KØBENHAVNS UNIVERSITET

Master's Thesis

Programmer-Directed Region Management in ReML

Mikkel Willén (bmq419)

Supervisor: Martin Elsman

June 5, 2026
Revised version

Abstract

Region-based memory management, as implemented in the MLKit compiler for Standard ML, assigns each heap-allocated value to a region and reclaims memory through constant-time region deallocation. When the built-in garbage collector is disabled, programmers must cooperate with the region system to keep memory bounded, using reset primitives to reclaim region contents in long-running loops. However, the existing primitives can only reset all regions appearing in a value's type, cannot accept explicit region variables, and provide no mechanism for inspecting region state at runtime. This forces programmers into fixed-interval collection strategies and unsafe force-resets, with no way to make data-driven reclamation decisions from within the running program.

We present three contributions that address these limitations in ReML, a Standard ML extension with explicit region annotations. First, we extend the MLKit's compiler so that the reset primitives accept explicit region variables as additional arguments, and we extend the foreign function interface to pass explicit regions pointers to C functions. Second, we implement two SML modules that build on these extensions: a `Region` module that provides typed access to per-region metadata at runtime, and a `Size` combinator library for computing the memory footprint of Standard ML values as they are laid out in regions. Third, we apply these tools to a chat service that processes packets in a long-running loop with the built-in garbage collector disabled, demonstrating adaptive threshold-based garbage collection where the program decides when to reclaim memory based on measured region usage. The case study achieves bounded memory usage with runtime comparable to MLKit's built-in garbage collector.

Contents

1 Introduction **3**

2 Background **5**
2.1 Memory Management in Constrained Systems . . . 5
2.2 Standard ML and the MLKit Compiler . . . 6
2.3 Region-Based Memory Management . . . 9
2.4 Region Polymorphism and Annotated Types . . . 12
2.5 Region Inference and Effect System . . . 15
2.6 Storage Modes and the Runtime Memory Model . . . 17
2.7 The Region Reset Primitives . . . 19
2.8 ReML: Programming with Explicit Regions . . . 20
2.9 Related work . . . 22

3 Programming with Explicit Regions **24**
3.1 Design Goals and Motivating Example . . . 24
3.2 Extension 1: Explicit Region Arguments for Reset Primitives . . . 25
3.3 Extension 2: Explicit Region Arguments for C Calls . . . 34
3.4 Soundness Argument . . . 40
3.5 Limitations . . . 41

4 Modules for Region Operations **43**
4.1 Design Rationale . . . 43
4.2 The REGION Signature . . . 43
4.3 Implementation: Region.sml and the C runtime backing (Region.c) . . . 45
4.4 `Size` Combinator . . . 48

5 Testing **52**
5.1 Testing Framework . . . 52
5.2 Reset Primitives . . . 52
5.3 Region Module . . . 53
5.4 Size Module . . . 54
5.5 Limitations and Future Testing . . . 55

6 Case study (chat service) **56**
6.1 Reference Project and Motivation . . . 56
6.2 Architecture . . . 56
6.3 Manual Garbage Collection . . . 57
6.4 Attack Surface of `TcpState` . . . 58
6.5 Summary of Changes from the Reference Project . . . 59

7 Evaluation **60**
7.1 Memory Usage: Incremental improvements . . . 60
7.2 Runtime . . . 64
7.3 Comparison with Built-in Garbage Collection . . . 65
7.4 Discussion . . . 67
7.5 Threats to Validity . . . 68

8 Conclusion **69**
8.1 Future work . . . 69

A Declaration of Using Generative AI Tools **74**

1 Introduction

Programs that manage dynamic data structures must reclaim memory that is no longer needed. In general-purpose computing, garbage collection handles this automatically freeing the programmer from the burden of tracking object lifetimes. However, garbage collection introduces two properties that can be problematic in memory-constrained, long-running systems. These are unpredictable pause times and space overhead. Tracing collectors must periodically traverse the live object graph, pausing the application for a duration that depends on the heap size and collector algorithm [17]. Conventional collectors also require the heap to be a constant factor larger than the maximum live data set in order to amortise the cost of collection [14]. In systems that operate under tight memory budgets, such as unikernels whose memory is provisioned at boot, embedded controllers with limited RAM, or long-running daemons where memory consumption directly limits the number of clients that can be served, this multiplicative overhead may be unacceptable, and programmers have little influence over when collection occurs or which objects are reclaimed.

These constraints have motivated a range of language-level approaches to memory management that give programmers more control over when and how memory is reclaimed, including ownership systems [19], safe region dialects of C [13], scoped regions for mutable memory [20], and modal memory management [22]. Among these, region-based memory management, as developed by Tofte and Talpin [16, 6], takes an approach that is particularly attractive for the systems we consider. The compiler statically assigns each allocation to a region and inserts constant-time allocation and deallocation directives automatically, requiring no runtime traversal of the object graph.

In the Tofte-Talpin framework, a region is a container that can hold an arbitrary number of values. Regions are allocated and deallocated in a stack-like discipline, and every region operation (allocation, deallocation, and resetting) executes in constant time [6]. The MLKit compiler [23] implements this framework for Standard ML, using a region inference algorithm [7] to determine which region each value is placed into and where region lifetimes begin and end. ReML [21] extends Standard ML with source-level syntax for naming regions, placing values into them, and passing them as arguments to functions, giving programmer direct control over region placement while retaining the safety guarantees of the type-and-effect system.

While the original ambition behind region-based memory management was to replace garbage collection entirely [6], this alone turned out to be insufficient for some allocation patterns [12, 16]. Certain data structures, particularly those that are long-lived or escape their enclosing scope, end up in regions whose lifetimes span the entire program. To address this, the MLKit provides two built-in primitives, `resetRegions` and `forceResetting`, that allow programmers to request that a region's content be reclaimed without deallocating the region itself [23]. These primitives, combined with a technique known as double copying, enable bounded-memory programming in long-running loops. However, they have significant limitations that become acute in practice.

The existing reset primitives operate on the set of regions that appear free in the type of their argument value. Programmers cannot target a specific subset of those regions, cannot pass explicit region variables, and cannot reset a region independently of any particular value. If a value's type mentions five region variable but only one of them is growing problematically, there is no way to reset the problematic region without affecting the others. Furthermore, the storage mode analysis that guards `resetRegions` is conservative. It can reject resets that programmers knows to be safe, leaving `forceResetting` (which bypasses all safety checking) as the only alternative.

A second gap is the absence of runtime inspection. Programmers have no way to query, from SML, how much memory a region currently uses, how many pages it occupies, or whether it has been marked for resetting. The only ways to observe region behaviour is to compile with profiling enabled and analyse the output, or by inspecting the compiler's intermediate code to understand region assignments and storage modes. However, neither approach is available at runtime from within the program. This means that a long-running service cannot make data-driven decisions about when to reclaim memory. Programmers must either perform double copies at fixed internals, regardless of whether waste has accumulated, or guess when resetting is worthwhile. Kanne and Geisshirt [24]

identified threshold-based garbage collection as a natural improvement over fixed-interval collection in their work on unikernel protocol stacks, but could not implement it because no mechanism existed for inspecting region state at runtime. This thesis addresses both gaps through three contributions:

1. **Compiler extensions for explicit region arguments.** We extend the MLKit compiler so that `resetRegions` and `forceResetting` accept explicit region variables as additional arguments, and we extend the `prim` mechanism to pass explicit region pointers to C functions. These extensions allow programmers to target specific regions for resetting and to pass region pointers to runtime functions for inspection. The changes touch the `LambdaExp` intermediate language, the spreading phase, the storage mode analysis, and the drop-regions phase, but do not alter the region type system or its soundness properties.

2. **The Region module and Size combinator.** We design and implement two SML modules that build on the compiler extensions. The `Region` module provides typed access to per-region metadata, such as memory usage, page count, and allocation status, and a direct reset operations. The `Size` module is a combinator library that computes the memory footprint of an SML value as it is laid out in regions. Together, these modules enable programmers to compare live data size against total region usage and make informed decisions about when to reclaim memory.

3. **Case study and evaluation.** We apply the extensions and modules to a simulated chat service that reads packets from a file, reassembles them into messages, and processes them in a long-running loop without built-in garbage collector. The case study demonstrates adaptive, threshold-based garbage collection, where the program itself decides when to perform a double copy based on measured memory pressure, and shows that this approach achieves bounded memory usage with runtime comparable to the MLKit's built-in garbage collector.

The compiler extensions, modules, and case study code are available as a fork of the MLKit repository at: `https://github.com/mikkelwillen/mlkit/`.

The remainder of this thesis is organised as follows.

Chapter 2 provides the technical background necessary to follow the remainder of the thesis. It introduces memory management challenges in constrained systems, the Standard ML language and the MLKit compiler pipeline, and the theory and practice of region-based memory management, including region polymorphism, effect typing, storage modes, and the runtime memory model. The chapter also describes the existing reset primitives `resetRegions` and `forceResetting`. It concludes with an overview of ReML's explicit region syntax and a discussion of related work.

Chapter 3 describes the two compiler extensions that form the technical core of the thesis. For each extension, we walk through the changes at each stage of the compiler pipeline, illustrate the behaviour with examples and intermediate code, and argue that the extensions preserve the soundness of the region system.

Chapter 4 presents the `Region` module and the `Size` combinator, two SML modules that build on the compiler extensions from Chapter 3. The chapter covers the design rationale, the public API, the implementation in both SML and the C runtime, and the limitations of both modules.

Chapter 5 describes the test suites used to validate the compiler extensions and modules. the tests are unit-style correctness tests that verify compiler output, diagnostic messages and runtime behaviour across both single-threaded and parallel configurations.

Chapter 6 presents the case study. The chapter discusses the architecture, the manual garbage collection strategy, the use of the `Region` and `Size` modules to implement adaptive threshold-based collection, and the security implications of the assembler state.

Chapter 7 evaluates the contributions through a series of incremental improvements to the case study, showing how each technique reduces peak memory usage. The chapter includes memory profiling graphs, runtime measurements, and a comparison with the MLKit's built-in garbage collector.

Chapter 8 concludes with a summary of contributions, key takeaways, and directions for future work.

2 Background

2.1 Memory Management in Constrained Systems

Programs that use dynamic data structures like lists, trees, closures and references, must manage the memory in which those structures reside. The two dominant strategies, manual deallocation and automatic garbage collection (GC) are both well understood in the general case. In this section we focus on the specific difficulties each strategy poses when memory is scarce and timing requirements are strict, as is the case in embedded controllers, unikernel-based services and long-running network daemons.

Explicit allocation and deallocation give the programmer direct control over when memory is reclaimed. In a memory-constrained system this control is valuable. The programmer can ensure that buffers are freed immediately after use and that the working set never exceeds the available physical memory. The cost is the well-known class of errors, such as dangling pointers, double frees and space leaks, that become increasingly prevalent as programs grow more complex [3].

Garbage collection eliminates use-after-free, use-before-allocation and double-free errors by construction, which is a compelling advantage for software robustness. However, tracing collectors exhibit two properties that are problematic under tight resource budgets, namely unpredictable pause times and space overhead.

During some phases of a collection cycle, the application is paused while the collector traverses the live object graph. These pauses can range from sub-millisecond to several seconds, depending on the heap size and the collector algorithm [17]. For systems with real-time or near-real-time requirements the consequences can be severe. Network peers may declare a node unreachable, queues fill up, and deadlines are missed [17].

A tracing collector typically needs the heap to be a constant factor larger than the maximum live data set in order to amortize the cost of collection. Bacon et al. note that conventional collectors have typically required several times the maximum live data set size in order to run well [14]. On a device with very limited RAM, or in a unikernel whose memory is fixed at boot, and where memory usage can be a direct correlation with the number of clients that can be served, this multiplicative overhead may be unacceptable. Moreover, the collector's own metadata, like mark bits, forwarding pointers and remembered sets, consumes space that is unavailable to the application.

Even when a collector's throughput and space overhead are acceptable in aggregate, the programmer has little influence over when collection occurs or which objects are reclaimed. In a long-running service, whose allocation pattern varies over time, this means that memory usage can grow unpredictably between collection cycles, making it difficult to provide bounded-memory guarantees.

These difficulties have motivated a range of responses across programming language design. Rust [19] eliminates the need for a garbage collector through an ownership and borrowing system enforced at compile time, where each value has a single owner and is deallocated when the owner goes out of scope. Cyclone [13] extended C with safe, lexically scoped regions and a type system that prevents dangling pointers [11]. Flix [20] uses regions for scoped local mutable memory, where all mutable data belongs to a region and the effects associated with a region vanish when the region goes out of scope. OxCaml [22] extends OCaml with a `local` mode that prevents values from escaping their enclosing region, which allows the compiler to allocate them on the stack rather than the heap and reclaim them without involving the garbage collector, while remaining opt-in. Region-Based Memory Management, as developed by Tofte and Talpin [6], takes another approach. The compiler statically assigns each allocation to a region and inserts constant-time allocation and deallocation directives automatically. While the original ambition behind Region-Based Memory Management was to replace garbage collection [6], this alone turned out to be insufficient [12, 16]. Certain allocation patterns, particularly those involving long-lived global data structures, produce memory in regions whose lifetimes span the entire program.

Region-based memory management thus occupies a point in the design space between fully manual deallocation and fully automatic collection. Every region operation executes in constant time, so to the extent that the programmer can shift reclamation work from the garbage collector to the region

system, the unpredictable pauses discussed above are replaced by predictable bounded costs [23].

2.2 Standard ML and the MLKit Compiler

Standard ML is a statically typed, call-by-value functional programming language with parametric polymorphism, algebraic data types, pattern matching, higher-order functions, mutable references, exceptions, and a module system with support for structures, signatures, and functors [23]. The language is formally defined, and its type system guarantees that well-typed programs do not access memory in unsafe ways [23]. A comprehensive standard library, the Standard ML Basis [15], provides common data structures, I/O facilities, and system interfaces.

Several implementations of SML exist, including SML/NJ, Moscow ML, Poly/ML, and MLton. Among these, the MLKit is distinguished by its use of region-based memory management as the primary mechanism for reclaiming heap memory. The remainder of this section introduces the MLKit's architecture and compilation pipeline, which form the foundation for the compiler extension described in Chapter 3.

Overview and project history

The MLKit with Regions is a compiler infrastructure for Standard ML, developed at the Department of Computer Science at University of Copenhagen (DIKU) with development beginning in 1992 [23]. The stated goal of the project has been to combine the advantages of a high-level programming language with a computational model that allows programmers to reason about how much space and time their program use [23]. The compiler supports all of Standard ML, including modules, and implements most of the Standard ML Basis Library [23]. MLKit is hosted on GitHub [1] and is distributed under a combination of the GNU GPL (compiler) and MIT (runtime system and Basis Library) licenses.

The MLKit provides two backends. The native backend generates x64 machine code for Linux and macOS and uses region-based memory management as its primary memory-management discipline. A second backend, SMLtoJs, compiles Standard ML to JavaScript for execution in web browsers, though it does not use region-based memory management and instead relies on the JavaScript engine's garbage collector [18]. In this thesis we are concerned exclusively with the native backend.

Compilation pipeline

The native backend compiles Standard ML source programs through a sequence of typed intermediate languages and analyses [23]. Figure 1 summarizes the major phases. We describe them here insofar as they are relevant to our compiler extensions.

The source program is first elaborated into *LambdaExp*, a lambda-calculus-like intermediate language in which pattern matching has been compiled away and functions may take multiple arguments [23]. Module-language constructs, like structures, signatures and functors, are eliminated during elaboration and do not appear in subsequent phases. Consequently, modules have no runtime overhead [8]. The LambdaExp term is then rewritten by an optimiser that performs inlining, specialisation, and dead-code elimination, subject to the constraint that rewrites must not increase space usage [23].

The optimised LambdaExp term is the input to region inference, the central analysis of the compiler. Region inference transforms LambdaExp into *RegionExp*, the same language extended with explicit region annotations. Every value-producing expression is annotated with the region into which its result will be placed, and `letregion` bindings delimit region lifetimes. A preparatory pass called spreading, first introduces region variables by annotating types with places and effects. Region inference then determines the actual bindings and allocation sites [23].

After region inference, *multiplicity inference* annotates each binding region variable with a *multiplicity*, which is a statically determined upper bound on the number of times a value is stored into the region. The three possible multiplicities are 0, 1, and ∞. A region with multiplicity less than

[1] `https://github.com/melsman/mlkit`

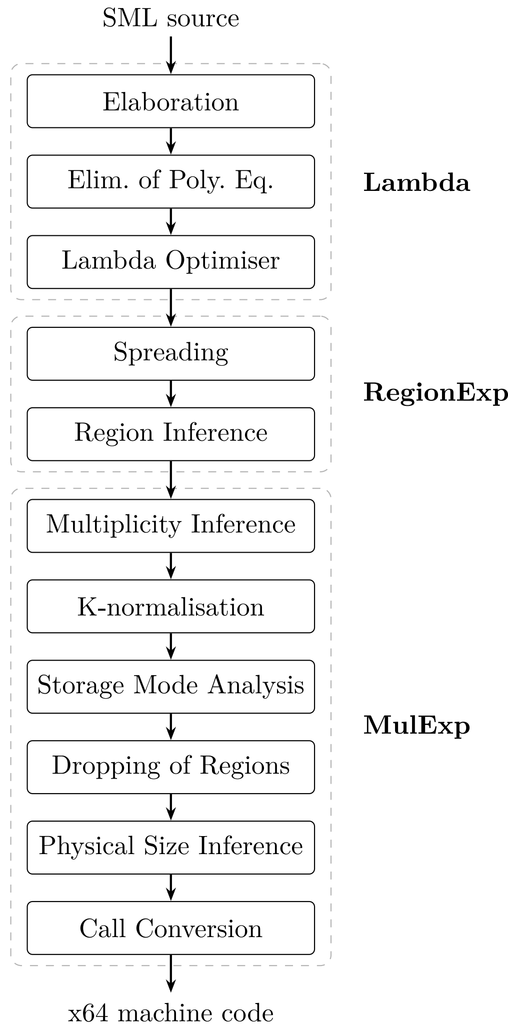

Figure 1: The MLKit native compilation pipeline. Each solid box represents a transformation (function) from one program representation to another. Arrows denote function composition, with the output of one transformation passed as input to the next. SML source and x64 machine code are the input and output of the pipeline respectively. Dashed boxes group transformations that operate on the same intermediate language, named to their right. The first transformation in each group takes a program in the language of the previous group as input and produces a program in the language of the current group. Adapted from the MLKit reference manual [23].

∞ can be allocated as a finite region directly on the runtime stack. Regions with multiplicity ∞ are allocated as infinite regions, represented as linked lists of fixed-size pages drawn from a free list [23, 5]. The resulting intermediate language is called *MulExp*.

Several further analyses refine the MulExp term. Storage mode analysis determines, for each allocation point, whether a value is stored at the top of a region (`attop`) or whether the region can first be reset to reclaim earlier contents (`atbot`) [23, 5]. *Dropping of regions* removes region parameters from functions when those parameters carry only `get` effect, i.e. the function only reads from, but never writes to, the region. *Physical size inference* determines the compile-time size, in words, of finite regions. Finally, *call conversion* transforms the program into a call-explicit form suitable for machine code generation [23].

The entire pipeline involves more than 25 passes over three typed intermediate languages [23]. An important invariant is that, with the exception of exception handling, every region directive inserted by the compiler executes in constant time and constant space [23].

The runtime system

The MLKit runtime system is written in C and is small, less than 30Kb of compiled code [23]. It provides operations for allocating and deallocating regions, extending regions when they run out of page space, managing the free list of region pages, recording region-profiling information (when profiling is enabled), and implementing low-level operations required by the Standard ML Basis Library.

The address space is partitioned into a stack and a heap. Finite regions and region descriptors for infinite regions are kept on the stack. The heap consists of fixed-size region pages that are managed through a global free list. When an infinite region is allocated, a page is obtained from the free list and a region descriptor, containing pointers to the first and last page in the region's linked list, is pushed onto the stack. When an infinite region is deallocated, its pages are spliced back onto the free list in constant time, and the region descriptor is popped [23].

Every region is identified at runtime by a 64-bit region name. For finite regions this is a pointer into the stack. For infinite regions it is a pointer to the region descriptor. Each region also has a runtime type, that describes the layout of the values it contains [23].

Value representations

The MLKit distinguishes between *boxed* and *unboxed* values. Unboxed values, integers, booleans, words, characters and the unit value, are stored directly in registers or machine words and do not reside in regions. Boxed value, pairs, records, reals, function closures, exception values, and most constructed values, are represented as pointers into regions. Lists are represented with an unboxed tag in the least significant bits of the pointer, so a list occupies one region for its auxiliary cons-cell pairs plus whatever regions its elements require [23]. This representation matters for the `size` module we introduce in Chapter 4, where we compute the memory footprint of a value by accounting for which components are boxed and which are not.

Garbage collection in the MLKit

As discussed in Section 2.1, the MLKit enables reference-tracing garbage collection by default. The garbage collector works in combination with the region system [23]. Region deallocation handles the bulk of memory reclamation, and the garbage collector cleans up the remaining allocations, like those in long-lived or global regions, that regions inference alone cannot reclaim. The MLKit also supports generational garbage collection as a further refinement, which is enabled with `-gengc`[23].

When garbage collection is enabled, all values are tagged so that the collector can trace pointers correctly [23]. This has a visible effect on the language. The default integer type becomes `Int63.int` rather than `Int64.int`, because one bit is reserved for the tag [23]. Additionally, the type system is strengthened to prevent dangling pointers. Values captured in a closure are forced to reside in a region that lives at least as long as the closure itself, which may cause some region bindings to be widened compared to the GC-disabled case [23].

Disabling garbage collection removes tagging overhead and allows the full 64-bit integer range, but places the entire responsibility for memory reclamation on the region system. As we noted in Section 2.1, some parts of the Basis Library accumulate data in global regions that grow without bound unless a garbage collector is present. More broadly, any allocation pattern where a value outlives the `letregion` scope that the compiler would naturally assign to it can cause a region to be *lifted* to a wider scope, potentially all the way to a global region, where it will stay longer than needed, or potentially never be deallocated.

In practice this means that programmers who disable GC must actively cooperate with the region system. They must understand which regions the compiler has inferred for their data, use the region profiler to identify regions that grow unexpectedly, and restructure code or insert explicit resets to keep those regions bounded. This is a significantly higher burden than writing ordinary Standard ML, where a garbage collector silently compensates for the cases that region inference cannot handle.

Calling C functions

The MLKit allows Standard ML programs to call C functions through a foreign-function interface [23]. The compiler allocates regions for the result of a C call before the call is made and deallocates them at the appropriate point afterwards. C functions can construct ML data structures using macros provided by the runtime system. This mechanism is the foundation for the `Region` module introduced in Chapter 4. Each operation in the module is implemented as a C function that the MLKit calls through `prim`, with region parameters threaded through the compiler pipeline in the same way as for any other region-polymorphic call.

Separate compilation and ML Basis Files

The MLKit supports file-based separate compilation through ML Basis Files (`.mlb` files) [23]. An MLB file lists the source files that make up a project and can reference other MLB files, forming a directed acyclic graph of dependencies. The compiler serialises symbol-table information, including region-annotated type schemes, to disk for each compilation unit, and recompiles a unit only when its source or its assumption have changed. This infrastructure allows the MLKit to scale to large programs [23].

Inspecting intermediate representations

The MLKit provides a family of command line flags for printing the program at various stages of the compilation pipeline, which is important for understanding how region inference, storage mode analysis, and other passes have transformed the source program [23]. The most commonly used flags in our work have been `-Pcee`, which prints the call-explicit MulExp expression produced after the final compilation phase, and `-Ptypes`, which includes region-annotated types in the printed output, making it possible to see exactly which regions appear in the type of each binding and how region variables are instantiated at call sites.

Reading the intermediate code has been our primary tool for understanding how the compiler assigns values to regions, where region lifetimes begin and end, and which allocation sites are candidates for explicit resetting. We rely on this technique extensively in the case study of Chapter 6 and in validating the compiler changes described in Chapter 3.

