows "external code" to operate inside a region, which is crucial for libraries and reuse—not all uses of a region can be predicted and supplied in its interface. In combination with the region isolation and the single window of mutability this allows the formulation of topological invariants which are useful for a programmer to control and reason about object liveness and implications of memory management (G1) and can be leveraged for efficient implementation of memory management. Memory management costs are only incurred by the active region (one per thread), and data accesses within that region—whether reading, writing, tracing, or reference counting—never need atomic operations to coordinate with other threads.

ACKNOWLEDGMENTS

This work was partially supported by a grant from the Swedish Research Council (2020-05346), and partially by the Royal Society of New Zealand Te Apārangi Marsden Fund Te Pūtea Rangahau a Marsden grants CRP1801 and CRP2101, and Agoric. We thank the anonymous reviewers at OOPSLA'23 for their input that greatly improved the presentation of this paper.

REFERENCES

Parastoo Abtahi and Griffin Dietz. 2020. Learning Rust: How Experienced Programmers Leverage Resources to Learn a New Programming Language. In *CHI Extended Abstracts*. 1–8. https://doi.org/10.1145/3334480.3383069

Jonathan Aldrich and Craig Chambers. 2004. Ownership domains: Separating aliasing policy from mechanism. In *ECOOP*, Vol. 4. Springer, 1–25. https://doi.org/10.1007/978-3-540-24851-4_1

J. Armstrong. 2007. A History of Erlang. In *HOPL III*. https://doi.org/10.1145/1238844.1238850

Ellen Arvidsson, Elias Castegren, Sylvan Clebsch, Sophia Drossopoulou, James Noble, Matthew J. Parkinson, and Tobias Wrigstad. 2023. Reference Capabilities for Flexible Memory Management: Extended Version. *arXiv:2309.02983 [cs.PL]* (10 2023). https://doi.org/10.48550/arXiv.2309.02983

Henry G. Baker. 1990. Unify and Conquer (Garbage, Updating, Aliasing,...) in Functional Languages. In *LISP and Functional Programming*. 218–226. https://doi.org/10.1145/91556.91652

Aria Beingessner. 2015. *You can't spell Trust without Rust.* Master's thesis. Computer Science, Carleton University.

Aria Beingessner. 2019. Learn Rust With Entirely Too Many Linked Lists. `https://rust-unofficial.github.io/-too-many-lists`. Accessed 2023-04-14.

Eli Bendersky. 2021. Rust data structures with circular references. `eli.thegreenplace.net/2021/rust-data-structures-with-circular-references/`.

David Blaser. 2019. Simple Explanation of Complex Lifetime Errors in Rust. (2019). Bachelor Thesis, ETH Zürich.

Robert Bocchino. 2011. Deterministic Parallel Java. In *Encyclopedia of Parallel Computing*. 566–573. https://doi.org/10.1007/978-0-387-09766-4_119

Gregory Bollella, Tim Canham, Vanessa Carson, Virgil Champlin, Daniel L. Dvorak, Brian Giovannoni, Mark B. Indictor, Kenny Meyer, Alex Murray, and Kirk Reinholtz. 2003. Programming with non-heap memory in the real time specification for Java. In *OOPSLA Companion*. 361–369. https://doi.org/10.1145/949344.949443

Chandrasekhar Boyapati and Martin Rinard. 2001. A Parameterized Type System for Race-Free Java Programs. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*. Association for Computing Machinery, New York, NY, USA, 56–69. https://doi.org/10.1145/504282.504287

Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, and Martin Rinard. 2003. Ownership Types for Safe Region-Based Memory Management in Real-Time Java. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. New York, NY, USA, 324–337. https://doi.org/10.1145/781131.781168

John Boyland. 2001. Alias burying: Unique variables without destructive reads. *Software: Practice and Experience* 31, 6 (2001), 533–553. https://doi.org/10.1002/spe.370 Publisher: Wiley.

