Dynamic Region Ownership for Concurrency Safety

FRIDTJOF PEER STOLDT, Uppsala University, Sweden
BRANDT BUCHER, Microsoft, USA
SYLVAN CLEBSCH, Microsoft, UK
MATTHEW A. JOHNSON, Microsoft, UK
MATTHEW J. PARKINSON, Microsoft, UK
GUIDO VAN ROSSUM, Microsoft, USA
ERIC SNOW, Microsoft, USA
TOBIAS WRIGSTAD, Uppsala University, Sweden

The ways in which the components of a program interact with each other in a concurrent setting can be considerably more complex than in a sequential setting. The core problem is unrestricted shared mutable state. An alternative to unrestricted shared mutable state is to restrict the sharing using Ownership. Ownership can turn what would have been a race into a deterministic failure that can be explained to the programmer. However, Ownership has predominantly taken place in statically typed languages.

In this paper, we explore retrofitting an existing dynamically typed programming language with an ownership model based on regions. Our core aim is to provide safe concurrency, that is, the ownership model should provide deterministic dynamic failures of ownership that can be explained to the programmer. We present a dynamic model of ownership that provides ownership of groups objects called regions. We provide dynamic enforcement of our region discipline, which we have implemented in a simple interpreter that provides a Python-like syntax and semantics, and report on our first steps into integrating it into an existing language, Python.

CCS Concepts: • **Software and its engineering → General programming languages**; • **Computing methodologies → Concurrent programming languages**.

Additional Key Words and Phrases: Ownership, Concurrency Safety, Dynamic languages, Python

ACM Reference Format:
Fridtjof Peer Stoldt, Brandt Bucher, Sylvan Clebsch, Matthew A. Johnson, Matthew J. Parkinson, Guido van Rossum, Eric Snow, and Tobias Wrigstad. 2025. Dynamic Region Ownership for Concurrency Safety. *Proc. ACM Program. Lang.* 9, PLDI, Article 210 (June 2025), 26 pages. https://doi.org/10.1145/3729313

1 Introduction

When a sequential program or programming language graduates to concurrent execution, the process can be painful [52, 59]. The ways in which the components of a program interact with each other in a concurrent setting can be considerably more complex than in a sequential setting. Consider a simple example of the `append` method on a `List` in Python. The Python representation has an object that contains a length, a capacity, and a pointer to a block of memory that contains

Authors' Contact Information: Fridtjof Peer Stoldt, Information Technology, Uppsala University, Uppsala, Sweden, fridtjof.stoldt@it.uu.se; Brandt Bucher, Microsoft, Redmond, WA, USA, brandtbucher@microsoft.com; Sylvan Clebsch, Azure Research, Microsoft, Cambridge, UK, sylvan.clebsch@microsoft.com; Matthew A. Johnson, Azure Research, Microsoft, Cambridge, UK, mattpark@microsoft.com; Matthew J. Parkinson, Azure Research, Microsoft, Cambridge, UK, mattpark@microsoft.com; Guido van Rossum, Microsoft, Redmond, WA, USA, gvanrossum@microsoft.com; Eric Snow, Microsoft, Redmond, WA, USA, eric.snow@microsoft.com; Tobias Wrigstad, Information Technology, Uppsala University, Uppsala, Sweden, tobias.wrigstad@it.uu.se.

ACM 2475-1421/2025/6-ART210
https://doi.org/10.1145/3729313

the elements. The append method increases the length of the list by one, and if it does not exceed the current capacity of the list, it writes the new element into the block of memory. If it exceeds the capacity, it allocates a new block of memory, copies the elements from the old block to the new block, and writes the new element into the new block. If we naively use this method in a concurrent setting, then racing calls to append can result in data loss, and even worse, data corruption.

To enable programs to run in a concurrent context, synchronisation must be added to the program. Failure to correctly add synchronisation typically manifests as non-deterministic bugs that are difficult to reproduce, diagnose, and fix. This is not a great programming experience and makes the migration from sequential to concurrent programming a difficult one. The core problem is unrestricted shared mutable state.

An alternative to unrestricted shared mutable state is to restrict the sharing. This restriction can be a shared-nothing approach such as JavaScript's WebWorkers or Erlang's Actors, or more elaborate approaches often described as *Ownership*. Many forms of ownership have been proposed over the years [2, 6, 16, 18, 26, 48] with multiple versions being developed to provide concurrency safety [11, 17, 26, 51, 56]. Ownership for concurrency safety can turn what would have been a data race into a deterministic failure that can be explained to the programmer. However, with few exceptions [23, 32, 50], Ownership has predominantly been explored in statically typed languages.

But what happens if you are in a dynamically typed language? Can we enforce ownership dynamically to provide safe concurrency? How do we surface ownership without a static type system to carry the ownership information? Can we add ownership to an existing language like Python to enable a seamless migration from sequential to concurrent execution of programs?

In this paper, we explore retrofitting an existing dynamically typed programming language, like Python, with a region-based ownership model building on Arvidsson et al.'s Reggio [4]. Our core aim is to provide safe concurrency [11, 17, 26, 56], that is, the ownership model should provide deterministic dynamic failures of ownership that can be explained to the programmer, rather than allowing racing accesses to potentially corrupt sequential invariants of data structures.

Contributions. We present a dynamic model of ownership called *Lungfish* (§ 2) that provides ownership of groups of objects called *regions*. We use a novel notion of a *local* region that captures existing sequential code without modification, and provides a dynamic notion of *borrowed* references that allow objects in the local region to refer into other regions. This approach is integrated with deeply *immutable* objects and *cowns*, concurrent owners [13], to allow for safe sharing of objects between regions, while protecting the programmer from data races.

We provide dynamic enforcement of our region discipline (§ 3). Our implementation uses a write-barrier to enforce that the region discipline is obeyed as the object graph is updated. For legacy sequential code, this introduces a couple of untaken conditional branches for each write to a field. For code using regions it prevents connecting regions in ways that would potentially lead to data races. To rapidly prototype and explore our ownership model, we have developed a simple interpreter that provides a Python-like syntax and semantics (§ 4). The prototype outputs graphical traces of the changes to, and enforcement of, the ownership model.

We report on our first steps of integrating Lungfish into Python. This has led to several interesting challenges (§ 5). As this paper shows, many of the challenges stem from the pragmatics of programming language design and implementation, and we expect our insights to be as useful as they are surprising. To evaluate our design, we compare it to our goals for Lungfish (§ 7). As our work is ongoing towards a complete Python prototype, not all our goals can yet be empirically evaluated. However, we provide qualitative arguments for why we believe they will be met.

Why Python? Python was designed as a single-threaded language with a *global interpreter lock* (GIL) that ensures that only one thread at a time executes Python bytecode in the interpreter [61].

Concurrency was introduced through threading, allowing developers to write concurrent code, but without true parallelism thanks to the GIL. To support true parallelism, Python recently added support for "subinterpreters"—multiple threads each running its own isolated interpreter with its own GIL [54], inside a single Python process. Python's multiple subinterpreters is an example of share nothing. Communication between subinterpreters is necessarily by copy as different subinterpreters run their own memory managers and can only manage objects they themselves created. Shared-nothing approaches permits sequential reasoning inside each subinterpreter (in the case of Python) with clearly demarcated interaction points with other threads.

Recently, development has begun [65] on "free-threaded Python" which removes the GIL in a single interpreter [28]. With the GIL removed, free-threaded Python trades sequential reasoning away for increased performance enabled by permitting threads to operate directly on shared state. Where the shared state is mutable, data races are possible. Work on free-threaded Python is concerned with ensuring that such data-races cannot compromise the integrity of the runtime, but does not prevent data-races in Python code. When work on free-threaded Python is done, Python will have joined the most common approach to concurrency and parallelism: threads with shared mutable state.

We are at a point where Python graduates from sequential to concurrent execution. We believe that using dynamic ownership can reduce the challenges for the programmer in this process.

2 Growing an Ownership Model for Safe Concurrency

Let us set the stage by enumerating our design goals, which we use to both motivate and evaluate our design choices. Only the first five goals are in the scope of this paper, but without stating the remaining, some design decisions would be inexplicable. The goal of this work is to retrofit Python, an existing, "morally sequential" dynamic programming language, with concurrency in a way that is (G1) safe from data races. In addition to data-race safety we want backwards compatibility for existing programs: (G2) existing sequential Python programs should work without modification, and (G3) Python libraries which are morally safe in a multithreaded setting should be usable when multithreading has been added with little or no effort by the library maintainers. Our design should (G4) embrace the value set of dynamic languages (*e.g.*, delay errors until they occur, reject as few programs as possible), and (G5) introduce a low number of new syntactic elements to the Python language. We also strive to (G6) deliver performance — in particular, (G6a) sequential programs should not run considerably slower, although we (G6b) are willing to take a small performance hit for multithreaded programs, speculating that it can be offset against performance increases due to parallelism. It also (G6c) should not disable or counteract existing performance optimisations. Finally, (G7), our design should not add considerable complexity to Python's runtime, to ease the burden of maintaining the language using a small and largely unpaid volunteer work-force.

These goals have been developed while iterating our design over many months with a group of core committers to Python (some of which eventually ended up co-authors of this paper; *c.f.*, § 9). In the coming sections, we evolve our design to meet (most of) them.

2.1 A Notion of Safe Sharing and Ownership Transfer

As an initial model, the single thread of our program has exclusive access to all data, *i.e.*, it owns the data. Our first goal is to organise that data to permit it to be shared safely with other threads, or alternatively to see which data cannot safely be shared. By safe sharing we mean absence of data races and uncoordinated access.

First, we distinguish between immutable and mutable data. The former is always safe to operate on without exclusive access (but *c.f.* § 5.3.5), but the latter requires some form of synchronisation to ensure "effective exclusive access", or ownership transfer.

For ownership transfer to be safe it must not give rise to sharing of mutable data across threads (which would invalidate exclusive access). Transferring an object *a* from one thread to another requires ensuring that the transitive closure of mutable state under *a* that becomes reachable to the receiver is no longer reachable on the sender side (*b* in Fig. 1). Synchronisation has a similar requirement: sharing *a* (without transfer) requires that accesses to the transitive closure of mutable state under *a* is properly guarded, both by the sender and the receiver. Finally, for an immutable object to be safe to share, when object *a* is made immutable (we call this *freezing*), the transitive closure of mutable state under *a* must be made immutable too. Fig. 1 shows the subtleties involved when transfering, sharing and freezing data structures with subobjects.

From the above follows that a programmer must be able to reason about the transitive closure of mutable state from objects in their concurrent programs, regardless of if they want to share them as immutable or mutable objects, or transfer them. If all object references are unique, then *a* automatically dominates all state reachable from *a*, but if more flexible reference structures are permitted, either as part of a data structure (*e.g.*, a doubly-linked list) or as part of an operation (*e.g.*, an iterator over a collection), reasoning can become arbitrarily complicated.

