Linear Scala is All You Need for Safe Static Memory and Alias Management

by

Amirhossein Pashaeehir

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2026

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Rust has become one of the most popular languages for systems programming. This popularity is largely driven by Rust's ability to provide safe static memory management without garbage collection, eliminating GC-induced pauses and runtime overhead that can be difficult to predict and control. In addition, Rust enforces alias and mutability control through its ownership and borrowing discipline, enabling features such as fearless concurrency and stronger compiler optimizations while preserving memory safety. Scala Native brings Scala to systems-level targets by compiling to LLVM IR. However, it still relies on third-party garbage collectors for memory management and does not provide Rust-style static guarantees for safe memory management, aliasing, and mutability control.

This thesis presents *imem*, a library that brings Rust-inspired ownership and borrow checking to Scala, and *Scinear*, a minimal compiler plugin that adds linear types to Scala and integrates them with capture checking and polymorphism. imem proves that, given Scala's type system, linearity is the only missing ingredient needed to implement most of Rust's ownership and borrowing discipline as a library rather than a dedicated language feature. imem provides linear `Box` values and immutable and mutable references, enforces ownership rules, and statically controls aliasing and mutability, following the Stacked Borrows model. In addition, imem offers optional runtime verification to detect potential safety violations when users apply workarounds to the static rules.

To demonstrate practicality, the thesis develops a safe linked-list case study and compares imem against Rust, vanilla Scala, and linear Scala. The evaluation shows that imem matches Rust's level of expressiveness, so it can support list operations and iterators alongside statically enforcing ownership rules and controlling mutability.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Ondřej Lhoták, whose kindness, guidance, and support made this thesis possible.

I am deeply grateful to my partner, Niloufar, whose unwavering love and encouragement provided a foundation of strength throughout my graduate journey.

Dedication

This thesis is dedicated to the people of Iran, whose steadfast courage and persistence keep pushing forward despite hardship and obstacles, in the ongoing pursuit of freedom. May we, as Iranians, soon reach a time marked by peace and liberty.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Rust is popular in systems programming because safe memory management and alias control are built into the language. Static memory management removes the need for garbage collectors, which can introduce performance problems. Garbage collection ranges from stop-the-world pauses to more complex approaches such as generational garbage collection. However, these approaches can still slow programs at runtime in ways that are not fully controllable by the programmer. Therefore, systems programmers often prefer languages that are not garbage collected.

Removing garbage collection can improve performance, but it also increases the risk of unsafe memory management. Unsafe memory management can lead to use after free errors, double free errors, and memory leaks. Manual memory management, as used in C and C++, is vulnerable to these errors. Rust addresses these problems through ownership rules and borrowing, which limit the set of valid programs and enforce memory safety statically.

In addition, alias and mutability control enables features such as fearless concurrency, aggressive compiler optimizations, and safe iteration. Rust provides alias and mutability control through its borrow checker.

Scala is a language that supports both object-oriented and functional programming. Alongside that, Scala also provides a safe and strict static type system with strong theoretical foundations. In 2016, Scala also entered systems-level programming by supporting a native [21] output backend that emits LLVM IR.

Scala historically relies on the JVM and its garbage collection for memory management. Scala Native instead relies on configurable third-party garbage collectors designed for LLVM languages, namely Boehm-Demers-Weiser [10] and Immix [4].

The motivating example in the next section demonstrates that these garbage collectors, combined with functional programming paradigms such as immutability, can perform

poorly without high-level language support. Furthermore, even a well-configured garbage collector can still have performance problems that static memory management would resolve. Also, Scala Native users currently do not have the alias and mutability control that Rust provides.

imem enables Rust-style ownership rules and borrow checking in Scala as only a library. To support this design, imem also depends on a minimal compiler plugin, Scinear, which adds linear types to Scala. imem illustrates that the Scala type system, together with the recent addition of capture checking, is expressive enough that linearity is the only remaining feature needed, and that all the rest of Rust-style ownership and borrow checking can be implemented as a library.

This thesis presents the following:

- **Scinear:** A minimal compiler plugin that adds linear types to Scala and enforces linearity rules. The plugin also mixes linearity with capturing types.
- **imem:** A library that implements Rust-style ownership and borrow checking. imem defines boxes and immutable and mutable references, and it provides borrowing interfaces to borrow boxes immutably and mutably. The library enforces static memory-safety rules and alias and mutability control in the portion of memory that all boxes point to.
- A case study that implements safe linked lists to demonstrate the capabilities of imem and to compare imem with Rust, vanilla Scala, and Scala with linearity rules.

1.2 Motivation

The following example is a minimal version of the web-crawlers benchmark presented in the Web Crawlers Bench project [1] project, which identified performance problems in Scala Native caused by garbage collection. The program below implements a simple, single-threaded web crawler in Scala:

```
def crawl(toCrawlLinks: Queue[String], shouldStop: Boolean): Unit =
  if shouldStop then toCrawlLinks.toList.foreach(println(_))
  else
    val (currentLink, q) = toCrawlLinks.dequeue
    val newLinks = process(httpGet(currentLink))
    println(currentLink)
    crawl(q.enqueueAll(newLinks), checkShouldStop())
```

This function is tail-recursive and takes two arguments. The first argument is the queue of links to crawl, and the second argument indicates whether execution should stop. The `crawl` function first checks the stop condition. If execution should stop, the function prints

[1] https://github.com/amsen20/web-crawlers-bench/tree/main

all remaining links in the queue. Otherwise, it removes a link from the queue, extracts the links from the web content of that link, prints the current link, and then recursively calls `crawl`.

The `toCrawlLinks` queue and its underlying linked list grow as the program progresses. The Scala Native garbage collector scans the entire linked list at each GC pause. Because the linked list continues to grow, the duration of each pause also increases over time. This behavior leads to longer pauses and a reduction in Scala Native performance.

One way to prevent these pauses is to use Scala Zones [8] and allocate the queue inside a zone. The zone is defined before calling `crawl`. When the initial value of `toCrawlLinks` is allocated, it is explicitly specified that the queue must be allocated in that zone. As a result, all elements of the queue, meaning the nodes of the underlying linked list, are allocated in the same zone as the initial queue value.

In the case that a zone is used to allocate the queue, the queue and its elements are automatically freed when the zone goes out of scope. However, this approach leads to a memory leak. The popped links, which the program prints, are no longer needed, but they remain allocated in the zone, and they are not freed individually and remain in memory until the entire zone, and therefore the whole queue, is released.

The following demonstrates a simplified version of the same program that uses imem:

```
1  def crawl[OQ^, ...](toCrawlLinks: Box[Queue[String, OQ^], OQ], shouldStop
       : Boolean)(...): Unit =
2    if shouldStop then
3      foreach(toCrawlLinks, println(_))
4    else
5      val toCrawlLinks2 =
6        val lf = Lifetime[...]()
7        val (queueMutRef, queueBoxHolder) = borrowMutBox[...](toCrawlLinks)
8        val (queueMutRef2, newLinks) =
9          val lf2 = Lifetime[...]()
10         val (queueRefForDequeue, queueMutRefHolder) = borrowMut[...](
       queueMutRef)
11
12         val currentLinkBox: Box[String, {lf2, ...}] = popQueue(
       queueRefForDequeue)
13         val (currentLink, _) = borrowImmutBox[...](currentLinkBox)
14         println(currentLink)
15         val newLinks = process(httpGet(currentLink))
16
17         (unlockHolder(lf2.getKey(), queueMutRefHolder), newLinks)
18
19       enqueueAll(queueMutRef2, newLinks)
20       unlockHolder(lf.getKey(), queueBoxHolder)
21
22     crawl(toCrawlLinks2, checkShouldStop())
```

The implementation details of what imem interfaces are and how to use them are

explained thoroughly in the imem (chapter 4) and evaluation (chapter 5) chapters. The following explains the example briefly.

The program accesses the queue through a `Box`. Boxes, similar to Rust's boxes, are references to which imem controls program access statically. imem enforces static ownership and borrowing rules for accessing a box. The `Box` and the `Queue` data structure both take `OQ^` as a type parameter. `OQ^` is similar to Rust's lifetime annotations. It indicates that both the box and the queue to which it points remain valid as long as all lifetime capabilities in the set that instantiates `OQ^` are available.

The function first checks whether it should stop. If execution should stop, the function uses `foreach` to print all links that remain in the queue.

If execution should not stop, the function defines a new lifetime `lf`. This is the lifetime of the mutable reference that the program borrows to access the queue through `toCrawlLinks`. imem does not allow direct access to a box referent, which is called a resource. Instead, the program has to borrow the box, either immutably or mutably, and then access the box resource through the borrowed references.

After borrowing `toCrawlLinks`, the program loses access to `toCrawlLinks` because `Box` is a linear type and its instances follow the linearity rule (described in section 2.2).
The `borrowMutBox` function returns a mutable reference to the queue, `queueMutRef`, and a value holder, `queueBoxHolder`. The `queueBoxHolder` holds the queue until the program unlocks it using the key of lifetime `lf`. When the program unlocks the holder using `unlockHolder(lf.getKey(), queueBoxHolder)`, the lifetime `lf` expires. Then, the program can no longer mention the mutable reference and all other objects that include `lf` in their lifetime set, and they can be freed. This behavior enables imem to give temporary access to the box resource through mutable or immutable references, and then return the box while also invalidating the borrowed references.

After borrowing `toCrawlLinks`, the program re-borrows `queueMutRef` as `queueRefForDequeue` because `queueMutRef` is linear and the function needs access twice. One use is for popping from the queue, and the other use is for pushing to the queue. Similarly, the `borrowMut` function returns a value holder, `queueMutRefHolder`, which holds the mutable reference `queueMutRef` as long as `lf2` is available.

The function pops `currentLinkBox` from the queue and then borrows it immutably to print the link and process the web content of that link. After that, the function unlocks `queueMutRefHolder` to get `queueMutRef2` back. The program enqueues all new links using `enqueueAll(queueMutRef2, newLinks)`, and then unlocks `queueBoxHolder` to get the queue box back. Finally, the function uses that box to recursively call `crawl`.

Regarding memory management:

- The queue lifetime is `OQ^`, which is determined by the function caller and known to the program statically. Therefore, there is no need for a garbage collector to watch this value and determine when it is possible to free the object.

- The `currentLinkBox`, which is the popped link, has a lifetime set that includes `lf2`. Therefore, when `lf2` expires at `(unlockHolder(lf2.getKey(), queueMutRefHolder), newLinks)`, this box and its resource are ready to be freed, and the program can no longer access the box or its resource. This behavior resolves the memory leakage problem.

A final note is that the current implementation of imem does not manage memory allocations. Instead, it only applies ownership and borrowing rules to enforce safe memory management, alias control, and mutability control statically. The future works (section 7.2) sketches how runtime memory management can be supported in the future.

Chapter 2

Background

2.1 Introduction

This chapter summarizes the concepts that are the basics for understanding the proposed static memory management library `imem`.

2.2 Linear types

The Linear Types Can Change the World paper [24] proposes linear types to address a problem in the way functional programming languages model updates to the *world*. Here, the *world* refers to the large program state around which functionality is built, for example, the file system in a program that performs many file-system operations. In functional programming languages, the program can duplicate or discard this state at any point. However, both actions may lead to performance and correctness problems.

To address this problem, program memory is divided into two parts:

- **Linear memory:** All accesses to this part of memory follow the linearity rule. That is, values of linear types have exactly one reference.
- **Nonlinear memory:** This part of memory has no restrictions, except that a value of nonlinear type cannot reference a value of linear type.

As a result, the linear memory forms a collection of trees. Each linear value has exactly one reference. For the roots of these trees, the references are variables in the program environment. The linear memory may reference the nonlinear memory, but not vice versa.

The paper extends a typed version of the lambda calculus by adding linear types to the type system. Moreover, it statically ensures that the linearity rule is always satisfied. To

achieve this property, each expression is associated with an assumption list that contains all free variables that have a linear type that the expression may reference. The type system then guarantees that each expression consumes all variables in its assumption list. In addition, the list is divided into disjoint parts, which ensures that each variable is used exactly once. Moreover, the expression cannot reference any linear variable other than those listed in its assumption list.

For example, consider the following Scala code:

```
def f(x: LinearInt, y: LinearInt): Unit =
  println(x)
  println(y)
```

The expression `println(x); println(y)` has the assumption list `x` and `y`. The expression `println(x)` has the assumption list `x`. Also, the expression `println(y)` has the assumption list `y`.

As a result, the following program is rejected by the linear type system:

```
def f(x: LinearInt): LinearInt =
  println(x)
  x
```

The expression `println(x); x` has the assumption list `x`. However, both the sub-expression `println(x)` and the sub-expression `x` require `x` in their assumption lists. Therefore, one of them would have to omit `x`, which violates the linearity rule. Consequently, the program is invalid.

2.3 Scala Abstract Syntax Tree

The Scala compiler, similar to other compilers, processes a program through several phases[1]. The program starts as text and eventually becomes executable/interpretable code for the target platform. Each phase receives the program, modifies, annotates, or validates it, and then produces an updated version. The input and output of each phase are intermediate representations (IRs) of the program.

In Scala, these IRs are represented as Abstract Syntax Trees (ASTs) until the final backend phases. Scala plugins[2] can access and process the AST at any phase. The AST becomes typed after the typer phase and retains Scala type information until the erasure phase.

Scinear is a compiler plugin that processes the AST after the `cc` phase, which performs capture checking and annotates types with capture information, and before the `erasure` phase.

[1] https://www.scala-lang.org/api/3.4.2/docs/docs/contributing/architecture/phases.html
[2] https://docs.scala-lang.org/scala3/reference/changed-features/compiler-plugins.html

2.3.1 AST Nodes

The following is the list of AST nodes that Scinear supports:

- `tpd.ValDef`: A value definition, such as `val name = rhs`.
- `tpd.DefDef`: A method or function definition, such as `def name(param1, ..., paramN) = rhs`.
- `tpd.Block`: A block expression `{stat1; ...; statN; expr}` .
- `tpd.If`: A conditional expression `if cond then thenp else elsep`.
- `tpd.Match`: A pattern-match expression `selector match  case1 ...  caseN` .
- `tpd.CaseDef`: A single case clause `case pat if guard => body` inside a match expression.
- `tpd.Try`: A try-catch-finally expression `try block catch  case1 ...  caseN  finally finalizer`.
- `tpd.Assign`: A variable assignment `x = rhs`.
- `tpd.WhileDo`: A while loop `while cond do body`.
- `tpd.Select`: A field or method selection `qualifier.member`.
- `tpd.Typed`: A type ascription `expr:  T`, where the expression is explicitly annotated with a type.
- `tpd.SeqLiteral`: A sequence literal `[elem1, ..., elemN]`.
- `tpd.NamedArg`: A named argument in a function call, such as `name(x = arg)`.
- `tpd.Annotated`: An annotated expression `arg:  @annotation`.
- `tpd.Return`: A return statement `return expr`.
- `tpd.Labeled`: A labeled expression used to represent named blocks that can be exited early.
- `tpd.Inlined(expansion)`: An inlined expression that results from macro expansion, where `expansion` is the inlined body.
- `tpd.TypeDef`: A type or class definition, such as `class name  stat1; ...; statN` .
- `Util.Call`: A function or method call `ref(arg1, ..., argN)`, where `ref` is the function reference and the arguments are flattened.

- `Util.NewExpr`: An object instantiation `new TypeName(arg1, ..., argN)`.
- `Util.PolyFun`: A polymorphic function literal, such as `[TypeNameT] => rhs`.
- `tpd.ClosureDef`: A closure or anonymous function definition, such as `name => rhs`.

2.4 Path Dependent Types

Scala provides dependent types through path-dependent types, which are formalized in DOT [1] and in Foundations of Path-Dependent Types [2]. The following briefly explains the features of path-dependent types that are relevant to imem.

In Scala, classes can declare type members in addition to methods and immutable or mutable fields. For example, in the following code, class `A` defines a field `x`, a method `f`, and a type member `C`:

```
class A:
  val x = 1
  def f(): Unit = ()
  class C()
end A
```

In Scala path-dependent types, a path typically begins with an immutable variable. It is followed by zero or more selections of immutable fields and ends with a selection of a type member. Each such path denotes a unique type. For example, consider the following code:

```
val a = A()
val b = A()

val _: a.C = a.C() // ok
val _: b.C = b.C() // ok
val _: a.C = b.C() // compile error:
//           ^^^^^
//           Found:    b.C
//           Required: a.C
```

In the example above, the program can instantiate `a.C` and `b.C`. However, when it attempts to assign an instance of `b.C` to a variable of type `a.C`, a compile-time error occurs because these two types are distinct.

In imem, path-dependent types are used by defining a class that contains an `opaque` type member. This usage is illustrated in the following example:

```
class Lifetime:
  opaque type Key = Object

  def getKey(): Key = Object()
end Lifetime
```

In this way, the program can access an instance of the `Key` type associated with a given path only by using the same path to invoke the `getKey` method. This behavior is illustrated below:

```
val lf = Lifetime()
val _: lf.Key = lf.getKey() // ok
val _: lf.Key = lf.Key() // error
val _: lf.Key = Object() // error
val lf2 = Lifetime()
val _: lf.Key = lf2.getKey() // error
```

2.5 Capturing Types

Capture checking is a research project [5] and is currently an experimental feature [20] in the Scala type system that allows values to track capabilities. This section explains a simplified version of capture checking.

2.5.1 Overview

In capture checking, values track capabilities through their types. In Scala, with this feature enabled, every type `T` is augmented with a *capture set* `c1, c2, ..., cn`, where each member of the set is a capability. Capture checking refers to types with a non-empty capture set as capturing types, written as `T^c1, c2, ..., cn`.

In capture checking, a capability is either the universal capability `cap` or a local variable or method or class parameter that has a type with a non-empty capture set. Therefore, all capabilities are derived from other capabilities, except for `cap`, which is the root capability. The notation `T^` denotes a type that captures only the universal capability, that is, `T^cap`.

In the following example, `fs` is a capability because its type captures `cap`:

```
class Logger(fs: FileSystem^): ...
```

2.5.2 Function Types

Capture checking augments function types with capture sets, similar to other types. A function type `A -> B` denotes a function with an empty capture set, whereas a function type `A => B` denotes a function that captures the universal capability, `cap`. A function can also have a specific capture set. For example, `A -> {c, d} B` denotes a function that captures `{c, d}`.

A function captures the capabilities that it refers to in its body, or the capabilities captured by functions that it calls in its body. Functions with empty capture sets are

called pure functions, whereas functions that capture the universal capability are called impure functions.

For example, the `log` method below captures `fs` in its capture set:

```
val fs: FileSystem^ = ...

def log(s: String): Unit = fs.println(s)
```

2.5.3 Sub-typing and Sub-capturing

A type `T1^C1` is a subtype of `T2^C2`, written as `T1^C1 <: T2^C2`, if `T1 <: T2` and `C1 <: C2`. A capture set `C1 = c11, c12, ..., c1n` sub-captures another capture set `C2 = c21, c22, ..., c2m` if, for every capability `c` in `C1`, one of the following conditions holds:

- `c in C2`, or
- the type of `c` has a capture set `C'` such that `C' <: C2`.

As an expected result, every capture set sub-captures any capture set that contains the universal capability, `cap`.

2.5.4 Capture Set Type Parameters

Type parameters can only be instantiated with types that have empty capture sets. To accept arguments whose types have non-empty capture sets, the argument type must be annotated with `cap` as its capture set. For example, consider the following function:

```
def f[T](x: T^): ...
```

`f` takes an argument that is an instance of `T` and can capture any capability, because every type `T^C` is a subtype of `T^`, that is, `T^cap`. Therefore, the argument of `f` is implicitly polymorphic in its capture set. To make this polymorphism explicit, a new feature[3] allows programs to declare type parameters that represent capture sets. For example, the following is a rewrite of `f` with explicit capture set type parameters:

```
def f[T, CS^](x: T^{CS}): ...
```

In the definition of `f` above, `CS^` is a capture set type parameter that can be instantiated with any capture set.

This explicit form gives the program more control over capture set polymorphism. Because capture set parameters are explicit, it is possible to enforce specific relationships between them. In addition, the program can specify lower and upper bounds for them. The following examples demonstrate how to enforce that `CS^` always includes the `fs` capability:

[3] https://contributors.scala-lang.org/t/experimental-capture-checking-new-syntax-for-explicit-capture-polymorphism/7095

```
def f[T, CS^ >: {fs}](x : T^{CS}): ...
```

Furthermore, it is possible to use capture sets in a class type argument without capturing them. This design allows the program to enforce sub-capturing rules explicitly where needed, instead of having these rules applied implicitly in the background. For example, the following illustrates a stricter restriction on the capture set when it is expressed as a capture-set type parameter rather than as an annotated capture set.

```
class AAnn: ...
class AParam[CS^]: ...

val aAnn: AAnn^{fs} = ...
val bAnn: AAnn^ = ann // ok

val aParam: AParam[{fs}] = ...
val bParam: AParam[{cap}] = aParam // error
```

In the example above, the `AAnn` instance `aAnn` can be assigned to `bAnn` without any issue, because `fs <: cap`. However, the `AParam[fs]` instance in `aParam` cannot be assigned to `bParam` because the type parameter is not covariant.

To inform the typer that a function body uses a capability referenced by a capture set type parameter, the function must annotate that type parameter with `@caps.use`. This annotation is similar to including the capture set type parameter in the function's own capture set.

```
def f[T, @caps.use CS^](x: T^{CS}): Unit = ...
```

In the above example, the function type is: `f: [T, CS^] =>> [x: T^{CS}] ->{CS} Unit`.

2.5.5 Escape Checking

Capture checking does not allow certain capture sets to include `cap`. One such case is a type parameter. A type parameter `T` cannot be instantiated with a type that explicitly captures `cap`, for example `S^`. However, `T` can be instantiated with `S`, in which case `T^` becomes `S^`. For example, consider the following:

```
def withFs[T](body: Filesystem^ => T): T = ...

withFs(fs => () => fs.write()) // error
```

At the use site of `withFs`, `T` is instantiated with `() ->fs Unit`. Since `fs` is not defined in the scope of `T`, the typer replaces it with the smallest superset of `fs`, which is `cap`. The process of replacing a capture set with the smallest non-local superset of it is called avoidance. As a result, `T` is instantiated with `() ->cap Unit`, which captures `cap` and is therefore not allowed.

This mechanism enables a functionality called escape checking. A function that receives a continuation-passing-style argument can ensure that the argument will not leak the value passed to it, meaning the value is not stored in another object, captured by a closure, or returned after applying the continuation to the value. This guarantee is achieved by annotating the closure argument with the universal capability. In the example above, the first argument of `body` is escape-checked.

2.5.5.1 Escape Checking Propagation

To ensure that all values derived from an escape-checked argument remain escape-checked, the program must follow specific patterns. Two patterns are particularly useful in the context of imem.

First, if a function receives a value that may be escape-checked and intends to propagate escape checking to its return value, the function must take a universally capturing argument, and the return type must capture that argument.

```
def f(x: Object^): String^{x} = ...
```

For example, if an `Object` instance is escape-checked and passed to `f`, the resulting `String` instance is bound to the same scope as the argument `x`. In other words, the resulting `String` is also escape-checked.

Second, if a class instance may be escape-checked and needs to propagate escape checking to its fields, the class fields must capture `this`. For example, in the following definition, if an instance of `MyPair` is bound to a scope using escape checking and cannot leak from that scope, then its `first` and `second` fields are also bound to the same scope.

```
class MyPair[T, S](_first: T, _second: S):
  val first: T^{this} = _first
  val second: S^{this} = _second
```

The fields `first` and `second` are not defined directly as constructor arguments because `this` is not available in the scope of the constructor parameters. Therefore, their types cannot capture `this` at that point.

2.5.6 Access Control

A use case of explicit capture set type parameter bounds, called Access Control, ensures that a continuation-passing-style argument is instantiated with a closure that does not use a specific capability. For example, consider the following:

```
// This is a 'brand' capability that marks access to the filesystem or
   network
object FileSystem extends caps.Capability
object Network extends caps.Capability
```

```
4
5 val printerToFs: Printer^{FileSystem} = ...
6 val printerToNet: Printer^{Network} = ...
7
8 // The ‘body‘ may refer to (and use) the filesystem, but not the network
9 def run[C^ <: {FileSystem}](body: () ->{C} Unit): Unit = ...
10
11 run(() => printerToFs.write("hey")) // ok
12 run(() => printerToNet.write("hey")) // error
```

In the example above, any continuation that instantiates `body` cannot use the `Network` capability and may only refer to the `FileSystem` capability.

2.6 Rust

Rust [12] [18] is a programming language that provides static memory management and static mutability control through ownership rules[4] and the borrow checker[5]. This section provides a brief introduction to Rust's static ownership rules and borrow checking.

2.6.1 Ownership

Rust enforces the following static ownership rules:

- Each value in Rust has a variable that's called its owner.
- There can only be one owner for each value at a time.
- When the owner goes out of scope, the value will be dropped.

In the following example, `s1` is the owner of its value, which is a string `"hello"` in the heap:

```
1 {
2   let s1 = String::from("hello");
3 } // s1's value is (freed) dropped here
```

During assignment, except for trivial types such as integers and other copyable types, Rust transfers ownership of the right-hand side value to the left-hand side of the assignment. This ownership transfer is called a move. As a result, values in Rust behave similarly to linear values in linear type systems (discussed in section 2.2); when they are passed to functions or assigned to other variables, access to the original variable is lost.

[4] https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html
[5] https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html

```rust
let s1 = String::from("hello");
let s2 = s1; // s1 is moved to s2

println!("{}, world!", s1); // error
```

In the examples above, the ownership of the value that `s1` points to is transferred from `s1` to `s2`.

2.6.1.1 Box Pointer

Rust provides `Box<T>` pointers that allow a program to create pointers to values whose size is not known statically. These values reside on the heap. Such pointers are useful for defining recursive types, as in the following example:

```rust
enum List {
    Cons(i32, List), // error
    Nil,
}

enum List {
    Cons(i32, Box<List>), // ok
    Nil,
}
```

The ownership of `Box<T>` pointers is similar to that of other values. They have a single owner, which is a variable, at each point during program execution, and when the owner goes out of scope, the pointer and the value it points to are dropped (freed).

2.6.2 Borrowing

To allow a program to access a value without taking ownership of it, that is, without moving it, Rust provides borrowing. A program can borrow a value and create references to it. Borrowing can be mutable or immutable, which results in either a mutable reference or an immutable reference. The difference between holding a reference to a value and owning it directly is that a reference does not own the value. When the reference goes out of scope, only the reference is dropped, and the value it points to is not dropped. The following program demonstrates a Rust example that borrows a string value, first mutably and then immutably:

```rust
fn main() {
    let mut s = String::from("hello");

    let r1 = &mut s; // borrow ‘s‘ mutably
    r1.push_str(" world"); // update the string

    let r2 = &s; // borrow ‘s‘ immutably
    println!("Immutable borrow: {}", r2); // print the string
}
```

A lifetime represents a statically known consecutive part of a program, which is also referred to as a non-lexical scope. Every reference has a lifetime, which is the part of the program during which the reference is valid. The following code shows the previous example with annotated lifetimes:

```
fn main() {
    let mut s = String::from("hello");     // ------------------+-- 'a
                                           //                   |
    let r1 = &mut s;                       // -+-- 'b           |
    r1.push_str(" world");                 // -+                |
                                           //                   |
    let r2 = &s;                           // -+-- 'c           |
    println!("Immutable borrow: {}", r2); // -+                 |
}                                          // ------------------+
```

In the example above, the lifetimes of both references, `r1` and `r2`, which are `'b` and `'c`, lie within the non-lexical scope in which `s` owns the string, namely `'a`. In addition, the lifetime `'b` of the mutable reference `r1` ends before the lifetime `'c` of the immutable reference `r2` begins.

The borrow checker ensures the following properties by reasoning about reference lifetimes statically:

- At any given time, the program can have either one mutable reference or any number of immutable references to a value, which limits aliasing.
- The value that a reference points to should not be dropped, meaning that there are no dangling pointers.

The first goal is to ensure static mutability control by preventing the existence of a mutable reference together with another mutable reference or with immutable references at the same time. In the example above, at the point where the lifetime of the immutable reference `r2` begins, the lifetime of the mutable reference `r1` ends.

The second goal of the borrow checker is to ensure that no dangling pointers exist at any point during execution. To achieve this goal, Rust enforces a stricter rule regarding reference lifetimes: a reference lifetime must lie within the non-lexical scope in which the owner of the value at the time of borrowing remains the owner of that value. Therefore, if a value is borrowed and a reference is created, and the value is then moved, rather than dropped, the reference becomes invalid and its lifetime ends, as the example below shows:

```
fn main() {
    let s1 = String::from("Hello"); // s1 is the Owner
    let r1 = &s1; // r1 borrows s1

    let s2 = s1; // both s1 and r1 become invalid

    println!("{}", r1); // error
}
```

2.7 Stacked Borrows Model

Stacked Borrows [11] is an aliasing model for Rust programs. It is a dynamic model that ensures that memory accesses follow the borrowing semantics that are statically enforced by the borrow checker. The Stacked Borrows model tracks memory accesses without relying on reasoning about lifetimes.

The main motivation for this model is to validate Rust `unsafe` programs at runtime. A Rust `unsafe` program can violate borrowing and ownership rules without getting into compiler errors. In most cases, `unsafe` Rust is used to implement low-level data structures, such as doubly linked lists and atomic data structures, which are either impossible to implement under Rust's ownership and borrowing rules or significantly less efficient when those rules are strictly followed.

Miri [13], a Rust interpreter that detects Undefined Behavior at runtime, uses the Stacked Borrows Model for alias validation. This section focuses only on a subset of this model.

2.7.1 Overview

Based on the reference overview, Rust's borrow checker enforces a couple of rules for mutable and immutable aliases, which achieve the borrow checker's first goal (described in section 2.6.2) that at each point of execution, the program has access to either one mutable reference or multiple immutable references for a value at the same time. The Stacked Borrows model restates those rules to achieve the same goal without depending on lifetimes.

First, in the Stacked Borrows model, when a mutable reference is borrowed from a referent, which can be a value owner or another mutable reference: *every use of that reference (and anything derived from it) must occur before the next use of the referent after the reference was created.* Here, *use* includes moving, dropping, borrowing, re-borrowing, dereferencing, updating (mutating), reading, and similar operations.

In the following example, the program uses `y` after `x`, where `y` is a reference borrowed from `x` as its referent. This usage results in an error and violates the Stacked Borrows model:

```
let mut local = 0;
let x = &mut local;
let y = &mut *x; // re-borrow ‘x‘ to create ‘y‘
*x = 1; // use ‘x‘ again
*y = 2; // error: ‘y‘ used after ‘x‘ got used!
```

Second, based on the Stacked Borrows Model, when an immutable reference is borrowed from a referent, which can be a value owner, a mutable reference, or another immutable reference: *every use of the immutable reference (and everything derived from it) must occur*

before the next mutating use of the referent (after the reference got created), and moreover the reference must not be used for mutation.

```
let mut local = 42;
let x = &mut local;
let ir1 = &*x; // immutably re-borrow `x` to create `ir1`
let ir2 = &*x; // immutably re-borrow `x` to create `ir1`
let val = *x; // reading `x`
let val = *ir1; // reading `ir1`, ok
let val = *ir2; // reading `ir2`, ok
*x += 17; // mutate `local` through `x`
let val = *ir1; // error: `ir1` is used after its reference, `x`, is used
    mutably.
```

In the example above, reading from `x` and then from `ir1` and `ir2` is valid. However, after writing to the value of `local` through `x`, the program can no longer read from `ir1` or `ir2`.

2.7.2 Formalization

The subset of the Stacked Borrows model that is in imem's interest models memory as a mapping from locations to pairs consisting of a scalar and a stack.

$$\text{Mem} : \text{Loc} \to \text{Scalar} \times \text{Stack} \tag{2.1}$$

Each scalar is either a value, an integer, or a pointer. A pointer $\text{Pointer}(l, t)$ is defined as a pair composed of a location l and a unique pointer identifier t.

$$\text{Scalar} : \text{Pointer}(l, t) \mid z \quad \text{where } z, t \in \mathbb{Z} \tag{2.2}$$

Each location is associated with a stack, which is an ordered collection of tags. Each tag is either a unique tag, $\text{Unique}(t)$, or a shared tag, $\text{Shared}(t)$. Both kinds of tags contain a unique pointer identifier t.

$$\text{Tag} : \text{Unique}(t) \mid \text{Shared}(t) \tag{2.3}$$

$$\text{Stack} : \text{List}(\text{Tag}) \tag{2.4}$$

When the program allocates a new location in memory, the stack associated with that location initially contains a single unique tag with pointer ID zero, $\text{Unique}(0)$, which is the tag of the owner of the location.

2.7.3 Rules

To give an overview, each memory location is annotated with a stack. This stack contains tags, and each tag represents an active reference that the program has to that location. A Shared tag represents an immutable reference, and a Unique tag represents a mutable reference. Each reference, and the tag that represents it, has a unique ID, so there is a one-to-one mapping between references and their associated tags. If a reference ID is not present in the stack associated with the location that the reference points to, that reference is no longer active. Then, if the program tries to access that location through the reference, the model rejects the access.

The following rules are enforced by the Stacked Borrows model to validate memory accesses and the creation of new mutable or immutable references. Only the rules that are relevant to imem are presented in this section.

USE-1: When the program uses a pointer value Pointer(l, t) mutably, the stack for l must contain Unique(t). If Unique(t) is not present in the stack for l, the mutable access to this location is not valid. If there are tags above Unique(t), the program must pop them until Unique(t) becomes the top of the stack.

NEW-MUTABLE-REF: When the program creates a new mutable reference using `&mut expr` from an existing pointer value Pointer(l, t), the mutable access to location l must be valid according to rule **USE-1**. Let t' be a fresh pointer ID. The new reference value is Pointer(l, t'), and the program must push Unique(t') on top of the stack for l.

The following examples demonstrate how the stack associated with a location changes according to the *NEW-MUTABLE-REF* rule, and how the model validates memory accesses according to the *USE-1* rule:

```
let mut local = 0; // allocates memory l with owner `local` = Pointer(l, 0)
// Stack: [ Unique(0) ]

let x = &mut local; // borrows `local` mutably using NEW-MUTABLE-REF, resulting in `x` = Pointer(l, 1)
// Stack: [ Unique(0), Unique(1) ]

let y = &mut *x; // re-borrows `x` mutably using NEW-MUTABLE-REF, resulting in `y` = Pointer(l, 2)
// Stack: [ Unique(0), Unique(1), Unique(2) ]

*x = 1; // uses `x` mutably via USE-1 rule, which pops everything above `Unique(1)`
// Stack: [ Unique(0), Unique(1) ]

*y = 2; // attempts to use `y` via USE-1, causing error because `Unique(2)` is not in the stack
```

READ-1: When the program reads a pointer with value Pointer(l, t), the stack for l must contain a tag with pointer ID t. If no such tag exists in the stack for l, the read

access to this location is not valid. If there are tags above the tag with ID t, the program must pop tags until all tags above the tag with ID t are Shared(_).

NEW-SHARED-REF-1: When the program creates a new immutable reference using `&expr` from an existing pointer value Pointer(l, t), the read access to location l must be valid according to rule **READ-1**. Let t' be a fresh pointer ID. The new reference value is Pointer(l, t'), and the program must push Shared(t') on top of the stack for l.

The following examples demonstrate how the stack associated with a location changes according to the *NEW-MUTABLE-REF* and *NEW-SHARED-REF-1* rules, and how the model validates memory accesses according to *USE-1* and *READ-1* rules:

```
let mut local = 42; // allocates memory l with owner `local` = Pointer(l, 0)
// Stack: [ Unique(0) ]

let x = &mut local; // borrows `local` mutably using NEW-MUTABLE-REF, resulting in `x` = Pointer(l, 1)
// Stack: [ Unique(0), Unique(1) ]

let ir1 = &*x; // borrows `x` immutably using NEW-SHARED-REF-1, resulting in `ir1` = Pointer(l, 2)
// Stack: [ Unique(0), Unique(1), Shared(2) ]

let ir2 = &*x; // borrows `x` immutably again using NEW-SHARED-REF-1, READ-1 on `x` allows `Shared(2)` to stay, resulting in `ir2` = Pointer(l, 3)
// Stack: [ Unique(0), Unique(1), Shared(2), Shared(3) ]

let val = *x; // reads `x` via READ-1, `Shared` tags above `Unique(1)` are allowed to stay
// Stack: [ Unique(0), Unique(1), Shared(2), Shared(3) ]

let val = *ir1; // reads `ir1` via READ-1, `Shared(3)` above `Shared(2)` is allowed to stay
// Stack: [ Unique(0), Unique(1), Shared(2), Shared(3) ]

let val = *ir2; // reads `ir2` via READ-1, `Shared(3)` is at the top
// Stack: [ Unique(0), Unique(1), Shared(2), Shared(3) ]

*x += 17; // uses `x` mutably via USE-1 rule, which pops everything above `Unique(1)` (both shared tags)
// Stack: [ Unique(0), Unique(1) ]

let val = *ir1; // attempts to read `ir1` via READ-1, causing error because `Shared(2)` is not in the stack
// Stack: [ Unique(0), Unique(1) ] -> UNDEFINED BEHAVIOR
```

After the rules are applied, each stack starts, from bottom to top, with a unique tag that represents the owner of the location. This tag is followed by a sequence of other unique tags, which represent mutable references that are each borrowed from the previous mutable

reference. The stack can continue with a set of shared tags, which represent immutable references that are borrowed either from the top unique tag or from other immutable references.

2.7.4 imem

The Stacked Borrows model provides a dynamic verification of the static rules enforced by Rust's borrow checker. Because it is a dynamic analysis, it is more complete, meaning, some Rust programs that do not violate the Stacked Borrows model and do not exhibit undefined behavior are still rejected by the borrow checker.

imem follows the Stacked Borrows model in two independent directions:

- At compile time: imem enforces static rules that are sound. Therefore, accepted programs follow the Stacked Borrows model at runtime. However, similar to Rust, these static rules are more restrictive and may reject programs that are in fact valid.
- During Runtime: imem enforces a subset of the Stacked Borrows model within its memory model to demonstrate that statically accepted programs do not violate the dynamic rules. In addition, if a program does not follow the guidelines provided by imem or uses Scala features that break imem's static guarantees, the dynamic analysis verifies memory accesses at runtime.

Chapter 3

Scinear: A linear type plugin for Scala

3.1 Introduction

Scinear is a compiler plugin for the Scala 3 language. It introduces linear types to standard Scala code and strictly enforces their usage rules. This enforcement mechanism provides the minimal feature set necessary to implement the `imem` library. Consequently, the plugin prioritizes `imem` requirements rather than serving as a general-purpose linearity tool.

3.2 Defining Linear Classes

Defining a linear class is the same as defining a standard class in Scala. A class, and in general a type, is linear if it extends the `scinear.Linear` trait. This extension informs the scinear compiler plugin to enforce linearity rules on the class. The following demonstrates a basic declaration:

```
1 import scinear.Linear
2
3 class LinearInt(val value: Int) extends Linear
```

The original linear types paper [24] views linear types as immutable algebraic data types. Scinear adopts this view by enforcing specific rules on field and method definitions within a linear class.

3.2.1 Field Definition Rules

In a linear class, fields are either linear or nonlinear.

Linear field: A field of a linear class is linear if the type of the field is also linear.

In the following rules, Scinear handles linear fields and nonlinear fields differently.

3.2.1.1 Internal and External Fields

In Scala, a class field may reference other fields within the same class during initialization. However, this access violates the linearity rule during the construction of linear classes. For example, consider the following definitions:

```
class LinearDouble(val value: Double) extends Linear

class Circle(val r: LinearDouble) extends Linear:
  val circumference = LinearDouble(2 * math.Pi * r.value)
end Circle

@main def main(): Unit =
  val circle = Circle(LinearDouble(5.0))
  // circle.r <-- error accessing an internal member
  circle.circumference // ok external member access
end main
```

If Scinear permits accessing `r` outside the class body, such as in the `main` function, `circle.r` is read twice, once during the construction of `circle` and again in the `main` function.

Scinear-fields-rule-1: Every linear field of a linear class should be exclusively either an internal field or an external field. These categories are defined as follows:

- **Internal fields** that other fields refer to in the class body (i.e., during the execution of the constructor).
- **External fields** that are referred to from outside the class instance (i.e., through a selector).

This rule manages access scope. During initialization, it allows the initial value expressions of some fields to refer to an internal linear field. After construction, it permits access to an external linear field through a selector (e.g., `linearInstance.linearField`). The rule mutually excludes internal and external fields, meaning the program does not read a linear field of a linear instance more than once.

Scinear analyzes the class body, starting with the class parameters, then proceeding through the statements in program order. The usage section (section 3.3) explains the order's effect.