Region profiler

The MLKit includes a region profiler that produces graphical representations of region sizes as a function of time [23]. The profiler can generate three kinds of graphs. Region profiles, which is total memory in each region over time, stack profiles, which is stack usage over time, and object profiles, which is allocation within a single region, broken down by program point.

We compile programs for profiling with the `-prof` flag, run the resulting executable to produce profiling data with `./run -microsec x`, where x is the how often a snapshot should be taken. Then we use the `rp2ps` tool to generate PostScript graphs that can be converted to PDF (e.g., with `ps2pdf` [23]). Throughout this thesis the region profiler serves as our primary validation tool. Once we have identified a memory-usage pattern by inspecting the intermediate code, we use region profiles to confirm that our changes produce the expected reduction in memory consumption. The memory-usage graphs presented in Chapters 6 and 7 are all produced using this workflow.

2.3 Region-Based Memory Management

In region-based memory management, every heap-allocated value is placed into a region, which is a container that can hold an arbitrary number of values. Regions, rather than individual objects, are the unit of deallocation. The basic concept dates back at least to the late 1960s, where Ross's AED Storage Package organised memory into a hierarchy of zones the could be freed all at once [1], and similar ideas later appeared in Hanson's arena allocators [2]. These early incarnations were manual.

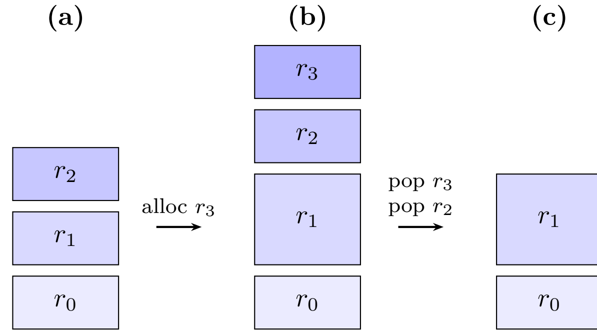

Figure 2: Three snapshots of a region stack. (a) Three regions are live; values (shown as circles) reside inside their respective regions. (b) A new region r_3 has been allocated on top, and region r_1 has grown as more values are stored into it. (c) Regions r_3 and r_2 have been deallocated; all values they contained are reclaimed at once. Adapted from [23].

The programmer decided which objects belonged to which region and when each region could be freed.

Tofte and Talpin generalised the idea to higher-order, polymorphically typed languages by introducing *region inference*, a static analysis that automatically determines region placement and lifetimes [4, 6]. Their framework made it possible to obtain the efficiency benefits of arena-style allocation, while having the compiler verify that no region is deallocated while live pointers into it still exist. This section introduces the general concept before showing how it is realised concretely in the MLKit.

The idea

The fundamental observation behind region-based memory management is that many values in a program share a common lifetime. Consider a function that builds a temporary list, transforms it, and returns a single scalar result. The intermediate list is dead after the function returns. If all of its cons cells reside in the same region, the entire list can be reclaimed by deallocating that one region, which is a constant time operation, rather than by tracing each cell individually.

More precisely, a region is a block of memory into which the program can allocate values. Regions have three operations:

1. **Allocate** a fresh, empty region
2. **Store** a value into an existing region (the region can grow as needed)
3. **Deallocate** a region, reclaiming all of its contents at once

In the Tofte-Talpin framework, region lifetimes follow the lexical structure of the program. A region is allocated at the entry of a `letregion` expression and deallocated at its exit. This means that regions are organised as a stack. If region ρ_1 is allocated before region ρ_2, then ρ_2 is deallocated before ρ_1. One can therefore think of the region stack as a generalisation of the call stack, where each "frame" can hold an arbitrary collection of heap-allocated values rather than just local variables[6].

Figure 2 illustrates the region stack at three successive points in a computation. Unlike a traditional heap with garbage collection, no runtime traversal of the live object graph is required to reclaim memory. Unlike manual `malloc` / `free`, the decision of which region a value is placed into and when each region is deallocated can be made, and checked for safety, by the compiler. Region-based memory management thus sits between fully manual and fully automatic memory management. The compiler determines the allocation and deallocation points, but the cost model is transparent and every operation runs in constant time.

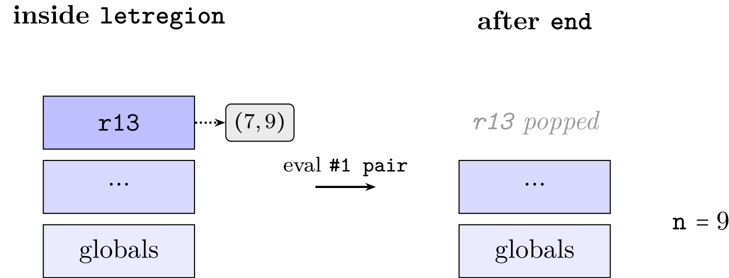

Figure 3: Region stack for the pair-projection example. Inside the `letregion`, region `r13` holds the pair $(7, 9)$. After the body is evaluated and the projection extracted, the `letregion` ends and `r13` is popped, reclaiming the pair in constant time. The unboxed result $n = 9$ does not reside in any region.

The `letregion` construct

The Tofte-Talpin framework introduces a single syntactic form for delimiting region lifetimes:

$$\texttt{letregion}\ \rho\ \texttt{in}\ e\ \texttt{end}$$

At runtime, a fresh region is allocated and bound to the region variable ρ. The body e is evaluated, and any value-producing subexpression of e that has been assigned to ρ stores its result there. When e finishes, the region is deallocated and all of its contents are reclaimed.

A key property is that `letregion` bindings are not written by the programmer. They are inserted automatically by a program analysis called region inference, which determines both the placement of values into regions and the lifetimes of those regions [7]. The source program remains standard, unmodified code.

A small example

To make the idea concrete, consider the following Standard ML function, which creates a pair and immediately projects out one component (this example is adapted from [23]):

```
val n = let val pair = (3 + 4, 4 + 5)
        in #2 pair end
```

Region inference determines that the pair is temporary. It is allocated only to be decomposed, and neither component retains a pointer to it. The compiler therefore introduces a local region for the pair:

```
val n = let region r13:1
            val pair = (3 + 4, 4 + 5) at r13
        in #1 pair end
```

The annotation `at r13` indicates that the pair is stored in region `r13`. The `:1` is the multiplicity. Because the multiplicity is 1, the region holds exactly one value and can be allocated as a small, fixed-size block directly on the runtime stack. When evaluation reaches the `end` of the `letregion`, the region is popped and its memory is reclaimed. Note that Standard ML uses 1-indexed projection for pairs, while the intermediate languages LambdaExp and MulExp use 0-indexed projections.

Figure 3 shows the region stack before and after evaluation of this fragment. This example is intentionally simple, but the same principle scales to arbitrarily complex data. A function that builds a temporary tree in a local region, extracts a result, and returns it will have the entire tree reclaimed when the `letregion` goes out of scope, regardless of how large the tree grew.

Cross-region pointers and non-nested lifetimes

Values in one region can contain pointers to values in other regions. Both forward and backward pointers occur naturally [23]. The type system guarantees that a region is never deallocated while

references to values stored in it are still in scope. The `letregion` for a region ρ can only be placed around an expression e if ρ does not appear free in the type of e or in the types of any variables that are live after e.

Not all value lifetimes are neatly nested, however. When region inference determines that a value may outlive the function that created it, for instance it escapes through the return type, the value must be placed in a region with a longer lifetime. In the limit, this means placing it in a global region that exists for the entire program. Values in global regions are never reclaimed by region deallocation alone. This is the primary reason the MLKit combines region inference with garbage collection by default, as discussed in Section 2.2.

To handle situations where values have genuinely non-nested lifetimes within a bounded scope, the MLKit supports region resetting. This is a mechanism for reclaiming all the contents of a region without removing the region from the stack. We describe the existing reset primitives, `resetRegions` and `forceResetting`, and their limitations in Section 2.7, and present our extensions to these primitives in Chapter 3.

Properties and trade-offs

The key advantages of region-based memory management, are that the safety of deallocation is checked by the compiler, that the compiler in many cases can detect potential space leaks, that each region operation executes in constant time and constant space, and that the region profiler can relate runtime memory consumption back to specific allocation points in the source program[23].

The principal trade-off is that the programmer must be aware of how values are assigned to regions. Region inference is automatic, but it is not always intuitive. A value that the programmer expects to be short-lived may end up in a long-lived or global region due to escaping references or higher-order function application [23]. Understanding these assignments requires reading the compiler's intermediate output (Section 2.2) and reasoning about region polymorphism and effects, which we introduce next.

2.4 Region Polymorphism and Annotated Types

Standard ML's type system is parametrically polymorphic. A function such as `length : 'a list -> int` works uniformly over all element types. Region inference extends this polymorphism to regions. A function that is polymorphic in the regions of its argument and result can place its output into different regions at different call sites, which is essential for keeping region lifetimes short and separated [23] [6]. This section introduces the type-level machinery that makes this possible. Region-annotated types, region-annotated type schemes, and the instantiation mechanism that connects them to the `letregion` bindings are discussed in Section 2.3.

Region-annotated types with places

ML type inference assigns a type to every expression. Region inference refines this by assigning a *region-annotated type with place*, conventionally written μ. The grammar is

$$\mu ::= (\tau,\ \rho)\ \mid \tau$$

where τ is a region-annotated type, which may itself contain nested types with places, and ρ is a region variable indicating the region in which the value resides [23]. Every type constructor that represents a boxed value is paired with a region variable. Type constructors that represent unboxed values carry no region annotation, because these values are stored directly in registers or machine words and do not reside in regions (Section 2.2).

Below are some examples:

`unit`	The type of 0-tuples. Unboxed
$(\texttt{string},\ \rho)$	A string residing in region ρ.
$(\texttt{int} \times (\texttt{string},\ \rho_1),\ \rho_2)$	A pair in ρ_2 whose second component is a string in ρ_1.

Notice that a single compound value can span multiple regions. A pair of strings, for instance, has the type $(((\texttt{string}, \rho_1) \times (\texttt{string}, \rho_2)), \rho_3)$, involving three region variables, one for each string and one for the pair cell itself. In practice, region inference often unifies ρ_1 and ρ_2 when both strings are allocated in the same context, but the type system permits them to be distinct, for example, when one string escapes to a wider scope than the other.

Lists illustrate the pattern well. The region-annotated type of a list is $(\mu, [\rho])$ `list`, where μ is the type with place of the elements and ρ is the region holding the auxiliary cons-cell pairs [23]. The list constructor have the following region-annotated type schemes:

$$\texttt{nil} : \forall \alpha \, \rho. \, (\alpha, [\rho]) \, \texttt{list}$$
$$\texttt{::} : \forall \alpha \, \rho \, \varepsilon. \, (\alpha \times (\alpha, [\rho]) \, \texttt{list}, \rho) \xrightarrow{\varepsilon.\varnothing} (\alpha, [\rho]) \, \texttt{list}$$

Both are polymorphic in the region variable ρ, which means that different lists can reside in different regions. The type of `::` forces the new cons cell to be placed in the same region as the existing cons cells, and it forces the new element to have the same type with place as the existing elements. The annotation $\varepsilon.\varnothing$ on the function arrow is called an arrow effect. Its first component ε is an effect variable that tracks dependencies between effects at call sites, and its second component $\varnothing$ is the latent effect, the set of region operations that evaluating the function body may perform. Here the latent effect is empty, meaning the body performs no region operations that are visible to the caller [23]. Arrow effects and effect variables are explained in Section 2.5.

Region-annotated type schemes

Just as Standard ML generalises types to type schemes by universally quantifying over type variables, the MLKit generalises region-annotated types to *region-annotated type schemes* by additionally quantifying over region variables and effect variables:

$$\sigma \; ::= \; \forall \alpha_1 \cdots \alpha_n \; \rho_1 \cdots \rho_k \; \varepsilon_1 \cdots \varepsilon_m . \; \tau$$

where $\alpha_1, \ldots, \alpha_n$ are the type variables, $\rho_1, \ldots, \rho_k$ are region variables, $\varepsilon_1, \ldots, \varepsilon_m$ are effect variables, and τ is a region-annotated type. [23].

Every binding of a variable to a boxed value, whether by **fun**, **let**, or **fn**, is associated with a region-annotated type scheme with place, that is, a pair (σ, ρ) where ρ identifies the region that holds the value. For unboxed bindings the place is omitted.

Region polymorphism

A function whose region-annotated type scheme contains bound region variables is called region-polymorphic. Different call sites can instantiate the bound region variables to different actual regions, so the function can place its result in a different region each time it is called [23].

As an example, consider a function that produces a list containing x repeated n times:

```
1 fun repeat(x, 0) = []
2   | repeat(x, n) = x :: repeat(x, n - 1)
3
4 val r = let val a = repeat(42, 3)
5         in  length(repeat(7, 5)) + length a
6         end
```

Because `repeat` produces a list, the compiler infers a region-annotated type scheme that is polymorphic in the cons-cell region. Compiling with `-Pdre` (and `-maximum_inline_size 0` to avoid inlining) reveals the intermediate code after dropping of unused regions parameters:

```
1 fun repeat [r13:inf] (v126, v127) =
2     case v126 of
3       0 => nil
```

```
    | _ => :: (v127,
              let val v138 = v126 - 1
              in repeat[r13] (v138, v127)
              end) attop r13

val r =
  let region r23:inf;
      val a = repeat[r23] (3, 42)
  in  let region r27:inf
      in  length (repeat[r27] (5, 7))
      end + length a
  end
```

The formal region parameter `[r13:INF]` in the declaration of `repeat` indicates that the function is region-polymorphic. It takes a region as an implicit argument and places the cons cells of its result list into that region. At the first call site, the bound variable is instantiated to `r23`, at the second, to `r27`. Each of these regions is introduced by its own `letregion` binding, so the two lists reside in separate regions with separate lifetimes.

Region `r27` is scoped inside the inner `letregion`, so the second list, produced by `repeat(7, 5)` is deallocated as soon as its length has been computed. Region `r23` has a wider scope, keeping the first list alive until both `length` calls have completed. Without region polymorphism, both lists would share a single region, and that region could not be deallocated until the last list was dead, defeating much of the purpose of region-based memory management.

Endomorphisms and exomorphisms

A useful distinction arises from the relationship between the regions in a function's argument type and the regions in its result type[23].

A function is a *region exomorphism* if the region variables in its result type are disjoint from those in its argument type. Such a function may place its output in fresh regions that are independent of its input. A function is a *region endomorphism* if its result must reside in the same regions as its argument, because the function's type forces input and output to share regions.

The distinction is not always sharp. A function may have both an endomorphic and an exomorphic side, and even a region exomorphic function can be forced to act as an endomorphism by its calling context [23]. Endomorphisms are particularly important for bounded-memory programming, though not because they prevent accumulation of their own. An endomorphic loop will grow its region without bound, since the region never goes out of scope. Because the result of each iteration resides in the same, statically known region as the input, the programmer knows where the data lives, and can issue a reset at any point deemed appropriate. An exomorphic function used as a tail call in a state loop may have its old region reclaimed automatically when it goes out of scope, but this requires both that the compiler can determine that the old state is unreachable, and that the programmer copies the all input values into new regions at each step, which is both costly and non-trivial.

The compiler infers the endomorphic or exomorphic character of a function from its structure, without requiring any annotation from the programmer [23]. Consider the following function, which prepends a fixed sentinel string to a list:

```
fun prependStart l = "start" :: l
```

The expression `"start" :: l` conses a new cell onto the input list `l`. The tail of the new cell is `l` itself, so the new cons cell must reside in the same region as the existing cons cells of `l`. The region-annotated type scheme assigned to `prependStart` is:

$$\forall \rho_a\, \rho_b.\ ((\mathtt{string}, \rho_b), [\rho_a])\ \mathtt{list} \longrightarrow ((\mathtt{string}, \rho_b), [\rho_a])\ \mathtt{list}$$

Here we have omitted the effect of the function. We see that the argument type and the result type are identical. The cons cells of both the input and the output list reside in ρ_a, and the string values

both reside in ρ_b. Neither region variable is fresh in the result. The same is visible in the compiled intermediate code:

```
let fun prependStart
        :all
            r21,r19.
            ((string,r19), [r21]) list->((string,r19), [r21]) list
        at r1
        [r21:1, r19:1]
        l =
        :: ("start", l)at r21
in  {|
    prependStart: (
          all
            r21,r19.
            ((string,r19), [r21]) list->((string,r19), [r21]) list,
          r1
    )
    |}
end
```

The new cons cell is allocated `at r21`, the same region that holds the input list's cons cells.

Limitations of region polymorphism

Region polymorphism applies only to identifiers bound by `fun`. Lambda-bound identifiers `fn` are never region-polymorphic [23]. This limitation has a practical consequence for higher-order functions. A function that takes another function as an argument cannot instantiate the region parameters of that argument differently at different call sites within its body. The regions appearing in the type of the function argument must be allocated before the higher-order function is called and remain live for the duration of the call. The compiler inserts a `letregion` around the call site rather than inside the body of the higher-order function [23].

This means that the region lifetime of a value produced by a function argument is determined at the call site, not inside the higher-order function itself. In practice, this limits the region management strategies available when building abstraction over functions, and it is a main reason for the structural design of the case study in Chapter 6.

2.5 Region Inference and Effect System

The previous section introduced region-annotated types and region polymorphism, which describe where values reside. In this section we turn to the question of how the compiler determines region lifetimes, that is, where to place `letregion` bindings so that regions are deallocated as early as possible without freeing memory that is still needed. The answer lies in an effect system that tracks which regions are read from and written to by each expression [6] [23].

Atomic effects

Region inference assigns to every expression not just a region-annotated type with place, but also an effect, which is a finite set of atomic effects. The two most important atomic effects are:

- `put`(ρ): a value is stored into region ρ
- `get`(ρ): a value is read from region ρ

A third form of atomic effect is an effect variable ε, which is used to abstract over the effects of higher-order functions. We will return to effect variables below.

As a simple example, recall the pair-projection fragment from Section 2.3:

```
1 val n = let region r13:1
2             val pair = (3 + 4, 4 + 5) at r13
3         in #1 pair end
```

Region inference determines that the body of the `letregion` has effect $\{\texttt{put}(r_{13}), \texttt{get}(r_{13})\}$, meaning the pair is stored into r_{13} and then read from r_{13} by the projection[23].

The `letregion` safety criterion

Effects are the mechanism by which the compiler decides where it is safe to insert a `letregion` binding. The rule, informally stated, is that a region variable ρ may be bound by a `letregion` around an expression e if ρ occurs free in the effect of e but occurs free neither in the region-annotated type with place of e nor in the region-annotated type scheme with place of any program variable in the domain of the type environment [23]. The intuition is that if ρ appears only in the effect but not in any type that is visible after e finishes, then the region is used only locally within e and no live references to it escapes. It is therefore safe to deallocate it when e completes.

Returning to the pair example. The body expression has type `int`, which is an unboxed type with no region annotations, and no free variable in scope after the `let` has r_{13} in its type. The region variable r_{13} appears only in the effect, so the compiler wraps a `letregion` around the body, deallocating r_{13} after the projection.

This criterion has been proved correct for a skeletal version of the region-annotated language [6]. It is conservative, so there may be cases where a region could safely be deallocated earlier than the criterion allows, but the criterion never deallocates a region too early.

Arrow Effects and Latent Effects

In a first-order language, `put` and `get` would suffice. In Standard ML, however, functions are first-class values, so they can be stored in data structures, passed as arguments, and returned as results. A function's body may read from and write to regions, but these effects do not occur until the function is applied, which may happen much later than the point where the function is created.

To handle this, region-annotated function types carry an *arrow effect*, written $\varepsilon.\varphi$, where ε is an effect variable and φ is the latent effect. A *latent effect* is the set of atomic effects that will occur when the function is applied[23]:

$$(\mu \xrightarrow{\varepsilon.\varphi} \mu', \rho)$$

Here μ and μ' are the argument and result types with places, ρ is the region holding the closure, ε is the effect variable that serves as a handle for the arrow effect, and φ is the latent effect.

The effect variable ε plays an important role in higher-order functions. When a function is passed as an argument to another function, the callee does not know the concrete latent effect of its argument, it only knows the effect variable. At each concrete call site, the effect variable is instantiated to the actual latent effect of the function being passed. This polymorphism in effects makes it possible to distinguish between the effects of different actual arguments, which in turn allows region inference to assign different region lifetimes to the different uses [23].

How effects determine region lifetimes

The effect system and the `letregion` safety criterion work together to determine region lifetimes. We can summarise the process as follows. First, region inference assigns a region-annotated type with place and an effect to every subexpression. Then, for each region variable ρ that appears in the effect of some expression e but does not appear free in the type of e or in the types of variables that are live after e, the compiler introduces a `letregion` ρ `in` e `end`. The resulting program has the property that every region is allocated at the latest possible point, which is at the innermost enclosing expression that uses it, and deallocated at the earliest safe point.

In practice, region inference operates in two phases in the MLKit pipeline. First, the spreading pass annotates types with fresh region variables at every type constructor. Then, the region inference pass unifies region variables, computes effects, and inserts `letregion` bindings based on the safety criterion [7].

Example: `repeat`

To make the interaction between effects and region lifetimes concrete, consider the `repeat` function from Section 2.4. Its region-annotated type scheme is:

$$\forall \rho\,\varepsilon.\ [\texttt{int},\ \texttt{int}] \xrightarrow{\varepsilon.\{\texttt{put}(\rho)\}} (\texttt{int},\ [\rho])\ \texttt{list}$$

The latent effect $\{\texttt{put}(\rho)\}$ records that applying `repeat` stores cons cells into ρ. At the call site `repeat[r39] (5, 7)`, the bound variable ρ is instantiated to r_{39}, so the effect at the call site includes $\texttt{put}(r_{39})$. Because r_{39} does not appear in the type of the enclosing expression, the compiler introduces $\texttt{letregion}\ r_{39}\ \texttt{in}\ \ldots\texttt{end}$ around the call, and the list is deallocated as soon as its length has been computed.

2.6 Storage Modes and the Runtime Memory Model

When a value is stored into an existing region, the allocator must decide whether to preserve the region's current contents or discard them. This choice is determined by the storage mode analysis, an optimisation pass that can reduce memory consumption by reusing region space without allocating new regions [23].

The three storage modes

Every allocation point, that is, every $\texttt{at}\ \rho$ annotation in the intermediate code, is refined by the storage mode analysis into one of three forms[23].

$\texttt{attop}\ \rho$ Store the value at the top of region ρ, increasing the region's size. The existing contents of the region are preserved.

$\texttt{atbot}\ \rho$ First reset region ρ, reclaiming all its contents, then store the new value. This mode is safe only if the region contains no live data at the allocation point. When it is applicable, `atbot` effectively reuses the region's memory without the programmer writing any explicit reset instructions.

$\texttt{sat}\ \rho$ (somewhere at) Defer the choice between `attop` and `atbot` to runtime. This mode is used when the region variable ρ is a formal region parameter of a region-polymorphic function. Different call sites may instantiate ρ to regions with different liveness properties, so the decision cannot be made statically.

Storage modes are only significant for infinite regions, since finite regions reside on the stack and are always fully consumed by a single value [23].

The storage mode analysis

The storage mode analysis is a backwards-flow analysis that runs after region inference and multiplicity inference in the MLKit pipeline [5]. For each allocation point $\texttt{at}\ \rho$, it computes a set of locally live variables, which are the variables whose values are still needed by the remainder of the computation up to the end of the enclosing function. It assigns a storage mode based on the following rules, informally stated[23].

Case A: global region If ρ is a global region, the mode is `attop`. Global regions may be referenced by other compilation units, so it is never safe to reset them automatically [23].

Case B: inside a function If the allocation point occurs inside a function body, three sub-cases arise depending on where ρ is bound:

B1. ρ is bound outside the smallest enclosing lambda abstraction, and is not a formal region parameter of that function, use `attop`. The analysis does not attempt to determines liveness for non-locally-bound regions.