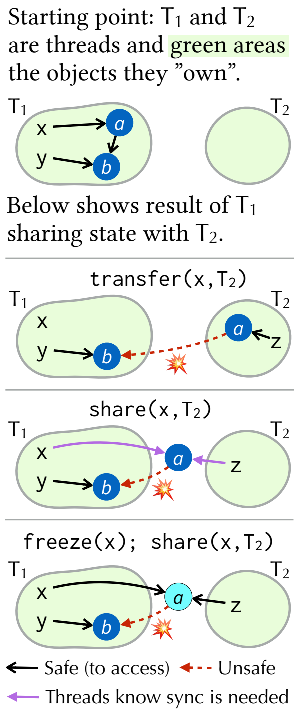

Starting point: T_1 and T_2 are threads and green areas the objects they "own".
T_1
T_2
x
y
a
b
Below shows result of T_1 sharing state with T_2.
transfer(x,T₂)
z
share(x,T₂)
freeze(x); share(x,T₂)
Safe (to access)
Unsafe
Threads know sync is needed

Fig. 1. Dangers when sharing data structures across threads.

2.2 Region-Based Ownership

To decouple ownership from the unit of concurrency, we use *regions* as isolated groups of mutable objects. At any point in time, a thread has exclusive access to the regions it can reach. Each thread has an implicit *local region* associated with it. It contains the thread's stack frames as well as new heap-allocated objects, which can move to other regions. New regions can be created dynamically.

Local regions give an ownership semantics to all existing Python programs: all its data is owned by the thread that created it (G2). To satisfy our performance goal (G6a), assigning between locally-owned objects must be cheap. We revisit this in §7.

With the exception of the local region, every region contains a *bridge object* [32] that reifies the region and acts as a handle to it along with its contained objects. Bridge objects are *externally unique* [15], *i.e.*, at most one of its incoming references may originate from outside the region itself, or the local region. Only externally unique references to bridge objects may cross region boundaries (Fig. 2). This is the case for reference $c \longrightarrow R'$ making R the *parent* of R'. Because x is in the local region, R is *free*. The region topology forms a forest of region trees.

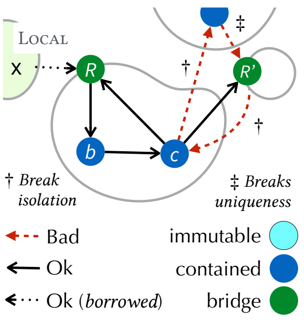


Fig. 2. Region R with objects b, c, bridge R, and nested region R'.

2.2.1 *Assigning Ownership and Borrowing.*

In a statically typed language, an ownership system can be layered on-top of the program as type annotations. In Reggio [4],

```
iso Foo y = new Foo()
```

creates a new `Foo` object which automatically becomes a bridge object for a new region by virtue of the **`iso`** annotation on the type of `y`. If **`iso`** is replaced by **`mut`**, the object created would instead become a "contained object" inside the equivalent of the local region.[1]

In an untyped language, we do not have the ability to layer such metadata on-top of code. Instead, ownership is determined dynamically as a side-effect of a field update, and region creation is unified with bridge object creation. This allows us to use Python's object creation syntax for region creation, in accordance with (G5). Thus, our equivalent of the Reggio code is (on three separate lines)

```
y = Foo() 1    x = Region() 2    x.f = y 3
```

This creates a new `Foo` object `y` in the local region, a bridge object `x` for a fresh region R, and finally moves `y` into R. (See Fig. 3; R, drawn on the region border, is the bridge object and b is `Foo`.)

We treat ownership in the implicitly created local region differently from explicitly created regions. The local region's ownership is *ephemeral*, meaning that as soon as an object in the local region is referenced from another region, we transfer the ownership of the object to the other region. (This is what happens to b on line 3 in Fig. 3.) Transferring an object from the local region will transfer all other objects in the local region that the first object reaches. If a propagating transfer encounters a reference to an object in a third region, the transfer terminates successfully if the referenced object is a bridge object. (This is what happens to $b \longrightarrow R'$ on line 3 in Fig. 3, where R' becomes nested inside R.) If the referent is not a bridge object, the entire transfer fails (and an exception is raised), as the object cannot be moved and the reference breaks region isolation.

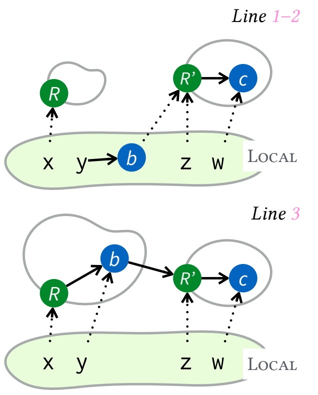


Fig. 3. Interaction between local region and explicit region R.

Just as ephemeral ownership gives up ownership of objects on incoming references, it permits outgoing references without claiming ownership. Any residual references from the local region to an object that is moved into another region becomes a *borrowed* reference. (The $\mathtt{y} \longrightarrow b$ on line 3 of Fig. 3.) Borrowed references are only permitted from the local region (as no other region has ephemeral ownership). Just like inference, this design can lead to unintended object ownership, but unlike inference, this will be discovered at *run-time* (if it conflicts with the rest of the program).

2.2.2 Regions Facilitate Ownership and Transfer. When there are no references to the region (or nested regions) from outside, except to its bridge object, and no references from inside the region (or nested regions) to outside, the region is fully encapsulated — we say that it is *closed*. This is the case for both regions in Fig. 2. As a closed region is dominated by the single reference to its bridge object, the holder of such a reference has exclusive access to — owns — the region and its contents. Safe transfer of a closed region to another thread only requires the proper move of the single reference to the bridge object. Thus, it suffices to reason about a single reference to guarantee absence of residual aliasing for the entire transitive closure of state under it. If the objects a and b in the transfer example in Fig. 1 were created as R and b in Fig. 3, transfer could proceed in two ways: *detect* that the region is not closed and *fail*, or *invalidate* the borrowed reference in `x` and *succeed*.

2.2.3 Regions Facilitate Freezing. Regions help reasoning about the propagation of freezing: freezing a closed region through a path p is guaranteed to not affect other reachable variables or fields. For example in `freeze(x); y.f = 42`, if `x` is closed, then `y` cannot refer to `x` or any other object in the same region, meaning `freeze(x)` cannot cause `y.f = 42` to fail because the object `y` refers to

[1]Reggio does not have a local region, but only one mutable region at a time so **`mut`** `Foo` means the currently mutable region.

has been made immutable. Once a region is frozen, the region itself is no longer needed because immutable objects can be safely shared across threads without any need for exclusive access.

2.2.4 Regions Facilitate Coordinating Concurrent Access to Shared Data. So far, we have shown that regions facilitate concurrent programming with objects which are *mutable but not shared* or *shared but not mutable.* To simultaneously support mutability and sharing we must add a dynamic component to govern who has exclusive ownership at any given point. To this end we add a new object kind: the *cown*, short for "concurrent owner" due to Cheeseman et al. [13].

Cowns are cells that hold a reference to a bridge object (or another cown or immutable object). Cowns offer mutability without topological restrictions. Instead, they cannot be accessed immediately: a thread that wants to access a cown must first *acquire* it. An access to a cown by a thread that has not acquired the cown throws an exception. We leave the semantics of acquisition outside of this paper, but it may be a lock-like *synchronous* operation or an message-passing like *asynchronous* operation, depending on the desired concurrency model. In short, exclusive access is mediated by the run-time. A thread that has acquired a cown may transfer ownership of a region stored in the cown to itself along with its objects and nested regions. Before the cown is *released*, the thread may transfer the region (or another value) back into the cown. The local region cannot be transferred because it does not have a bridge object that can be stored in a cown. Cowns are released when created unless they contain any object that is borrowed. In that case, the cown is *pending release* and will become *released* when the final borrow is removed. An acquired cown cannot be released manually, *c.f.*, §2.3.1. When a cown is released, its contained region *must* be closed.

By nesting a region, cowns enable ownership-based structural lock correlation, which ensures that the appropriate locks are held at the time of access [6, 39].

2.3 Summary and Overview of the "Lungfish" Ownership Model

Table 1. Conditions for a reference from a source object S to a target T (denoted $S \longrightarrow T$) in Lungfish. ✓ Permitted (sometimes with additional constraint). ✗ Not permitted. → Triggers transfer of ownership to satisfy E. Additional constraints: E if S and T belong to the same region; U if S's reference to T is the only reference to T originating from outside of T; B the $\longrightarrow$ reference is borrowed.

$S \longrightarrow T$	T				
S	local	immutable	bridge	contained	cown
local	✓	✓	✓^{B}	✓^{B}	✓
immutable	✗	✓	✗	✗	✓
bridge	→	✓	✓$^{E \vee U}$	✓^{E}	✓
contained	→	✓	✓$^{E \vee U}$	✓^{E}	✓
cown	✗	✓	✓^{U}	✗	✓

Each thread implicitly allocates stack frames and objects inside its local region. To safely share (or transfer) objects with (to) other threads, explicit regions must be created and objects from the local region moved into these regions. References across regions must be externally unique references to bridge objects or borrowed references (*i.e.*, originate in the local region). References into the local region cause objects to be transferred to the source region.

We distinguish three kinds of owned objects: *contained objects* which are mutable and encapsulated inside a region; bridge objects which are like contained objects but additionally permit a single incoming reference from outside its encapsulating region; and *local objects* whose ownership can be transferred *implicitly* by an incoming pointer, and hold borrowed references.

As ownership means exclusive access, owned objects can never be *reachable* by more than a single thread at any given point. They are thus safe to access *immediately* without any need for synchronising with anyone. In other words, they permit sequential reasoning. Additionally, we distinguish two kinds of un-owned objects: immutable objects and cowns. As they are not owned, they can be freely shared in the system. As immutable objects are guaranteed to not change, they enjoy the same immediate access and sequential reasoning as owned objects. Cowns however must be acquired before being accessed, which guarantees exclusive access dynamically.

Table 1 shows all the reference constraints of our system, which we call Lungfish[2].

In addition to the above, Lungfish defines a number of functions operating on entire regions (via bridge objects b), paths ($p ::= \texttt{x} \mid p\,.\,\texttt{f}$), cowns ($c$), immutable objects ($i$), or values ($v ::= b \mid c \mid i$):

- `is_closed`(b) Returns true if b is closed, else false.
- `freeze`(b) Makes the contents of a region (tree) deeply immutable and removes the region(s).
- `merge`(b_1, b_2) Merges free region b_1 into region b_2 and returns b_1 as a contained object. If the second argument is omitted, b_1 is merged into the local region, essentially resetting ownership.
- `get`(c) and `set`(c, v) read and update the value of an acquired cown. The `get`(c) returns the value and `set`(c, v) installs the new value and returns the previous value.
- The keyword **`move`** p Moves the contents of a variable or field. This is the equivalent of a destructive read [7, 32] or a swap [30] with `None` (in the case of Python). A new keyword is motivated as no other Python operators have a side-effect of nullifying a variable or field (G5).