3.2.1.2 Fields immutability

The program reads each field of a linear instance only once, so defining mutable fields in a linear class is unnecessary.

Scinear-fields-rule-2: All fields of a linear class should be strictly evaluated immutable values (val), not mutable variables (var) or lazy values (lazy val). Each field's type can be either linear or nonlinear.

This rule enforces immutability on linear types and simplifies the evaluation order.

```
class LinearList(val next: LinearList) extends Linear:
  // var size: Int = 1 + next.size <-- error mutable field

  // lazy val size: Int = 1 + next.size <-- error lazy val field

  val size: Int = 1 + next.size // ok
end LinearList
```

3.2.2 Method Definition Rules

3.2.2.1 Methods and Linear Fields

When a class member method references a field in its body, every invocation of the method potentially reads the field. This access causes a problem if the field is linear. For example, in the following, every call to `getSize` would result in reading the `next` field:

```
class LinearList(val next: LinearList) extends Linear:
  def getSize: Int = 1 + next.getSize // error ‘next‘ is not accessible
   here
end LinearList
```

Scinear-method-rule1: The methods defined in a linear class cannot refer to the class's linear field.

This rule prevents a linear field of the linear class from leaking into its methods. Furthermore, this restriction ensures that linear classes behave similarly to an algebraic datatype.

3.2.2.2 Exempted Methods

Inherited and compiler-generated methods are inevitable. Also, the developer cannot modify the implementation of these methods. Moreover, linear classes are useful only if they can be decomposed into their fields. In Scala, this operation requires the class to implement the `unapply` method. However, following Scinear rules, implementing an `unapply` method is impossible. For example:

```
class LinearPair(val x: LinearInt, val y: LinearInt) extends Linear
object LinearPair:
  def unapply(p: LinearPair): (LinearInt, LinearInt) = (p.x, p.y)
end LinearPair
```

It is not possible to implement the `unapply` method without referring to `x` and `y` fields.

Scinear-method-rule2: In a linear class, inherited methods from Object, compiler-generated methods, and the `unapply` method defined in the linear class companion object are exempt from all linear rules.

These method exemptions maintain linearity within a standard Scala 3 codebase.

3.2.3 Complementary Rules

3.2.3.1 `this` in Linear Classes

Accessing `this` implies reading the value of the instance referred to by the keyword. This reference would violate the linearity rule if the instance is linear. For example:

```
class LinearList(val data: Int, val next: LinearList) extends Linear:
  def append(newData: Int): LinearList =
    LinearList(newData, this) // error ‘this‘ is not allowed to refer to
   linear values
end LinearList
```

Every call of the `append` method on a `LinearList` instance would result in reading the instance.

Scinear-self-reference-rule: A program must not use the keyword `this` in any scope where it refers to an instance of a linear class.

This rule ensures that the fields of a linear class cannot refer to the linear instance during the construction of the instance. Thus, the instance is always available for use after construction. Furthermore, body of a method declared in a linear class does not refer to the class instance during a method call, meaning that after reading a linear instance to get access to its method, the linear instance is not reconsumed by calling the method.

3.2.3.2 Type Hierarchy Separation

Scinear-inheritance-rule: A linear class can only extend linear classes, linear traits, and `Object`.

This rule separates the linear class hierarchy from other types, joining them at `Object`.

3.2.3.3 Class Definition Simplification

Scinear-Nesting-rule: Linear class definitions cannot include nested class definitions.

This rule reduces the complexity of defining linear classes without sacrificing much practicality.

3.3 Using Linear Types

Scinear follows the same rule as the main linearity paper [24]. That is, a program accesses a linear value through a single reference, and this reference is read only once during execution.

To enforce this rule statically in a Scala program, Scinear tracks identifiers bound to values of linear types, called linear variables.

Linear variable: A linear variable is an identifier that can be a method parameter, a class parameter, or a local variable that is bound to a value whose type is linear.

When an expression mentions a linear variable, the expression is using the variable. The following rules limit the usage of linear variables in a program.

Scinear-usage-at-most-rule: For each linear variable l and any node u that uses l, no node v preceding u in the AST traversal order (discussed in section 3.8.1) may use l.

To put it another way, after a linear variable is used, it expires. Keep in mind that the AST traversal order is a partial order.

Scinear-usage-at-least-rule: The program has to use every linear variable statically. To put it another way, for every linear variable, some expression in the program should mention it.

```
1 class LinearInt(val value: Int) extends Linear
2
3 @main def main(): Unit =
4   val x = LinearInt(1)
5   val y = LinearInt(2)
6   val z = // error haven't used ‘z‘ at all
7     LinearInt(x.value + y.value) // cannot use ‘x‘ or ‘y‘ again after
    this point
8 end main
```

3.3.1 Control flow

Usage rules influence the validity of mentioning linear variables within control-flow structures. The following sections explain how Scinear treats control flow structures.

3.3.1.1 If-Else Expressions

```
if <condition expression> then <then expression> else <else expression>
```

Scinear first traverses the `condition` expression. This means that the `then` expression, the `else` expression, and the rest of the program cannot refer to linear variables mentioned in the `condition`. Because there is no traversal order between the `then` and `else` expressions, both expressions can refer to the same linear variables. However, the plugin marks a linear variable as used for the remainder of the program if the variable appears in at least one branch.

```
class LinearInt(val value: Int) extends Linear

def select(a: LinearInt, b: LinearInt, c: LinearInt, d: LinearInt):
    LinearInt =
  if (a.value > b.value)
    // cannot use `a` and `b` here
    d
  else
    // cannot use `a` and `b` here
    c
  // cannot use `a`, `b`, `c`, and `d` here
end select
```

3.3.1.2 Case-Match Expressions

```
<match expression> match
  case <case pattern> => <case expression>
  ...
```

Scinear starts with the `match` expression. The cases and the rest of the program cannot refer to the linear variables mentioned in this expression. For each case, the traversal order is first the `case pattern` and then the `case expression`, meaning that the `case expression` should use linear variables defined in the `case pattern`. Because there is no traversal order among cases, the plugin marks a linear variable as used for the remainder of the program if the variable appears in at least one of the `case pattern` or `case expression`.

```
trait LinearList extends Linear
class Cons(val head: Int, val tail: LinearList) extends LinearList
class Nil() extends LinearList

def addToHead(lst: LinearList, offset: LinearInt): LinearList = lst match
  case Cons(head, tail) =>
    // cannot use `lst` here
    Cons(head + offset.value, tail)
  case nil: Nil =>
    // cannot use `lst` here
    nil
```

```
  // cannot use `lst` and `offset` here
end addToHead
```

3.3.1.3 Try Catch Finally Expressions

```
try
  <try expression>
catch
  case <case pattern> => <case expression>
  ...
finally
  <finally expression>
```

Scinear traverses the `try` expression first. The catch `case` expressions, the `finally` expression, and the rest of the program cannot use linear variables mentioned in the try expression. Similar to `match-case`, for each catch case, the traversal order is first the `case pattern` and then the `case expression`. Because there is no traversal order between catch cases, these cases can use the same linear variables. Scinear traverses the `finally` expression last. As a result, the `finally` expression can only use linear variables that remain unused in the `try` expression and in all catch `case patterns` and `case expressions`.

```
def tryF(f: LinearInt => LinearInt, x: LinearInt, default: LinearInt):
    LinearInt =
  try
    val y = LinearInt(x.value + 1) // local to this block
    f(y)
  catch
    case _ =>
      // cannot use `f` and `x` here
      default
  // cannot use `f`, `x` and `default` here
end tryF
```

3.3.1.4 Loops

The loop body and condition are evaluated multiple times during execution. Consequently, if a loop body or condition uses a linear variable, the program reads that variable multiple times during execution.

For example, the following demonstrates an incorrect implementation of a length function for a linear linked list.

```
def length(lst: LinearList): Int =
  var curr = lst
  var len = 0

  while !curr.isInstanceOf[Nil] do // error cannot use `curr` in loop's
    condition
```

```
    curr = curr match // error cannot use ‘curr‘ in loop’s body
      case Cons(_, tail) =>
        tail
      case nil: Nil =>
        nil
    len += 1
  end while

  len
end length
```

In every iteration of the loop, the program reads the `curr` variable. The correct way to implement the `length function` is using recursion, which is described in the next section.

To address this issue, Scinear enforces a specific rule for loops.

Scinear-usage-loop-rule: The loop condition and body cannot use linear variables defined in the loop's enclosing scopes.

Recursion offers an alternative approach to this restriction. A recursive function safely consumes a linear value multiple times at runtime.

3.3.2 Method and Function Definitions and Calls

Methods and functions present a similar problem to loops. If a method or function body uses a linear variable, the program accesses the linear value via that variable multiple times during execution.

For example:

```
def add(x: LinearInt, y: LinearInt): LinearInt =
  def addByXDef(z: LinearInt): LinearInt =
    LinearInt(x.value + z.value) // error: ‘x‘ cannot be accessed here

  val addByXFunc = (z: LinearInt) =>
    LinearInt(x.value + z.value) // error: ‘x‘ cannot be accessed here

  LinearInt(x.value + y.value) // ok
end add
```

Every call to `addByXDef` or `addByXFunc` would result in reading linear variable `x`.

To resolve this issue, Scinear enforces a specific rule for anonymous functions and all definitions, including local, class member, and top-level package member methods.

Scinear-usage-function-rule: The body of a method or a function cannot refer to a non-local linear variable. In other words, the body can only refer to the linear variables defined as parameters or defined in the body itself.

The main linearity paper [24] addresses this issue by treating the function itself as a linear value. However, Scinear avoids this complexity by enforcing the rule above, at the expense of expressiveness.

The recursion approach to list length would look like this:

```
def length(lst: LinearList): Int = lst match
  case Cons(head, tail) => 1 + length(tail)
  case nil: Nil => 0
end length
```

3.3.3 Fields in other types

Following the original linearity paper [24], Scinear does not allow classes that are not linear to have linear fields.

Scinear-usage-nonlinear-type-field-rule: Nonlinear classes should not have linear fields.

```
class Buffer(val list: LinearList) // error: nonlinear classes are not
    allowed to store linear fields
```

3.4 Polymorphism

Standard Scala 3 codebases use different aspects of polymorphism. Scinear integrates linear types into this polymorphism through two rules.

3.4.1 Polymorphic promotion

Instantiating a polymorphic type with a linear type argument results in a nonlinear value that stores a linear value, as below:

```
val ls: List[LinearInt] = List(LinearInt(42))
ls(0) // accessing to a linear value
ls(0) // accessing to a linear value
```

This outcome contradicts the main linearity paper [24]'s rule that nonlinear values cannot retain linear values. Also, Scinear enforces a specific rule to prevent nonlinear classes from storing linear fields. But this rule is limited to class definitions. Scinear addresses linear instantiations through linear promotion.

Linear promotion: If Scinear promotes a nonlinear type, the plugin enforces all linearity rules on instances of that type. Put another way, Scinear treats promoted types as linear types.

One approach is to promote all linear instantiations of polymorphic types. However, this strategy is unsound because the internal implementation of promoted types may violate Scinear rules.

Furthermore, banning all linear instantiations is impractical because linear types require the ability to decompose into fields. Scala implements this decomposition by returning an `Option[Tuple[...]]` of fields during `match-case` and `unapply` operations. This means that supporting linear type arguments for these two polymorphic types is inevitable. As an example, the `addToHead` example requires implementing an `unapply` method to actually work:

```
object Cons:
  def unapply(l: Cons): Option[(Int, LinearList)] =
    Some((l.head, l.tail)) // Scinear exempts ‘unapply‘ method body from
    any linear rule.
end Cons
```

Moreover, the simplicity of `Option[T]` and `Tuple[T]` ensures that promoting these types is safe.

To address all the discussed issues safely, Scinear enforces a minimal promotion rule.

Scinear-polymorphic-promotion-rule: If a method parameter, class parameter, local variable, or a `this` reference v possesses type $T[T_1, ..., T_n]$ where T is not a linear type and type parameter T_i is instantiated with a linear type, one of the following cases applies:

1. If T is `Option` or `Tuple`, Scinear promotes T to a linear type and treats v as a linear variable. 2. If T_i has the `@HideLinearity` annotation, Scinear emits a warning. 3. Otherwise, Scinear emits an error.

The following example presents an alternative implementation of `addToHead` for `Cons`.

```
def addToHead(cons: Cons, offset: LinearInt): Cons =
  val deconstructedOpt = Cons.unapply(cons) // promoted ‘Option‘
  val deconstructed = deconstructedOpt.get // promoted ‘Tuple‘
  val (head, tail) = deconstructed
  val updatedHead = head + offset.value
  Cons(updatedHead, tail)
end addToHead
```

3.4.2 Polymorphic function calls

Polymorphic functions prevent code duplication. Plus, they are common in the Scala standard library. Calling these functions with linear values instantiates type parameters with linear types. Scinear follows a specific rule to track this instantiation.

Scinear-polymorphic-function-call-rule: During a function call, if type parameter T is instantiated with a linear type, three cases can happen:

1. T has a linear upper bound: Scinear continues with no errors or warnings. 2. T is annotated with `@HideLinearity` annotation: Scinear emits a warning. 3. Otherwise, Scinear emits an error.

In the first case, Scinear validates the function body with the knowledge that T is linear. The second case is a way for developers to implement functions that do not follow linearity rules while still having linear parameters. The last case prevents unintentional leakage of linear values through polymorphic interfaces.

```
class LinearInt(val value: Int) extends Linear

def LinearIdentity[T <: Linear](x: T): T = // case 1
  // println(x) <-- error: would result in using ‘x‘ twice
  x
end LinearIdentity

def logIdentity[@HideLinearity T](x: T): T = // case 2
  println(x)
  x
end logIdentity

def normalIdentity[T](x: T): T = x // case 3

def checkIdentity(): Unit =
  LinearIdentity(LinearInt(42)) // ok
  logIdentity(LinearInt(42)) // ok
  normalIdentity(LinearInt(42)) // error: cannot pass linear value as
   polymorphic parameter
end checkIdentity
```

3.4.3 `@HideLinearity`

Scinear provides a method to bypass linearity rules in both polymorphic promotion and polymorphic function calls using the `@HideLinearity` annotation. With this annotation, the program can store linear instances in polymorphic nonlinear classes. In the program below, `nonLinearInt` is an instance of the `NonlinearWrapper` class, where the type parameter `T` is instantiated with `LinearInt`.

```
class LinearInt(val value: Int) extends scinear.Linear
class NonlinearWrapper[@scinear.HideLinearity T](val content: T)

val linearResource = LinearInt(17)
val nonLinearInt = NonlinearWrapper[LinearInt](linearResource)
```

Also, the program can pass linear instances to polymorphic functions as polymorphic parameters. This would enable the function to use the parameter without linear restrictions.

```
def log[@scinear.HideLinearity T](linearVal: T): T =
  println(linearVal)
  linearVal

val x = LinearInt(42)
val y = log(x)
```

In the example above, the `log` function hides the linearity of the type parameter `T` so that it can print the value and then return it, thereby preserving access to it.

Similar to Rust's `unsafe` keyword, `@HideLinearity` allows libraries like imem to provide safe external interfaces while breaking safety rules internally.

Strict adherence to linearity rules prevents holding multiple references to a single linear object, which is essential to `imem`'s use case. Therefore, a mechanism to disable linearity rules for specific program areas is inevitable.

This annotation is intended for limited library use cases. As a result, Scinear emits a warning when the program uses this annotation.

3.5 Capture checking

Scala includes a new experimental feature called capture checking. Based on section 2.5, a capture set consists of a set of capabilities. A capture set appears in a type in two locations:

- As a capturing type's capture set annotation: `T^/*capture set*/`
- As a type argument instantiating a type parameter with a capture set: `T[/*capture set*/]`.

Linear capability: A linear capability is a capability whose type is linear. In other words, the capability's type extends the `scinear.Linear` trait.

Scinear-capture-set-rule: Let AST node u be an expression of type T. Assume T has a type parameter $CS >: caps.CapSet$ and that the argument instantiating it includes a linear capability l. l should not be expired in node u, meaning no node v that precedes u in traversal order may refer to l.

This rule disallows mentioning a linear variable, which is also a capability, in a capture set after the program has used it.

```
class Tracker[LinearCaps^]:
  def check(): Unit = ()
end Tracker

@main
```

```
def main(): Unit =
  val x: LinearInt^ = LinearInt(10)
  val y: LinearInt^ = LinearInt(20)
  val tracker: Tracker[{x, y}] = Tracker()
  tracker.check()
  println(x.value)
  // tracker.check() <-- error: `tracker` mentions `x` which is expired
   in its capture set.
  println(y.value)
end main
```

In this example, `Tracker` is a nonlinear type that tracks linear capabilities within its type parameter `LinearCaps`. The program can refer to `tracker` only if all linear capabilities in the `LinearCaps` type argument, `x, y` in the example above, are not expired.

Keep in mind that this rule only tracks capture sets that instantiate a type parameter. However, capture sets annotating capturing types may still include an expired linear capability. The `imem`'s use case motivates this design choice.

In general, capture sets annotating capturing types are covariant. This covariance means that they can be widened to `cap` through avoidance at any point in the program. The widened capture set, `cap`, is more restrictive than the original set. However, it excludes the linear capability.

The following example uses an `Object`'s capture set to track `x, y`. Scinear enforces no limitations on `tracker` in this case. Also, this example demonstrates that the tracking list is easily eliminated through avoidance.

```
@main
def main(): Unit =
  val x: LinearInt^ = LinearInt(10)
  val y: LinearInt^ = LinearInt(20)
  val tracker: Object^{x, y} = Object()
  val avoidedTracker: Object^ = tracker // does not mention `x` or `y` in
    its type.
  tracker // ok
  println(x.value)
  tracker // ok
  println(y.value)
end main
```

3.6 Memory model

This section explains how Scinear rule enforcement shapes the graph of linear and nonlinear values at runtime. Because `@HideLinearity` significantly affects the memory model, this discussion is structured into two parts.

For simplicity, this section only concerns programs that do not cast linear types to nonlinear types.

3.6.1 With no `@HideLinearity`

3.6.1.1 Memory Overview

In the absence of `@HideLinearity`, references to a linear type are not stored within a nonlinear type, other than `Option` and `Tuple`, due to the nonlinear-type-field-rule. Additionally, linear values are accessible only by other linear values or by linear variables.

Reachable linear values form a tree rooted in the program environment. The program environment includes linear variables available in the scope, while the remaining nodes represent linear values stored within other linear values. Furthermore, linear values may refer to nonlinear values. Figure 3.1 illustrates this memory overview.

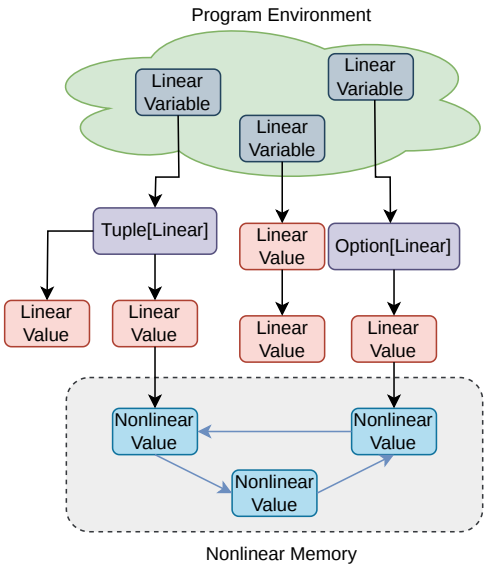

Figure 3.1: Linear Memory model overview

3.6.1.2 Why

Operations related to linear values fall into one of the following categories:

- Reading a field or calling a method: This interaction expires the linear variables and designates the result of the method or field access as a new variable.
- Creating a linear variable: This process results in a new linear variable. Any linear variable used as an argument to the type constructor becomes a child of this new variable as a linear value.

Based on the operations, the tree structure is consistently preserved.

3.6.2 With `@HideLinearity`

The `@HideLinaerity` annotation allows bypassing all Scinear rules. Specifically, wrapping the code that causes the error in a function annotated with `@HideLinaerity` that accepts a linear value as a polymorphic argument converts an error into a warning. This mechanism is similar to Rust's `unsafe` keyword. As a result, the use of this annotation makes any object graph configuration possible. Similar to Rust's unsafe, imem uses this feature to build basic memory management blocks. While these blocks are internally unsafe, they expose a safe interface.

3.7 Plugin implementation

The following section provides an overview of the Scinear implementation. For additional details, refer to the Scinear Plugin repository[1].

The Scinear implementation consists of two passes. The first pass follows the linear type system [24] typing rules and enforces linearity rules. The second pass applies rules that are specific to Scinear.

3.8 Linearity Rule Pass

3.8.1 Traversing AST

The following section presents most of the AST nodes that Scinear supports and specifies the order in which Scinear traverses their children. This order is a partial order, which means that, for some nodes, there is no specific order among their children. Applying this to the AST of a program produces a directed acyclic graph (DAG). This DAG represents the traversal order that Scinear uses to enforce the linearity rule.

Sequential Ordered Nodes:

- `Util.Call`: first `ref`, then arguments (`arg1`, ..., `argN`) in argument order after flattening.
- `Util.NewExpr`: arguments (`arg1`, ..., `argN`) in argument order after flattening.
- `tpd.Block`: first statements (`stat1`, ..., `statN`) in program order, then `expr`.

[1] https://github.com/amsen20/scinear

- `tpd.SeqLiteral`: elements (`elem1`, ..., `elemN`) in program order.
- `tpd.CaseDef`: first `pat`, then `guard`, then `body`.
- `tpd.TypeDef` **(linear)**: template statements (`stat1`, ..., `statN`) in program order.

Branching Nodes:

- `tpd.If`: first `cond`, then `thenp` and `elsep`, no order between `thenp` and `elsep`.
- `tpd.Match`: first `selector`, then cases, `case1`, ..., `caseN`, no order among cases.
- `tpd.TypeDef` **(non-linear)**: template statements, `stat1`, ..., `statN`, independently, no order among them.
- `tpd.Try`: first `block`, then cases, `case1`, ..., `caseN`, no order among cases, then `finalizer` after `block` and all `cases`.

The following AST nodes each have a single child; therefore, the traversal order is trivial:

- `tpd.Select`: `qualifier`
- `tpd.Typed`: `expr`
- `tpd.NamedArg`: `arg`
- `tpd.Assign`: `rhs`
- `tpd.Annotated`: `arg`
- `tpd.Return`: `expr`
- `tpd.Labeled`: `expr`
- `tpd.Inlined`: `expansion`
- `tpd.ValDef`: `rhs`

The following nodes isolate some of their children. Therefore, these children appear as independent sources in the ordering DAG:

- `tpd.WhileDo`: the `cond` and `body` children are isolated; therefore, both are sources of the DAG.
- `tpd.DefDef`, `tpd.ClosureDef`, `Util.PolyFun`: the `rhs` child is isolated and thus a source of the DAG.

3.8.2 The Pass Overview

The first pass traverses the AST using the `checkNode` function in traversal order. This traversal follows the typing rules described in the main linearity paper [24].

The recursive `checkNode` function maintains an assumption list when it reaches an AST node during traversal. This assumption list contains the linear variables that are accessible at that node. The `checkNode` function then recursively visits the children of the node in traversal order and ensures that no linear value is used more than once. In the case of children with no order between them, the intersection of their returned assumption lists is the resulting assumption list. If the AST node is an expression, the function removes the used linear variables from the assumption list and returns the remaining list. If the AST node is a block, a function definition, or a type definition, the `checkNode` function verifies that the remaining assumption list is empty. This requirement ensures that every linear value is used exactly once. Consider the following example:

```
def f(x: LinearInt, y: LinearInt): LinearInt =
  //                 ^ Linear argument y is not used
  if x.value > 0 then
    x // Linear value x is being used twice, or is not accessed directly
  else
    x // Linear value x is being used twice, or is not accessed directly
```

Scinear invokes `checkNode` on the body of the function definition with the assumption list containing `x` and `y`. The block contains a single `if` expression. Therefore, the `if` node is checked recursively with the same assumption list, `x` and `y`.

The traversal order of the `if` expression is as follows:

- First, the condition node `x.value > 0` is checked. After this step, the remaining assumption list is `y`, because `x` is consumed in the condition.
- Next, the `then` and `else` branches are checked, in no specific order. Both branches reference `x`. However, `x` is no longer in the assumption list, which results in an error.

Finally, when control returns to the block node, `y` is still present in the assumption list. Consequently, Scinear reports an error indicating that `y` is not used.

3.9 Scinear Specific Rules Pass

In this pass, Scinear checks the following properties:

- As discussed in section 3.2.1.1, the linear fields of a linear class are divided into two disjoint sets: internal fields and external fields. Internal fields are not used outside the class definition, and external fields are not used within the class definition.

- As discussed in section 3.4.2, polymorphic type parameters that are instantiated with linear types are either annotated with `@HideLinearity` or have a linear upper bound.

This pass is implemented as a conventional pattern-matching traversal.

Chapter 4

Memory management library `imem`

4.1 Introduction

imem is a static memory management library that ensures programs safely access and modify resources tracked by imem, while guaranteeing that reference accesses adhere to a minimal form of the Stacked Borrows Model. The formal basis of imem's static memory management is described in the section 4.2. In addition to static enforcement, imem also utilizes dynamic verification to inform the user when workarounds to the static rules lead to potential safety violations.

During runtime, imem monitors each resource accessed by different references to ensure that none violate the Stacked Borrows Model rules. This runtime verification is optional, so it can be disabled to prioritize performance.

During compilation, imem utilizes Scala type system features, including capture checking, dependent types, and linear types provided by the scinear plugin, to implement ownership and borrow checking. This approach is similar to Rust's ownership system and borrow checker. However, imem is a library rather than a compiler component.

The static enforcement in imem is not entirely sound and contains loopholes. The soundness section (section 4.6) discusses the differences between the implemented version of imem and the formally described version. It also explains the existing loopholes and suggests minimal guidelines to avoid them.

Additionally, imem does not manage the actual allocation and deallocation of memory. The future works (section 7.2) sketches how imem can be connected to actual memory management in Scala Native.

4.2 Memory Management in imem

This section defines the meaning of memory management in imem, the safety guarantees that imem statically ensures, and the reasons why imem is more expressive than pure linear data structures.

4.2.1 Pure Linear Memory

A pure linear program that follows all linearity rules described in the original linearity paper [24], or in scinear terms, that is, a program without `@HideLinearity`, structures memory as a tree-like data structure.

4.2.1.1 Linear Memory Definition

The following presents a formal definition of a linear program's memory at a given point during execution. The environment splits into linear and nonlinear environments.

$$\rho = \rho_L \cup \rho_{NL} \tag{4.1}$$

The linear environment is a mapping from linear variables to linear locations.

$$\rho_L : \mathrm{Var}_L \to \mathrm{Loc}_L \tag{4.2}$$

The nonlinear environment is a mapping from nonlinear variables to nonlinear locations or integer values.

$$\rho_{NL} : \mathrm{Var}_{NL} \to \{\mathrm{Loc}_{NL} \cup \mathbb{Z}\} \tag{4.3}$$

The memory is also divided into linear and nonlinear.

$$\sigma_L : \mathrm{Loc}_L \to \mathrm{Val}_L \tag{4.4}$$

$$\sigma_{NL} : \mathrm{Loc}_{NL} \to \mathrm{Val}_{NL} \tag{4.5}$$

Linear values consist of linear references, nonlinear references, and integer values.

$$\mathrm{Val}_L = \{\langle v_1, \ldots, v_k \rangle | v_i \in \mathrm{Loc}_L \cup \mathrm{Loc}_{NL} \cup \mathbb{Z}\} \tag{4.6}$$

Nonlinear values consist of nonlinear references and integer values.

$$\mathrm{Val}_{NL} = \{\langle v_1, \ldots, v_k \rangle | v_i \in \mathrm{Loc}_{NL} \cup \mathbb{Z}\} \tag{4.7}$$

4.2.1.2 Well-Formedness

The memory is well-formed if the linear part consists of disjoint trees, each rooted at a linear environment variable.

Formally, every linear value has exactly one other linear value or linear variable whose location serves as a reference, and linear variables can reach all linear values.

Reference Uniqueness: Every linear location $l \in \text{dom}(\sigma_L)$ has to be mentioned exactly by its parent, which can be a linear value or environment variables.

Parents of l are defined as:

$$P(l) = \{x \in \text{dom}(\rho_L) \mid \rho_L(x) = l\} \cup \{k \in \text{dom}(\sigma_L) \mid l \in \text{Refs}(\sigma_L(k))\} \tag{4.8}$$

Where $\text{Refs}(v)$ is the set of references linear value $v = \langle v_1, \ldots, v_k \rangle$ mentions:

$$\text{Refs}(v) = \{v_i | v_i \in \text{Loc}_L\} \tag{4.9}$$

In a well-formed memory $\text{WF}(\rho_L, \sigma_L)$ every location has one parent.

$$\text{WF}(\rho_L, \sigma_L) \implies \forall l \in \text{dom}(\sigma_L), |P(l)| = 1 \tag{4.10}$$

Par is the function that maps each linear location to its unique parent, meaning:

$$\text{Par}(l) \in P(l) \tag{4.11}$$

Reachability of Every Linear Value: The variables in the environment must reach all linear locations in memory.

In other words, for every linear location l, the following condition must hold:

$$\text{WF}(\rho_L, \sigma_L) \implies \forall l \in \text{dom}(\sigma_L), \exists n \geq 0 \text{ s.t. } Par^n(l) \in \text{range}(\rho_L) \tag{4.12}$$

4.2.1.3 An Example

The following is a two-item list expressed with linear values:

$$\rho_L = \{\texttt{list} \mapsto l_1\} \tag{4.13}$$

$$\sigma_L = \{l_1 \mapsto \langle 42, l_2 \rangle, \quad l_2 \mapsto \langle 99, 0 \rangle\} \tag{4.14}$$

In Scala, using Scinear, the following code constructs the memory described above:

```
trait LinearList extends scinear.Linear
class Cons(val head: Int, val tail: LinearList) extends LinearList
class Nil() extends LinearList

val list = Cons(42, Cons(99, Nil()))
```

To allow the program to access the second element (for example, for reading) without ever creating a non-well-formed memory state, it must deconstruct the first element of the linear list, store that value in a separate variable, and then retain a linear variable that refers to the second element. The process in Scala is as follows:

```
val (elem1, tail) = Cons.unapply(list).get
tail match
  case Cons(elem2, tail2) => ...
```

After accessing the second element, the memory would become the following:

$$\rho'_L = \{\texttt{tail} \mapsto l_2\} \tag{4.15}$$

$$\rho'_{NL} = \{\texttt{head} \mapsto 42\} \tag{4.16}$$

To change the first element of the list, for example, by multiplying it by two, the program must reconstruct the list after deconstructing the first element and replacing it with the new value, as shown below:

```
val (head, tail) = Cons.unapply(list).get
val listPrime = Cons(head * 2, tail)
```

This results in:

$$\rho''_L = \{\texttt{list'} \mapsto l_3\} \tag{4.17}$$

$$\sigma''_L = \{l_3 \mapsto \langle 84, l_2\rangle, \quad l_2 \mapsto \langle 99, 0\rangle\} \tag{4.18}$$

Overall, in pure linear data structures, accessing a node in the tree requires deconstructing the entire path from the root to that node. Moreover, when updating a nonlinear field of a linear value, the value must first be accessed, and then the entire path from the root must be reconstructed.

4.2.1.4 Memory Management

At any point in a well-formed memory, deleting a linear variable and, accordingly, freeing all linear locations reachable from that variable is safe. The notion of safety is defined as follows:

- No double-free occurs, because each linear location appears exactly once in the memory, either in another linear value or in a variable.
- No dangling references remain after deletion, because each linear location is only referenced by one other linear value or variable. As a result, freeing a location propagates through its parent structure, which is also freed.

Based on linearity rules, each linear variable is mentioned exactly once in the program. Therefore, each variable is either used in another expression or safely freed, resulting in no memory leaks and enabling static memory management for linear data structures.

4.2.1.5 Effect of `@HideLinearity`

In Scinear, `@HideLinearity` allows a program to construct memories whose linear part is not well formed. For example, the following code creates a cyclic data structure between two linear values.

```
def cycle[@HideLinearity T >: LinearLink <: LinearLink](): LinearLink =
  var x: T = LinearLink(None) // due to `@HideLinearity`, ok to have a `
   var` of type `T`
  x = LinearLink(Some(x))
  x
```

imem uses `@HideLinearity` to provide a more expressive approach while still performing safe memory management statically.

4.2.2 imem memory

imem extends linear memory by introducing new kinds of references. To explain the imem memory model more clearly, the first part describes these additional references, how a program uses them, the expected well-formedness, and the need for lifetimes. Then, the section introduces lifetimes and explains their role in tracking the availability of the additional references.

4.2.2.1 Memory With imem References

4.2.2.1.1 New References imem introduces new references that make accessing different parts of linear memory easier while preserving memory safety. The following is the list of the new references.

Box References

A box is a linear value that resembles conventional references, such as a C++ `unique` reference or a Rust `Box` reference. A box reference points to a linear or nonlinear location,

and the object residing in that location is called the resource in the box. Because boxes are linear, they form a tree structure. It is important to note that unlike the Rust counterpart, a box in imem does not own its resource, which is discussed in the next part (section 4.2.2.2).

Mutable and Immutable References

Mutable and immutable references are the only way that the program can access the resource in a box. An immutable reference provides read-only access, whereas a mutable reference allows both read and write access. To obtain such a reference, the program should first borrow the box. Then, the program can access the resource through the interface of the reference.

Immutable references are not linear, so they can be replicated without any restriction. In contrast, mutable references are linear.

Both reference types support re-borrowing. Because an immutable reference is not a linear value, re-borrowing is equivalent to replicating the immutable reference. Also, the program can re-borrow a mutable reference to derive either a new immutable reference or a new mutable reference.

4.2.2.1.2 Definition The program state remains the same as the linear program state:

$$(\rho, \sigma) \tag{4.19}$$

imem extends linear values by introducing two kinds of linear references:

$$\begin{aligned} \mathrm{Val}_L = \{\langle v_1, \ldots, v_k\rangle \mid v_i \in \mathrm{Loc}_L \cup \mathrm{Loc}_{NL} \cup \mathbb{Z}\} \\ \cup \{\mathrm{box}(l) \mid l \in \mathrm{Loc}_L \cup \mathrm{Loc}_{NL}\} \\ \cup \{\mathrm{mref}(l) \mid l \in \mathrm{Loc}_L \cup \mathrm{Loc}_{NL}\} \end{aligned} \tag{4.20}$$

These linear references are:

- A box(l) is a general reference that the program can borrow immutable and mutable references from.
- A mref(l) is a mutable reference, which provides read-write access to location l.

In addition, imem extends nonlinear values by adding immutable references:

$$\begin{aligned} \mathrm{Val}_{NL} = \{\langle v_1, \ldots, v_k\rangle \mid v_i \in \mathrm{Loc}_{NL} \cup \mathbb{Z}\} \\ \cup \{\mathrm{iref}(l) \mid l \in \mathrm{Loc}_L \cup \mathrm{Loc}_{NL}\} \end{aligned} \tag{4.21}$$

An immutable reference iref(l) is similar to a mutable reference but provides read-only access to the location and is a nonlinear value.

4.2.2.1.3 Program Required Operations To make these references practical, the program must be able to create references from one another and use them to access specific values in memory. The following list of operations models the features required by a program to create and use imem references. These operations are defined as functions that take a program state and additional arguments, and either validate that the requested operation is valid or return a new state that results from applying the operation to the memory state.

4.2.2.1.4 Operations on Boxes

- ***Create a New Box***: The operation gets a location l and creates a box that points to l. A fresh linear location $k \in \text{Loc}_L$ points to the new box.

$$\mathsf{newBox}\big((\rho, (\sigma_L, \sigma_{NL})),\ l,\ k\big) = \big(\rho, (\sigma_L[k \mapsto \text{box}(l)], \sigma_{NL})\big) \tag{4.22}$$

- ***Borrow a Box Immutably***: borrowImmutBox gets a box and creates an immutable reference that points to the same location. A fresh nonlinear location $k_{ir} \in \text{Loc}_{NL}$ points to the new immutable reference. Formally, if $\sigma_L(k_b) = \text{box}(l)$ and $k_{ir} \notin \text{dom}(\sigma_{NL})$:

$$\mathsf{borrowImmutBox}\big((\rho, (\sigma_L, \sigma_{NL})),\ k_b,\ k_{ir}\big) = \big(\rho, (\sigma_L, \sigma_{NL}[k_{ir} \mapsto \text{iref}(l)])\big) \tag{4.23}$$

- ***Borrow a Box Mutably***: The borrowMutBox operation behaves similar to borrowImmutBox, but it creates a mutable reference that points to the same location as the given box. As a result, a fresh linear location $k_{mr} \in \text{Loc}_L$ points to the new mutable reference. The operation is valid if $\sigma_L(k_b) = \text{box}(l)$ and $k_{mr} \notin \text{dom}(\sigma_L)$:

$$\mathsf{borrowMutBox}\big((\rho, (\sigma_L, \sigma_{NL})),\ k_b,\ k_{mr}\big) = \big(\rho, (\sigma_L[k_{mr} \mapsto \text{mref}(l)], \sigma_{NL})\big) \tag{4.24}$$

4.2.2.1.5 Operations on Immutable References

- ***Borrow an Immutable Reference***: This operation re-borrows an immutable reference and creates a new immutable reference that points to the same location as the given immutable reference. The memory stores the new immutable reference at a fresh nonlinear location $k_{ir} \in \text{Loc}_{NL}$. As a formal definition, if $\sigma_{NL}(k_{ir}) = \text{iref}(l)$ and $k'_{ir} \notin \text{dom}(\sigma_{NL})$, then:

$$\mathsf{borrowImmut}_{imm}\big((\rho,(\sigma_L,\sigma_{NL})),\ k_{ir},\ k'_{ir}\big) = \big(\rho,(\sigma_L,\sigma_{NL}[k'_{ir}\mapsto \mathrm{iref}(l)])\big) \tag{4.25}$$

- ***Reading an Immutable Reference***: This operation accesses the resource of an immutable reference through that immutable reference. It does not change the memory state. Instead, it only validates the program's request to use this reference. This access is valid as long as the immutable reference is reachable from the program environment. The definition uses $\mathrm{Reachable}_{Ref}(\rho,\sigma)$, which is defined in the well-formedness definition subsection. Briefly, $\mathrm{Reachable}_{Ref}(\rho,\sigma)$ is the set of all boxes and references that the environment can reach. Formally, if $k_{ir} \in \mathrm{dom}(\sigma_{NL})$ and $\sigma_{NL}(k_{ir}) = \mathrm{iref}(l)$, then:

$$\mathsf{readImmut}\big((\rho,(\sigma_L,\sigma_{NL})),\ k_{ir}\big) = \begin{cases} (\rho,(\sigma_L,\sigma_{NL})) & \text{if } \mathrm{iref}(l) \in \mathrm{Reachable}_{Ref}(\rho,\sigma) \\ \mathsf{nonvalid} & \text{otherwise.} \end{cases} \tag{4.26}$$

4.2.2.1.6 Operations on Mutable References

- ***Borrow a Mutable Reference Immutably***: The operation $\mathsf{borrowImmut}_{mut}$ gets a mutable reference and creates an immutable reference that points to the same location. The new immutable reference is stored at a fresh nonlinear location $k_{ir} \in \mathrm{Loc}_{NL}$. To define it formally, if $\sigma_L(k_{mr}) = \mathrm{mref}(l)$ and $k_{ir} \notin \mathrm{dom}(\sigma_{NL})$:

$$\mathsf{borrowImmut}_{mut}\big((\rho,(\sigma_L,\sigma_{NL})),\ k_{mr},\ k_{ir}\big) = \big(\rho,(\sigma_L,\sigma_{NL}[k_{ir}\mapsto \mathrm{iref}(l)])\big) \tag{4.27}$$

- ***Borrow a Mutable Reference Mutably***: This operation is similar to $\mathsf{borrowImmut}_{mut}$, but it creates a new mutable reference that resides at a fresh location $k'_{mr} \in \mathrm{Loc}_L$. If $\sigma_L(k_{mr}) = \mathrm{mref}(l)$ and $k'_{mr} \notin \mathrm{dom}(\sigma_L)$, then the operation is:

$$\mathsf{borrowMut}\big((\rho,(\sigma_L,\sigma_{NL})),\ k_{mr},\ k'_{mr}\big) = \big(\rho,(\sigma_L[k'_{mr}\mapsto \mathrm{mref}(l)],\sigma_{NL})\big) \tag{4.28}$$

- ***Writing a Mutable Reference***: Similar to reading an immutable reference, this operation does not change the memory state and only validates the program's access to a mutable reference. Also, similar to $\mathsf{readImmut}$, the operation is valid if the mutable reference is reachable from the program environment. Formally, if $k_{mr} \in \mathrm{dom}(\sigma_L)$ and $\sigma_L(k_{mr}) = \mathrm{mref}(l)$, then:

$$\mathsf{writeMut}\big((\rho,(\sigma_L,\sigma_{NL})),\ k_{mr}\big) = \begin{cases} (\rho,(\sigma_L,\sigma_{NL})) & \text{if } \mathrm{mref}(l) \in \mathrm{Reachable}_{Ref}(\rho,\sigma) \\ \mathsf{nonvalid} & \text{otherwise.} \end{cases} \tag{4.29}$$

4.2.2.1.7 An Example The following example demonstrates how a program can reach a simple memory state by using the defined operations. It also shows how certain uses of these operations can lead the memory state to an incorrect form. The example constructs a memory that contains two boxes: an outer box and an inner box. The outer box points to the inner box, and the inner box points to a nonlinear integer.

