A Framework for Gradual Memory Management

A safe, performant technique for integrating custom allocators and garbage collection mechanisms

Jonathan Goodwin
jondgoodwin@gmail.com

September 12, 2017

Abstract

This paper details a technique that gives the programmer more control over memory use, including support for both static and runtime garbage collection, as well as multiple custom allocators, within a single program. This technique marries Rust's lexical owner model, Pony's reference capabilities, and the proposed allocator types. Use of these flexible mechanisms supports optimization of performance, memory safety, and memory use without the usual trade-offs. The proposed mechanism is gradual, allowing the programmer to progressively opt-in to its many capabilities, thereby easing the learning curve.

1 Introduction

Programming languages are opinionated about memory management, offering built-in support for a limited subset of proven allocation and garbage collection (GC) strategies. Given that memory management techniques vary in their delivered benefits and costs, supporting only a few strategies constrains a language's ability to serve a wider range of application requirements. At the risk of oversimplifying issues more carefully explored in section 3, the selected memory management strategy can impose these comparative trade-offs:

- Manual memory management sacrifices safety enforcement for performance and flexibility
- Runtime GC-based languages sacrifice performance and responsiveness for programmer convenience, flexibility and safety

- Static lexical memory management (as implemented by Rust's single owner regional inference model) sacrifices data structure flexibility, programmer convenience and sometimes memory utilization for safety and "zero-cost" collection.

For many programs, such trade-offs are not problematic. Performance and safety is "good enough", memory is abundant and cheap, and programmers are comfortable with their suite of tools. However, business, technology and user constraints can make certain classes of programs less tolerant of such trade-offs. For example:

- **Internet-delivered services**. Corporations are eager to adopt convenient techniques that deliver better performance, particularly when it translates to lower infrastructure costs. They would also welcome compile-time safety guarantees that help lower their risk to malicious hackers able to exploit unsafe memory practices.
- **Mobile and Internet of Things apps**. Such apps need to safely deliver increasingly rich and responsive capability using small devices that severely constrain available memory, CPU cycles and power.
- **Interactive 3D games**. 3D engines need to distribute and render massive, realistic, mutable 3D content reliably every 17 milliseconds. Satisfying this punishing requirement strains even high-end equipment. High-performance and responsive memory management is a difference-maker. Safety is increasingly becoming one as well, as content delivery and interactivity increasingly becomes Internet-centric.

Section 2 of this paper proposes a promising type system that gives such programs more control over choosing the integrated mix of custom allocation and garbage collection strategies that better satisfy its demanding requirements for compile-time safety guarantees, performance, responsiveness, memory utilization, flexibility and programmer convenience.

This paper's proposed type system builds on proven mechanisms baked into Rust and Pony, two recent, ground-breaking languages. Both offer comprehensive, compile-time safety guarantees that result from differing memory management strategies and type systems:

- Rust lexical memory management rests on its **single-owner alias model**, which uses compile-time knowledge about alias lifetimes, borrowed references and mutability exclusion to generate code that is memory-safe[Rust-own].

- Pony's reference-counted memory management is protected by its flexible **reference capability** model that ensures during compilation that concurrent data access is always safe[Pony-cap].

The proposed technique marries together their best-in-class alias lifetime (sections 2.2) and reference capability (section 2.4) type systems with these innovations:

- It introduces the allocator type (section 2.3), which enables programs to use multiple "plug and play" memory management mechanisms.
- It distinguishes allocator type from value type (section 2.3.2), allowing static enforcement of permissions even on "wrapped" values.
- It relaxes Rust's owner alias concept (section 2.3.3) to allow multiple, non-lexical owner aliases to refer to the same object.
- It adds the "lock" permission (section 2.4), another mechanism for static enforcement of shared, concurrent mutability.
- It promotes a "gradual" way (section 3) to take advantage of its interlocking mechanisms, so that a program can begin its life using a simple subset and then gradually make use of more advanced features as it adapts to more demanding requirements.

Section 2 does more than describe four alias types. It also describes aliasing mechanisms the compiler uses, such as lexical isolation, static enforcement, coercions, move semantics, and generated allocator behaviors, to deliver greater programmer flexibility and compile-time safety guarantees. At times, it includes code excerpts to help illustrate these mechanisms.

After explaining what "gradual memory management" means, section 3 assesses the benefits and costs of the proposed memory management type systems in terms of safety (section 3.1), performance and responsiveness (section 3.2), memory utilization (3.3) and programmer convenience (3.4).

Before we dive in, let me warn any reader expecting too much: **This is not an academic paper** that summarizes concrete research. It proposes no language specifications or detailed algorithms. It is too **early** for that. Since the technique has yet to be implemented nor formalized into a model, no measurements or proofs are offered that support the claims it makes. Since the described techniques are relatively new, supporting citations or relevant prior research work are hard to find and few in number.

That said, this paper is more than an outline or thought experiment. It is a detailed, carefully constructed, working hypothesis about a promising

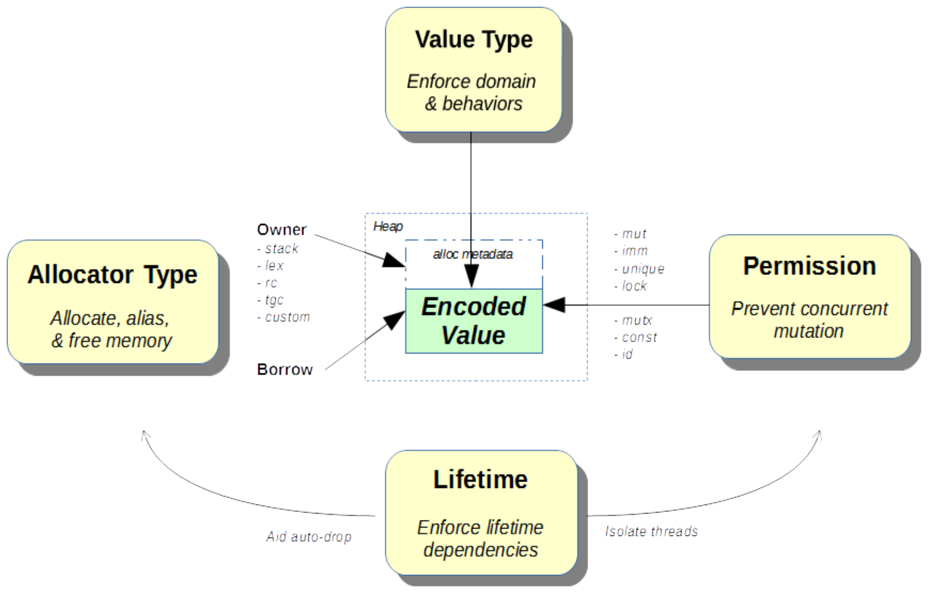

Figure 1: Alias Type Properties

technique. Sufficient detail is provided to helpfully guide a future implementation. Its claims may not yet be proven, but they rest comfortably on top of Rust's and Pony's working (and proven) implementations as well as established, hard-won wisdom about memory management trade-offs.