John Boyland. 2013. Fractional Permissions. In *Aliasing in Object-Oriented Programming*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Springer-Verlag, Berlin, Heidelberg, 270–288. https://doi.org/10.1007/978-3-642-36946-9_10

John Boyland, James Noble, and William Retert. 2001. Capabilities for Sharing. In *Proceedings of the 15th European Conference on Object-Oriented Programming ECOOP, Budapest, Hungary, June 18-22, 2001*. Springer Berlin Heidelberg, 2–27. https://doi.org/10.1007/3-540-45337-7_2

Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. 2022. Effects, Capabilities, and Boxes: From Scope-Based Reasoning to Type-Based Reasoning and Back. *Proc. ACM Program. Lang.*, Article 76 (apr 2022), 30 pages. https://doi.org/10.1145/3527320

Nicholas Cameron. 2015. What's the "best" way to implement a doubly-linked list in Rust? `http://featherweightmusings.blogspot.com/2015/04/graphs-in-rust.html`. Accessed 2023-04-14.

Elias Castegren and Tobias Wrigstad. 2016. Reference Capabilities for Concurrency Control. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*. 5:1–5:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.5

Elias Castegren and Tobias Wrigstad. 2017. Relaxed Linear References for Lock-free Data Structures. In *Proceedings of the 31st European Conference on Object-Oriented Programming ECOOP (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 74)*. 6:1–6:32. https://doi.org/10.4230/LIPIcs.ECOOP.2017.6 ISSN: 1868-8969.

Luke Cheeseman, Matthew J. Parkinson, Sylvan Clebsch, Marios Kogias, Sophia Drossopoulou, David Chisnall, Tobias Wrigstad, and Paul Liétar. 2023. When Concurrency Matters. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (10 2023). https://doi.org/10.1145/3622852

David Clarke. 2001. *Object Ownership and Containment.* Ph.D. Dissertation. University of New South Wales.

Dave Clarke and Sophia Drossopoulou. 2002. Ownership, Encapsulation and the Disjointness of Type and Effect. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Seattle, Washington, USA) *(OOPSLA '02)*. Association for Computing Machinery, New York, NY, USA, 292–310. https://doi.org/10.1145/582419.582447

Dave Clarke and Tobias Wrigstad. 2003. External Uniqueness Is Unique Enough. In *ECOOP 2003 – Object-Oriented Programming*, Luca Cardelli (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 176–200. https://doi.org/10.1007/978-3-540-45070-2_9

David Clarke, Tobias Wrigstad, and James Noble. 2013. *Aliasing in Object-oriented Programming: Types, Analysis and Verification*. Vol. 7850. Springer. https://doi.org/10.1007/978-3-642-36946-9

Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. 2008. Minimal Ownership for Active Objects. In *Programming Languages and Systems*, G. Ramalingam (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 139–154. https://doi.org/10.1007/978-3-540-89330-1_11

David G. Clarke, John M. Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vancouver, British Columbia, Canada) *(OOPSLA '98)*. Association for Computing Machinery, New York, NY, USA, 48–64. https://doi.org/10.1145/286936.286947

Sylvan Clebsch and Sophia Drossopoulou. 2013. Fully Concurrent Garbage Collection of Actors on Many-core Machines. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. 553–570. https://doi.org/10.1145/2509136.2509557

Sylvan Clebsch, Juliana Franco, Sophia Drossopoulou, Albert Mingkun Yang, Tobias Wrigstad, and Jan Vitek. 2017. Orca: GC and Type System Co-design for Actor Languages. *PACMPL* 1, OOPSLA (2017), 72:1–72:28. https://doi.org/10.1145/3133896

Cliff Click, Gil Tene, and Michael Wolf. 2005. The Pauseless GC Algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE '05)*. ACM, New York, NY, USA, 46–56. https://doi.org/10.1145/1064979.1064988

Michael Coblenz, Michelle L. Mazurek, and Michael Hicks. 2022. Garbage Collection Makes Rust Easier to Use: A Randomized Controlled Trial of the Bronze Garbage Collector. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1021–1032. https://doi.org/10.1145/3510003.3510107

Russell Cohen. 2018. Why Writing a Linked List in (safe) Rust is So Damned Hard. `https://rcoh.me/posts/rust-linked-list-basically-impossible/`. Accessed 2023-04-14.