B2. ρ is bound by a `letregion` inside the same function, use `atbot` if no locally live variable at the allocation point has ρ free in its region-annotated type scheme with place. Otherwise use `attop`. The intuition is the same as the `letregion` safety criterion from Section 2.5. If no live variable points into ρ, the region contains no live data and can be reset.

B3. ρ is a formal region parameter of the enclosing region-polymorphic function, use `sat` if it is safe, and `attop` otherwise. Determining safety requires checking region aliasing. Because ρ is only a formal parameter, different call sites may instantiate it to different actual regions, and the actual region my coincide with a region that holds live data.

Case C: outside any function. If the allocation point does not occur inside a function and ρ is bound by a `letregion`, use `atbot` if no locally live variable has ρ free in its type. Otherwise use `attop` [23].

Region aliasing and the region flow graph

Case B3 deserves special attention because it introduces the problem of region aliasing, which means that two different region variables in the source program may denote the same region at runtime[23]. If a formal region parameter ρ is instantiated at a call site to an actual region variable ρ', and ρ' also appears in the type of a live variable, then resetting ρ inside the function would destroy live data, even though ρ itself does not appear in any live variable's type.

To handle this, the MLKit builds a *region flow graph* for each compilation unit [23]. The nodes of the graph are region variables and effect variables. A directed edge from ρ_1 to ρ_2 is created whenever ρ_1 is a formal region parameter of a function and ρ_2 is the corresponding actual parameter at some call site. For exported functions, edges are also added from each formal region parameter to the global region of the same runtime type, since the function may be called from other compilation units with unknown actual regions.

Let $\langle\rho\rangle$ denote the set of region variables reachable from ρ in the region flow graph. The precise version of rule B3 states: use `sat` if, for every locally live variable l and every region variable ρ' free in the type of l, we have $\langle\rho\rangle \cap \langle\rho'\rangle = \varnothing$. Use `attop` otherwise [23]. In other words, `sat` is safe only when the formal parameter can never be aliased with a region that holds live data at the allocation point.

Runtime representation of storage modes

At runtime, a region name is a 64-bit pointer, to a stack-allocated finite region or to a stack-allocated region descriptor for an infinite region. Because pointers to word-aligned data always have two zero bits in their least significant position, the MLKit uses these bits to encode status information [23]

The infinity bit indicates whether the region is finite or infinite. This allows the runtime to skip unnecessary operations on finite region (e.g. page-list manipulation).

The atbot bit holds the current storage mode of the region. When a region-polymorphic function receives a region parameter with mode `sat`, the runtime tests this bit to decide whether to reset the region or append to it.

When a region is passed as an actual parameter to a function call with mode `atbot`, the atbot bit is set on the region name that the callee receives. With mode `attop`, the bit is cleared. This encoding means that the `sat` check at an allocation point inside the function body reduces to a single bit test, with no additional data structures or lookups [23].

2.7 The Region Reset Primitives

The storage mode analysis described in the previous section automatically inserts `atbot` annotations where it can prove that a region contains no live data. The problem is that the analysis is conservative, meaning it reasons about all possible executions of the program and cannot exploit knowledge that is only available at a specific program point. In these situations the programmer has information that the static analysis cannot see, and the MLKit provides two built-in primitives that let the programmer request resetting explicitly [23].

Syntax and Semantics

Both primitives take a single value identifier as argument:

```
1 resetRegions vid
2 forceResetting vid
```

When the compiler encounters one of these primitives, it collects the set of region variables that appear free in the region-annotated type scheme with place of `vid`. These are the regions that the primitive targets for resetting. The argument must be a value identifier, not an arbitrary expression, so that the compiler can look up its type scheme directly[23].

The two primitives differ in how they interact with the storage mode analysis. With `resetRegions`, the compiler respects the storage mode analysis. If the analysis determined `attop` the reset is suppressed and the compiler prints a diagnostic warning explaining the conflict. If the mode is `sat`, the reset is deferred to runtime, where it takes effect only if the atbot bit on the region is set. Only when the mode is `atbot` does the reset proceed unconditionally.

With `forceResetting`, the reset is performed regardless of the storage mode. The compiler still prints a warning when the mode is `sat` or `attop`, but it generates the reset instruction anyway. This makes `forceResetting` the only mechanism through which a programmer can introduce a potential crash via the region system, because if the region does in fact contain live data, the program will access freed memory[23].

Both primitives return unit. To port programs that use these primitives to other Standard ML implementations, one can simply declare them as identity functions on unit:

```
1 fun resetRegions _ = ()
2 fun forceResetting _ = ()
```

Compiler Diagnostics

When `resetRegions` encounters a region that cannot be safely reset, the compiler produces a detailed diagnostic. To illustrate, consider a function that repeatedly halves a list by dropping every other element:

```
1  local
2    fun shrink [] = []
3      | shrink [x] = []
4      | shrink (x::_::xs) = x :: shrink xs
5
6    fun loop [] = 0
7      | loop xs =
8          let val xs' = shrink xs
9          in resetRegions xs; 1 + loop xs'
10         end
```

```
in
  val res = loop [1, 2, 3, 4, 5, 6, 7, 8, 9]
end
```

The function `shrink` is an exomorphism. It produces a shorter list in a fresh region. After `shrink` returns, `xs` is no longer needed, so the intent is to reclaim its region before recursing on the smaller list. Compiling with `-Pdre` reveals the relevant parts of the intermediate code:

```
fun shrink at r9 [r17:INF] var1 =
    case var1 of
      nil => nil
    | :: v97 =>
        ...
        :: (v100, shrink[r17] v102) at r17

fun loop at r35 [r39:1] var2 =
    case var2 of
      nil => 0
    | _ =>
        let val xs = var2
            region r43:INF
            val xs' = shrink[r43] var2
            val _ = resetRegions [r39] xs
        in  1 + loop[r43] xs'
        end
```

We can see that `loop` has formal region parameter `r39`, which holds the cons cells of the input list. The call `shrink[r43] var2` produces the shortened list `xs'` in a fresh region `r43`. At this point `xs` is dead and its region `r39` could be reclaimed. However, the recursive call `loop[r43] xs'` instantiates `loop`'s formal parameter to `r43`, creating an edge from `r39` to `r43` in the region flow graph

The compiler therefore rejects the reset:

```
resetRegions(xs):
   You have suggested resetting the regions that appear free
   in the type scheme with place of 'xs', i.e., in
   ('a2, [r39]) list
   (1)
        'r39': there is a conflict with the locally
        live variable
        xs' :('a2, [r43]) list
        from which the following region variables can be reached
        in the region flow graph:
             {r43}
        Amongst these, 'r43' can also be reached from 'r39'.
        Thus I have given 'r39' storage mode "attop".
```

Because `r43` is reachable from `r39` in the region flow graph, and `xs'`, which lives in `r43` is live at the point where `resetRegions` appears and thus invokes rule B3 of the storage mode analysis, forces storage mode `attop`, and the reset is suppressed.

This is a case where the analysis is too conservative. The value bound to `xs` is genuinely dead after `shrink` has consumed it, but the region flow graph cannot distinguish the current call's region from the recursive call's region because it does not track calling context. The programmer's only option with the existing primitives is to switch to `forceResetting`, bypassing all safety checking.

2.8 ReML: Programming with Explicit Regions

In standard MLKit usage, region variables exist only in the compiler's intermediate representation. The programmer writes ordinary Standard ML and never refers to a region by name. *ReML* is a higher-order, statically typed functional language that extends Standard ML with syntax for being

explicit about the regions in which values reside and the effects that functions may have [21]. ReML is integrated with the MLKit's region inference. A non-annotated Standard ML program is a valid ReML program, and the programmer may choose to annotate only certain parts of a program while leaving the rest to the inference [21]. When the compiler is invoked with the `-reml` flag, the parser accepts additional syntax for naming regions, binding them, placing values into them, and passing them as explicit arguments to functions and primitives.

This section introduces the ReML syntax. Our compiler extensions and the `Region` module both require `-reml` to be enabled.

Naming and binding regions

An explicit region variable is written with a backtick prefix, for example `` `r `` or `` `my_region ``. A new explicit region is introduced with the `let with` binding form:

```
let with r
    val s : string`r = "hello"
in String.size s
end
```

The `let with r` construct allocates a fresh region and binds it to the explicit region variable `` `r ``. The region is deallocated when evaluation reaches `end`, just like a compiler-inserted `letregion`. The difference is that the programmer has chosen the name and can refer to it elsewhere in the body.

Multiple regions can be bound in a single `let with`:

```
let with r1 r2
    val s : string`r1 = "hello"
    val t : string`r2 = "world"
in s ^ " " ^ t
end
```

Placing values into explicit regions

A type annotation with a backtick suffix places a value into a named region. In the example above, `` string`r `` means "a string residing in region `` `r ``". This corresponds to the region-annotated type (`string`, ρ) from Section 2.4, but written in source syntax rather than in the compiler's intermediate notation.

For compound types, multiple region annotations can appear:

```
fun timeToGC `[r1 r2 r3]
      (ss : (conn * string`r1)`r3 list`r2) : bool =
    let val total = Region.memoryUsageOfRegion `[r1] ()
                  + Region.memoryUsageOfRegion `[r2] ()
        val live  = Size.size (...) ss
    in  live < total div 2
    end
```

Here the type annotation `` (conn * string`r1)`r3 list`r2) `` places the string in `` `r1 ``, the cons-cell pairs in `` `r2 ``, and the list element pair cells in `` `r3 ``, giving the programmer visibility into exactly which regions hold which parts of the data structure. This example is taken from our case study (Chapter 6, where we use explicit region annotations to dictate when garbage collection of specific regions is warranted.

Explicit region parameters on functions

A function can declare explicit region parameters using the `` `[...] `` syntax after the function name:

```
fun myFun `[r] (x : string`r) = String.size x
```

The region variable `` `r `` becomes a formal region parameter of `myFun`, analogous to the compiler-generated formal region parameters described in Section 2.4. At call sites, the region is passed explicitly using the same bracket notation:

```
let with my_region
    val s : string`my_region = "test"
in myFun `[my_region] s
end
```

It is also possible to pass multiple regions explicitly to a function, as seen in the `TimeToGC` example.

Interaction with region inference

Explicit region variables coexist with the compiler's automatically inferred regions. When a value is annotated with an explicit region, region inference respects that annotation, so when a value has no annotations, the compiler infers its region as usual [21].

Explicit region variables interact with the effect system in the expected way. A `put` effect on the explicit region is included in the effect on any expression that stores into it, and the `letregion` safety criterion applies to explicit regions just as it does to inferred ones. In particular, the compiler prevents an explicit region from escaping its binding scope. If the region variable appears free in the type of the result expression, the compiler reports an error rather than silently deallocating the region and creating a dangling pointer. If a function needs to return a value in an explicit region, the region must be passed in as a parameter from a wider scope rather than bound locally with `let with`.

An important design choice is that explicit region variables are never unified with each other by the compiler [21]. If the programmer annotates two values with distinct explicit regions `` `r1 `` and `` `r2 ``, and the program's typing requires these regions to be the same, the compiler reports an error rather than silently merging them. This property gives the programmer a modular guarantee, namely that an explicitly annotated function will place its values in the specified regions regardless of how the surrounding code changes or how the compiler's optimisations evolve.

A calling context may, however, pass the same actual region for different formal explicit region parameters. This is consistent with ordinary region polymorphism. The callee treats the parameters as distinct, but the caller is free to instantiate them to the same region if the types permit it.

Effect constraints

Beyond explicit regions, ReML also supports effect constraints, which are assertions about the relationship between effects on different subexpressions, expressed through so called `while` types [21]. For example, a constraint of the form `e1 ## e2` specifies that two effects have no overlapping `put` effects, meaning the associated function do not allocate into the same regions. This mechanism was designed primarily for ensuring the absence of allocation races in parallel programs and is used by the MLKit's `par` function to guarantee that two threads allocate into disjoint regions.

2.9 Related work

The theoretical foundation for our work is the region calculus of Tofte and Talpin [6], which formalised the type-and-effect system that enables safe region deallocation, and the region inference algorithm of Tofte and Birkedal [7], which made the approach practical in the MLKit compiler. Our extensions preserve the structure of this system but add new surface-level constructs for programmer direct region operations.

Prior work on the MLKit has explored combining region inference with garbage collection. Hallenberg, Elsman, and Tofte [12] showed that the two strategies are complementary, with regions handling predictable lifetimes and garbage collection handling the rest. Elsman and Hallenberg [9] further integrated region management with tag-free generational garbage collection. Our work takes the opposite direction. Rather than supplementing regions with a garbage collector, we provide tools

that let the programmer manage memory entirely within the region system, targeting environments where garbage collection is undesirable.

ReML [21] is the direct predecessor of our work. It extends Standard ML with explicit region annotations and effect constraints, allowing programmers to control which values go in which regions. We build on ReML by extending its reset primitives with explicit region arguments and adding a module for runtime region introspection, neither of which is available in the original ReML system.

Several other languages have explored explicit programmer control over memory regions. Cyclone [13] is a safe dialect of C that extended the language with programmer-managed regions, using a type system to prevent dangling pointers [11]. Cyclone's regions are lexically scope and explicitly named, similar to ReML's `with` construct. Cyclone uses default annotations and local type inference to reduce the annotation burden, but does not provide a whole program region inference algorithm that can annotate an unannotated program as the MLKit does. Cyclone also does not support resetting a region's memory for reuse. Rust [19] achieves memory safety through an ownership and borrowing system whose lifetime annotations play a role analogous to region variables. Both Rust and ReML aim to ensure safety while enabling early deallocation, but the mechanisms differ. Rust deallocates individual values when they go out of scope, whereas ReML groups values into named regions that can be deallocated as a unit. Flix [20] uses regions to encapsulate scoped local mutable memory. All mutable data belongs to a region, and the effects associated with a region vanish when the region goes out of scope, allowing internally imperative functions to present a pure interface. Flix's regions serve a different purpose than ReML's. They scope mutability rather than controlling allocation and deallocation. OxCaml [22] extends OCaml with a `local` mode that prevents values from escaping their enclosing scope, which allows the compiler to allocate them on the stack and reclaim them without the garbage collector. ReML provides a greater degree of control. The programmer can name regions, pass them as arguments to functions, place specific values into specific regions, and reset them.

Wang and Appel [10] demonstrated that a tracing garbage collector can be implemented as a well-typed function within the language it collects, by layering a stop-and-copy collector on top of a region calculus. In their system, the heap resides in a single region that serves as the from-space; the collector allocates a fresh to-space region, deep-copies all live roots into it using per-type copy functions, and then safely deallocates the from-space. Type soundness of the combined mutator and collector follows from the region type system, removing the collector from the trusted computing base. Their work shows that regions and tracing collection are compositional rather than opposed: one can serve as the foundation for the other. This observation complements the direction taken in the present thesis. Where Wang and Appel build a collector on top of regions, and Hallenberg et al. build a collector alongside regions [12], we aim to make the region system itself sufficient by giving the programmer direct control over region lifetimes. Their per-type copy functions, which deep-copy values between regions, are also conceptually related to our `Size` combinator library and to the copy combinator we sketch as future work in Chapter 8.

Kanne and Geisshirt [24] implemented a modular protocol stack in SML using region-based memory management for unikernels. They demonstrated the double-copy method for bounded memory in long running services and identified threshold-based garbage collection as future work. Our case study in Chapter 6 builds on their service architecture and implements the adaptive GC strategy they proposed, using the `Region` and `Size` modules to make data driven GC decisions at runtime.

3 Programming with Explicit Regions

This chapter describes two compiler extensions that give programmers finer-grained control over region lifecycle in ReML. The first extension adds explicit region arguments to the existing resetRegions and forceResetting primitives. The second extension allows the prim mechanism to accept explicit region parameters, enabling SML code to pass region pointers to C function, which is a prerequisite for the Region module described in 4.

Both extensions are implemented within the existing MLKit compiler pipeline. The changes touch the LambdaExp intermediate language, the region inference spread phase, the storage mode analysis, and the drop regions phase, but do not alter the fundamental region type system or its soundness properties.

3.1 Design Goals and Motivating Example

The existing reset primitives have several limitations that motivate the extensions described in this chapter.

No resetting without a value. Both primitives reset all regions that appear free in the type of their argument. The programmer cannot target a specific subset of those regions. If a value's type mentions five region variables, but only one of them is growing problematically, or worse, one of those regions contains live data that must not be reset, there is no way to reset the problematic regions.

No resetting of explicit variables. In the existing MLKit, the primitives do not accept explicit region variables. Even when compiling with -reml, the programmer cannot write resetRegions ‘[r] to target a specific named region.

Conservative analysis suppresses valid resets. As illustrated by the shrink example, the storage mode analysis can be too conservative. The region flow graph may create aliasing edges that prevent an atbot or sat mode even when resetting would be sound. The only option is forceResetting, which bypasses all safety checking and risks memory corruption if the programmer's reasoning is wrong.

These limitations become acute in long running programs that use tail-recursive loops to process an unbounded stream of inputs. In such programs, the standard region lifecycle, does not reclaim memory within a single loop iteration, because the enclosing letregion scope spans the entire loop. The programmer must instead reset regions explicitly to keep memory bounded.

With the existing primitives, this requires the programmer to construct a value whose type happens to mention exactly the right regions, and then pass that value to resetRegions. If the target regions are spread across multiple values, there is no way to reset without deallocating live data.

Our first extension addresses the two first limitations, by allowing explicit region variables as additional arguments to both reset primitives. The programmer can now write resetRegions 'r1 value to reset the region `r1 in addition to the regions inferred from value's type, or resetRegions `r1 () to reset a single named region without reference to any value. The storage mode analysis applies its safety rules to the explicit regions just as it does to inferred ones, so resetRegions remains safe while forceResetting remains the programmer's option for cases where the analysis is too conservative.

Our second extension serves a different but complementary purpose. The Region module needs to pass explicit region pointers to the C runtime function, for example, to query the memory usage of a specific region. The existing prim mechanism had no way to accept explicit region parameters independently of the function's return type. We extend it so that prim [`r] ("c_function_name", ()) resolves `r in the region static environment, adds a put effect for the region, and prepends the region pointer to the C function's argument list at code generation. Without this extension, the Region module could not exist.

The remainder of this chapter walks through each extension in detail, showing the compiler pipeline changes with code excerpts from the modified source files, illustrating the behaviour with examples and intermediate code, and arguing that both extensions preserve the soundness of the region system.

3.2 Extension 1: Explicit Region Arguments for Reset Primitives

This section describes the first compiler extension, which allows `resetRegions` and `forceResetting` to accept explicit region variables as additional arguments. We present the new syntax, walk through the changes at each stage of the compiler pipeline and show how the extended primitives behave on concrete examples.

Syntax

The extended primitives accept an optional list of explicit region variables in the `` `[...] `` bracket notation, placed between the primitive name and the argument expression:

```
resetRegions `[r1 r2] value
forceResetting `[r] value
```

The explicit regions are merged with the regions that the compiler infers from the value's type scheme. That is, writing ``resetRegions `[r1] xs`` targets both `` `r1 `` and the regions appearing free in the type of `xs`.

A second syntactic change allows `unit` as the argument expression

```
resetRegions `[r] ()
forceResetting `[r1 r2] ()
```

Since unit has no regions in its type, the reset targets only the explicitly named regions. This form is useful, for instance, when the programmer wants to reset a region that is part of some variables type.

Without explicit region arguments, the primitive behave exactly as before, so the extension is fully backwards-compatible.

Relevant data structures

Before walking through the pipeline changes, we briefly describe the compiler-internal data structures involved.

Region variables (`regvar`) A `regvar` is the compiler's representation of a source-level explicit region name, meaning the `` `r `` that the programmer writes. It is a symbolic name, that exists from parsing through to the Lambda intermediate language. At this stage, a `regvar` does not yet correspond to a runtime region, it is simply a name awaiting resolution.

Places (`place` / `effect`) A `place` is the compiler's internal representation of a region after region inference. Places are nodes in the effect graph, each representing a concrete runtime region. The type `place` is an alias for `effect` in the compiler's effect graph representation. Every region variable in the compiler-generated intermediate code corresponds to a place. Explicit region variables also correspond to places once they have been resolved.

The region static environment (`RSE`) The region static environment is a mapping from `regvar` names to `place` nodes. When the spreading phase encounters an explicit region variable, it calls `RSE.lookupRegVar rse rv` to find the place that `rv` refers to. The `RSE` is populated by `let with` declarations and by explicit region parameters on functions. If a lookup fails, the region variable is not in scope and the compiler reports and error.

Region-annotated types with places (mus) A mu is a region-annotated type, meaning a pair of an ML type and the place where values of that type reside. The function `R.ann_mus mus0 extras` extracts all place annotations from a list of mus, appending the `extras` list. The result is a list of all places, both inferred and explicit, that are associated with the type. The predicate `Eff.is_rho` filters this list to keep only region-level places, and `Eff.mkPut` constructs a `put` effect on a place.

Pipeline overview

Figure 4 shows the MLKit compilation pipeline with the source files responsible for each phase, highlighting the four files modified for this extension. The change begins at elaboration, where `COMPILE_DEC.sml` extracts explicit region variables from the parse tree and attaches them to the primitive constructors in the LambdaExp intermediate language. `LAMBDA_EXP.sml` defines the extended primitive constructors, handles pretty-printing of the new `regvars` field, including the modified diagnostic messages for reset primitives, and updates the pickling functions that serialise Lambda expressions for cross-module inlining. The region variables are then carried unchanged through the Lambda optimiser and related passes in `LAMBDA_BASICS.sml` and `OPT_LAMBDA.sml`, which propagate the new field without acting on it. The spreading phase, in `SPREAD_EXPRESSION.sml`, resolves the symbolic region names to concrete places in the effect graph and computes their effects, while the storage mode analysis, in `AT_INF.sml`, determines whether each region can safely be reset and produces diagnostic messages. We now walk through each stage in the order the compiler processes them.

Stage 1: Elaboration to Lambda (`CompileDec.sml`

The parser already recognises the `` `[...] `` syntax on function application as part of the existing ReML infrastructure [21]. When `CompileDec` encounters an application of `resetRegions` or `forceResetting`, it extracts the optional explicit region variables from the parse-tree annotation `regvars_opt` and attaches them to the primitive constructor. Below is the modified `RESET_REGIONS` case, the `FORCE_RESET_REGIONS` case is analogous:

```
(* CompileDec.sml - CE.RESET_REGIONS case *)
| CE.RESET_REGIONS =>
    let val tau' =
            case info_from_app
              of SOME(TypeInfo.VAR_INFO
                     {instances = [tau]}) =>
                   compileType tau
               | _ => die "compileExp: wrong type info"
        val regvars =
            regvarsFromRegvarsAndInfoOpt regvars_opt
    in PRIM(RESET_REGIONSprim
              {instance = tau', regvars = regvars},
            [arg'])
    end
```

The helper `regvarsFromRegvarsAndInfoOpt` extracts the list of `regvar` values from the optional annotation on the application. If no explicit regions are supplied, the list is empty. The `instance` field carries the type of the argument, used later to extract inferred regions, and the new `regvars` field carries the explicit region variables as a list of symbolic names. At this point, the region variables are unresolved, they are simply names from the source program.

