Safe Region for Scala Native

YAWEN GUAN, Semester Project at LAMP, Supervised by Prof. Martin Odersky

Region-based memory management is known to be less space-demanding and more predictable than generic garbage collection techniques [Tofte et al. 2004], and is especially well-suited for use in performance-critical applications where fine-grained control over memory is required. However, it can be challenging to program with region-based memory management as most of the implemented approaches rely on the users to avoid accessing invalid memory. To address this challenge, this report proposes the design of safe regions. The memory safety of safe regions is guaranteed by *capture checking*, and operations are adapted to preserve the garbage-collected heap's integrity. Safe regions in implemented in Scala Native and its performance is tested on Hans Boehm's GCBench benchmark [Systems 2021]. The results indicate that programs utilizing safe regions is up to 37.6% faster compared to those without, and this improvement is slightly amplified with the addition of operations increasing the access frequency of some memory locations.

1 INTRODUCTION

1.1 Region-based Memory Management

Region-based memory management is a dynamic memory management discipline that is less space-demanding and more predictable than generic garbage collection techniques, such as generational garbage collection [Tofte et al. 2004]. In region-based memory management, all objects are allocated in *regions*. A *region*, also called a *zone*, *arena* or *memory context*, is a block of memory that is associated with a computation unit (typically a thread or a method). Regions are freed when the corresponding unit finishes its execution, and all objects in the same region are deallocated simultaneously [Salagnac et al. 2005].

The utilization of region-based memory management is a prevalent practice. For example, the Real-Time Specification for Java (RTSJ) [Bollella and Gosling 2000] adopts a region-based approach that relies on memory scopes that are automatically freed upon termination of their lifetime. This allows the turning off of garbage collection during the execution of critical tasks, thus ensuring real-time guarantees [Higuera-Toledano et al. 2012]. RC [Gay and Aiken 2001] is a compiler for C with regions that prevents unsafe region deletions by keeping a count of references to each region. These approaches define APIs which can be used to explicitly and manually handle the allocation and deallocation of objects within a program, but programming with these APIs is error-prone because care must be taken when objects are mapped to regions to avoid dangling references [Salagnac et al. 2005].

Another approach is to use *region inference* [Tofte et al. 2004] technique, in which the creation and deallocation of region as well as assignment of objects to particular regions are inserted by the compiler at compile-time. Region inference is used in some programming languages, such as Rust [Matsakis and Klock II 2014], which aims to provide memory safety guarantees while still allowing low-level control over memory.

1.2 Scala Native

Scala Native [Contributors 2017] is a ahead-of-time compiler and optimizing runtime for Scala. It compiles Scala code directly to machine code, bypassing the Java Virtual Machine (JVM) and its associated overhead. Scala Native supports the full Scala language including standard libraries. Additionally, it supports low-level primitives (including unsafe region allocation) and seamless

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

interoperation with C and C++ code, making it a suitable choice for systems programming and other performance-critical tasks.

1.2.1 Scala Native Compilation. As shown in [Shabalin 2020], the compilation of Scala Native can be divided into the following three steps:

1. Compile Scala Native to *Native Intermediate Representation* (NIR) relying on a modified Scala compiler that emits NIR instead of JVM bytecode.
2. Link NIR files, run Scala Native link-time optimization, and lower NIR to LLVM IR relying on the Scala Native toolchain.
3. Link LLVM IR files, run LLVM link-time optimization, and generate an optimized statically linked binary that includes all of the application code in addition to the implementation of the garbage collector.

1.2.2 Runtime Support for Garbage Collection. Scala Native provides runtime support for garbage collection, utilizing *conservative* garbage collectors (GC) which treat values as ambiguous references and conservatively classify them as a potential heap reference if they are within the valid memory range [Shabalin 2020]. It supports four distinct conservative garbage collectors: No GC, Boehm GC, Immix GC, and Commix GC.