Let $k_{\mathsf{outer}}, k_{\mathsf{inner}} \in \mathrm{Loc}_L$ and $n_{42} \in \mathrm{Loc}_{NL}$. Assume the initial nonlinear memory already contains the integer:

$$(\rho_0, (\sigma_{L_0}, \sigma_{NL_0})) = \big(\emptyset,\ (\emptyset,\ \{n_{42} \mapsto 42\})\big) \tag{4.30}$$

First, the program creates a box that points to the nonlinear integer:

$$(\rho_1, (\sigma_{L_1}, \sigma_{NL_1})) = \mathsf{newBox}\big((\rho_0, (\sigma_{L_0}, \sigma_{NL_0})),\ n_{42},\ k_{\mathsf{inner}}\big) \tag{4.31}$$

This results in:

$$\sigma_{L_1} = \{k_{\mathsf{inner}} \mapsto \mathrm{box}(n_{42})\}, \qquad \sigma_{NL_1} = \{n_{42} \mapsto 42\} \tag{4.32}$$

Second, the program creates a second box that points to the inner box:

$$(\rho_2, (\sigma_{L_2}, \sigma_{NL_2})) = \mathsf{newBox}\big((\rho_1, (\sigma_{L_1}, \sigma_{NL_1})),\ k_{\mathsf{inner}},\ k_{\mathsf{outer}}\big) \tag{4.33}$$

This results in:

$$\sigma_{L_2} = \{k_{\mathsf{inner}} \mapsto \mathrm{box}(n_{42}), \quad k_{\mathsf{outer}} \mapsto \mathrm{box}(k_{\mathsf{inner}})\} \qquad \sigma_{NL_2} = \{n_{42} \mapsto 42\} \tag{4.34}$$

Assume that the linear environment contains a variable `outer` that points to the outer box:

$$\rho_{L_2} = \{\texttt{outer} \mapsto k_{\mathsf{outer}}\}, \qquad \rho_{NL_2} = \emptyset \tag{4.35}$$

Let $k_{\mathsf{outerIr}} \in \mathrm{Loc}_{NL}$ be fresh. In the next step, the program borrows the outer box immutably:

$$(\rho_3, (\sigma_{L_3}, \sigma_{NL_3})) = \mathsf{borrowImmutBox}\big((\rho_2, (\sigma_{L_2}, \sigma_{NL_2})),\ k_{\mathsf{outer}},\ k_{\mathsf{outerIr}}\big) \tag{4.36}$$

So:

$$\sigma_{NL_3} = \{n_{42} \mapsto 42, \quad k_{\mathsf{outerIr}} \mapsto \mathrm{iref}(k_{\mathsf{inner}})\} \tag{4.37}$$

Assume that the nonlinear environment stores the borrowed reference in the variable `outerImm`:

$$\rho_{NL_3} = \{\texttt{outerImm} \mapsto k_{\mathsf{outerIr}}\} \tag{4.38}$$

Next, the program accesses the `outerImm` reference using readImmut. This operation is valid if $\mathrm{iref}(k_{\mathsf{inner}})$ is reachable from the environment.

$$\mathsf{readImmut}\big((\rho_3, (\sigma_{L_3}, \sigma_{NL_3})),\ k_{\mathsf{outerIr}}\big) = (\rho_3, (\sigma_{L_3}, \sigma_{NL_3})) \tag{4.39}$$

To illustrate that this operation can produce a memory state that appears incorrect, the following presents two cases in which the operation result contains faults. The later well-formedness definition rules out these incorrect states.

Let $k'_{\mathsf{outer}} \in \mathrm{Loc}_L$ be fresh. The program can create a second box that also points to k_{inner}:

$$(\rho_4, (\sigma_{L_4}, \sigma_{NL_4})) = \mathsf{newBox}\big((\rho_3, (\sigma_{L_3}, \sigma_{NL_3})),\ k_{\mathsf{inner}},\ k'_{\mathsf{outer}}\big) \tag{4.40}$$

This results in:

$$\begin{aligned} \sigma_{L_4} = \{k_{\mathsf{inner}} \mapsto \mathrm{box}(n_{42}), \quad k_{\mathsf{outer}} \mapsto \mathrm{box}(k_{\mathsf{inner}}), \quad k'_{\mathsf{outer}} \mapsto \mathrm{box}(k_{\mathsf{inner}})\} \\ \sigma_{NL_4} = \{n_{42} \mapsto 42, \quad k_{\mathsf{outerIr}} \mapsto \mathrm{iref}(k_{\mathsf{inner}})\} \end{aligned} \tag{4.41}$$

In this state, two boxes point to the same inner box location k_{inner}. This is not aligned with the linear tree memory structure that imem tries to preserve through the boxes. The later well-formedness definition rules out this state.

In addition, the program can borrow the inner box mutably while the immutable reference to the outer box remains reachable. Let $k_{\mathsf{innerMr}} \in \mathrm{Loc}_L$ be fresh. Given the third memory state, $(\rho_5, (\sigma_{L_5}, \sigma_{NL_5}))$, as input, the program can borrow the inner box mutably:

$$(\rho_5, (\sigma_{L_5}, \sigma_{NL_5})) = \mathsf{borrowMutBox}\big((\rho_3, (\sigma_{L_3}, \sigma_{NL_3})),\ k_{\mathsf{inner}},\ k_{\mathsf{innerMr}}\big) \tag{4.42}$$

This results in:

$$\begin{aligned} \sigma_{L_5} = \{k_{\mathsf{inner}} \mapsto \mathrm{box}(n_{42}), \quad k_{\mathsf{outer}} \mapsto \mathrm{box}(k_{\mathsf{inner}}), \quad k_{\mathsf{innerMr}} \mapsto \mathrm{mref}(n_{42})\} \\ \sigma_{NL_5} = \{n_{42} \mapsto 42, \quad k_{\mathsf{outerIr}} \mapsto \mathrm{iref}(k_{\mathsf{inner}})\} \end{aligned} \tag{4.43}$$

In this state, the program can access the immutable reference `outerImm`, and it can also access the mutable reference $\mathrm{mref}(n_{42})$. This combination is not acceptable because the mutable reference can update the resource that the immutable reference can reach. The later well-formedness definition rules out this situation.

4.2.2.1.8 Well-formedness Additional Definitions

References Mentioned in a Value: The references that appear in a value v, denoted $\text{Refs}(v)$, are defined as follows:

$$\text{Refs}(v) = \begin{cases} \{\, v_i \mid v = \langle v_1, \ldots, v_k \rangle \ \wedge \ v_i \in \text{Loc}_L \cup \text{Loc}_{NL} \,\} & \\ \{l\} & v \in \{\text{box}(l),\ \text{iref}(l),\ \text{mref}(l)\} \\ \emptyset & \text{otherwise.} \end{cases} \tag{4.44}$$

Resource: A resource is a location that has an imem box reference pointing to it:

$$l \in \text{Res}(\sigma) \iff \exists k \in \text{dom}(\sigma_L).\ \sigma_L(k) = \text{box}(l) \tag{4.45}$$

Reachability: The domain of the full memory is:

$$\text{dom}(\sigma) = \text{dom}(\sigma_L) \ \cup \ \text{dom}(\sigma_{NL}) \tag{4.46}$$

For any $k \in \text{dom}(\sigma)$, $\sigma(k)$ denotes $\sigma_L(k)$ if $k \in \text{dom}(\sigma_L)$, and $\sigma_{NL}(k)$ if $k \in \text{dom}(\sigma_{NL})$.

A location l reaches another location l' in one step if and only if:

$$l \to l' \iff l \in \text{dom}(\sigma) \ \wedge \ l' \in \text{Refs}(\sigma(l)) \tag{4.47}$$

A location l reaches a location l' if and only if:

$$l \to^* l' \iff \exists n \geq 1,\ \exists l_0, \ldots, l_n \text{ such that } l_0 = l \ \wedge \ l_n = l' \ \wedge \ \forall i \in \{0, \ldots, n-1\}.\ l_i \to l_{i+1} \tag{4.48}$$

A location l is reachable from a linear variable $x \in \text{dom}(\rho_L)$ iff:

$$x \rightsquigarrow l \iff \rho_L(x) \to^* l \tag{4.49}$$

A reaching path $\pi \in \text{Paths}(x, l)$ from variable x to location l is a finite sequence of locations $(l_1, \ldots, l_n)$ such that:

- $l_1 = \rho_L(x)$,
- $l_n = l$,
- and $l_i \to l_{i+1}$ for all $1 \leq i < n$.

Reachable References: The set of reachable references, Reachable_{Ref}, contains every box, immutable reference, and mutable reference that is stored at a location reachable from a linear environment variable. In addition, Reachable_{Ref} contains every immutable reference stored at a location that a nonlinear environment variable points to.

$$\begin{aligned}\text{Reachable}_{Ref}(\rho,\sigma) = \{\, r \mid{}& \exists x \in \text{dom}(\rho_L).\ \exists k \in \text{dom}(\sigma).\ x \rightsquigarrow k \ \wedge\ \sigma(k) = r \\ & \wedge\ r \in \{\text{box}(\_),\ \text{iref}(\_),\ \text{mref}(\_)\}\,\} \\ & \cup\ \{\, r \mid \exists y \in \text{dom}(\rho_{NL}).\ \exists k \in \text{dom}(\sigma_{NL}).\ \rho_{NL}(y) = k \\ & \wedge\ \sigma_{NL}(k) = r\ \wedge\ r \in \{\text{iref}(\_)\}\,\}.\end{aligned} \tag{4.50}$$

Direct Boxes: The set of direct boxes of a location l is called $D(l)$. It contains all linear locations that store a box that points to l:

$$D(l,\sigma) = \{\, k \in \text{dom}(\sigma_L) \mid \sigma_L(k) = \text{box}(l)\,\} \tag{4.51}$$

Properties

An imem memory state (ρ,σ) is well formed if it satisfies the following properties. As indicated by the formal definitions of these properties, well-formedness mainly concerns about the portion of memory that is reachable from the environment.

Direct Box Uniqueness: Every linear location $l \in \text{Loc}_L$ has at most one reachable direct box that points to l. In other words, every linear location either:

- Is not reachable from environment.
- Or exactly one linear composite value stores it.
- Or exactly one reachable box points to it.

Formally:

$$\forall l \in \text{Loc}_L.\ \Rightarrow |\{k \in D(l,\sigma) \mid \sigma_L(k) \in \text{Reachable}_{Ref}\}| \leq 1 \tag{4.52}$$

No Cyclic Box: A box $b = \text{box}(l)$ must not reach itself through dereferencing:

$$\neg\exists k \in \text{dom}(\sigma).\ \big(\sigma(k) = \text{box}(l)\big)\ \wedge\ \big(l \rightarrow^* k\big) \tag{4.53}$$

Borrowing Validity: For every mutable or immutable reference that is reachable in memory, there exists a reachable box that points to the same location.

Formally:

$$\forall k \in \mathrm{dom}(\sigma_L).\ \forall l.\ \Big(\sigma_L(k) = \mathrm{mref}(l) \in \mathrm{Reachable}_{Ref}\Big) \Rightarrow \Big(\exists k_b \in \mathrm{dom}(\sigma_L).\ \sigma_L(k_b) = \mathrm{box}(l) \in \mathrm{Reachable}_{Ref}\Big) \tag{4.54}$$

$$\forall k \in \mathrm{dom}(\sigma_{NL}).\ \forall l.\ \Big(\sigma_{NL}(k) = \mathrm{iref}(l) \in \mathrm{Reachable}_{Ref}\Big) \Rightarrow \Big(\exists k_b \in \mathrm{dom}(\sigma_L).\ \sigma_L(k_b) = \mathrm{box}(l) \in \mathrm{Reachable}_{Ref}\Big) \tag{4.55}$$

This property means that every immutable and mutable reference is borrowed from a box that points to the same location.

Immutable Reference Not Reaching Mutable Reference: A reachable immutable reference $ir = \mathrm{iref}(l_i)$ should not reach a location that a reachable mutable reference points to:

$$\forall l_i, l_m.\ \Big(\mathrm{iref}(l_i) \in \mathrm{Reachable}_{Ref}(\rho, \sigma)\ \wedge\ \mathrm{mref}(l_m) \in \mathrm{Reachable}_{Ref}(\rho, \sigma)\Big) \Rightarrow \neg(l_i \rightarrow^* l_m) \tag{4.56}$$

This property results in immutable references reaching a constant portion of memory as long as they remain reachable.

4.2.2.1.9 Example Cont. The previous example produces two memory states that are not well formed. In the fourth state $(\rho_4, (\sigma_{L_4}, \sigma_{NL_4}))$, the location k_{inner} has two direct boxes, stored at k_{outer} and k'_{outer}. This violates ***Direct Box Uniqueness*** because $D(k_{\mathsf{inner}}, \sigma_{L_4})$ contains two reachable direct boxes.

In the fifth state $(\rho_5, (\sigma_{L_5}, \sigma_{NL_5}))$, the immutable reference `outerImm` is $\mathrm{iref}(k_{\mathsf{inner}})$, and the mutable reference is $\mathrm{mref}(n_{42})$. This violates ***Immutable Reference Not Reaching Mutable Reference*** because $k_{\mathsf{inner}} \rightarrow^* n_{42}$.

4.2.2.1.10 Well-formed But Not Correct The following example shows that the current definition is not sufficient. The example produces a memory state that satisfies the well-formedness properties, yet it still permits an incorrect access pattern. This observation motivates the addition of lifetimes and value holders to imem.

Let $k_b, k_{\mathsf{mr}_1}, k_{\mathsf{mr}_2} \in \mathrm{Loc}_L$ and $n_{42} \in \mathrm{Loc}_{NL}$. The initial memory state is just an integer:

$$(\rho_0, (\sigma_{L_0}, \sigma_{NL_0})) = \big(\emptyset,\ (\emptyset,\ \{n_{42} \mapsto 42\})\big) \tag{4.57}$$

The program starts by creating a box that points to the nonlinear integer:

$$(\rho_1, (\sigma_{L_1}, \sigma_{NL_1})) = \mathsf{newBox}\big((\rho_0, (\sigma_{L_0}, \sigma_{NL_0})),\ n_{42},\ k_b\big) \tag{4.58}$$

Also, let the linear environment contain a variable `b` that points to the box:

$$\rho_{L_1} = \{\mathtt{b} \mapsto k_b\}, \qquad \rho_{NL_1} = \emptyset \tag{4.59}$$

Then, the program mutably borrows the box. Let k_{mr_1} be fresh:

$$(\rho_2, (\sigma_{L_2}, \sigma_{NL_2})) = \mathsf{borrowMutBox}\big((\rho_1, (\sigma_{L_1}, \sigma_{NL_1})),\ k_b,\ k_{\mathsf{mr}_1}\big) \tag{4.60}$$

And the linear environment stores this mutable reference in $\mathtt{mr}_1$:

$$\rho_{L_2} = \{\mathtt{b} \mapsto k_b, \quad \mathtt{mr}_1 \mapsto k_{\mathsf{mr}_1}\} \tag{4.61}$$

After that, the program mutably borrows the mutable reference. Let k_{mr_2} be fresh:

$$(\rho_3, (\sigma_{L_3}, \sigma_{NL_3})) = \mathsf{borrowMut}\big((\rho_2, (\sigma_{L_2}, \sigma_{NL_2})),\ k_{\mathsf{mr}_1},\ k_{\mathsf{mr}_2}\big) \tag{4.62}$$

Assume that the linear environment stores the second mutable reference in $\mathtt{mr}_2$:

$$\rho_{L_3} = \{\mathtt{b} \mapsto k_b, \quad \mathtt{mr}_1 \mapsto k_{\mathsf{mr}_1}, \quad \mathtt{mr}_2 \mapsto k_{\mathsf{mr}_2}\} \tag{4.63}$$

At this point, the linear memory contains two mutable references that point to the same location n_{42}:

$$\sigma_{L_3} = \{k_b \mapsto \mathrm{box}(n_{42}), \quad k_{\mathsf{mr}_1} \mapsto \mathrm{mref}(n_{42}), \quad k_{\mathsf{mr}_2} \mapsto \mathrm{mref}(n_{42})\} \qquad \sigma_{NL_3} = \{n_{42} \mapsto 42\} \tag{4.64}$$

Now, the program writes through $\mathtt{mr}_1$ by using writeMut. This operation succeeds because $\mathrm{mref}(n_{42})$ is reachable from the environment:

$$\mathsf{writeMut}\big((\rho_3, (\sigma_{L_3}, \sigma_{NL_3})),\ k_{\mathsf{mr}_1}\big) = (\rho_3, (\sigma_{L_3}, \sigma_{NL_3})) \tag{4.65}$$

After that, the program also writes through $\mathtt{mr}_2$. This operation also succeeds for the same reason:

$$\mathsf{writeMut}\big((\rho_3, (\sigma_{L_3}, \sigma_{NL_3})),\ k_{\mathsf{mr}_2}\big) = (\rho_3, (\sigma_{L_3}, \sigma_{NL_3})) \tag{4.66}$$

This behavior is not aligned with the Stacked Borrows Model because a mutable reference derived from another mutable reference should not remain usable after the first mutable reference is accessed for writing. To fix this, imem adds a mechanism to expire $\mathtt{mr}_2$ after writing to $\mathtt{mr}_1$.

4.2.2.1.11 Overview An imem memory is a linear memory extended with additional references and a specific definition of well-formedness. In a well-formed memory state, boxes, due to being linear values and imem additional well-formedness properties, form a tree structure.

imem also allows programs to borrow immutable and mutable references from boxes. Adding edges from immutable and mutable references to their target locations, the box tree graph becomes a directed acyclic graph. The graph remains acyclic because these references do not introduce cycles.

In addition, well-formedness requires that immutable references do not reach mutable references. As a result, if an immutable reference is reachable, then it cannot reach any location that a reachable mutable reference targets.

Figure 4.1 demonstrates the reachable part of a well-formed memory.

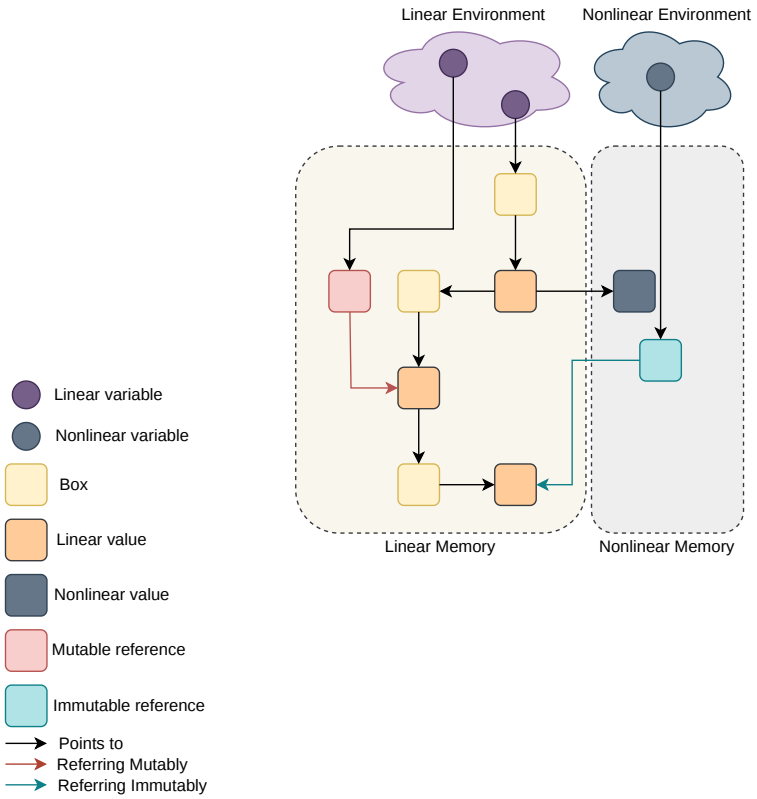

Figure 4.1: Imem Memory Overview

4.2.2.2 Memory with imem References And Lifetimes

To address the discussed issues, imem must relate the availability of some references to the unavailability of other references. For example, borrowing a box mutably should invalidate all immutable references that are borrowed from the same box. To achieve this, imem introduces lifetimes and a new type of reference, called a value holder, and augments existing references with lifetime sets. In addition, lifetimes enable static memory management.

A lifetime is an entity that becomes available once during program execution and then expires. The memory state tracks the lifetimes that are currently available:

$$(\rho, \sigma, \Lambda) \tag{4.67}$$

Where $\Lambda \subseteq \mathcal{L}$ is the set of available lifetimes. The set of linear values is:

$$\begin{aligned} \mathrm{Val}_L = \{&\langle v_1, \ldots, v_k\rangle \mid v_i \in \mathrm{Loc}_L \cup \mathrm{Loc}_{NL} \cup \mathbb{Z}\} \\ &\cup \{\mathrm{box}(l, \tau) \mid l \in \mathrm{Loc}_L \cup \mathrm{Loc}_{NL},\ \tau \subseteq \mathcal{L}\} \\ &\cup \{\mathrm{mref}(l, \tau) \mid l \in \mathrm{Loc}_L \cup \mathrm{Loc}_{NL},\ \tau \subseteq \mathcal{L}\} \\ &\cup \{\mathrm{hold}(l, \alpha) \mid l \in \mathrm{Loc}_L \cup \mathrm{Loc}_{NL},\ \alpha \in \mathcal{L}\} \end{aligned} \tag{4.68}$$

Boxes and mutable references are augmented with a lifetime set τ. A reference is available as long as all lifetimes in its associated set have not expired. In addition, Val_L now includes a new reference $\mathrm{hold}(l, \alpha)$, called a value holder, which prevents access to location l until lifetime α expires.

Furthermore, immutable references are also augmented with a lifetime set. As a result, the set of nonlinear values takes the following form:

$$\begin{aligned} \mathrm{Val}_{NL} = \{&\langle v_1, \ldots, v_k\rangle \mid v_i \in \mathrm{Loc}_{NL} \cup \mathbb{Z}\} \\ &\cup \{\mathrm{iref}(l, \tau) \mid l \in \mathrm{Loc}_L \cup \mathrm{Loc}_{NL},\ \tau \subseteq \mathcal{L}\} \end{aligned} \tag{4.69}$$

4.2.2.2.1 New and Changed Program Needed Operations The following operations update the previous definitions accordingly.

4.2.2.2.2 Operations on Boxes

- ***Create a New Box***: The operation gets a location l and a lifetime set τ, and it creates a box $\mathrm{box}(l, \tau)$. A fresh linear location $k \in \mathrm{Loc}_L$ points to the new box:

$$\mathsf{newBox}\big((\rho, (\sigma_L, \sigma_{NL}), \Lambda),\ l,\ \tau,\ k\big) = \big(\rho, (\sigma_L[k \mapsto \mathrm{box}(l, \tau)], \sigma_{NL}), \Lambda\big) \tag{4.70}$$

- ***Borrow a Box Immutably***: The operation borrowImmutBox gets a linear variable x_b that points to a box, a new reference lifetime set τ_{ir}, and a value holder lifetime $\alpha \in \tau_{ir}$. It borrows the box by removing x_b from the linear environment. Then, it adds a fresh linear variable x_h that points to a fresh location k_h that stores a value holder $\text{hold}(k_b, \alpha)$. This value holder points to the original box location k_b. Finally, the operation creates an immutable reference $\text{iref}(l, \tau_{ir})$ that points to the same location l, and it stores this reference at a fresh nonlinear location k_{ir}.

Formally, if $x_b \in \text{dom}(\rho_L)$, $\rho_L(x_b) = k_b$, $\sigma_L(k_b) = \text{box}(l, \tau_b)$, and there exist x_h, k_h, k_{ir} such that $x_h \notin \text{dom}(\rho_L)$, $k_h \notin \text{dom}(\sigma_L)$, and $k_{ir} \notin \text{dom}(\sigma_{NL})$, then:

$$\begin{aligned}&\textsf{borrowImmutBox}\big((\rho, (\sigma_L, \sigma_{NL}), \Lambda),\ x_b,\ \tau_{ir},\ \alpha\big) = \\ &\big((\rho_L \setminus \{x_b\}) \cup \{x_h \mapsto k_h\}\ \cup\ \rho_{NL},\ (\sigma_L[k_h \mapsto \text{hold}(k_b, \alpha)],\ \sigma_{NL}[k_{ir} \mapsto \text{iref}(l, \tau_{ir})]),\ \Lambda\big)\end{aligned} \tag{4.71}$$

- ***Borrow a Box Mutably***: The operation borrowMutBox is similar to borrowImmutBox, but it creates a mutable reference $\text{mref}(l, \tau_{mr})$ with a new lifetime set τ_{mr}. The operation borrowMutBox gets a linear variable x_b that points to a box, a new reference lifetime set τ_{mr}, and a value holder lifetime $\alpha \in \tau_{mr}$. It borrows the box by removing x_b from the linear environment. Then, it adds a fresh linear variable x_h that points to a fresh location k_h that stores a value holder $\text{hold}(k_b, \alpha)$. This value holder points to the original box location k_b. Finally, the operation creates a mutable reference $\text{mref}(l, \tau_{mr})$ that points to the same location l, and it stores this reference at a fresh linear location k_{mr}.

Formally, if $x_b \in \text{dom}(\rho_L)$, $\rho_L(x_b) = k_b$, $\sigma_L(k_b) = \text{box}(l, \tau_b)$, and there exist x_h, k_h, k_{mr} such that $x_h \notin \text{dom}(\rho_L)$, $k_h \notin \text{dom}(\sigma_L)$, and $k_{mr} \notin \text{dom}(\sigma_L)$, then:

$$\begin{aligned}&\textsf{borrowMutBox}\big((\rho, (\sigma_L, \sigma_{NL}), \Lambda),\ x_b,\ \tau_{mr},\ \alpha\big) = \\ &\big((\rho_L \setminus \{x_b\}) \cup \{x_h \mapsto k_h\}\ \cup\ \rho_{NL},\ (\sigma_L[k_h \mapsto \text{hold}(k_b, \alpha)][k_{mr} \mapsto \text{mref}(l, \tau_{mr})],\ \sigma_{NL}),\ \Lambda\big)\end{aligned} \tag{4.72}$$

4.2.2.2.3 Operations on Immutable References

- ***Borrow an Immutable Reference***: This operation re-borrows an immutable reference and creates a new immutable reference that points to the same location. It does not create a value holder. Formally, if $\sigma_{NL}(k_{ir}) = \text{iref}(l, \tau)$ and $k'_{ir} \notin \text{dom}(\sigma_{NL})$, then:

$$\mathsf{borrowImmut}_{imm}\big((\rho,(\sigma_L,\sigma_{NL}),\Lambda),\ k_{ir},\ \tau',\ k'_{ir}\big) = \big(\rho,(\sigma_L,\sigma_{NL}[k'_{ir}\mapsto \mathrm{iref}(l,\tau')]),\Lambda\big) \quad (4.73)$$

- ***Reading an Immutable Reference***: The operation readImmut is valid if the immutable reference is reachable from the environment, and every value holder on every reaching path is associated with a lifetime that is not in Λ. Formally, if $k_{ir} \in \mathrm{dom}(\sigma_{NL})$ and $\sigma_{NL}(k_{ir}) = \mathrm{iref}(l,\tau)$, then:

$$\mathsf{readImmut}\big((\rho,(\sigma_L,\sigma_{NL}),\Lambda),\ k_{ir}\big) = \begin{cases} (\rho,(\sigma_L,\sigma_{NL}),\Lambda) & \Big(\exists y \in \mathrm{dom}(\rho_{NL}).\ \rho_{NL}(y) = k_{ir} \lor \\ & \exists x \in \mathrm{dom}(\rho_L).\ x \rightsquigarrow k_{ir}\Big) \\ & \quad \land \Big(\forall x \in \mathrm{dom}(\rho_L).\ \forall \pi \in \mathrm{Paths}(x,k_{ir}).\ \forall i.\ \big(\sigma(\pi_i) = \mathrm{hold}(\_,\alpha) \Rightarrow \alpha \notin \Lambda\big)\Big) \\ \mathsf{nonvalid} & \text{otherwise.} \end{cases} \quad (4.74)$$

4.2.2.2.4 Operations on Mutable References

- ***Borrow a Mutable Reference Immutably***: The operation $\mathsf{borrowImmut}_{mut}$ gets a linear variable x_{mr} that points to a mutable reference, a new reference lifetime set τ_{ir}, and a value holder lifetime $\alpha \in \tau_{ir}$. It borrows the mutable reference by removing x_{mr} from the linear environment. Then, it adds a fresh linear variable x_h that points to a fresh location k_h that stores a value holder $\mathrm{hold}(k_{mr},\alpha)$. This value holder points to the original mutable reference location k_{mr}. Finally, the operation creates an immutable reference $\mathrm{iref}(l,\tau_{ir})$ that points to the same location l, and it stores this reference at a fresh nonlinear location k_{ir}.

Formally, if $x_{mr} \in \mathrm{dom}(\rho_L)$, $\rho_L(x_{mr}) = k_{mr}$, $\sigma_L(k_{mr}) = \mathrm{mref}(l,\tau_{mr})$, and there exist x_h, k_h, k_{ir} such that $x_h \notin \mathrm{dom}(\rho_L)$, $k_h \notin \mathrm{dom}(\sigma_L)$, and $k_{ir} \notin \mathrm{dom}(\sigma_{NL})$, then:

$$\mathsf{borrowImmut}_{mut}\big((\rho,(\sigma_L,\sigma_{NL}),\Lambda),\ x_{mr},\ \tau_{ir},\ \alpha\big) = \big((\rho_L \setminus \{x_{mr}\}) \cup \{x_h \mapsto k_h\} \ \cup\ \rho_{NL},\ (\sigma_L[k_h \mapsto \mathrm{hold}(k_{mr},\alpha)],\ \sigma_{NL}[k_{ir} \mapsto \mathrm{iref}(l,\tau_{ir})]),\ \Lambda\big) \quad (4.75)$$

- ***Borrow a Mutable Reference Mutably***: The operation $\mathsf{borrowMut}$ is similar to $\mathsf{borrowImmut}_{mut}$, but it creates a new mutable reference $\mathrm{mref}(l, \tau'_{mr})$ with a new lifetime set τ'_{mr}. The operation $\mathsf{borrowMut}$ gets a linear variable x_{mr} that points to a mutable reference, a new reference lifetime set τ'_{mr}, and a value holder lifetime $\alpha \in \tau'_{mr}$. It borrows the mutable reference by removing x_{mr} from the linear environment. Then, it adds a fresh linear variable x_h that points to a fresh location k_h that stores a value holder $\mathrm{hold}(k_{mr}, \alpha)$. This value holder points to the original mutable reference location k_{mr}. Finally, the operation creates a new mutable reference $\mathrm{mref}(l, \tau'_{mr})$ that points to the same location l, and it stores this reference at a fresh linear location k'_{mr}.

Formally, if $x_{mr} \in \mathrm{dom}(\rho_L)$, $\rho_L(x_{mr}) = k_{mr}$, $\sigma_L(k_{mr}) = \mathrm{mref}(l, \tau_{mr})$, and there exist x_h, k_h, k'_{mr} such that $x_h \notin \mathrm{dom}(\rho_L)$, $k_h \notin \mathrm{dom}(\sigma_L)$, and $k'_{mr} \notin \mathrm{dom}(\sigma_L)$, then:

$$\begin{aligned} \mathsf{borrowMut}\big((\rho, (\sigma_L, \sigma_{NL}), \Lambda),\ x_{mr},\ \tau'_{mr},\ \alpha\big) = \\ \big((\rho_L \setminus \{x_{mr}\}) \cup \{x_h \mapsto k_h\}\ \cup\ \rho_{NL}, \\ (\sigma_L[k_h \mapsto \mathrm{hold}(k_{mr}, \alpha)][k'_{mr} \mapsto \mathrm{mref}(l, \tau'_{mr})],\ \sigma_{NL}),\ \Lambda\big) \end{aligned} \tag{4.76}$$

- ***Writing a Mutable Reference***: The operation $\mathsf{writeMut}$ is valid if the mutable reference is reachable from the environment, and every value holder on every reaching path is associated with a lifetime that is not in Λ. Formally, if $k_{mr} \in \mathrm{dom}(\sigma_L)$ and $\sigma_L(k_{mr}) = \mathrm{mref}(l, \tau)$, then:

$$\mathsf{writeMut}\big((\rho, (\sigma_L, \sigma_{NL}), \Lambda),\ k_{mr}\big) = \begin{cases} (\rho, (\sigma_L, \sigma_{NL}), \Lambda) & \text{if } \exists x \in \mathrm{dom}(\rho_L).\ x \rightsquigarrow k_{mr} \\ & \quad \wedge \Big(\forall x \in \mathrm{dom}(\rho_L). \\ & \forall \pi \in \mathrm{Paths}(x, k_{mr}). \\ & \forall i.\ \big(\sigma(\pi_i) = \\ & \mathrm{hold}(\_, \alpha) \Rightarrow \alpha \notin \Lambda\big)\Big) \\ \mathsf{nonvalid} & \text{otherwise.} \end{cases} \tag{4.77}$$

To rule out incorrect memory states, the following subsections define extended well-formedness rules related to lifetimes and value holders.

A complete definition of these operations, their interaction with the memory state, and proofs that a correct program always results in a well-formed memory state is left as future work. The operations presented in this part are mainly those that relate to imem's goals, ownership and borrow checking. They illustrate how the new reference forms interact with each other and, therefore, motivate the well-formedness rules.

4.2.2.2.5 Well-formedness Similar to the previous version, memory well-formedness concerns only the part that is reachable from the environment and, in addition, available along the entire path from the environment to the location.

Additional Definitions

The *Resource*, *Reachability*, and *Direct Boxes* definitions remain the same. The definition of *References Mentioned in a Value* changes slightly by adding value holder references:

$$\mathrm{Refs}(v) = \begin{cases} \{\, v_i \mid v = \langle v_1, \ldots, v_k \rangle \ \wedge \ v_i \in \mathrm{Loc}_L \cup \mathrm{Loc}_{NL} \,\} & \\ \{l\} & \text{if } v \in \{\mathrm{box}(l,\tau),\ \mathrm{ref}(l,\tau),\ \mathrm{mref}(l,\tau),\ \mathrm{hold}(l,\alpha)\} \\ \emptyset & \text{otherwise.} \end{cases} \tag{4.78}$$

The following presents the new definitions.

Availability of References: imem associates a set of lifetimes $\tau \subseteq \mathcal{L}$ with the references $\mathrm{box}(l,\tau)$, $\mathrm{mref}(l,\tau)$, and $\mathrm{iref}(l,\tau)$. If any lifetime in τ expires, the reference becomes unavailable. Formally, a value $v \in \{\mathrm{box}(l,\tau), \mathrm{mref}(l,\tau), \mathrm{iref}(l,\tau)\}$ is available iff:

$$\mathrm{Avail}(v,\Lambda) \iff \tau \subseteq \Lambda \tag{4.79}$$

Available Linear Environment: The set of available linear variables, Var_{Avail}, consists of all linear variables in ρ_L for which all references they reach are available:

$$\mathrm{Var}_{Avail}(\rho,\sigma,\Lambda) = \{\, x \in \mathrm{dom}(\rho_L) \mid \forall r \in \mathrm{RefsReach}(x,\rho,\sigma).\ \mathrm{Avail}(r,\Lambda) \,\} \tag{4.80}$$

Where RefsReach is defined as follows:

$$\mathrm{RefsReach}(x,\rho,\sigma) = \{\, r \mid \exists k.\ x \rightsquigarrow k \ \wedge \ \sigma(k) = r \ \wedge \ r \in \{\mathrm{box}(\_,\_), \mathrm{ref}(\_,\_), \mathrm{mref}(\_,\_)\} \,\} \tag{4.81}$$

Available Reachable References: The set of available reachable references, AR_{Ref}, contains every available box, immutable reference, and mutable reference that is stored at a location reachable from an available linear environment variable:

$$\begin{aligned} \mathrm{AR}_{Ref}(\rho,\sigma,\Lambda) = \{\, r \mid \exists x \in \mathrm{Var}_{Avail}(\rho,\sigma,\Lambda).\ \exists k \in \mathrm{dom}(\sigma).\ x \rightsquigarrow k \ \wedge \ \sigma(k) = r \\ \wedge \ r \in \{\mathrm{box}(\_,\_),\ \mathrm{iref}(\_,\_),\ \mathrm{mref}(\_,\_)\} \ \wedge \ \mathrm{Avail}(r,\Lambda) \,\}. \end{aligned} \tag{4.82}$$

Access Expiration: Accessing a reference $r \in \{\text{box}(l, \_), \text{iref}(l, \_), \text{mref}(l, \_)\}$ expires a lifetime set τ if either τ is already expired, or every path that starts from an available linear variable and reaches r passes through a value holder $\text{hold}(\_, \alpha)$ such that $\alpha \in \tau$. Formally:

$$\begin{aligned}\text{AExp}_{(\rho,\sigma,\Lambda)}(r, \tau) \iff \tau \cap \Lambda \neq \emptyset \vee \forall x \in \text{Var}_{Avail}(\rho, \sigma, \Lambda).\ \forall k \in \text{dom}(\sigma).\ \Big(\\ \sigma(k) = r \wedge x \rightsquigarrow k \Rightarrow \\ \forall \pi \in \text{Paths}(x, k).\ \exists \alpha \in \tau.\ \exists i.\ \sigma(\pi_i) = \text{hold}(\_, \alpha) \Big)\end{aligned} \tag{4.83}$$

4.2.2.2.6 Properties An imem memory state (ρ, σ, Λ) is well-formed if it satisfies the following properties.

Direct Box Uniqueness: Every linear location $l \in \text{Loc}_L$ has at most one reachable available direct box that points to l:

$$|D(l, \sigma) \cap \text{Box}_{AR}(\rho, \sigma, \Lambda)| \leq 1 \tag{4.84}$$

No cyclic box: The same as previous definition.

No dangling references: For any two references $r_1 \in \{\text{box}(l_1, \tau_1), \text{iref}(l_1, \tau_1), \text{mref}(l_1, \tau_1)\}$ and $r_2 \in \{\text{box}(l_2, \tau_2), \text{iref}(l_2, \tau_2), \text{mref}(l_2, \tau_2)\}$, and for any location k such that $\sigma(k) = r_2$, if l_1 reaches k and r_2 becomes unavailable, then r_1 also becomes unavailable.

$$l_1 \rightarrow^* k \implies \tau_2 \subseteq \tau_1 \tag{4.85}$$

This property ensures that all references reachable from available references are available.

Borrowing Validity: For every reference $r \in \{\text{mref}(l, \tau_r), \text{iref}(l, \tau_r)\}$, there exists a box $b = \text{box}(l, \tau_b)$ that $\tau_b \subseteq \tau_r$.

This property means that every immutable and mutable reference is borrowed from a box, and that box always outlives the reference.

Box reaching Reference: If a box $b = \text{box}(l_b, \tau_b)$ reaches a location that a mutable or immutable reference $r \in \{\text{iref}(l_r, \tau_r), \text{mref}(l_r, \tau_r)\}$ points to it, then accessing b should expire r:

$$l_b \rightarrow^* l_r \implies \text{AExp}_{(\rho,\sigma,\Lambda)}(b, \tau_r) \tag{4.86}$$

Mutable Reference reaching Immutable Reference: If a mutable reference $mr = \text{mref}(l_m, \tau_m)$ reaches a location that an immutable reference $ir = \text{iref}(l_i, \tau_i)$ points to, then accessing mr should expire ir:

$$l_m \rightarrow^* l_i \implies \text{AExp}_{(\rho,\sigma,\Lambda)}(mr, \tau_i) \tag{4.87}$$

Mutable Reference reaching Mutable Reference: If a mutable reference $m_1 = \text{mref}(l_1, \tau_1)$ reaches another mutable reference $m_2 = \text{mref}(l_2, \tau_2)$:

- If $l_1 \neq l_2$: Accessing m_1 should expire m_2, meaning: $(l_1 \rightarrow^* l_2) \land (l_1 \neq l_2) \implies \text{AExp}_{(\rho,\sigma,\Lambda)}(m_1, \tau_2)$.
- If $l_1 = l_2$: Either accessing m_1 should expire m_2 or vice versa, formally: $(l_1 \rightarrow^* l_2) \land (l_1 = l_2) \implies \Big(\text{AExp}_{(\rho,\sigma,\Lambda)}(m_1, \tau_2) \lor \text{AExp}_{(\rho,\sigma,\Lambda)}(m_2, \tau_1)\Big)$.