Stage 2: Lambda representation (`LambdaExp.sml`)

The Lambda intermediate language defines the primitive constructors that carry the type and region information through the pipeline. We extended both reset primitives constructors with a `regvars` field:

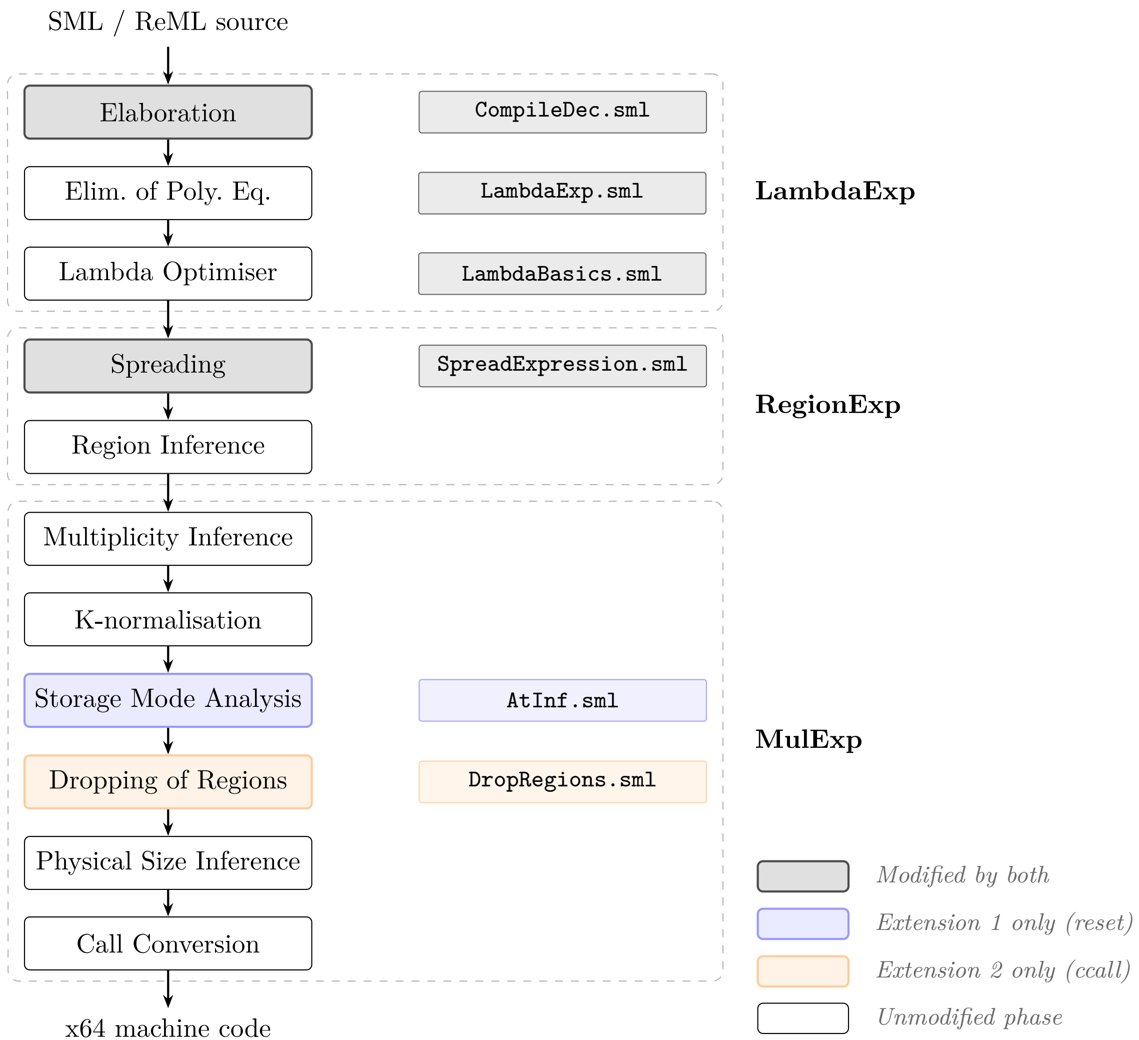

Figure 4: The MLKit compilation pipeline showing which phases were modified by the extensions described in this chapter. Solid boxes represent transformations, and arrows denote function composition. Shading indicates which extension modified each phase, as shown in the legend. Source files positioned to the right of each phase identify the implementing module, with matching shading. Dashed boxes group transformations that operate on the same intermediate language. Not all phases and files are shown. The figure focuses on the phases relevant to the extensions described in this chapter.

```
1 (* LambdaExp.sml - prim datatype (excerpt) *)
2 datatype 'Type prim =
3     ...
4   | RESET_REGIONSprim of
5       {instance : 'Type,
6        regvars   : regvar list}
7   | FORCE_RESET_REGIONSprim of
8       {instance : 'Type,
9        regvars   : regvar list}
```

Previously, both constructors carried only the `instance` field. Adding `regvars` requires updating every function in the compiler that pattern-matches on these constructors. The changes are mechanical, the `regvars` field is bound in the pattern and propagated unchanged. For example, in `LambdaBasics.sml`, the substitution and renaming function pass the field through:

```
1 (* LambdaBasics.sml - on_prim, substitution case *)
```

```
2 | RESET_REGIONSprim {instance, regvars} =>
3     RESET_REGIONSprim
4       {instance = on_Type S instance,
5        regvars = regvars}
6 | FORCE_RESET_REGIONSprim {instance, regvars} =>
7     FORCE_RESET_REGIONSprim
8       {instance = on_Type S instance,
9        regvars = regvars}
```

The substitution acts on the `instance` type but leaves `regvars` untouched. The symbolic region variable names are not affected by type substitution. They will be resolved to places in the next stage. Similarly, in `OptLambda.sml`, the equality check for primitives ignores the `regvars` field, so two reset primitives are considered equal if their instance types are equal, regardless of which explicit regions are named. This can cause the optimiser to collapse a conditional whose branches target different regions, so that both execution paths reset the same region. We discuss this further in Section 3.5.

The layout function, which is responsible for pretty-printing, is also extended to display explicit region variable when `-print_types` is enabled:

```
1 (* LambdaExp.sml - layoutPrim, reset case *)
2 | RESET_REGIONSprim {instance, regvars} =>
3     if !Flags.print_types then
4       PP.NODE{start = "resetRegions(",
5               finish = ")", indent = 2,
6               children = [layoutType instance,
7                           layoutRegVars regvars],
8               childsep = PP.RIGHT ","}
9     else PP.LEAF "resetRegions"
```

Stage 3: Region inference and spreading (`SpreadExpression.sml`)

The spreading phase is where the symbolic `regvar` names become concrete `place` nodes in the effect graph.

Original code In the unmodified compiler, the spreading case for `RESET_REGIONSprim` required the argument to be a variable and computed the effect from the argument's type alone:

```
1 (* SpreadExpression.sml - original RESET_REGIONSprim *)
2 | E.PRIM(E.RESET_REGIONSprim{instance = _},
3          [e0 as (E.VAR _)]) =>
4     let
5       val (B, t as E'.TR(e',meta0,_), _, tvs) =
6           S(B, e0, false, NOTAIL)
7       val mus0 =
8           unMus "S.RESET_REGIONSprim" meta0
9       val mu   = R.unitType
10      val phi =
11          Eff.mkUnion(
12            map Eff.mkPut(
13              List.filter Eff.is_rho
14                (R.ann_mus mus0 [])))
15    in
16      case e' of
17        E'.VAR{il_r as ref il, ...} =>
18          (case R.un_il (#1 il) of
19            ([],[],[]) =>
20              (B, E'.TR(
21                 E'.RESET_REGIONS(
```

```
22                       {force = false,
23                        regions_for_resetting = []},
24                       t),
25                     E'.Mus [mu], phi),
26                   NOTAIL, tvs)
27              | _ => crash_resetting false)
28          | _ => crash_resetting false
29      end
```

The call `R.ann_mus mus0 []` extracts the place annotations from the argument's region-annotated type `mus0`, appending an empty list of extras. The result is filtered by `Eff.is_rho` to keep only region-level places, discarding effect variables, and `Eff.mkPut` creates a `put` effect for each. These effects are unified into φ, which is attached to the `RESET_REGION` expression node. The `regions_for_resetting` field is set to the empty list. The storage mode analysis populates it later by collecting the region variables that appear free in the region-annotated type scheme with place of the argument. Note that the pattern requires `e0` to be a `VAR`. Any other argument expression causes an error.

Extended code The extended version makes three changes. It resolves explicit region variables, merges them with the inferred regions, and accepts unit as an argument. Here is the full modified case:

```
1  (* SpreadExpression.sml - extended RESET_REGIONSprim *)
2  | E.PRIM(E.RESET_REGIONSprim
3           {instance = _, regvars = regvars},
4         [e0]) =>
5      let
6        val (B, t as E'.TR(e',meta0,_), _, tvs) =
7            S(B, e0, false, NOTAIL)
8        val mus0 =
9            unMus "S.RESET_REGIONSprim" meta0
10       val mu  = R.unitType
11
12       (* 1. Resolve explicit region variables *)
13       val regvars' =
14           map (fn rv =>
15                case RSE.lookupRegVar rse rv of
16                  NONE =>
17                    deepError rv
18                      ("Explicit region variable "
19                       ^ RegVar.pr rv
20                       ^ " not in scope")
21                | SOME p => p) regvars
22
23       (* 2. Merge with inferred regions *)
24       val rhos =
25           Eff.remove_duplicates
26             (List.filter Eff.is_rho
27               (R.ann_mus mus0 regvars'))
28
29       (* 3. Compute effect *)
30       val phi = Eff.mkUnion(map Eff.mkPut rhos)
31     in
32       case e' of
33         E'.VAR{il_r as ref il, ...} =>
34           (case R.un_il (#1 il) of
35             ([],[],[]) =>
36               (B, E'.TR(
37                  E'.RESET_REGIONS(
38                    {force = false,
```

```
39                        regions_for_resetting = regvars'},
40                      t),
41                    E'.Mus [mu], phi),
42                  NOTAIL, tvs)
43             | _ => crash_resetting false)
44         (* NEW: accept unit as argument *)
45         | E'.RECORD(NONE, nil) =>
46                 (B, E'.TR(
47                    E'.RESET_REGIONS(
48                      {force = false,
49                       regions_for_resetting = regvars'},
50                      t),
51                    E'.Mus [mu], phi),
52                  NOTAIL, tvs)
53         | _ => crash_resetting false
54       end
```

We walk through the three numbered steps in detail.

1. Resolve explicit region variables. Each `regvar` from the `LambdaExp` representation is looked up in the region static environment `rse` via `RSE.lookupRegVar`. This maps the source level name (e.g. `` `r ``) to the corresponding `place` node in the effect graph. If the lookup fails, the region variable is not in scope, and the compiler reports a compile time error via `deepError`. The result `regvars'` is a list of resolved `place` nodes.

2. Merge with inferred regions. The resolved places are appended to the list of places extracted from the argument's type by passing them as the second argument to `R.ann_mus`. The function appends the `regvars'` to the list of places it extracts from `mus0`, giving a combined list of all regions, both those from the value's type and those named explicitly. This combined list is then filtered by `Eff.is_rho` to keep only region variables and places, discarding effect variables; duplicates are removed by `Eff.remove_duplicates`. If the programmer names a region that also appears in the value's type, it appears only once in the merged set.

3. Compute effect The effect φ is the union of `put` effects on every region in the merged set. Including `put` effects for the explicit regions is essential. Without them, the drop-regions pass will conclude that the region is unused and remove it from the function's region parameter list, which would mean the region pointer never reaches the reset point at runtime.

Futher changes. Three additional modifications are visible in the code. First, the pattern on the argument expression `e0` is relaxed from `[e0 as (E.VAR _)]` to just `[e0]`, because the argument may now be either a variable or unit. Therefore a new match branch for `E'.RECORD(NONE, nil)` has been added to handle the unit case. `RECORD(NONE, nil)` is the compiler's internal representation of the SML value `()`. When the argument is unit, `mus0` contains only `unitType`, which has no free regions, so the merged set `rhos` consists entirely of the explicit region variables. Lastly, the resolved places are stored in the `regions_for_resetting` field of the `RESET_REGIONS` node. Previously this field was always the empty list. Now it carries the explicit places so that the storage mode analysis can inspect them.

The `FORCE_RESET_REGIONSprim` case is identical except that the `force` flag is set to `true`.

Stage 4: Storage mode analysis (`AtInf.sml`)

The storage mode analysis determines whether each region in the target set can safely be reset at the current program point, or whether the reset must be suppressed for `resetRegions` and accompanied by a warning for both `resetRegions` and `forceResetting`. The analysis uses the rules described in Section 2.6

Original code In the original compiler, the `RESET_REGIONS` case pattern-matched on a variable argument, extracted free regions from the argument's type, and reported diagnostics using the variable name:

```
(* AtInf.sml - original RESET_REGIONS case *)
| RESET_REGIONS(
    {force, liveset = SOME liveset, ...},
    tr as (TR(VAR{lvar,...}, meta, _, _))) =>
  (case meta of
    MulExp.RegionExp.Mus [mu] =>
      let
        val free_regions =
            Eff.remove_duplicates(RType.frv_mu mu)
        val (place_at_list, conflicts) =
            analyse_rhos_for_resetting
              (sme, liveset, free_regions)
        val conflicts' =
            if force then
              foldl (fn (SAT rho, acc) =>
                        FORMAL_REGION_PARAM rho
                          :: acc
                    | (_, acc) => acc)
                conflicts place_at_list
            else conflicts
      in
        case conflicts' of
            [] => ()
          | _  => warn
              (PP.reportStringTree
                (lay_report
                  (force,lvar,mu,conflicts')));
        RESET_REGIONS(
          {force = force,
           regions_for_resetting = place_at_list,
           liveset = NONE},
          sma_trip sme tr)
      end
  | _ => die "RESET_REGIONS: ...")
```

The function `analyse_rhos_for_resetting` checks each region against the live set and the region flow graph, producing a list of `(place, storage_mode)` pairs and a list of conflicts. For `forceResetting`, regions with storage mode `SAT` are reclassified as `FORMAL_REGION_PARAM` conflicts, and the compiler warns but proceeds with the reset. For `resetRegions`, any conflict causes the reset to be suppressed by assigning storage mode `attop`.

Extended code The extended version makes three changes. It incorporates the explicit regions into the analysis, generalises the pattern to accept unit arguments, and dispatches to the appropriate diagnostic format:

```
(* AtInf.sml - extended RESET_REGIONS case *)
| RESET_REGIONS(
    {force, liveset = SOME liveset,
     regions_for_resetting},
    tr as (TR(e, meta, _, _))) =>
  (case meta of
    MulExp.RegionExp.Mus [mu] =>
      let
        (* Merge explicit regions with inferred *)
        val rhos = map #1 regions_for_resetting
```

```
11          val free_regions =
12              Eff.remove_duplicates
13                (RType.frv_mu mu @ rhos)
14          val (place_at_list, conflicts) =
15              analyse_rhos_for_resetting
16                (sme, liveset, free_regions)
17          val conflicts' =
18              if force then
19                foldl (fn (SAT rho, acc) =>
20                          FORMAL_REGION_PARAM rho
21                            :: acc
22                      | (_, acc) => acc)
23                  conflicts place_at_list
24              else conflicts
25        in
26          case conflicts' of
27              [] => ()
28            | _  =>
29              (* Dispatch on argument form *)
30              case e of
31                VAR{lvar, ...} =>
32                  warn (PP.reportStringTree
33                    (lay_report_lvar
34                      (force,lvar,mu,conflicts')))
35              | RECORD(NONE, nil) =>
36                  warn (PP.reportStringTree
37                    (lay_report_regvars
38                      (force,rhos,conflicts')))
39              | _ =>
40                  die "RESET_REGIONS: argument should be a variable or unit";
41          RESET_REGIONS(
42            {force = force,
43             regions_for_resetting = place_at_list,
44             liveset = NONE},
45            sma_trip sme tr)
46        end
47    | _ => die "RESET_REGIONS: ...")
```

We describe each change:

Merging explicit regions. The explicit places from `regions_for_resetting` are extracted with `map #1` and appended to the free regions of the argument's type. The function `analyse_rhos_for_resetting` then analyses the merged set using exactly the same liveness and reachability rules as before.

Generalised pattern. The pattern on the argument triple is relaxed from `TR(VAR{lvar, ...},...)` to `TR(e,...)`, meaning the expression `e` is no longer required to be a variable, because it may also be unit.

Diagnostic dispatch. When conflicts exists, the code inspects the argument expression to choose the appropriate diagnostic format. If `e` is a `var`, the original `lay_report_lvar` produces the familiar diagnostic referencing the variable name and its type scheme. If `e` is `RECORD(NONE, nil)`, the new function `lay_report_regvars` produces a diagnostic that lists the explicit region variables directly.

The new diagnostic functions are:

```
1 (* AtInf.sml - new diagnostic for explicit regions *)
2 fun lay_header_regvars (force, regvars) =
3     if force
```

```
4       then PP.NODE{...,
5         children =
6           [PP.LEAF "You have requested resetting the regions ",
7            PP.LEAF ("{" ^ regvars ^ "}."),
8            PP.LEAF "I have done as you requested, but I cannot guarantee that
        it is safe.",
9            PP.LEAF "Here are my objections ..."]}
10      else PP.NODE{...,
11        children =
12          [PP.LEAF "You have suggested resetting the regions ",
13           PP.LEAF ("{" ^ regvars ^ "}.")]}
14
15  fun lay_report_regvars (force, rhos, conflicts) =
16      let
17        val regvars =
18            String.concatWith ", "
19              (map Eff.pp_eff rhos)
20        val head =
21            if force
22            then "forceResetting(" ^ regvars ^ "): "
23            else "resetRegions(" ^ regvars ^ "): "
24      in
25        PP.NODE{start = head,
26                finish = "",
27                indent = 3,
28                childsep = PP.NOSEP,
29                children =
30                  lay_header_regvars(force, regvars)
31                  :: lay_conflicts(force, conflicts)}
32      end
```

For `resetRegions`, the header reads "You have suggested resetting the region {...}" and the analysis explains why it has not done so. For `forceResetting`, the header reads "You have requested resetting the regions {...}. I have done as you requested, but I cannot guarantee that it is safe," followed by the conflict details. The conflict details themselves, are produced by the unchanged `lay_conflicts` function and are identical in both the variable and unit-argument cases.

Example: revisiting the shrink/loop problem

Recall the `shrink`/`loop` example from Section 2.7. With the explicit region extension, the programmer can rewrite `loop` to name the input list region explicitly and reset it with the unit argument form:

```
1 fun loop `[r] [] = 0
2   | loop `[r] (xs : int list`r) =
3       let val xs' = shrink xs
4       in  resetRegions `[r] ();
5           1 + loop xs'
6       end
```

Here `` `r `` is an explicit region parameter to `loop`, and ``resetRegions `[r] ()`` targets exactly that one region using the unit argument. Compiling with `-reml -Pdre` reveals the intermediate code:

```
1 fun loop at r1 [`r:1] var2 =
2     case var2 of
3       nil => 0
4     | _ =>
5       let region r41:INF
6           val xs' = shrink[r41] var2
7           val _ = resetRegions [`r] ()
```

```
8        in   1 + loop[r41] xs'
9        end
```

Several things are visible in the intermediate code. `loop`'s formal region parameter is the explicit name `` `r ``, rather than a compiler generated name like `r39`. This is a consequece of ReML's explicit region parameter syntax [21]. What is new is the reset expression ``resetRegions [`r] ()``. It is a reset with an explicit region argument and unit as the value argument, which the unmodified compiler would not accept. Furthermore, we see that `shrink` still produces `xs'` in a fresh region `r41:INF`, and the recursive call `loop[r41] xs'` instantiates `` `r `` to `r41`, creating the same region flow graph as before. This is expected, since out extension does not modify the region flow graph construction or the region inference algorithm.

Indeed, the storage mode analysis detects the same conflict and suppresses the reset:

```
1  *** Warnings ***
2  resetRegions(`r):
3    You have suggested resetting the regions
4    {`r}.
5    I have NOT done as you requested.
6    (1) '`r': there is a conflict with the
7        locally live variable
8          xs' : (int, [r41]) list
9        from which the following region variables
10       can be reached in the region flow graph:
11         {r41}
12       Amongst these, 'r41' can also be reached
13       from '`r'.
14       Thus I have given '`r' storage mode
15       "attop".
```

The diagnostics uses the new format produced by `lay_report_regvars`. The conflict is precisely the one described in Section 2.7.

This confirms that `resetRegions` with explicit region arguments still respect the storage mode analysis. To actually perfrom the reset in this situation, the programmer must switch to `forceResetting` as mentioned.

The key improvements over the original situation are twofold. First, it is targeted. Only region `` `r `` is reset, not all regions in the type of some value. If `xs` had a complex type involving regions the programmer did not want to reset, those regions would be left untouched. Secondly, the reset does not require constructing a variable whose type happens to mention the target region.

The ability to target specific regions has several practical benefits. A compound data structure often spans multiple regions, some of which may be shared with other live values or may still contain data needed later in the computation. Targeted reset allows the programmer to reclaim only the regions that are genuinely dead, leaving the shared or still live regions untouched. This also enables finer grained manual memory management decisions. For example, in a long running loop, not all regions accumulate garbage at the same rate, and some may not be worth resetting in a given iteration. Furthermore, when the storage mode analysis approves resetting some of a value's regions but rejects others, targeting lets the programmer use `resetRegions` on the safe regions. Finally, it makes it possible to reset regions that are not reachable from any single value's type, for instance a region passed as a parameter from a wider scope but not mentioned in any local variable's type scheme.

3.3 Extension 2: Explicit Region Arguments for C Calls

This section describes the second compiler extension, that allows the `prim` mechanism to accept explicit region parameters. This extension is a prerequisite for the `Region` module, which needs to pass region pointers to C runtime functions in order to query or manipulate individual regions.

Motivation

The MLKit's C call mechanism, accessed through `prim` in source code and represented as `CCallprim` in the Lambda intermediate language, allows SML programs to call C functions. Region pointers are passed to C functions automatically when their function's return type requires allocation into regions. For example, a C function returning a string receives a region pointer for the string region. These automatically provided region pointers are collected in a list called `rhos_for_result` during the spreading phase.

However, some C functions need a region pointer, not because they allocate into the region, but because they inspect or operate on it. For example, the `Region` module's `isAtbot` function needs a region pointer to read the region's allocation pointer, and `resetRegion` needs one to clear the region's page list. These regions do not appear in the function's return type, so the existing mechanism provides no way to pass them.

The extension solves this problem by allowing explicit region variables on `prim` calls, using the same `` `[...] `` syntax as for reset primitives:

```
prim `[r] ("is_Atbot", ())
prim `[r] ("resetRegion", ())
prim `[r] ("get_Region_Memory_Usage_Bytes", ())
```

The explicit region variable `` `r `` is resolved in the region static environment, and the corresponding region pointer is prepended to the C function's argument list at code generation. On the C side, the function receives the region pointer as its first argument.

Pipeline overview

Figure 4 shows the files modified for this extension alongside those modified for Extension 1. Three of the four files overlap. `CompileDec.sml` is extended to extract and forward explicit region variables for `prim` calls, `LambdaExp.sml` adds a `regvars` field to the `CCALLprim` constructor, with corresponding changes to pretty-printing and pickling, and `SpreadExpression.sml` resolves the explicit region variables and prepends them to the C call's `rhos_for_result` list so that the region pointers reach the C function at code generation. The propagation through `LambdaBasics.sml` and `OptLambda.sml` is mechanical, as with Extension 1. The one file unique to this extension is `DropRegions.sml`, which is modifed to prevent the drop regions phase from removing explicit region parameters that would otherwise be considered unused. We now walk through each stage in the order the compiler processes them.

Stage 1: Elaboration to Lambda (`CompileDec.sml`)

In the unmodified compiler, the function `compile_application_of_prim` handles `prim` calls. When the primitive name is not recognised as a built in opeation, it falls through to the function `compile_application_of_c_function`, which constructs a `CCALLprim` node. We extend both functions to accept and propagate the optional `regvars_opt` annotation:

```
(* CompileDec.sml - compile_application_of_c_function *)
and compile_application_of_c_function
      env info name args regvars_opt =
  ...
  let ...
      val regvars =
          regvarsFromRegvarsAndInfoOpt regvars_opt
  in
    TLE.PRIM(CCALLprim
      {name = name,
       tyvars = tyvars_fresh,
       regvars = regvars,
       Type = on_Type subst tau,
```

```
14        instances = ...},
15     map (compileExp env) args)
16   end
```

As with the reset extension, the `regvars` list is empty when no explicit regions are supplied, preserving backwards compatibility.