2.3.1 Concurrency Model. Lungfish can be used with different concurrency models such as threads and locks, or behaviour-oriented concurrency [13]. For concreteness, we give all examples following the style of free-threaded Python, *i.e.*, threads and locks. What sets this work apart from free-threaded Python's approach is ownership-based structural lock correlation [39]: it is not possible to access any mutable state unless you have exclusive read and write access to that state. A thread is created in Python by instantiating a `Thread` object and passing it a callable object that implements a `run()` method and arguments to that function. These objects will become accessible by both the *creating* thread and the *created* thread. For this to be safe we require these objects to be transferred, cowns, or immutable, following the three scenarios in Fig. 1. We can spawn a new thread starting in `my_method` with a bridge object `x`, a cown `y`, and an immutable object `z` like so:

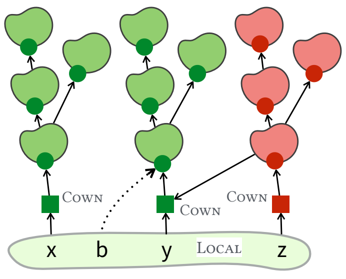
```
t = threading.Thread(target=my_method, kwargs={"p1"=move x, "p2"=y, "p3"=z})
```

```
... # no access to x's or y's regions
b = None
with CS(x, y): # CS is a cown set
# x and y are now acquired
  ... # access to x's and y's regions
  b = get(y)
  ... # or newly created regions
  b = None
# x and y are now released
```

Fig. 4. Acquiring and releasing two cowns using a **`with`** statement.

This creates a new thread t, *moves* the bridge object `x` from the local thread into t, and *shares* the cown `y` and the immutable object `z` with t. The **`move`** invalidates the local variable `x`. When starting the thread by `t.start()` an exception is thrown if any of the constructor arguments are not bridge objects of *closed* regions, cowns, or immutable objects. Without loss of generality, `t.join()` returns nothing as a cown can be used to obtain results from threads (*c.f.*, Fig. 13).

For acquiring and releasing cowns, we use a block-scoped construction that acquires zero or more cowns before the start of the block and automatically releases the cowns at the end. In a synchronous concurrency model, this can be achieved using a context manager [62] that gets called on entry and exit to a **`with`** statement (the `CS` class in Fig. 4).[3] Fig. 5 shows what is accessible



```
with (CS(x, y)): ...
```

Fig. 5. Exclusive access inside the **`with`** in Fig. 4. (Only drawing bridge objects to reduce clutter).

[2]As lungfish both have lungs and gills (*c.f.*, §7.1.2), and resemble snakes (at least to a computer scientist).

[3]Note that **`with`** blocks do not create new scopes in Python. In examples, we assume variables used only inside the **`with`** are removed at the end of the block. This can be achieved with reflection or a combination of decorators and nested functions.

inside the `with CS(x, y)` block, notably not `z`. A dark green cown is acquired and a light green region is exclusively accessible to the acquiring thread. Regions nested in unacquired cowns are dark red. If the `b = None` was removed, the region will escape its acquiring block. However, that means that the region isn't closed, and the release will raise an exception. We will discuss our intended concurrency model and its Lungfish interface in a future paper.

2.3.2 Limitations. The granularity and flexibility of Lungfish's concurrency is controlled by the heap structure. It is not possible to operate concurrently on two objects that reference each other as those objects must be inside the same region. A region structure that does not fit an intended parallelism can be dealt with by merging the region into the local region, which permits its objects to be moved to new regions which can subsequently be operated on in parallel. Whether such overhead is acceptable will vary from case to case. Parallel operations on arrays require that cells are cowns. As long as an array's elements do not overlap, we can turn an array into disjoint regions á la Arr-O-Matic [68].

3 Dynamic Ownership Enforcement

Next we explain how to check for ownership violations dynamically. As a result of design iteration with maintainers of dynamic programming languages, there has been a constant shift towards delaying ownership checking closer to the time of error (G4). In the end, we treat the act of trying to make a structure that would make data-races possible visible to other threads as an error, rather than raising the error only if the data races are observed. Thus, errors are caused by the actions of a single thread, not by several threads operating on shared data.

Motivated by (G4), we require region-based ownership to be enforced only when data is transferred or shared, that is when spawning a new thread and releasing a cown. In Lungfish, this requires that the region is closed. If the region is open, then the errors outlined in Fig. 1 are possible, *i.e.*, mutable state becoming directly reachable from multiple threads. Note that we raise an exception when sharing occurs, not if a data race actually takes place on that shared state.

3.1 Tracking Ownership

In Lungfish, every object knows its owning region using a field `owner`. It is initially set to Local, which represents that the object is in the local region. When a local object is moved into another region, we update the object's `owner` field to reference the region r. For immutable, and cown, objects the `owner` field to Imm, and Cown, respectively. Lungfish distinguishes between shared, inter-region and intra-region references. Shared references are references to immutable and cown objects, and are always allowed. Intra-region references are always allowed and do not need to be checked. Inter-region references are split into two types of reference: *borrowed* and *external.* A borrowed reference is any inter-region reference starting in the local region. Note that all stack variables are borrowed references as they are fields of a frame object in the local region. All other inter-region references are external references and must satisfy the *external uniqueness* constraint.

External uniqueness requires that there is at most one external reference to each region and that the target of the reference is a bridge object. Regions have a `parent` attribute which is either `None` (when the region is a root of a region tree), Cown (when the region is stored in a cown), or the parent attribute is the enclosing region. Regions initially do not have a `parent` region, and are *captured* when their bridge object is referenced from a region other than itself or the local region. Capturing an already captured region throws an exception. To make a captured region free, we simply remove the external reference to the bridge object, which updates its `parent` to `None`.

We track a region's incoming borrowed references through a *local reference count* (LRC). The LRC allows us to efficiently check if a region is closed: when the LRC is `0` the region is closed.

Note that this means that regions referenced from the stack (or fields in the local region) cannot be closed, which is intentional: they are directly reachable from the current thread, and closed implies they are safe to transfer or share.

We also track the existence of borrowed references to child regions (aka nested regions). When we increase the local reference count of a child region, if it moves from an LRC of zero to one, we increase the LRC of the parent region. Similarly, when it moves from one to zero, we decrease the LRC of the parent region. This is done recursively up the region tree. This ensures that a region is not closed if there are borrowed references to any of its child regions.

3.2 Implementation

The core challenge of the implementation is to dynamically enforce the ownership constraints in Table 1 efficiently. We enforce these constraints through a write-barrier, that is, some additional logic triggered on stores to fields, assignment to local variables and binding of arguments to formal parameters. As we model local variables and formal parameters as fields of a frame object it suffices to just implement the write-barrier for fields.

To track the correct usage of regions, the write-barrier must maintain three things: (1) the region that an object belongs to, (2) the local reference count, and (3) the parent region of the region, if there is one. The first is represented by the `owner` field of each object, which refers to the region that owns the object. The second and third values are represented using two special fields that are stored in the region: `lrc` and `parent`, respectively. It also contains a field that references the `bridge` object. The region is a separate allocation to the bridge object, which simplifies handling cases where the bridge becomes unreachable, but there are still borrowed references into the region. The separate allocation allows the bridge to be collected as soon as it becomes unreachable without needing to update the `owner` of still reachable objects.

Let us consider a store that updates the field f of an object `src` with the value `tgt`, where the field previously contained `oldtgt`. The store triggers a write-barrier shown in Fig. 6. The first thing the write-barrier must do is to check if the assignment is allowed by checking that the region of the updated object `src` is not Imm or Cown. If the store is allowed, the write barrier updates the region topology for the new edge from `src` to `tgt`, and removes the old edge from `src` to `oldtgt`. In Fig. 6, we define two operations on the region topology for this purpose: `addReference` and `removeReference`. See Fig. 7 for a graphical example of the barrier in use.

The `removeReference` operation removes a reference from `src` to `oldtgt`. If `src` and `oldtgt` are in the same region or `oldtgt` is a cown or immutable, then there are no region topology updates. If `tgt` is a cown or immutable, then the reference count is decreased atomically, and in all other cases, it is decreased non-atomically. If the reference is borrowed, that is, from the local region into another region, then the local reference count of the region owning `oldtgt` is decremented. The final case handles instances where `src` and `oldtgt` are in different regions. From the external uniqueness constraint, we know `oldtgt` is a bridge object, and `src` contains the only owning reference to `oldtgt` from outside of `oldtgt`'s region. The code removes the parent pointer from the returned heap.

The `addReference` operation adds a reference from `src` to `tgt`. This follows a similar pattern to `removeReference`. If `src` and `tgt` are in the same region or `tgt` is a cown or immutable, then there are no region topology updates. If `tgt` is a cown or immutable, then the reference count is increased atomically, and in all other cases it is increased non-atomically. If the new reference is from the local region into another region, a borrowed reference, then the local reference count of the region owning `tgt` is incremented. Otherwise, we need to move the `tgt` object, and all the local objects it references, into the region owning `src`. This may fail, if it does not obey the region discipline.

The `addToRegion` operation is responsible for updating the local reference count for the region that is being moved into. It assumes reference counted memory management for individual objects.

```
def writeBarrier(src, oldtgt, tgt):
  if src.owner in {Imm, Cown}:
    raise Error
  addReference(src, tgt)
  removeReference(src, oldtgt)

def addReference(src, tgt):
  # Check if in same region
  if src.owner is tgt.owner:
    return incRC(tgt)
  # Check if tgt is a cown or immutable
  if tgt.owner in {Imm, Cown}:
    return incRCAtomic(tgt)
  incRC(tgt)
  # Check if borrowed reference
  if src.owner is Local:
    return incLRC(tgt.owner)
  # Add object to a region (may fail)
  addToRegion(tgt, src.owner)

def removeReference(src, oldtgt):
  # Check if in same region
  if src.owner is oldtgt.owner:
    return decRC(oldtgt)
  # Check if oldtgt is a cown or immutable
  if oldtgt.owner in {Imm, Cown}:
    return decRCAtomic(oldtgt)
  reg = oldtgt.owner
  decRC(oldtgt)
  # Check if borrowed reference
  if src.owner is Local:
    return decLRC(reg)
  # Handle removing parent of region.
  return removeParent(reg)

def addToRegion(curr, reg):
  if curr.owner is Local:
    # Add references to the object,
    # minus the one we just followed.
    reg.lrc += curr.rc - 1
    # Move into the region
    curr.owner = reg
    # Recursively move the fields...
    # ...Recursively move the fields
    for next in fields(curr):
      addToRegion(next, reg)
    return
  # Account for internal reference
  if curr.owner is reg
    reg.lrc -= 1
    return
  curr_reg = curr.owner
  # Ignore external references
  if curr_reg in {Imm, Cown}:
    return
  # Check for bridge object
  if not curr_reg.bridge is curr:
    raise Error
  # Check not already in a region
  if not curr_reg.parent is None:
    raise Error
  # Check for cycle in region topology
  if curr_reg in parents(region):
    raise Error
  curr_reg.parent = region
```

Fig. 6. Pseudo code for write-barrier. We highlight: operations that may trigger deallocation of multiple objects; operations that **do not** run in $O(1)$; and Python reference count manipulations.

While moving objects into the region, we add to the local reference count the reference count of the objects added to the region and remove a reference count for each internal reference that is created by the move. If `addToRegion` is given an object that is already in a region, then it is responsible for adding a parent to a bridge object. This can fail if the object is not a bridge object, already has a parent, or if adding the parent would create a cycle in the region topology. These checks guarantee that external uniqueness is preserved.