1.2.3 Unsafe Manual Memory Management. Scala Native also supports unsafe manual memory management including region allocation, stack allocation, heap allocation using standard C memory allocation functions such as MALLOC and FREE [Contributors 2017].

1.3 Capabilities and Capture Checking

Capture checking is a research project in Scala, which involves tracking the free variables captured by values and representing them in types. Guiding the implementation of capture checking, $\mathrm{CC}_{<:\square}$ is a calculus where such captured variables are succinctly represented in types, allowing for the safe implementation of effects and effect polymorphism through scoped *capabilities* [Odersky et al. 2022]. $\mathrm{CC}_{<:}$ [Boruch-Gruszecki et al. 2021] forms the foundational framework for $\mathrm{CC}_{<:\square}$ and operates as a calculus devoid of the use of boxing.

[Odersky et al. 2022] elucidates the following core concepts of capture checking:

- A *capturing type* is of the form $\{c_1, c_2, ..., c_n\}\,T$ where T is a type and $\{c_1, c_2, ..., c_n\}$ is a *capture set* of capabilities.
- A capability is a parameter or local variable that has as type a capturing type with non-empty capture set. These capabilities, referred to as *tracked variables*, derive their authority from more sweeping capabilities, with the ultimate source of authority being the *universal capability* "$*$".

2 DESIGN OF SAFE REGION

The design of safe regions primarily concentrates on ensuring the memory safety of regions, as well as the allocation strategy of regions.

2.1 Key Properties for Ensuring Memory Safety

Memory safety is a property that guarantees that users do not have access to any memory locations that are deemed invalid, thereby preventing potential undefined behavior. In order to ensure memory safety, a region should not permit users to access memory that has either not been allocated or has been deallocated.

Syntax

$$
\begin{array}{llll}
S ::= \ldots & & \textbf{Shape Type} & \\
 & \text{Region} & & \text{region capability} \\
t ::= \ldots & & \textbf{Terms} & \\
 & \textbf{region}\, x \,\textbf{in}\, t & & \text{region block} \\
 & x.\,\textbf{new}[T]\ t & & \text{object creation in region} \\
v ::= \ldots & & \textbf{Values} & \\
 & x & & \text{variable}
\end{array}
$$

Type assignment $\boxed{\Gamma \vdash t : T}$

$$
\frac{\Gamma,\ x : \{*\}\ \text{Region} \vdash t : T \qquad \Gamma, x : \{*\}\ \text{Region} \nvdash \{x\} <: \mathrm{cv}(T)}{\Gamma \vdash \textbf{region}\, x \,\textbf{in}\, t : T} \qquad (\textsc{Region})
$$

$$
\frac{\Gamma \vdash x : C\ \text{Region} \qquad \Gamma \vdash t : T}{\Gamma \vdash x.\,\textbf{new}[T]\ t : \{x\}\ T} \qquad (\textsc{New})
$$

Fig. 1. Extending $\mathsf{CC}_{<:\square}$ with support for safe regions.

As mentioned in 1.2.3, unsafe regions are already supported in Scala Native, which utilizes API that employ the built-in equivalent of C's pointers to access objects allocated in these regions. Pointers are unsafe since it provides a mechanism for users to access invalid memory. So:
Key Property 1: Access to objects allocated in safe regions should be restricted to the objects themselves rather than pointers that reference them.

Restricting access to objects in regions to objects themselves prevents users from accessing unallocated memory. A region is essentially a lexically delimited scope which starts from where it is created and ends at where the entire region is deallocated [Boruch-Gruszecki et al. 2021]. The deallocated memory may still be accessible in certain circumstances:

(1) Any reference to data allocated in a region remains after the region is left. [Boruch-Gruszecki et al. 2021]
(2) Any reference from objects allocated in regions to objects in the heap is not recognized by the garbage collector, leading to the possibility of the referenced object being deallocated by garbage collector prior to the deallocation of the region.