The three reaching properties ensure that references follow the Stacked Borrows Model. Accessing a box expires all references borrowed from it, as well as references borrowed from reachable boxes that point to locations reachable from the box. Accessing a mutable reference expires all mutable and immutable references borrowed from it or borrowed from its reachable boxes.

Immutable Reference not reaching Mutable Reference: An available reachable immutable reference $ir = \text{iref}(l_i, \tau_i)$ should not reach a location that an available reachable mutable reference points to:

$$\begin{aligned} \forall ir, mr.\ \Big(&ir \in \text{AR}_{Ref}(\rho, \sigma, \Lambda) \land mr \in \text{AR}_{Ref}(\rho, \sigma, \Lambda) \land \\ &ir = \text{iref}(l_i, \tau_i) \land mr = \text{mref}(l_m, \tau_m)\Big) \implies \\ &\qquad \neg(l_i \rightarrow^* l_m) \end{aligned} \tag{4.88}$$

4.2.2.2.7 Overview imem extends the previous model with lifetimes and value holders, $\text{hold}(l, \alpha)$.

Based on the well-formedness rules, whenever a mutable or immutable reference points to a resource in a box, a value holder points to that box. Similarly, whenever an immutable or mutable reference points to the resource of a mutable reference, a value holder points to that mutable reference.

By adding value holders and their connections to the resources they reference to the graph described in the previous part, the graph remains a DAG. Figure 4.2 illustrates the available and reachable parts of a well-formed imem memory:

4.2.2.3 An Example

The following example shows a two-element list encoded in imem.

The set of active lifetimes is:

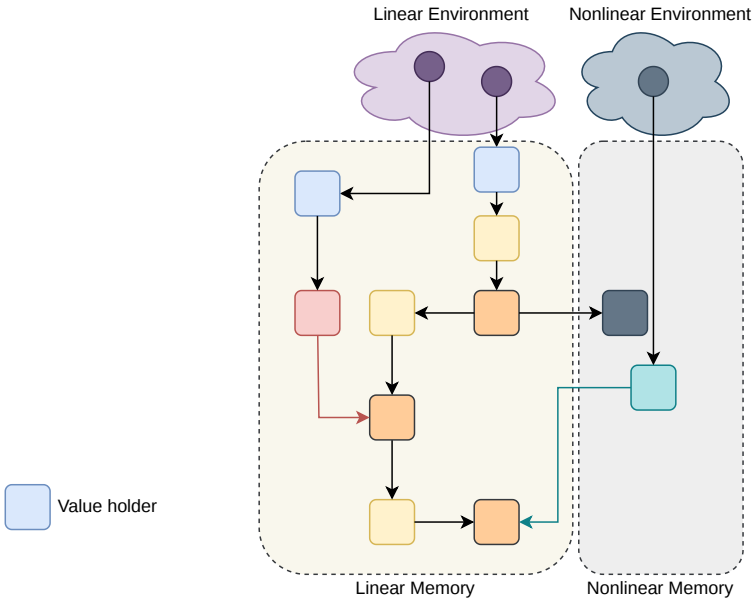

Figure 4.2: Imem Memory model Overview With Value Holders

$$\Lambda = \{\alpha\} \tag{4.89}$$

The linear environment contains a single variable that points to the list:

$$\rho_L = \{\texttt{list} \mapsto l_{\mathsf{list}}\} \tag{4.90}$$

The linear memory is composed of boxes and linear values that represent the structure of the list and its elements:

$$\begin{aligned}\sigma_L = \{&l_{\mathsf{list}} \mapsto \mathrm{box}(l_1, \{\alpha\}),\\ &l_1 \mapsto \langle l_{\mathsf{h1}},\ l_{\mathsf{t1}}\rangle,\\ &l_{\mathsf{h1}} \mapsto \mathrm{box}(n_{42}, \{\alpha\}),\\ &l_{\mathsf{t1}} \mapsto \mathrm{box}(l_2, \{\alpha\}),\\ &l_2 \mapsto \langle l_{\mathsf{h2}},\ l_{\mathsf{t2}}\rangle,\\ &l_{\mathsf{h2}} \mapsto \mathrm{box}(n_{99}, \{\alpha\}),\\ &l_{\mathsf{t2}} \mapsto \mathrm{box}(l_{\texttt{none}}, \{\alpha\}),\\ &l_{\texttt{none}} \mapsto \langle\ \rangle\}\end{aligned} \tag{4.91}$$

The nonlinear memory stores the integers:

$$\sigma_{NL} = \{n_{42} \mapsto 42, \quad n_{99} \mapsto 99\} \tag{4.92}$$

The variable `list` points to a location that contains a box referencing the first element of the list. This first element is stored at l_1 and is a linear value composed of two boxed fields. The first field, located at l_{h1}, is a box that points to a nonlinear integer value. The second field, located at l_{t1}, is a box that points to the second element of the list.

The following shows the result of applying the borrowMutBox and borrowImmutBox operations to the first and second elements, respectively, while keeping the memory well formed.

The set of available lifetimes now includes three elements:

$$\Lambda = \{\alpha, \beta, \gamma\} \tag{4.93}$$

The lifetimes β and γ are newly introduced. Lifetime β is added to the lifetime set of the mutable reference, $\tau_m = \{\alpha, \beta\}$, while lifetime γ is added to the lifetime set of the immutable reference, $\tau_i = \{\alpha, \beta, \gamma\}$.

The environments now contain three variables:

$$\rho_L = \{\texttt{list} \mapsto l_{\mathsf{list}}, \quad \texttt{firstMut} \mapsto l_{\mathsf{mut}}\} \qquad \rho_{NL} = \{\texttt{secondImm} \mapsto n_{\mathsf{secondImm}}\} \tag{4.94}$$

To support these references, the linear memory is extended with additional holds and a mutable reference:

$$\begin{aligned}
\sigma_L = \{ \quad & \underbrace{l_{\mathsf{list}} \mapsto \mathrm{hold}(l_{\mathsf{listBox}}, \beta)}_{\texttt{list} \text{ reaches the root box through a } \beta\text{-hold}}, \\
& l_{\mathsf{listBox}} \mapsto \mathrm{box}(l_1, \{\alpha\}), \\
& l_1 \mapsto \langle l_{\mathsf{h1}},\ l_{\mathsf{t1}} \rangle, \\
& l_{\mathsf{h1}} \mapsto \mathrm{box}(n_{42}, \{\alpha\}), \\
& l_{\mathsf{t1}} \mapsto \mathrm{box}(l_2, \{\alpha\}), \\
& l_2 \mapsto \langle l_{\mathsf{h2}},\ l_{\mathsf{t2}} \rangle, \\
& l_{\mathsf{h2}} \mapsto \mathrm{box}(n_{99}, \{\alpha\}), \\
& l_{\mathsf{t2}} \mapsto \mathrm{box}(l_{\texttt{none}}, \{\alpha\}), \\
& l_{\texttt{none}} \mapsto \langle\ \rangle, \\
& \underbrace{l_{\mathsf{mut}} \mapsto \mathrm{hold}(l_{\mathsf{mutCell}}, \gamma)}_{\texttt{firstMut} \text{ reaches the mref through a } \gamma\text{-hold}}, \\
& l_{\mathsf{mutCell}} \mapsto \mathrm{mref}(n_{42}, \{\alpha, \beta\})\}
\end{aligned} \tag{4.95}$$

The nonlinear memory is also extended to include an immutable reference:

$$\sigma_{NL} = \{n_{42} \mapsto 42, \quad n_{99} \mapsto 99, \quad n_{\mathsf{secondImm}} \mapsto \mathrm{iref}(n_{99}, \{\alpha, \beta, \gamma\})\} \tag{4.96}$$

In this configuration, imem allows the program to hold a mutable reference to the first list element and an immutable reference to the second element at the same time, without requiring the list to be deconstructed.

If the program accesses the box that points to the first element, the access must pass through the hold at location l_{list}. This action expires lifetime β, which in turn causes both the mutable reference and the immutable reference to become unavailable.

Similarly, accessing the mutable reference passes through the hold at location l_{mut}. This access expires lifetime γ, which results in the immutable reference becoming unavailable.

4.2.2.4 Memory Management

In a well-formed imem memory, it is safe to free a location that points to an unavailable box, as well as the location that the unavailable box points to, if no available direct box still points to that location. This safety follows from the fact that once a box becomes unavailable, it is no longer reachable from any available linear variable, and all references to it are also unavailable.

It is also safe to free a location that points to an immutable or mutable reference. For the same reason, no available variable can reach such references once they become unavailable.

Moreover, as in a purely linear memory, when a linear location that points to a composite linear value is freed, the locations contained within that value can also be safely freed, unless the locations point to a box, an immutable reference, or a mutable reference.

In this way, all memory regions reachable through references are managed safely and statically.

4.2.3 imem Implementation

The Scala library imem enforces static rules to ensure that the part of program memory it manages remains well formed throughout execution. These static rules include ownership rules and borrow-checking rules. Because Scala provides many features that may interfere with these rules, the soundness section (section 4.6) presents guidelines that a program must follow to keep its memory well formed.

4.3 Runtime

This section details the runtime verification mechanics of the imem library. It explains how imem validates operations against the Stacked Borrows Model rules and how the object graph evolves during runtime. This section focuses solely on the dynamic behavior of the user-facing components. The presented class definitions and interfaces include only their runtime functionality. Static mechanisms, such as ownership and borrow checking, and their implementation details are reserved for subsequent sections.

4.3.1 Internal components

4.3.1.1 Resource

As defined in the memory management section (section 4.2.2.1), a resource is an instance of any class for which imem manages access. The resource class can be a built-in class, a user-defined class, or a non-private imem class, such as `Box`, `ImmutRef`, or `MutRef`.

4.3.1.2 Unsafe Reference

imem stores each resource in an unsafe reference. Storing means that each unsafe reference contains a Scala reference to the resource as a field. These unsafe references are the only memory objects that directly refer to a resource. In addition, the references are private, so the user never has direct access to them.

```
1 private[imem] class UnsafeRef[T](private val resource: T):
2   private[imem] def applyAction[S](action: T => S): S = action(resource)
3
4   private[imem] def unsafeGet(): T = resource
5
6   ... // rest of the implementation
7 end UnsafeRef
```

The `UnsafeRef` interface provides two methods for interacting with the stored resource. The `applyAction` method follows a continuation-passing style, accepting an action and applying it to the resource. The `unsafeGet` method returns the resource directly, and imem uses this method to move resources.

4.3.2 Internal Reference

imem stores each unsafe reference inside an internal reference. This internal reference records all accesses to the unsafe reference and ensures that these accesses follow the Stacked Borrows Model rules. These accesses include read or write operations, requests

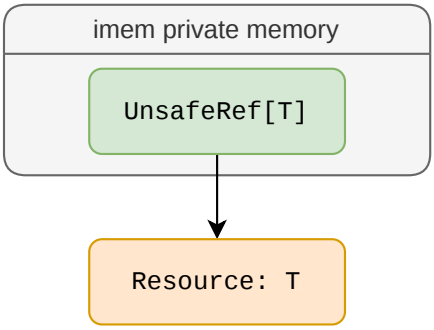

Figure 4.3: Unsafe Reference

for new read or write permissions, and checks on whether a permission remains valid. All accesses are performed through the methods of `InternalRef`.

Internal references are mutable and provide an interface for updating the underlying unsafe reference at runtime. Similar to unsafe references, internal references are private and inaccessible outside the imem package.

```
1  private[imem] class InternalRef[T](private var unsafeRef: UnsafeRef[T]):
2    private type Timestamp = Long
3
4    private[imem] enum Tag(val timestamp: Timestamp):
5      case Uniq(override val timestamp: Timestamp) extends Tag(timestamp)
6      case Shr(override val timestamp: Timestamp) extends Tag(timestamp)
7    end Tag
8
9    private class Stack:
10     val borrows = scala.collection.mutable.Stack[InternalRef[T]#Tag]()
11   end Stack
12
13   private var currentTimeStamp = 0
14   private val stack = Stack()
15
16   /** Checks (NEW-MUTABLE-REF) rule
17     */
18   @throws(classOf[IllegalStateException])
19   private[imem] def newUnique(derived: InternalRef[T]#Tag): InternalRef[T
      ]#Tag = ...
20
21   /** Checks (USE-1) rule
22     */
23   @throws(classOf[IllegalStateException])
24   private[imem] def useCheck(tag: InternalRef[T]#Tag): Unit = ...
25
26   /** Checks (NEW-SHARED-REF-1) rule
```

```
27      */
28    @throws(classOf[IllegalStateException])
29    private[imem] def newShared(derived: InternalRef[T]#Tag): InternalRef[T
       ]#Tag = ...
30
31    /** Checks (READ-1) rule
32      */
33    @throws(classOf[IllegalStateException])
34    private[imem] def readCheck(tag: InternalRef[T]#Tag): Unit = ...
35
36    @throws(classOf[IllegalStateException])
37    private[imem] def read[S](tag: InternalRef[T]#Tag, readAction: T => S):
        S = ...
38
39    @throws(classOf[IllegalStateException])
40    private[imem] def write[S](tag: InternalRef[T]#Tag, writeAction: T => S
       ): S = ...
41
42    private[imem] def unsafeGet(): T = unsafeRef.unsafeGet()
43
44    private[imem] def setResource(resource: T): Unit =
45      unsafeRef = UnsafeRef(resource)
46
47    private[imem] def drop(): Unit = ...
48
49    ... // rest of the implementations
50  end InternalRef
```

`InternalRef` tracks accesses by using tags. Each tag represents an access point. When the program needs to read from or write to the resource, it must present a tag. Then, `InternalRef` checks whether the tag is valid and expires tags that should become unavailable when that access occurs, based on the Stacked Borrows Model.

A tag is of one of two kinds: it is either unique (`Unq`), which permits both read and write access, or shared (`Shr`), which allows read-only access. Also, every tag has a unique timestamp, where higher values indicate more recent tags.

Each internal reference maintains a stack of all live tags in `stack.borrows`. The `newUnique` and `newShared` methods create a new tag by deriving it from an existing tag, starting with a root tag created during the internal reference construction. These methods check whether the tag holder has permission to request a new tag based on the `NEW-MUTABLE-REF` and `NEW-SHARED-REF-1` rules, respectively, given the current stack state. If the check succeeds, the methods return the new tag; otherwise, they throw an `IllegalStateException`.

The `read` and `write` methods perform read and write actions on the resource. Before applying an action, these methods call `readCheck` and `useCheck` to determine whether the `READ-1` and `USE-1` rules permit the requested actions. If the tag does not have sufficient access permissions or is no longer live, the methods throw an `IllegalStateException`.

Note that the rule checks performed by `readCheck` and `useCheck` are impure. In other words, these checks may modify `stack.borrows` by expiring tags that must be popped from the stack according to the Stacked Borrows Model rules.

The `setResource` method updates the underlying resource by creating a new unsafe reference that points to the resource.

The `drop` method clears the stack. `imem` uses this method, along with `unsafeGet`, to move resources between internal references.

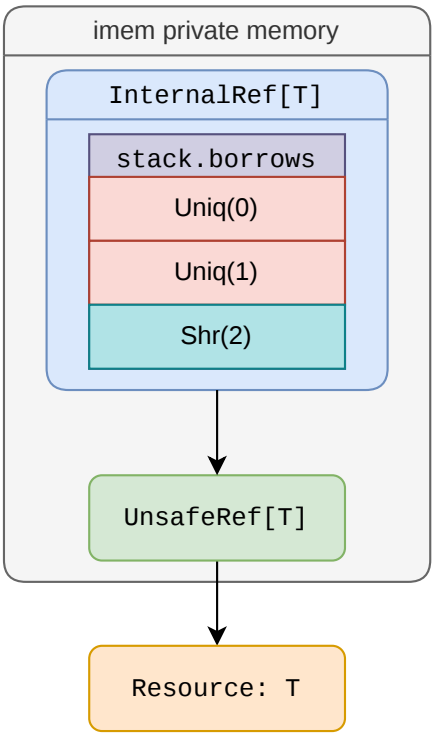

Figure 4.4: Internal Reference

4.3.3 User Facing Components

4.3.3.1 Box

The `Box` class implements the box references defined in section 4.2.2.1. Structurally, a box contains an internal reference and its root tag. The root tag has a unique (`Unq`) type, and its timestamp is the smallest. As a result, all derived tags have a larger timestamp than the root tag.

The `Box` class provides two functions, `borrowImmutBox` and `borrowMutBox`, which implement borrowImmutBox and borrowMutBox operations. They borrow a box and create, respectively, an immutable or a mutable reference to the resource pointed to by the box. During borrowing, the box derives a new tag from its own tag. This derivation produces either an immutable or a mutable reference, depending on the borrowing mode.

Similar to internal references, `Box` types are mutable and can modify their underlying resource through the associated internal reference.

```
class Box[T, ...](
  private [imem] val tag: InternalRef[T]#Tag,
  private [imem] val internalRef: InternalRef[T]
) extends scinear.Linear

def borrowImmutBox[... T, ...](self: Box[T, ...]^)(...): (ImmutRef[T,
    ...], ...) = ...
def borrowMutBox[... T, ...](self: Box[T, ...]^)(...): (MutRef[T, ...],
    ...) = ...

def setBox[... T, ...](self: Box[T, ...]^, resource: T)(...): Box[T,
    ...]^{self} = ...
def swapBox[... T, ...](self: Box[T, ...]^, other: Box[T, ...]^)(...): (
    Box[T, ...]^{self}, Box[T, ...]^{other}) = ...

... // rest of the interface
```

Since `Box` is a linear type, its interface is defined by functions rather than methods. The functions `borrowImmutBox` and `borrowMutBox` take a `Box` as input and return an immutable or mutable borrowed reference, respectively.

The `setBox` and `swapBox` functions provide mutability for a `Box`. The `setBox` function updates the resource of a box with a new value, whereas `swapBox` exchanges the resources of two boxes. These operations perform a `useCheck` on the internal reference of the box.

4.3.3.2 Mutable and Immutable References

`ImmutRef` and `MutRef` classes implement immutable and mutable references, iref and mref, defined in imem memory management (section 4.2.2.1):

```
class ImmutRef[T, ...](
  private[imem] val tag: InternalRef[T]#Tag,
  private[imem] val internalRef: InternalRef[T]
)

def borrowImmut[... T, ...](self: ImmutRef[T, ...]^)(...): ImmutRef[T,
    ...] = ...
def read[... T, ...](self: ImmutRef[T, ...]^, readAction: ... ?-> T^ -> S
    )(...): S = ...
... // rest of the interface

class MutRef[T, ...](
  private[imem] val tag: InternalRef[T]#Tag,
  private[imem] val internalRef: InternalRef[T]
) extends scinear.Linear

def borrowMut[... T, ...](self: MutRef[T, ...]^)(...): (MutRef[T, ...],
    ...) = ...
def borrowImmut[... T, ...](self: MutRef[T, ...]^)(...): (ImmutRef[T,
    ...], ...) = ...
def write[... T, ...](self: MutRef[T, ...]^, writeAction: ... ?-> T^ -> S
    )(...): S = ...
... // rest of the interface
```

Both reference types hold an internal reference and an access tag.

The `borrowImmut` and `borrowMut` functions enable reference re-borrowing, implementing the borrowImmut$_{imm}$, borrowImmut$_{mut}$ and borrowMut operations. The ownership section (section 4.4) explains the mechanism that invalidates derived references once the primary mutable reference is used at compile time. At runtime, accessing the primary mutable reference removes all derived reference tags from stack of the internal reference.

```
// assume box: `Box[Int]` exists
val (mutRefPrimary, _) = imem.borrowMutBox[Int, ...](box)
val (mutRefDerived, _) = imem.borrowMut[Int, ...](mutRefPrimary)

imem.write[Int, ...](mutRefPrimary, _ => ())
imem.write[Int, ...](mutRefDerived, _ => ()) // error: `mutRefDerived`'s
    tag is popped
```

Furthermore, `ImmutRef` and `MutRef` provide `read` and `write` interfaces, respectively, which implement the readImmut and writeMut operations.
These functions follow a continuation-passing style, applying the given action to the resource if the Stacked Borrows Model allows it.

All interfaces perform either a `readCheck` or a `useCheck` on the underlying internal reference, depending on the type of operation.

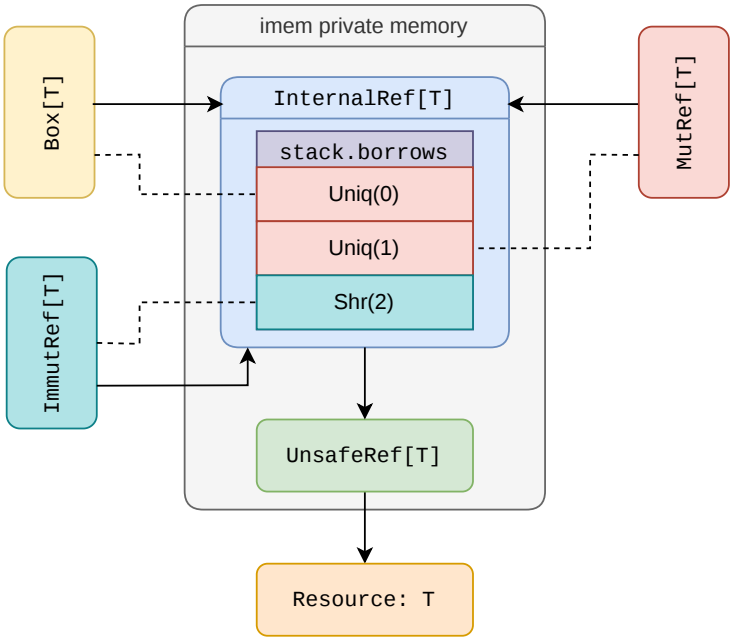

Figure 4.5: Box, Immutable and Mutable Reference

4.3.3.3 Nested Boxes

Boxes are most useful when they are nested, meaning that the resource in a box is another box is another box. Using this structure, boxes can form common data structures such as linked lists and trees.

```
// assume outerBox: `Box[Box[Int]]` exists
val (refOuter, _) = imem.borrowMutBox[Box[Int], ...](outerBox)

val refInner = imem.write[Box[Int], ...](refOuter,
  innerBox =>
    val (refInner, _) = imem.borrowImmutBox[Int, ...](innerBox)
    refInner
)
```

At runtime, the nested box in the example above structures the object graph shown in Figure 4.6.

As the example demonstrates, to access the inner box, the program first borrows the outer box and then calls the `read` or `write` function on the borrowed reference, `refOuter`. The `read` or `write` function can return a borrowed reference to the inner box, `refInner`. This process results in simultaneous references to both the inner and outer boxes.

The program can follow a similar process to have an immutable reference to the outer

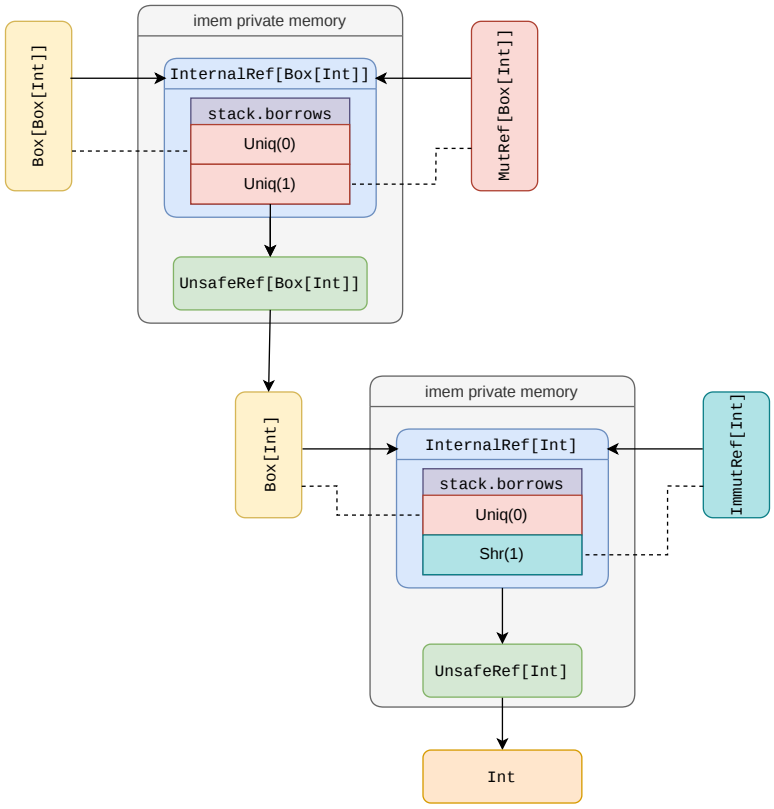

Figure 4.6: Two Nested Boxes

box and a mutable reference to the inner box:

```
val (refMutOuter, boxHolder) = imem.borrowMutBox[Box[Int], ...](outerBox)
// The borrow checking section explains what a `boxHolder` is.
// For now, assume it stores the borrowed `Box`.

val refMutInner = imem.write[Box[Int], ...](refMutOuter,
  innerBox =>
    val (refMutInner, _) = imem.borrowMutBox[Int, ...](innerBox)
    refMutInner
)

// gets the `outerBox` out of the holder
val outerBox2 = imem.unlockHolder(..., boxHolder) // <-- static imem
    rules expires `refMutInner` here

val refImmutOuter = imem.borrowImmutBox[Box[Int], ...](outerBox2)
```

Similar to the original Stacked Borrows paper [11], imem's runtime verification allows `refImmutOuter` and `refMutInner` to coexist. The runtime permits this coexistence as long

as `refImmutOuter` does not immutably borrow the inner Box.

```
// ‘refMutInner‘ and ‘refImmutOuter‘ coexists here based on the runtime
    rules.
// imem.write[Int, ...](refMutInner, _ => ()) <-- ok with no runtime
    error

imem.read[Box[Int], ...](refImmutOuter,
  innerBox =>
    // asks for a new ‘Shr‘ tag deriving from the box’s ‘Unq‘ tag.
    // pops ‘Unq‘ tag for ‘refMutInner‘.
    imem.borrowImmutBox[Int, ...](innerBox)
)

imem.write[Int, ...](refMutInner, _ => ()) // runtime error
```

Similar to Miri [13] and Rust's borrow checker, imem's static rules are more restrictive than its runtime verification. In the example above, the `refMutInner` would expire as soon as the `refMutOuter` is reborrowed immutably. The borrow checking section explains why `refMutInner` and `refImmutOuter`, which are references to two nested boxes, cannot coexist in a program that successfully compiles using the imem library.

4.3.4 Object Graph

Imem objects follow the same tree structure as linear objects. Figure 4.7 demonstrates imem's memory overview, if the program does not violate imem dynamic verification, and scinear rules.

This illustration shows only reachable boxes and references whose tags remain in their corresponding internal borrow stacks. In other words, the diagram represents the state of live references and boxes at a specific point during the program's execution. Boxes are linear, but they can have borrowed references from other boxes as their resources, so they form a Directed Acyclic Graph, DAG, structure. At some nodes in the graph, the program borrows the box either mutably or immutably, yielding a mutable or an immutable reference, respectively.

4.4 Ownership

This section explains how imem statically manages ownership of boxes and resources.

4.4.1 Ownership Goals

Ownership management in imem's implementation aims to satisfy the following properties of a well-formed imem memory:

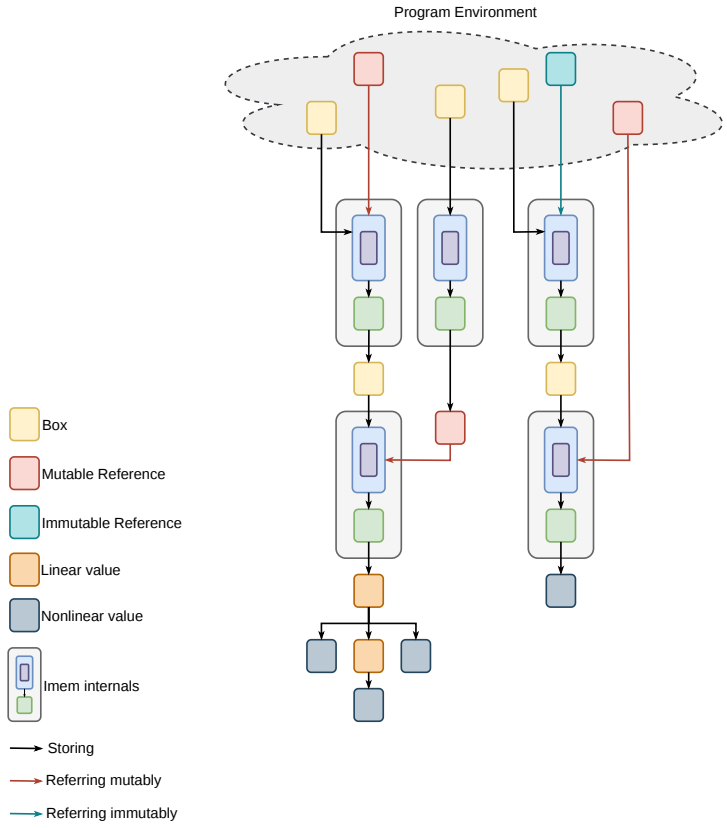

Figure 4.7: Runtime Memory Overview

- ***No dangling boxes:*** This property is a subset of the *No dangling references* well-formedness property. This subset states that, during execution, all boxes reachable from available variables are available themselves. The combination of ownership and borrow checking derives the full *No dangling references* property, and the details are described in the section 4.6.1.
- ***Direct Box Uniqueness*:** This property is defined in section 4.2.2.1.

4.4.2 Lifetime

In the imem implementation, the `Lifetime` class, is a linear capability class whose instances own boxes, references, and other lifetimes. A lifetime capability owns a box, reference, or another lifetime if their types include a type parameter instantiated with a capture set that includes the lifetime capability.

This ownership relation is many-to-many. That is, multiple lifetime instances can own a box, a reference, or another lifetime, and a single lifetime instance can own multiple boxes, references, or lifetimes.

The program section between a lifetime capability definition and its usage is called the lifetime's region, where the program can refer to entities owned by the instance. Those entities become out of reach of the program, also known as unavailable, after the lifetime region ends.

```
class Lifetime[OwnersInput^] extends scinear.Linear, caps.Capability:
  type Owners^ = {OwnersInput, this}
  ... // rest of the implementation
end Lifetime
```

The `Lifetime` class takes the `OwnersInput^` capture-set type parameter, which represents the lifetime capabilities that own this lifetime and therefore outlive it. The class also includes an `Owners^` type member, which denotes the list of the lifetime's owners together with the lifetime capability itself. The following illustrates how to define a lifetime and use it:

```
val lf = Lifetime[{}]() // region of ‘lf‘ begins
// ‘lf‘ is available here
lf.consume() // region of ‘lf‘ ends
// ‘lf‘ is not available anymore
```

Lifetimes can also be structured in a nested manner:

```
val outerLf = Lifetime[{}]() // region of ‘outerLf‘ begins
val innerLf = Lifetime[outerLf.Owners]() // region of ‘innerLf‘ begins
// both ‘outerLf‘ and ‘innerLf‘ are available here
innerLf.consume() // region of ‘innerLf‘ ends
// ‘outerLf‘ is still available here
outerLf.consume() // region of ‘outerLf‘ ends
// neither are available anymore
```

The program can also define and use lifetimes intertwined as long as they do not own each other:

```
val lf1 = Lifetime[{}]() // region of ‘lf1‘ begins
val lf2 = Lifetime[{}]() // region of ‘lf2‘ begins
// both ‘lf1‘ and ‘lf2‘ are available here
lf1.consume() // region of ‘lf1‘ ends
// ‘lf2‘ is still available here, but ‘lf1‘ is not
lf2.consume() // region of ‘lf2‘ ends
// neither are available anymore
```

If two lifetimes are intertwined and the first lifetime owns the second lifetime, then the second lifetime becomes unavailable after the first lifetime is used. Any subsequent use of the second lifetime results in an error.

```
val lf1 = Lifetime[{}]() // region of ‘lf1‘ begins
val lf2 = Lifetime[lf1.Owners]() // region of ‘lf2‘ begins
// both ‘lf1‘ and ‘lf2‘ are available here
lf1.consume() // region of ‘lf1‘ ends
// ‘lf2‘ is not available anymore because its owner ‘lf1‘ expired
lf2.consume() // error: Linear value lf1 is mentioned in the type of lf2
    but is either not in scope or used already.
```

4.4.3 Box Ownership

Unlike in Rust, boxes in imem do not own themselves; instead, they are owned by lifetimes. From the imem perspective, boxes are linear instances whose resources can be accessed by converting them into immutable or mutable references through borrowing. On the other hand, as in Rust, the program can change the owner of a box by moving it.

4.4.3.1 Creating Boxes

The following is the `Box` class definition:

```
class Box[T, @caps.use Owner^](
  private [imem] val tag: InternalRef[T]#Tag,
  private [imem] val internalRef: InternalRef[T]
) extends scinear.Linear
```

In addition to the fields related to the internal reference, it includes a capture-set type parameter `Owner^`, which can be instantiated with a set of lifetime capabilities.

imem provides the following function for creating a new box:

```
def newBox[@scinear.HideLinearity T, Owner^](resource: T): Box[T, Owner]
    = ...
```

The `newBox` function takes a resource instance as an argument and a capture set of owners as type arguments, and it returns a box that holds the given resource and is owned by the set of owners specified by the `Owner^` type argument.

Note that the program provides the resource, and imem assumes that the resource is neither copied nor stored elsewhere. The resource must be linear, such as an instance of `Box`, and the Scinear plugin statically ensures that the resource value is reachable only from the expression passed to `newBox`. However, if the resource is a linear class that contains non-linear fields, then the program is responsible for handling any side effects that arise from parts of the resource that have additional aliases. More details about this are explained in the soundness section (section 4.6).

The following is an example of creating a box with a lifetime owner set:

```
val box1 = newBox[LinearInt, {lf1}](LinearInt(42))
val box2 = newBox[LinearInt, {lf1, lf2}](LinearInt(42))
```

As with other linear types, boxes can also appear as fields of other linear classes:

```
// `0^` is the owner set of the box fields in `BoxPair` and, consequently
    , the owner set of the `BoxPair` itself.
class BoxPair[0^](...) extends scinear.Linear:
  val first: Box[Int, 0] ...
  val second: Box[Int, 0] ...
end BoxPair
```

The primary way to use boxes is to define data-structure classes as linear, but instead of storing them directly as fields within one another, to store a `Box` that wraps the type. This approach simplifies working with the data structure compared to an entirely linear implementation, while also guaranteeing safe memory management and alias control in contrast to a non-linear version.

The following section provides an overview of the internal structures that define a linked list using imem:

```
type Link[T <: scinear.Linear, 0^] = Option[Box[Node[T, 0], 0]]

class Node[T <: scinear.Linear, 0^](...) extends scinear.Linear:
  val elem: Box[T, 0] ...
  val next: Box[Link[T, 0], 0] ...
end Node

class List[T <: scinear.Linear, 0^](...) extends scinear.Linear:
  val head: Box[Link[T, 0], 0] ...
end List
```

The Evaluation (chapter 5) details more about implementing a linked list using imem.

4.4.3.2 Type Checking Dangling Boxes

Boxes are useful when they are nested, either directly, `Box[Box[...], ...]`, when the resource of one box is another box, or indirectly when the resource is a class that has boxes as fields.

```
// Two nested boxes, with the same owners, which is {lf1}:
val outerBox1 = newBox[Box[LinearInt, {lf1}], {lf1}](newBox(LinearInt(1))
    )

// Two nested boxes, with different owners, which are {lf1} for the outer
     box and {lf2} for the inner box:
val outerBox2 = newBox[Box[LinearInt, {lf2}], {lf1}](newBox(LinearInt(1))
    )
```

As illustrated in the example above, nested boxes may have different owners, which means that an inner box can be owned by a different set of lifetimes than the outer box. This situation may appear to allow the owners of the inner box to expire before those of the outer box, which would result in a dangling box:

```
// Two nested boxes, with different owners, which are {lf1} for the outer
    box and {lf2} for the inner box:
val outerBox2 = newBox[Box[LinearInt, {lf2}], {lf1}](newBox(LinearInt(1))
    )

// expiring the inner box owner, which is {lf2}:
lf2.consume()

// Can the program access `outerBox2` here?
```

Although this situation is possible, as shown above, it does not lead to dangling boxes. Unlike the *No dangling references* property defined in memory management, the imem implementation does not enforce that a box reaching another box, either directly or indirectly, has an owner lifetime that is a superset of the owner set of the box that is reached. However, the *dangling-box-unavailability-lemma* proves that the implementation ensures a box becomes unavailable when it can reach a box with an expired lifetime in its owner set.

As a demonstration, in the previous example, the outer box does not mention the expired lifetime capability directly. However, Scinear still rejects any attempt to refer to the outer box after that the lifetime capability expires:

```
// Two nested boxes, with different owners, which are {lf1} for the outer
    box and {lf2} for the inner box:
val outerBox2 = newBox[Box[LinearInt, {lf2}], {lf1}](newBox(LinearInt(1))
    )

// expiring the inner box owner, which is {lf2}:
lf2.consume()

outerBox2.consume() // error: Linear value lf2 is mentioned in the type
    of outerBox2 but is either not in scope or used already.
```

4.4.3.3 Modifying Boxes

Section 4.5 explains how to borrow a box to access its resource, but, at the same time, imem provides two interfaces that allow direct modification of a box.

One function is for setting a box's resource:

```
def setBox[@scinear.HideLinearity T, Owner^, ...](self: Box[T, Owner]^,
    resource: T)(...): Box[T, Owner]^{self} = ...
```

Because `Box` is linear, the `setBox` function returns a new `Box` reference that points to the same `Box` instance passed as the `self` argument.

Two aspects of the function signature make `setBox` compatible with other `imem` interfaces. First, the `self` argument uses a capturing type with the annotated capture set `cap`. The type `Box[T, Owner]^` is equivalent to `Box[T, Owner]^cap`. This choice allows

an escape-checked box to be passed to the function. Resource ownership (section 4.4.4) explains why this property is essential.

Second, the returned `Box` captures `self`, as indicated by the type `Box[T, Owner]^self`. Therefore, if the `self` is escape-checked, then the returned reference is also escape-checked, meaning it remains bound to the same scope as the input argument.

The other function is for swapping the resources of two boxes:

```
def swapBox[@scinear.HideLinearity T, @caps.use Owner^, @caps.use
    OtherOwner^, ...](
  self: Box[T, Owner]^, other: Box[T, OtherOwner]^
)(...): (Box[T, Owner]^{self}, Box[T, OtherOwner]^{other}) = ...
```

Like `setBox`, `swapBox` returns two new `Box` instances that point to the same `Box` instances passed as `self` and `other`.

Both arguments use the `cap` capturing annotation, which allows the program to pass escape-checked box variables to the function safely. In addition, each returned reference captures its corresponding argument, so the results remain bound to the same scopes as the original references.

4.4.3.4 Moving Boxes

Moving a box means changing its owner.

The main reason to change ownership is when a box separates from or joins other boxes. For example, when an element is removed from a list, its lifetime is no longer tied to the list. If both are represented as boxes, they should then have different owners.

It is important to note that a `Box` instance can change its owner. However, the owner associated with a variable that refers to the box instance is part of the variable's type and does not change.

The following function in imem enables moving a Box:

```
def moveBox[@scinear.HideLinearity T, Owner^, NewOwner^, ...](
  self: Box[T, Owner]^
)(...): Box[T, NewOwner] = ...
```

The `moveBox` function takes the current and the new owner as type parameters and returns a new `Box` instance with the updated owner. As with `setBox`, the `self` parameter captures the universal capability, but the returned box does not. Otherwise, the box instance resulting from the move would be unable to escape the scope of the original reference, which would make it impractical.

A plain move operation can violate the *Direct box uniqueness* goal when the program applies it to a box that is reached through `read` or `write` operations on references. This would result in the moved box instance remaining reachable from available variables after

the move. The operation rules in the borrow checking section (section 4.5.4.2) detail more about this problem. To prevent this situation, imem enforces restrictions on move operations.