CWE 2022. 2021 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html.

David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-First Garbage Collection. In *Proceedings of the 4th International Symposium on Memory Management* (Vancouver, BC, Canada) *(ISMM '04)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/1029873.1029879

Werner Dietl, Sophia Drossopoulou, and Peter Müller. 2007. Generic Universe Types. In *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*. 28–53. https://doi.org/10.1007/978-3-540-73589-2_3

Tamar Domani, Gal Goldshtein, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. 2002. Thread-Local Heaps for Java. In *Proceedings of the 3rd International Symposium on Memory Management* (Berlin, Germany) *(ISMM '02)*. Association for Computing Machinery, New York, NY, USA, 76–87. https://doi.org/10.1145/512429.512439

Martin Elsman and Niels Hallenberg. 2021. Integrating region memory management and tag-free generational garbage collection. *J. Funct. Program.* 31 (2021), e4. https://doi.org/10.1017/S0956796821000010

Kiko Fernandez-Reyes, Isaac Oscar Gariano, James Noble, Erin Greenwood-Thessman, Michael Homer, and Tobias Wrigstad. 2021. Dala: A Simple Capability-Based Dynamic Language Design For Data Race-Freedom. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Chicago, IL, USA) *(Onward! 2021)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/3486607.3486747

Cormac Flanagan and Stephen N. Freund. 2000. Type-Based Race Detection for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 219–232. https://doi.org/10.1145/349299.349328

Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Lugano, Switzerland, August 29 - September 2, 2016*. 13:1–13:9. https://doi.org/10.1145/2972206.2972210

Matthew Fluet and Greg Morrisett. 2006. Monadic regions. *J. Funct. Program.* 16, 4-5 (2006), 485–545. https://doi.org/10.1145/1016848.1016867

Matthew Fluet, Greg Morrisett, and Amal J. Ahmed. 2006. Linear Regions Are All You Need. In *ESOP*, Vol. 3924. 7–21. https://doi.org/10.1007/11693024_2

Matthew Francis-Landau, Bing Xue, Jason Eisner, and Vivek Sarkar. 2016. Fine-grained parallelism in probabilistic parsing with Habanero Java. In *IA3 '16: Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms*. IEEE Press, Piscataway, NJ, USA, 78–81. https://doi.org/10.1109/IA3.2016.020

Juliana Franco, Sylvan Clebsch, Sophia Drossopoulou, Jan Vitek, and Tobias Wrigstad. 2018. Correctness of a fully concurrent Garbage Collector for Actor Languages. In *European Symposium on Programming (ESOP)*, Vol. 10801. https://doi.org/10.1007/978-3-319-89884-1_31

David Gay and Alex Aiken. 1998. Memory Management with Explicit Regions. *SIGPLAN Not.* 33, 5 (may 1998), 313–323. https://doi.org/10.1145/277652.277748

David Gay and Alex Aiken. 2001. Language Support for Regions. *SIGPLAN Not.* 36, 5 (may 2001), 70–80. https://doi.org/10.1145/381694.378815

Laure Gonnord, Ludovic Henrio, Lionel Morel, and Gabriel Radanne. 2023. A Survey on Parallelism and Determinism. *ACM Comput. Surv.* 55, 10, Article 210 (feb 2023), 28 pages. https://doi.org/10.1145/3564529

Colin S. Gordon. 2020. Designing with Static Capabilities and Effects: Use, Mention, and Invariants (Pearl). In *ECOOP*, Robert Hirschfeld and Tobias Pape (Eds.). 10:1–10:25. https://doi.org/10.4230/LIPIcs.ECOOP.2020.10

Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and reference immutability for safe parallelism. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*. 21–40. https://doi.org/10.1145/2384616.2384619

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-based memory management in Cyclone. *ACM Sigplan Notices* 37, 5 (2002), 282–293. https://doi.org/10.1145/543552.512563 Publisher: ACM.