Note that the `addToRegion` operation is not constant time. Its time complexity is proportional to the size of the subgraph being moved combined with the number of parents of the region being moved into. The second component is due to the check for cycles in the region topology. While the first comes from the graph walk. The `incLRC` and `decLRC` operations are also not constant time, because they may need to update the local reference count of the parent regions.[4]

Our implementation assumes that breaking the region topology is an unrecoverable error. To make the error recoverable, we need to implement two versions of `addToRegion`: one that checks the update is valid, and a second that actually performs the update. If added, a double pass will increase the cost of adding objects to a region, but will provide a better programming experience.

3.2.1 Move. The **move** keyword can be implemented by using a temporary variable in the local region to store the value of the field being moved. For example: `y.f = `**`move`**` x.f` can simply be encoded using a fresh temporary variable, `tmp`, as four statements: `tmp = x.f`; `x.f = None`; `y.f = tmp`; `del(tmp)`. This is a simple encoding of the **move** operation that does not require any additional logic in the write-barrier. The line `x.f = None` will result in removing the parent of the region, and `y.f = tmp` will add the parent of the region to the new object. Finally `del(tmp)` removes `tmp`.

[4]We do not expect the region hierarchy to be deep, so the cost of these operations should be low. (Potanin et al. [47] measure average ownership depth to be 5–6 in Java, using a more fine-grained ownership model; we expect less.) If that does not prove to the be case in Python, we can consider a more elaborate data structure to track the local reference count from nested regions lazily. This would make closing a region more expensive, but would make the write-barrier more efficient.

```
1 class Obj: pass # a
2 x = Obj() # b
3 x.g = Obj() # c
4 r = Region() # R
5 r.h = x # trigger transfer
6 x.f = 7 # immutable
7 x = None # immutable
```

Line No. Code	WB	Comment
1	10	local to local
2	10	local to local
3	10	local to local
4	17	borrowed ref.
5(a)	19	add *b* to *R*
5(b)	19	add *a* to *R*
5(c)	19	add *c* to *R*
6	13	7 is immutable
7	14, 31	stop borrowing

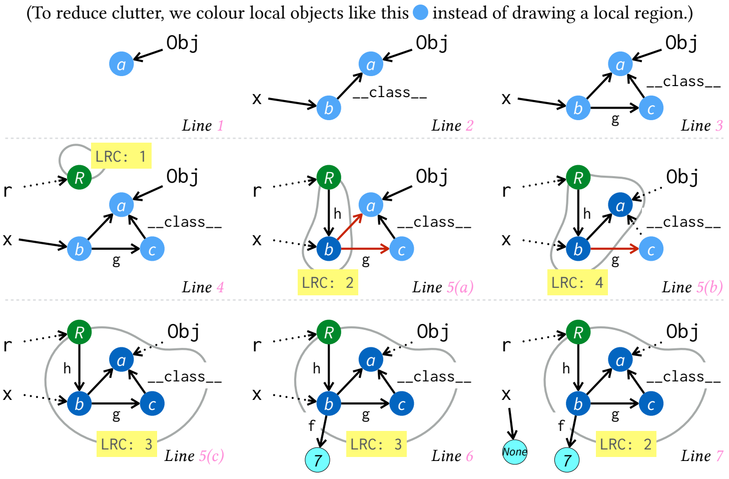


Fig. 7. Example of exercising the write-barrier. The table explains what line of code from the snippet ends in what lines on the write-barrier logic in Fig. 6. Line 1 evaluates to a local name bound to a type object. To reduce clutter, we do not draw the local region or show all __class__ pointers.

4 Prototyping Language Design

$$
\begin{array}{ll}
P ::= b & \mid\ e \\
b ::= s \mid s\ s & \mid\ F \\
F ::= \mathtt{def}\ f(x^*)\colon b & e ::= x \\
L ::= \{\ \} \mid \mathit{str} \mid \mathit{bool} & \mid\ \mathtt{move}\ x \\
s ::= \mathtt{while}\ e\colon b & \mid\ e.f \\
\quad \mid\ \mathtt{for}\ x, x\ \mathtt{in}\ x\colon b & \mid\ x[\mathit{str}] \\
\quad \mid\ \mathtt{if}\ e\colon b & \mid\ x \mathrel{!=} y \\
\quad \mid\ \mathtt{if}\ e\colon b\ \mathtt{else}\colon b & \mid\ x == y \\
\quad \mid\ \mathtt{return}\ e & \mid\ f(e^*) \\
\quad \mid\ e.f = e & \mid\ e.f(e^*) \\
\quad \mid\ x = e & \mid\ L
\end{array}
$$

Fig. 8. Syntax of 𝔉ranken𝔖cript.

To guide our design, we implemented 𝔉ranken𝔖cript, a toy sequential programming language. Its goal is to enable ourselves and the Python community to interact with and understand Lungfish's design space and has been instrumental to explain the semantics of regions and how the write-barrier works.

𝔉ranken𝔖cript implements all of Lungfish's region model explained thus far, including some design alternatives which are described in the next section. Since it is sequential, it does not have **with** blocks, get and set or similar mechanisms. In the future, we will extend 𝔉ranken𝔖cript with a concurrency model, but our priority has been the design of the region ownership model. 𝔉ranken𝔖cript is available as an artefact [57], with a small collection of programs.

When executing a program, the 𝔉ranken𝔖cript interpreter generates a markdown file with one Mermaid diagram[5] for each line that shows the region and object topology of the program, highlighting borrowing and region openedness, etc. This allows us to test and communicate consequences of different designs without biting the bullet of making patches to the Python runtime (which is itself a moving target).

Fig. 8 shows the syntax of 𝔉ranken𝔖cript, which draws syntactically from Python and semantically from Python and JavaScript. It uses Python-style semantic indentation, which is not shown for brevity. Metavariables x and f range over names of variables, fields and functions. *str* denotes a double-quoted string. Literals L are empty objects { }, strings and boolean values. The language is prototype-based to avoid the need for special syntax for classes. Objects are dictionaries born empty; methods are added by storing functions in object fields. Every object has a __proto__ field which can be assigned to any object which will subsequently be used to lookup missing names.

[5]See https://mermaid.js.org/, and also the live editor https://mermaid.live/ to view diagrams.

The built-in function `Region()` creates a fresh region and returns its bridge object. `Cown()` takes a reference to a root of a region tree and stores it in a freshly created cown and returns it. `freeze()` makes the contents of a region immutable. `None` is a built-in immutable object. Mutable objects are shown in light gray, immutable objects and references in ice blue and borrowed references as orange.

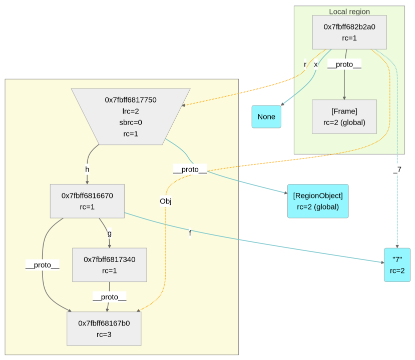


Fig. 9. State of the heap after line 7 of Fig. 7 as rendered by FrankenScript.

Running a FrankenScript program produces a file with one diagram per line showing the heap's evolution called `mermaid.md`. Fig. 9 shows the heap after line 7 of running the FrankenScript equivalent of the code in Fig. 7. See supplemental material for the entire execution.

Despite being sequential FrankenScript can still be used to explore safe concurrency. Successful transfer of a region into a cown means that the region (its contents and nested regions) is disjoint from state reachable from the current thread and therefore could be operated on in parallel as soon as the cown is released. To explore reachability, and ensure oneself that local state is not reachable from a closed region in a cown, the `mermaid_show_tainted()` function can be used to colour the transitive closure of mutable state reachable from the taint starting point.

FrankenScript is implemented in Trieste, a C++ DSL for tree rewriting [20]. It can be built on Linux, Windows and Mac using standard tools and is available as an artefact [57] for this paper.

5 Design Alternatives and Challenges

Let us now review alternatives and challenges due to common features in dynamic languages.

5.1 Implicit Freezing

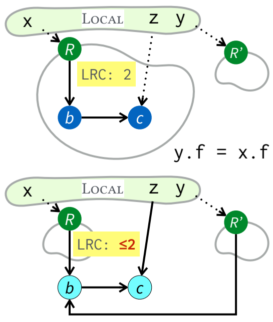


Fig. 10. Implicit freezing of b and c destroys the integrity of the local reference count for R.

Our write-barrier raises an error when a reference would otherwise be created from one region to an object in another region. The errors are raised on lines 56 and 59 of `addToRegion()`. An alternative approach to this behaviour is to "fix" the problematic reference by freezing the transitive closure reachable from the shared object. This is straightforward to implement by replacing each `Error` by a call to `freeze(curr)` on the aforementioned lines as well as have `Thread`'s `start()` method freeze contained and local objects passed to the thread constructor.

The obvious downside of implicit freezing is that it may cause objects to become immutable unintentionally. When freezing a *region*, the effects will not propagate beyond the programmer-curated region, simplifying reasoning about the results of freezing. On the other hand, there are objects that one might want to freeze implicitly, such as strings and types (*c.f.*, §5.3.2). A possible middle ground is to support implicit freezing for a select number of types, or by opt-in.

However, implicit freezing interacts badly with our local reference count technique. Consider the region R in Fig. 10. If we create a reference to b from another region, the object will be frozen along with c. Because b and c were frozen, R is actually closed. However, we are unable to detect that the reference $\mathtt{z} \longrightarrow c$ should no longer contribute to R's local reference count, unless we trace the entire memory of R (or the local region), which is an expensive operation when the region is large.

A pragmatic approach can be taken to handle this: we mark the local reference count as "dirty" (shown by red colour in Fig. 10). This means that the counter is an upper bound and may prevent LRC from reaching zero making the region closed. To resolve this we have to resort to other means such as traversing the local region to invalidate all borrowed references or count them to establish a correct value for the counter. An alternative approach would be to record more information about borrowed references, in particular what objects in a region have incoming borrowed references. This would allow us to remove the total number of incoming references to the frozen objects from R's local reference counter, at the expense of increasing the cost for tracking borrowed references.

𝔉ranken𝔖cript permits freezing the transitive closure reachable from any object and supports implicit freezing by default. This behaviour can be turned off in favor of the strict freezing protocol described in §2.3 by calling the built-in function `pragma_disable_implicit_freezing()`.

5.2 Invalidation of Borrowed References

Closing a region requires the absence of borrowed references to the region (and its nested regions). In statically typed languages, the duration of borrowing is typically known at compile-time from scoping. The most straightforward definition of borrowing only permits borrowed references on the stack [7]. More powerful definitions permit borrowed references on the heap, using typing mechanisms to delimit their existance (*e.g.*, Clarke and Wrigstad [15]). In Rust [36], the borrow checker ensures that the lifetime of the borrowed value outlives the duration of the borrow. Since Lungfish permits borrowed references from the local region, borrowed references can be in local variables as well as fields in objects in the local region. To satisfy (G5), we do not introduce any syntax for borrowing, but rely on programmers to manually invalidate borrowed references, or write their programs with clever uses of scopes such that local variables are invalidated automatically. In languages with prompt collection of memory, such as Python which uses reference counting, borrowed references in temporary objects such as iterators will be cleaned up once the reference to the iterator object is invalidated. A language implementation using a tracing GC would require more manual invalidation.