Stage 2: Lambda representation (`LambdaExp.sml`)

The `CCALLprim` constructor gains a `regvars` field:

```
1 (* LambdaExp.sml - CCALLprim constructor *)
2 | CCALLprim of
3     {name       : string,
4      instances  : 'Type list,
5      regvars    : regvar list,
6      tyvars     : tyvar list,
7      Type       : 'Type}
```

As with the reset primitives, all downstream passes that pattern match on `CCALLprim`, in `LambdaBasics`, `OptLambda`, `EliminateEq`, `LambdaStatSem`, are updated to propagate the field. The pickling function is also extended to serialise the `regvars` list for separate compilation. The pretty-printing function is extended to display the explicit region variables alongside the type instances when `-print_types` is enabled:

```
1  (* LambdaExp.sml - layoutPrim, CCALLprim case *)
2  | CCALLprim {name, instances, regvars,
3              tyvars, Type} =>
4      if !Flags.print_types then
5        let val layout_instances =
6                PP.NODE{start = "<", finish = ">",
7                        indent = 2,
8                        children =
9                          map layoutType instances,
10                       childsep = PP.LEFT ", "}
11       in PP.NODE{start = "ccall (" ^ name ^ " ",
12                  finish = ")", indent = 2,
13                  children = [layout_instances,
14                              layoutRegVars regvars],
15                  childsep = PP.RIGHT ","}
16       end
17     else ...
```

Stage 3: Region inference and spreading (`Spreadexpression.sml`)

The spreading phase for `CCALLprim` is where the explicit region variables become concrete region pointers that will reach the C function. To understand the change, we first describe the data structure that carries region pointers through to code generation.

The `rhos_for_result` list. When the compiler spreads a C call, it computes `rhos_for_result`, which is a list of `(place, int option)` pairs representing the regions that the C function needs as arguments. In the unmodified compiler, this list is populated automatically from the function's return type. At code generation, the region pointers in this list are prepended to the C function's value arguments, so the C function receives them as its first arguments.

Original code. In the original spreading case for `CCALLprim`, the `rhos_for_result` list is computed from the return type alone:

```
(* SpreadExpression.sml - original CCALLprim (key lines) *)
| E.PRIM(E.CCALLprim
    {name, instances, tyvars, Type}, es) =>
  ...
  val rhos_for_result =
      if length es = 0
      then R.c_function_effects0 mu_r
      else R.c_function_effects (sigma, mu_r)

  val e' = E'.CCALL(
    {name = name, mu_result = mu_r,
     rhos_for_result = rhos_for_result}, trs')
  in
    retract(B, E'.TR(e', E'.Mus [mu_r],
      Eff.mkUnion(eps_phi0 :: phis)),
    NOTAIL, tvs)
  end
```

The function `R.c_function_effects` inspects the return type `mu_r` and collects all the region places that the C function will need to allocate its result. Each place is paired with an optional multiplicity.

Extended code. The extended version adds the same three-step pattern as in the reset extension 3.2, but adapted for the C call context:

```
(* SpreadExpression.sml - extended CCALLprim *)
| E.PRIM(E.CCALLprim
    {name, instances, regvars, tyvars, Type}, es) =>
  ...
  (* 1. Resolve explicit region variables *)
  val regvars' =
      map (fn rv =>
          case RSE.lookupRegVar rse rv of
            NONE =>
              deepError rv
                ("Explicit region variable "
                 ^ RegVar.pr rv
                 ^ " not in scope")
          | SOME p => (p, NONE)) regvars

  (* 2. Compute effect for explicit regions *)
  val rhos =
      Eff.remove_duplicates (map #1 regvars')
  val regvars_phi =
      Eff.mkUnion(map Eff.mkPut rhos)

  ...
  val rhos_for_result = ...  (* as before *)

  (* 3. Prepend explicit regions to rhos_for_result *)
  val e' = E'.CCALL(
    {name = name, mu_result = mu_r,
     rhos_for_result =
       regvars' @ rhos_for_result}, trs')
  in
    retract(B, E'.TR(e', E'.Mus [mu_r],
      Eff.mkUnion(eps_phi0
                  :: regvars_phi :: phis)),
    NOTAIL, tvs)
  end
```

The differences from the reset extension are worth noting.

Each resolved place is wrapped as `(p, NONE)` rather than stored as a bare place. This matches the type of `rhos_for_result`. We pass `NONE` because explicit region parameters have no statically known multiplicity.

The explicit regions are prepended to `rhos_for_result` with list append rather than merged into a set. Order matters here, since the code generator emits region pointer arguments in the order they appear in this list, so the explicit regions arrive as the first arguments to the C function, before any automatically inferred region pointers. On the C side, the function signature must match. For a function with one explicit region parameter, the first argument is the explicit region pointer.

The `put` effects for the explicit regions, `regvars_phi`, are included in the effect union alongside the function's latent effects, `esp_phi0`, the effects of the argument expression, `phis`. As with the reset extension, this addition prevents later passes from concluding that the explicit region is unused and removing it.

Stage 4: Dropping of regions (`DropRegions.sml`)

The drop regions phase removes unnecessary region parameters from function definitions and their call sites. The central function is `drop_places`, which decides which of a function's formal region parameters to keep. It works by marking regions that have `put` effects in the function's arrow effects, then producing a boolean list indicating which formals were visited. Unvisited formals are dropped, so their region pointers are not passed at runtime.

To understand the problem, we need to explain how region parameters are dropped for named functions. In the MLKit's intermediate language, all `fun` declarations, whether recursive or not, compile to `FIX` nodes, which is the construct for region polymorphic function definitions. When the drop regions phase encounters a `FIX` node, it calls `drop_places` on each function's formal region parameters:

```
(* DropRegions.sml - tr_function, inside FIX *)
fun tr_function {lvar, rhos_formals = ref formals,
                 epss, ...} =
  let
    val rhos = map #1 formals
    val bool_list = drop_places(rhos, epss)
    val formals' = filter_bl(bool_list, formals)
    val drop_formals' =
        filter_bl(map not bool_list, formals)
    ...
  in (lvar, bool_list, formals', ...)
  end
```

The `bool_list` is then stored in the environment as `FIXBOUND bool_list`. At call sites, the same boolean list is used to filter the actual region arguments:

```
(* DropRegions.sml - APP case *)
| SOME (FIXBOUND bool_list) =>
    let val actuals' = filter_bl(bool_list, actuals)
    ...
```

So if `drop_places` decides to drop a formal region parameter, the corresponding actual region argument is also removed at every call site.

The problem: `BOT_RT` regions. The function `drop_places` contains the following sequence:

```
(* DropRegions.sml - drop_places, original *)
fun drop_places (places, arreffs) =
  let
    ...
    val _ = visit_put_rhos arreffs
```

```
6     val _ = unvisit_bot_rhos places
7     val bl = map is_visited places
8   in reset_bucket(); bl
9   end
```

After `visit_put_rhos` marks all regions that has a `put` effect, `unvisit_bot_rhos` unconditionally unmarks any region whose runtime type is `BOT_RT`. To understand why this causes a problem, we need to explain how runtime types are assigned to explicit region variables.

Every region in the MLKit has a runtime type that describes what kind of value it holds, `STRING_RT` for strings, `PAIR_RT` for pairs, `TOP_RT` for closures and general records, and so on. The runtime type `BOT_RT` is a special case. It is assigned to explicit region variables that do not yet correspond to any value type [23].

When an explicit region variable is first introduced, either by a `with` declaration or as a function parameter, the spreading phase creates a fresh region node with runtime type `BOT_RT` via `Eff.freshRhoEpsRegVar`. If a value is subsequently allocated into the region through an explicit annotation, like `val s : string`r = "hello"`, the spreading phase promotes the runtime type from `BOT_RT` to the type appropriate for the stored record, in this example `STRING_RT`. Concretely, the function `maybe_explicit_rho` in `SpreadExpression.sml` checks the region's current runtime type. If it is `BOT_RT`, it calls `Eff.setRunType rho rt` to set it to the correct type. If it is already a non`BOT` type, it verifies that the types match [21]. If the region is never used for allocation, the runtime type remains `BOT_RT`.

In the standard compiler, without our extension, a `BOT_RT` region is never used at runtime, so `unvisit_bot_rhos` correctly removes it from the argument list.

The problem arises for functions like `Region.isAtbot` in our `Region` module:

```
1 fun isAtbot `[r] () = prim `[r] ("is_Atbot", ())
```

Here `` `r `` is a formal region parameter of `isAtbot`. Because no value is allocated into `` `r ``, its runtime type is never promoted from `BOT_RT`. During spreading, we added a synthetic `put` effect for `` `r ``, so `visit_put_rhos` correctly marks it. but `unvisit_put_rhos` then immediately unmarks it because it has runtime type `BOT_RT`. The result is that `` `r `` is dropped from `isAtbot`'s formal parameters, and every call site the actual region argument is filtered out.

This does not affect existing ReML programs where explicit region variables appear in the function's type. For example, in ``fun copyString `[r1 r2] (s: string`r1) : string`r2 = ...``, both `` `r1 `` and `` `r2 `` are promoted to `STRING_RT` because each appears in a type annotation associated with a value. Since neither has runtime type `BOT_RT`, `unvisit_bot_rhos` does not touch them. Our extension is the first case where a `BOT_RT` region genuinely needs to survive as a formal parameter, because it is passed to a C function for inspection rather than associated with any value.

Note that using ``prim`[r] ("is_Atbot())`` directly works without any change to `DropRegions`, because the `CCALL` node handles its `rhos_for_result` list independently. The problem only manifests when the `prim` call is inside a function whose formal region parameter has type `BOT_RT`.

The fix We comment out the `unvisit_bot_rhos` call:

```
1 (* DropRegions.sml - drop_places, extended *)
2 fun drop_places (places, arreffs) =
3   let
4     ...
5     val _ = visit_put_rhos arreffs
6     (* val _ = unvisit_bot_rhos places *)
7     val bl = map is_visited places
8   in reset_bucket(); bl
9   end
```

With this change, any region that has a `put` effect will remain visited and its pointer will be passed at runtime. We believe this change does not cause unnecessary region pointers to be passed in practice. The only way a `BOT_RT` region acquires a `put` effect is through our extension, where we add

synthetic `put` effects. In both cases, the region pointer is genuinely needed at runtime. A function with an explicit region parameter that is simply unused, like `fun f `[r] (x : int) : int = x + 1`, does not acquire a `put` effect for that region, so `visit_put_rhos` does not visit it and it is still correctly dropped regardless of our change. We discuss this further in Section 3.5.

3.4 Soundness Argument

In this section we argue informally that neither extension compromises the soundness properties of the existing region system. We treat each extension in turn, then address the `DropRegions` change that are shared between them.

Extension 1: explicit region arguments for reset primitives

The existing `resetRegions` primitive collects the free regions of its argument's type and submits them to the storage mode analysis in `AtInf.sml`. For each region in the target set, `AtInf` checks whether any locally live variable holds a reference into that region. If a conflict is found, the region is not reset and a warning is emitted. If `forceResetting` is used, the region is reset regardless and the warning is upgraded to note that safety cannot be guaranteed [23].

Our extension adds explicit region variables to the target set before it reaches the storage mode analysis. The explicit regions are merged with the inferred regions and then submitted to exactly the same `AtInf` conflict analysis. We do not bypass, weaken, or modify this analysis. Thus, for `resetRegions`, the same safety invariant holds. A region is only reset if the storage mode analysis determines that no locally live variable references it. Adding explicit regions to the target set can only increase the set of regions that the storage mode analysis inspects, never circumvent the check. For `forceResetting`, the programmer already accepts responsibility for correctness. Our extension does not change this contract, it just allows the programmer to name additional regions to force reset.

The synthetic `put` effects that we add for explicit regions during spreading ensure that the region variables are not dropped before the reset expression is reached. These effects are an over-approximation in the sense that they do not correspond to actual writes. However, the region and effect type system tracks them through type inference like any other effect, ensuring that each region with a synthetic `put` effect is still live at the point where the reset or C call occurs. The worst case consequence is that a `letregion` binding that could otherwise have been discharged earlier is kept alive longer, which may delay deallocation but cannot cause a dangling pointer.

Extension 1: unit argument

The original `resetRegions` and `forceResetting` require a variable as argument, and the target set of regions to reset is derived from the free regions of that variable's type. Our extension additionally allows `()` as the argument expression for reset primitives. When unit is used, the argument's type contains no free regions, so the target set is determined entirely by the explicit region annotations.

This aspect does not weaken the safety analysis. The explicit regions still pass through `AtInf`'s conflict checking, which examines the storage mode and live variable information for each region. The change simply gives the programmer a way to reset a specific set of named regions without needing to find a variable whose type happens to mention them. The responsibility for naming the correct regions shift to the programmer, but the compiler's safety check for `resetRegions`, and the documented unsafety of `forceResetting`, remain unchanged.

Extension 2: explicit region arguments for C calls

The ccall extension passes region pointers to C functions via the `rhos_for_result` list. From the perspective of the region type system, the relevant question is whether the extension can cause a region to be used in a way that violates the region typing rules.

The synthetic put effects added during spreading serve the same purpose as in Extension 1. They prevent the region from being removed by the storage mode analysis before the C call. As argued above, this is a conservative over-approximation that cannot introduce unsoundness.

What happens on the C side is, of course, outside the scope of the SML type system's guarantees. A C function that receives a region pointer could in principle corrupt the region's internal data structures. This is not a new concern. The existing `prim` mechanism already allows arbitrary C functions to be called, and soundness of C code is the programmer's responsibility. Our extension does not introduce any new avenue for unsoundness beyond what already exists in the FFI.

The `DropRegions` change

As describe in Section 3.3, we commented out `unvisit_bot_rhos` call in `drop_places`. This change is conservative, since it may cause region pointers to be passed to functions in cases where the original code would have dropped them. Passing an extra region pointer that is not used by the function body is harmless, it occupies a register or stack slot, but does not affect the function's semantics or the integrity of the region it points to.

As we argued in Section 3.3, we believe this situation does not arise in practice, because the only way a `BOT_RT` region acquires a `put` effect is through our extension, and in both cases the pointer is genuinely needed.

3.5 Limitations

Only explicit named regions can be targeted

Both extensions require explicit region variables introduced by `let with` declarations or as function parameters. The programmer cannot target compiler-inferred regions without restructuring the code to name them explicitly. A mechanism for discovering and naming inferred regions would require a different approach, and will be discussed as future work in Chapter 8.

OptLambda equality check does not compare region variables

The function `eq_prim` in `OptLambda.sml` determines when two primitive expressions are equivalent. This equality is used in several optimisations. Switch elimination removed a conditional when all branches produce equal expressions. Contract analysis tracks known values through the program and uses them to simplify expressions, for example by propagating a constant or replacing a variable with its known definition. These rely on `eq_prim` to decide whether two primitives are the same.

For reset primitives and `CCALLprim`, the comparison checks the `instance` type, and for C calls also the name and type scheme, but does not compare the `regvars` field:

```
1 (* OptLambda.sml - eq_prim, current *)
2 | (RESET_REGIONSprim {instance=t, regvars=rvs},
3    RESET_REGIONSprim {instance=t', regvars=rvs'})
4     => eq_Type(t, t')
5
6 | (CCALLprim {name=n, instances=il, regvars=rvs, tyvars=tvs, Type=t},
7    CCALLprim {name=n', instances=il', regvars = rvs', tyvars=tvs', Type=t
   '})
8     => n = n' andalso eq_Types(il, il')
9        andalso eq_sigma((tvs, t), (tvs', t'))
```

This causes incorrect behaviour. Consider a conditional where each branch resets a different explicit region:

```
1 if b then forceResetting `[r1] ()
2      else forceResetting `[r2] ()
```

The optimiser considers both branches equal because they have the same `instance` type and `regvars` is not compared. It eliminates the conditional and keeps only one branch. Inspection of the intermediate code confirms that both paths produce the same `forceResetting [atbot `r2] ()`, meaning `r1` is never reset. The same behaviour occurs when the two branches contain direct `prim` calls with different explicit region variables.

However, the bug does not manifest when the C call is made through a normal SML function, as is the case for the `Region` module. A call like `Region.memoryUsageOfRegion `[r1] ()` compiles to a function application `APP(VAR{regvars=[r1],...}, ...)` at the Lambda level. The equality function `eq_lamb0` handles the `VAR` case separately and does compare region variables via `eq_regvars`. Two calls with different explicit region variables are therefore distinguished, and the optimiser preserves the conditional.

The bug affects only primitives that appear directly as `prim` nodes in the syntax tree, which is the case for the built-in `resetRegions` and `forceResetting` primitives and for direct `prim` calls. The fix is to extend `eq_prim` to include the `regvars` field in the comparison for these cases. Until this is fixed, programs that branch on which region to reset or query must be compiled with the Lambda optimiser disabled.

Finite regions are promoted infinite

The synthetic `put` effects added during spreading cause multiplicity inference to assign multiplicity ∞ to explicit region parameters, so they are always heap-allocated as infinite regions. This means the `Region` module cannot be used to inspect finite regions. One possible solution would be to introduce a new atomic effect, distinct from `put`, that indicates a region is accessed without implying a write. Such an effect would keep the region alive without inflating its multiplicity. This would allow operations like `memoryUsageOfRegion` to report the the size of finite regions (just word size), and would allows us to query whether a region is finite or infinite. The current limitation stems from the pipeline rather than a fundamental design constraint, and we consider lifting it in future work 8.1.

4 Modules for Region Operations

The compiler extensions describe in Chapter 3 make it possible to pass explicit region pointers to C functions via the `prim` mechanism. In this chapter we build on that foundation to provide a typed SML library, the `Region` module, that gives programmers direct access to region metadata. We also present the `Size` module, a combinator library for computing the memory footprint of SML values.

4.1 Design Rationale

Using `` prim `[r] ("c_function", ()) `` directly is possible but inconvenient. The programmer must know the exact C function name, remember to pass `()` as the dummy argument, and has no SML-level type check support. A mistake in the function name produces a linker error, in the best case, rather than a type error, and the intent of the call is obscured by FFI boilerplate.

The `Region` module addresses this by wrapping each C function in a named SML function with a clear type. The module's operations fall into two groups. Per region operations accept an explicit region variable and report on or modify that specific region. Global operations take no region argument and report system-wide allocation state by reading counters maintained by the runtime allocator. Together, these operations let a programmer observe how memory is distributed across regions, decide which regions to reset, and verify that a reset had the intended effect.

The `Size` module complements the `Region` module from a different angle. Where `Region` queries the runtime allocator, `Size` computes the memory footprint of a value structurally. The programmer builds a size descriptor that mirrors the shape of a data type, and applies it to a value to obtain a byte count. This is useful when a programmer wants to know how much memory a particular value occupies, rather than how much memory an entire region uses. By summing the sizes of all values known to reside in a region, the programmer can estimate how much of the region's allocated memory is live data, and by comparing this to the total reported by `memoryUsageOfRegion`, determine how much is waste eligible for reclamation by copying the live data to a fresh region and resetting the original.

4.2 The REGION Signature

The `REGION` signature defines the public interface of the module. Every function takes `unit` as its value argument. The per region operations additionally accept an explicit region variable using the `` `[r] `` syntax. The full signature is shown in Figure 5

```
signature REGION =
  sig
    (* Per-region operations *)
    val resetRegion          : unit -> unit   (* `[r] *)
    val isAtbot              : unit -> bool   (* `[r] *)
    val numPagesOfRegion     : unit -> int    (* `[r], pages *)
    val memoryUsageOfRegion : unit -> int    (* `[r], bytes *)

    (* Global runtime operations *)
    val getPageSizeBytes            : unit -> int   (* bytes *)
    val getNumAllocatedPages        : unit -> int   (* pages *)
    val getFreeListSize             : unit -> int   (* pages *)
    val getThreadFreeListSize       : unit -> int   (* pages *)
    val giveThreadFreeListToGlobal : unit -> unit
  end
```

Figure 5: The `REGION` signature. Comments indicate which functions assume an explicit region variable and the unit of the returned value.

Per region operations

Each per region operation is called with an explicit region variable and unit, for example `Region.isAtbot `[r] ()`. The region variable must be in scope at the call site, introduced either by a `let with` declaration or as a function parameter.

**`resetRegion `[r] ()`** resets the named region by calling the C runtime's `resetRegion` directly, bypassing the built-in `resetRegions` primitive and its AtInf safety analysis. This means the operations is not guaranteed to be sound. If live references into the region exist, the reset proceeds anyway without any compiler warning. The function is intended for situations where the programmer is sure that the region contains no live data.

**`isAtbot `[r] ()`** returns `true` if the `atbot` status bit is set on the region's runtime representation. This bit is set by the storage mode analysis when it determines that the region contains no live data at a particular program point, and that the next allocation may safely reset the allocation pointer to the bottom of the region.

`numPagesOfRegion `[r] ()` returns the number of pages currently allocated to the region. Each page is a fixed size block (currently 8Kb) obtained from the runtime's free list. A freshly created region has one page.

`memoryUsageOfRegion `[r] ()` returns the memory usage of the region in bytes. The value is computed as the total number of pages multiplied by the page size, minus the free space remaining in the region's last page.

Global runtime operations

The global operations take no region argument and report on the state of the runtime allocator as a whole.

`getPageSizeBytes ()` returns the fixed size of a single region page in bytes. This value is a compiler time constant of the runtime system.

`getNumAllocatedPages ()` returns the total number of pages that have been obtained from the operating system, including pages in the free list. This function counts only region pages, it does not include the runtime stack, code segments, or other runtime bookkeeping.

`getFreeListSize ()` returns the number of pages in the global free list. When a region is deallocated, its pages are returned to this list for reuse. The number of pages currently in use by regions can be computed as `getNumAllocatedPages () - getFreeListSize ()`.

`getThreadFreeListSize ()` returns the number of pages on the thread local free list. In single threaded programs, this is equivalent to `getFreeListSize`. In programs compiled with `-par`, each thread maintains a local free list so that most allocations and deallocations can proceed without acquiring the lock on the global free list.

`giveThreadFreeListToGlobal ()` splices the thread local free list into the global free list under a lock. In single threaded programs this does nothing. The operation is useful for long-running threads that remain alive but have finished a phase of heavy allocation. Returning the thread's cached free pages to the global list makes them available to other threads.

4.3 Implementation: Region.sml and the C runtime backing (Region.c)

The `Region` module is implemented in two layers. A thin SML wrapper that uses the extended `prim` syntax from Chapter 3, and a set of C functions in the MLKit runtime, in `Region.c`, that read or modify the region data structures.

SML layer in `Region.sml`

Each function in the module is a one-line wrapper around a `prim` call. Per region operations use the extended syntax `` prim `[r] ("c_function", ()) ``, which passes the explicit region pointer as the first argument to the C function. Global operations use the standard `prim ("c_function", ())` syntax with on region argument. The full implementation is shown in Figure 6.

```
structure Region : REGION =
  struct
    fun resetRegion `[r] () = prim `[r] ("resetRegion", ())

    fun isAtbot `[r] () = prim `[r] ("is_Atbot", ())
    fun numPagesOfRegion `[r] () = prim `[r] ("num_Pages", ())
    fun memoryUsageOfRegion `[r] () = prim `[r] ("
   get_Region_Memory_Usage_Bytes", ())

    fun getPageSizeBytes () = prim ("get_Page_Size_Bytes", ())
    fun getNumAllocatedPages () = prim ("get_Num_Allocated_Pages", ())
    fun getFreeListSize () = prim ("get_Free_List_Size", ())

    fun getThreadFreeListSize () = prim ("get_Thread_Free_List_Size", ())
    fun giveThreadFreeListToGlobal () = prim ("
   give_Thread_Free_List_To_Global", ())
  end
```

Figure 6: The `Region` structure. Per-region operations use the `` prim `[r] `` syntax; global operations use the standard `prim` syntax.