Olivier Gruber and Fabienne Boyer. 2013. Ownership-Based Isolation for Concurrent Actors on Multi-core Machines. In *ECOOP 2013 – Object-Oriented Programming*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 281–301. https://doi.org/10.1007/978-3-642-39038-8_12

Philipp Haller and Alexander Loiko. 2016. LaCasa: lightweight affinity and object capabilities in Scala. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. 272–291. https://doi.org/10.1145/2983990.2984042

Philipp Haller and Martin Odersky. 2010. Capabilities for Uniqueness and Borrowing. In *Proceedings of 24th European Conference on Object-Oriented Programming, ECOOP, June 21-25*. https://doi.org/10.1007/978-3-642-14107-2_17

Douglas E. Harms and Bruce W. Weide. 1991. Copying and Swapping: Influences on the Design of Reusable Software Components. *IEEE Trans. Softw. Eng.* 17, 5 (May 1991), 424–435. https://doi.org/10.1109/32.90445 Publisher: IEEE Press.

Michael W. Hicks, J. Gregory Morrisett, Dan Grossman, and Trevor Jim. 2004. Experience with safe manual memory-management in Cyclone. In *ISMM*. 73–84. https://doi.org/10.1145/1029873.1029883

John Hogg. 1991. Islands: Aliasing Protection in Object-Oriented Languages. In *Conference proceedings on Object-oriented programming systems, languages, and applications - OOPSLA '91*. ACM Press, 271–285. https://doi.org/10.1145/117954.117975

J. Hogg, D. Lea, A. Wills, D. de Champeaux, and R. Holt. 1992. The Geneva Convention on the Treatment of Object Aliasing. *OOPS Messenger* 3, 2 (April 1992). https://doi.org/10.1145/130943.130947

Vivian Hu. 2020. Rust Breaks into TIOBE Top 20 Most Popular Programming Languages. (June 2020). InfoQ.

Daniel H. Ingalls. 1981. Design Principles Behind Smalltalk. *BYTE* 6, 8 (August 1981), 286–298.

Richard Jones, Antony Hosking, and Eliot Moss. 2016. *The garbage collection handbook: the art of automatic memory management*. CRC Press.

Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked Borrows: An Aliasing Model for Rust. *Proc. ACM Program. Lang.* 4, POPL, Article 41, 32 pages. https://doi.org/10.1145/3371109

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *PACMPL* 2, POPL, Article 66 (Jan. 2017), 66:1–66:34 pages. https://doi.org/10.1145/3158154

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2020. Safe Systems Programming in Rust: The Promise and the Challenge. *Communications of the ACM* (2020). https://doi.org/10.1145/3418295

Steve Klabnik and Carol Nichols. 2019. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press.

Paul Krill. 2021. Microsoft forms Rust language team. (Feb. 2021). InfoWorld.

Butler Lampson, Jim Horning, Ralph London, Jim Mitchell, and Gerry Popek. 1977. Report on the Programming Language Euclid. *ACM Sigplan Notices* 12, 3 (March 1977), 18–79.

Doug Lea. 1998. *Concurrent Programming in Java* (2nd ed.). Addison-Wesley.

Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. 1993. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley.

Paley Li, Nicholas Cameron, and James Noble. 2012. Sheep Cloning with Ownership Types. In *Foundations of Object-Oriented Programming Languages (FOOL)*. ACM.

Per Lidén and Stefan Karlsson. 2018. The Z Garbage Collector—Low Latency GC for OpenJDK. http://cr.openjdk.java.net/pliden/slides/ZGC-Jfokus-2018.pdf

Karl J. Lieberherr and Ian Holland. 1989. Assuring Good Style for Object-Oriented Programs. *IEEE Software* September (1989), 38–48. https://doi.org/10.1109/52.35588

Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. 2015. Use at Your Own Risk: The Java Unsafe API in the Wild. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 695–710. https://doi.org/10.1145/2814270.2814313

Mae Milano, Joshua Turcotti, and Andrew C. Myers. 2022. A flexible type system for fearless concurrency. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 458–473. https://doi.org/10.1145/3519939.3523443

Peter Müller and Arnd Poetzsch-Heffter. 1999. Universes: A type system for controlling representation exposure. In *Programming Languages and Fundamentals of Programming*, Vol. 263. Technical Report 263, Fernuniversität Hagen. https://doi.org/10.1007/BFb0054091

ndrewxie. 2019. What's the "best" way to implement a doubly-linked list in Rust? `https://users.rust-lang.org/t/-whats-the-best-way-to-implement-a-doubly-linked-list-in-rust/27899/7`. Accessed 2023-04-14.