2 Aliases and their types

The complexity of memory management stems, in part, from programs holding multiple pointer references to the same data object. In this paper, we call every reference an alias. An alias can be a local or parameter variable, a specific element in a collection, or a function's return value. These references are called aliases because when you alter the data value through one alias, all other aliases that refer to that data value now see the new value, and not the old one.

In statically-typed languages, every alias is typed. This type information establishes constraints that the compiler enforces whenever the alias is used. Typing constraints make it harder to write code that fails to do what it is intended to do, thereby supporting a language's safety guarantees.

This paper's memory management technique relies on every alias having four types (see figure 1):

- **Value type**, the type of the object. This ensures the object only contains valid values. When applied to pointers, it ensures that the referenced object is alive and valid.
- **Lifetime**, the valid scope of an alias. This ensures that every object remains available so long as any alias refers to it and assists permission types with protecting against simultaneous mutation of the same object.
- **Allocator type**, the memory management technique that allocates and frees the memory needed by the object the alias refers to.
- **Permission type**, constraints on alias usage. This offers both lock-based and lock-less techniques for preventing mutability collisions between multiple aliases that reference the same object.

Static languages make use of value types. Some languages offer permission annotations. Rust introduced lifetimes. Allocator type, as a separate type for an alias, is new and lives at the heart of this paper. The purpose and rules are defined progressively for each, laying the foundation for the allocator type whose mechanisms depend on value type and lifetime information.

2.1 Value type

The value type restricts the domain of content values that an alias can refer to and the operations that can be performed on it. For example, the variable *counter* might only hold 32-bit integers (i32) and the variable *coord* may only hold a Point3D structure containing three floating point numbers for x, y, and z. Armed with this information, the compiler can stop us from trying to fit a square peg in a round hole. Thus, it would prevent us from trying to copy the data contents held by *coord* into *counter*.

Value types are well-trod ground, and do not need much exposition here. However, four aspects of value types are worth mentioning in the context of memory management: dynamic typing, size determinism, pointer safety, and 'copy' values.

Dynamic typing. Although this paper describes value types as being statically typed, the proposed memory management techniques applies just as well for dynamically typed aliases. It might seem odd for a language to specify lifetimes, allocators and permissions statically, while allowing value types to be dynamically typed. However, just such a situation would arise in a language that supports both gradual typing[Siek] and gradual memory management.

Size determinism. The memory size for all value types must be known to the compiler. This information is necessary when allocating memory for a new object of that type. This applies to dynamic value types as well, which are typically all encoded within a consistent, fixed-size, self-typed data structure.

Pointer Safety. If compile-time enforcement of memory safety is desired, the language needs to carefully isolate all pointer arithmetic to ensure that every alias always refers to a valid, live object of the intended value type. One way to accomplish this is to permit pointer arithmetic only within 'unsafe' blocks or allocator code. Language support for slices can also help.

'Copy' values. For performance reasons, many languages support what Rust calls 'copy' types: word-sized, primitive value types, such as integers and floating point numbers. Other languages call these primitive types or data types, and distinguish them from reference types. Space for 'copy' type values are wholly allocated on the stack or within structures. Reference type values have two parts: a pointer that references separately allocated contents.

A variable holding a reference type is an alias. A variable holding a 'copy' type is **not** an alias. Given that this paper's primary focus is on aliases, let's quickly address here a few aspects of 'copy' type handling.

Assignment has a different meaning for 'copy' value types (e.g., i32) than non-'copy' value types (e.g., Point), as demonstrated by this Rust excerpt:

```
let mut a = 2; // 2 is a copy value
let mut b = a; // b gets a separate clone of a's value
b = 3; // changing b does not change a's value
println!("{}", a); // a is still 2

// Point is not a copy value. Aliases change same object.
let mut p = Point {x: 3.3, y: 2.2};
{
  let q = &mut p; // q points to the same object as p
  q.x = 2.2; // changing q also changes p
}
println!("{}", p.x); // p.x is now 2.2
```

That said, it is possible to create multiple aliases that point to the same 'copy' value:

```
let mut a = 2;
{
  let b = &mut a; // b points to a's value
  *b = 3; // changing b also changes a
}
println!("{}", a); // a is now 3
```

This paper focuses on how to safely manage this sort of aliasing behavior where multiple aliases might point to the same allocated content.

2.2 Lifetimes

Most languages structure code using lexically-scoped blocks that establish a single "exit" point for all logic contained within the block. With this constraint in place (or by using control flow analysis as Rust intends to do for sub-lexical lifetimes[Rust-nll]), the compiler can easily infer the lifetime of most variable aliases, as they are bounded by the lexical scope they are declared within. The lifetime of such an alias begins with its declaration and expires at the exit from that block. Thus, the lifetime of an alias, unlike other alias properties, rarely requires explicit declaration.

Knowing the lifetime of an alias facilitates some valuable benefits:

- It helps determine when to free an object that is no longer needed.
- It protects against referencing an object that no longer exists.
- It provides a useful code isolation barrier for aliases owning exclusive rights to an object.

2.2.1 Lifetime-driven Frees and Moves

The lifetime of an alias and the lifetime of the object the alias points to are often not the same. The lifetime of an object spans from the time it was created (and memory was allocated for it) until, ideally, when we know a program will no longer access its contents. Since last access can sometimes be hard to determine, a more conservative strategy is typically employed, which establishes that the lifetime of an object expires when we no longer have any alias that refers to it. This means that an object's lifetime will never be shorter than the lifetime of any alias pointing to it, but will in fact span across the lifetimes of all aliases that refer to it.

The purpose of automatic garbage collection is to determine when to free an allocated object's memory after its lifetime has expired, when its contents are no longer accessible by any program alias. Freeing it too soon is unsafe. Freeing it too late or not at all can cause a program to require more memory than it actually needs.

Different garbage collection schemes vary in the heuristics they use to determine that an object's lifetime has expired. For example:

- **Reference Counting**. During run-time, count how many aliases refer to the object. The lifetime expires when the count reaches zero.

- **Tracing**. During run-time, mark all references reachable from the roots (execution stack and globals) as alive. Free any allocated object that is not marked alive.

- **Lexical** (e.g., Rust). During compilation, generate code that automatically frees an object at the end of the lexical block containing an owner alias that does not transfer ownership to another alias.

- **Manual**. The programmer uses their understanding of the program's logic to (hopefully correctly) determine the right time to explicitly free allocated objects.

A language can use the relationship between the lifetime of an object and the known lifetimes of its aliases to determine when to free an object that is no longer needed. For example:

- If we know for sure that only one alias refers to some object, as is true for lexical aliases, the compiler can safely conclude that the lifetime of the object and alias are the same. The compiler can then automatically and safely free each such object at the end of the block where the lifetime of its exclusive alias expires. Rust's "single owner" lexical memory management makes effective use of this mechanism for "zero cost" garbage collection.