By relying on Static Permission Management of Operations, as described in section 4.5, imem doesn't allow moving inside `read` or `write` operations. As a result, when a box instance is reachable through other boxes, imem provides a dedicated dereferencing operation designed specifically for moving, called `derefForMoving`.

imem's interface for dereferencing a box for moving is as follows:

```
def derefForMoving[@scinear.HideLinearity T, ..., @scinear.HideLinearity
    S, ...](
  self: Box[T, Owner]^,
  moveAction: ... ?->{...} T^ ->{...} S
)(...): S =
```

The `derefForMoving` receives a `self: Box`, which is the box that the operation dereferences. The function also receives a `moveAction`, which is a continuation-passing-style argument. The `moveAction` receives the resource, escape-checked, as an argument.

Typically, `moveAction` traverses the given resource to reach other boxes. It then either calls `derefForMoving` on those boxes to explore their resources or calls `moveBox` to set their owners to a new set of lifetimes.

`derefForMoving` only returns the result of `moveAction`. Therefore, the box instance passed as the `self` argument expires after `derefForMoving`, and there is no way to recover it. This behavior is intentional, and the *direct-box-lemma* proves that moving interfaces in the imem memory management implementation preserves the *Direct Box Uniqueness* property.

In imem, moving is a sequence of operations applied to the values referenced by available variables:

- Dereference a box using `derefForMoving`
- Decompose the linear value into its fields
- Move the box to a new owner

Here is an example of moving the inner box of a nested box `Box[Box[Int, lf1], lf1]` to `lf2`:

```
val outerBox = newBox[Box[LinearInt, {lf1}], {lf1}](newBox(LinearInt(42))
    )

val movedInnerBox = derefForMoving[Box[LinearInt, {lf1}], ..., Box[
    LinearInt, {lf2}], ...](
  outerBox,
  inner => moveBox[LinearInt, {lf1}, {lf2}, ...](inner)
)
```

4.4.4 Resource Ownership

Resources can have any type and do not necessarily include a capture-set parameter for owners. In imem, resources, together with the unsafe and internal references that hold them, are owned by the same set of lifetime capabilities that own the direct box of the resource.

Imem enforces this ownership by allowing the program to access the resource only through escape-checked interfaces.

The program can access a resource through the `read`, `write`, and `derefForMoving` interfaces. In each case, the continuation that receives access to the resource has a type of the form `... T^ ->{...} S ...`, which indicates that the resource `T` is escape-checked. As a result, the program can use the resource only within the `writeAction`, `readAction`, or `moveAction` scope and cannot leak it, return it, or store it elsewhere.

Imem can express complex data structures when a box's resource is another `Box`, `ImmutRef`, or `MutRef`. To support this, all imem's interfaces accept an argument with a capturing type that includes the universal capability, such as `self: Box[...]^`, `self: MutRef[...]^`, and `self: ImmutRef[...]^`. In addition, when a provided function may return a `Box`, `MutRef`, or `ImmutRef` that points to the same internal reference as its argument, the return type captures that argument. As a result, the returned value cannot be leaked and is bound to the same scope as the argument.

For example, in `setBox`, the returned type is `Box[T, Owner]^self`. In `swapBox`, both arguments, `self: Box[T, Owner]^` and `other: Box[T, OtherOwner]^`, capture the universal capability and the return types `(Box[T, Owner]^self, Box[T, OtherOwner]^other)` capture each argument respectively.

4.5 Borrow Checking

This section describes how imem manages borrowing and the references that result from it.

4.5.1 Borrow Checking Goals

4.5.1.1 Dereferencing Definition

Dereferencing a box or reference is the implementation counterpart of computing $\sigma(l)$, where l is the location to which the box or reference points. This means that given a box $b : \text{Box}[T, \ldots]$ whose resource is $t : T$, dereferencing the box, $deref(b)$, yields t. That is, $deref(b) = t$. And given a reference $r : \text{Mut}[T, \ldots]$ or $r : \text{Immut}[T, \ldots]$ whose resource is $t : T$, dereferencing the reference, $deref(r)$ would result in t. That is, $deref(r) = t$.

4.5.1.2 Goals

imem performs borrow checking to ensure that the following well-formedness properties are satisfied:

- ***Borrowing Validity:*** The implementation implication of this property is that during execution, for every immutable or mutable reference $r \in \{\text{ImmutRef}[T, O'], \text{MutRef}[T, O']\}$, there is a box $b : \text{Box}[T, O]$ that r is borrowed from, meaning $deref(b) = deref(r)$ and $O \subset O'$.

- ***Box reaching Reference:*** In implementation, this property ensures that if a box $b : \text{Box}[T, O]$ resource reaches a reference $r \in \{\text{ImmutRef}[T', O'], \text{MutRef}[T', O']\}$ resource, in other words $deref(b)$ reaches $deref(r)$, then accessing b by an available variable causes r to become unavailable.

- ***Mutable Reference reaching Immutable Reference:*** The implementation translation of this property means that if a mutable reference $mr : \text{MutRef}[T, O]$ resource reaches an immutable reference $ir \in \text{ImmutRef}[T', O']$ resource, in other words $deref(mr)$ reaches $deref(ir)$, then accessing mr by an available variable causes ir to become unavailable.

- ***Mutable Reference reaching Mutable Reference:*** Similar to its formal definition, in implementation, this property implies that if a mutable reference $m_1 : \text{MutRef}[T, O]$ resource reaches a mutable reference $m_2 : \text{MutRef}[T', O']$ resource, in other words $deref(m_1)$ reaches $deref(m_2)$, then:

- If $deref(m_1) \neq deref(m_2)$: Accessing m_1 by an available variable causes m_2 to become unavailable.

- If $deref(m1) = deref(m2)$: Either accessing m_1 by an available variable causes m_2 to become unavailable, or accessing m_2 by an available variable causes m_1 to become unavailable.

The following two properties are not included in the list of memory well-formedness properties described in the memory management section. However, the imem implementation needs to ensure them due to the addition of moving in the implementation:

- ***Mutable reference structurally constant view:*** If a mutable reference $mr : \text{MutRef}[T, O]$ is available, the structure of the subtree rooted at $deref(mr)$ remains unchanged for as long as mr is available. This means that, for any box $b : \text{Box}[T', O']$ whose resource lies in that subtree, meaning $deref(mr)$ reaches $deref(b)$, the program does not perform `derefForMoving` or `moveBox` on b while mr is available.

- ***Immutable reference constant view:*** This property ensures the ***Immutable Reference not reaching Mutable Reference*** well-formedness property and extends it to account for moving. The property implies that if an immutable reference ir : ImmutRef$[T, O]$ is available, then the entire subtree of $deref(ir)$ remains read-only. This means that:
- There is no available mutable reference mr : MutRef$[T', O']$ whose resource is in that subtree, meaning $deref(ir)$ reaches $deref(mr)$.
- For any box b : Box$[T', O']$ whose resource is in that subtree, $deref(ir)$ reaches $deref(b)$, the program does not perform `setBox`, `swapBox`, `derefForMoving` or `moveBox` on b as long as ir is available.

imem ensures *Borrowing Validity* through Reference Ownership Management. It enforces reaching properties through Box and Reference Holding and Reference Owner Aggregation. It also guarantees the *Mutable reference structurally constant view* and the *Immutable reference constant view* through Static Permission Management.

4.5.2 Reference Ownership Management

References, similar to boxes, are owned by a set of lifetime capabilities.

```
class ImmutRef[T, Owner^](
  private[imem] val tag: InternalRef[T]#Tag,
  private[imem] val internalRef: InternalRef[T]
)

class MutRef[T, Owner^](
  private[imem] val tag: InternalRef[T]#Tag,
  private[imem] val internalRef: InternalRef[T]
) extends scinear.Linear
```

Both the `ImmutRef` and `MutRef` classes include an `Owner^` capture-set type parameter, which the program instantiates with a set of lifetimes.

imem does not provide interfaces for creating references out of thin air; instead, references are borrowed from boxes or from other references.

4.5.2.1 Borrowing a Box

To borrow a box, imem provides the following interfaces:

```
def borrowImmutBox[@scinear.HideLinearity T, Owner^, ..., newOwner^ >:
    {..., Owner}, ...](
  self: Box[T, Owner]^
)(...): (ImmutRef[T, newOwner], ...) = ...
```

```
4
5 def borrowMutBox[@scinear.HideLinearity T, Owner^, ..., newOwner^ >:
    {..., Owner}, ...](
6   self: Box[T, Owner]^
7 )(...): (MutRef[T, newOwner], ...) = ...
```

`borrowImmutBox` and `borrowMutBox` borrow a box immutably and mutably, respectively. Both operations take a `newOwner` type parameter and return a reference whose owner is `newOwner`. Note that the resulting reference does not capture `self`. Therefore, escape-checking `self` has no effect on the returned reference.

For both functions, the owner type parameter `newOwner` of the returned reference must be a superset of the `Owner` type parameter. As a result, owners of the reference include all the lifetime capabilities owners of the box from which the reference is borrowed. By enforcing the constraint `newOwner^ >: ..., Owner`, imem ensures *Borrowing validity* when borrowing a box.

4.5.2.2 Borrowing an Immutable Reference

At any point, multiple immutable references may simultaneously share a resource. To enable this, the program first borrows a box immutably and then reborrows the resulting immutable reference as many times as needed. The following is imem's interface for reborrowing an immutable reference:

```
1 def borrowImmut[@scinear.HideLinearity T, Owner^, ..., newOwner^ >: {...,
    Owner}, ...](
2   self: ImmutRef[T, Owner]^
3 )(...): ImmutRef[T, newOwner] = ...
```

`borrowImmut` for an immutable reference is similar to `borrowImmutBox`. The returned reference does not capture `self`. Therefore, even if `self` is escape-checked, the program can copy the returned reference and store it freely. In addition, the `newOwner` type parameter, with the constraint `newOwner^ >: ..., Owner`, ensures that reborrowed immutable references live no longer than the original immutable reference, in line with the *Borrowing validity* goal.

4.5.2.3 Borrowing a Mutable Reference

imem allows the program to reborrow a mutable reference either mutably or immutably.

4.5.2.3.1 Mutably borrowing a mutable reference `borrowMut` mutably borrows a mutable reference and produces another mutable reference.

```
1 def borrowMut[@scinear.HideLinearity T, Owner^, ..., newOwnerKey,
    newOwner^ >: {..., Owner}, ...](
2   self: MutRef[T, Owner]^
3 )(...): (MutRef[T, newOwner], ...) = ...
```

The signature is identical to that of `borrowMutBox`, and ownership monotonicity is preserved. This means that the owner of the derived reference, `newOwner`, is a superset of the owner of the original reference, `Owner`. As a result, the original reference, and transitively the box, outlives the derived reference.

4.5.2.4 Immutably borrowing a mutable reference

To immutably borrow a mutable reference, imem provides `borrowImmut`:

```
def borrowImmut[@scinear.HideLinearity T, Owner^, ..., newOwner^ >: {..., Owner}, ...](
  self: MutRef[T, Owner]^
)(...): (ImmutRef[T, newOwner], ...)
```

This is the same as `borrowMut`, except that `borrowImmut` returns an immutable reference.

4.5.3 Box and Reference Holding

4.5.3.1 Value Holder

To ensure that `MutRef` and `Box`, or `ImmutRefs`, do not coexist for the same resource at the same time and to follow the Stacked Borrows Model, imem uses a mechanism that ties the existence of one entity to the non-existence of another. In the imem memory management (described in section 4.2.2.2), hold references are responsible for tying reference accesses to the expiration of other references. In the implementation, a combination of `ValueHolder` and `Lifetime` enables this mechanism.

```
class ValueHolder[KeyType, T](private[imem] val value: T) extends scinear.Linear
```

`ValueHolder` is a simple linear class parameterized by `KeyType` and containing a field of type `T` that stores a value of any type. The value can only be accessed through the following function:

```
def unlockHolder[KeyType, @scinear.HideLinearity T](
  key: KeyType,
  holder: ValueHolder[KeyType, T]^
): T^{holder} = holder.value
```

The `unlockHolder` function returns the value only when a key instance that matches the `KeyType` type parameter of the holder is provided. To make the function usable by other imem interfaces, the holder captures the universal capability, and the returned value captures the holder in case the holder is subject to escape checking.

Imem ties a `ValueHolder` to a `Lifetime` by using an internal opaque dependent type as the key. In this way, unlocking the holder always causes the associated lifetime to expire.

```
class Lifetime[OwnersInput^] extends scinear.Linear, caps.Capability:
  type Owners^ = {OwnersInput, this}
  opaque type Key = Object

  def getKey[O^](): Key = Object()
end Lifetime
```

In the Lifetime class, `Key` is an opaque type member. The only way to obtain an instance of this type is through the `getKey` method of `Lifetime` because the type is opaque outside the class. At the same time, because `Key` is a type member, each instance of type `Lifetime` has its own path-dependent `Key` type that is unique to that instance.

As shown below, a `ValueHolder[lf1.Key, ...]` can only be opened by `lf1` and by no other lifetime.

```
val lf1 = Lifetime[{}]()
val lf2 = Lifetime[{}]()

val holder = ValueHolder[lf1.Key, String]("My Secured Resource") //
    locked with ‘lf1‘’s key

val illegalAccess = unlockHolder(lf2.getKey(), holder)
//                                                ^^^^^^
// Found:    (holder : imem.ValueHolder[lf1.Key, String])
// Required: imem.ValueHolder[KeyType, T]^
// where:    KeyType is a type variable with constraint >: lf2.Key
//           T       is a type variable

val okAccess = unlockHolder(lf1.getKey(), holder) // ok
```

4.5.3.2 Borrowing and Holder Results

The `Box` and `MutRef` references are linear types. This means that when the program borrows them, using the borrowing interfaces, the instance passed to the interfaces expires. But the program might need the passed references and imem provides the expired instances in `ValueHolder` tied to the `Lifetime` capability that the newly borrowed reference is associated with.

The following list presents all borrowing functions provided by imem:

```
def borrowImmutBox[@scinear.HideLinearity T, Owner^, ..., newOwnerKey,
    newOwner^ >: {..., Owner}, ...](
  self: Box[T, Owner]^
)(...): (ImmutRef[T, newOwner], ValueHolder[newOwnerKey, Box[T, Owner]^{
    self}]) = ...

def borrowMutBox[@scinear.HideLinearity T, Owner^, ..., newOwnerKey,
    newOwner^ >: {..., Owner}, ...](
  self: Box[T, Owner]^
```

```
)(...): (MutRef[T, newOwner], ValueHolder[newOwnerKey, Box[T, Owner]^{
    self}]) = ...

def borrowMut[@scinear.HideLinearity T, Owner^, ..., newOwnerKey,
    newOwner^ >: {.., Owner}, ...](
  self: MutRef[T, Owner]^
)(...): (MutRef[T, newOwner], ValueHolder[newOwnerKey, MutRef[T, Owner]^{
    self}]) = ...

def borrowImmut[@scinear.HideLinearity T, Owner^, ..., newOwnerKey,
    newOwner^ >: {..., Owner}, ...](
  self: MutRef[T, Owner]^
)(...): (ImmutRef[T, newOwner], ValueHolder[newOwnerKey, MutRef[T, Owner
    ]^{self}])
```

All borrowing functions return a pair. The first element of the pair is the borrowed reference, and the second element is a value holder locked with the `newOwnerKey` type parameter and storing a value of the same type as the `self` parameter. The `T` type parameter of the value holder is instantiated with the type of `self`, which captures `self` in case `self` is subject to escape checking.

The following example demonstrates how a program can borrow a reference from a box and then retrieve the original box:

```
// a new lifetime to borrow `box`
// the owner set of the new lifetime for borrowing the `box` should be a
    superset of the `box` owner set, which is `{lfBox}`
val lfRef = Lifetime[{lfBox}]()

// borrow `box` immutably
val (immutRef, boxHolder) = borrowImmutBox[LinearInt, {lfBox}, ..., lfRef
    .Key, lfRef.Owners, ...](box)

// the program cannot access the `box` here:
// box <--- error: Linear value box is being used twice, or is not
    accessed directly.

// the program has access to `immutRef` here:
immutRef // ok

// unlock `boxHolder` using `lfRef`'s key to retrieve the original `box`
val retrievedBox = unlockHolder(lfRef.getKey(), boxHolder)

// immutRef <--- error: Linear value lfRef is mentioned in the type of
    immutRef but is either not in scope or used already.
retrievedBox // ok, we have the original box back
```

The program borrows a box or a reference using `lf.Owners` as the owners of the borrowed reference and `lf.Key` as the locking key for the holder. It then unlocks the holder by calling `lf.getKey`, which retrieves the original box or reference, causes the lifetime to expire, and consequently makes the borrowed reference unavailable.

4.5.4 Static Permission Management of Operations

4.5.4.1 Using References

The main purpose of a reference is to provide access to a resource after the program borrows it. An immutable reference provides only read access through the `read` function, and a mutable reference provides both read and write access through the `write` function.

```
def read[@scinear.HideLinearity T, Owner^, @scinear.HideLinearity S,
    ...](
  self: ImmutRef[T, Owner]^,
  readAction: ... ?-> T^ ->{...} S
)(...): S =

def write[@scinear.HideLinearity T, Owner^, @scinear.HideLinearity S,
    ...](
  self: MutRef[T, Owner]^,
  writeAction: ... ?->{...} T^ ->{...} S
)(...): S = ...
```

The `read` function takes an immutable reference, `self`, together with a continuation-passing-style argument, `readAction`. Within `readAction`, the program is granted access to the underlying resource. The signature of `readAction` is `... ?-> T^ ->... S`, where the `T^` parameter is the means by which the action accesses the resource. This access is escape-checked because the `T^` parameter captures the universal capability. At the same time, the return type `S` does not capture the argument, which prevents the action from leaking or storing the resource beyond its scope. The `read` function returns the result produced by the action.

The example below illustrates how the program can borrow a box to access its resource:

```
// borrow the `box` immutably
val (immutRef, boxHolder) = borrowImmutBox[LinearInt, {boxLf}, ..., lf.
    Key, lf.Owners, ...](box)

// access the `immutRef` resource, which is `LinearInt`
read[LinearInt, lf.Owners, Unit, ...](
  immutRef,
  resource =>
    // `resource` is escape-checked and can only be used in this scope
    println(resource.value)
)
```

The `write` function follows the same structure as the `read` function. The key difference is that the `self` parameter of `write` is a mutable reference, `MutRef`. Because `MutRef` is linear, the program cannot access the mutable reference after calling the `write` function, nor within the body of `writeAction`. As a result, when the program accesses the resource of a mutable reference through the `write` function, the mutable reference is consumed and cannot be used again.

4.5.4.2 Operation Rules

The `read`, `write`, and `derefForMoving` functions each take a continuation-passing-style argument that provides controlled access to the resource of an immutable reference, a mutable reference, and a box, respectively. Within the scope of these actions, the program must not have access to all interfaces provided by imem.

More precisely, the following restrictions apply.

Inside `write`'s `writeAction`: The program must not be able to call `moveBox` or `derefForMoving`. Allowing these operations could produce references or boxes that point to a moved box, which violates the *Direct Box Uniqueness* property. The example below demonstrates that allowing moving while accessing a box resource can violate well-formedness properties:

```
val outerBox = newBox[Box[LinearInt, {lf1}], {lf1}](newBox(LinearInt(42))
    )

// new lifetime to borrow the `outerBox`
val lfRef = Lifetime[{lf1}]()

// borrow the `outerBox` mutably to access its resource, which is the
    inner box
val (mutRef, boxHolder) = borrowMutBox[Box[LinearInt, {lf1}], {lf1}, ...,
     lfRef.Key, lfRef.Owners, ...](outerBox)

// accessing the inner box
val movedInnerBox = write[Box[LinearInt, {lf1}], lfRef.Owners, Box[
    LinearInt, {lf2}], ...](
  mutRef,
  innerBox =>
    // moving the inner box and returning it
    moveBox[LinearInt, {lf1}, {lf2}, ...](innerBox) // !
)

// unlock the `boxHolder` using `lfRef`'s key to retrieve the original `
    outerBox`
val retrievedOuterBox = unlockHolder(lfRef.getKey(), boxHolder)

movedInnerBox // ok to access
retrievedOuterBox // ok to access
// Both `movedInnerBox` and resource of the `retrievedOuterBox`, which is
     the inner box, point to the same underlying, the same `LinearInt`
    instance.
```

Inside `read`'s `readAction`: As with `writeAction`, the program must not be able to call `moveBox` or `derefForMoving`. In addition, the program must not be able to call `setBox` or `swapBox`, because these operations would modify a box while it is reachable through an immutable reference, which violates the *Immutable reference constant view* goal. For the same reason, the program must not be able to call `borrowMutBox`, `borrowMut`,

or `write`, since these operations could create a mutable reference to a resource that is already reachable through an immutable reference. The following example demonstrates that permitting mutation and mutable borrowing in `readAction` can lead to non-well-formed memory:

```
val outerBox = newBox[Box[LinearInt, {lf1}], {lf1}](newBox(LinearInt(42))
    )

// new lifetime to borrow the `outerBox`
val lfRef = Lifetime[{lf1}]()

// borrow the `outerBox` immutably to access its resource
val (immutRef, _) = borrowImmutBox[Box[LinearInt, {lf1}], {lf1}, ...,
    lfRef.Key, lfRef.Owners, ...](outerBox)

// accessing the resource of `immutRef`, which is the inner box
read[Box[LinearInt, {lf1}], lfRef.Owners, Unit, ...](
  immutRef,
  innerBox =>
    // modify the inner box's resource while it is reachable through an
    immutable reference
    setBox[LinearInt, {lf1}, ...](innerBox, LinearInt(99)) // ! violates
    *Immutable reference constant view*
)

// accessing the resource of `immutRef`, which is the inner box
val returnedMutRef = read[Box[LinearInt, {lf1}], lfRef.Owners, MutRef[
    LinearInt, lfRef.Owners], ...](
  immutRef,
  innerBox =>
    // borrow the `innerBox` mutably
    val (innerMutRef, _) = borrowMutBox[LinearInt, {lf1}, ..., lfRef.Key,
    lfRef.Owners, ...](innerBox) // !

    // return the mutable reference
    innerMutRef
)

immutRef // ok to access
returnedMutRef // ok to access
// Both `immutRef` and `returnedMutRef` are now available in the same
    scope.
// The immutable reference `immutRef` reaches the same `LinearInt`
    instance that `returnedMutRef` can reach and mutate.
// This violates the *Immutable Reference not reaching Mutable Reference*
    property.
```

Table 4.1 summarizes the scopes and the actions permitted within them:

Access type	Scopes that have it	Enables
read	default scope, `moveAction`, `writeAction`, `readAction`	`borrowImmutBox`, `borrowImmut`, `read`
write	default scope, `moveAction`, `writeAction`	`setBox`, `swapBox`, `borrowMutBox`, `borrowMut`, `write`
move	default scope, `moveAction`	`moveBox`, `derefForMoving`

Table 4.1: Access types, scopes, and enabled actions.

4.5.4.3 Write and Move capabilities

Imem uses the access control use case of capture checking to enforce the operation rules described above.

Imem defines two capabilities:

- `WriteCap`: Any function that captures this capability in its type is allowed to call imem interfaces that require write access.
- `MoveCap`: Any function that captures this capability in its type is allowed to call imem interfaces that require move access.

Because imem permits read access in every scope, imem does not define a separate capability for it.

These capabilities are propagated through the program using an implicit context parameter that every imem interface requires. The following shows the `Context` class definition:

```
class Context[WC^, MC^] private[imem] ()
```

The `Context` is a simple class with two type parameters, both of which are capture sets:

- `WC^`: imem instantiates it with the write capability.
- `MC^`: imem instantiates it with the move capability.

A program that uses imem must pass its entry point to the `withImem` function:

```
def withImem[T](block: [@caps.use WC^, MC^] => Context[WC, MC]^ => T): T
    =
  object WC extends caps.Capability
  object MC extends caps.Capability
  val ctx = Context[{WC}, {MC}]()
  block[{WC}, {MC}](ctx)
```

This function defines the capabilities, instantiates a context, and calls the program entry point. Each imem interface receives a `Context` instance as an implicit argument:

```
// require read access:
def borrowImmutBox[..., WC^, MC^](...)(using ctx: Context[WC, MC]^{...}): ... = ...
def borrowImmut[..., WC^, MC^](...)(using ctx: Context[WC, MC]^{...}): ... = ...
def borrowImmut[..., WC^, MC^](...)(using ctx: Context[WC, MC]^{...}): ... = ...
def read[..., WC^, MC^](...)(using ctx: Context[WC, MC]^{...}): ... = ...

// require write access:
def borrowMutBox[..., @caps.use WC^, MC^](...)(using ctx: Context[WC, MC]^{...}): ... = ...
def borrowMut[..., @caps.use WC^, MC^](...)(using ctx: Context[WC, MC]^{...}): ... = ...
def write[..., @caps.use WC^, MC^](...)(using ctx: Context[WC, MC]^{...}): ... = ...
def setBox[..., @caps.use WC^, MC^](...)(using ctx: Context[WC, MC]^{...}): ... = ...
def swapBox[..., @caps.use WC^, MC^](...)(using ctx: Context[WC, MC]^{...}): ... = ...

// require move access:
def derefForMoving[..., WC^, @caps.use MC^](...)(using ctx: Context[WC, MC]^{...}): ... = ...
def moveBox[..., WC^, @caps.use MC^](...)(using ctx: Context[WC, MC]^{...}): ... = ...
```

All functions have the capture set type parameters `WC^` and `MC^`, as well as an implicit parameter `ctx: Context[WC, MC]`. Functions that require write access use the write capability `WC^`, which the compiler recognizes through the `@caps.use` annotation. Similarly, functions that require move access use the move capability `MC^`.

To distinguish which scopes have write or move access and which do not, imem annotates each continuation-passing-style argument with the exact capabilities it is allowed to use:

```
def read[... T, Owner^, ... S, ctxOwner^, WC^, MC^](
  ...,
  readAction: Context[WC, MC]^{...} ?->{} T^ ->{Owner, ctxOwner} S
  //                                  ^                          ^ does not mention ‘WC‘ nor ‘MC‘.
)(...): S = ...

def write[... T, Owner^, ... S, ctxOwner^, @caps.use WC^, MC^](
  ...,
  writeAction:
    Context[{WC}, {MC}]^{...} ?->{WC} T^ ->{Owner, ctxOwner, WC} S
  //                            ^                              ^ mentions only ‘WC‘.
)(...): S = ...

```

```
def derefForMoving[... T, Owner^, ctxOwner^, ... S, WC^, @caps.use MC^](
  ...,
  moveAction: Context[WC, MC]^{...} ?->{WC, MC} T^ ->{Owner, ctxOwner, WC
    , MC} S
  //                                        ^   ^                        ^
     ^ mentions both `WC` and `MC`.
)(...): S = ...
```

Each action is annotated with the set of access capabilities it may use. In this way, as described in access control, the program can, in each scope, call only the interfaces that imem's annotations permit for that scope. The codeblock below shows that the capture checking results in an error if the program uses `setBox` inside `readAction`:

```
// access the outer box's resource, the inner box, through a read
   operation on a immutable reference borrowed from the outer box
read[Box[LinearInt, {lf}], lfRef.Owners, Unit, ..., WC, MC](
  immutRef,
  innerBoxResource => setBox(innerBoxResource, LinearInt(42))
//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
// Found:    (contextual$1: imem.Context[WC, MC]^?) ?->{WC}
//   imem.Box[scinear.utils.LinearInt^?, {lf}]^? ->{contextual$1, WC}
   Unit
// Required: (imem.Context[WC, MC]^{lf, ctx, lfRef}) ?->{lf, ctx, lfRef}
//   imem.Box[scinear.utils.LinearInt, {lf}]^ ->{lf, ctx, lfRef} Unit
// ...
//==> subcaptures {WC} <:< {O1, ctx, lfRef} in open varState
//  ==> {O1, ctx, lfRef} accountsFor WC  = false
//  ==> {O1, ctx, lfRef} accountsFor cap  = false
// ...
)
```

In the error message above, the compiler reports an error because the capability `WC` is not included in `O1, ctx, lfRef`, which is the capture set of the function.

4.5.5 Reference Owner Aggregation

imem implements Reference Owner Aggregation to ensure that program memory follows the Stacked Borrows Model. In other words, it ensures that the reaching properties are not violated.

4.5.5.1 Box and Reference Holding Insufficience

By implementing Box and Reference Holding, imem ensures that, for a given value, all boxes and references that directly point to this value follow the Stacked Borrows Model. As a result, accessing any of these boxes or references expires the references that appear above it on the borrow stack.

However, this mechanism is not sufficient when the program contains nested boxes and references that point to different boxes, where one box reaches another. In the following example, accessing the outer box through its holder should expire the mutable reference to the inner box; otherwise, the *Stacked Borrows* invariant is violated.

```
// new lifetime to borrow `outerBox` mutably
val lfOuter = Lifetime[lfBox.Owners]()

// borrow `outerBox` mutably
val (outerMut, outerHolder) = borrowMutBox[Box[LinearInt, {lfBox}], {lfBox}, ..., lfOuter.Key, lfOuter.Owners, {WC}, {MC}](outerBox)

// new lifetime to borrow the inner box mutably
val lfInner = Lifetime[lfBox.Owners]()

// access resource of `outerMut`, which is the inner box, and borrow it mutably
val innerMut = write[Box[LinearInt, {lfBox}], lfOuter.Owners, MutRef[LinearInt, lfInner.Owners], ..., {WC}, {MC}](
  outerMut,
  innerBox =>
    // borrowing the inner box mutably
    borrowMutBox[LinearInt, {ctx}, ..., lfInner.Key, lfInner.Owners, {WC}, {MC}](innerBox)._1 // ?
)

// unlock the `outerHolder` to retrieve the `outerBox` again
val retrievedOuterBox = unlockHolder(lfOuter.getKey(), outerHolder)

innerMut // can the program access `innerMut` here?
```

In the example above, if the code compiles without errors, then the program can access the inner box resource, namely the `LinearInt` instance, both through the borrowed `retrievedOuterBox` and by performing a `write` operation on `innerMut`. This behavior violates the *Stacked Borrows* properties.

4.5.5.2 Owner Aggregation

To address this issue, imem ensures that when a reference is borrowed from another reference, either directly or indirectly, the lifetime set of the derived reference is a superset of the lifetime set of the original reference. In the example above, the owners of `outerMut1` should therefore be a subset of the owners of the `innerMut` reference.

More precisely, suppose the program borrows $r_O \in \{\text{Box}[T, O], \text{ImutRef}[T, O], \text{MutRef}[T, O]\}$ using one of imem's borrowing interfaces. Assume that the borrowing point is reached through a sequence of references $r_1, r_2, \ldots, r_n$, meaning that the borrow occurs within nested action scopes of `write` or `read` operations on these references.
Here, $r_i \in \{\text{Box}[T, O_i], \text{ImutRef}[T, O_i], \text{MutRef}[T, O_i]\}$. Then, the resulting reference $r_d \in$

$\{\text{MutRef}[T, O']\}$ must have an owner set that is a superset of the owner sets of all involved references, and r_O. Formally:

$$O' \supseteq O \cup \bigcup_{i=1}^{n} O_i \tag{4.97}$$

As a consequence, in the corrected version of the previous example shown below, accessing the outer box causes the mutable reference to the inner box to expire.

```
...
// new lifetime to borrow the inner box mutably
val lfInner = Lifetime[{lfBox, lfOuter}]() // the reference owner aggregation
                                           // requires that the owner set of
                                           // the inner reference be a superset
                                           // of the owner set of ‘outerMut‘,
                                           // namely ‘{lfBox, lfOuter}‘.
...
innerMut // error: Linear value lfOuter is mentioned in the type of innerMut but is either not in scope or used already.
```

4.4.5.3 Context Owner Aggregation

In imem, the capture set of `Context` aggregates reference owners. Each imem interface receives an implicit context parameter, whose capture set represents the lifetimes accumulated throughout the program.

To do this aggeregation, within the action scopes of `read` and `write` operations, imem extends the context capture set to include the owners of the accessed references.

```
def read[... T, Owner^, ... S, ctxOwner^, ...](
  self: ImmutRef[T, Owner]^,
  readAction: Context[...]^{ctxOwner, Owner} ?-> T^ ->{...} S
  //                        ^^^^^^^^^^^^^^^^^
)(
  using ctx: Context[...]^{ctxOwner}
): S = ...

def write[... T, Owner^, ... S, ctxOwner^, ...](
  self: MutRef[T, Owner]^,
  writeAction: Context[...]^{ctxOwner, Owner} ?->{...} T^ ->{...} S
//                          ^^^^^^^^^^^^^^^^^
)(
  using ctx: Context[...]^{ctxOwner}
): S = ...
```

In both the `read` and `write` functions, the capture set of the context parameter is `ctxOwner^` and the context passed to the action captures `ctxOwner, Owner`, which extends the capture set with the owners of the accessed reference.

Furthermore, the owners of references should include these accumulated owners. To do this, in the borrowing functions, the owners of the newly created reference, `newOwner`, must form a superset of both the original reference or box owner, `Owner`, and the aggregated owners, `ctxOwner`. In other words, the constraint is `newOwner^ >: ctxOwner, Owner`.

```
def borrowImmutBox[... T, Owner^, ctxOwner^, newOwnerKey, newOwner^ >: {ctxOwner, Owner}, ...](
  //
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  self: Box[T, Owner]^
)(
  using ctx: Context[...]^{ctxOwner}
): (ImmutRef[T, newOwner], ...) = ...

def borrowMutBox[... T, Owner^, ctxOwner^, newOwnerKey, newOwner^ >: {ctxOwner, Owner}, ...](
  //
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  self: Box[T, Owner]^
)(
  using ctx: Context[...]^{ctxOwner}
): (MutRef[T, newOwner], ...) = ...

def borrowMut[... T, Owner^, ctxOwner^, newOwnerKey, newOwner^ >: {ctxOwner, Owner}, ...](
  //
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  self: MutRef[T, Owner]^
)(
  using ctx: Context[...]^{ctxOwner}
): (MutRef[T, newOwner], ...) = ...

def borrowImmut[... T, Owner^, ctxOwner^, newOwnerKey, newOwner^ >: {ctxOwner, Owner}, ...](
  //
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  self: MutRef[T, Owner]^
)(
  using ctx: Context[...]^{ctxOwner}
): (ImmutRef[T, newOwner], ...) = ...

def borrowImmut[... T, Owner^, ctxOwner^, newOwnerKey, newOwner^ >: {ctxOwner, Owner}, ...](
  //
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  self: ImmutRef[T, Owner]^
)(
```

```
  using ctx: Context[...]^{ctxOwner}
): ImmutRef[T, newOwner] = ...
```

Based on these two features, the context carries the union of the owners of all references used to reach a given program point, and the owners of any newly created reference are a superset of this aggregated set.

4.5.6 Object Graph

Figure 4.8 illustrates the object graph described in section 4.3.4, extended by introducing value holders as linear values and enforcing ownership and borrow-checking rules.

4.6 Soundness

This section explains why imem's implementation ensures that the part of program memory managed by imem follows the well-formedness properties defined in the memory management section (section 4.2). It then discusses loopholes and unwanted API uses, whether they break imem's guarantees, and the guidelines for avoiding those that do.

4.6.1 Satisfying Well-formedness Properties

The following lists the memory well-formedness properties in imem and explains how the implementation satisfies each property:

- **Direct Box Uniqueness:** The imem implementation of ownership and Scinear linearity rules follows the theoretical definitions of ownership and linearity described in section 4.2.2.2. The only difference appears in the case of moving, where the *direct-box lemma* proves the preservation of *Direct Box Uniqueness*.
- **No cyclic box:** The linearity of the `Box` class ensures that a program has only one access to a box instance. In addition, borrow checking guarantees that the program can access either a mutable reference to a box or the box itself, but not both at the same time. Together with the fact that boxes are the only source of mutability in imem, these properties ensure that the program cannot use `setBox` to assign a box's value to itself or to a reference that reaches the same box. As a result, the implementation satisfies *No cyclic box*.
- **No dangling references:** The imem implementation does not fully satisfy the theoretical definition of this property. For example, if one `Box` instance reaches another `Box`, the capture set type argument of the reaching box is not always a superset of that of the reached box. The *dangling-box-unavailability lemma* shows

that, although defining dangling boxes is possible, such boxes become unavailable once the box they reach expires.

- **Borrowing Validity:** The borrowing interfaces align with the borrowing operations. They satisfy the property by ensuring that a reference resulting from borrowing does not outlive the original box.

- **Reaching Properties**: The borrow-checking goals sub-section (section 4.5.1.2) explains in detail how the implementation satisfies the reaching properties, namely *Box reaching Reference*, *Mutable Reference reaching Immutable Reference*, *Mutable Reference reaching Mutable Reference*, and *Immutable Reference not reaching Mutable Reference*. The implementation enforces these properties through Box and Reference Holding, Reference Owner Aggregation, and Static Permission Management.

4.6.1.1 Supporting Lemmas

4.6.1.1.1 Implementation Definition of Availability In the imem implementation, the definition of availability differs slightly from the definition in the memory management section (section 4.2.2). The implementation defines availability based on the entire box or reference type, rather than only on its direct owner set.

Box and Reference Availability At each point during program execution, a box or a reference is available if its type does not include any type parameter, direct or nested, instantiated with a capture set that contains an expired linear capability.

Variable Availability: A variable is available in the program scope at a point during execution if referring to the variable in an expression does not at that point produce a compiler error. In the imem context, this means:

- The variable is defined in one of the enclosing scopes.
- If the variable has a linear type, it has not expired.
- If the declared type of the variable includes type parameters instantiated with capture sets, any linear capability in those capture sets has not expired.

4.6.1.1.2 Lemmas ***dangling-box-unavailability-lemma:*** If a value $v : V$ reaches a box instance $b' : \text{Box}[T', Owner']$ and a lifetime capability $lf \in Owner'$ expires, then v becomes unavailable.

Proof. Let $v : V$ be a value that reaches a box instance $b' : \text{Box}[T', Owner']$. The value v can be either a box $b : \text{Box}[T, Owner]$ or a linear value $l : L$. If a lifetime capability lf is in $Owner'$, $lf \in Owner'$, the following proves that lf occurs in at least one capture set instantiation of a type parameter of V:

If a box b reaches another box b', then either $b = b'$, or $deref(b)$ reaches b'. If a linear value l reaches a box b', then at least one field value $f : F$ reaches b'.

The proof uses induction on the number of reachability operations needed to obtain b' from v. Let $n \geq 0$ be such that there exists a sequence $Op_1, \ldots, Op_n$ with $Op_1 \circ \cdots \circ Op_n(v) = b'$.

Induction statement for n operations: If $v : V$ reaches $b' : \text{Box}[T', Owner']$ using n operations and $lf \in Owner'$, then lf occurs in at least one capture-set instantiation of a type parameter of V.

Base case ($n = 0$): With zero operations, $v = b'$. Therefore, V is $\text{Box}[T', Owner']$, and $lf \in Owner'$ appears in the instantiation of a type parameter of V.

Inductive step ($n \geq 1$): The inductive step depends on the first operation from v. In each case, the induction-hypothesis occurrence of lf in the type of the intermediate value is propagated back to a type-parameter instantiation of V.

- If v is a box $b : \text{Box}[T, Owner]$, then $deref(b) : T$ reaches b'. Based on the induction hypothesis, there exists a type parameter p of T that is instantiated with a capture set containing lf. Since T is a type parameter of $\text{Box}[T, Owner]$, the instantiation of T includes the instantiation of p, and therefore includes lf.
- If v is a linear value $l : L$, then some field value $f : F$ reaches b'. By the induction hypothesis there exists a type parameter p of F that is instantiated with a capture set containing lf. Each type parameter of F is instantiated either with types defined in L or with types built from type parameters of L. Since linear classes do not allow nested type definitions, there exists a type parameter of L that p is instantiated with, and this instantiation contains a capture set that includes lf.

By induction, if a value $v : V$ reaches a box instance $b' : \text{Box}[T', Owner']$ and $Owner'$ includes a lifetime lf, then the capability lf appears in a capture-set instantiation of a type parameter of V. Therefore, if lf expires, the value $v : V$ becomes unavailable, proving the lemma.

direct-box-lemma: Assume that every resource has exactly one direct box. After a `moveBox` or `derefForMoving` operation, every resource still has exactly one direct box.

Proof. The proof consists of two parts, depending on the operation.

`moveBox`**:** Assume that the program applies `moveBox` to a box $b : \text{Box}[T, Owner]$. Since `Box` is a linear type, the program cannot reach b after b is passed to `moveBox`. The operation returns a box $b' : \text{Box}[T, Owner']$, which is a direct box for the resource of b. Before the operation, b is the only reachable direct box for that resource. After the operation, b is unreachable and b' is reachable. Therefore, the resource has exactly one reachable direct box after the operation.