It is possible to think of a borrowed reference as "weak" in the sense that it is only valid as long as the region it refers into is open. Thus, closing a region could be made to invalidate borrowed references dynamically. An obvious downside to this proposal is that references may be invalidated "under foot": Closing a region may cause references to become undefined elsewhere in the thread.

The `close(`b`)` function in 𝔉ranken𝔖cript will invalidate all references from the local region into the region. The function `is_closed(`b`)` returns **`True`** if the region is closed.

5.3 Dealing with Pervasive (Mutable) State

5.3.1 Mutable Types. Despite studies showing that most objects are thread-local in languages like Java [66, 67], the sharing situations in Fig. 1 are very common in languages that reify types and where each object has a reference to its creating type or class. This is the case in Python and JavaScript, and in both cases, *the type objects are mutable*. Thus, even the act of instantiating two objects of the same type in different threads creates an opportunity for the threads to race on the type object. Freezing an object also requires freezing the type object, or it could change in ways that could effectively violate immutability. For example, overwriting a getter method on the class with one that returns a fixed number in the style of Abadi and Cardelli [1] can be indistinguishable from a change to the object to a client of the object.

Python 3.12's multiple subinterpreters model addresses this problem by having one type object for each type per subinterpreter. This avoids races, but instead creates problems if an object is passed (in a serialised fashion) between subinterpreters with different interpretations of its type.

In Lungfish, we require that a type is frozen before it is used in more than a single region. This permits a program to use Python's reflective capabilities and modify the type up to the point where the type object is shared across regions. The implicit freezing in § 5.1 handles freezing of types automatically as soon as a type object becomes reachable from a second region. Without such an automatic solution, programs may become littered with annotations to freeze a type to ensure that they can run without throwing exceptions, or handling such exceptions.

5.3.2 *Immutable Types.* But one does not simply freeze a type. In contrast to static languages, a Python program is constructed dynamically during its execution. This poses challenges with making reflective objects immutable, as the following example will show. Consider the use of a conditional expression to determine the base class of a type: `class A(B if cond else C):`. This evaluates to a new type object bound to the local name `A`, but also updates the `tp_subclasses` list in the type object for `B` or `C`, depending on *cond*. The `tp_subclasses` list can be queried through the `__subclasses__()` method. There are two aspects to this: If making type objects immutable is to be interpreted strictly, then freezing a type would prevent deriving from it. We could not program analyse our way out of this problem and create an immutable, fully populated subclass list from the start as we cannot determine `A`'s base type without running the program.

A straightforward solution is to store the `tp_subclasses` data elsewhere in the system, or remove it from the immutable fields. This brings us to the second aspect: (a) calls to `__subclasses__()` racing with concurrent subclassing, or (b) concurrent subclassing racing on writing to `tp_subclasses`.

Problem (b) is the simpler problem. Since concurrent class creation will always lead to creation of discrete type objects, additions to `tp_subclasses` are not order sensitive. Thus, concurrent subclassing can be supported by making accesses to the `tp_subclasses` list thread-safe. Problem (a) is trickier as it permits different interleavings of an execution to change the behaviour of a program branching on the observable content of `__subclasses__()`. Our solution is to disable the `__subclasses__()` method on mutable and immutable type objects.

To support querying a class for its subclasses (and similar operations), we introduce a per-class mirror [8] hidden inside a cown that must be acquired to access these parts of the reflective API, as show in Fig. 11. The mirror represents a snapshot of the class taken at acquisition. This allows concurrent subclassing directly without the mirror (b) in a way that is unobservable through a concurrent mirror. To observe added subclasses, the mirror must be relinquished and re-acquired. *Note that types must remain immutable since we permit immediate concurrent access to type objects.*

```
m = A.mirror() # m is a cown
with m:
  sc = \
    get(m).__subclasses__()
  for c in sc:
    print(c.__name__)
```

Fig. 11. Using a mirror to query a class' subclasses.

To support our goal of backwards compatibility with sequential programs (G2), we only require access through mirrors from the point where a second thread is created in the program. When libraries developed for single-threaded Python are used in a multi-threaded context, we expect most uses of reflection to break. However, due to the dangers of reflection in a concurrent setting, forcing programmers to review and refactor this code is not without merit. A strength of the design is permitting common cases, such as subclassing, to remain unchanged.

FrankenScript models mutable type objects through its prototype-based design. Every object has a `__proto__` field pointing to a mutable prototype object and lookups of names to an object delegate to the prototype when the names cannot be found directly in the object. Freezing a prototype is possible either directly by `freeze()`, by turning on implicit freezing and sharing a prototype across regions, or by placing prototypes in a region at the start of a program and then freezing it.

5.3.3 *Module State.* Python modules define a top-level namespace, which they can refer to internally using `globals()`. This namespace is a dictionary which is accessible to an importer through

```
1  state = Region()
2  ... # populate state
3  guard = Cown(move state)
4
5  def random():
6    with CS(guard): # acquire
7      ...
8      return result
9      # implicit release
```

```
10 state = Region()
11 ... # populate state
12 guard = Cown(move state)
13
14 def random():
15   assert acquired(guard), "Error"
16   tmp = get(guard) # borrow
17   ...
18   return result
```

```
20 state = Region()
21 ... # populate state
22 guard = Cown(move state)
23
24 def random(state):
25   ...
26   return (result, \
27           move state)
```

Fig. 12. Three examples of a module using a cown to store its mutable state using different approaches to ensure exclusive access to it (G3). Left: module encapsulates cown acquisition (Line 6), making calls to `random()` thread-safe. Middle: module assumes its cown has been acquired by the caller (Line 15) and then borrows the the content of the cown. (The assert is not necessary for safety.) Right: module requires a state region to be moved into it (Line 24), and moves the state back to the caller together with the result (Line 26–27).

the name of the imported module. Modules make use of this dictionary to hold mutable state crucial for its operation (*e.g.*, the `random` module) or immutable constants such as `math.pi` in the `math` module. As dictionaries are mutable by default, a proponent of the Indiana Pi Bill [64] is free to import `math` and subsequently assign 3.2 to `math.pi` without intervention, or add state to a module whose definition is stateless. Thus, regardless of a module's internal behaviour, using a module across two different threads, gives rise to the undesirable sharing situations in Fig. 1.

Forcing a module to freeze its top-level namespace make stateless and immutably stateful libraries safe to use. However, this will not work for modules such as `random`. To support gradual Lungfish adoption, we extend Python so that modules can be marked as Lungfish compliant. Compliant modules will be frozen imediately after import, at least once the program uses more than one thread. Recall that freezing does not propagate through cowns. This permits a compliant module to refactor its necessarily mutable state into a cown. Some libraries may encapsulate this fact and hide acquisition of the cown entirely inside the library. This works well in the case of `random` but not when it is necessary to reason atomically about interaction spanning multiple calls to the same library. In this case, the library can require that the cown has already been acquired by the caller. Fig. 12 shows three approaches to mutable state in a module using cowns.

5.3.4 Migration and Non-Compliant Modules. A very crude support for a non-compliant module is possible by silently lifting its entire top-level namespace into a region shared by all non-compliant modules wrapped inside a global cown called `NonCompliantModule` (G3). The import would also wrap all the module's functions in a trampoline that performs an assert on the cown:

```
    assert acquired(NonCompliantModule), "Error: NonCompliantModule not acquired"
```

This will effectively make all interaction with non-compliant modules take place in a critical section, permitting their use at the cost of suspending all parallelism. The reason for all non-compliant libraries sharing the same "cown guard" is to facilitate using non-compliant libraries that depend on other non-compliant libraries. With a single cown per non-compliant library, a user would have to know the total set of non-compliant dependencies for a library to acquire them.

Interaction with a non-compliant `random` module would look thus (four separate lines):[6]

```
import random as r1    x = 02    with NonCompliantModule: x = r.random()3    print(x)4
```

The call to `r.random()` performs the assertion above, and thus requires that the `NonCompliant` cown is acquired, which happens on line 3. Note that the code is written assuming blocking interaction

[6]If the `random` module was made compliant according to Fig. 12, simply replace `NonCompliantModule` by `r.guard` which is the cown wrapping the module's `state` region created on Lines 1–3, 10–12 and 20–22.

with cowns. It would be possible to use behaviour-oriented concurrency [13], in which case line 4 would be nested in the **with** in line 3, or become an explicit continuation spawned inside the **with**. If line 4 is inside the **with**, the printing is part of the critical section, otherwise not.

If the result of `r.random()` would create a persistent reference into the module's implicitly created region, the release would not succeed because the region in the releasing cown would not be closed (or alternatively, the object is implicitly frozen (*c.f.*, §5.1)).

5.3.5 *Observing Reference Counts.*

The Python function `sys.getrefcount(obj)` returns the value of the reference counter for the object `obj`. This function makes it possible to observe mutation of an immutable object's reference counter. Inspecting the value of the reference counter is done to check for ownership in certain parts of the Python source code and in libraries such as `numpy`. If the value of the reference counter is 1, that means that the object is unique, and therefore safe to update in-place, which can be very useful to avoid large or frequent allocations. Similar tricks are played in *e.g.*, Koka [49] to match deallocation with allocation in close proximity.

To avoid code believing it can "own" an immutable object, and to prevent code from observing concurrent creation of references to a known object, we redefine `sys.getrefcount()` to return `INTMAX` when called with an immutable argument (G3). This is consistent with how Python handles "immortal" objects[7] to reduce cache invalidation and avoid data races.

This brings us back to our claim in §2.1 that immutable objects are safe to share. While this is true from the perspective of the programmer, underlying details such as memory management may still warrant special treatment of immutable objects. In Lungfish, we require atomic reference count manipulations of immutable objects (Fig. 6), just like for cowns.

6 Putting it All Together

Fig. 13 shows a small concrete example written in Python extended with Lungfish. We model a bank with one bank account per cown. This fine-grained concurrency permits operating on accounts in parallel, although our example does not as all operations access the same accounts. We illustrate multi-cown acquisition to atomically `transfer` money between accounts, and read/only cown acquisition to `tally` the total balance of all accounts in the bank. Transfer operations are logged, and we share a single log entry across two accounts involved in the same transfer. This requires the log entries to be immutable. We illustrate both implicit and explicit freezing of log entries. In the first case `(*)`, the log entry is created as a local object, then moved into the **self** account's region before becoming implicitly frozen when the log entry is appended to the `dest` account's region. In the second case `(**)`, the entry is moved into a region and frozen.

Let us look closer at the transfer of funds from account `a1` to account `a2` on line 48. The **with** block handles acquisition and release in as described in §2.3.1. Because of write acquisition, the two cowns' regions are given write permission. This is handled by the `CS` cown set class under the hood by implementing the context manager interface.

The calls to `ba()` opens the regions of both accounts' cowns and places references to the account objects on the stack. This borrowing is handled by lines 16–17 in `addReference()` and increases both accounts' regions' LRCs from zero to one, opening them. In this case, both regions are roots in region trees rooted in separate cowns, meaning they have no parent regions to inherit permissions from. When the block ends, the borrowed references have been invalidated, causing LRC to drop back to zero for both regions — they are now closed. Thus, the releasing of the cowns will succeed.