- If we are keeping an accurate count of how many aliases currently refer to an object (as reference counting garbage collectors do), the compiler can accurately decrement the counter for an object whenever the lifetime of each "counted" alias expires at the end of its block, Once the decremented count reaches zero, the object's lifetime has expired; we can safely free it.

But what if we do not want a lexically-allocated object to be freed when the lifetime of its exclusive alias expires? Instead, it would be useful to be able to transfer exclusive ownership of an object from one alias to an alias in a different lexical scope, only freeing it at the end of its journey. This technique allows an object to "surf" across multiple lifetime scopes, as this Rust code illustrates.

```
struct Point { x: i32, y: i32}
fn create() -> Point
  { Point { x: 5, y: 10} } // allocate
fn transform(p: Point) -> Point
  { Point { x: p.x + 1, y: p.y + 1} } // free p & alloc
fn drop(p: Point)
  { println!("Drop {},{}", p.x, p.y); } // free p
fn main() {
  let p0 = create(); // Ownership moves from create to main
  let p1 = p0; // Ownership moves to p1.
  // p0 is no longer usable
  let p2 = transform(p1); // Two moves: p1 in, p2 out
  // p1 is no longer usable
  drop(p2); // p2 moves from main to drop
  // p2 is no longer usable
}
```

That is the purpose of Rust's move semantic (and Pony's consume): A new alias receives exclusive ownership of an object held by a previous alias. Once that transfer has occurred, the old alias is deactivated and can no longer be used to access that object. In a sense, we can say its lifetime has been abbreviated, expiring before the end of the block it is declared within.

To summarize: the lifetime end of any still usable (not moved) alias is a meaningful event to the compiler, triggering de-aliasing behavior that varies depending on the memory manager used to allocate the object. It might free the object, do some runtime bookkeeping, or do nothing. We will return to this topic in section 2.3.1.

2.2.2 Lifetime Dependency Safety

Another way that alias lifetimes are valuable relates to lifetime dependencies. In particular, we want to make it impossible to ask an alias to refer to an object whose lifetime we know will not last as long as the lifetime of the alias. Since this mechanism depends on compile-time information about lifetimes, it is useful only when applied to static (section 2.3) and borrowed (section 2.3.3) aliases. Lifetime dependency is not enforced with owner aliases that refer to objects managed by run-time or manual garbage collection.

The compiler's ability to prevent unsafe lifetime dependencies rests on the fact that the lifetimes of two active aliases are quite easily compared, as alias lifetimes are bound to lexical block scopes and lexical block scopes are always nested within each other. Thus, the lifetime of an alias declared in an inner block is knowably shorter than an alias declared in an enclosing, outer block.

Using this information, a compiler can ensure safe aliasing between two interacting aliases that have different lifetimes. For example, a compiler can generate an error whenever an alias is assigned a reference to an object that does not live as long as the alias, as Rust would here:

```
let mut a = 4;
let b = &a;
{
  let mut x = 8;
  b = &x; // Error: x lifetime is shorter than b
}
    // If allowed, b would refer to a freed value
```

A similar enforcement mechanism can also be applied when assigning one alias a reference to an object that another alias points to. The compiler can generate an error whenever a longer lasting (outer block) alias is assigned a reference to an object pointed at by a shorter-lasting (inner block) alias. This restriction is not always ideal, as we are using the lifetime of one alias as a proxy for the lifetime of its object. It won't make the program unsafe to prevent such assignments, but it may make the compiler overzealously protective (and inflexible) should it turn out that the referenced object will assuredly outlive its new alias.

This challenge is particularly acute when aliasing object references across functional boundaries, where the compiler essentially loses visibility to lifetime dependencies across both directions of a function call with regard to sent parameters and returned values. In this situation, the compiler requires guidance from the program about lifetime dependencies between parameters and return values in the form of lifetime annotations[Rust-life]. The compiler can use the function signature's lifetime annotations to prevent unsafe lifetime aliasing not only within the function itself, but also on all its callers.

For example, this Rust function uses the lifetime annotation 'a to indicate that the return value alias has the same relative lifetime as the parameters x and y:

```
fn max<'a> (x: &'a i32, y: &'a i32) -> &'a i32 {
  if x>y {x} else {y}
}
```

Will it cause a problem if we declare that the lifetimes of x and y are the same, when it is likely they are different? Fortunately no. For lifetime comparison purposes during function application, the compiler will coerce the lifetime of ‘a to be the shorter of the two aliases lifetimes. It will use this coerced lifetime to ensure the lifetime of the returned value will be handled safely by any function that makes use of the max function.

Lifetime annotations can also be valuable when a function needs to know (and enforce) that the lifetime of one parameter alias is greater than the lifetime of another parameter alias. Rust makes use of lifetime subtyping to assert that lifetime ‘b exceeds lifetime ‘a. For example:

```
fn foo<'a, 'b:'a>(mut na: &'a Point, mut nb: &'b Point) {
  na = nb;
  //nb = na; // Error: violates dependency 'b outlives 'a
}
```

2.2.3 Code Isolation Barrier and Move Semantics

There is one final benefit we get from knowing that the lifetime of some aliases are restricted to an inner block scope. So long as we harden that code isolation barrier, protecting carefully what comes in and out, we can use it to borrow an alias’s exclusive rights to an object within that inner scope and recover those rights upon exit from that inner scope. This recover block coercion mechanism will be discussed when describing permission enforcement.

2.3 Allocator Type

We have talked about aliases pointing to objects, but have not described how those objects are allocated or freed. Many languages use a single, default allocation/free mechanism: often a tracing or reference counting run-time garbage collector, but sometimes lexical memory management or “wild west” manual memory management.

The problem with restricting a language to a single memory management technique is that you restrict its programs to that technique’s particular strengths and weaknesses. Often, this restriction means sacrificing one or more of performance, flexibility, ease-of-use or safety. Enabling a language to support multiple memory management techniques reduces this sacrifice substantially, as it allows a program to optimize for both performance and flexibility without losing any compile-time safety.

That is the purpose of the allocator type, another declarable property of an alias: its use allows a program to specify, for each alias, the allocation/free mechanism that handles the object the alias refers to. By using different allocators on an alias-by-alias basis, programs gain the ability to exploit multiple memory management strategies.

Each allocator type encapsulates the static and runtime state and behavior of a specific memory management mechanism. For example:

- **local** might allocate fixed-size objects directly on the stack
- **lex** might mimic Rust's compile-time ("no cost"), single owner memory management
- **rc** might support a simple ref-counting garbage collection strategy.
- **tgc** might support generational, incremental mark-compact-and-sweep.
- **pool**, offering a performant, cache-friendly technique for allocating and freeing small, fixed-size objects.