`derefForMoving`**:** Assume that the program applies `derefForMoving` to a box $b : \text{Box}[T, Owner]$. Since `Box` is a linear type, the program cannot reach b after it is passed

to `derefForMoving`, and the program also cannot reach b during the `moveAction` continuation.

During `moveAction`, the value $t = deref(b)$ is passed to the continuation as an escape-checked argument. The value t is not a resource because there is no available reachable box pointing to it. Moreover, the only box that provides access to t is b, and b is inaccessible inside `moveAction`. Also, because the `newBox` function requires a resource argument with an empty capture set, the program, inside `moveAction`, cannot create a new box that turns t into a resource again.

After `moveAction`, since t was an escape-checked argument in `moveAction`, the program can only access t after `moveAction` if `moveAction` returns t. In that case, t is still not a resource, and the statement of the lemma continues to hold.

4.6.2 Possible Loopholes and Guidelines

The following subsections list possible loopholes or unwanted cases and discuss whether a valid program can exhibit them and, if so, guidelines that would prevent reaching any memory state that is not well-formed.

4.6.2.1 General

In general, any use of `asInstanceOf` breaks static type safety and, as a result, all guarantees imem provides. Similarly, programs that include reflection can manipulate types at runtime, which breaks all imem's static enforcements. For these features, the guideline is to either avoid their use in an imem program or use them with full awareness of their effects on imem. imem assumes that programs do not use these features.

4.6.2.2 Linearity

imem bases its ownership and borrow checking on Scinear linearity rules. As a result, any program that does not follow these rules breaks the guarantees provided by imem. This includes programs that upcast linear types to non-linear types, such as `Object`, or that use `@HideLinearity`. Such programs may lead to incorrect memory states and, transitively, to unsafe behavior.

4.6.2.3 Mutability

imem assumes that boxes are the only source of mutation. In Scala, a class field or a variable can be mutable when it is defined using `var`. Because imem is not a compiler plugin and does not parse the program that uses it, it cannot determine whether `var`s are present.

However, imem aims to provide safe mutability through static rules enforcing the Stacked Borrows Model. The only mutations that imem manages are calls to the `swapBox` and `setBox` functions on imem boxes. Other sources of mutation are outside the control of imem, and when a program includes them, imem cannot guarantee their safety.

4.6.2.4 Escape Checking

imem uses escape-checking to prevent resources from leaking outside their intended scope. As discussed in section 2.5.5.1, for escape-checking to propagate from a class instance to its fields, each field must either be a capability or have a type that captures `this`. Otherwise, when an instance is subject to escape-checking, its fields are not bound to the same scope as the instance.

This limitation is particularly important when the resource in a box is a linear class with multiple fields. This is illustrated by the following example:

```
class LeakyPair(_first: LinearInt, _second: LinearInt) extends scinear.
    Linear:
  val first: LinearInt = _first // leaky because does not capture `{this
    }`
  val second: LinearInt = _second // leaky because does not capture `{
    this}`
end LeakyPair

val lf = Lifetime[{ctx}]()
val box: Box[LeakyPair, {lf}] = newBox(LeakyPair(LinearInt(1), LinearInt
    (2)))
val (immutRef, boxHolder) = borrowImmutBox[LeakyPair, {lf}, {ctx}, lf.Key
    , lf.Owners, WC, MC](box)

val leakedField = read[LeakyPair, lf.Owners, LinearInt, {ctx}, WC, MC](
  immutRef,
  resource =>
    resource.first // ! leaking the field of the resource of the box
)
```

If the class definition does not follow the escape-checking propagation rules, a program may leak the class fields. imem assumes that every linear class that is a resource in a box satisfies these requirements.

4.6.2.5 imem Components

There are several possible workarounds related to imem's components. imem addresses some of these by special features in the imem implementation that are specific to each loophole. At the same time, there are other workarounds that imem does not resolve and are instead left to guidelines. The following section lists these cases.

4.6.2.5.1 Internal Components

imem doesn't allow the program to have access to the internal components by:

- Making `UnsafeRef` and `InternalRef` private to the imem package.
- Marking all fields of `Box`, `ImmutRef` and `MutRef` class as private.

In this way, the program is not able to neither import internal components, instantiate them, nor access them through publicly accessible classes in imem.

4.6.2.5.2 Lifetime

In imem, a Lifetime represents a static, continuous region of a program. This region starts at the lifetime definition and ends at the point where the program uses it. Therefore, a program that uses imem should not pass a lifetime around, except to `unlock`, nor return or copy it. All such operations are a use of the lifetime, which causes every `Box`, `ValueHolder`, `ImmutRef`, and `MutRef` associated with that lifetime to expire at the point of use.

Another consideration is that restricting lifetime usage to a continuation-passing-style argument, similar to how resources are accessed in the `read` and `write` functions, would simplify the model. In this approach, the continuous part of a program would correspond to a scope, which is more convenient for static rule enforcement. However, this restriction would imply that, when a reference includes two lifetimes, one lifetime must start before the other and end after it, creating an indirect dependency between the two lifetimes. Such a constraint would make imem less expressive and would prevent the use of intermediate references between two lifetimes without forcing one lifetime to dominate the other.

Finally, the lifetime `Key` type is an opaque path-dependent type. This means that the only way to obtain a key instance for a lifetime, for example `lf`, is by calling `lf.getKey()`. Calling `lf.getKey()` is a use of the linear capability `lf`, which causes all references that include `lf` in their lifetime set to expire. Therefore, it is not possible to leak or pass around a lifetime key while the lifetime remains unexpired.

4.6.2.5.3 Box Resource

The `Box` class constructor is private, which means that a program can create a box only through the imem interfaces. These interfaces take a resource as input and return a box that points to that resource. imem assumes that the only Scala reference to the resource is the one passed to the box constructor. In other words, the program does not store the Scala reference of the resource, or any object that can reach that resource, elsewhere in memory. In the following example, the program creates two boxes that refer to the same resource:

```
1 class LinearArray(_first: Array[Int]) extends scinear.Linear:
2   val first: Array[Int]^{this} = _first
3 end LinearArray
```

```
4
5 val lf = Lifetime[{ctx}]
6 val arr: Array[Int] = Array(1, 2, 3)
7
8 val arrBox1: Box[LinearArray, {lf}] = newBox(LinearArray(arr))
9 val arrBox2: Box[LinearArray, {lf}] = newBox(LinearArray(arr))
10 //! both resource of the ‘arrBox1‘ and ‘arrBox2‘ point to the same array
```

imem controls only access to a resource in a box, not the resource reaching memory. This means that the rest of the memory that the resource reaches is not controlled by imem, unless that memory is itself another box or a box's resource. Therefore, if a program creates multiple resources that point to the same underlying object via Scala references, imem regulates access to those resources, not to the object they point to. Managing access to the shared object itself remains the program's responsibility.

It is important to note that imem's static guarantees no longer apply when a program reaches the non-linear part of memory. A resource in a box must be a linear class, and access to linear instances is statically controlled according to Scinear rules. If the fields of the resource class are also linear, Scinear statically manages access to the memory associated with those fields as well, and this property propagates transitively. For this reason, imem recommends keeping memory linear as much as possible.

One important guideline for linear memory is to replace direct linear fields with box fields that point to linear classes. For example, in the following `LinearBinTree` class, the fields have type `LinearBinTree`:

```
1 class LinearBinTree(
2   _left: Option[LinearBinTree],
3   _right: Option[LinearBinTree]
4 ) extends scinear.Linear:
5   val left: Option[LinearBinTree]^{this} = _left
6   val right: Option[LinearBinTree]^{this} = _right
7 end LinearBinTree
```

According to imem's guarantees, this implementation is correct. However, when the program has access to the children of a `LinearBinTree` instance, it cannot borrow them immutably or mutably. It also cannot reference both of a node's children simultaneously or iterate over the tree without consuming the tree and destructing and reconstructing it during the traversal.

```
1 class LinearBinTree(
2   _left: Option[LinearBinTree],
3   _right: Option[LinearBinTree]
4 ) extends scinear.Linear:
5   val left: Option[LinearBinTree]^{this} = _left
6   val right: Option[LinearBinTree]^{this} = _right
7 end LinearBinTree
8
9 // a three node binary tree
10 val leftLeaf = LinearBinTree(None, None)
```

```
val rightLeaf = LinearBinTree(None, None)
val rootNode = LinearBinTree(Some(leftLeaf), Some(rightLeaf))

val lf = Lifetime[{ctx}]()
val treeBox = newBox[LinearBinTree, {lf}](rootNode)

val (treeRef, treeHolder) = borrowImmutBox[LinearBinTree, {lf}, {ctx}, lf
    .Key, lf.Owners, WC, MC](treeBox)

read[LinearBinTree, lf.Owners, Unit, {ctx}, WC, MC](
  treeRef,
  root =>
    root.left // ! Accessing this node is only possible
              // ! in here, the program cannot have a
              // ! reference to this node.
    ()
)
```

The following example shows that when a `LinearBinTree` class refers to its children through `Box` instances, it is possible to hold an immutable reference to both children at the same time.

```
class LinearBinTree[O^](
  _left: Box[Option[LinearBinTree[O]], O],
  _right: Box[Option[LinearBinTree[O]], O]
) extends scinear.Linear:
  val left: Box[Option[LinearBinTree[O]], O]^{this} = _left
  val right: Box[Option[LinearBinTree[O]], O]^{this} = _right
end LinearBinTree

val lf = Lifetime[{ctx}]()

val leftLeaf = LinearBinTree[{lf}](newBox(None), newBox(None))
val rightLeaf = LinearBinTree[{lf}](newBox(None), newBox(None))
val rootNode = LinearBinTree[{lf}](
  newBox(Some(leftLeaf)),
  newBox(Some(rightLeaf))
)
val treeBox = newBox[LinearBinTree[{lf}], {lf}](rootNode)

val lfRef = Lifetime[{ctx}]()
val (rootImmutRef, rootHolder) = borrowImmutBox[LinearBinTree[{lf}], {lf
    }, {ctx}, lfRef.Key, lfRef.Owners, {WC}, {MC}](treeBox)
val leftImmutRef = read[LinearBinTree[{lf}], lfRef.Owners, ImmutRef[
    Option[LinearBinTree[{lf}]], {lf, ctx}], {ctx}, {WC}, {MC}](
  rootImmutRef,
  ctx ?=> root =>
    val (leftImmutRef, leftHolder) = borrowImmutBox[Option[LinearBinTree
    [{lf}]], {lf}, {ctx}, NeverUsableKey, {ctx}, {WC}, {MC}](root.left)
    leftHolder.consume()
    leftImmutRef
)
```

```
28
29 val rightImmutRef = read[LinearBinTree[{lf}], lfRef.Owners, ImmutRef[
       Option[LinearBinTree[{lf}]], {lf, ctx}], {ctx}, {WC}, {MC}](
30   rootImmutRef,
31   ctx ?=> root =>
32     val (rightImmutRef, rightHolder) = borrowImmutBox[Option[
       LinearBinTree[{lf}]], {lf}, {ctx}, NeverUsableKey, {ctx}, {WC}, {MC
       }](root.right)
33     rightHolder.consume()
34     rightImmutRef
35 )
36
37 leftImmutRef // the program has a reference to the left child of the root
38 rightImmutRef // the program has a reference to the right child of the
       root
39 rootImmutRef // the program has a reference to the root
```

Owner set

In imem, the `Box` class has a capture set type parameter `Owner^`, which the program instantiates with the set of lifetimes that should outlive a box instance. imem does not statically enforce that all capabilities in this capture set are lifetime capabilities, because current Scala features do not allow such enforcement.

Although it does not seem convenient to include a capability that is not a `Lifetime` instance in the owner set of a `Box`, `ImmutRef`, or `MutRef`, this does not hurt imem's soundness. As a capture set becomes larger, capture checking applies more restrictions, which preserves the soundness of imem.

4.6.2.6 Borrow Checking

4.6.2.6.1 Borrowing New Owners When the program borrows a box or a reference, either immutably or mutably, through any imem interface, the function requires a new owner capture set type parameter, `newOwner^`, and the corresponding lifetime key, `newOwnerKey`, for that owner set. imem assumes that the program instantiates these parameters using one of the following two ways:

- **A lifetime instance with its `Owners` and `Key` type members:** This option enables the program to get back the source of the borrowing. In this case, the program can unlock the value holder only by using an instance of the associated lifetime key, which causes the lifetime to expire.
- `NeverUsableKey` **with any lifetime set:** This option is useful when the program no longer needs the source of the borrow and only requires the borrowed reference for use. Because `NeverUsableKey` is an opaque type, the program cannot create instances of it, and therefore cannot unlock the value holder.

There is no static enforcement that restricts valid programs to these two cases. This means that a program can misuse these parameters and break imem's guarantees:

```
val lf = Lifetime[{ctx}]()
val box: Box[LinearInt, {lf}] = newBox(LinearInt(42))

val (immutRef, holder) = borrowImmutBox[LinearInt, {lf}, {ctx}, Object,
    lf.Owners, WC, MC](box) // ! borrowing using `Object` as the key type

// unlocking the holder using an instance of `Object`
val retrievedBox = unlockHolder(Object(), holder)

immutRef // ! the program has access to the reference
retrievedBox // ! the program has access to the box as well
```

This constraint is not enforced statically because there is no way to use a path as a type parameter and require other type parameters to be path-dependent type members of that parameter. In addition, since the `Key` type member of `Lifetime` is a path-dependent opaque type, it is not possible to place type bounds on type parameters that may be instantiated with it. The future works (described in section 7.2) include an approach that would make misuse of these interfaces more explicit.

4.6.2.6.2 Context Write and Move Capabilities

The `Context` constructor is private, and `withImem` is the only imem interface that provides a `Context` instance to a program. Because the `WC^` and `MC^` type parameters of `Context`, which represent write and move capabilities, are neither covariant nor contravariant, a program cannot pass around the `Context` instance provided to it as a `Context` with different capture sets for these parameters. As a result, the write and move capability sets remain consistent and identical throughout the entire program.

Owner Aggregation

Context owner aggregation uses the capture-set annotation of the `Context` capturing type, `Context[WC, MC]^ctxOwner`, to represent the set of accumulated owners that are in scope. Unlike `WC^` and `MC^`, the `ctxOwner^` capture set is not a type parameter of `Context`, and therefore it is covariant. As a result, a program can assign a value of type `Context[WC, MC]^A` to a value of type `Context[WC, MC]^B` when `A <: B`.

This behavior is safe as long as the new capture set still includes the relevant lifetime capabilities. However, because the universal capability `cap` is a superset of all capture sets, a program can assign a context of type `Context[WC, MC]^ctxOwner` to one of type `Context[WC, MC]^cap`. In this case, lifetime expiration no longer limits the context and the references borrowed using that context, which breaks imem rules.

imem assumes that programs do not alter the accumulated owners of a context. Due to heavy limitations on types capturing the universal capability, `cap`, I was unable to find a concrete example in which this behavior breaks imem's rules without also causing compile-time errors.

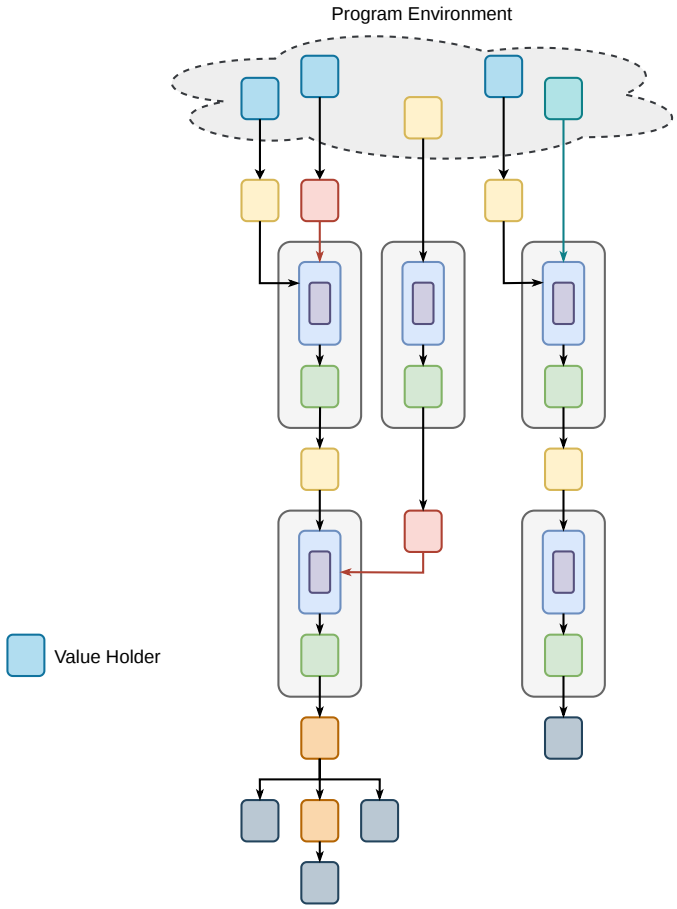

Figure 4.8: Imem Implementation Memory Overview

Chapter 5

Evaluation

5.1 Introduction

This chapter focuses on building a practical and commonly used data structure, a linked list, using imem. The linked list presented in this chapter is a simple polymorphic stack that provides the following interfaces to the user:

- checking whether the list is empty.
- pushing an item onto the stack.
- popping an item from the stack.
- peeking at the first item without popping it.
- iterating over the stack.

The main idea behind the linked list implementation is to avoid exposing the internal list representation to the user. Therefore, all user-facing functions and methods return only linked list objects, element objects, or references, and never the internal data structures used by the list.

To evaluate the performance and practicality of imem, and to clarify what imem enables in practice, the imem implementation is compared against the following baselines:

- ***Rust:*** A minimal implementation in Rust is used as Rust is the main inspiration for imem's design. This baseline allows a direct comparison between imem's ownership and borrow checking and Rust's corresponding mechanisms. The implementation follows the approach presented in the tutorial Learn Rust With Entirely Too Many Linked Lists [19]. The list here corresponds to the one in the section *An Ok Stack*[1] in that tutorial.

[1] https://rust-unofficial.github.io/too-many-lists/second.html

- ***Vanilla Scala:*** A simple Scala implementation that demonstrates how the data structure would look in Scala without ownership or mutability control. This version is largely a direct translation of the Rust implementation into Scala.
- ***Linear Scala:*** This implementation aims to provide as much functionality as possible while strictly adhering to linearity, Scinear, rules. The comparison between this version and the imem implementation illustrates the practical advantages of imem over a purely linear approach.

The chapter also includes examples of non-trivial cases in which violating imem's static rules results in errors, and it demonstrates different situations where this occurs. Finally, the chapter discusses a comparison between the baseline implementations and the imem implementation in terms of functionality support, verbosity, and debuggability.

5.2 Base lines

5.2.1 Rust

This Rust implementation follows the approach described in the tutorial Learn Rust With Entirely Too Many Linked Lists [19].

5.2.1.1 Internal Structures

Internal structures of the linked list and their definitions are as follows:

```
pub struct List<T> {
  head: Link<T>,
}

type Link<T> = Option<Box<Node<T>>>;

struct Node<T> {
  elem: T,
  next: Link<T>,
}
```

The implementation simply models the list as a sequence of nodes linked together. Each node contains an element and a reference to the next node. A link may be `None`, which indicates that there is no subsequent node.

It is important to note that `Link<T>` is defined as an optional `Box` that refers to a node. As a result, each node owns the next node through a `Box`, which causes the entire list to be owned by the first element.

The `List<T>` contains a link to the first node if the list is not empty. Any variable bound to this value therefore owns the entire list.

In Rust, values are moved by default rather than copied. As a result, `List`, `Node`, and `Link` instances are moved when they are not accessed through references, and they therefore behave like linear values.

5.2.1.2 User Interface

In Rust, push, pop, and peek are implemented as follows:

```
impl<T> List<T> {

  pub fn new() -> Self {
    List { head: None }
  }

  pub fn push(&mut self, elem: T) {
    let new_node = Box::new(Node {
      elem: elem,
      next: self.head.take(),
    });
    self.head = Some(new_node);
  }

  pub fn pop(&mut self) -> Option<T> {
    self.head.take().map(|node| {
      self.head = node.next;
      node.elem
    })
  }

  pub fn peek(&self) -> Option<&T> {
    self.head.as_ref().map(|node| &node.elem)
  }

  pub fn peek_mut(&mut self) -> Option<&mut T> {
    self.head.as_mut().map(|node| &mut node.elem)
  }

}
```

The `new` function just creates a list with no node.

The `push` function takes a mutable reference to the list and an element. It first creates a new node for the given element. While constructing this node, the call to `self.head.take()` replaces the current value of `self.head` with `None` and returns the previous value. Finally, a new `Box` is created for the node, and `self.head` is updated to point to this new box.

In the `pop` function, `self.head` is first swapped with `None`, and the resulting `Option` is then mapped. If the list is non-empty, execution enters the function passed to `map`. This function updates `self.head` to the next node of the previously first node. Then it returns the element stored in the popped node, which results in moving that value.

The `peek` and `peek_mut` functions both return an option, since the list may be empty. When present, the option contains a reference to the element of the first node. The reference returned by `peek` is immutable, whereas `peek_mut` returns a mutable reference. Both functions borrow a reference to the first node, if it exists, by using `as_ref` and `as_mut`. They then dereference this node reference and borrow its element, which occurs in the expression `&node.elem` and `&mut node.elem`.

5.2.1.3 Iterators

In this section, all versions try to implement two kinds of iterators: a consuming iterator and a mutable iterator.

5.2.1.3.1 Consuming Iterator A consuming iterator traverses the list by taking ownership of it, which means the list is moved into the iterator and stored as its fields. The following shows the implementation of the consuming iterator in Rust:

```
pub struct IntoIter<T>(List<T>);

impl<T> List<T> {

  pub fn into_iter(self) -> IntoIter<T> {
    IntoIter(self)
  }

}

impl<T> Iterator for IntoIter<T> {

  type Item = T;

  fn next(&mut self) -> Option<Self::Item> {
    // access fields of a tuple struct numerically
    self.0.pop()
  }
}
```

The `IntoIter` structure contains a field that holds the list. The `into_iter` function takes ownership of the list by moving it as an argument, not getting it as a reference, and stores it inside the consuming iterator, `IntoIter<T>`. During iteration, the `next` method repeatedly pops elements from the list, one at a time.

5.2.1.3.2 Mutable Iterator A mutable iterator borrows the list mutably and, unlike a consuming iterator, does not take ownership of the list. The following presents the implementation of the mutable iterator:

```
pub struct IterMut<'a, T> {
  next: Option<&'a mut Node<T>>,
}

impl<T> List<T> {

  pub fn iter_mut(&mut self) -> IterMut<'_, T> {
    IterMut {
      next: self.head.as_deref_mut(),
    }
  }

}

impl<'a, T> Iterator for IterMut<'a, T> {

  type Item = &'a mut T;

  fn next(&mut self) -> Option<Self::Item> {
    self.next.take().map(|node| {
      self.next = node.next.as_deref_mut();
      &mut node.elem
    })
  }

}
```

The `IterMut<'a, T>` structure contains a field, `next`, which stores an optional mutable reference to the next node that the iterator should visit. It is an `Option` because, once iteration reaches the end of the list, there is no node left to visit and the value becomes `None`.

The structure is parameterized by the lifetime `'a` because the iterator holds a reference into the list. Therefore, the iterator must not outlive the mutable borrow of the list. The `'a` lifetime makes this relationship feasible, and it informs the compiler that the reference lifetime can differ from one iterator instance to another.

The `iter_mut` function creates a mutable iterator reference. Its return type is `IterMut<'_, T>`, which instructs the compiler to replace `'_` with the only other lifetime in the function signature, the lifetime of the `&mut self` reference. The function then constructs an `IterMut` instance by dereferencing `self.head` and borrowing it mutably using `as_deref_mut`, if `self.head` is not `None`.

The `next` method first swaps `self.next` with `None` and retrieves the previous value. If this value is not `None`, execution enters the function passed to `map`. At this stage, the method dereferences the reference to the next node and borrows it mutably. It then returns

a mutable reference to the element stored in the current node. The returned reference has the same lifetime as the iterator itself.

It is important to note that although the mutable iterator yields mutable references to elements, the list's structure does not change during iteration. Furthermore, the program can use the list elsewhere and reborrow it mutably or immutably, which causes the reference in the iterator's field to expire and, consequently, the iterator to expire.

5.2.2 Vanilla Scala

This subsection focuses on translating the Rust implementation into Scala and does not follow a pure functional programming model. In Scala, references are managed by a garbage collector, both on the JVM and in Scala Native. As a result, this implementation assumes no ownership relationship between variables and references, and it also assumes that reference lifetimes are not known statically.

Furthermore, as of Scala version 3.7.3, there is no non-experimental support for compile-time mutability control. Therefore, this implementation does not restrict the mutability of objects that are reachable from an element in the list. Instead, it only provides interfaces that allow the element itself to be updated by replacing the reference currently stored in the list with another reference.

5.2.2.1 Internal Structures

The internal structures mirror the Rust implementation:

```
1 class List[T](var head: Link[T] = None)
2
3 type Link[T] = Option[Node[T]]
4
5 class Node[T](var elem: T, var next: Link[T])
```

Similar to the Rust version, the list is modeled as a sequence of nodes linked together. The `List` structure holds a link that may be empty and that refers to the first node.

Each node contains an element and a link to the next node, which may or may not exist.

In this implementation, the `List.head` and `Node.next` fields are mutable, because the structure of the list can change during execution. The `Node.elem` field is also mutable, since updating an element is performed by replacing the reference to the stored object with a new reference.

5.2.2.2 User Interface

The user interfaces are defined as follows:

```
class List[T](var head: Link[T] = None):

  def push(elem: T): Unit = {
    val newNode = new Node(elem, this.head)
    this.head = Some(newNode)
  }

  def pop(): Option[T] = {
    this.head.map { node =>
      this.head = node.next
      node.elem
    }
  }

  def peek: Option[T] = {
    this.head.map(_.elem)
  }

  def peekMut(): Option[Node[T]] = {
    this.head
  }

  ...

end List
```

The overall logic of this implementation matches the Rust version, except that it does not include ownership or borrow checking. One notable difference is that `peekMut` returns an `Option[Node[T]]`, which conflicts with the goal of hiding internal data structures from the user. However, this design choice is necessary, because there is no alternative way to return a value such that modifying the element also updates the element stored in the list. Returning a reference to the element and assigning a new value to that reference would not modify the element stored in the list's node.

5.2.2.3 Iterators

5.2.2.3.1 Consuming Iterator Because standard Scala does not support object ownership, an iterator that consumes a list and becomes its owner cannot be implemented directly in Scala. However, Scala can mimic the Rust implementation of a consuming iterator. The following illustrates the translation of a consuming iterator in Scala:

```
class List[T](var head: Link[T] = None):

  ...

  def intoIter(): Iterator[T] = new Iterator[T] {
    def hasNext: Boolean = head.isDefined
    def next(): T = pop().getOrElse(throw new NoSuchElementException("
   next on empty iterator"))
```

```
  }

  ...

end List
```

Similar to the Rust version, this iterator pops elements from the list as it iterates. When the iterator reaches the end of the list, the `Iterator[T]` trait requires the iterator to throw an exception. Throughout the iteration, the iterator yields the elements themselves.

```
class List[T](var head: Link[T] = None):

  ...

  def iterMut(): Iterator[Node[T]] = new Iterator[Node[T]] {
    private var current: Link[T] = head
    def hasNext: Boolean = current.isDefined
    def next(): Node[T] = {
      current match {
        case Some(node) =>
          current = node.next
          node
        case None =>
          throw new NoSuchElementException("next on empty iterator")
      }
    }
  }

  ...

end List
```

Due to the lack of borrowing and reference mutability control in this version, the mutable iterator must yield `Node[T]` values. In this design, updating a node directly updates the corresponding element in the list. The remainder of the implementation is straightforward.

It is important to note that this approach does not produce compile-time errors when multiple iterators traverse the list simultaneously or when elements are pushed to or popped from the list during iteration.

5.2.3 Linear Scala

The Linear Scala implementation of the linked list serves as a baseline for evaluating the practicality of imem. This implementation uses the Scinear plugin to enforce linearity rules.

This implementation focuses on the case where the elements themselves are linear, and it does not bypass linearity rules using the `@HideLinearity` annotation. A non-linear type can still be an element of the list with the help of a linear wrapper.

5.2.3.1 Internal Structures

The internal structures are quite similar to the Rust and Vanilla Scala implementations:

```
class List[T <: Linear](val head: Link[T]) extends Linear

type Link[T <: Linear] = Option[Node[T]]

class Node[T <: Linear](val elem: T, val next: Link[T]) extends Linear


// `unapply` functions to destruct a node into its element and its link
   to the next node:
object Node:
  def unapply[T <: Linear](node: Node[T]): Option[(T, Link[T])] =
    Some((node.elem, node.next))
end Node
```

As in the other baselines, this version's implementation of a linked list is a chain of nodes connected to one another. Each node contains an element and a link to the next node. A link is an `Option[Node[T]]`, and the list itself holds a link to the first node, if one exists.

All structures are linear, because their fields have linear types, and `Link[T]` is an `Option` that is promoted to a linear type. To allow access to both fields of a `Node`, an `unapply` method is implemented for the `Node` type.

5.2.3.2 User Interface

Because of Scinear rules, this version defines the user interface as functions rather than methods, because in linear classes, methods do not have access to the linear fields and `this`:

```
def push[T <: Linear](list: List[T], elem: T): List[T] =
  val newNode = Node(elem, list.head)
  List(Some(newNode))

def pop[T <: Linear](list: List[T]): (Option[T], List[T]) =
  list.head match
    case Some(Node(elem, next)) =>
      (Some(elem), List(next))
```

```
9       case None =>
10        (None, List(None))
```

Because the elements are linear, it is not possible to implement `peek` or `peekMut` without violating linearity rules. In linear memory, linear objects form a tree structure. The return value of a `peek`/`peekMut` operation would refer to an element while the list node still exists, which breaks this tree structure.

The `push` function takes a list instance and returns a modified list with the new element prepended. It first creates a new node whose `next` field points to the current first node of the list. It then constructs a new list that points to this newly created node.

Similarly, the `pop` function takes a list as input and returns a modified list. In addition, it returns an option containing the removed element. The function checks whether `list.head` is `None`. If so, no element is removed. Otherwise, it returns the first node's element and creates a new list that points to the `next` link of the former node.

5.2.3.3 Iterators

5.2.3.3.1 Consuming Iterator Because a consuming iterator takes ownership of the list by consuming it, the iterator can be implemented without violating linearity rules:

```
1  def intoIter[T <: Linear](list: List[T]): IntoIter[T] = IntoIter(list)
2
3  class IntoIter[T <: Linear](val list: List[T]) extends Linear
4
5  def hasNext[T <: Linear](iter: IntoIter[T]): (IntoIter[T], Boolean) =
6    val (head, notHasNext) = scinear.utils.peekLinearOption(iter.list.head)
7    (IntoIter(List(head)), !notHasNext)
8
9  def next[T <: Linear](iter: IntoIter[T]): (Option[T], IntoIter[T]) =
10   val (elem, nextList) = pop(iter.list)
11   (elem, IntoIter(nextList))
```

As in the other implementations, the `IntoIter` structure simply stores the list as one of its fields. Both `IntoIter` and `List` are linear types, which ensures that no other object can reference the `List` instance once it is passed to `IntoIter`.

The `hasNext` function checks whether the head of the list is empty. The function uses `scinear.utils.peekLinearOption`. Scinear provides this to allow the program to inspect whether an option is `None` without consuming its value. After this check, `hasNext` reconstructs the iterator and the list using the retrieved head node option.

The `next` function calls `pop` to remove the first element of the list, if one exists. It then returns the popped element and constructs a new iterator pointing to the list produced by the `pop` operation.

5.2.3.3.2 Mutable Iterator Similar to the `peek` and `peekMut` functions, mutable iterators are incompatible with linearity rules. A linear mutable iterator would need to hold a reference to the list while another reference to the same list already exists, which would violate linear memory well-formedness by allowing multiple linear objects to point to the same linear value.

Thus, a mutable iterator is not possible for the Linear Scala implementation of the linked list.

5.3 imem

This section illustrates the use of the imem library using an implementation of the linked list as an example.

The imem library uses capture checking, which is currently an experimental compiler feature that is still under development. As a result, the compiler has limited ability to infer capture set type parameters, and the inference also contains bugs that are inconsistent with the capture checking rules. For this reason, this implementation instantiates all type parameters explicitly instead of leaving them to be inferred. This explicit requirement makes imem more difficult to use and results in a more verbose implementation.

5.3.1 Internal Structures

The internal structures of the linked list are implemented in imem as follows:

```
class List[T <: scinear.Linear, 0^](
  _head: Box[Link[T, 0], 0] = newBox[Link[T, 0], 0](None)
) extends scinear.Linear:
  val head: Box[Link[T, 0], 0]^{this} = _head
end List

type Link[T <: scinear.Linear, 0^] = Option[Box[Node[T, 0], 0]]

class Node[T <: scinear.Linear, 0^](_elem: Box[T, 0], _next: Box[Link[T,
    0], 0]) extends scinear.Linear:
  val elem: Box[T, 0]^{this} = _elem
  val next: Box[Link[T, 0], 0]^{this} = _next
end Node

object Node:
  def unapply[T <: scinear.Linear, 0^](node: Node[T, 0]^): (Box[T, 0]^{
    node}, Box[Link[T, 0], 0]^{node}) =
     (node.elem, node.next)
end Node
```

Similar to the linear implementation, all internal structures are linear. This is because `imem.Box` is linear and is a field of the `Node` and `List` classes.

All class fields have an internal field, which starts with `_` and is passed as a constructor argument, and a corresponding external field. The external field has the same value as the internal field, but its type captures `this`. This approach for defining fields is necessary for escape-checking to propagate to the fields of the class and to prevent fields from escaping when an instance is bound to a scope.

All classes also have a capture set type parameter `O^`. This type parameter, similar to Rust lifetime annotations, is a polymorphic type parameter that is instantiated with the set of lifetimes that form the owner set of the list. In other words, all boxes in the list, including the boxes that store list elements, the boxes that point to the next node, and the head box that points to the first node, are available as long as all lifetimes in `O^` are available. Therefore, `O^` represents the owner set of the list.

The `LinkedList` class has two type parameters. The first parameter is the element type `T`, which must be linear, similar to the linear implementation. The second parameter is the owner capture set of the linked list instance, `O^`. The class has a single field, `head`, which is a `Box` containing a `Link`, written as `Box[Link[T, O], O]`, and may point to the first node. Both the `Box` and the `Link` are instantiated with `O`, which makes their lifetimes equal.

A `Link` is an `Option` containing a boxed `Node`, `Option[imem.Box[Node[T, O], O]]`. This definition is equivalent to the one that the Rust version uses, except that `O` instantiates the `Owner^` type parameter of `Box` and the `O^` type parameter of `Node`.

In this version, as in other versions, a node has two fields. One field stores the element that the node references, and the other field points to the next node, if it exists.

The `elem` field of `Node` is a `Box` that points to a value of type `T`, with `O` as its owner, which is `Box[T, O]`. This field represents a reference to an object of type `T`.

The `next` field of `Node` is a `Box` that points to a `Link`, with `O` as its owner, that is `Box[Link[T, O], O]`.

Finally, as in the linear version, since `Node` is a linear class with two fields, an `unapply` method allows access to both fields of a node.

It is important to note the difference between Rust's `List` and `Node` fields and their counterparts in imem. The imem version wraps the fields in an additional `Box` compared to Rust.

This difference exists because, in Rust, holding a mutable reference to an object allows transitive mutation of its fields, which effectively makes those fields mutable. In imem, this behavior does not apply.

To support a similar mechanism, the imem guidelines advise wrapping every field in a `Box`. Since boxes are the source of statically controlled mutability in imem, this approach

allows the program to mutate the fields of an object when the program has a mutable reference to the object.

As a result, the implementation of the user interface functions becomes slightly more complex.

5.3.2 User Interface

To clarify the imem implementation and its details, this section describes each user interface function in turn.

The following presents the implementation of a helper function, `isEmpty`, which checks whether a list is empty:

```
/**
  * @param O1 The list owner set
  * @param O2 The list reference (‘self‘) owner set
  */
def isEmptyList[T <: scinear.Linear, @caps.use O1^, @caps.use O2^, WC^,
    MC^](
  self: ImmutRef[List[T, O1], O2]
)(
  using ctx: Context[WC, MC]^
): Boolean =
  // access the ‘self‘’s reference resource
  read[List[T, O1], O2, Boolean, {ctx}, WC, MC](self,
      list =>
        // new lifetime for borrowing the list’s head
        val lf = Lifetime[{ctx, O2, O1}]()

        // borrow the list’s head immutably, the new immutable reference
    will be ‘headRef‘
        val (headRef, listHolder) = borrowImmutBox[Link[T, O1], O1, {ctx,
     O2}, lf.Key, lf.Owners, WC, MC](list.head)
        // access the head’s (‘headRef‘’s) resource which is a ‘Link‘ to
    the next node
        // then check if the link whether is empty or not
        val isListEmpty = read[Link[T, O1], lf.Owners, Boolean, {ctx, O2
    }, WC, MC](headRef, link => link.isEmpty)

        // unlock the ‘listHolder‘, to use both ‘lf‘ and ‘listHolder‘
    linear variables
        unlockHolder(lf.getKey(), listHolder)
        // return the result
        isListEmpty
  )
```

First, the function signature includes five type parameters:

- `T`: the element type.

- `O1^`: the owner capture set of the list.
- `O2^`: the owner capture set of the immutable reference.
- `WC^`: the capture set that instantiates the `Context` write capability.
- `MC^`: the capture set that instantiates the `Context` move capability.

If `O1^` and `O2^` were the same type parameter, the function could not accept an immutable reference whose lifetime differs from the runtime of the list itself, which is an important use case.

Second, the function takes two arguments. The first argument, `self`, is an immutable reference to the list. The second argument is an implicit parameter, `ctx`, which is the imem `Context` instance.

The function receives an immutable reference to the list because, semantically, `isEmpty` only reads the head of the list to determine whether the list is empty. Therefore, passing a mutable reference is unnecessary. If a mutable reference were passed, the implementation would remain the same, except that the first `read` operation would be replaced with a `write`. Moreover, if the function took the list directly as an argument, then, because a list is a linear value, `isEmpty` would have to return the list instance. Otherwise, calling the function would result in losing access to the list.

The function is simple. It accesses the list, which is the resource of the `self` reference, through the `read` function. Then, a new borrowing lifetime is defined. Next, the function borrows the head of the list and accesses the link stored in the head's `Box` to check whether it is empty. Since the lifetime `lf` and the value holder `listHolder` are linear, the program must consume them, which it does using `unlockHolder`.