The module is packaged as part of the ReML basis library. The file `reml.mlb` bundles `REGION.sig` and `Region.sml` and is included by `basis-reml.mlb` for sequential programs, and `par-reml.mlb` for parallel programs. User programs gain access to the module by referencing one of these MLB files.

C layer in `Region.c`

The C functions are added to `Region.c`, the existing runtime file that implements region allocation and deallocation. They divide into the same groups as their SML counterparts.

Per-region functions. Each per-region function receives the region pointer as its first argument. The pointer carries status bits in its low-order positions, used by the runtime to encode storage mode information, so most functions begin by clearing these bits with the existing `clearStatusBits` before accessing the region descriptor. The functions are declared using the existing `REG_POLY_FUN_HDR` macro, which adapts the function signature for profiling builds:

```
#ifdef PROFILING
#define REG_POLY_FUN_HDR(name, ...) \
    name ## Prof(__VA_ARGS__, size_t pPoint)
#else
#define REG_POLY_FUN_HDR(name, ...) \
    name(__VA_ARGS__)
#endif
```

In non-profiling builds, `REG_POLY_FUN_HDR(is_Atbot, Region r)` expands to `is_Atbot (Region r)`. In profiling builds, it expands to `is_AtbotProf(Region r, size_t pPoint)`, where `pPoint` identifies the source location of the call, enabling the profiler to attribute the operation to a specific program point.

Since our C functions only read region metadata and do not allocate into regions, the `REG_POLY_FUN_HDR` macro is all that is needed. The same function body works in both profiling and non-profiling builds [23]. If a C function were to allocate into a region, the function body would additionally need to use the profiling versions of the runtime's allocation macros (e.g. `allocRecordMLProf` instead of `allocRecordML`) passing `pPoint` to each allocation call so that the profiler can attribute the allocated bytes to the correct program point.

`is_Atbot` checks the `atbot` status bit directly on the region pointer using the existing `is_atbot` macro and returns the result as a tagged ML boolean via the existing `convertBoolToML` macro:

```
size_t REG_POLY_FUN_HDR(is_Atbot, Region r) {
  return convertBoolToML(is_atbot(r));
}
```

`num_Pages` clears the status bits using the existing `clearStatusBits` macro and traverses the region's page list via the existing `NoOfPagesInRegion` helper:

```
size_t REG_POLY_FUN_HDR(num_Pages, Region r) {
  Region r_cleared = clearStatusBits(r);
  return convertIntToML(NoOfPagesInRegion(r_cleared));
}
```

`get_Region_Memory_Usage_Bytes` computes the memory usage as the total page space minus the free space remaining in the region's last page:

```
size_t REG_POLY_FUN_HDR(
    get_Region_Memory_Usage_Bytes, Region r) {
  Region r_cleared = clearStatusBits(r);
  return convertIntToML(
    NoOfPagesInRegion(r_cleared)
      * REGION_PAGE_SIZE_BYTES
    - freeInRegion(r_cleared)
      * WORD_SIZE_BYTES);
}
```

The existing `freeInRegion` function returns the number of free words in the region's last page, so we multiply by `WORD_SIZE_BYTES` to obtain a byte count.

`resetRegion` calls the runtime's existing `resetRegion` function directly with the region pointer, bypassing the built in `resetRegions` primitives and its safety analysis.

Global functions. The global functions take no region argument and read counters that the runtime allocator already maintains.

`get_Page_Size_Bytes` returns the compile time constant `REGION_PAGE_SIZE_BYTES`, which is currently set to 8Kb:

```
size_t get_Page_Size_Bytes () {
  return convertIntToML(REGION_PAGE_SIZE_BYTES);
}
```

get_Num_Allocated_Pages returns the existing global counter `rp_total` corresponding to the total number of allocated pages.

```
size_t get_Num_Allocated_Pages () {
  return convertIntToML(rp_total);
}
```

get_Free_List_Size returns the length of the global free list via the existing `size_free_list` function.

```
size_t get_Free_List_Size () {
  return convertIntToML(size_free_list());
}
```

get_Thread_Free_List_Size and give_Thread_Free_List_To_Global These functions are thin wrappers around new helper functions that operate on the thread local free list:

```
size_t get_Thread_Free_List_Size () {
  return convertIntToML(size_thread_free_list());
}

void give_Thread_Free_List_To_Global () {
  free_thread_free_list();
}
```

The helper `size_thread_free_list` acquires the free-list lock, traverses the list accessed via the existing `FREELIST` macro, and releases the lock. `FREELIST` resolves to the thread-local list when the runtime is compiled with `PARALLEL`, and to the global list otherwise:

```
size_t size_thread_free_list() {
  Rp* rp;
  size_t i = 0;
  MAYBE_DEFINE_CONTEXT;
  LOCK_LOCK(FREELISTMUTEX);
  for ( rp = FREELIST ; rp ; rp = rp->n )
    i++;
  LOCK_UNLOCK(FREELISTMUTEX);
  return i;
}
```

The helper `free_thread_free_list` splices the thread-local list into the global list under the same lock. The entire body is guarded by `#ifdef PARALLEL`, making it a no-op in single-threaded builds:

```
void free_thread_free_list() {
#ifdef PARALLEL
  MAYBE_DEFINE_CONTEXT;
  LOCK_LOCK(FREELISTMUTEX);
  if ( FREELIST != NULL ) {
    Rp* fl_tmp = global_freelist;
    global_freelist = FREELIST;
    Rp* last = global_freelist;
    while ( last->n != NULL )
      last = last->n;
    last->n = fl_tmp;
    FREELIST = NULL;
  }
  LOCK_UNLOCK(FREELISTMUTEX);
#endif
}
```

The function first saves the current head of the global free list in `fl_temp`. It then points `global_freelist` to the head of the thread-local list, effectively prepending the thread-local pages. It walks to the end of the (former) thread-local list and links its last page to the saved global head, concatenating the two lists. Finally, it sets `FREELIST` to `NULL`, clearing the thread-local list so that subsequent allocations in this thread will request fresh pages from the global free list.

Value conversion. All return values pass through the existing `convertIntToML` or `convertBoolToML` macros, which convert C values to the MLKit's tagged representation. When value tagging is enabled, which is on by default and required when garbage collection is enabled, unboxed integers are represented as tagged 63-bit values on 64 bit platforms, and these macros shift and tag the value appropriately.

4.4 Size Combinator

The `Size` module provides a combinator library for computing the memory footprint of an SML value as it is laid out in regions. The programmer builds a size descriptor that mirrors the shape of a data type, then applies it to a value to obtain a byte count.

Design

The key idea is the abstract type `'a sz`. In the implementation, `'a sz` is defined as `'a -> int`, which is a function that, given a value of type `'a`, returns its size in bytes, but this representation is hidden from clients. The signature is shown in Figure 7.

```
1  signature SIZE =
2  sig
3    type 'a sz
4
5    (* Primitives *)
6    val int    : int sz
7    val bool   : bool sz
8    val char   : char sz
9    val word   : word sz
10   val real   : real sz
11   val unit   : unit sz
12   val string : string sz
13
14   (* Combinators *)
15   val list   : 'a sz -> 'a list sz
16   val option : 'a sz -> 'a option sz
17   val tup2   : 'a sz -> 'b sz -> ('a * 'b) sz
18   val tup3   : 'a sz -> 'b sz -> 'c sz
19                -> ('a * 'b * 'c) sz
20
21   val size   : 'a sz -> 'a -> int
22 end
```

Figure 7: The `SIZE` signature. The type `'a sz` is abstract. Clients construct size descriptors using the primitives and combinators and apply them via `size`.

The module is sealed with an opaque signature constraint, `:> SIZE`, so the representation of `'a sz` and internal definitions such as the `word_size` constant are hidden from the client code. The function `size :` α `sz ->` α `-> int` is the only way to apply a descriptor to a value.

Primitives

The primitives handle base types. Each returns the number of bytes the value occupies in region-allocated heap space:

```
1 val word_size = 8  (* bytes, 64-bit *)
2
3 val int     = fn _ => 0
4 val bool    = fn _ => 0
5 val char    = fn _ => 0
6 val word    = fn _ => 0
7 val real    = fn _ => word_size
8 val unit    = fn _ => 0
```

Unboxed scalars, `int`, `bool`, `char`, `word`, `unit`, return 0 because they are stored inline as tagged words and do not require separate region allocation. The function `real` returns `word_size`. A real is a boxed 64-bit floating point number stored as a single word, when garbage collection is disabled, in a region of runtime type `TOP_RT` [23].

Strings require a more involved computation. A string in the MLKit is stored as a header word followed by the character data rounded up to the nearest word boundary:

```
1 fun string s =
2   let val n = String.size s
3   in  word_size                                  (* header *)
4     + (n + word_size - 1) div word_size
5         * word_size                              (* data   *)
6   end
```

Combinators

The combinators compose primitives into size descriptors for compound types. Each adds the structural overhead and recurses into the components using the supplied element descriptor.

`list`. folds over the list, adding 2 words per cons cell, one for the space to hold an element and one for the tail pointer. The initial accumulator accounts for zero words for representing the `nil` constructor:

```
1 fun list sa xs =
2   List.foldl
3     (fn (x, acc) => acc + 2 * word_size + sa x)
4     0 xs
```

`option`. `NONE` occupies one word, a tag. `SOME` adds 2 words of overhead, a header plus one field, and recurses into the content:

```
1 fun option sa opt =
2   case opt of
3     NONE   => word_size
4   | SOME x => 2 * word_size + sa x
```

`tup2` and `tup3`. Pairs and triples in the MLKit are stored without a header word, they reside in regions with runtime types `PAIR_RT` and `TRIPLE_RT`, which the runtime treats specially [23]. A pair therefore adds 2 words of overhead, one per field, and a triple adds 3 words:

```
1 fun tup2 sa sb (a, b) =
2   2 * word_size + sa a + sb b
3
4 fun tup3 sa sb sc (a, b, c) =
5   3 * word_size + sa a + sb b + sc c
```

Usage example

To compute the size of a list of string pairs, the programmer composes the combinators to match the data structure:

```
val myList = [("hello", "world"), ("foo", "bar")]

val sz = Size.size
  (Size.list (Size.tup2 Size.string Size.string))
  myList
```

The descriptor `Size.list (Size.tup2 Size.string Size.string)` mirrors the type `(string * string) list`. The `list` combinator handles the cons cells, the `tup2` combinator handles the pair overhead, and the `string` primitive computes the size of each string based on its actual length.

Limitations

Sharing. Because each combinator traverses the value structurally, shared subvalues, where the same region-allocated object is referenced from multiple places, will be counted once per reference, not once per object. This means `size` can overcount when data structures share subvalues. Standard ML does not provide pointer identity checks, but the MLKit's FFI could be used to obtain the address of a value as an integer via a small C function. We would then need to maintain a set of visited addresses during traversal, skipping values already seen. This would require changing the type of size descriptors from `'a -> int` to something like `'a -> address_set -> int * address_set`, threading state through the traversal. It is feasible but a more involved design that we leave as future work.

Extensibility. The module provides combinators for the most common compound types, but several MLKit native types are not covered. These are records and tuples with four or more field, which, unlike pairs and triples, carry a header word, references, arrays and vectors, and function closures. Each of these could be added as a new combinator following the same pattern.

For user defined datatypes, the programmer must write a custom size function that pattern-matches on the constructors and sums the overhead and component sizes. Since `'a sz` is abstract, such a function cannot be wrapped as an `'a sz` value and therefore cannot be composed with the existing combinators. There are two ways to address this. One is to add a function `fromFn : ('a -> int) -> 'a sz` to the signature, allowing the programmer to inject custom size functions into the combinator framework. The other is to relax the module's opaque signature constraint from `:> SIZE` to `: SIZE`, making the type abbreviation `'a sz = 'a -> int` visible to clients so they can construct descriptors directly as functions. We prefer `fromFn` over relaxing the constraint for two reasons. First, transparent matching permanently exposes the internal representation. If the module were later extended with, for example, metadata for sharing aware traversal or GC mode dependent sizing, changing the representation of `'a sz` would break all client code that depends on it being a plain function. Second, with an opaque constraint, any module that matches the `SIZE` signature can serve as a drop-in replacement, regardless of its internal representation. Adding `fromFn` to the signature does not prevent this. A future implementation could use a richer internal type and implement `fromFn` by wrapping the provided function into that type. With transparent matching, clients that construct `'a sz` values directly as plain functions bypass the module entirely, and a new implementation with a different internal type would break them even if the signature is unchanged.

Sizes assume GC is disabled. The module's size computations assume the non-GC representation throughout. Pairs and triples have no header, and reals occupy 1 word. When garbage collection is enabled, the runtime adds a header word to records with four or more fields, and to boxed reals, so the reported sizes will undercount. A fix would be to detect at runtime whether GC is enabled. This could be done by inspecting `Int.maxInt`. When GC is enabled, the default integer type is `Int63.int`

with `maxInt = ` $2^{62} - 1$, whereas without GC it is `Int64.int`, with `maxInt = ` 2‘63 – 1. The module could use this to add the extra header word where appropriate.

Boxity-dependent representation. The combinators assume a fixed overhead per constructor, but the MLKit's unboxing scheme [23] means that the actual representation depends on the number of constructors for the data type and the properties of each constructor. The current combinators assume the boxed representation throughout, which over-counts for low-unboxed and high-unboxed instantiations. A more precise module could carry boxity information alongside each size descriptor, which would suffice for the types currently covered by the module. For user-defined datatypes with multiple constructors, however, the MLKit determines boxity through an iterative algorithm over the full datatype declaration, which in not accessible from source-level SML.

Design sketch: a copy combinator

The same combinator pattern could be used to build a deep copy function `'a -> 'a` that copies a value into fresh regions. Each primitive would copy a base value, each combinator would allocate the structural overhead in a target region and recurse into the components.

We do not implement this in the present work, but note that a copy combinator would address one of the practical challenges of region based programming. Copying live data out of a region before resetting it requires the programmer to construct a new value in the target region. Simply returning an existing value does not allocate it into a new region, the compiler either unified the source and target regions, or, when explicit region variables are used, rejects the program because distinct explicit region variables are never unified [21]. The programmer must instead use operations that force fresh allocation. for example, `s ^ ""` to copy a string, or recursive traversal to reconstruct each cons cell of a list. A combinator library would encapsulate this knowledge, making copying composable and type safe. We disciss this further as future work in Chapter 8.

5 Testing

This Chapter describes the test suites used to validate the compiler extensions from Chapter 3 and the modules from Chapter 4. The tests are unit-style correctness tests that compile and run small programs, comparing their output, and, where relevant, the compiler log messages, against the expected results. All tests are run using the MLKits `kittester` framework, which we extended with two new options, namely `nogc`, for compilation with the `-no_gc` flag, and `parallel`, for compilation with the `-par` flag.

5.1 Testing Framework

The MLKits test runner, `kittester`, reads a `.tst` file that lists source files together with option tokens. For each entry, it compiles and runs the program, then compares the output against a `.out.ok` file. If the token `ccl` is present, the compiler log is also compared against a `.log.ok` file, which is used to verify that the compiler produces the expected warnings. The token `ecte` indicates that the test is expected to fail at compile time.

We added two options to the tester to support our use case:

```
(* Tester.sml - additions *)
(if opt "nogc" then "-no_gc " else "") ^
(if opt "parallel" then "-par " else "")
```

These allow test entries to specify GC-free and parallel compilation modes. Most of our tests are run in both `nogc` and `nogc parallel` modes to verify correct behaviour in single threaded and parallel configurations.

5.2 Reset Primitives

The reset primitive tests are located in `test/explicit_regions/` and are part of the existing ReML test suite. They verify the compiler extension from Section 3.2. The tests are organised in marched pairs, each `resetRegions` test has a corresponding `forceResetting` test. Table 1 summarises the test cases. Each test allocates a literal into an explicit region using a `with` declaration, performs a

Test	Tokens	Property
`reset.sml`	`noopt`	Resetting regions of a dead variable succeeds
`reset2.sml`	`ccl noopt`	Resetting regions of a live variable produces a warning and no reset
`reset3.sml`	`noopt`	Resetting an explicit region (`` `[r] () ``) on a dead value succeeds
`reset4.sml`	`ccl noopt`	Resetting an explicit region on a live value produces a warning and no reset
`reset5.sml`	`noopt`	Mixed: one explicit region + inferred regions, all dead - both reset
`reset6.sml`	`ccl noopt`	Mixed: one explicit region + inferred regions, all live - warning, no reset
`reset7.sml`	`noopt`	Multiple explicit regions in one reset (`` `[r1 r2] () ``)
`resetErr.sml`	`ccl ecte noopt`	Out-of-scope region variable is a compile-time error
`forceReset.sml`	`noopt`	Force-resetting a dead variable succeeds
`forceReset2.sml`	`ccl noopt`	Force-resetting a live variable produces a warning but resets anyway
`forceReset3.sml`	`noopt`	Force-resetting an explicit region on a dead value succeeds
`forceReset4.sml`	`ccl noopt`	Force-resetting an explicit region on a live value warns but resets
`forceReset5.sml`	`noopt`	Mixed explicit + inferred, all dead
`forceReset6.sml`	`ccl noopt`	Mixed explicit + inferred, all live - warns but force-resets

Table 1: Reset primitive tests. The `ccl` token enables compiler log comparison to verify warning messages. The `ecte` token marks expected compile-time errors. The `noopt` token disables the Lambda optimiser to prevent the optimiser from eliminating the test code.

reset, and then either allocates a new value in the same region, to verify that the reset succeeded,

or reads the original value, to verify the compiler correctly refused or warned about the reset. The tests use the `__bytetable_size` primitive to obtain the size of a string, which serves as a lightweight check that the value is accessible, and to prevent the optimiser from removing it.

The tests with `ccl` compare the compiler log against the expected output to verify that the AtInf safety analysis produces the correct warnings. For example, `reset2.sml` expects the compiler to report that is has not reset the region because the variable is live, while `forceReset2.sml` expects a warning that the reset was performed despite the live reference.

Scope of the reset tests. These tests verify the compiler pipeline, that the extended syntax is accepted, that explicit region variables are propagated correctly through the intermediate representations, and that AtInf produces the expected warnings. They do not verify that region content is actually reset at runtime, because the tests use literals, and the MLKits stores boxed literals as compile time constants in the program's data segment rather than in regions. As confirmed by inspecting the intermediate code, a binding like `val s : string `r = "hello"` produces no region annotation on the string, only dynamically constructed strings, from operations like `^`, `CharVector.tabulate` or arithmetic on reals, are allocated through the region system. We additionally confirmed this at runtime using the `Region` module. The function `memoryUsageOfRegion` reports no increase after binding a string literal into an explicit region. Consequently, the explicit regions in these tests receive multiplicity 0. Runtime level reset behaviour is instead tested by the `Region` module tests, which use `CharVector.tabulate` to force actual region allocation.

An implication of this observation is that the AtInf warnings produced in tests like `reset2.sml` are arguably spurious. The analysis warns about a live reference into a region that contains no data. AtInf bases its analysis on the region annotated type of the variable, not on the actual region content, which is the correct conservative approach for soundness. However, a more precise analysis could take into account that literals are not region allocated. We note this as a potential improvement in Section 5.5.

5.3 Region Module

The `Region` module tests are located in `test/region_info/` and verify the API described in Chapter 4. Each test is an MLB project that imports the `Region` module via `reml.mlb`. Every test is run in both `nogc` and `nogc parallel` modes. The tests print `OK: name` on success and `FAIL: name` on failure, and the expected output files contain the corresponding `OK` line.

Per-region operations

`resetRegion` (5 tests). The function `resetRegion_doesNotIncreaseMemoryUsage` allocates a string larger than half a page, resets, and verifies that memory usage decreased. The function `resetRegion_idempotent_memoryUsage` resets twice and verifies that the memory usage is the same after both resets. The function `resetRegion_keepsAtbot_onEmpty` resets an empty region and verifies that `isAtbot` remains true. The function `resetRegion_setsAtbot_afterAllocation` allocates, resets, and verifies that `isAtbot` becomes true. The function `resetRegion_resets_with_liveRef` allocates a string, resets while the string is still live, and verifies that memory usage decreased, confirming that `Region.resetRegion` bypasses the AtInf safety analysis.

`isAtbot` (2 tests). `isAtbot_true_initial`: a freshly created region has `isAtbot` true.
`isAtbot_false_afterAllocation`: after allocating a string, `isAtbot` is false.

`numPagesOfRegion` (2 tests). `numPages_nonnegative_initial`: page count is non-negative on a fresh region. `numPages_increasing_afterAllocation`: allocating strings that exceed one page increases the page count.

memoryUsageOfRegion (2 tests). memoryUsage_nonnegative_initial: memory usage is non-negative on a fresh region. memoryUsage_increases_afterAllocation: allocating a string increases memory usage.

Global operations

getPageSizeBytes (2 tests). The test getPageSizeBytes_positive verifies that the page size is positive. A second test, getPageSizeBytes_stable, checks that two consecutive calls return the same value.

getNumAllocatedPages (2 tests). The test getNumAllocatedPages_nonnegative verifies that the count is non-negative. A second test, getNumAllocatedPages_getFreeList, verifies that the total allocated pages is at least the free list size.

getFreeListSize (2 tests). The test getFreeListSize_nonnegative checks that the free list has at least one page. The test freeList_lAllocatedPages verifies that the free list is strictly smaller than total allocated pages.

getThreadFreeListSize (2 tests). The test getThreadFreeListSize_nonnegative verifies that the thread-local free list has at least one page. A second test, threadFree_lTotalAllocated, checks that the thread-local free list is strictly smaller than total allocated pages.

giveThreadFreeListToGlobal (6 tests). These tests verify the properties of the splice operation. The test giveBack_globalFreeList_nondecreasing checks that the global free list does not decrease. Two idempotency tests, giveBack_idempotent_globalFreeList, giveBack_idempotent_threadFreeList, verify that calling the operations twice has no additional effect on the global and thread-local free lists respectively. The test giveBack_preservesFreeListSum verifies that the sum of the two free lists is preserved, and giveBack_preservesTotalAllocatedPages checks that total allocated pages is unchanged. Finally, giveBack_threadFreeList_nonincreasing confirms that the thread-local free list does not increase.

Cross function invariant

The test region_usageWithinAllocatedPages verifies that the memory usage of a region does not exceed the number of pages times the page size. This test uses IntInf arithmetic to avoid overflow and exercises three functions together: numPagesOfRegion, memoryUsageOfRegion, getPageSizeBytes.

5.4 Size Module

The Size module tests verify the combinator library from Section 4.4. Each test is an MLB project that imports size.mlb. All tests are run in nogc mode. Table 2 summarises the test cases. The last

Test	Property
size_int_zero	Unboxed integer size is 0
size_string_positive	Non-empty string has positive size
size_list_nil_is_word	Empty list is one word (8 bytes)
size_list_grows	Longer list has larger size
size_tup2_positive	Pair of integers has positive size
size_option_some_gt_none	SOME is larger than NONE
size_leq_regionUsage	Computed size ≤ region memory usage

Table 2: Size module tests.

test, size_leq_regionUsage, bridges the Size and Region modules. It allocates a string into an

explicit region using `CharVector.tabulate`, computes its size with `Size.size Size.string`, and verifies that the result does not exceed the region's memory usage as reported by `Region.memoryUsageOfRegion`. The test uses `CharVector.tabulate` rather than a string literal because string literals are not region allocated, as discussed in Section 5.2.

5.5 Limitations and Future Testing

The current test suite validates the compiler pipeline and the runtime behaviour of the `Region` and `Size` modules, but several areas remain untested or could benefit from deeper coverage.

Runtime reset verification for the reset primitives. As discussed in Section 5.2, the reset primitive uses boxed literals that are not region allocated, so they verify compiler output and AtInf warnings but not actual runtime reset behaviour. These tests should be extended to use dynamically allocated values and the `Region` module to confirm that memory usage decreases after a reset.