This is not intended to be an exhaustive list of possible allocator types. Many variants of these techniques could also be allocator types, addressing concurrency, weak references, reference counting that uses tracing to handle cyclic data structures, and many other flavors of custom allocators. Ideally, a language would support the programmatic creation of allocator types, each optimized for different use-case constraints and requirements.

2.3.1 Allocator Behavior and Lifetimes

Allocators play well together with the lifetime mechanisms described earlier. Rather obviously, allocators wrap around the lifetime of an object: allocating the object at the start of its life and freeing it at the end. Less obviously, allocators also play a critical role with aliasing and de-aliasing (when an alias's lifetime expires).

Thus, every allocator type specifies its own distinct logic for these core behaviors:

- **Allocate**. This allocates a memory block for an object of a specific size. It may initialize metadata with that block, such as a counter (for RC), color flags (for tracing), or the finalization code to run before freeing the object. For tracing, it could also trigger runtime execution of some portion of its tracing or sweep logic.

- **Alias**. This copies a pointer for an object into a new alias. With RC, this would increment the counter. With incremental or generational tracing, this would activate the write (or read) barrier. With lexical, this would invoke move semantics, transferring ownership from the old alias to the new one.

- **De-alias**. With lexical, it frees the singly-owned object. With RC, it decrements the counter and frees when zero. With tracing, it does nothing, as the required trace and sweep activity is triggered independently from de-aliasing.

- **Free**. This de-allocates the memory block, making the space available for future use. Prior to this memory de-allocation, it might also be necessary to invoke value type-specific finalizer or destructor logic.

To illustrate how alias lifetimes (section 2.2.1) interact with these allocator behaviors, consider this simple pseudo-code. 'rc' designates a reference counting allocator type:

```
struct Point { x: i32, y: i32}
fn transform(p: rc Point) {
  p.x = p.x + 1;
} // De-alias of p decreases ref count to 1

fn main() {
  // Allocate Point structure using rc allocator
  rc p1 = Point { x: 5, y: 10};
  // Function call aliases p1, increasing ref count
  transform(p1);
} // p1 de-alias decreases ref count to 0. It is freed.
```

If we change this code's 'rc' to 'lex' (lexical allocator), it would still work. However, instead of altering the alias's reference counter with each alias and de-alias, the call to transform() would move p1 into that function and then free it at the end of that function.

In addition to the four core behaviors listed above, allocator types might also specify additional behaviors, such as support for weak aliases or the ability to alter various run-time configuration controls. Allocator types that use a "global" state would need start-up logic to initialize the state when the program is run and clean-up logic that is performed upon program termination.

An allocator type's programmatic behaviors would be generated by the compiler as inline code, some of which may call API's whose implementation is loaded as part of the appropriate companion library. Additionally, the compiler will need to automatically generate tracing and sweep logic for all data structures (including the stack) that refer to objects allocated by a tracing GC allocator, so that all owned references allocated by the tracing allocator can be comprehensively marked and freed. The compiler might also need to generate concurrent GC safe points for execution points when it knows the stack map is valid for GC activity.

2.3.2 Allocator vs. "Wrapper" Value Types

Use of allocator types is one way to support multiple memory management strategies. Rust accomplishes this in a different way. It offers built-in "wrapper" value types[Rust-wrap]. For example, Rc<RefCell<T>> supports single-threaded reference-counted memory management for an object of type T.

Supporting run-time garbage collection this way, applying wrappers around the content's value type, can result in unnecessary complexity and lost capability:

- The zealous single-ownership protections that are necessary for lexically allocated objects are too restrictive when applied to runtime allocated objects that benefit from allowing multiple pointer references.
- Mutation guarantees cannot be statically enforced on wrapped values, requiring the need for "interior mutability" runtime mechanisms.
- The coding logic required to borrow from and work with wrapped values is unnecessarily verbose and hard to learn.
- The compiler lacks the ability to generate the extra tracing logic needed for a tracing GC.

Cleanly separating allocator types from value types, as this paper proposes, offers attractive benefits:

- The compiler and programmer can work with every kind of allocator type as simple plug-and-play equals, adjusting gracefully to how they handle their memory management strategy differently under the covers.
- The compiler can enforce every alias's permissions (section 2.4) directly to the referenced object, at compile-time, regardless of where that object came from.

2.3.3 Owner vs. Borrowed references

Rust distinguishes between owner aliases and borrowed reference aliases:

- In Rust, only one **owner alias** can refer to a specific object. This single-owner constraint is critical to lexical's automatic memory management. It powers the aliasing "move semantic" (section 2.2.1) which enables an object to surf across multiple functional scopes. It also powers the automatic object free when the lifetime of its last alias ends.

- Multiple **borrowed reference aliases** may borrow an object's reference from any owner or borrowed alias. Borrowed references behave differently than owner aliases. Borrowed references are passed to functions as simple copies that do not transfer ownership. Likewise, when the lifetime of a borrowed reference alias expires, no object free behavior is triggered.

  Handling and passing around borrowed references is safe because borrowed references are guaranteed by the compiler to have a shorter lifetime than the owner alias they borrowed from (section 2.2.2). When we use a borrowed reference, we know at least one owner reference also exists that ensures the object stays alive and safe to reference.

This paper's proposal adopts this distinction and applies it to allocator types. Every alias using the allocator types mentioned so far (e.g., lex, rc, tracing gc) is an owner alias. However, unlike Rust, not all owner aliases are constrained to be single-owner aliases. Only the static allocators (such as lex) preserve this constraint. For run-time managed objects, multiple owner aliases may point to the same object at the same time.

Borrowed references are handled as a special allocator type: 'borrowed'. The 'borrowed' allocator type plays no role in allocating or freeing memory for objects. It is just a signal for the compiler to treat borrowed aliases using a special rule set, exactly as described above for Rust.

The following pseudo-code (adapted from section 2.3.1) illustrates the use of borrowed aliases. Like Rust, it uses '&' to declare a borrowed reference:

```
struct Point { x: i32, y: i32}
fn transform(p: &Point) {
  p.x = p.x + 1;
}

fn main() {
    rc p1 = Point { x: 5, y: 10}; // owner alias
    transform(p1); // alias to a borrowed reference
  }
```

A borrowed alias "forgets" the allocator type of the owner alias it borrowed from; it is just a pointer to an object's typed contents. As such, there is only one 'borrowed' allocator type, as opposed to multiple borrowed types, one per owner allocator type. Borrowed aliases are a melting pot across allocator types. And since it does not know the owner's allocator type, the borrowed alias generates none of the allocator-specific aliasing and de-aliasing behavior that an owner alias would, including runtime GC bookkeeping.

This freedom from allocator-specific behaviors makes borrowed references attractive to use:

- **Program performance:** Borrowed references avoid the bookkeeping cost imposed by runtime allocators, since aliasing using borrowed references has no impact on the object's reference counter nor are borrowed references ever traced.