The following is the implementation of the push function:

```
/**
  * @param O1 The list owner set
  * @param O2 The list reference (`self`) owner set
  */
def push[T <: scinear.Linear, @caps.use O1^, @caps.use O2^ >: {O1}, @caps
    .use WC^, MC^](
  self: MutRef[List[T, O1], O2]^,
  elem: T
)(
  using ctx: Context[WC, MC]^
): Unit =
  // understand whether the list is empty:
  val (isListEmpty, self2) =
    // new lifetime for borrowing `self` immutably
    val lf = Lifetime[{ctx, O2}]()

    // re-borrow `self` immutably
```

```
17      val (listRef, selfHolder) = borrowImmut[List[T, O1], O2, {ctx, O2},
    lf.Key, lf.Owners, {WC}, {MC}](self)
18      // checks if the list is empty
19      val isListEmpty = isEmptyList(listRef)
20
21      // return the result and unlock the `selfHolder` to get the `self`
    reference
22      (isListEmpty, unlockHolder(lf.getKey(), selfHolder))
23
24  // create a the new node that is going to be pushed:
25  val newNode = Node(
26    newBox[T, O1](elem), // node's element
27    newBox[Link[T, O1], O1](None) // node's next link, which is initially
    `None`
28  )
29
30  if isListEmpty then
31    // access the list mutably
32    writeWithLinearArg[List[T, O1], {O2}, Unit, {ctx}, newNode.type, {WC
    }, {MC}](
33      self2, // the mutable reference to the list
34      newNode,
35      ctx ?=> (list, newNode) =>
36        // set the list's head to point to the new node
37        setBox[Link[T, O1], {O1}, {ctx}, {WC}, {MC}](list.head, Some(
    newBox(newNode)))
38      )
39  else
40    // access the list mutably
41    writeWithLinearArg[List[T, O1], {O2}, Unit, {ctx}, newNode.type, {WC
    }, {MC}](
42      self2,
43      newNode,
44      ctx ?=> (list, newNode) =>
45        // create a temporary box pointing to a link that points to the
    new node
46        val tempHead = newBox[Link[T, O1], O1](Some(newBox[Node[T, O1],
    O1](newNode)))
47        // the state at here: {(temp head -> new node), (list -> previous
     first node), (new node -> None)}
48
49        // swap the `tempHead` and the `list.head`, so that the `list.
    head` points to the new node
50        val (tempHead2, listHead) = swapBox(tempHead, list.head)
51        // the state at here: {(temp head -> previous first node) and (
    list -> new node), (newNode -> None)}
52
53        // new lifetime for borrowing the list's head mutably
54        val lf = Lifetime[{ctx, O1}]()
55
56        // borrow the list's head mutably
```

```
57          val (listHeadRef, listHeadHolder) = borrowMutBox[Link[T, O1], O1,
     {ctx}, lf.Key, lf.Owners, {WC}, {MC}](listHead)
58
59          // access the `listHeadRef`'s resource, which is list's head,
    mutably
60          writeWithLinearArg[Link[T, O1], lf.Owners, Unit, {ctx, O2},
    tempHead2.type, {WC}, {MC}](
61            listHeadRef,
62            tempHead2,
63            ctx ?=> (head, tempHead2) =>
64              // get the head's box, that is pointing to the newly created
    node
65              val nodeBox = head.get
66
67              // new lifetime for borrowing the new node mutably
68              val lfInner = Lifetime[{ctx, O1}]()
69
70              // borrow the new node mutably
71              val (nodeRef, nodeBoxHolder) = borrowMutBox[Node[T, O1], O1,
    {ctx}, lfInner.Key, lfInner.Owners, {WC}, {MC}](nodeBox)
72              // access the `nodeRef`'s resource, which is the new node,
    mutably
73              writeWithLinearArg[Node[T, O1], lfInner.Owners, Unit, {ctx},
    tempHead2.type, {WC}, {MC}](
74                nodeRef,
75                tempHead2,
76                ctx ?=> (node, tempHead2) =>
77                  // swap the new node's next and the temporary box
78                  swapBox[Link[T, O1], {O1}, {O1}, {ctx}, {WC}, {MC}](node.
    next, tempHead2)
79                  // the state at here: {(temp head -> None), (list -> new
    node), (new node -> previous first node)}
80                  // by here, the new node is pushed to the list
81                  ()
82              )
83
84              // unlock the `nodeBoxHolder`, to use both `lfInner` and `
    nodeBoxHolder` linear variables
85              unlockHolder(lfInner.getKey(), nodeBoxHolder)
86              ()
87            )
88
89          // unlock the `listHeadHolder`, to use both `lf` and `
    listHeadHolder` linear variables
90          unlockHolder(lf.getKey(), listHeadHolder)
91          ()
92      )
```

The function type parameters are similar to those of `isEmptyList`, with a minor change: `O2` must be a superset of `O1`. This requirement enables pushing by ensuring that the reference does not outlive the list. The `push` function modifies the values of some of the

boxes, so the program signature has to annotate the `WC` type parameter with `@cap.use`. The function also takes an additional argument, `elem`, which is the value of the new element that will be pushed onto the list.

The `push` implementation uses another imem interface, `writeWithLinearArg`. This interface works like `write`, but it also allows the program to pass linear values to the `writeAction` via an explicit argument. This is useful because an anonymous function passed as the `writeAction` cannot use linear values available in the scope that encloses the call site of `writeWithLinearArg`. However, in some cases, the program needs to use such linear values, and it is acceptable for the program to expire them after calling `writeWithLinearArg`.

To push a new node onto the list, the program must update the box that the list head link points to, or in Scala terms, `list.head.get`, so that it points to the newly created node. In addition, the newly created node must point to the node that was previously the first node in the list, which is the node referenced by the `list.head.get` box.

Figure 5.1 illustrates the list structure.

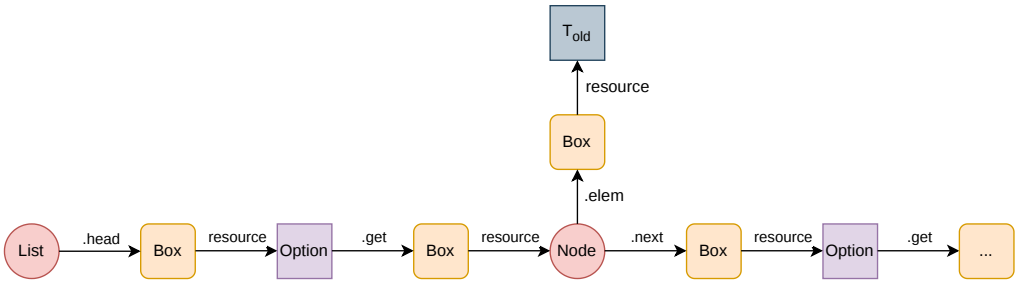

Figure 5.1: List Overview

Moreover, Figure 5.2 demonstrates what the result would be after pushing the new element.

If the list is empty, only the first modification is enough, which is setting `list.head.get` to point to the new node.

If the list is not empty, an additional step is required. First, the function creates a temporary head that points to the new node. Then the `push` function swaps the temporary head with the list head. Now the list points to the new node, and the temporary head points to the previously first node. Finally, the function swaps the temporary head link with the `next` link of the newly created node. This results in the list head pointing to the new node, and the new node pointing to the rest of the list from before the push.

The pop functionality implementation in imem is similar to push:

```
1 /**
2   * @param O1 The list nodes owner set
3   * @param O2 The list reference (‘self‘) owner set
```

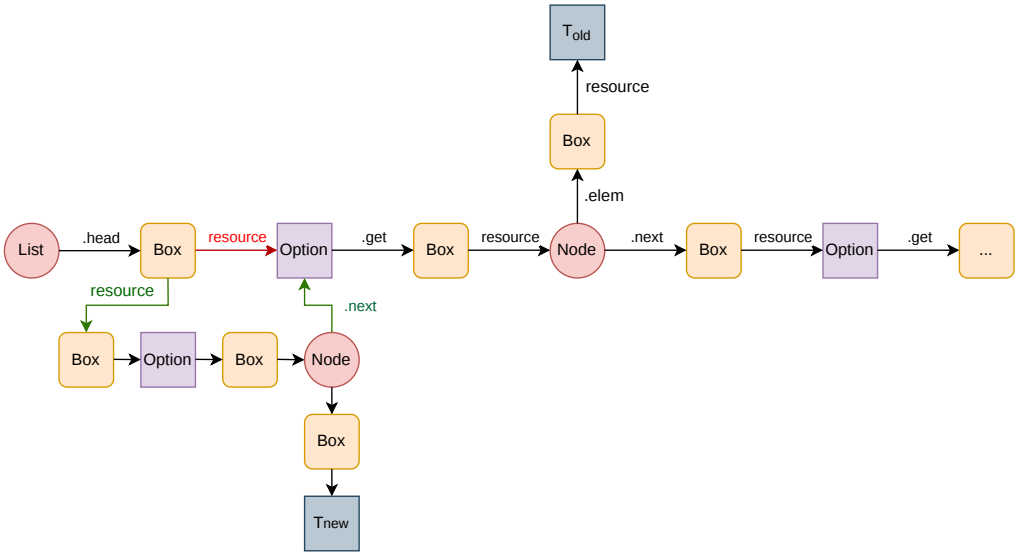

Figure 5.2: List After Push Overview

```
  * @param O3 The returned box owner set (which contains the popped
    element)
  */
def pop[T <: scinear.Linear, @caps.use O1^, @caps.use O2^ >: {O1}, @caps.
    use O3^ >: {O2}, @caps.use WC^, @caps.use MC^](
  self: MutRef[List[T, O1], O2]
)(
  using ctx: Context[WC, MC]^
): Option[Box[T, O3]] =
  // understand whether the list is empty:
  val (isListEmpty, self2) =
    // new lifetime for borrowing ‘self‘ immutably
    val lf = Lifetime[{ctx, O2}]()

    // re-borrow ‘self‘ immutably
    val (listRef, selfHolder) = borrowImmut[List[T, O1], O2, {ctx, O2},
    lf.Key, lf.Owners, {WC}, {MC}](self)
    // checks if the list is empty
    val isListEmpty = isEmptyList(listRef)

    // return the result and unlock the ‘selfHolder‘ to get the ‘self‘
    reference
    (isListEmpty, unlockHolder(lf.getKey(), selfHolder))

  if isListEmpty then
    // converge if branches by consuming ‘self2‘
    self2.consume()
    // return None, no element to pop
    None
```

```
29  else
30    // temporary box to hold the list's head
31    val (tempHead, self3) =
32      // new lifetime for borrowing `self2` mutably
33      val lf = Lifetime[{ctx, O3}]()
34
35      // borrow `self2` mutably
36      val (listRef, selfHolder) = borrowMut[List[T, O1], O2, {ctx, O2},
    lf.Key, lf.Owners, {WC}, {MC}](self2)
37
38      // create the temporary box
39      // the temporary box's lifetime capture set is `{O3}`, as it is
    going to be returned
40      val tempHead = newBox[Link[T, O1], O3](None)
41      // the state at here: {(temp head -> None), (list -> first node), (
    first node -> second node and the rest of the list)}
42
43      // access the `listRef`'s resource, which is the list, mutably
44      val tempHead2 = writeWithLinearArg[List[T, O1], lf.Owners, Box[Link
    [T, O1], O3], {ctx}, tempHead.type, {WC}, {MC}](
45        listRef,
46        tempHead,
47        ctx ?=> (list, tempHead) =>
48          // swap the `tempHead` and the `list.head`, so that the `
    tempHead` points to the first node
49          val (newTempHead, listHead) = swapBox[Link[T, O1], {O3}, {O1},
    {ctx}, {WC}, {MC}](tempHead, list.head)
50          // the state at here: {(temp head -> first node), (list -> None
    ), (first node -> second node and the rest of the list)}
51
52          // `listHead` is a linear variable and have to be used
53          // use `listHead` by consuming
54          listHead.consume()
55          // return the newly created temporary box pointing to the first
     node
56          newTempHead
57      )
58
59      // return the temporary box and unlock the `selfHolder` to get the
    `self2` reference
60      (tempHead2, unlockHolder(lf.getKey(), selfHolder))
61
62    val tempHead2 =
63      // new lifetime for borrowing `tempHead` mutably
64      val lf = Lifetime[{ctx, O3}]()
65
66      // borrow `tempHead` mutably
67      val (tempHeadRef, tempHeadHolder) = borrowMutBox[Link[T, O1], O3, {
    ctx, O2}, lf.Key, lf.Owners, {WC}, {MC}](tempHead)
68
69      // access the `tempHeadRef`'s resource, which is the link to the
```

```
    list's first node, mutably
70      writeWithLinearArg[Link[T, O1], lf.Owners, Unit, {ctx}, self3.type, {WC}, {MC}](
71          tempHeadRef,
72          self3,
73          ctx ?=> (head, self3) =>
74            // new lifetime for borrowing the list's first node mutably
75            val lfInner = Lifetime[{ctx, O1}]()
76
77            // borrow the list's first node mutably
78            val (firstNodeRef, firstNodeBoxHolder) = borrowMutBox[Node[T, O1], O1, {ctx}, lfInner.Key, lfInner.Owners, {WC}, {MC}](head.get)
79
80            // access the `self3`'s resource, which is the list, mutably
81            writeWithLinearArg[List[T, O1], {O2}, Unit, {ctx}, firstNodeRef.type, {WC}, {MC}](
82              self3,
83              firstNodeRef,
84              ctx ?=> (list, firstNodeRef) =>
85
86                // access the `firstNodeRef`'s resource, which is the first node, mutably
87                writeWithLinearArg[Node[T, O1], lfInner.Owners, Unit, {ctx}, list.type, {WC}, {MC}](
88                  firstNodeRef,
89                  list,
90                  ctx ?=> (firstNode, list) =>
91                    // swap the node's next and the list's head
92                    swapBox[Link[T, O1], {O1}, {O1}, {ctx}, {WC}, {MC}](firstNode.next, list.head)
93                    // the state at here: {(temp head -> first node), (list -> second node and the rest of the list), (first node -> None)}
94                    ()
95                )
96            )
97
98            // unlock the `firstNodeBoxHolder`, to use both `lfInner` and `firstNodeBoxHolder`
99            unlockHolder(lfInner.getKey(), firstNodeBoxHolder)
100     )
101
102   // return the temporary box and unlock the `tempHeadHolder` to get the `tempHead` reference
103     unlockHolder(lf.getKey(), tempHeadHolder)
104
105   // in here, the `tempHead2` points to the popped first node and the list points to the rest of the list
106   // the runtime memory state is correct but the lifetime capture set of the `tempHead2` still contains `{O1}`
107   // `tempHead2`'s type is: `Box[Option[Box[Node[T, O1], O1]], O3]`
108
```

```
109     // access the `tempHead2`'s resource, which is the link to the popped
        node
110     derefForMoving[Link[T, O1], O3, {ctx}, Option[Box[T, O3]], {WC}, {MC
        }](
111       tempHead2,
112       link =>
113         // box to the popped node
114         val nodeBox = link.get
115         // access the `nodeBox`'s resource, which is the popped node
116         val movedElem = derefForMoving[Node[T, O1], O1, {ctx}, Box[T, O3
        ], {WC}, {MC}](
117           nodeBox,
118           // move the box to popped node's element to the new owner
119           node => moveBox[T, O1, {O3}, {WC}, {MC}](node.elem)
120         )
121
122         // return the moved element box
123         Some(movedElem)
124     )
```

The `pop` function introduces a new capture set type parameter, `O3^`, which represents the lifetime set associated with the popped element. `O3^` must be a superset of `O2`, the reference lifetime set, because during the `pop` implementation, there is a point where a reference with lifetime set `O3` reaches a reference with lifetime set `O2`. After the element is popped, the program can freely move the `Box` to any lifetime, with no required relationship to either `O3` or `O2`.

Also, the function returns an `Option` to a `Box` to an element. The `Box` lifetime capture set is `O3`.

To pop an element from the list, the program must update the box that the head link points to, in Scala terms, `list.head.get`, so that it points to the second node, and the function must return a box that points to the first node's element.

Figure 5.3 illustrates the state of the list after a `pop` operation.

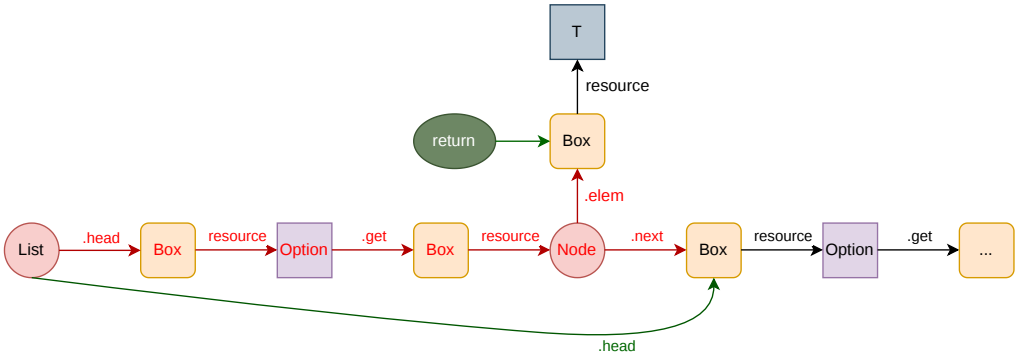

Figure 5.3: List After Pop Overview

To preserve the imem memory well-formedness requirement of having only one direct box, the first next link of the first node must no longer point to the second node. If the list is empty, a pop operation returns `None`. If the list is nonempty, the function first creates a temporary box and swaps it with the head of the list.

After this swap, the temporary box points to the first node, and the head of the list points to nothing. Next, the `pop` function accesses the first node and swaps the first node's box that points to the link that is pointing to the second node with the list's head. After this swap, the temporary box points to the first node, the list's head points to the remainder of the list without the first node, and the first node points to nothing.

At this point, the first node is removed from the list. However, the function must return a box that points to the element with the required lifetime, `O3`. Therefore, the `pop` function dereferences the temporary box, then dereferences the first node, and finally moves out the box that points to the first node's element. As a result, the function returns a box that points to the first node's element, and that box has lifetime `O3`.

The `peek` function implementation in imem is the code that follows:

```
/**
  * @param O1 The list nodes owner set
  * @param O2 The list reference (‘self‘) owner set
  * @param O3 The context owner set, i.e. the context aggregated set of owners
  * @param O4 The returned reference owner set (which contains the peeked element)
  */
def peek[T <: scinear.Linear, @caps.use O1^, @caps.use O2^, O3^, O4Key, @caps.use O4^ >: {O1, O2, O3}, WC^, MC^](
  self: ImmutRef[List[T, O1], O2]
)(
  using ctx: Context[WC, MC]^{O3}
): Option[ImmutRef[T, O4]] =
  // access the ‘self‘’s reference resource, which is the list
  read[List[T, O1], O2, Option[ImmutRef[T, O4]], {ctx}, {WC}, {MC}](self, ctx ?=> list =>
    // borrow the list’s head immutably
    val (headRef, listHeadHolder) = borrowImmutBox[Link[T, O1], O1, {ctx}, O4Key, O4, {WC}, {MC}](list.head)

    // use ‘listHeadHolder‘ by consuming because it is a linear variable that is no longer needed
    listHeadHolder.consume()

    // access the head’s (‘headRef‘’s) resource, which is a link to the next node
    read[Link[T, O1], O4, Option[ImmutRef[T, O4]], {ctx}, {WC}, {MC}](headRef,
      ctx ?=> linkOpt =>
        // without consuming the option, check whether ‘linkOpt‘ is empty or not
```

```
25          val (linkOpt2, isListEmpty) = scinear.utils.peekLinearOption(
    linkOpt)
26
27          // the behavior changes based on list emptiness
28          if isListEmpty then
29            // converge if branches by consuming `linkOpt2`
30            linkOpt2.isEmpty
31            // return None, no element to peek
32            None
33          else
34            // get the box pointing to the list's first node
35            val nodeBox = linkOpt2.get
36            // borrow the first node immutably
37            val (nodeRef, nodeBoxHolder) = borrowImmutBox[Node[T, O1], O1
    , {ctx}, O4Key, O4, {WC}, {MC}](nodeBox)
38            // use `nodeBoxHolder` by consuming because it is a linear
    variable that is no longer needed
39            nodeBoxHolder.consume()
40
41            // access the `nodeRef`'s resource, which is the first node
42            read[Node[T, O1], O4, Option[ImmutRef[T, O4]], {ctx}, {WC}, {
    MC}](
43              nodeRef,
44              ctx ?=> node =>
45                // borrow the first node's element immutably
46                val (nodeRef, nodeElemHolder) = borrowImmutBox[T, O1, {
    ctx}, O4Key, O4, {WC}, {MC}](node.elem)
47                // use `nodeElemHolder` by consuming because it is a
    linear variable that is no longer needed
48                // the function only needs the reference to the element
49                nodeElemHolder.consume()
50                // return the reference to list's first node's element
51                Some(nodeRef)
52            )
53      )
54  )
```

The `peek` function signature introduces two new type parameters: `O3^`, which is the `Context` lifetime capture set, and `O4^`, which is the lifetime set of the reference the `peek` returns. The result of `peek` is a reference to the first element of the list. Therefore, the reference must expire before the list, before the reference to the list, and before all lifetimes aggregated by the `Context` up to this function. For this reason, the constraint `O4^ >: O1, O2, O3` defines `O4^` as a superset of all these capture sets.

The implementation of the function is straightforward. The `peek` function must borrow the box that points to the element of the first node. To reach this box, it uses the `read` function to go through the reference to the list and access the head of the list. Then, it borrows the list's head, follows the link to the first node, and, if the list is nonempty, reaches the box pointing to the first node. Finally, it borrows the box pointing to the first node and accesses the box that points to the first node's element.

The implementation of `peekMut` follows the same structure as `peek`:

```
/**
  * @param O1 The list nodes owner set
  * @param O2 The list reference (‘self‘) owner set
  * @param O3 The context owner set, i.e. the context aggregated set of owners
  * @param O4 The returned reference owner set (which contains the peeked element)
  */
def peekMut[T <: scinear.Linear, @caps.use O1^, O2^, O3^, O4Key, @caps.use O4^ >: {O1, O2, O3}, @caps.use WC^, MC^](
  self: MutRef[List[T, O1], O2]
)(
  using ctx: Context[WC, MC]^{O3}
): Option[MutRef[T, O4]] =
  // access the ‘self‘’s reference resource, which is the list
  write[List[T, O1], O2, Option[MutRef[T, O4]], {ctx}, {WC}, {MC}](self,
    ctx ?=> list =>
      // borrow the list’s head mutably
      val (headRef, listHeadHolder) = borrowMutBox[Link[T, O1], O1, {ctx}, O4Key, O4, {WC}, {MC}](list.head)

      // use ‘listHeadHolder‘ by consuming because it is a linear variable that is no longer needed
      listHeadHolder.consume()

      // access the head’s (‘headRef‘’s) resource, which is a link to the next node
      write[Link[T, O1], O4, Option[MutRef[T, O4]], {ctx}, {WC}, {MC}](headRef,
        ctx ?=> linkOpt =>
          // without consuming the option, check whether ‘linkOpt‘ is empty or not
          val (linkOpt2, isListEmpty) = scinear.utils.peekLinearOption(linkOpt)

          // the behavior changes based on list emptiness
          if isListEmpty then
            // converge if branches by consuming ‘linkOpt2‘
            linkOpt2.isEmpty
            // return None, no element to peek
            None
          else
            // get the box pointing to the list’s first node
            val nodeBox = linkOpt2.get
            // borrow the first node mutably
            val (nodeRef, nodeBoxHolder) = borrowMutBox[Node[T, O1], O1, {ctx}, O4Key, O4, {WC}, {MC}](nodeBox)
            // use ‘nodeBoxHolder‘ by consuming because it is a linear variable that is no longer needed
            nodeBoxHolder.consume()
```

```
40
41              // access the `nodeRef`'s resource, which is the first node
42              write[Node[T, O1], O4, Option[MutRef[T, O4]], {ctx}, {WC}, {
    MC}](
43                nodeRef,
44                ctx ?=> node =>
45                  // borrow the first node's element mutably
46                  val (nodeRef, nodeElemHolder) = borrowMutBox[T, O1, {ctx
    }, O4Key, O4, {WC}, {MC}](node.elem)
47                  // use `nodeElemHolder` by consuming because it is a
    linear variable that is no longer needed
48                  // the function only needs the reference to the element
49                  nodeElemHolder.consume()
50                  // return the mutable reference to list's first node's
    element
51                  Some(nodeRef)
52              )
53        )
54    )
```

The implementation of the `peekMut` function closely follows that of `peek`. The differences are as follows:

- The `peekMut` function annotates the `WC^` type parameter with `@caps.use` because it calls `borrowMutBox` in its body.
- The `peekMut` function uses `write` and `borrowMutBox` instead of the `read` and `borrowImmutBox` functions used by `peek`.

5.3.3 Iterators

Both consuming and mutable iterators can be implemented in imem.

5.3.3.1 Consuming iterators

The following is the class definition of a consuming iterator in imem:

```
1 /**
2   * @param O The consuming iterator owner set
3   */
4 class ConsumingIterator[T <: scinear.Linear, O^](_list: Box[List[T, O], O
    ]) extends scinear.Linear:
5   val list: Box[List[T, O], O]^{this} = _list
6 end ConsumingIterator
```

The consuming iterator is a linear class with an external `list` field. This field is a `Box[List[T, O]]` that points to the list that the iterator consumes. The class has two

type parameters: the element type `T` and the ownership set of the list, `O^`. The iterator stores the list in a box that has the same lifetime as the list.

The following shows the implementation of the "has next" functionality:

```
/**
  * @param O1 The consuming iterator owner set
  * @param O2 The reference to the consuming iterator (`self`) owner set
  */
def hasNextCI[T <: scinear.Linear, @caps.use O1^, @caps.use O2^ >: {O1}, WC^, MC^](
  self: ImmutRef[ConsumingIterator[T, O1], O2]^
)(
  using ctx: Context[WC, MC]^
): Boolean =
  // access the `self`'s reference resource, which is the consuming iterator
  read[ConsumingIterator[T, O1], O2, Boolean, {ctx}, {WC}, {MC}](self,
    ctx ?=> iter =>
      // new lifetime for borrowing the iterator's list box immutably
      val lf = Lifetime[{ctx, O2}]()

      // borrow the iterator's list box immutably
      val (listRef, listHolder) = borrowImmutBox[List[T, O1], O1, {ctx}, lf.Key, lf.Owners, {WC}, {MC}](iter.list)
      // check whether the list is empty
      val hasNext = !isEmptyList[T, {O1}, lf.Owners, {WC}, {MC}](listRef)

      // use both `lf` and `listHolder` linear variables
      unlockHolder(lf.getKey(), listHolder).consume()

      // return the result
      hasNext
  )
```

The function type parameters follow the same pattern as those in the `push`, `pop`, and `peek` linked list functions. The `hasNextCI` function takes an immutable reference to a consuming iterator, `ImmutRef[ConsumingIterator[T, O1], O2]^`, which is the iterator. The function then borrows this reference to access the iterator, borrows the box that points to the iterator's list immutably, and calls `isEmptyList` to determine whether the list is empty. Finally, it returns the result.

The `nextCI` function's signature is similar:

```
/**
  * @param O1 The consuming iterator owner set
  * @param O2 The reference to the consuming iterator (`self`) owner set
  */
def nextCI[T <: scinear.Linear, @caps.use O1^, @caps.use O2^ >: {O1}, @caps.use WC^, @caps.use MC^](
  self: MutRef[ConsumingIterator[T, O1], O2]^
)(
```

```
  using ctx: Context[WC, MC]^
): Option[Box[T, O2]] =
  // access the 'self''s reference resource, which is the consuming iterator
  val listRef = write[ConsumingIterator[T, O1], O2, MutRef[List[T, O1], {O2, ctx}], {ctx}, {WC}, {MC}](self,
    ctxInner ?=> iter =>
      // borrow the iterator's list box mutably
      val (listRef, listHolder) = borrowMutBox[List[T, O1], O1, {ctxInner}, NeverUsableKey, {O2, ctx}, {WC}, {MC}](iter.list)
      // use 'listHolder' by consuming because it is a linear variable that is no longer needed
      listHolder.consume()
      // return the mutable reference to the iterator's list
      listRef
  )

  // pop an element from the list
  val poppedElem = pop[T, {O1}, {O2, ctx}, {O2, ctx}, {WC}, {MC}](listRef)
  // here the popped element's runtime value matches the function's return value
  // but, its lifetime capture set is {O2}, and the function returns a box with
  // with lifetime capture set {O2}

  // move the 'poppedElem' from {O2, ctx} to {O2}
  val movedPoppedElem =
    // without consuming the option, check whether 'poppedElem' is empty or not
    val (poppedElem2, isNone) = scinear.utils.peekLinearOption(poppedElem)
    if isNone then
      // converge if branches by consuming 'poppedElem2'
      poppedElem2.isEmpty
      // return None with the correct type
      None
    else
      // move the box pointing to the popped element to the needed owner
      Some(moveBox[T, {O2, ctx}, {O2}, {WC}, {MC}](poppedElem2.get))

  // return the moved popped element
  movedPoppedElem
```

The function takes a mutable reference to the iterator as an argument. The function requires this reference to be mutable because the function modifies the structure of the list in the iterator in order to return the box containing the first element.

The implementation starts by popping an element from the list. To achieve this, the function first borrows the iterator reference to access the list. It then mutably borrows the box that points to the list and returns that reference.

When borrowing the box that points to the list, which is `iter.list`, the program no longer needs to access the box later and only needs a mutable reference to the list. Therefore, the box is borrowed with `O2, ctx` as the owner of the reference and `NeverUsableKey` as the key. This approach allows the program to borrow the reference without tying it to a new lifetime. The `NeverUsableKey` is an opaque type for which no instances can be created, which makes the `ValueHolder` impossible to unlock. Further details are provided in section 4.6.

At this point, the function has a mutable reference to the list and uses it to pop an element. The popped element is the element that the iterator emits, but its type is not correct. This mismatch occurs because the popped element's box has the lifetime set `O2, ctx`, which is the same as the mutable reference.

The lifetime set cannot be `O2` because the `pop` function signature specifies that the returned box lifetime, `O3^` in `pop`'s signature, must be a superset of the reference lifetime, `O2^` in `pop`'s signature. At this usage point, the reference lifetime is `O2, ctx`.

To change the lifetime set of the popped box from `O2, ctx` to `O2`, the function moves the box if the result of `pop` is not `None`.

The `intoIter` function creates a new consuming iterator:

```
/**
  * @param O1 The list and the box to the list (‘self‘) owner set
  * @param O2 The consuming iterator owner set
  */
def intoIter[T <: scinear.Linear, @caps.use O1^, O2^, WC^, @caps.use MC
    ^](
  self: Box[List[T, O1], O1]
)(
  using ctx: Context[WC, MC]^
): ConsumingIterator[T, O2] =
  // dereference the ‘self‘ box, to access the ‘self‘’s resource, which
    is the list
  derefForMoving[List[T, O1], O1, {ctx}, ConsumingIterator[T, {O2}], {WC
    }, {MC}](
    self,
    list =>
      // move all elements from owner {O1} to owner {O2}
      val movedListHead = moveAllElems[T, {O1}, O2, {WC}, {MC}](list.head
    )
      // create a new list in owner {O2} with the moved head
      val newList = List[T, {O2}](movedListHead)
      // create a new box pointing to the new list with owner lifetime
    set {O2}
      val newListBox = newBox[List[T, {O2}], {O2}](newList)
      // create and return the consuming iterator
      ConsumingIterator[T, {O2}](newListBox)
  )
```

The function gets a box pointing to a list and returns a consuming iterator. The important thing about the signature is that the list and its box's lifetime set is `O1` while the consuming iterator type parameter that is the consuming iterator lifetime set is instantiated with `O2`. Furthermore `O1^` and `O2^` have no relation with each other.

The `intoIter` function moves all of elements of the list from the `O1` lifetime set to the `O2` lifetime set to match the return argument. This is unnecessary if the consumed list and the consuming iterator both use the list's lifetime parameter. However, tying the consuming iterator to the list's lifetime parameter makes the implementation less expressive, because a consuming iterator should be able to have its own lifetime while it consumes the list and later returns its elements one by one, regardless of the lifetime of the consumed list.

The `moveAllElems` function moves the elements:

```
/**
  * @param O1 The list and the box to the list (`self`) owner set
  * @param O2 The target owner set to move the list to
  */
def moveAllElems[T <: scinear.Linear, @caps.use O1^, O2^, WC^, @caps.use MC^](
  self: Box[Link[T, O1], O1]^
)(
  using ctx: Context[WC, MC]^
): Box[Link[T, O2], O2] =
  // dereference the `self` box, to access the `self`'s resource, which is a link to the next node
  derefForMoving[Link[T, O1], O1, {ctx}, Box[Link[T, O2], O2], {WC}, {MC}](
    self,
    ctx ?=> nextOpt =>
      // without consuming the option, check whether `nextOpt` is empty or not
      val (nextOpt2, isEmpty) = scinear.utils.peekLinearOption(nextOpt)

      // the behavior changes based on whether there is a next node or not
      if isEmpty then
        // converge if branches by consuming `nextOpt2`
        nextOpt2.isEmpty
        // return a box pointing to None
        newBox[Link[T, O2], O2](None)
      else
        // get the box pointing to the next node
        val nodeBox = nextOpt2.get
        // access the `nodeBox`'s resource, which is the next node
        val newBoxToLink = derefForMoving[Node[T, O1], O1, {ctx}, Box[Link[T, O2], O2], {WC}, {MC}](
          nodeBox,
          node =>
            // get the element and the next link of the node
            val (nodeElem, nodeNext) = Node.unapply(node)
```

```
32              // move the element
33              val movedElemBox = moveBox[T, {O1}, O2, {WC}, {MC}](nodeElem)
34              // move all elements from the next link recursively
35              val movedNextBox = moveAllElems[T, {O1}, O2, {WC}, {MC}](
    nodeNext)
36              // create a new node in owner lifetime set {O2} with the
    moved element and moved next link
37              val newNode = Node[T, O2](movedElemBox, movedNextBox)
38              // return a box pointing to a link that points to the new
    node
39              newBox[Link[T, O2], O2](Some(newBox[Node[T, O2], O2](newNode)
    ))
40          )
41          // return the new box to link that points the new next node
42          newBoxToLink
43    )
```

The `moveAllElems` function takes a box that points to a link referencing a node, `Box[Link[T, O1], O1]`, and returns the same structure with a different lifetime set as the owner, `Box[Link[T, O2], O2]`. The function recursively traverses the nodes by dereferencing the boxes, moves the elements in each node, and attaches them to new nodes that use the new lifetimes.

5.3.3.2 Mutable iterator

The following presents the definition of the mutable iterator class:

```
1 /**
2   * @param O1 The list to be iterated owner set
3   * @param O2 The mutable iterator and the mutable reference to the list
     owner set
4   */
5 class MutableIterator[T <: scinear.Linear, O1^, O2^ >: {O1}](_boxToLink:
    Box[MutRef[Link[T, O1], O2], O2]) extends scinear.Linear:
6   val boxToLink: Box[MutRef[Link[T, O1], O2], O2]^{this} = _boxToLink
7 end MutableIterator
```

To implement a mutable iterator in imem, the structure must hold a mutable reference to a node in the list. Because this mutable reference changes each time the iterator advances to the next node, it must be stored in a mutable field. According to the imem guidelines, `Box` is the only source of mutation that is statically controlled by imem. As a result, the iterator's `boxToLink` field is a box containing a mutable reference that points to a link, which either refers to the next node to be visited or is `None`.

Another important aspect is that the `MutableIterator` class has two capture set type parameters:

- `O1^`: the owner set of the list and, transitively, the nodes and links.

- `O2^`: the owner set of the mutable reference, which must be a superset of `O1^`.

Because the class includes both parameters, an iterator instance expires when any lifetime capability in either `O1^` or `O2^` expires. Since `O2^` is a superset of `O1^`, this condition is equivalent to the expiration of any member of `O2^`.

The `iterMut` function provides an interface to create a mutable iterator from a mutable reference to a list:

```
/**
  * @param O1 The list to be iterated owner set
  * @param O2 The mutable reference to the list (‘self‘) and the
    resulting mutable iterator owner set
  * @param O3 The context owner set, i.e. the context aggregated set of
    owners
  */
def iterMut[T <: scinear.Linear, @caps.use O1^, O3^, O2^ >: {O1, O3}, @
    caps.use WC^, MC^](
  self: MutRef[List[T, O1], O2]
)(
  using ctx: Context[WC, MC]^{O3}
): MutableIterator[T, O1, O2] =
  // access the ‘self‘’s reference resource, which is the list
  write[List[T, O1], O2, MutableIterator[T, O1, O2], {ctx}, {WC}, {MC}](
    self,
    ctx ?=> list =>
      // borrow the list’s head mutably
      val (headRef, listHeadHolder) = borrowMutBox[Link[T, O1], O1, {ctx
    }, NeverUsableKey, {O2}, {WC}, {MC}](list.head)
      // use ‘listHeadHolder‘ by consuming because it is a linear
    variable that is not needed
      listHeadHolder.consume()
      // create and return the mutable iterator
      MutableIterator[T, O1, O2](newBox[MutRef[Link[T, O1], O2], O2](
    headRef))
  )
```

The `iterMut` function takes three type parameters:

- `O1^`: as in the `MutableIterator` class definition, this is the owner set associated with the list.
- `O2^`: as in `MutableIterator`, this is the owner set of both the mutable reference and the mutable iterator.
- `O3^`: the owner set aggregated by the context.

The owner set `O2^` must be a superset of `O1^`, due to the same requirement as in the definition of `MutableIterator`. It must also be a superset of `O3^` because, during iteration,

the program repeatedly borrows list nodes and updates the mutable reference to point to newly borrowed mutable references. If the context-aggregated owner set is not a subset of the reference owner set, the newly borrowed references cannot use `O2^` as their lifetime set.

The function takes a mutable reference to the list and returns a mutable iterator that points to the first node of the list. To construct this mutable reference, the function accesses the list, borrows its head link to the first node, and creates a new box that points to the borrowed head reference.

```
/**
  * @param O1 The list to be iterated owner set
  * @param O2 The mutable iterator owner set
  * @param O3 The reference to the mutable iterator (`self`) owner set
  */
def hasNextMI[T <: scinear.Linear, @caps.use O1^, O3^, @caps.use O2^ >: {O1, O3}, WC^, MC^](
  self: ImmutRef[MutableIterator[T, O1, O2], O3]^
)(
  using ctx: Context[WC, MC]^{O3}
): Boolean =
  // access the `self`'s reference resource, which is the mutable iterator
  read[MutableIterator[T, O1, O2], O3, Boolean, {ctx}, {WC}, {MC}](self,
    ctx ?=> iter =>
      // new lifetime for borrowing the iterator's box to the link,
      // which is pointing to the next node to be visited, immutably
      val lf = Lifetime[{O2}]()

      // borrow the iterator's box to the link, immutably
      val (refToLink, boxToLinkHolder) = borrowImmutBox[MutRef[Link[T, O1], O2], O2, {ctx}, lf.Key, lf.Owners, {WC}, {MC}](iter.boxToLink)
      // access the `refToLink`'s resource, which is the mutable reference to the link pointing to the next node to be visited
      val hasNext = read[MutRef[Link[T, O1], O2], lf.Owners, Boolean, {ctx, O3}, {WC}, {MC}](refToLink,
        ctx ?=> linkMutRef =>
          // new lifetime for borrowing the mutable reference to the link immutably
          val lf = Lifetime[{ctx, O2}]()

          // borrow the link mutable reference immutably
          val (linkRef, linkHolder) = borrowImmut[Link[T, O1 ], O2, {ctx, O3}, lf.Key, lf.Owners, {WC}, {MC}](linkMutRef)
          // access the `linkRef`'s resource, which is a link to the next node to be visited
          val hasNext = read[Link[T, O1], lf.Owners, Boolean, {ctx, O3}, {WC}, {MC}](linkRef,
            ctx ?=> link =>
              // check whether the link is empty or not
              !link.isEmpty
          )
```

```
34
35          // use both `lf` and `linkHolder` linear variables
36          unlockHolder(lf.getKey(), linkHolder).consume()
37
38          // return the result
39          hasNext
40      )
41
42      // use both `lf` and `boxToLinkHolder` linear variables
43      unlockHolder(lf.getKey(), boxToLinkHolder).consume()
44
45      // return the result
46      hasNext
47  )
```

The `hasNextMI` function uses the same type parameters as `iterMut`. It does not modify or move any boxes and does not borrow any reference or box mutably. Therefore, it does not require annotating `WC` or `MC` with `@caps.use`.

The function takes an immutable reference to the iterator. It accesses the iterator by reading the reference argument and then borrows the box that holds the mutable reference to a link in the list. Next, it reads through the borrowed reference and the mutable reference to access the link and checks whether the link is empty. Because the reference to the link is mutable and the function avoids using the `write` function, which would capture `WC`, it first borrows the mutable reference immutably and then accesses the link.