Spurious AtInf warnings for string literals. AtInf can report spurious conflicts when a live variable is bound to a boxed literal such as a string or a real. These literals are stored as compile-time constants in the data segment and are never placed into the region that appears in their type. However, AtInf sees only the variables type, which associates the literal with a region variable ρ, and concludes that ρ contains live data. With `resetRegions`, this causes the storage mode to be `attop`, suppressing a reset that would in fact be safe, since no data belonging to the literal resides in ρ. With `forceResetting`, the reset proceeds correctly but the programmer sees a warning that cannot be resolved.

AtInf is a type-bases analysis by design. Its `lvar` environment maps each variable to the region variables in its region-annotated type scheme, without recording how the variable was defined. A variable bound to a string literal has type `(string,` ρ`)`, identical to a variable bound to a dynamically allocated string. Although MulExp preserves literals as distinct `STRING` and `REALS` constructors, AtInf does not inspect the binding expression when populating its `lvar` environment. A partial fix would be to check whether the right-hand side of a `LET` binding is a literal and record this in the environment, but this would only cover direct bindings. If a literal is passed as a function argument, AtInf only sees the parameter's type and cannot determine whether the actual argument is a literal or a runtime allocation.

Stress testing and memory bounds. The current tests verify basic properties, but do not test behaviour under memory pressure, for example, whether `resetRegions` followed by reallocation correctly reuses pages, or whether the free list grows as expected when many regions are created and destroyed in a loop.

Size module accuracy. The `size_leq_regionUsage` test checks an inequality but does not verify that the computed size is exact. More precise tests could allocate a single value into an empty region and compare the computed size against the observed memory usage increase, accounting for region overhead. Beyond testing, the MLKit's unboxing scheme is complex, with representation decisions that depend on the number and types of constructors in a datatype, and it may be difficult for programmer to determine the exact size of a value without detailed knowledge of these decisions. Furthermore, as discussed in Section 4.4, sharing and cycles in data structures can cause the `Size` module to over-count, since each reference to a shared subvalue is counted separately.

6 Case study (chat service)

This chapter presents a case study that exercises the compiler extensions and modules from Chapters 3 and 4. The case study is a simulated chat service that reads packets from a file, reassembles them into complete messages, and processes them in a long-running loop, all without the garbage collector. We discuss how explicit region management enables adaptive, controlled memory reclamation and compare our approach to a reference project [24].

6.1 Reference Project and Motivation

Kanne and Geisshirt [24] implemented a modular protocol stack in SML on top of a unikernel framework, using region-based memory management and manual garbage collection. Their library implements Ethernet, ARP, IPv4, and TCP, and uses the double-copy method to reclaim dead memory in the protocol state at a fixed interval. Every nth iteration of the main loop, the entire protocol context is copied to fresh regions, the old regions are reset, and the copy is copied back. The frequency n is a tuning parameter that trades memory consumption against runtime overhead.

Their work identified a key limitation that we address in this case study, namely that the fixed-interval GC is blind to memory pressure. The double copy runs every nth iteration regardless of whether waste has accumulated. Kanne and Geisshirt note that "an alternative strategy, would be to measure the memory consumption and perform the double copy method if the memory meets a certain threshold" [24], but do not implement such a mechanism.

Additionally, our case study reads input from files rather than from a network device, which raises the question of which I/O library to use. The MLKit's `TextIO` structure has two properties that make it unsuitable for long-running loops without GC. First, its `openIn` and `openOut` functions use the `IO.Io` exception constructor. As explained in Section 11.2 of the MLKit manual [23], the MLKit's region inference takes a conservative approach to exceptions. A constructed exception value is placed in a region that is live at least as long as the exception constructor is in scope. Since `IO.Io` is declared at the top level of the `IO` structure, its scope is the entire program, so every application of the constructor allocates into a global region that is never reclaimed [23]. Second, `TextIO`'s input stream buffering does not interact well with the region resetting. The manual notes that constant space operation requires lower level I/O primitives, such as `Posix.IO.readVec`, that understand storage modes [23]. In a long-running loop without GC, these global allocations would accumulate indefinitely. We avoid these problems by implementing a minimal I/O module, `SimpleIO`, using direct C primitives, as described in Section 6.2.

6.2 Architecture

Our case study consists of four modules:

- `SimpleIO` provides minimal file I/O using direct C primitives, bypassing the standard basis library's `TextIO`. As discussed in Section 6.1, `TextIO`'s exception handling and buffering cause allocations into global regions that would accumulate in a long-running loop. `SimpleIO` avoids both issues. It uses `prim` calls that allocate into the caller's local regions, and it does not use exceptions in memory-problematic ways.

- `TcpState` implements a packet reassembler. Each connection maintains an ordered prefix of received packets and a bag for out-of-order packets. When a packet arrives, it is inserted into the bag, any packets that are now contiguous with the prefix are moved from the bag to the prefix. When the service extracts a completed message, the prefix is concatenated into a single string.

- `Service` maintains per-connection message buffers. Completed messages from the reassembler are appended to a connection's buffer until a double newline is found, at which point the message is printed and the connection is removed.

- `Driver` is a functor parameterized over `TcpState` and `Service`:

```
functor Driver
    (structure TcpState: TCP_STATE
     structure Service: SERVICE) :> DRIVER
```

  This separates the main loop and GC logic from both the reassembler and the service, making it easy to swap in a different service or TCP implementation, without changing the others.

A key architectural decision is the separation of `TcpState` and `Service` into independent state values, passed as a tuple `(state, ss)` through the main loop. This decision allows us to GC each independently and at different rates, as we describe in the next section.

6.3 Manual Garbage Collection

The main loop, in `Driver.loop`, applies the double-copy pattern to both state components, but with different strategies for each.

Assembler state: GC on every completed message

When a message is extracted from the reassembler, the driver immediately performs a double copy on the assembler state:

```
(* Double copy GC on the assembler state *)
val state3 = copyState state2
val _      = forceResetting state2
val _      = forceResetting state
val state4 =
  case connOpt of
    NONE => state3
  | SOME conn => closeConn (conn, state3)
val state5 = copyState state4
```

The driver uses `forceResetting` rather than `resetRegions` because the old state values, `state` and `state2`, are still in scope as arguments to the `loop`, so AtInf would refuse to reset them. Since the live data has already been copied to `state3`, the programmer knows the old regions contain only dead data.

The second copy, from `state4` to `state5`, ensures that the live data resides in the same regions that will be passed to the next iteration of the loop. Between the two copies, the driver also closes any completed connection, removing its assembler entry so that it is not copied going forward.

The message string itself is copied to a fresh region before being passed to the service.

```
val msg' = msg ^ ""
```

The concatenation with an empty string forces the runtime to allocate a new string in the service's regions rather than sharing a pointer into the assembler's regions, which are about to be reset.

Service state: adaptive GC

The service state is not garbage collected on every message. Instead, the `Service.timeToGC` function determines whether waste has accumulated to the point where a double copy is worthwhile:

```
val ss'' =
  if Service.timeToGC ss' then
    let val temp = Service.copySs ss'
        val _ = resetRegions ss'
    in Service.copySs temp
    end
  else ss'
```

The implementation of `timeToGC` is where the `Region` and `Size` modules come together:

```
fun timeToGC `[r1 r2 r3]
    (ss : (conn * string`r1)`r3 list`r2) : bool =
  let
    val total = Region.memoryUsageOfRegion `r1 ()
              + Region.memoryUsageOfRegion `r2 ()
    val live = Size.size
      (Size.list
        (Size.tup2 Size.int Size.string)) ss
  in live < total div 2
  end
```

The function's type signature uses explicit region parameters, `` `[r1 r2 r3] ``, to name the three regions that the service state occupies, `r1` for strings, `r2` for list cons cells, and `r3` for pairs. It queries the memory usage of `r1` and `r2` using `Region.memoryUsageOfRegion`, computes the live data size using the `Size` combinator, and triggers GC when the live data is less than half the total region usage.

This is the adaptive, threshold based GC strategy that Kanne and Geisshirt suggested as future work [24]. It has two advantages over their fixed-interval approach. It avoids unnecessary double copies when the state is small, and it responds promptly when a burst of completed messages leaves behind significant waste.

The `copySs` function

The service's copy function uses the same string concatenation trick as the driver:

```
fun copySs [] = []
  | copySs ((c, buf)::rest) =
      (c, buf ^ "") :: copySs rest
```

Each buffer string is concatenated with the empty string, forcing allocation into a fresh region. This makes `copySs` exomorphic, which is required for the double-copy pattern to work, if the copy shared pointers into the old regions, resetting those regions would invalidate the copied data.

6.4 Attack Surface of `TcpState`

The `TcpState` module maintains a per-connection state that grows as packets arrive. From a security perspective, an adversary could attempt to exhaust the service's memory by exploiting the assembler's state:

- **Many connections.** Opening many connections without completing any message causes the assembler's connection list to grow. Even if packets arrive in order, so the prefix advances, each connection entry persists in the state indefinitely. If early packets are withheld, out-of-order packets additionally accumulate in each connection's sorted bag.

- **Slow delivery.** Sending packets slowly for a single connection causes the prefix to grow over time as contiguous data accumulates. As long as the message eventually completes, the manual GC will reclaim the waste. However, if delivery stalls indefinitely, the partial state is never reclaimed.

- **No early packets.** Sending packets out of order fills the bag, which is maintained as a sorted list. If many packets arrive before the missing early packets, the bag grows without bound and insertions become expensive.

Our manual GC strategy is effective when messages complete. The assembler state is double-copied on every completed message, so waste from completed connections is reclaimed promptly. The `closeConn` function removes completed connections from the state before the second copy, ensuring their data is not held going forward.

All three attack vectors exploit the fact that incomplete connections are never reclaimed. An adversary who opens many connections and send partial or out-of-order data without ever completing a message can cause unbounded growth in the assembler state. A production system would need connection timeouts or per connection memory limits to handle this, neither of which is implemented in this case study.

6.5 Summary of Changes from the Reference Project

Table 3 summarises the differences between our case study and the reference project.

Aspect	**Reference project**	**This work**
GC trigger	Fixed interval (`resetCount`)	Every message (assembler); adaptive threshold (service)
GC decision	Iteration counter	`Region.memoryUsageOfRegion` + `Size` module
I/O	Network device (`Netif`)	`SimpleIO` (C primitives, avoids global region leaks)
Service modularity	Service function baked into transport handler	`Driver` functor parameterised over `Service`
Reset primitive	`resetRegions`	`forceResetting` (assembler); `resetRegions` (service)
Network	Full protocol stack (Ethernet, IPv4, TCP)	File-based simulation

Table 3: Comparison of the reference project and the case study.

7 Evaluation

This chapter evaluates the contributions presented in Chapters 3 and 4 through a series of incremental improvements to the case study from Chapter 6. We use MLKit's region profiler to measure memory usage over time, identifying the dominant source of waste at each step and applying the appropriate technique to address it. All measurements use the same input file, `big_test_packets.txt`, and are compiled with `-reml -no_gc`.

The input file, `big_test_packets.txt`, contains 2473 packets, roughly 146 KB, spread across 200 connections, all of which complete. Each connection consists of up to 20 packets that arrive out of order, and connection ID's are interleaved.

7.1 Memory Usage: Incremental improvements

We present the memory profile of the case study at six stages, each building on the previous. The stages correspond to the design decision described in Chapter 6, and we worked by identifying the dominant memory consumer at each step, applying a fix, and re-profiling.

Baseline: no GC, no manual GC

The baseline version uses the simplest possible loop. Read a packet, insert it into the assembler, extract any constructed prefix, and pass it to the service. No memory is reclaimed.

```
1  fun loop is (state, ss) =
2    case read is of
3      NONE => raise EOF
4    | SOME (msgID, packet) =>
5      case insert (msgID, packet, state) of
6        NONE => loop is (state, ss)
7      | SOME state1 =>
8        case extract state1 of
9          NONE => loop is (state1, ss)
10       | SOME (msgID, msg, state2) =>
11         let val (_, ss') =
12               Service.service (ss, msgID, msg)
13         in loop is (state2, ss')
14         end
```

Figure 8 shows the resulting memory profile. Memory grows linearly in all visible regions throughout the program's execution, reaching a peak of approximately 34 MiB. Every iteration allocates new data in the assembler and service regions, and nothing is ever freed.

Manual GC on the assembler state

We add a double-copy GC on the assembler state after each prefix extraction. The key addition is:

```
1  val msg' = msg ^ ""
2  val (connOpt, ss') =
3    Service.service (ss, msgID, msg')
4
5  (* Double copy GC on the assembler state *)
6  val state3 = copyState state2
7  val _      = forceResetting state2
8  val _      = forceResetting state
9  val state4 =
10   case connOpt of
11     NONE => state3
12   | SOME conn => closeConn (conn, state3)
13 val state5 = copyState state4
```

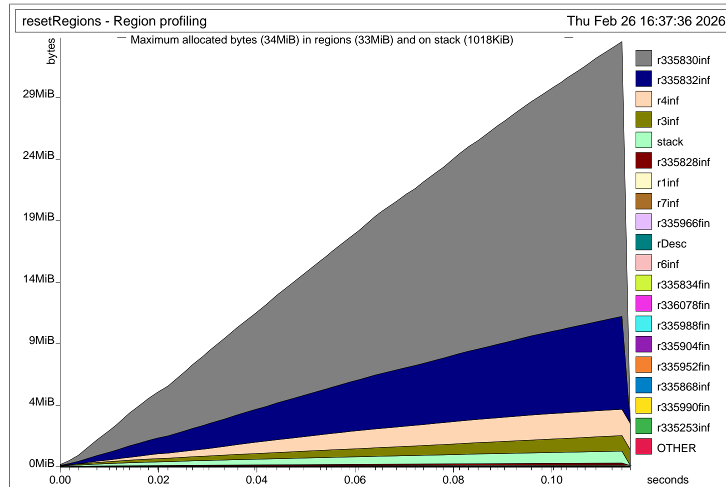

Figure 8: Memory profile without GC and without manual memory management. Memory grows linearly to 34 MiB

The message string is copied with `msg ^ ""` to decouple it from the assembler's regions before they are reset. The function `forceResetting` is used because the old state values are still in scope as loop arguments. Figure 9 shows the result. Peak allocation drops from 34 MiB to approximately 12 MiB. The shape of the graph shows a very slight decreasing curve but is still essentially linear growth. The remaining growth comes from the service state and other regions that are not yet reclaimed.

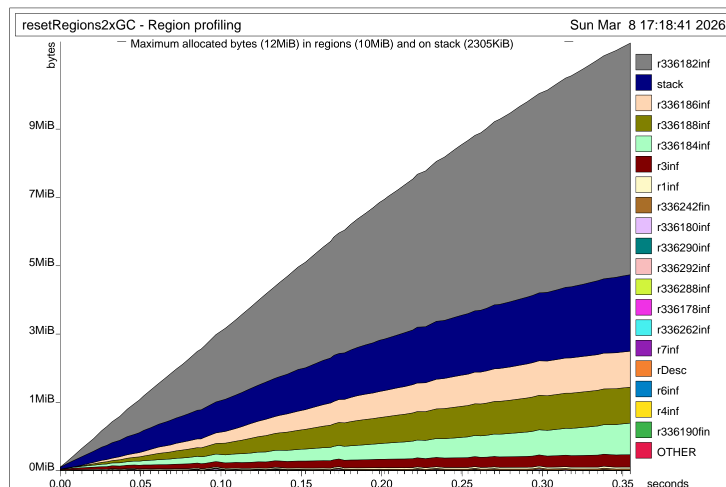

Figure 9: Memory profile with manual GC on the assembler state. Peak allocation drops to 12 MiB, but growth is still essentially linear.

Manual GC on the service state

We add a double-copy GC on the service state, initially triggered on every iteration:

```
1 (* Double copy GC on the service state *)
2 val temp = Service.copySs ss'
3 val _    = resetRegions ss'
4 val ss'' = Service.copySs temp
```

Figure 10 shows the result. The peak allocation drops to approximately 2709 KiB, of which 2244 KiB is the runtime stack. The region usage itself is increasing but with a slightly decreasing curve, and then remains roughly linear from 0.1 seconds onward. The stack is now the dominant band, which suggests that making the loop tail-recursive would be the next improvement. The region profiler also shows that region `r3`, which is the global regions for strings, continues to grow, indicating allocations from the basis library's `TextIO` module.

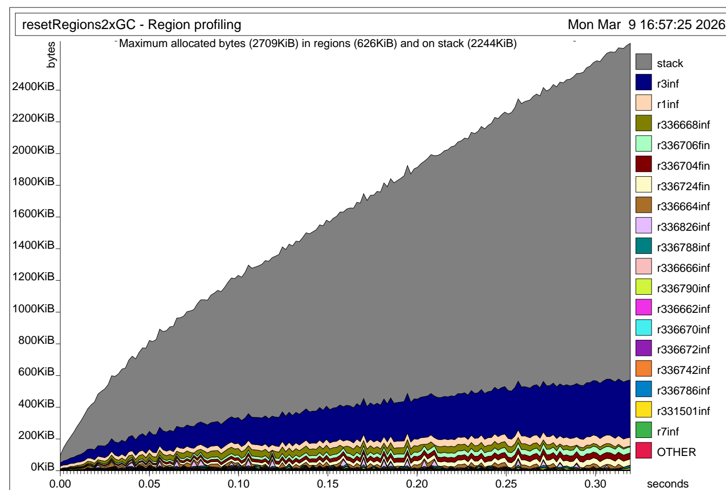

Figure 10: Memory profile with manual GC on both assembler and service state. Stack usage (2244 KiB) dominates, and `r3` still grows.

Tail-recursive loop

Figure 11 shows the effect of making the main loop tail-recursive. The peak allocation drops to 516 KiB. The shape is similar to the previous graph. Memory still grows but with a slightly decreasing rate. The global region `r3` is still visible and growing, pointing to `TextIO` as the remaining source of unbounded growth.

SimpleIO

We replace `TextIO` with out `SimpleIO` module, which avoids the two sources of global region allocation identified in Section 6.1. The critical difference is in stream creation:

```
1 fun openIn (f: string) : instream =
2   {ic=prim("openInStream",
3            (getCtx(), f, CannotOpen)),
4    name=f}
5   handle exn => raise Fail "openIn"
```

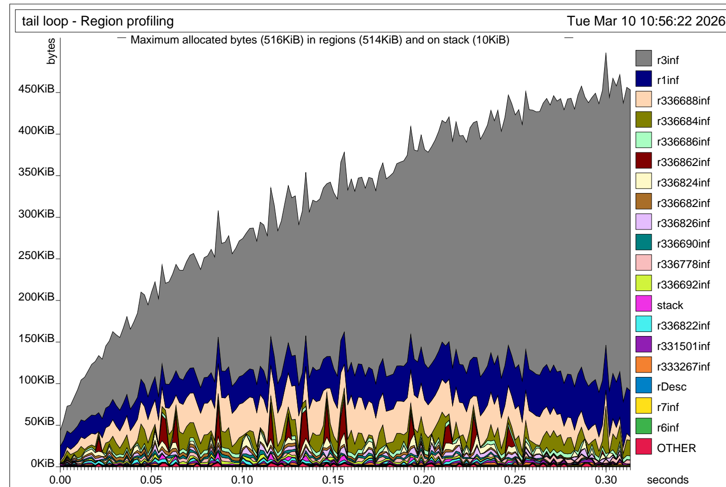

Figure 11: Memory profile with tail-recursive loop. Peak allocation drops to 516 KiB, but with `r3` still growing.

Figure 12 shows the result. The peak allocation drops to 150 KiB and the graph now shows a sawtooth pattern, characteristic of bounded memory usage where allocations are periodically reclaimed. The global string region `r3` no longer grows, confirming that the `TextIO` allocations were the cause of the remaining unbounded growth. At this point, the service state is still garbage collected on every iteration.

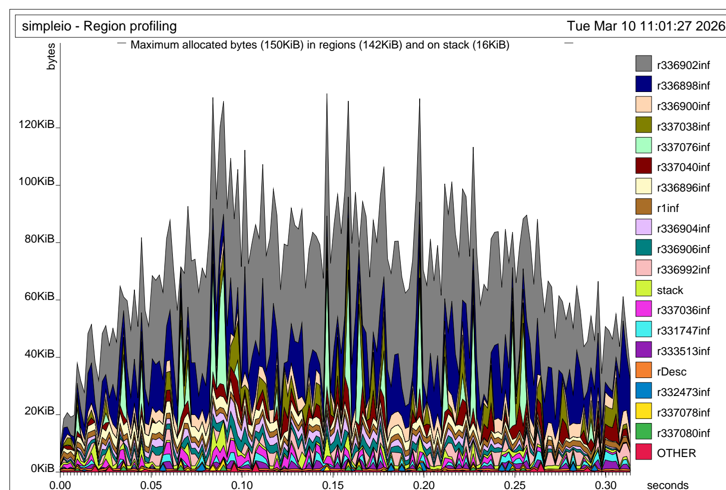

Figure 12: Memory profile with `SimpleIO`. Peak allocation is 150 KiB, and the sawtooth pattern indicates bounded memory usage.

Adaptive GC with the Region module

Finally, we replace the per iteration service GC with the adaptive `timeToGC` function, which uses the `Region` and `Size` modules to decide when the double copy is worthwhile:

```
val ss'' =
  if Service.timeToGC ss' then
    let val temp = Service.copySs ss'
        val _ = resetRegions ss'
    in Service.copySs temp
    end
  else ss'
```

The threshold is controlled by `timeToGC`, as described in Section 6.3.

Figure 13 uses a threshold of 2×, meaning the service state is GC'd when the total region usage exceeds twice the live data size. The peak allocation is 185 KiB, slightly higher than the per iteration version, because waste is allowed to accumulate before being reclaimed. The graph shows a sawtooth pattern and memory remains bounded.

Figure 14 relaxes the threshold to 10×, so GC occurs when the total region usage exceeds ten times the live data. The peak allocation rises to 533 KiB, The sawtooth pattern is still visible and memory remains bounded, but the teeth are wider and fewer, since it is larger collections rather than many small ones.

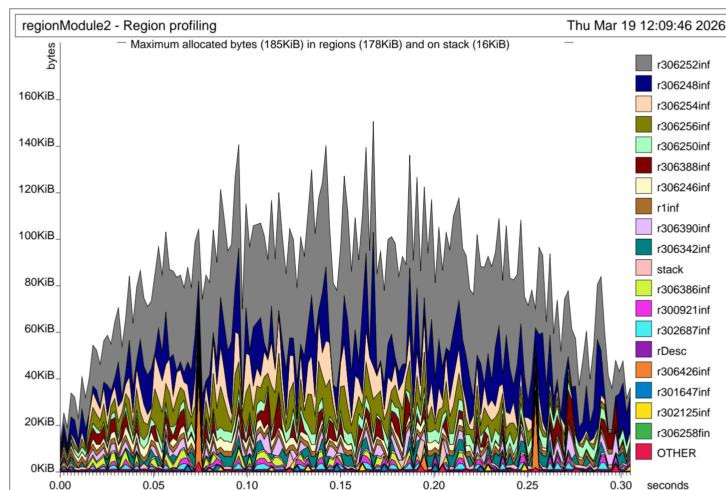

Figure 13: Adaptive GC with 2× threshold. Peak allocation is 185 KiB, and memory is bounded.

Table 4 summarises the progression.

7.2 Runtime

All runtimes were measured using the shell `time` command on the compiled executable. Table 5 summarises the results.

The baseline without manual GC completes in approximately 104 ms. The version with manual GC on every iteration completes in approximately 198 ms, roughly 1.9× slower. This overhead we suspect comes from the doubly-copy operations, which traverse and copy all live state on every prefix extraction for the assembler and on every iteration for the service.