- **Code reuse:** functions using only borrowed references for parameters will support object access and mutation regardless of the object's allocator type.

Because of these benefits, functions (and methods) should be declared to use and return borrowed references whenever possible. Notable exceptions would obviously be functions that create or destroy objects (constructors and destructors). Another exception would be functions that store an object reference within a data structure of undetermined lifetime.

Note: Functions using borrowed references may have to specify lifetime annotations when lifetime dependencies (section 2.2.2) exist between parameters and return values. Also, use of borrowed aliases requires special handling if borrowing from a moving-GC allocator, one which fixes alias references whenever an object is moved to a new location. If any borrowed reference might point to that object, the GC's trace map has to be complete enough to find and fix it.

2.3.4 Allocator Polymorphism

Although allocator type is distinct from value type, it is still a type. This means that functions that use owner aliases as parameters or returns values will only work for those aliases' specified allocator types. Thus, a function that creates a new Array managed by the lexical allocator must be different from a similar function that creates a new Array managed by a tracing GC allocator. Such parametric polymorphism is a well-known type challenge for languages.

A popular solution for this challenge is generic programming or templates, which allow the programmer to define a single function implementation that the compiler can appropriately generate for all the allocator types it is applied to. Another approach might involve passing the allocation type as a separate runtime parameter (e.g., Haskell typeclasses).

Extensive use of borrowed references minimizes this polymorphic overhead.

2.3.5 Allocator Cross-References

Each allocator has a collection of allocated objects it manages. Because every allocator's heuristic never deletes any object that is still referred to, reference validity is guaranteed between all objects it manages. However, reference safety cannot always be ensured for references that cross allocator boundaries, as it would be entirely possible for one allocator to delete an object that another allocator's object still refers to, since those references might be unknown to it.

One foolproof way to ensure memory safety would be to prevent all allocator cross-references. In this scheme, every allocator is treated like walled garden of objects, such that:

- Every object belongs to only one allocator.
- Every owner alias for the same object maps uses the same allocator type.
- Every object an object refers to also belongs to the same allocator.

Compiler enforcement of allocator walled gardens is straightforward:

- Since allocators are types, generate a "type mismatch" error any time a reference held by one allocator is assigned to an alias declared to a different allocator.
- Do not allow fields within a collection (e.g., a structure or array) to declare an allocator type. This ensures that an object can only refer to another object within the same allocator.

However, this simplistic scheme is unnecessarily restrictive, as it forbids potentially useful cross-allocator references that a compiler could still guarantee to be safe. For example:

- **Borrowed references** have a different allocator type than the owner aliases they borrow from, yet the compiler can use lexical lifetime dependency checks to ensure the borrowed alias always expires before the owner alias.

- **Runtime GC allocator cross-references**: A tracing GC reference can be safely nested within an RC-managed object. The reverse is also true. Cross-references between runtime allocators must be done using Indirect object nesting, as direct multi-allocator references won't work. For example, an RC-typed reference would improperly try to increment a non-existent reference counter for an object that was allocated by a tracing GC and therefore holds a different kind of bookkeeping metadata.

Given these examples, the compiler can still enforce reference safety by following these more permissive rules:

- **Aliasing**. Allow any allocator's references to be aliased to a borrowed reference. Generate an error message for any other allocator type mismatch.

- **Collections**. Allow a structure's fields to declare a runtime GC allocator. However, if a structure has fields that declare a specific allocator, only allow that structure to be allocated by a runtime GC allocator.

The walls can be softened further by allowing programs to copy or move content between allocators. This capability is conceptually similar to value type conversions, except here we are "converting" an allocated object from one allocator to another. Such conversions require a cooperative handshake between the sending and receiving allocator. For example:

- **Copy**: The receiving allocator allocates space for a new object (setting up the appropriate metadata) and copies the content pointed at by a borrowed reference. Particular care must be taken when copying over an object that references other objects; any copy sent across must be a "deep" clone that comprehensively ensures the entire chain of referenced objects are also copied. Quite obviously, any mutation of one copy will have no impact on the contents of the other.

- **Move**: This works like a copy, except the sending allocator deletes (or de-aliases) the copied object after the copy is complete. If the moved object references other objects, a "deep" move of all referenced objects must be performed. Such a deep move might require copying, to ensure all moved objects have the correct bookkeeping metadata for the new allocator. In some cases, such as resizeable objects, the move might only need to copy header information and not the content pointed at by the header, whose ownership is just transferred over.

2.4 Permissions and Race safety

A language promoting memory safety needs to safeguard concurrent mutation. It is known: concurrent mutation of the same object is dangerous. This risk can be mitigated, however, by constraining any of these rights: aliasing, mutation, or concurrency. The strong concurrency guarantees that Rust and Pony make are enforced by their use of such lock-less, compile-time permission constraints applied to every alias.

Let's clarify the meaning of alias mutation. For an alias that refers to an object, it could mean:

- Altering the alias to point to a different object
- Changing the contents of the object it points to

We use the latter meaning from now on when we talk about any permission that allows or forbids an alias from being used to mutate an object.

There are several permission schemes that languages offer to help manage race safety. Pony offers the most extensive approach, defining six distinct permissions that it calls reference capabilities[Pony-cap]. These will be introduced one-at-a-time, in the context of describing permission schemes that are a subset of Pony's. Each permission is given a name that is more familiar (and hopefully easier to remember) than Pony's chosen names.

2.4.1 Locked mutation

Many imperative programming languages treat aliases as mutable by default. No aliasing restrictions are imposed on them within and between threads. To prevent concurrent mutation, the programmer manually wraps locks (or some other synchronization mechanism such as CPU intrinsics) around any get or put access to thread-shared aliases that refer to alterable objects.

This manual approach has well-known safety and performance drawbacks. With regard to safety, it is notoriously difficult to ensure synchronization mechanisms are consistently and correctly specified. As for performance, synchronizing frequently accessed objects can noticeably slow down a program's throughput or create deadlocks. Despite these drawbacks, there are concurrency problems that require synchronization to handle correctly. For example, concurrent GCs that manage cross-thread aliases to the same object require the use of synchronization at critical moments (e.g., when altering an object's reference counter).

To improve safety, a language's compiler could be enriched to validate that locks are properly specified on aliases shared between threads. This scheme requires that every alias has one of two permissions:

- **mut**. A mut alias may access or change the contents it refers to. Multiple aliases may exist that point to the same object, but only within a single thread. mut aliases may never be sent to or referenced by any other thread. (Pony calls this ref).
- **lock**. A lock alias may access or change the contents it refers to. Multiple aliases may exist that point to the same object in any thread. However, the compiler protects against concurrent access by requiring that any read or write to that reference is always performed behind a specified atomic lock.

2.4.2 Immutable-only

Some functional programming languages take an opposing approach. They eschew all mutation, instead building programs that use only immutable objects, never changing their contents after they are created. In this scheme, all aliases are implicitly declared:

- **imm**. An imm alias may retrieve, but never change, the contents it refers to. Multiple aliases may exist that point to the same object in any thread. Such objects are globally accessible and immutable. (Pony calls this val).