[7]While our immutable objects are not immortal, the immortality concept was motivated by a desire to make the reference counts of immutable objects immutable too [55].

```
 1 # Account class for contained objects
 2 class Account:
 3   def __init__(self, balance):
 4     self.balance = balance
 5     self.log = []
 6   def transfer(self, dest, amount):
 7     if self.balance >= amount:
 8       dest.balance += amount
 9       self.balance -= amount
10       # implicitly immutable entry (*)
11       log = Log()
12       self.log.append(log)
13       log.entry = "Transfer OK"
14       dest.log.append(log) # freeze!
15     else:
16       # explicitly immutable entry (**)
17       log = entry("Transfer fail")
18       self.log.append(log)
19       dest.log.append(log)
20       raise Exception("Overdraft")
21 # Opt-in implicit freeze support
22 class Log(Freezable):
23   pass
24 # Create log entry
25 def entry(msg):
26   r = Region()
27   r.obj = Log()
28   r.obj.entry = msg
29   freeze(r)
30   return r.obj # immutable
31 # Create account
32 def account(b): # b = balance
33   r = Region()
34   r.account = Account(b)
35   return Cown(move r)
36 # Borrows account from cown
37 def ba(c):
38   return get(c).account
39 # Tally balance of accounts
40 def tally(acc, result):
41   sum = 0
42   with CS([a for a in acc]):
43     for c in acc:
44       sum += ba(c).balance;
45   with CS(result):
46     set(result, sum)
47 # Transfer money between accounts
48 def transfer(a1, a2, m)
49   with CS(a1, a2):
50     ba(a1).transfer(ba(a2), m)
51 # Swap contents of two cowns
52 def swap(a1, a2):
53   with CS(a1, a2):
54     tmp = set(a1, get(a2))
55     set(a2, move tmp)
56 # import the Thread class
57 from threading import Thread
58 # Create accounts and perform tasks
59 a1 = account(100)
60 a2 = account(20)
61 Thread(transfer, a1, a2, 10).start()
62 Thread(swap, a1, a2).start()
63 result = Cown(None) # Empty released cown
64 # Note: tuple of cowns is immutable
65 t = Thread(tally, (a1, a2), result)
66 t.start()
67 t.join()
68 with CS(result): # Print tally result
69   print(get(result))
```

Fig. 13. Example illustrating regions, cowns, read and write cown acquisition, implicit and explicing freezing, borrowing and region transfer between cowns.

The tally operation demonstrates the use of a cown as a way to pass data back from a child thread to a parent. The `tally()` function takes an empty cown `result` where the result is stored. We use the `join()` method to ensure that printing does not start until the result is available.

Failure Modes. When the code in Fig. 13 is executed, it spawns three threads that all access the two accounts `a1` and `a2`, racing on transferring money, swapping the two accounts, and tallying the sum of their balances[8]. The thread creation on lines 61, 62 and 65 all take cowns or immutable objects. If `a1` or `a2` were mutable objects, thread creation would have failed (§ 2.3.1): neither contained nor local objects may be passed as arguments to threads; if the regions inside `a1` or `a2` were not wrapped in cowns, they would not be closed at thread start as the original variables are copied, not moved. If they were moved, they would have been moved to the thread created on line 15, and subsequent threads would have been passed `None`. In summary, Lungfish will prevent the accounts from being shared between threads as directly accessible mutable objects.

As the accounts are cowns they are sharable, because Lungfish ensures that concurrent accesses to them are properly synchronised. Contents of cowns are only accessible through `get` and `set`. Such calls will fail outside an acquiring **`with`**. Because the **`with`** blocks on acquiring cowns, they will serialise the execution, and avoid data races. At the end of the block the cowns are released, which throws an exception unless the cowns' regions are closed. This prevents a reference to a mutable object in a region from escaping and being accessed outside of the block that dynamically ensures exclusive access to it, as shows in Fig. 4.

7 Evaluation of Design Goals and Implementation Goals

Table 2 summarises our goal fulfilment and points to the discussion in the paper. Implementation goals that are too early to tell are included to make sense of our design decisions. We expect that

[8]In a large system, it is easy to imagine concurrent operations on the same accounts being initiated from different places and needing to be ordered in some way to avoid racing. To keep things simple, we spawn them in a single place.

the full implementation will take two person years to complete. In our current state, we have implemented most of the functionality for immutable objects, and the diff is around 2 KLOC.

7.1 Design Goals

7.1.1 (G1). Because Lungfish only allows access to data that is immutable or exclusively accessible to the current thread, it is data-race free. This ticks (G1).

7.1.2 (G2) and (G3). Existing sequential programs allocate and manipulate only inside the local region. They will therefore work unmodified despite actually using (ephemeral) ownership.

Lungfish's isolation permits us to leverage Python 3.12's multiple subinterpreters so that each thread runs its own subinterpreter (with its own GIL [54]). A necessary prerequisite of this is to change how memory is managed since Lungfish permits objects created by one thread to be accessed directly by other threads. In Python 3.12, each subinterpreter manages its own isolated memory using non-atomic reference count manipulations and its own cycle detector. Since mutable regions are only visible to one thread at a time and all shared objects guaranteed to not change, each subinterpreter can safely be extended to handle cycle detection inside its reachable memory. Reference count manipulations on immutable objects and cowns must become atomic (line 13 in Fig. 6).

Running a per-thread subinterpreter removes the performance limitations of the GIL without needing to remove the GIL. As a side-effect of this design, existing code that uses old Python threads or uses parallelism at the C-level, *and therefore relies on the GIL for safety in both cases*, will not crash or compromise the Lungfish-enabled CPython runtime This ticks (G2) and (G3).

As its name suggests, Lungfish thus both has GILs and no GILs, and is close enough to pre-Lungfish Python that existing code should work, even if it uses pre-Lungfish threads or parallelism at the C-level. (But this code is not guaranteed to be data-race free.)

7.1.3 (G4). Our design delays the point of enforcing region isolation to where it would otherwise lead to mutable state becoming directly accessible to multiple threads. This is stricter than a data-race detector which would only err in case of a data race on such shared state, but such tools can let bugs slip into production that are hidden in unlikely interleavings. While it is possible to extend borrowing to all regions and let regions intersect while open, it would require a more costly solution than LRC to track "foreign references". While on the upside, this would make it possible to move objects beween regions manually, such code could easily break since a move must reason about all incoming alias to an object. With these caveats, we believe that we tick (G4).

7.1.4 (G5). We mostly piggyback on existing language constructs to construct and manipulate regions and cowns, *e.g.*, function calls, constructors and context managers. The only new syntax is **`move`**, but this is notably a convenience only — not fundamental. A move can always be implemented manually by assigning `None` to a variable because we never perform a closed check immediately after a transfer, only on cown release, freeze, or thread start. This ticks (G5).

7.2 Implementation Goals

We do not claim to meet these goals, but reason why we believe that the full implementation will.

7.2.1 (G6a). Single-threaded programs opt out of most of the overhead of ownership checking: they only allocate in the local region, never perform implicit transfer nor topological changes, and they never implicitly freeze objects. Thus they exercise fast constant-time parts of our write-barrier. Some checks can be delayed until the start of the second program thread.

We speculate that the write-barrier will not cause a significant slow-down. A store in CPython such as `x.f = 42` goes through many levels of indirection, branching on *e.g.*, the object's representation and if its class replaces the default behaviour for setting attributes with user code.

Table 2. Goal fulfilment. ✓= goal fulfilled. ?= Too early to tell, but we expect so.

Goal	Description	Status	Discussion
(G1)	Data-race freedom	✓	§2, §3
(G2)	Existing sequential programs should work unmodified	✓	§2.2, §7.1.2
(G3)	De facto safe libraries should be easy to make compliant	✓	§5.1, §5.3.4, §7.1.2
(G4)	Delay errors; "favour no false positives over no false negatives"	✓	§3, §7.1.3
(G5)	Low number of new syntactic additions	✓	§2.3, §7.1.4
(G6a)	Almost no slow-down of sequential programs	?	§7.2.1
(G6b)	Moderate overhead for multithreaded programs	?	§7.2.2
(G6c)	Compatible with existing performance optimisations	?	§7.2.3
(G7)	Preserve maintainability of the runtime	?	§7.2.4

7.2.2 (G6b). Multi-threaded programs that create many regions will pay a cost for moving objects into regions which is proportional to the number of objects in the region. The upper bound for a topological change is the depth of the deepest region tree, but in practise most region movement will be cheaper. The cost for freezing objects is linear in the size of the number of objects being frozen. The program will also pay for implicit freezing of type objects and literals, many of which are shallow object structures.

When immutable objects are frozen, they can be made more efficient. For example, the overhead needed to ensure consistency between non-atomic operations on dictionaries and lists can be completely avoided. It may also be possible to lay out the objects in memory more efficiently.

With respect to memory management of immutable objects, we plan to build on work by Parkinson et al. [46] that shows how a single atomic reference count can be used to track the liveness of an entire immutable strongly connected component as its objects share the same lifetime. Notably, this approach deals with cyclic immutable garbage without tracing.

Partitioning the mutable heap into regions has positive implications for garbage collection. With the exception of cowns, cycles in the mutable heap cannot cross region boundaries. Therefore, cycle detection can be incrementalised and easily parallelised. When Lungfish is added to Python, each region will have its own cycle detector. Furthermore, programmers are now able to safely drop a bridge object to quickly deallocate an entire region. We thus expect that making garbage collection more efficient may offset some of the overhead of ownership checking.

At the present, the lifetime of cowns is managed using atomic reference counting. Since cowns are mutable, the tricks we employ for immutable objects do not apply. We have not yet developed any cycle detection for cowns. For now, we rely on programmers to manually break cycles.

7.2.3 (G6c). Because our design is not contingent on removing the GIL, most of the CPython implementation can stay unchanged, and there are no far-propagating changes due to GIL removal as in the case for free-threaded Python. (See more below.)

7.2.4 (G7). To see how Lungfish can contribute to preserving the maintainability of the CPython runtime, let us compare with free-threaded Python.

Because our ownership model is strict and does not permit simultaneous access to mutable objects from multiple threads, much of the Python runtime should be able to stay the same when support for multiple threads is added, as well as optimisations which have been developed in a sequential setting (G6c). In particular, reference count manipulations can stay the same, *i.e.*, a simple non-atomic increment or decrement, with the exception of immutable objects and cowns, as these are shared across multiple threads. If we compare with the proposed changes for free-threaded Python [28], this means no tricks are needed to ensure reference count accesses racing with deallocation and no need for biased or deferred refence counting to reduce the amount of atomic reference count manipulations. Also, cycle detection can stay the same, and built-in data structures like lists, dictionaries and sets do not need to be rewritten to become thread-safe. We need to increase the object header with enough bits to identify the object's owning region but we believe this space could be created by replacing the intrinsic doubly-linked list that CPython uses to track memory allocations (and that thus adds two pointers to each object header). Notably, free-threaded Python relies on mimalloc [37] to be able to free up enough space in the object header to be size neutral. This alone is an addition of 15 KLOC to the CPython runtime [28].

Most of our checks are performed in the write-barrier in Fig. 6. Python has no write-barrier infrastructure in place, and piggybacking on reference count manipulation is not an option as these are encoded as different C macros that inject code in a context assuming existence of variables with certain names. Once a write-barrier is added to the codebase, much of the heavy lifting is done.