The above-mentioned violations of memory safety indicate the following key properties of safe regions:
Key Property 2: No reference to data allocated in the safe region remains after safe region is left.
Key Property 3: Garbage collector must be able to scan safe regions for references to objects within the heap to preserve garbage-collected heap's integrity [Bollella and Gosling 2000].

2.2 Safe Regions As Capabilities

2.2.1 Extending $CC_{<:\square}$ with Suppports for Safe Regions. Key Property 2 is a natural fix for being expressed with $\mathsf{CC}_{<:}$ [Boruch-Gruszecki et al. 2021]. Figure 8 in [Boruch-Gruszecki et al. 2021] shows the extension of $\mathsf{CC}_{<:}$ with support for region-based memory management in which users access objects allocated in regions through pointers, violating Key Property 1.

Figure 1 presents a preliminary design for augmenting $CC_{<:\square}$ with safe regions, incorporating both Key Property 1 and 2 into consideration. Similar to the discussion in Section 3.3 of [Boruch-Gruszecki et al. 2021], the soundness of the extension is jeopardized if either the region variable or an object within the region leaves the region. This potential issue is addressed by the non-derivation subcapturing precondition on rule (REGION).

2.2.2 *Safe Region API Using Capturing Types.*

The safe region extension of $CC_{<:\square}$ guides the design of safe region API in Scala. The API for creating and deallocating a safe region, as well as utilizing it within an explicit scope, is displayed in Listing 1.

```
trait SafeZone {
  /** Deallocate the entire region. */
  def close(): Unit
}
object SafeZone {
  /** Create a new safe region. */
  final def open(): {*} SafeZone
  /** Run given function with a fresh zone and deallocate it afterwards. */
  final def apply[T](f: ({*} SafeZone) => T): T = {
    val sz: {*} SafeZone = open()
    try f(sz)
    finally sz.close()
  }
}
```

Listing 1. Basic Safe Region API. The name `SafeZone` is used to distinguish it from the unsafe region, known as `Zone` in Scala Native.

2.2.3 *Instance Creation Expression Specifying a Safe Region.*

We introduce a new and intuitive syntax for creating instances within a specified safe region, which is in form of `new { CaptureRef } ConstrApp`. The complete definition is available in [EPFL 2021].

Informally, the syntax can be represented as **`new`** `{sz} T(args)` where `sz` is a `SafeZone` instance, `T` is a type, `args` are any arguments such that **`new`** `T(args)` is a valid instance creation expression. The result type of **`new`** `{sz} T(args)` is $C \cup \{sz\}\, S$, where $C\, S$ represents the result type of **`new`** `T(args)`. Listing 2 shows an example using the new syntax.

```
class A {}
class B (a: {*} A) {}
SafeZone { sz0 =>
  SafeZone { sz1 =>
    val array: {sz0} Array[{sz1} A] = new {sz0} Array[{sz1} A](10)
    val b: {sz0, sz1} B = new {sz0} B(new {sz1} A)
  }
}
```

Listing 2. An example using new syntax. In this example, `array` is allocated in safe region `sz0` containing 10 `null` references. These references has potential to point to instances of class `A` which are allocated in `sz1`. Similarly, `b` is allocated in `sz0` containing a reference pointing to its class `A` member allocated in `sz1`.

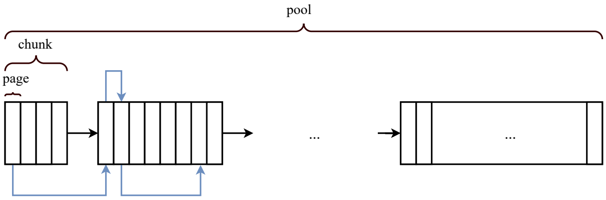

Fig. 2. Allocation Strategies of Safe Regions. An example region is represented by a linked list of four memory pages, illustrated by the blue arrows.