There are several benefits to this approach. Programs can be composed using pure functions, making them easier to reason about using formal proofs. Immutable-only programs support concurrency without any additional constraints.

There are drawbacks to immutable-only. The performance and memory churn of complex, immutable data structures is often noticeably worse than their mutable equivalents. Furthermore, some programmers do not value constraining their code to use only immutable data structures and pure functions that eschew all side effects.

2.4.3 Exclusive mutation

To marry together the distinct benefits of mutable and immutable aliases, some languages support both. This is what Rust does, but with an important twist: its mutable reference permission is different from the one defined earlier, as Rust allows only one mutable alias to an object at a time. To distinguish Rust's mutable permission from mut, we give it its own name:

- **unique**. A unique alias may access or change the contents it refers to. At any time, only one such alias has permission to mutate its object and no other able to read it. However, a unique alias may transfer its singular object reference to another alias (even one declared in another thread), after which point the original alias is no longer usable. (Pony calls this iso)

This lockless scheme offers several useful advantages. We can use mutable data structures that perform better and simplify algorithmic complexity. Additionally, safe concurrency becomes a bit faster (without locks) and more flexible, since mutable data may now also be transferred from one thread to another. Indeed, unique aliases do not just make concurrent mutability safer, they also make single-threaded mutability safer. The unique permission prevents a number of data interference problems that can arise when multiple aliases may be created that refer to the same mutable object.[Rust-single]

There are downsides to unique, most notably that it prevents a program from creating multiple mutable aliases to the same object. This single-alias restriction of the unique permission makes it impossible to create a number of useful mutable data structures, such as double-linked lists, graphs that include multiple-use nodes, or cyclic, self-referential graphs.

2.4.4 Midori's permissions

For several years, Microsoft researchers worked on an internal operating-system project code-named Midori[Duffy-perm]. Its purpose was to create a safer foundation for hosting fast, concurrent-capable programs. Its design increasingly centered around lockless, language-declared permissions whose concurrent safety could be enforced at compile time. There are many notable design decisions in common between Midori and Pony.

A Midori alias was declared using one of four permissions: mutable, isolated, immutable and readonly. These correspond exactly to the permissions we call: mut, unique, imm, and:

- **const**. A reference to an object which may not be changed by this alias. Other const, imm or mut aliases may exist that also refer to the same object. However the const aliases are restricted to the same thread. A const alias may not send or share its object reference with another thread. (Pony calls this box).

The difference between imm and const is subtle but important. Declare a function parameter to be const for an object reference that the function promises not to change. The function does not care if the reference was borrowed from a mut, imm, or const alias. However, since it is declared const (rather than imm), the function is prevented from passing this potentially mutable reference to another thread.

Midori's permissions extend the benefits of the exclusive-mutation scheme. The mut permission makes possible the use of mutable network graph data structures that incorporate multiple references to the same object. Such thread-specific data structures can sometimes even be packaged together into a unique object reference which can be bounced around between threads.

2.4.5 Pony's permissions

Pony adds two more permissions beyond Midori:

- **mutx**. An exclusive mutable reference that can be held by only one alias at a time and cannot be transferred to another thread. Multiple immutable references may be borrowed from it (via const), but only in the same thread. (Pony calls this trn).

- **id**. Multiple id references can exist of the same object in multiple threads. One cannot read or write to the content of this object, but can still compare identity and call methods on the object. (Pony calls this tag).

2.4.6 Permissions Summary

Choosing between seven permissions feels overwhelming when meeting them for the first time. How does one quickly determine which is the right one for each alias, in a way that plays well across many functions and modules? It is not as difficult as it first seems.

Owner aliases generally use one of four permissions:

- **mut**. The natural default for changeable data. It cannot leave its thread.

- **imm**. For data that never changes after creation. It is shareable everywhere.
- **unique**. The natural default for data created by a constructor, as it transitions easily to mut or imm. Also, unique is useful for sending mutable data to another thread.
- **lock**. For changeable data shared by multiple threads.

The other permissions are usually used by borrowed references as function parameters. They allow a single function to gracefully accept multiple permissions on the object references they get, thereby avoiding having to deal with permission-based parametric polymorphism:

- **const**. The natural default for data the function will not change, accepting any other permission (except lock and id).
- **mutx**. For data the function wants to be able to change, accepting unique and mut.

id is typically used by a thread to refer to another thread for communication purposes. It can also be used to compare that two known objects are the same. An id alias is borrowed from an alias of any other permission.

2.5 Permission Enforcement & Coercion

The compiler constrains alias use using principles derived from the rights that a permission grants or denies:

- Fail prohibited content access. Any attempt to access or alter the contents referred to by an alias will fail to compile if the alias does not have the appropriate read or write permissions. With lock, all access must be contained within an atomic lock.
- Fail unsafe aliasing. If a new alias attempts to gain rights that the old alias does not have, the compiler should fail the program with helpful error messages.
- Support safe coercions. Allow a new alias to specify a different permission than the old alias, so long as permission rights are further curtailed and not expanded.
- Honor aliasing restrictions for unique and mutx. The old alias should be made unusable for the lifetime of a new alias (permanently if ownership has been transferred to the new alias).

Let's examine in detail how that plays out when aliasing within a thread, to a data structure, and between threads.

2.5.1 In-Thread Aliasing

This table uses 'x' to show legal coercions when aliasing between old aliases (rows) to new aliases (columns). "move" indicates either that ownership is transferred if new alias is an owner or that the old alias is not usable until the new borrowed reference alias goes out of scope.

	unique	**mut**	**imm**	**lock**	**mutx**	**const**	**id**
unique	move	move	move	move	move	move	x
mut		x			x	x	x
imm			x			x	x
lock				x			x
mutx					move	move	x
const						x	x
id							x

2.5.2 Combining Properties for Structures

A layer of complexity is added when dealing with data structures whose named fields each specify their own permissions. When accessing or aliasing a field, the compiler computes and enforces a new permission that combines the restrictions of the alias referencing the data structure (row) and the permission of the requested field (column).

	unique	**mut**	**imm**	**lock**	**mutx**	**const**	**id**
unique	unique	id	imm	id	id	id	id
mut	unique	mut	imm	lock	mutx	const	id
imm	imm	imm	imm	id	mutx	const	id
lock	n/a	n/a	n/a	n/a	n/a	n/a	n/a
mutx	unique	mutx	const	lock	mutx	const	id
const	id	const	imm	id	const	const	id
id	n/a	n/a	n/a	n/a	n/a	n/a	n/a

2.5.3 Cross-thread Communications

Let's first define what we mean by thread. For this paper, thread refers to a variety of mechanisms that allow multiple execution stacks to run concurrently or in parallel, such as OS-managed threads, actors, Erlang processes, or cooperative green threads (e.g., co-routines or generators). The term thread does not encompass OS-managed processes that cannot share memory in common and must communicate via sockets.