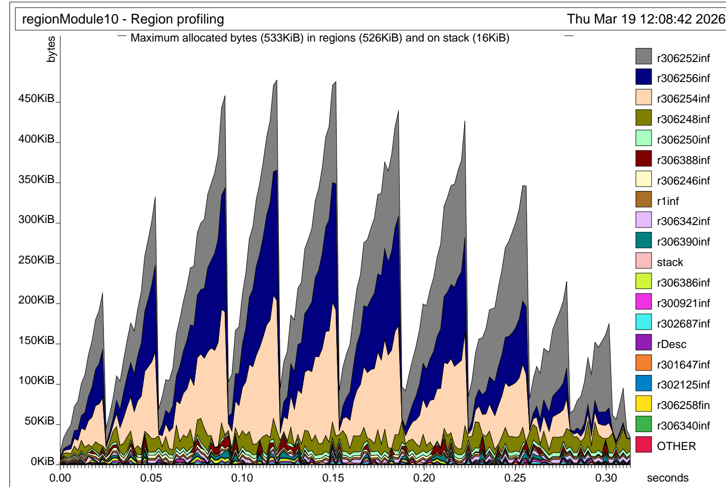

Figure 14: Adaptive GC with 10× threshold. Peak allocation is 533 KiB, and there are fewer, larger collections.

Stage	Peak alloc.	Pattern
No GC, no manual GC	34 MiB	Linear growth
+ assembler GC	12 MiB	Near-linear growth
+ service GC (every iteration)	2709 KiB	Stack-dominated
+ tail-recursive loop	516 KiB	Near-constant, `r3` grows
+ SimpleIO	150 KiB	Sawtooth, bounded
+ Region module (2×)	185 KiB	Sawtooth, bounded
+ Region module (10×)	533 KiB	Sawtooth, bounded

Table 4: Peak memory allocation at each stage of the case study.

The adaptive thresholds tell a different story. The 2× threshold completes in approximately 98 ms and the 10× threshold in approximately 96 ms, both faster than the baseline without any GC. The per-iteration configuration copies state on every iteration regardless of whether the regions contain significant waste, whereas the adaptive configurations only copy when the waste exceeds the threshold, performing far fewer copies overall. The baseline is slower than the adaptive configurations despite performing no collection, because its unbounded memory growth causes repeated page allocations from the operating system, as reflected in its higher system time (42 ms versus 34 ms for the adaptive configurations).

7.3 Comparison with Built-in Garbage Collection

To contextualise the manual approach, we compiled the same case study with MLKit's built-in reference tracing garbage collector and no manual memory management.

Figure 15 shows the result. The memory profile exhibits a sawtooth pattern and appears bounded, with a peak allocation of 234 KiB. The runtime is approximately 86 ms.
Table 5 compares the built-in GC with selected manual configurations.
The built-in GC achieves bounded memory with no programmer effort and is the fastest configuration at 86 ms. The adaptive manual configuration are close behind at 96-98 ms, achieving lower peak memory (185 KiB vs. 234 KiB for the 2× threshold) with only a modest runtime penalty.

The per-iteration configuration achieves the lowest peak memory (150 KiB) but at roughly 2.3×

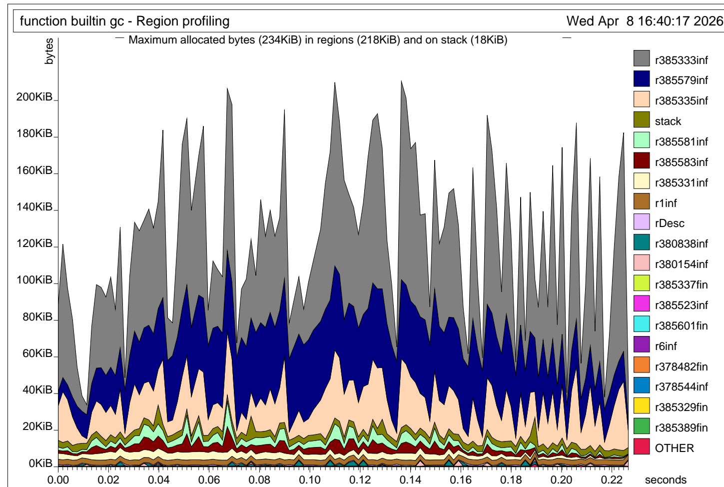

Figure 15: Memory profile with the built-in garbage collector, and no manual memory management. Peak allocation is 234 KiB, showing a sawtooth pattern.

Configuration	Peak alloc.	Wall time	Sys time
No GC, no manual GC	34 MiB	104 ms	42 ms
Manual GC, every iteration	150 KiB	198 ms	100 ms
Manual GC, Region 2×	185 KiB	98 ms	36 ms
Manual GC, Region 10×	533 KiB	96 ms	34 ms
Built-in GC, no manual GC	234 KiB	86 ms	8 ms

Table 5: Comparison of manual GC configurations with the built-in garbage collector. All manual configurations include assembler GC on every prefix extraction, SimpleIO, and a tail-recursive loop. Runtimes are wall-clock and system times measured without profiling.

the runtime of the built-in GC, because it copies state unconditionally on every iteration. The adaptive thresholds avoid the cost by copying only when the measured waste exceeds the threshold, which on this workload results in faster execution than even the no-GC baseline.

The manual approach offers two advantages. First, it gives the programmer control over exactly what is traversed, and when. Second, it avoids whole-heap traversals, which in the built-in GC contribute both to longer pause times and to overall runtime, since the collector must trace all reachable data regardless of how much waste exists. A selective approach that only traverses the state components known to accumulate waste should scale better as the total live set grows.

On this workload, however, the built-in GC is faster than all manual configuration. We believe the main bottleneck is the double-copy, which traverses the live state twice on each GC cycle. Reducing this cost would be necessary to achieve a runtime advantage. One possible solution is double buffering, where we maintain two sets of regions and alternate between them on each GC cycle. The current approach copies live state to temporary regions, resets the originals, and copies back. With double buffering, the programmer would copy the state once into the alternative regions and then reset the previous regions, halving the number of traversals.

7.4 Discussion

What the modules enable

The `Region` module and `Size` combinator together enable a pattern that was not previously possible in MLKit without GC, namely data-driven garbage collection decisions at the SML level. The programmer can inspect region usage at runtime, compare it to the computed size of live data, and decide whether the cost of a double copy is justified.

The adaptive approach also decouples the GC frequency from the program's control flow. In the reference project, the GC interval is a static parameter, that must be tuned for each workload. With the `Region` module, the GC frequency adapts automatically to the actual memory behaviour, and the threshold parameter controls the trade-off between memory usage and GC overhead rather than being tied to iteration counts.

Region introspection is also available without profiling builds. Normally, understanding region sizes requires compiling with `-prof` and analysing the profiler output offline. The `Region` module exposes the same information at runtime in production builds, which enables not only adaptive GC but also runtime monitoring, logging, and debugging of memory behaviour in deployed programs.

The incremental development story in Section 7.1 also illustrates a broader methodology. The region profiler identifies the dominant source of waste, the programmer applies the appropriate technique, and re-profiles to verify. The `Region` module complements this workflow by allowing the program itself to perform the same kind of reasoning at runtime, comparing region usage to live data, rather than relying solely on offline profiling.

However, the `Region` module can only inspect regions that the programmer has identified and made explicit in advance. The programmer must already suspect which regions might accumulate waste and ensure they appear as explicit region parameters, so that they can be passed to `memoryUsageOfRegion`. Problems that do not manifest in explicit regions are invisible to the module. For example, the `TextIO` functions leak into global region `r3` and the stack growth from a non-tail-recursive loop were both identified through the region profiler, not through the `Region` module, because neither involves a programmer declared explicit region. The module is therefore most useful once the programmer has already identified the relevant regions, either through profiling or through understanding of the MLKit's value representation.

Finally, explicit region annotations enable independent GC of separate state components. Because the assembler and service regions are named explicitly, the programmer can target one without traversing the other. This selective approach scales better than whole program GC when only a subset of the program accumulates waste.

What requires manual effort

Despite the module support, significant manual effort remains. The most labour intensive task is writing exomorphic copy functions. The programmer must implement `copyState` and `copySs` by hand, reconstructing each data structure element by element. For strings, the `s ^ ""` trick forces allocation into a new region, but this idiom is specific to strings. Other boxed types require their own type-specific copy operations. Getting a copy function wrong, for example by forgetting co copy a nested field, results in sharing into old regions that will be reset, leading to use-after-free errors that are difficult to diagnose.

Writing the explicit region annotations also requires care. The `timeToGC` function must name the regions of its argument type explicitly, so that it can query them individually. This requires understanding the MLKit's value representation and how types map to regions. For example, a value of type `(int * string) list` occupies three regions, one for strings, one for list cons cells, and one the element pairs. The programmer must know this decomposition to write the correct type annotations, and must also remember to include all relevant regions in the computation. In the case study, `timeToGC` declares three explicit region parameters, but only sums the usage of two, omitting the pair region from the threshold computation. This causes the function to underestimate live data,

and thus overestimate total waste.

The programmer must also choose the appropriate reset primitive. Both `resetRegions` and `forceResetting` pass through the AtInf safety analysis, which checks whether any live variables reference the regions being reset and prints warnings if so. The difference is in what happens next. The function `resetRegions` respects the analysis and refuses to reset regions with live references. `forceResetting` prints the same warnings but resets regardless. The programmer must therefore understand whether live references identified by AtInf are genuine. In the assembler GC, the old state value are still in scope as arguments to the tail recursive `loop`, so AtInf identifies them as live. However, the programmer knows that the data is not used again, making `forceResetting` safe despite the warnings. In the service GC, `resetRegions` suffices by using a local temporary to store the updated state.

Beyond the extensions themselves, the programmer must also be aware of which basis library modules allocate into global regions and avoid them. We replaced `TextIO` with `SimpleIO` for this reason, as we discussed in Section 6.1. Understanding why `TextIO` leaks requires familiarity with the exception handling mechanism and why it allocates into global regions, which is knowledge that is not obvious from the module signatures alone.

Finally, the incremental development process itself relies on interpreting region profiler output to identify which regions grow and why. This also requires correlating region identifiers with tracing regions in the intermediate code to understand where allocation originates.

7.5 Threats to Validity

- The case study reads packets from files, not from a network device. Real network I/O introduces latency, jitter, and asynchronous arrival patterns that may affect the relative cost of the double-copy operations.

- All measurements use a single input file containing 2473 packets across 200 connections. The adaptive threshold may behave differently under other workloads. A more thorough evaluation would vary the input file size, the size of individual packets, the number of packets per connection, and the number of concurrent connections, to understand how each parameter affect memory usage and runtime independently.

- As discussed in Section 4.4, the `Size` module does not account for shared subvalues and assumes the non-GC value representation. This means `timeToGC` may overestimate the live data size, causing GC to trigger less often than intended.

- Runtimes were measured using the shell `time` command, which report wall-clock, user, and system time for the entire process. This includes startup and shutdown costs that are shared across all configurations but may represent a non-trivial fraction of the total, especially for short-running benchmarks like ours. A more precise measurement would use an in-process timer around the main computation, excluding process initialization. We report a single run rather than a summary over multiple runs, so the measurements are subject to OS scheduling, cache effects, and other system-level variance. Multiple runs would reduce this variance, but would not eliminate it.

8 Conclusion

This thesis has presented three contributions that extend the MLKit compiler and its ReML language with tools for explicit, data-driven region management.

The first contribution is a pair of compiler extensions that give programmers finer-grained control over region lifecycle. We extended the `resetRegions` and `forceResetting` primitives to accept explicit region variables as additional arguments, and we extended the `prim` mechanism to pass explicit region pointers to C functions. These changes touch the `LambdaExp` intermediate language, the spreading phase, the storage mode analysis, and the drop-regions phase, but preserve the structure of the region type system and its soundness properties. The extensions allows programmers to target specific regions for resetting rather than being forced to reset all regions in a value's type, and to reset regions independently of any particular value using unit as the argument expression. This shifts the programming model. Programmers no longer needs to construct values whose types happen to mention the right regions, but can instead name the regions directly. The storage mode analysis applies its safety rules to explicit regions just as it does to inferred ones, so `resetRegions` remains safe while `forceResetting` remains programmers option for cases where the analysis is too conservative.

The second contribution is two SML modules that build on the compiler extensions. The `Region` module provides typed access to per-region metadata and a direct reset operation, all available at runtime without requiring profiling compilation. The `Size` module is a combinator library for computing the memory footprint of an SML value as it is laid out in regions. Together, these modules enable a pattern that was not previously possible in the MLKit. It is now possible to compare live data size against total region usage from within the running program, and using the result to make informed decisions about when to reclaim memory. The key insight behind the `Size` module is that the abstract type `'a sz`, revealed internally as `'a -> int`, allows size descriptors to be composed from primitives and combinators in the same way that parsers or pretty-printers are composed in other combinator libraries.

The third contribution is a case study that exercises the extensions and modules in a realistic setting. We applied them to a chat service that reads packets from a file, reassembles them into messages, and processes them in a long-running loop with the built-in garbage collector disabled. Through a series of incremental improvements (adding manual garbage collection on the assembler state, on the service state, making the loop tail-recursive, replacing `TextIO` with a minimal I/O module, and finally introducing adaptive threshold-based collection using the `Region` and `Size` module) we reduced peak memory usage from 34 MiB to 185 KiB while maintaining runtime comparable to the built-in garbage collector (98 ms versus 86 ms). The adaptive approach implements the threshold-based collection strategy that Kanne and Geisshirt [24] identified as future work in their unikernel protocol stack, and demonstrates that the `Region` module makes such strategies practical.

The case study also revealed the significant manual effort that remains. Writing exomorphic copy functions requires reconstructing each data structure element by element, with type specific ways to copy, such as `s ^ ""` for strings. Annotating function types with explicit region parameters requires understanding the MLKit's value representation and how types decompose into regions. Choosing between `resetRegions` and `forceResetting` requires interpreting the storage mode analysis diagnostic and reasoning about whether the identified conflicts are genuine. And avoiding global region leaks from the standard basis library requires familiarity with the MLKit's exception handling mechanism. These tasks demand deep knowledge of the compiler's behaviour, and reducing this burden is the central motivation for the future work we outline next.

8.1 Future work

Several directions could reduce the manual effort identified above and extend the capabilities of the modules.

Region discovery. Both the compiler extensions and the `Region` module require programmers to name regions explicitly. A function that, given a value, returns a list of foreign pointers to the regions

appearing in its type would allow the `Region` module's operations to be used even when not all regions are explicit. This would require a new compiler primitive that inspects the region-annotated type of its argument and produces region pointers at runtime. A challenge is ensuring that the returned pointers remain valid for as long as the programmer uses them. If a region goes out of scope, while the programmer still holds a pointer to it, the pointer becomes dangling. A solution would need to either prevent the regions from being deallocated while pointers are outstanding, or constrain the API so that pointers cannot escape the scope in which the regions are live.

Polymorphic deep copy. The most labour-intensive aspect of the manual region management is writing exomorphic copy functions. A polymorphic function `exomorphCopy : 'a -> 'a` that deep-copies a value into fresh regions would eliminate this burden entirely. A possible solution is a compiler-generated approach. The compiler already knows the region-annotated type of every value, so at each call site of `exomorphCopy` it could generate a specialised copy function that allocates fresh regions matching the type structure, traverses the value, and copies each boxed component into corresponding fresh region. Another option would be a C-level function that receives a value together with a runtime type descriptor, traverses the value following the descriptor, and copies each component into freshly allocated regions. This would require making type layout information available at runtime in a form that the runtime system can interpret, which is not currently the case.

Copy combinator. As a more incremental alternative a fully polymorphic deep copy, the combinator pattern used in the `Size` module could be adapted to build copy functions. An abstract type `'a cp` would represent a function that, given a value of type `'a`, produces a deep copy in fresh regions. Programmers would compose copy descriptors from primitives and combinators in the same way as size descriptors, and apply them via a `copy: 'a cp -> 'a -> 'a` function. This approach does not require runtime layout information but still requires programmers to build a descriptor that matches the shape of each data type. Such a copy combinator would have many of the same issues and limitations as the `Size` module.

Double buffering. The double-copy pattern traverses the live state twice per garbage collection cycle. Double buffering, where we maintain two sets of regions and alternate between them, would halve the number of traversals, reducing the runtime overhead of manual GC.

Finite region support. The synthetic `put` effects added during spreading cause explicit region parameters to be promoted to infinite regions, even when the region would otherwise be finite. Introducing a new atomic effect, distinct from `put`, that indicates a region accessed without implying a write would allow the `Region` module to inspect finite regions and would avoid inflating their multiplicity.

OptLambda equality fix. The `eq_prim` function in `OptLambda.sml` does not compare the `regvars` field, which can cause the optimiser to collapse conditionals whose branches target different explicit regions. Extending the comparison to include `regvars` is a straightforward fix that would remove the current need to disable the Lambda optimiser in programs that branch on which region to reset or query.

Sharing-aware size computation. As discussed in Section 4.4, the `Size` module counts shared subvalues once per reference rather than once per object, which can cause it to over-count when data structures share region-allocated components. A sharing-aware variant would need to track visited addresses during traversal, skipping values already seen. Since Standard ML does not provide pointer identity checks, this would require a small C helper to obtain the address of a value, together with a change to the size descriptor type to thread a visited set through the computation.

Garbage collector-aware size computation. The `Size` module assumes the non-GC value representation throughout. When garbage collection is enabled, the runtime adds header words to certain boxed values, and the default integer type narrows from `Int64.int` to `Int63.int`. The module could detect the active representation at runtime, by inspecting `Int.maxInt`, and adjust its overhead calculations accordingly, making it usable in program compiled with the built-in garbage collector.

References

[1] Douglas T. Ross. "The AED free storage package". In: *Communications of the ACM* 10 (8 Aug. 1967), pp. 481–492. ISSN: 15577317. DOI: 10.1145/363534.363546.

[2] David R. Hanson. "Fast allocation and deallocation of memory based on object lifetimes". In: *Software: Practice and Experience* 20 (1 Jan. 1990), pp. 5–12. ISSN: 1097-024X. DOI: 10.1002/SPE.4380200104. URL: https://onlinelibrary.wiley.com/doi/full/10.1002/spe.4380200104%20https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380200104%20https://onlinelibrary.wiley.com/doi/10.1002/spe.4380200104.

[3] Paul R. Wilson. "Uniprocessor garbage collection techniques". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 637 LNCS (1992), pp. 1–42. ISSN: 1611-3349. DOI: 10.1007/BFB0017182. URL: https://link.springer.com/chapter/10.1007/BFb0017182.

[4] Mads Tofte and Jean-Pierre Talpin. "Implementation of typed call-by value λ-calculus using a stack of regions". In: *Conference Record of the Annual ACM Symposium on Principles of Programming Languages* (1994), pp. 188–201. ISSN: 07308566. DOI: 10.1145/174675.177855.

[5] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. "From region inference to von Neumann machines via region representation inference". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. St. Petersburg Beach, Florida, USA: Association for Computing Machinery, 1996, pp. 171–183. ISBN: 0897917693. DOI: 10.1145/237721.237771. URL: https://doi.org/10.1145/237721.237771.

[6] Mads Tofte and Jean Pierre Talpin. "Region-Based Memory Management". In: *Information and Computation* 132 (2 Feb. 1997), pp. 109–176. ISSN: 0890-5401. DOI: 10.1006/INCO.1996.2613.

[7] Mads Tofte and Lars Birkedal. "A Region Inference Algorithm". In: *ACM Transactions on Programming Languages and Systems* 20 (4 1998), pp. 724–767. ISSN: 01640925. DOI: 10.1145/291891.291894.

[8] Martin Elsman. "Program Modules, Separate Compilation, and Intermodule Optimisation". PhD thesis. Department of Computer Science, University of Copenhagen, Jan. 1999.

[9] Martin Elsman and Niels Hallenberg. "Integrating region memory management and tag-free generational garbage collection". In: *JFP* 31 (2001), p. 2021. DOI: 10.1017/S0956796821000010. URL: https://doi.org/10.1017/S0956796821000010.

[10] Daniel C. Wang and Andrew W. Appel. "Type-preserving garbage collectors". In: *SIGPLAN Not.* 36.3 (Jan. 2001), pp. 166–178. ISSN: 0362-1340. DOI: 10.1145/373243.360218. URL: https://doi.org/10.1145/373243.360218.

[11] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. "Region-based memory management in cyclone". In: (May 2002), pp. 282–293. DOI: 10.1145/512529.512563.

[12] Niels Hallenberg, Martin Elsman, and Mads Tofte. "Combining region inference and garbage collection". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2002), pp. 141–152. DOI: 10.1145/512529.512547.

[13] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. "Cyclone: A Safe Dialect of C". In: *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*. ATEC '02. USA: USENIX Association, 2002, pp. 275–288. ISBN: 1880446006.

[14] David F. Bacon, Perry Cheng, and David Grove. "Garbage collection for embedded systems". In: *EMSOFT 2004 - Fourth ACM International Conference on Embedded Software* (2004), pp. 125–136. DOI: 10.1145/1017753.1017776.

[15] *The Standard ML Basis Library*. Cambridge University Press, 2004.

[16] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. "A retrospective on Region-based Memory Management". In: *Higher-Order and Symbolic Computation* 17 (3 Sept. 2004), pp. 245–265. ISSN: 13883690. DOI: 10.1023/B:LISP.0000029446.78563.A4.

[17] David F. Bacon. "Realtime garbage collection". In: *ACM Queue* 5.1 (2007), pp. 40–49. DOI: 10.1145/1217256.1217268. URL: https://doi.org/10.1145/1217256.1217268.

[18] Martin Elsman. "SMLtoJs: hosting a standard ML compiler in a web browser". In: *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients*. PLASTIC '11. Portland, Oregon, USA: Association for Computing Machinery, 2011, pp. 39–48. ISBN: 9781450311717. DOI: 10.1145/2093328.2093336. URL: https://doi.org/10.1145/2093328.2093336.

[19] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "RustBelt: securing the foundations of the Rust programming language". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158154. URL: https://doi.org/10.1145/3158154.

[20] Magnus Madsen. "The Principles of the Flix Programming Language". In: *Onward! 2022 - Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, co-located with SPLASH 2022* 1 (Nov. 2022), pp. 112–127. DOI: 10.1145/3563835.3567661. URL: https://doi.org/10.1145/3563835.3567661.

[21] Martin Elsman. "Explicit Effects and Effect Constraints in ReML". In: *Proceedings of the ACM on Programming Languages* 8 (Jan. 2024), pp. 2370–2394. ISSN: 24751421. DOI: 10.1145/3632921.

[22] Anton Lorenzen, Leo White, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. "Oxidizing OCaml with Modal Memory Management". In: *Proc. ACM Program. Lang.* 8.ICFP (Aug. 2024). DOI: 10.1145/3674642. URL: https://doi.org/10.1145/3674642.

[23] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. *Programming with Regions in the MLKit (Revised for Version 4.7.16)*. Tech. rep. Department of Computer Science, University of Copenhagen, Denmark, Dec. 2025.

[24] Axel Prütz Kanne and Svante Geisshirt. *Implementing a Modular Protocol Stack using Region-Based Memory Management*. Student project. Department of Computer Science, University of Copenhagen, 2026.

A Declaration of Using Generative AI Tools

☒ **I/we have used generative AI as an aid/tool** *(please tick)*

☐ **I/we have <u>NOT</u> used generative AI as an aid/tool** *(please tick)*

List which GAI tools you have used and include the link to the platform (if possible):

- Copilot, copilot.microsoft.com
- Claude, claude.ai

Describe how generative AI has been used in the exam paper:

1) *Purpose (what did you use the tool for?)*
    - GAI was used to generate code (Copilot)
    - GAI was used for research (Claude)
    - GAI was used for the report (Claude)
2) *Work phase (when in the process did you use GAI?)*
    - For the implementation it was used to generate repetitive code
    - For research it was used to find source material, and improve my understanding of concepts
    - For the report it was used for grammar and spelling mistakes
3) *What did you do with the output? (including any editing of or continued work on the output)*
    - For the generated code, I verified that the code did what I expected, and my implementation was tested, including any generated code
    - For research, I read the output and the source material it provided
    - For the report, I made it give me a list with all the mistakes, found it in my report and fixed it