2.2.4 *Scala Native Object Memory Layout and Value Members.*

Memory layout of Scala Native objects is similar to that of Scala objects and Java objects. Typically, The object memory layout can be divided into the following three parts [Worldz 2021]:

- Header: save the object self's runtime information as well as typer pointer.
- Instance Data: saves the variables defined in the class, including the variable defined in the parent class. It stores value members (such as members of primitive data types) directly in the memory, and references for object members as pointers to the location of the referenced object in memory.
- Padding: if an object's instance data doesn't align, it needs padding.

```
case class A(v: Int) {}
SafeZone { sz => val a = new {sz} A(0) }
```

In the above example, the memory layout of a contains an integer (a.v) which is a value member. There seems to be a contradiction: a.v is stored in safe region sz, but the type of a.v is pure type Int which does not contain any safe region instance in its capture set, indicating a.v is allocated in heap. It should be noted, however, that the actual allocation location of value members such as a.v, is inconsequential as they are consistently duplicated onto the stack whenever utilized and lack references to any objects.

2.3 Safe Regions As GC Roots

As mentioned in Section 1.2.2, Scala Native limits itself to conservative garbage collectors. Each collection cycle of conservative garbage collectors starts with a *root scanning* phase, during with all references from *GC roots* are detected and the referenced objects on the heap are marked [Siebert 2001].

A *garbage collection root*, also known simply as a GC root, is an object that is accessible from outside the heap. All objects directly or indirectly referenced from a GC root are considered live objects thus not garbage collected. Key Property 3 stipulates the inclusion of every safe region into the set of GC roots, and its subsequent removal from the set upon deallocation of the region.

2.4 Allocation Strategy of Safe Regions

Figure 2 shows the allocation strategy of safe regions. A memory pool is constituted by a linked list of memory chunks, each being a contiguous block of memory outside the stack and garbage-collected heap. Each chunk is divided into a number of memory pages, with regions being represented by a linked list of these pages. The allocation of each object to a region begins by attempting to place it in the last page of the region. If there is insufficient space in this page, the region claims a new page from the memory pool. If there are insufficient pages available in the pool, the pool will attempt to allocate a new chunk. Region deallocation simply reclaims all pages of the linked back to the memory pool.

Typically, each chunk size is successively doubled until it reaches the maximum chunk size. The chunks are then divided into a number of standard memory pages with fixed size ($size_{standard}$). Specifically, if the size of the object ($size_{object}$) surpasses that of the standard pages, the region will attempt to acquire a specially designated large page, whose size ($size_{large}$) satisfies $size_{large} = size_{standard} \cdot 2^n$, where n is the minimum natural number such that $size_{large} \geq size_{object}$.

3 IMPLEMENTING SAFE REGION IN SCALA NATIVE

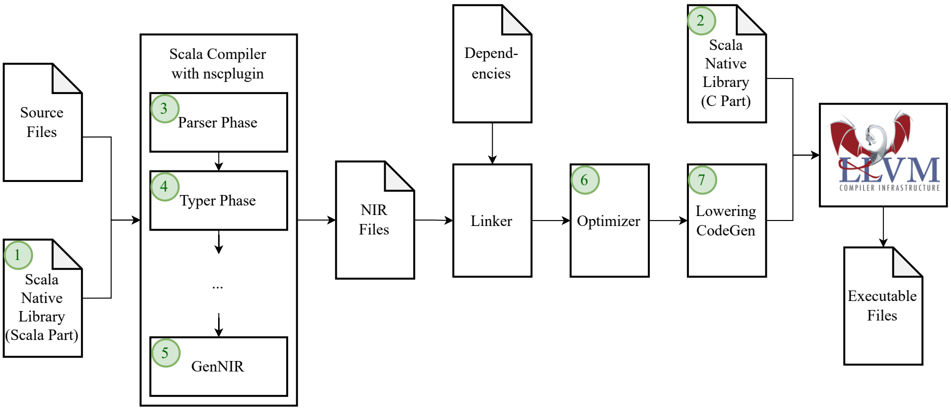

Fig. 3. Implementation Overview.