When each thread has its own isolated memory, there are no concerns about visibility, atomicity, or ordering of memory operations across threads. The Lungfish ownership model thus shifts the responsibility for ensuring memory safety from hardware mechanisms to run-time software checks, effectively reducing the need for strong memory consistency models.

Finally, most additions will be contained in a single module that handles the region logic.

8 Related Work

Although early work on *object encapsulation* started in Smalltalk [32], encapsulation has not been as common in dynamic languages as in statically typed languages. A reason may be the overhead necessary to impose encapsulation dynamically, or process encapsulation á la Erlang [3, 45]. With respect to *name-based encapsulation*, Smalltalk makes all instance variables private and methods public [25]. Recently, Thomas et al. [60] explored the overhead of supporting a protected modifier in dynamic languages. Python supports name-based encapsulation through convention. Ruby, Dart, Newspeak, and recent versions of JavaScript support private fields.

The early work on object encapsulation [32, 33] triggered a stream on research into managing, preventing or elucidating aliasing in programming. In this paper we have referred to this stream of work as "ownership", which is an umbrella term for hundreds of papers, including works on uniqueness, fractional permissions, universes and ownership types. Due to the success of Rust [34], the ownership moniker has become synonymous with affine types in mainstream programming.

Early work on ownership-based dynamic alias protection [44] and follow-up work [27] took a different approach to ownership and transfer than Lungfish. These works consider a method to execute inside an object (the current `this`) which becomes the owner of all objects created in the method. When this is the wrong default, an object may gift any object it owns to another object.

Preceeding our realisation that type objects must be frozen by 25 years, Noble et al. employ ownership to protect prototype objects from accidentally propagating changes when shared across different prototype chains [44]. Notably, neither of these papers restrict the object topology (like Lungfish does), but rather restrict uses of references that violate ownership (which Lungfish does not do). This is key in these systems to change ownership when multiple objects form complex relationships: If A and B are owned by the same owner and reference each other in a cycle, they must change owner "atomically" or references that violate ownership will exist temporarily.

This approach is not possible in Lungfish without a check on each dereference, which is something that we have carefully avoided due to its performance implications (reads are more common than writes). We furthermore consider this a bad fit when the application of ownership is concurrency safety and trying to save language implementers from having to protect the VM from incorrectly synchronised programs: if a reference is permitted to cross between two threads, we end up having to deal with concurrent reference count manipulations (if memory is managed like in Python), and lose the ability to perform cycle detection individually for each region.

Early work on applying ownership to concurrent programming [5, 6] extend ownership to permit threads to own objects (and later also actors [17, 21]) in statically typed languages. These designs too permit violating references to exist but restrict their use. Particularly exiting early works in this space [35, 38] do not rely on static isolation but instead *dynamically check* whether the current thread is the owner of an object, by checking that a path of owners end in the current thread. Lungfish uses this design for cowns: access is only permitted if the cown is acquired by the accessing thread. Objects in a Lungfish region that are reachable by a thread are *always owned by that thread*, and can be accessed without a check. A check is instead needed to ensure that a region can be closed for transfer (*i.e.*, no violating references remain with the sender).

In the context of dynamic languages, and outside the thread paradigm, E [42] and AmbientTalk [22] provide a form of dynamically enforced ownership through the *vat* concept. A vat isolates a single-threaded control loop that drives all access to the data inside the vat, but the data itself is not isolated. Any interaction with an object o from outside its enclosing vat v will be turned into an asynchronous operation sent back to v, ensuring that concurrent accesses to o are serialised. Racket supports isolated "places" [59], similar to actors and communicating using message passing.

Different ownership models use different granularity of ownership. Initial work [15, 18, 48] on ownership types [14] allowed objects to own other objects, and imposed a hierarchical decomposition of the heap. Uniqueness, a simple form of ownership has been used in *e.g.*, Clean, Eiffel, Smalltalk, and Rust to deliver control ownership on a per-object level [10, 32, 36, 43]. Vitek and Bokowski [63] proposed module-granularity ownership for Java. Arvidsson et al. [4] proposed region-based ownership for the Verona programming language, similar to *e.g.*, work on Cyclone, a memory-safe C [29, 31]. Uses of ownership to deliver safe concurrency often couple ownership with the concurrency model, or the unit of concurrency. In both AmbientTalk and E the unit of concurrency is also the owner. Several statically typed languages follow suit, for example Joelle [17], Pony [19], and Encore [9, 12] (actor based) and Loci [66] (thread based). In these systems, the actor (or equivalent) is the owner of objects and uniqueness is used to permit safe transfer of ownership of objects between actors. The actor-based Swift language recently introduced support for data-race freedom, using a region-based model reminiscent of [17] and directly drawing on work by [40], but without support for nested regions. Python's evolutionary path is not unique. Racket's runtime was highly tuned for sequential programs when concurrency was added, with similar goals to ours [58]. OCaml was recently retrofitted with support for parallelism [52, 53] removing reliance on the global "Runtime Lock" that effectively serialised the execution in a single process. There is now a proposal to extend OCaml with a form of regions called capsules [24], which statically guarantees data-race freedom, while providing backwards compatibility with sequential code.

Our ownership model is perhaps most similar to work by Fernandez-Reyes et al. [23] on Dala, a proposal for using ownership to make dynamic languages data-race free. Dala supports three kinds of objects: immutable (1), uniquely referenced (2), local to the current thread (3). A reference from an object A to an object B is permitted only if A's kind is higher or equal to B's. Dala does not couple ownership with unit of concurrency, but is more restrictive than Lungfish as it does not support sharing of mutable state, only immutable state and ownership transfer, and transferrable objects must be tree-shaped, as in the Kilim system [56]. Inspired by work on gradual ownership types [50], Dala provides a gradual type system to catch concurrency errors statically. We will explore something similar for Lungfish in Python, drawing from its closeness to Reggio [4].

The use of `with` blocks to acquire and release cowns is similar to many scoped concurrency mechanisms in other languages, such as `synchronized` in Java. However, the Java mechanism does not prevent access to the underlying state if it is not wrapped in a `synchronized` block and only protects a single object. The use of `with` blocks to acquire and release cowns is tied to our ownership model, and thus is much closer to Rust's standard library `Mutex` type, which uses the static borrow checker to ensure access does not escape the scope of the mutex usage. With Lungfish, we provide the same guard rails for the programmer as Rust, but in a dynamic language.

9 Interfacing with The Faster CPython Team

This project began by pitching an ownership model based on Reggio [4] to the the Faster CPython Team [41]. The initial reactions could be described as, at best, lukewarm. The Reggio design had very strict views of what was a permissable program. This was not a good fit for Python, which is a very permissive language. As time progressed, our shared understanding developed significantly, and through frequent discussions with the Faster CPython team, we adapted the Reggio design to fit with the Pythonic way of doing things. Under guidance from the Faster CPython team, the design morphed into Lungfish, which is a more permissive ownership model. From Reggio, we removed the explicit region enter construct, the single window of mutability, and added implicit freezing and the ability to extract isolated subgraphs from a region. We believe that the "programming experience" of Reggio can be improved by our insights from Lungfish.

We are very grateful to the team for discussing the project with us regularly (~every 2 months). By the time of submission, several Faster CPython team members had contributed sufficient ideas to warrant author credit. There are still many open questions, and we are looking forward to continuing the collaboration with the Faster CPython team to bring ownership to Python.

10 Conclusion

Many dynamic programming languages have evolved in a sequential setting. Their semantics, libraries and runtimes are optimised for single-threaded operation. When concurrency is added, features which are ubiquitous such as mutable types and module state easily lead to mutable data being shared across threads. Without dynamic ownership checks, data-races are inevitable.

We have presented Lungfish, a dynamically enforced region-based ownership model that makes data-races impossible by construction. While our target is the Python programming language, we believe it will fit other languages as well, such as JavaScript. We have built 𝔉𝔯𝔞𝔫𝔨𝔢𝔫𝔖𝔠𝔯𝔦𝔭𝔱, a toy programming language that demonstrates the semantics of a program graphically, to communicate the Lungfish semantics effectively to members of the Python community.

Sharing mutable objects across concurrent threads requires a memory model to define the semantics of all legal operations. When the work on free-threaded Python materialises, Python will need such a memory model — in the very least defined through the implementation of a data-race detector or similar debugging tool needed to weed out data-race bugs. With Lungfish, we limit what operations are legal, which shifts the cost from debugging concurrency bugs (which are timing-related) to ownership bugs (which are structure-related).

In future work, we will provide a dynamic semantics of Lungfish combined with concurrency along with proofs of data-race freedom. The region-based ownership model draws heavily on Reggio [4] which has been proven sound. However, Reggio is statically checked and more restrictive than Lungfish, so a separate formalism is required. While one cannot stress enough the importance of having formally stated properties proven to hold in a mathematical model, the foremost challenge in this work lies in the wicked realm that cannot be easily quantified: design — striking a "good enough" balance between strictness, performance, usability and backwards compatibility.

We believe Lungfish meets our goals of data-race freedom, backwards compatibility, performance, delaying checking to avoid false positives, and reasonable performance without adding much complexity to the Python runtime. With Lungfish, we hope to make Python's graduation to a concurrent language smoother and safer.

11 Data Availability Statement

The published artefact [57] is a Docker image containing the interpreter of 𝔉ranken𝔖cript.

The artefact includes example programs to introduce the syntax and functions of 𝔉ranken𝔖cript documented in §4, and scripts to recreate fig. 2, 3, 5, 7, 9 and 10 from this paper.

Acknowledgments

We would like to thank Mads Torgerson for his insightful question that started this research. We are grateful to Michael Droettboom, Irit Katriel, and Mark Shannon of the Faster CPython team for volunteering their time to discuss this work and contributing suggestions and ideas. Finally, we thank the anonymous reviewers for their constructive feedback that improved the paper.

This research was supported by the Swedish Research Council through the grants 2024-04565, "Data-race Freedom and Memory Safety for Untyped Languages", and 2023-05526, "Pervasive Memory Safety through Ownership Types".

References

[1] Martin Abadi and Luca Cardelli. 1996. *A Theory of Objects* (1st ed.). Springer-Verlag, Berlin, Heidelberg.

[2] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. 2002. Alias annotations for program understanding. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object -Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington, USA, November 4-8, 2002.* 311–330. https://doi.org/10.1145/582419.582448

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

[4] 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 (oct 2023), 31 pages. https://doi.org/10.1145/3622846

[5] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. 2000. Guava: a dialect of Java without data races. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Minneapolis, Minnesota, USA) *(OOPSLA '00).* Association for Computing Machinery, New York, NY, USA, 382–400. https://doi.org/10.1145/353171.353197

[6] 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 event-place: Tampa Bay, FL, USA.

[7] 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.

[8] Gilad Bracha and David Ungar. 2004. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vancouver, BC, Canada) *(OOPSLA '04).* Association for Computing Machinery, New York, NY, USA, 331–344. https://doi.org/10.1145/1028976.1029004

[9] Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I Pun, Silvia Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. 2015. Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore. In *Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy.* 1–56. https://doi.org/10.1007/978-3-319-18941-3_1

[10] T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer, and M. J. Plasmeijer. 1987. Clean — A language for functional graph rewriting. In *Functional Programming Languages and Computer Architecture*, Gilles Kahn (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 364–384.