Because threads are often asynchronous, communications between threads typically go in one direction: a send that does not wait for data to be received back. Permissions carefully control what sort of data can be sent to another thread. One cannot send either stack or borrowed reference aliases to another thread (the latter because we lose the ability to enforce lifetime dependencies).

Owner aliases may be sent only if they have the unique, lock, imm or id permission. unique always moves ownership of the data to the new thread. With the other permissions, ownership is transferred if it is lexical, otherwise these aliases must be protected by a concurrent, runtime GC allocator type.

2.5.4 Recover Block

Pony supports a special lexical structure called a recover block. It allows one to "open up" a unique value, work with it under isolated conditions and then close it back up to a unique or some other permission. Such a block is valuable for Pony, which does not support borrowed reference aliases. It may be redundant in a language that does support borrows.

Although the recover block executes within a thread, it is isolated as if it were a separate thread. Like thread sends, it restricts access to outer scope variables that are unique, lock, imm and id. The value it returns from inside the block may be cast to unique, mut or mutx (if mutable) or imm, const, or id otherwise.

3 Gradual Memory Management

The previous section presented the mechanisms that underlie four typing properties of aliases: value type, lifetime, allocator type, and permissions. Let's now explore how they work together to give programmers better control over memory use with regard to compile-time safety guarantees, performance, responsiveness, memory utilization, flexibility and programmer convenience. At the heart of this synergy lies the concept of gradual memory management.

The notion of making memory management "gradual" is inspired by gradual typing[Siek]. Although some may view gradual typing as a graceful way to evolve a dynamically-typed program into one that specifies static types, it is more properly viewed as a technique that gives the programmer a choice for every variable, whether to constrain it to a static type or else let its type be determined by the value(s) it receives at runtime.

Gradual memory management captures a similar idea: it offers the programmer a choice over how memory is used on an alias-by-alias basis. In the context of this paper, a language supporting gradual memory management could deliver three distinct value propositions:

- It is not opinionated. Rather than dictate the memory management strategy your program uses, the language lets you pick the strategy that best addresses your priorities. Alternatively, the language uses lexical and runtime performance analysis to algorithmically select the optimal mix of memory management strategies.

- It is not fussy. It offers programmer-friendly defaults and opt-in constraints and features so that the programmer need not learn the entire language nor fight a nagging compiler to write useful working programs. Programmers can incrementally learn and use more advanced features over time to take advantage of additional benefits.

- It is helpful. The language gives you new capabilities you never had before, such as the ability to safely integrate multiple memory management strategies within the same program.

By offering more choice, gradual memory management allows the programmer to optimize memory safety, performance, memory utilization and programmer convenience, with fewer trade-offs. Let's explore each of these in more depth.

3.1 Compile-time Safety

One of the primary benefits of typing systems is the role they play in enforcing safety constraints at compile time, long before the program has a chance to make a mistake during execution. With regard to memory management, Joe Duffy's three safeties[Duffy-safe]are a good summary:

- **Concurrency safety** prohibits unsafe concurrent, mutable use of shared memory. The compiler enforces this using the alias permissions, aided by lifetime information.

- **Type safety** prohibits use of memory that is at odds with the type allocated within that memory. The compiler enforces this using the value type.

- **Memory safety** prohibits access to invalid regions of memory (e.g., buffer overflow, use after free, and double frees). The compiler uses types to enforce this:
  - Constraining pointer arithmetic so that references always point to valid data.
  - Lifetime checks on aliases whose allocator type is lexical or borrowed.
  - Automatic memory allocator types that free objects when all references to it are gone.

This list is comparable to the robust, compile-time safety guarantees claimed for both Pony[Pony-safe] and Rust[Rust-safe]. These safety guarantees are not only successfully enforced by their respective compiler implementations, but are often also backed up by formal type system proofs. An additional benefit of enforcing safety at compile-time is that it rarely imposes any runtime performance cost.

There is room for flexibility with regard to safety. For example, Rust supports an "unsafe" block, wherein type violations will not be enforced, thereby moving the burden of trustworthiness to the programmer. Likewise, a program can safely use any of the five previously-described permutations (subsets) of the seven permission types. They may vary with regard to flexibility or performance, but they are all equally safe.

3.2 Performance and Responsiveness

A program's performance profile depends greatly on its design and the complex interplay of many factors beyond memory utilization. Thus, gradual memory management is not a magic bullet for improving performance, a notoriously challenging art. However, offering programmers greater control over memory management through the ability to select the optimal allocator and permission type on an alias-by-alias basis opens the door to more aggressive tuning of a program's performance.

Let's begin by comparing the performance profiles of different allocator types.

Runtime garbage collectors, such as RC and tracing, incur a performance overhead due to the object lifetime expiration bookkeeping they do regularly throughout the execution of the program: either keeping an accurate, ongoing count of references or else tracing all references from the roots to determine which are alive and which can be swept. This performance overhead increases as the number of allocated objects and references grows.

In addition to performance overhead, runtime GCs (particularly tracing) are more likely to struggle with maintaining a predictable real-time responsive to events, due to sporadic stop-the-world pauses at unexpected times. Such pauses are necessary whenever the garbage collector needs to perform atomic bookkeeping work whose integrity must not be damaged by continued execution of the program's logic.

By contrast, lexical memory management does not suffer from these bookkeeping costs, as object lifetimes are calculated at compile-time. All other factors being the same, lexical and runtime memory management incur similar runtime costs for dynamically allocating and freeing memory, but lexical's throughput will likely be faster by avoiding the performance overhead of runtime bookkeeping. Similarly, lexical's responsiveness will be better as it avoids the unpredictable stop-the-world pauses caused by runtime bookkeeping. Long pauses are still possible, but less likely and easier to manage deterministically. These benefits also apply to manual memory management, since the programmer performs lifetime bookkeeping in advance of program execution.

Comparing allocators solely on throughput and responsiveness, who would not prefer to use static memory management? Unfortunately, lexical memory management only works brilliantly for single-owner objects, failing with more complex multiple-owner data graphs. Additionally, mutable global data structures managed by lexical GC can leak memory.

Hybrid use of multiple allocator types within the same program offers a valuable middle ground between these two extremes. Use a runtime GC allocator type for global data and data that requires multiple owner references. Use a static allocator for all other working data for the program. The runtime GC performance of the hybrid approach will be smaller than if all objects are managed by a runtime GC, since the hybrid program has fewer objects to manage (usually new generation churn). This runtime overhead can be shrunk further by using borrowed aliases, which helps reduce how many references the runtime GC needs to trace or count. This hybrid approach also reduces the frequency and duration of stop-the-world pauses, thereby improving responsiveness.