(1) Add Safe Region Scala API in Scala Native library.
(2) Add Safe Region C API and implement safe region allocation stragety in C in Scala Native library. In Safe Region C API, each region operation needs to specify a region handle which is essentially a pointer to the low-level data structure of a region. Note that each page of the region is added to the GC roots when it's claimed from the pool, and is removed when it's reclaimed back to the pool.
(3) Add new syntax supporting instance creation expressions that specify a safe region.
(4) Translate expression generated from the added syntax to internal function call after parsing and before type checking. Details of Step 3 and 4 is introduced in Implementing Syntax for Instance Creation Expression Specifying a Safe Region.
(5) Enhance the NIR allocation operands, CLASSALLOC and ARRAYALLOC, by incorporating an additional parameter which is the handle of safe region. When generating NIR code, extract region handle from the translated internal function in Step 4 and pass it to the enhanced allocation operands.
(6) Update optimizer to handle the new parameters added in Step 5. Without this step, the newly added parameters will be optimized.
(7) Lower NIR allocation operands with specified safe regions to function call of Safe Region C API implemented in Step 2.

Implementing Syntax for Instance Creation Expression Specifying a Safe Region As introduced in Section 2.2.3, the syntax can be informally represented as `new {sz} T(args)`. An internal function `withSafeZone` is defined in Scala Native, which is used to preserve the safe region instance as a term in the source code:

Table 1. Experimental results.

Memory Management Strategy	Page Size	Operations	Average Execution Time
Only GC	Ø	Standard	944 ms
Using safe regions	2^{12} bytes	Standard	673.8 ms
Using safe regions	2^{13} bytes	Standard	589.3 ms
Using safe regions	2^{16} bytes	Standard	629.2 ms
Using safe regions	2^{32} bytes	Standard	651.6 ms
Only GC	Ø	Extra tree-walking	1056.4 ms
Using safe regions	2^{13} bytes	Extra tree-walking	645.8 ms

```
def withSafeZone[T](sz: {*} SafeZone, obj: T): {sz} T = obj
```

After parser phase, the corresponding expression is translated to `withSafeZone(sz, new T(args))`. Later, this internal function will be handled specifically in the GenNIR phase, as shown in Step 5.

4 EXPERIMENTAL RESULTS

4.1 Benchmark

Hans Boehm's GCBench [Systems 2021] is a widely-used benchmark for measuring the performance of garbage collection algorithms in binary trees. This benchmark is also used for comparing region-based memory management and garbage collection algorithms.

Hans Boehm's GCBench focuses on evaluating memory management algorithms in the context of binary trees. The performed operations includes allocating a temporary binary tree to "stretch" memory, allocating a long-lived binary tree which will live-on while other trees are allocated and deallocated, as well as allocating, walking and deallocateing many bottom-up binary trees.

A modified version Hans Boehm's GCBench in Scala in adapted to test safe regions in Scala Native.

4.2 Performance

The benchmark is ran on my personal laptop with 2.3 GHz quad-core Intel Core i7 processor and 16GB memory. Each case has ran for 20 times.

As shown in table 1, the use of safe regions significantly reduces the execution time of the tested program. Under standard operations, programs with safe regions are up to 37.6% faster in execution time. This number slightly increases to 38.9% with extra tree-walking operations added to those bottom-up binary trees to increase the access frequency of some memory locations.

5 FUTURE WORK

Formalize safe region. The next step in the development of safe region is to formally prove its memory safety, both in paper and in Coq. This work will provide a deeper understanding of the properties of safe regions and will help establish their use as a reliable and secure memory management technique for performance-critical applications.

Support sub-regioning. One possible extension is *sub-regioning*. The sub-region problem is: given two regions x and y, with y being shorter-lived (or more nested), can we pass an object of type $\{x\}\ T$ where an object of type $\{y\}\ T$ is expected? As discussed in [Boruch-Gruszecki et al. 2021], in order to support sub-regioning, we need to extend the current $CC_{<:\square}$ calculus with the ability to know $\{y\} <: \{x\}$ based on the bound of y, i.e. the ability to put lower bounds on capture sets of term variables.