[11] Elias Castegren. 2018. *Capability-Based Type Systems for Concurrency Control.* PhD Thesis. Uppsala UniversityUppsala University, Division of Computing Science, Computing Science. Backup Publisher: Uppsala University, Division of Computing Science ISBN: 978-91-513-0187-7 ISSN: 1651-6214 Issue: 1611 Series: Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology.

[12] Elias Castegren and Tobias Wrigstad. 2024. Encore: Coda. In *Active Object Languages: Current Research Trends*, Frank S. de Boer, Ferruccio Damiani, Reiner Hähnle, Einar Broch Johnsen, and Eduard Kamburjan (Eds.). Lecture Notes in Computer Science, Vol. 14360. Springer, 59–91. https://doi.org/10.1007/978-3-031-51060-1_3

[13] Luke Cheeseman, Matthew J. Parkinson, Sylvan Clebsch, Marios Kogias, Sophia Drossopoulou, David Chisnall, Tobias Wrigstad, and Paul Liétar. 2023. When Concurrency Matters: Behaviour-Oriented Concurrency. *Proc. ACM Program.*

Lang. 7, OOPSLA2, Article 276 (oct 2023), 30 pages. https://doi.org/10.1145/3622852

[14] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. 2013. Ownership Types: A Survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Lecture Notes in Computer Science, Vol. 7850. Springer, 15–58. https://doi.org/10.1007/978-3-642-36946-9_3

[15] Dave Clarke and Tobias Wrigstad. 2003. External Uniqueness Is Unique Enough. In *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2743)*, Luca Cardelli (Ed.). Springer, 176–200. https://doi.org/10.1007/978-3-540-45070-2_9

[16] David Clarke, Tobias Wrigstad, and James Noble. 2013. *Aliasing in Object-oriented Programming: Types, Analysis and Verification.* Vol. 7850. Springer.

[17] 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.

[18] David G Clarke, John M Potter, and James Noble. 1998. Ownership types for flexible alias protection. In *ACM SIGPLAN Notices*, Vol. 33. ACM, 48–64. Issue: 10.

[19] Sylvan Clebsch. 2017. *Pony: Co-designing a Type System and a Runtime.* PhD Thesis. Imperial College London.

[20] Sylvan Clebsch, Matilda Blomqvist, Elias Castegren, Matthew A. Johnson, and Matthew J. Parkinson. 2024. Trieste: A C++ DSL for Flexible Tree Rewriting. In *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering* (Pasadena, CA, USA) *(SLE '24)*. Association for Computing Machinery, New York, NY, USA, 184–189. https://doi.org/10.1145/3687997.3695647

[21] David Cunningham, Sophia Drossopoulou, and Susan Eisenbach. 2007. Universe Types for Race Safety. VAMP 07.

[22] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. 2006. Ambient-Oriented Programming in AmbientTalk. In *ECOOP*. 230–254.

[23] 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

[24] Aïna Linn Georges, Benjamin Peters, Laila Elbeheiry, Leo White, Stephen Dolan, Richard A. Eisenberg, Chris Casinghino, François Pottier, and Derek Dreyer. 2025. Data Race Freedom à la Mode. *Proc. ACM Program. Lang.* 9, POPL, Article 23 (Jan. 2025), 31 pages. https://doi.org/10.1145/3704859

[25] Adele Goldberg and David Robson. 1983. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc., USA.

[26] 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

[27] Donald Gordon and James Noble. 2007. Dynamic ownership in a dynamic language. In *Proceedings of the 2007 Symposium on Dynamic Languages* (Montreal, Quebec, Canada) *(DLS '07)*. Association for Computing Machinery, New York, NY, USA, 41–52. https://doi.org/10.1145/1297081.1297090

[28] Sam Gross. 2023. PEP 703 – Making the Global Interpreter Lock Optional in CPython. https://peps.python.org/pep-0703/.

[29] 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. Publisher: ACM.

[30] 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.

[31] Michael W. Hicks, J. Gregory Morrisett, Dan Grossman, and Trevor Jim. 2004. Experience with safe manual memory-management in Cyclone. In *ISMM*. 73–84.

[32] 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

[33] 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).

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

[35] B. Jacobs, K.R.M. Leino, F. Piessens, and W. Schulte. 2005. Safe concurrency for aggregate objects with invariants. In *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*. 137–146. https://doi.org/10.1109/SEFM.2005.39

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

[37] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. 2019. Mimalloc: Free list sharding in action. In *Programming Languages and Systems: 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1–4, 2019, Proceedings 17*. Springer, 244–265.

[38] K. Rustan M. Leino and Peter Müller. 2004. Object Invariants in Dynamic Contexts. In *ECOOP 2004 – Object-Oriented Programming*, Martin Odersky (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 491–515.

[39] Yi Lu, John Potter, and Jingling Xue. 2013. Structural Lock Correlation with Ownership Types. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 391–410.

[40] 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

[41] Jay Miller. 2022. A Team at Microsoft is Helping Make Python Faster. https://devblogs.microsoft.com/python/python-311-faster-cpython-team/.

[42] M. Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD Thesis. Johns Hopkins University, USA.

[43] Naftaly H. Minsky. 1996. Towards Alias-Free Pointers. In *ECOOP'96 - Object-Oriented Programming, 10th European Conference, Linz, Austria, July 8-12, 1996, Proceedings (Lecture Notes in Computer Science, Vol. 1098)*, Pierre Cointe (Ed.). Springer, 189–209. https://doi.org/10.1007/BFB0053062

[44] James Noble, David Clarke, and John Potter. 1999. Object Ownership for Dynamic Alias Protection. In *Proceedings of the 32nd International Conference on Technology of Object-Oriented Languages (TOOLS '99)*. IEEE Computer Society, USA, 176.

[45] Nikolaos Papaspyrou and Konstantinos Sagonas. 2012. On Preserving Term Sharing in the Erlang Virtual Machine . In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop (Erlang '12)*. Association for Computing Machinery, New York, NY, USA, 11–20. https://doi.org/10.1145/2364489.2364493 event-place: Copenhagen, Denmark.

[46] Matthew J. Parkinson, Sylvan Clebsch, and Tobias Wrigstad. 2024. Reference Counting Deeply Immutable Data Structures with Cycles: An Intellectual Abstract. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on Memory Management* (Copenhagen, Denmark) *(ISMM 2024)*. Association for Computing Machinery, New York, NY, USA, 131–141. https://doi.org/10.1145/3652024.3665507

[47] Alex Potanin, James Noble, and Robert Biddle. 2004. Checking ownership and confinement. *Concurrency - Practice and Experience* 16, 7 (2004), 671–687. https://doi.org/10.1002/cpe.799

[48] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. 2006. Generic ownership for generic Java. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 311–324. https://doi.org/10.1145/1167473.1167500

[49] Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: garbage free reference counting with reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 96–111. https://doi.org/10.1145/3453483.3454032

[50] Ilya Sergey and Dave Clarke. 2012. Gradual Ownership Types. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 579–599. https://doi.org/10.1007/978-3-642-28869-2_29

[51] Marco Servetto, David J. Pearce, Lindsay Groves, and Alex Potanin. 2013. Balloon Types for Safe Parallelisation over Arbitrary Object Graphs. In *4th Workshop on Determinism and Correctness in Parallel Programming*.

[52] KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting parallelism onto OCaml. *Proc. ACM Program. Lang.* 4, ICFP, Article 113 (Aug. 2020), 30 pages. https://doi.org/10.1145/3408995

[53] KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 206–221. https://doi.org/10.1145/3453483.3454039

[54] Eric Snow. 2022. PEP 684 – A Per-Interpreter GIL. https://peps.python.org/pep-0684/.

[55] Eric Snow and Eddie Elizondo. 2022. PEP 683 – Immortal Objects, Using a Fixed Refcount. https://peps.python.org/pep-0683/.

[56] Sriram Srinivasan and Alan Mycroft. 2008. Kilim: Isolation-Typed Actors for Java. In *ECOOP*. https://doi.org/10.1007/978-3-540-70592-5_6

[57] Fridtjof Stoldt, Matthew Parkinson, Sylvan Clebsch, Matthew Johnson, and Tobias Wrigstad. 2025. FrankenScript: A tool to explore and visualize region-based ownership in dynamic programming languages. https://doi.org/10.5281/zenodo.15235762.

[58] James Swaine, Burke Fetscher, Vincent St-Amour, Robert Bruce Findler, and Matthew Flatt. 2012. Seeing the futures: profiling shared-memory parallel racket. In *Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-Performance Computing* (Copenhagen, Denmark) *(FHPC '12)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/2364474.2364485

[59] Kevin Tew, James Swaine, Matthew Flatt, Robert Bruce Findler, and Peter Dinda pdinda@northwestern.edu. 2011. Places: adding message-passing parallelism to racket. In *Proceedings of the 7th Symposium on Dynamic Languages* (Portland, Oregon, USA) *(DLS '11)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/2047849.2047860

[60] Iona Thomas, Vincent Aranega, Stéphane Ducasse, Guillermo Polito, and Pablo Tesone. 2023. A VM-Agnostic and Backwards Compatible Protected Modifier for Dynamically-Typed Languages. *The Art, Science, and Engineering of Programming* 8, 1 (June 2023). https://doi.org/10.22152/programming-journal.org/2024/8/2

[61] Guido van Rossum. 2007. It isn't Easy to Remove the GIL. https://www.artima.com/weblogs/viewpost.jsp?thread=214235.

[62] Guido van Rossum and Alyssa Coghlan. 2005. PEP 343 – The "with" Statement. https://peps.python.org/pep-0343/.

[63] Jan Vitek and Boris Bokowski. 1999. Confined types. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Denver, Colorado, USA) *(OOPSLA '99)*. Association for Computing Machinery, New York, NY, USA, 82–96. https://doi.org/10.1145/320384.320392

[64] wikipedia 2024. Indiana pi bill. https://en.wikipedia.org/wiki/Indiana_pi_bill.

[65] Thomas Wouters. 2023. A Steering Council notice about PEP 703 (Making the Global Interpreter Lock Optional in CPython). https://discuss.python.org/t/a-steering-council-notice-about-pep-703-making-the-global-interpreter-lock-optional-in-cpython/30474.

[66] Tobias Wrigstad, Filip Pizlo, Fadi Meawad, Lei Zhao, and Jan Vitek. 2009. Loci: Simple Thread-Locality for Java. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5653)*, Sophia Drossopoulou (Ed.). Springer, 445–469. https://doi.org/10.1007/978-3-642-03013-0_21

[67] Nosheen Zaza. 2013. *Evaluating the Accuracy of Annotations in the Loci 3.0 Pluggable Type Checker.* Master's thesis. Uppsala University.

[68] Beatrice Åkerblom, Elias Castegren, and Tobias Wrigstad. 2020. Reference Capabilities for Safe Parallel Array Programming . *Programming Journal* 4, 1 (2020), 1. https://doi.org/10.22152/programming-journal.org/2020/4/1

Received 2024-11-14; accepted 2025-03-06