The implementation of the `nextMI` function is a bit tricky:

```
1  /**
2    * @param O1 The list to be iterated owner set
3    * @param O2 The mutable iterator owner set
4    * @param O3 The reference to the mutable iterator (`self`) owner set
5    */
6  def nextMI[T <: scinear.Linear, @caps.use O1^, O3^, @caps.use O2^ >: {O1,
       O3}, @caps.use WC^, MC^](
7    self: MutRef[MutableIterator[T, O1, O2], O3]^
8  )(
9    using ctx: Context[WC, MC]^{O3}
10 ): Option[MutRef[T, O2]] =
11   // access the `self`'s reference resource, which is the mutable
       iterator
12   write[MutableIterator[T, O1, O2], {O3}, Option[MutRef[T, O2]], {O2}, {
       WC}, {MC}](self,
13     ctx ?=> iter =>
14       val boxToLink = iter.boxToLink
15
16       // create a dummy box pointing to a mutable reference that points
       to a link that points to `None` for swapping with `boxToLink`:
17       val dummyBoxToLink =
18         // create the box pointing to a link that is `None`
19         val dummyLinkBox = newBox[Link[T, O1], O2](None)
```

```
20        // borrow the box pointing to a link
21        val (dummyLinkMutRef, dummyBoxHolder) = borrowMutBox[Link[T, O1],
     O2, {ctx}, NeverUsableKey, {O2}, {WC}, {MC}](dummyLinkBox)
22        // use `dummyBoxHolder` by consuming because it is a linear
    variable that is not needed
23        dummyBoxHolder.consume()
24        // create and return the box pointing to the mutable reference to
     the link, which is `None`
25        newBox[MutRef[Link[T, O1], O2], O2](dummyLinkMutRef)
26
27      // swap the iterators' box to link with a dummy box pointing to `
    None`, so that the function can access the box's value without:
28      // - expiring the iterators' box
29      // - being forced to define a new lifetime for borrowing that will
    appear in every subsequent reference borrowed
30      //   and making it incompatible with the function's return type
31      val (iterBoxToLink, tempBoxToLink) = swapBox[MutRef[Link[T, O1], O2
    ], {O2}, {O2}, {ctx}, {WC}, {MC}](boxToLink, dummyBoxToLink)
32
33      // borrow the `tempBoxToLink` mutably
34      val (refToLink, tempBoxToLinkHolder) = borrowMutBox[MutRef[Link[T,
    O1], O2], O2, {ctx}, NeverUsableKey, {O2}, {WC}, {MC}](tempBoxToLink)
35      // use `tempBoxToLinkHolder` by consuming because it is a linear
    variable that is no longer needed
36      tempBoxToLinkHolder.consume()
37      // access the `refToLink`'s resource, which is the mutable
    reference to the link pointing to the next node to be visited
38      writeWithLinearArg[MutRef[Link[T, O1], O2], O2, Option[MutRef[T, O2
    ]], {ctx}, Box[MutRef[Link[T, O1], O2], O2], {WC}, {MC}](refToLink,
39        iterBoxToLink,
40        ctx ?=> (linkMutRef, iterBoxToLink) =>
41          // access the `linkMutRef`'s resource, which is a link to the
    next node to be visited
42          writeWithLinearArg[Link[T, O1], O2, Option[MutRef[T, O2]], {ctx
    }, Box[MutRef[Link[T, O1], O2], O2], {WC}, {MC}](linkMutRef,
43            iterBoxToLink,
44            ctx ?=> (link, iterBoxToLink) =>
45              // without consuming the option, check whether `link` is
    empty or not
46              val (link2, isEmpty) = scinear.utils.peekLinearOption(link)
47
48              // the behavior changes based on whether there is a next
    node or not
49              if isEmpty then
50                // converge if branches by consuming `link2`
51                link2.isEmpty
52                // leave iterator's box to link unchanged, it is now
    pointing to `None`, which is correct
53                iterBoxToLink.consume()
54                // return None, no next element
55                None
```

```
56              else
57                // get the box pointing to the next node to be visited
58                val nodeBox = link2.get
59                // borrow the next node to be visited mutably
60                val (nodeRef, nodeBoxHolder) = borrowMutBox[Node[T, O1],
    O1, {ctx}, NeverUsableKey, {O2}, {WC}, {MC}](nodeBox)
61                // use `nodeBoxHolder` by consuming because it is a
    linear variable that is not needed
62                nodeBoxHolder.consume()
63                // access the `nodeRef`'s resource, which is the next
    node to be visited
64                val elemMutRef = writeWithLinearArg[Node[T, O1], {O2},
    MutRef[T, O2], {ctx}, Box[MutRef[Link[T, O1], O2], O2], {WC}, {MC}](
    nodeRef,
65                  iterBoxToLink,
66                  ctx ?=> (node, iterBoxToLink) =>
67                    // decompose the node to get its element and next
    link
68                    val (nodeElem, nodeNext) = Node.unapply(node)
69
70                    // borrow the box pointing to the link pointing to
    the next node mutably
71                    val (nextLinkMutRef, nextLinkBoxHolder) =
    borrowMutBox[Link[T, O1], O1, {ctx}, NeverUsableKey, {O2}, {WC}, {MC
    }](nodeNext)
72                    // use `nextLinkBoxHolder` by consuming because it is
     a linear variable that is not needed
73                    nextLinkBoxHolder.consume()
74
75                    // update the iterator's box to link to point to the
    next node's link
76                    setBox(iterBoxToLink, nextLinkMutRef)
77
78                    // borrow the node's element mutably
79                    val (nodeElemMutRef, nodeElemHolder) = borrowMutBox[T
    , O1, {ctx}, NeverUsableKey, {O2}, {WC}, {MC}](nodeElem)
80                    // use `nodeElemHolder` by consuming because it is a
    linear variable that is not needed
81                    nodeElemHolder.consume()
82
83                    // return the mutable reference to the node's element
84                    nodeElemMutRef
85                )
86
87                // return the mutable reference to the next element
88                Some(elemMutRef)
89            )
90      )
91  )
```

The `nextMI` function uses the same type parameters as `iterMut` and `hasNext`. It

borrows several boxes mutably, so the function captures `WC`. The function reflects this mutable borrowing capability in its signature by annotating the `WC^` type parameter with `@caps.use`.

The function returns a mutable reference to the current element and, at the same time, updates the iterator to point to the next node. To achieve this, the `nextMI` function performs the following steps:

1. Access the current node visited by the iterator.
2. Mutably borrow the node's box that contains the element.
3. Mutably borrow the node's box that contains the link to the next node.
4. Update the iterator's box to hold the newly borrowed mutable reference that points to the link of the next node.

Figure 5.4 illustrates the iterator structure and the operations required for the `nextMI` operation.

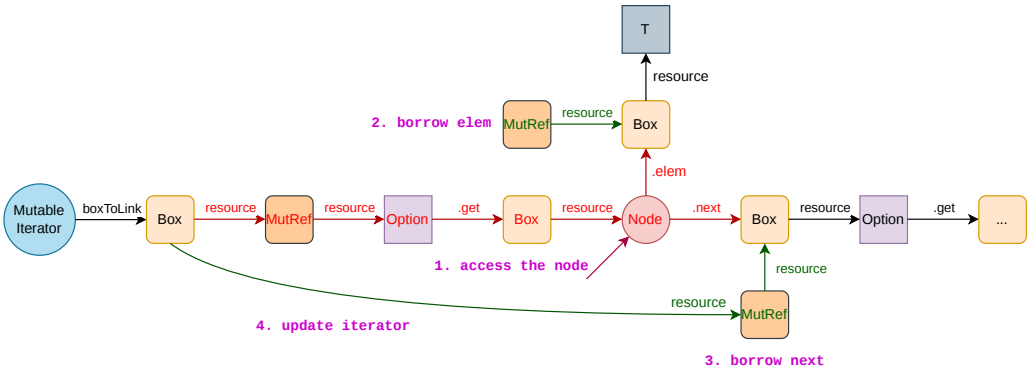

Figure 5.4: Overview of Iterating the List

Operations 2 and 3 involve borrowing new mutable references, `nodeElemMutRef` and `nextLinkMutRef`, from the fields of the node reached by the iterator's box. Operation 4 then updates the iterator's box, `iter.boxToLink`, which the function must access in order to reach the current node. As a result, the function must both access the resource held by the iterator's box, which requires borrowing it, and update that resource using the `write` function.

Borrowing a box's resource, or a box reachable from that resource, and then updating the same box, which is `iter.boxToLink` in this case, would expire the borrowed references due to the reaching properties (discussed in section 4.2). To avoid this issue, the `nextMI` function introduces a temporary box, `tempBoxToLink`, to temporarily hold the resource of

the iterator's box, `iter.boxToLink`. The function swaps the resources of `iter.boxToLink` and `tempBoxToLink`, which leaves the iterator's box pointing to `None`. It then accesses the resource in `tempBoxToLink`, accesses the node if it exists, and borrows the box containing the node's element as well as the box containing the link to the next node. Once both required references are available, namely the mutable reference to the current element and the mutable reference to the link to the next node, the function updates the iterator's box with the latter and returns the former.

5.4 imem in practice

This section explains how not following imem rules results in compile errors. This section also demonstrates the practicality of imem.

5.4.1 Linked List Implementation

The linked list implementation, described in section 5.3, follows the imem static rules. This subsection examines two selected parts of the implementation and demonstrates the errors that would result from modifying them.

5.4.1.1 Mutable Iterator Reference Owner Set

In the `iterMut` function, the type parameter `O2^` is statically bounded as a superset of the `O1^` and `O3^` type parameters, `O2 >: O1, O3`. In this function, `O2` denotes the iterator and its internal reference owner set, `O1` denotes the list owner set, and `O3` denotes the context's aggregated owner set. The following shows the `iterMut` function signature:

```
def iterMut[T <: scinear.Linear, @caps.use O1^, O3^, O2^ >: {O1, O3}, @
    caps.use WC^, MC^](
  self: MutRef[List[T, O1], O2]
)(
  using ctx: Context[WC, MC]^{O3}
): MutableIterator[T, O1, O2] = ...
```

The `self` argument is a mutable reference with `O2` as its owner set and `O1` as the owner set of the reference's resource, `List[T, O1]`. In addition, `O3` is the capture set of the capturing type `Context[WC, MC]^O3`, which represents the context.

The intention behind this relation is that the mutable iterator needs to borrow fresh mutable references from the list throughout the iteration. Moreover, the type of the internal reference, `MutableIterator.boxToLink`, namely `Box[MutRef[Link[T, O1], O2], O2]`, remains unchanged during iteration. This property means that every freshly borrowed reference must have the same `O2` owner set.

Based on imem's borrow checking rules, in each borrow, the resulting reference owner set must be a superset of the owner set of the borrowed box or reference and the owners aggregated by the context, which results in the invariant `O2^ >: O1, O3`. But, what happens if the `iterMut` function does not explicitly state this type bound? The following shows the function signature without this bound:

```
def iterMut[T <: scinear.Linear, @caps.use O1^, O3^, O2^, @caps.use WC^,
    MC^](
  self: MutRef[List[T, O1], O2]
)(
  using ctx: Context[WC, MC]^{O3}
): MutableIterator[T, O1, O2] = ...
```

And the following is the resulting compilation error:

```
[error] 672 |      val (headRef, listHeadHolder) = borrowMutBox[Link[T,
    O1], O1, {ctx}, NeverUsableKey, {O2}, {WC}, {MC}](list.head)
[error]     |
                                    ^
[error]     |Type argument scala.caps.CapSet^{O2} does not conform to
    lower bound scala.caps.CapSet^{ctx, O1}
[error]     |- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    - - - - - - - -
[error]     | Explanation (enabled by `-explain`)
[error]     |- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    - - - - - - - -
[error]     | ...
[error]     |     ==> subcaptures {ctx, O1} <:< {O2} in open varState
[error]     | ...
[error]     | The tests were made under the empty constraint
```

This is a capture-checking trace, which is typically a long list of unsuccessful attempts of the compiler to match inferred and annotated capture sets, that highlights the first borrow that depends on the invariant `O2^ >: O1, O3`. This borrow occurs during the creation of the mutable iterator and reports that `subcaptures ctx, O1 <:< O2` in `open varState` does not hold, as `borrowMutBox` signature requires it.

5.4.1.2 Write and Move Capability Annotations

All the functions in the linked list implementation using imem get `WC^` and `MC^` type parameters to instantiate the context instance (`Context[WC, MC]`) type parameters they are getting. `WC^` and `MC^` are type parameters instantiated by the write and move capability, respectively.

All the functions that include operations that require write access or move access annotate the `WC^` or `MC^` type parameter using `@caps.use` annotation. The following is an example of this annotation in the `pop` function:

```
def pop[..., @caps.use WC^, @caps.use MC^](...)(
  using ctx: Context[WC, MC]^
): ... = ...
```

The reason for this behavior is that the `pop` function both updates `list.head` and then moves the popped element. If the `pop` function did not annotate `WC^` and `MC^` with `@cap.use`, as shown below:

```
def pop[..., WC^, MC^](...)(
  using ctx: Context[WC, MC]^
): ... = ...
```

then the following would be the compilation error:

```
[error]     | ...
[error] 193 |      val (listRef, selfHolder) = borrowMut[List[T, O1], O2,
    {ctx, O2}, lf.Key, lf.Owners, {WC}, {MC}](self2)
[error]     |                                  ^^
[error]     |  Capture set parameter WC leaks into capture scope of
    method pop.
[error]     |  To allow this, the type WC should be declared with a @use
    annotation
[error]     | ...
[error] 267 |    derefForMoving[Link[T, O1], O3, {ctx}, Option[Box[T, O3
    ]], {WC}, {MC}]
[error]     |
              ^^
[error]     |  Capture set parameter MC leaks into capture scope of
    method pop.
[error]     |  To allow this, the type MC should be declared with a @use
    annotation
[error]     | ...
```

These errors occur at every point where the function uses imem interfaces or other functions that annotate `WC` or `MC` with `@cap.use`. Because the function does not explicitly state that it captures `WC` or `MC`, the error informs the programmer that `WC` and `MC` are leaking into the function's capture set.

5.4.2 A Familiar Example

The Rust documentation book[2] [18], in Chapter 10, Section 3, explains how to correctly annotate the `longest` function. This function takes two string references and returns a reference to the longer string:

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() { x } else { y }
}
```

[2]https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html

Implementing the function as above would result in the compile error below:

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:9:33
  |
9 | fn longest(x: &str, y: &str) -> &str {
  |               ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the
    signature does not say whether it is borrowed from `x` or `y`
help: consider introducing a named lifetime parameter
  |
9 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
  |           ++++     ++          ++          ++
```

Now let's implement the same function using imem. The following implements the `longest` function using the imem library:

```
def longest[O4^, O1^ >: {O4}, O2^ >: {O4}, O3^ >: {O1, O2}, WC^, MC^](
  a: ImmutRef[String, O1],
  b: ImmutRef[String, O2]
)(using ctx: Context[WC, MC]^{O4}): ImmutRef[String, O3] =
  val isALonger = read[String, O1, Boolean, O4, WC, MC](a, aVal =>
    read[String, O2, Boolean, O1, WC, MC](b, bVal =>
      aVal.length > bVal.length
    )
  )
  if isALonger then
    borrowImmut[String, O1, O4, NeverUsableKey, O3, WC, MC](a)
  else
    borrowImmut[String, O2, O4, NeverUsableKey, O3, WC, MC](b)
```

Similar to the Rust version, in which all references are annotated with the `'a` lifetime to inform the borrow checker that both arguments must outlive the result, this version uses `O3^ >: O1, O2` to inform the type checker that the returned reference expires earlier than both the `a` and `b` references. If the function does not explicitly state that `O3^` is a superset of `O1` or `O2`, one of the following errors occurs, respectively:

```
[error] 17 |    borrowImmut[String, O1, O4, NeverUsableKey, O3, WC, MC](a
   )
[error]    |                                                          ^
[error]    | Type argument O3 does not conform to lower bound scala.caps.
   CapSet^{O1}
[error]    | ...
[error] 22 |    borrowImmut[String, O2, O4, NeverUsableKey, O3, WC, MC](b
   )
[error]    |                                                          ^
[error]    | Type argument O3 does not conform to lower bound scala.caps.
   CapSet^{O2}
```

5.4.3 Using Iterators

The following examples show how imem enforces that the program has access to either the list or the mutable iterator, but not both at the same time, and that the list is no longer accessible after a consuming iterator is created.

```
// create an empty list with {ctx} as its lifetime
val list = newBoxFromBackground(newLinkedListFromBackground[LinearInt, {
    WC}, {MC}])

// a lifetime for the mutable iterator
val lf1 = Lifetime[{ctx}]()

// borrow the list mutably to use it for the mutable iterator
val (listRef, listHolder) = imem.borrowMutBox[List[LinearInt, {ctx}], {
    ctx}, {ctx}, lf1.Key, lf1.Owners, {WC}, {MC}](list)
// create the mutable iterator
val mutIter = iterMut[LinearInt, {ctx}, lf1.Owners, lf1.Owners, {WC}, {MC
    }](listRef)
// don't need `iter` anymore, so consume it
mutIter.consume()

// get the list back
val list2 = unlockHolder(lf1.getKey(), listHolder)

// create a consuming iterator from the list
val consumingIter = intoIter[LinearInt, {ctx}, {ctx}, {WC}, {MC}](list2)
// wrap it in a box
val iterBox = newBox[ConsumingIterator[LinearInt, {ctx}], {ctx}](
    consumingIter)
// don't need `iterBox` anymore, so consume it
iterBox.consume()
```

The program begins by creating the list using the `newLinkedListFromBackground` and `newBoxFromBackground` functions, which create a `List` and then a `Box` for the list, respectively, with `ctx` as their lifetime set. Here, `ctx` plays a role similar to the `'static` lifetime in Rust and serves as the base lifetime that the owner set of every newly borrowed reference must be a superset of. The program then mutably borrows the list to create a mutable reference. Because `list` is a linear variable, it is no longer accessible after the borrow. The program below attempts to use the list after borrowing it:

```
...
// borrow the list mutably to use it for the mutable iterator
val (listRef, listHolder) = imem.borrowMutBox[List[LinearInt, {ctx}], {
    ctx}, {ctx}, lf1.Key, lf1.Owners, {WC}, {MC}](list)
list
...
```

The program above triggers the following compile error:

```
...
```

```
[error] 36 |  list
[error]    |  ^^^^
[error]    |  Linear value list is being used twice, or is not accessed
    directly.
...
```

This example shows that the program can regain access to the list only by unlocking `listHolder`. To unlock the value holder, the program must use `lf1`'s key, which causes `lf1` and `mutIter` to expire. As a demonstration, the following program moves the use of `mutIter` to after unlocking `listHolder`:

```
...
// get the list back
val list2 = unlockHolder(lf1.getKey(), listHolder)
// don't need 'iter' anymore, so consume it
mutIter.consume()
...
```

The above code results in the compilation error below:

```
...
[error] 43 |  mutIter.consume()
[error]    |  ^^^^^^^
[error]    |Linear value lf1 is mentioned in the type of mutIter but is
    either not in scope or used already.
...
```

A similar argument applies to `list2` and `consumingIter`. Because `list2` is linear, the program can no longer access `list2` after creating `consumingIter`.

5.4.4 More Details Examples

The should work[3] and should not work[4] tests in the imem implementation provide more detailed examples of using all imem's functions, as well as the linked list implementation built on imem. The `ShouldWorkSuite` contains use cases of imem, the linked list, and their expected behavior. The `ShouldNotWorkSuite` contains functions that intentionally produce compilation errors, which illustrate violations of imem's rules.

5.5 Discussion

This section compares the performance of imem with other baselines in the case study of implementing a statically memory-managed safe linked list.

[3]https://github.com/amsen20/imem/blob/main/src/test/scala/ShouldWorkSuite.scala
[4]https://github.com/amsen20/imem/blob/main/src/test/scala/ShouldNotWorkSuite.scala

5.5.1 Functionality

Implementation	push & pop	peek	peekMut	Consuming Iterator	Mutable Iterator	Static Memory Management	Static Mutability Control
Rust	✓	✓	✓	✓	✓	✓	✓
Vanilla Scala	✓	✓	✓	✓	✓	×	×
Linear Scala	✓	×	×	✓	×	✓	×
imem	✓	✓	✓	✓	✓	✓	✓

Table 5.1: Comparison of Implementations

Rust enforces ownership and borrow checking, resulting in static memory management and static mutability control. As a result, the linked list can implement all required functionalities, including `push`, `pop`, `peek`, `peekMut`, a consuming iterator, and a mutable iterator.

Vanilla Scala uses garbage collection for references and does not provide static memory management. Also, vanilla Scala has no definition of mutability through a reference. This means that enabling `peekMut` and a mutable iterator requires returning a `Node` that exposes the internal structure, which the implementation should avoid if possible.

Linear Scala enforces linearity, through the Scinear plugin, and linearity enables statically managing memory. However, linear memory does not support mutability. Therefore, mutable features such as `peekMut` and a mutable iterator are not possible, and static mutability control is also not supported. In addition, immutable peeking is not possible. This limitation exists because pure linearity does not support borrowing. In a purely linear memory model, when the program gains access to a part of the memory, that memory becomes inaccessible from any other location. Otherwise, such access would violate the linear memory tree structure.

imem combines linearity from Scinear with capture checking and dependent types that are already part of the Scala type system. Together, these features enable borrow checking and ownership tracking. As a result, similar to Rust, imem supports full functionality, including static memory management and static mutability control.

5.5.2 Verbosity

Both the Rust and the vanilla Scala implementations are short and concise, whereas the Linear Scala implementation is slightly longer, and the imem implementation is significantly longer. Linear Scala follows a pattern in which a linear value is decomposed into its fields and then reconstructed so that no field is lost when accessing others. As a result, the code contains many `match` and `case` patterns. A similar linear access pattern appears in the imem code because resources are also linear values.

Rust is the only baseline that supports all features alongside imem. The following are some of the main differences between imem and Rust that make an imem program longer:

- **Rust's `take()`, `as_ref()`, and `as_mut()` methods:** These methods belong to `Option<T>` and simplify interaction with Rust's borrow checker for this enum. In contrast, imem requires the use of borrowing interfaces to borrow the box that points to `Option[Box[T]]`, and then the box that the option points to. The program must also use `read` and `write` operations to access the underlying resources.

- **Borrowing non-box values:** In Rust, the `head` field in `List`, as well as the `elem` and `next` fields in `Node`, are not wrapped in `Box<T>`. However, the program is able to have borrow-checked mutable and immutable references to these fields. In contrast, imem allows mutable and immutable references only to the resources inside boxes. As a result, the program has to wrap all values in boxes and access them layer by layer using `read` and `write` functions.

- **Language support for borrowing and dereferencing:** For example, the expression `&node.elem` in Rust first dereferences `node`, then accesses the `elem` field, and finally creates a reference to it. Since imem does not provide built-in language support for such operations, dereferencing must happen using continuation-passing-style interfaces.

The future works section (section 7.2) explains how small improvements can fill these gaps.

5.5.3 Error Clarity

Section 5.4.2 examines the implementations of the `longest` function in both Rust and imem. The incorrect Rust program produces the following complete error message:

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:9:33
  |
9 | fn longest(x: &str, y: &str) -> &str {
  |               ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the
    signature does not say whether it is borrowed from `x` or `y`
help: consider introducing a named lifetime parameter
  |
9 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
  |           ++++     ++          ++          ++
```

For comparison, the following is the error message for the same mistake made in Scala using imem:

```
[error]    | ...
[error]    | I tried to show that
[error]    |   imem.ImmutRef[String, 01] | imem.ImmutRef[String, 02]
[error]    | conforms to
```

```
[error]     |   imem.ImmutRef[String, O3]
[error]     | but none of the attempts shown below succeeded:
[error]     |
[error]     |   ==> imem.ImmutRef[String, O1] | imem.ImmutRef[String, O2]
      <:  imem.ImmutRef[String, O3]
[error]     |     ==> imem.ImmutRef[String, O1]  <:  imem.ImmutRef[String,
     O3]
[error]     |       ==> O3  <:  O1
[error]     |         ==> subcaptures {O3} <:< {O1} in open varState
[error]     |           ==> {O1} accountsFor O3  = false
[error]     |           ==> {O1} accountsFor cap  = false
[error]     |
```

The imem version is less understandable and more difficult to debug. In fact, the imem error is not a borrow-checking error. Instead, it explains the type checker's attempts to satisfy the capture-checking rules under the given constraints. Currently, I am not aware of a proper solution to address this problem without adding language support for imem.

Chapter 6

Related Works

6.1 Introduction

This chapter discusses the projects and research that inspire and relate to imem. First, it describes the main inspirations for imem and outlines the evolution of static memory management and static mutability control from Cyclone to Rust. Next, it compares imem with existing approaches to static memory management and related problems in Scala. Finally, it presents several recent and related works outside the Scala ecosystem and highlights their main differences from imem.

6.2 Cyclone, Vault and Rust

This section describes the key features, the main approach, and examples of three chosen well-known projects that target static memory management and static mutability control. These projects are discussed in chronological order.

6.2.1 Cyclone

Cyclone [9] is a type-safe programming language that is based on C. Cyclone provides region-based static memory management through its static type system. The memory of a Cyclone program is divided into regions. These regions are of three kinds:

- A single heap region that lives forever.
- Stack regions that correspond to stack frames containing local declarations.
- Dynamic regions that the program defines, whose lifetime is bounded by lexical scope.

Each value in memory resides in a region, and every pointer type is annotated with the region to which the pointer refers.

Cyclone allows functions and structures to be region-polymorphic. Therefore, the system is expressive and duplication is reduced. In addition, for expressiveness, Cyclone supports sub-typing. A pointer to one region can be used as a pointer to a different region if the pointer's region outlives the target region.

Cyclone prevents dangling pointers by disallowing the dereferencing of pointers whose associated region is no longer live. To prevent buffer overflow, Cyclone provides fat pointers that augment pointers with runtime bounds information. For static mutability, it enforces stricter rules than C on `const` pointers within its type system. Furthermore, to reduce manual region annotations, Cyclone performs pattern-based implicit region annotation.

To provide an overview of Cyclone, the following shows the `longest` function implemented in Cyclone:

```
char?`r longest<`r>(const char?`r s1, const char?`r s2) {
    if (strlen(s1) > strlen(s2)) {
        return s1;
    } else {
        return s2;
    }
}
```

The function takes two pointers to strings and returns a pointer to the longer string. The pointers are fat pointers to a character, which are annotated with the region `r`, that is, `const char?'r`.

This function is memory safe because all arguments and the returned pointer are annotated with the same region. Therefore, the region of the return type must be a subtype of the regions of all arguments. This requirement means that both argument regions must outlive the region of the returned pointer.

6.2.2 Vault

Vault [7] is an experimental programming language that extends the linear type system to address the restrictive aliasing rules of linear programs. In linear programs, only one reference to a linear value is allowed, and any structure that contains a linear field must itself be linear. This restriction is limiting and contrasts with many common programming patterns, such as sharing handles to a storage resource where one handle at a time has write access, while all handles have read access. To address this issue, Vault introduces non-linear, shareable, immutable views of a linear value after its *adoption*. Furthermore, during *focus*, one of these non-linear views is granted limited linear, mutable access to the underlying linear value.

Vault introduces the following concepts:

- **Key:** A key represents the static name of the memory address of an object.
- **Capability:** A capability defines a mapping between a key and its current type.

The compiler maintains a set of capabilities for each scope.

After adoption, the program transfers ownership of a linear value, called the *adoptee*, to another linear value, called the *adopter*. After this transfer, the program no longer has linear access to the adoptee, and the adoptee's key is removed from the capabilities. Instead, the program has non-linear access to the adoptee through a reference with a guarded type. This guarded type informs the compiler that the reference can access the value only if the *adopter* key is present in the capabilities.

During focus, which is a scoped operation, the program gains linear access through a guarded reference. Within this scope, the key of the guarded reference is removed from the set of capabilities. As a result, other guarded references with the same key cannot obtain focus views within that scope.

6.2.3 Rust

Rust [12], which is explained in detail in section 2.6, is inspired by Cyclone, Vault, and many others. It provides static memory management and static mutability control. Rust enforces static memory management by applying ownership rules. Moreover, it achieves memory safety and static mutability control through borrow checking. In addition, Rust performs fully automated region inference aided by lifetime annotations. When automatic pattern-based lifetime inference is insufficient, it allows manual lifetime annotations to guide the analysis.

6.2.4 imem's Standpoint

imem is primarily inspired by Rust. In particular, the concepts of lifetimes, boxes, and immutable and mutable access are borrowed from Rust.

imem follows rules that are similar to Rust in many places. However, the main difference is that imem is not a new programming language. Instead, imem is a library, and it does not require dedicated language support to provide static, safe memory management and mutability control.

imem relies on existing features of the Scala type system, including capture checking and dependent types. In addition, it uses a well-known foundation, linear types, which Scinear, a minimal compiler plugin, provides.

Overall, imem shows that static, safe memory management can be achieved by combining existing type system features, instead of designing new compiler features and introducing additional complexity to the type system from scratch for this purpose.

6.3 Surrounding Projects in Scala

This section focuses on attempts in the Scala ecosystem that target static and safe memory management.

6.3.1 Scala Safe Zones

Scala SafeZones [8] implement region-based, or zone-based, static memory management in Scala. They operate in Scala Native and integrate well with the garbage collector.

In brief, zones are lexical scopes that provide escape-checked access to an instance of `SafeZone`. When the program allocates a new object, it can pass the zone instance so that the object is allocated within that zone. When the scope of the zone ends, the zone and all objects allocated in it are freed. Moreover, the allocation result captures the zone instance and is therefore also bound to the zone's scope, making access to the object statically safe.

Safe zones address a simplified version of the problem that imem targets. They do not model borrowing or ownership. All aliases to an object have the same mutability access, and they share the same lifetime. In addition, objects cannot be moved from one zone to another without copying them.

Safe zones use a `nativelib` memory pool to manage zone and object allocations at runtime, which could serve as inspiration for imem in native Scala memory management.

6.3.2 Region-based Off-heap Memory For Scala

The Region-based off-heap memory for Scala report [22] presents a Cyclone style region-based memory management approach for Scala on the JVM.

It implements three versions of region-based memory management:

- **Unchecked:** This is the most performant version, with no static or dynamic checks to prevent dangling pointers.
- **Dynamic:** The dynamic version performs runtime checks on memory accesses to ensure the addressed page exists.
- **Static:** This version uses literal-based singleton types to assign a unique type to each region. With macros, all field accesses of off-heap classes require an implicit instance of the corresponding region. This guarantees, at compile time, that no dangling references exist, although regions must still be explicitly opened and closed.

Similar to Safe Zones, and compared to imem, this work addresses a simpler problem. It does not model ownership or borrowing, and therefore does not provide static alias management or mutability control.

The implementation includes a JVM-based paginated memory allocation runtime. The results show that the unchecked version can match GC performance while using roughly half the memory. This runtime could also be used by imem to manage memory on the JVM.

6.3.3 System Capybara

System Capybara [25] is an ongoing project that extends capture checking to support ownership and borrowing, enabling static alias and mutability control.

The extension ensures that fresh, non-consuming capabilities are mutually exclusive within the capture set in which they are instantiated. For consuming arguments, it guarantees that after a function call, the program cannot refer to anything that captures the consumed capabilities. The system also introduces read-only references whose capture sets may safely share capabilities.

Capybara addresses the same overall goal as imem, which is Rust-style ownership and borrowing, but takes a fundamentally different approach. The goal of imem is to rely primarily on existing features of the Scala type system, capture checking, and dependent types, and add well-studied linear types, and combine them with capture checking. In contrast, Capybara extends capture checking directly to support ownership and borrowing.

As a result, Capybara does not explore the potential of combining two distinct systems, capture checking and linearity, to achieve ownership and alias control, features that neither system fully targets on its own.

6.4 Outside Scala

Static memory management, along with static control of aliasing and mutability, is a well-known research problem that has been addressed by many projects. Approaches that tackle these challenges, fully or partially, often also aim to support fearless concurrency and guarantee statically data-race-free programs. The following section highlights some of these approaches outside the Scala ecosystem and discusses their relation to imem.

Reachability Types [3] enable ownership-style reasoning in higher-order functional languages by introducing a type system that tracks potential aliasing variables within a value's type. The system associates reachability sets with types, providing a selective separation mechanism that controls how values may alias one another. This design supports move semantics, ownership, and safe parallelism, while imposing fewer restrictions than Rust's aliasing rules. Although this approach targets goals similar to those of imem, it introduces an entirely new type system. In that sense, it is closer to capture checking than to imem's approach, which aims to demonstrate that these goals are feasible within the scope of a library.

Affe [17] is an extension of the ML programming language that adds support for linearity and alias management. Affe annotates types with kinds (which can be linear, affine, or unrestricted), and allows borrowing of linear values in both exclusive mutable and shared immutable ways. It does not require heavy annotations for linearity and provides fully automatic type and region inference. Affe targets goals similar to those of imem, and its expressive yet easy-to-use approach to linearity could inspire Scinear. However, Affe introduces a new type system, making it harder to integrate with existing programs, compared to imem, which is a library.

MLKit [23] is a compiler for Standard ML that provides fully automatic region inference without imposing restrictions on programs. Its static region inference guarantees safe memory management, ensuring the absence of dangling references. However, this fully automatic and unrestricted approach can cause some regions to outlive their actual lifetimes, leading to growing regions and potential memory leaks. To address this issue, MLKit offers both safe and unsafe program-level APIs to check whether a region's memory can be reset. It also includes a minimal garbage collector that tracks regions. MLKit does not target alias or mutability control and does not provide ownership or borrowing mechanisms. OwnML [16] is an extension to the MLKit compiler that aims to invalidate programs with memory leaks. It does so by ensuring that some variables own their regions, enforcing separation from other values in their scope. This limits cases where the region inference algorithm would otherwise merge objects with different lifetimes, which can lead to memory leaks.

OxCaml [14] extends the OCaml type system with `modes`, each defined along three axes:

1. Affinity: either `once` (a value can be used at most once) or `many` (it can be used multiple times).
2. Uniqueness: either `unique` (the value has no aliases) or `aliased` (it is referenced from multiple places).
3. Locality: either `local` (the value cannot escape its lexical region) or `global` (it may safely escape).

Through Uniqueness, OxCaml provides static ownership, Affinity ensures that this ownership remains sound, and Locality enables static alias management by allowing immutable borrowing. Compared to imem, Locality corresponds to a subset of capture checking, specifically escape checking. Uniqueness aligns with imem ownership and borrowing rules enforced through the interface, and Affinity is similar to Scinear's linearity, extended to support closures. OxCaml supports a hybrid static and dynamic memory management, as well as fearless concurrency, offering ideas that, if followed, can lead imem into practicality. OxCaml implementation of Affinity can be a guide for extensions to Scinear. However, at the end of the day, OxCaml is a language extension. Although backward compatible, it

modifies and adds complexity to the OCaml type system. In contrast, imem is a library that operates within the existing type system rather than extending it.

Vale [15] is a native programming language that provides memory safety through generational references. The memory slots managed by the allocator are augmented with a generation number, which is initially zero for all slots. During program execution, each time a slot is freed and later reused for a new value, its generation number is incremented. In Vale, references are fat pointers that contain both the memory address and the generation number of the slot at the time the reference is created. Whenever a reference is dereferenced, a runtime check verifies that the generation number stored in the reference matches the current generation of the slot. This check dynamically prevents use-after-free memory violations. Furthermore, Vale also supports linear and region-based patterns to reduce the number of runtime checks. Although Vale is a new programming language, it addresses problems similar to those targeted by imem. Its dynamic approach to memory safety, which does not rely on garbage collection, can therefore inspire solutions for managing non-statically managed memory in Scala native programs that use imem.

Chapter 7

Conclusion

7.1 Introduction

This thesis outlined Scinear, imem, and a case study that demonstrates imem capabilities.

The thesis showed that adding linear types to the Scala type system, and combining linearity with polymorphism and capture checking, lays the foundation for a library that enforces ownership and borrowing rules statically. This approach avoids extending the type system with features that target ownership and borrowing as a special case. Therefore, enabling Rust-style static memory management and alias control in Scala does not require foundational changes to the type system, and it can be implemented as a library.

To support this goal, the thesis explains the Scinear plugin rules for Scala, how Scinear mixes linearity with polymorphism and capture checking, and how Scinear is implemented. Furthermore, the thesis discusses the imem formal memory model, how imem uses the Scala type system alongside Scinear to enforce ownership and borrowing rules as a library, and the limitations of this approach. Finally, the thesis demonstrates imem by implementing a safe linked list. A comparison with Rust, vanilla Scala, and linear Scala shows that imem is as expressive as Rust, but imem needs additional features to reduce verbosity.

The remaining work that can make imem more capable and easier to use is discussed in the next section.

7.2 Future Works

The imem formal foundation includes a memory model definition and a well-formedness condition. However, imem soundness is not proved. A formal soundness proof for imem should start from a minimal foundation that represents Scala extended with the Scinear plugin. This foundation can be the $CC_{<:\square}$ calculus, described in Capturing Types [5], extended with linear types. Alternatively, it can be a typed lambda calculus that includes

linear types in its type system and includes escape checking and capture set type parameters as well-defined features. In addition, the proof should target imem correctness for safe static memory management, which implies no use-after-free and no double-free. The proof also has to address alias and mutability control by proving mutual exclusion of mutable access to resources.

imem implementations tend to be long, and they can include loopholes. Both issues can be mitigated by using Scala meta programming, as described in the foundational paper [6] and in the Scala 3 guide[1], and by adding more helper functions to the imem interface. Scala macros can enforce common patterns. For example, macros can enforce the borrowing pattern that introduces a lifetime `Owner` and `Key` type members in borrowing-interface calls, which makes avoiding loopholes easier. In addition, small but useful operations such as writing, setting, and swapping a box resource directly, without repetitive borrowings, can significantly reduce duplication in codebases that use imem. Similarly, a dereferencing operation that is similar to `read`, but for a box, can further reduce boilerplate.

In addition, imem currently does not manage memory allocation at runtime. To address runtime allocation, the `nativelib` implementation in Scala Zones [8] paves the way for runtime memory management in the native target. Also, the Region-based off-heap memory for Scala report [22] indicates how `sun.misc.unsafe` can enable memory management in imem statically for JVM target.

[1] https://docs.scala-lang.org/scala3/guides/migration/compatibility-metaprogramming.html

References

[1] Nada Amin, Adriaan Moors, and Martin Odersky. Dependent object types. In *19th International Workshop on Foundations of Object-Oriented Languages*, 2012.

[2] Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 233–249. ACM, 2014.

[3] Yuyan Bao, Guannan Wei, Oliver Bracevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. Reachability types: tracking aliasing and separation in higher-order functional programs. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–32, 2021.

[4] Stephen M. Blackburn and Kathryn S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 22–32. ACM, 2008.

[5] Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondrej Lhoták, and Jonathan Immanuel Brachthäuser. Capturing Types. *ACM Trans. Program. Lang. Syst.*, 45(4):21:1–21:52, 2023.

[6] Eugene Burmako. Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala, SCALA@ECOOP 2013, Montpellier, France, July 2, 2013*, pages 3:1–3:10. ACM, 2013.

[7] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In Jens Knoop and Laurie J. Hendren, editors, *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 13–24. ACM, 2002.

[8] Safe Zone: Memory-safe zone ensured by capture checking and GC. by yawen-guan · Pull Request #3120 · scala-native/scala-native — github.com. https://github.com/scala-native/scala-native/pull/3120. [Accessed 23-03-2026].

[9] Dan Grossman, J. Gregory Morrisett, Trevor Jim, Michael W. Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In Jens Knoop and Laurie J. Hendren, editors, *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 282–293. ACM, 2002.

[10] A garbage collector for C and C++ — hboehm.info. https://www.hboehm.info/gc/. [Accessed 22-03-2026].

[11] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. Stacked borrows: an aliasing model for Rust. *Proc. ACM Program. Lang.*, 4(POPL):41:1–41:32, 2020.

[12] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, 2018.

[13] Ralf Jung, Benjamin Kimock, Christian Poveda, Eduardo Sánchez Muñoz, Oli Scherer, and Qian Wang. Miri: Practical Undefined Behavior Detection for Rust. *Proc. ACM Program. Lang.*, 10(POPL):1383–1411, 2026.

[14] Anton Lorenzen, Leo White, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. Oxidizing ocaml with modal memory management. *Proc. ACM Program. Lang.*, 8(ICFP):485–514, 2024.

[15] Evan Ovadia. The Vale Programming Language. https://vale.dev/, 2022. [Accessed 22-03-2026].

[16] Amirhossein Pashaeehir. GitHub - amsen20/OwnML: A compile-time memory region segregation tool — github.com. https://github.com/amsen20/{O}wn{M}{L}. [Accessed 23-03-2026].

[17] Gabriel Radanne, Hannes Saffrich, and Peter Thiemann. Kindly bent to free us. *Proc. ACM Program. Lang.*, 4(ICFP):103:1–103:29, 2020.

[18] The Rust Programming Language - The Rust Programming Language — doc.rust-lang.org. https://doc.rust-lang.org/book/title-page.html. [Accessed 23-03-2026].

[19] Introduction - Learning Rust With Entirely Too Many Linked Lists — rust-unofficial.github.io. https://rust-unofficial.github.io/too-many-lists/index.html. [Accessed 23-03-2026].

[20] Capture Checking — docs.scala-lang.org. https://docs.scala-lang.org/scala3/reference/experimental/cc.html. [Accessed 22-03-2026].

[21] Scala Native — Scala Native 0.5.11-20260224-d696e69-SNAPSHOT documentation — scala-native.org. https://scala-native.org/en/latest/index.html. [Accessed 23-03-2026].

[22] Denys Shabalin and Martin Odersky. Region-based off-heap memory for Scala. *EPFL, Tech. Rep*, 2015.

[23] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ml kit. *DIKU Rapport*, 97:12, 1997.

[24] Philip Wadler. Linear Types can Change the World! In Manfred Broy and Cliff B. Jones, editors, *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*, page 561. North-Holland, 1990.

[25] Yichen Xu, Oliver Bračevac, and Martin Odersky. System Capybara: Capture Tracking for Ownership and Borrowing (The Scala Workshop 2025) - The Scala Workshop 2025 — 2025.workshop.scala-lang.org. https://2025.workshop.scala-lang.org/details/scala-2025/6/{S}ystem-{C}apybara-{C}apture-{T}racking-for-{O}wnership-and-{B}orrowing, 2025. [Accessed 23-03-2026].