Support `struct`. Explicitly specifying the potential memory areas of each member of an instance is not user-friendly. The idea is to support `struct`, a composite data type that groups together values of different data types under a single name. By declaring a `struct` instead of a `class`, the users choose to flatten the memory layout of all members and implicitly require all members to be allocated in the same memory area such as a safe region. This approach will provide a more intuitive and streamlined experience for the user. However, support `struct` is challenging as the argument in Section 2.2.4 no longer holds when considering the special value types `struct`. We need to extend the current type system to include the ability to infer the underlying memory layout between different value types.

Extend libraries to provide user-friendly API for commonly-used data structure. The current API of commonly-used data structures assumes objects should be allocated in the heap. For example, the `clone` method of `Array` assumes the cloned array should be allocated in the heap. The goal is to expand the libraries in the future to offer a user-friendly API that facilitates the utilization of safe regions.

6 CONCLUSION

This report introduces safe regions, an approach of region-based memory management whose memory safety is guarantee by the use of capture checking and GC adaptations that preserve the integrity of the garbage-collected heap. Safe regions in implemented in Scala Native, a ahead-of-time compiler and optimizing runtime for Scala which supports low-level primitives as well as the full Scala language, making it a even more suitable choice for systems programming and other performance-critical tasks. Experiment using the Hans Boehm's GCBench benchmark rograms using safe regions is up to 37.6% faster on execution time compared to those without, and this improvement is slightly amplified with the addition of operations increasing the access frequency of some memory locations. In the future, it would be interesting to formally prove the memory safety of safe region, support sub-regioning and struct as well as provide more user-friendly API in the library.

REFERENCES

Gregory Bollella and James Gosling. 2000. The real-time specification for Java. *Computer* 33, 6 (2000), 47–54.

Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, Ondřej Lhoták, and Martin Odersky. 2021. Tracking Captured Variables in Types. *arXiv preprint arXiv:2105.11896* (2021).

Scala Native Contributors. 2017. Scala Native: A ahead-of-time Compiler and Optimizing Runtime for Scala. https://scala-native.org/

EPFL. 2021. Dotty Internals - Syntax. https://dotty.epfl.ch/docs/internals/syntax.html.

David Gay and Alex Aiken. 2001. Language support for regions. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation.* 70–80.

M Teresa Higuera-Toledano, Sergio Yovine, and Diego Garbervetsky. 2012. Region-Based Memory Management: An Evaluation of Its Support in RTSJ. *Distributed, Embedded and Real-time Java Systems* (2012), 101–127.

Nicholas D Matsakis and Felix S Klock II. 2014. The rust language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104.

Martin Odersky, Aleksander Boruch-Gruszecki, Edward Lee, Jonathan Brachthäuser, and Ondřej Lhoták. 2022. Scoped Capabilities for Polymorphic Effects. *arXiv preprint arXiv:2207.03402* (2022).

Guillaume Salagnac, Sergio Yovine, and Diego Garbervetsky. 2005. Fast escape analysis for region-based memory management. *Electronic Notes in Theoretical Computer Science* 131 (2005), 99–110.

Denys Shabalin. 2020. *Just-in-time performance without warm-up.* Ph.D. Dissertation.

Fridtjof Siebert. 2001. Constant-time root scanning for deterministic garbage collection. In *Compiler Construction: 10th International Conference, CC 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS*

2001 Genova, Italy, April 2–6, 2001 Proceedings 10. Springer, 304–318.

Debian Systems. 2021. The Computer Language Benchmarks Game - Binary Trees. https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/binarytrees.html#binarytrees.

Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. 2004. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation* 17 (2004), 245–265.

Magic Worldz. 2021. Java Object Memory Layout. https://magicworldz.dev/blog/Posts/2020/2021-08-12-java-object-memory-layout/#mark-word Accessed: 2023-02-05.