Further performance benefits can be realized through selective use of stack or custom allocators, such as object pools or generational arenas, which can speed object allocation, reduce virtual memory paging and optimize cache locality, as compared to general-purpose allocators.

Broader use of permission types can also help improve performance over more restrictive permission schemes:

- Lockless concurrency mechanisms often run faster than mechanisms based on locks (by avoiding synchronization and thread switching delays) or “shared nothing” message passing (by reducing data cloning).

- Being able to mark data as **imm** allows the compiler to optimize for performance. The Midori project measured a significant performance boost from this, particularly with regard to global data.

- Being able to safely alter mutable data structures in place reduces the overhead of allocating space every time immutability might require the creation of a new altered object.

3.3 Memory utilization

It is the program’s responsibility to ask for the memory it needs. Alias types do not vary much in how much memory a program requests. However, the choice of allocator type can minimize memory fragmentation and leaks, which left unchecked might push overall memory use beyond the constraints of the program’s device (e.g., mobile or embedded system):

- To minimize memory leaks, particularly for long-lived mutable data structures, runtime GCs are a better choice than lexical GC, as the runtime GC may notice the lifetime expiration of such objects more quickly than when its owning lexical block, particularly the program’s main() function, finishes executing.

- To minimize memory fragmentation, a program can make use of space-efficient allocator types, such as mark-and-compact, object pools or arenas.

3.4 Programmer Convenience

This paper proposes a complex, clockwork mechanism. It borrows intricate gears from Rust and Pony and then injects its own intricate gears. Given the number of people that find Rust and Pony hard to learn precisely because of their gears, does this composite memory management mechanism even have a chance to be easy on a programmer?

Yes.

The secret sauce lies with its “gradual” nature. Similarly to how gradual typing languages ease a program’s transition from dynamic types to static types, gradual memory management facilitates a graceful transition from an easy, but workable subset towards more advanced solutions that take advantage of its flexibility.

One simple, but effective starting point for a language would be one whose default allocator is a tracing GC and whose default permission is mut. Use of such a language subset need not be any harder-to-learn nor more verbose than existing imperative languages. The programmer is fully protected by safety guarantees, but does not have to fear that their first exposure to the language will involve wrestling with the borrow checker, selecting the right allocators, or becoming comfortable with the nuances of seven different permissions.

Over time, as the skill of the programmer improves and the requirements for the program become more stringent with regard to performance or concurrency, the programmer can progressively make use of additional permissions and allocation types. The programmer might then add one more allocator (e.g., lexical) or permission (e.g., imm) to their palette. And so on, until the programmer has become masterful at making and assembling all needed gears to build performant, concurrent programs. This gradual progression can be facilitated by language documentation that goes beyond just technically describing the options, and also offers helpful guidance on where each option is best used.

Furthermore, if the language chooses defaults wisely, declarations need not become cluttered with permission and allocation type information. Borrowed references will be the norm and should be the easiest to specify. One can follow Pony's lead as well: its chosen permission defaults are good enough to avoid the need to explicitly declare one most of the time.

4 Conclusion

One of the holy grails of 21st century programming is the search for a systems programming language that is as flexible and performant as C/C++, as safe as Rust and Pony, and as rich and convenient as Python.

This paper proposes a comprehensive memory management mechanism that brings us closer to this goal. This mechanism marries together Rust's powerful lexical lifetime-based owner/borrow model with Pony's flexible reference capability permission model, including fusing together their distinctive approaches to compile-time enforcement of memory safety. It then adds to that synergistic mix several useful innovations:

- It introduces the allocator type, supporting the use of multiple "plug and play" memory management mechanisms by owner aliases. This improves the optimization of complex programs for performance, data structure flexibility, and memory efficiency, while preserving compile-time safety, thus reducing the restrictive trade-offs programs now have to make based on their chosen programming language.

- It relaxes owner aliases to selectively allow multiple aliases to refer to the same object. This improves data flexibility and programmer convenience.
- It distinguishes allocator type from value type, allowing static enforcement of permissions even on "wrapped" values. This improves performance and programmer convenience.
- It introduces the "lock" permission, another mechanism for static enforcement of shared, concurrent mutability. This adds both safety and flexibility, when lockless concurrency is insufficient.
- It promotes a “gradual" way to take advantage of its interlocking mechanisms, so that one can use a simple subset for simple problems and then gradually make use of more advanced features to handle more complex requirements. This improves programmer convenience, making these valuable features more palatable to people who are unfamiliar with them or uncomfortable by their constraints.

The technique described in this paper is currently nothing more than a promising idea, even though built on already-proven innovations. Hard work and good fortune will be needed to build formal proofs and coded implementations that demonstrate its viability and realize its promise.

Acknowledgments

I would like to thank Christopher J. Hall for his invaluable inspirations, helpful references and explanations, probing questions, thorough editing, and enthusiastic involvement as these ideas travelled from a wish into a promising framework. By every measure, this paper is significantly richer for his contributions.

I would also like to thank Paul Bone, Mike Lewis, Konrad Podlawski, Matthieu M, and Jared Weakling for taking time out of their busy lives to review early drafts of this paper and for offering a wealth of valuable feedback that strengthened its clarity. I am also grateful to Andy Chu for his wise warning that helped ensure the proposed framework was made gradual and optional.

I would be remiss not to mention the r/ProgrammingLanguages subreddit community, to which we all belong. Our many conversations and rants about memory management schemes opened my mind and provoked me to explore better ways to address decades-old challenges.

Finally, it should be obvious this framework would not exist without the excellent pioneering work accomplished by the Rust, Pony, and Midori teams, along with their inspiring predecessors. I salute you!

References

[Duffy-perm] Joe Duffy: 15 years of Concurrency. http://joeduffyblog.com/2016/11/30/15-years-of-concurrency/

[Duffy-safe] Joe Duffy: A Tale of 3 Safeties. http://joeduffyblog.com/2015/11/03/a-tale-of-three-safeties/

[Pony-cap] Pony Reference Capabilities. https://tutorial.ponylang.org/capabilities/

[Pony-safe] Pony Safety Guarantees. https://tutorial.ponylang.org/

[Rust-life] Rust Lifetime Annotations. https://doc.rust-lang.org/book/second-edition/ch10-03-lifetime-syntax.html

[Rust-nll] Rust non-lexical lifetimes. https://github.com/rust-lang/rust-roadmap/issues/16

[Rust-own] Rust Ownership. https://doc.rust-lang.org/book/second-edition/ch04-00-understanding-ownership.html

[Rust-safe] Rust Safe and Unsafe. https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html

[Rust-single] The Problem with Shared Mutability. https://manishearth.github.io/blog/2015/05/17/the-problem-with-shared-mutability/

[Rust-wrap] Choosing your Guarantees: Wrapper types. https://doc.rust-lang.org/nightly/book/first-edition/choosing-your-guarantees.html

[Siek] What is Gradual Typing? https://wphomes.soic.indiana.edu/jsiek/what-is-gradual-typing/