James Noble, Julian Mackay, and Tobias Wrigstad. 2022. Rusty Links in Local Chains. In *FTfJP*.

James Noble, Jan Vitek, and John Potter. 1998. Flexible Alias Protection. In *ECOOP'98 — Object-Oriented Programming*, Eric Jul (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 158–185.

James Noble and Charles Weir. 2000. *Small Memory Software: Patterns for Systems with Limited Memory*. Addison-Wesley.

Liam O'Connor, Zilin Chen, Christine Rizkallah, Vincent Jackson, Sidney Amani, Gerwin Klein, Toby Murray, Thomas Sewell, and Gabriele Keller. 2021. Cogent: uniqueness types and certifying compilation. *Journal of Functional Programming* 31 (2021), e25. https://doi.org/10.1017/S095679682100023X

Filip Pizlo, Antony L. Hosking, and Jan Vitek. 2007. Hierarchical Real-Time Garbage Collection. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '07)*. Association for Computing Machinery, New York, NY, USA, 123–133. https://doi.org/10.1145/1254766.1254784

J. Potter, J. Noble, and D. Clarke. 1998. The Ins and Outs of Objects. In *ASWEC*.

Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding memory and thread safety practices and issues in real-world Rust programs. In *PLDI*. 763–779. https://doi.org/10.1145/3385412.3386036

Ryan James Spencer. 2020. Four Ways To Avoid The Wrath Of The Borrow Checker. (2020). justanotherdot.com.

S. Srinivasan and A. Mycroft. 2008. Kilim: Isolation-Typed Actors for Java. In *ECOOP*. https://doi.org/10.1007/978-3-540-70592-5_6

Gil Tene, Balaji Iyengar, and Michael Wolf. 2011. C4: The Continuously Concurrent Compacting Collector. In *Proceedings of the International Symposium on Memory Management (ISMM '11)*. ACM, New York, NY, USA, 79–88. https://doi.org/10.1145/1993478.1993491

Tiobe 2022. TIOBE Index for June 2022. https://www.tiobe.com/tiobe-index/.

Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. 2004. A Retrospective on Region-Based Memory Management. *Higher Order Symbolic Computing* 17, 3 (2004), 245–265. https://doi.org/10.1023/B:LISP.0000029446.78563.a4

Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. 2021. *Programming with Regions in the MLKit (Revised for Version 4.6.0)*. Technical Report. Department of Computer Science, University of Copenhagen, Denmark.

Mads Tofte and Jean-Pierre Talpin. 1997. Region-based Memory Management. *Inf. Comput.* 132, 2 (1997), 109–176. https://doi.org/10.1006/inco.1996.2613

Aaron Turon. 2015. Fearless Concurrency with Rust. https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html

Mark Utting. 1995. Reasoning about aliasing. In *Fourth Australasian Refinement Workshop*.

Philip Wadler. 1990. Linear types can change the world. In *IFIP TC*, Vol. 2. Citeseer, 347–359.

Tobias Wrigstad. 2006. *Ownership-Based Alias Management*. Ph.D. Dissertation. Royal Institute of Technology, Stockholm.

Albert Mingkun Yang and Tobias Wrigstad. 2017. Type-Assisted Automatic Garbage Collection for Lock-Free Data Structures. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management* (Barcelona, Spain) *(ISMM 2017)*. Association for Computing Machinery, New York, NY, USA, 14–24. https://doi.org/10.1145/3092255.3092274

Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. 2021. GhostCell: separating permissions from data in Rust. In *ICFP*. https://doi.org/10.1145/3473597

Received 2023-04-14; accepted 2023-08-27