VRIJE
UNIVERSITEIT
BRUSSEL

OPTIMISING STATIC PROGRAM ANALYSIS USING SCALA NATIVE

Abdullah Sabaa Allil

May 31, 2023

Promotor: Prof. Dr. Coen De Roover
Advisors: Noah Van Es, Jens Van der Plas and Bram Vandenbogaerde

sciences and bioengineering sciences

Abstract

Analyzing programs using static analysis tools might not always be as fast as the programmer expects it to be. In this dissertation, we investigate some possibilities to optimise analyses created in MAF ("Modular Analysis Framework") by rewriting some performance-critical parts of MAF in a low-level implementation. Profiling has pointed out that abstract domains are a good candidate for optimisation. We present a low-level implementation for an abstract domain, supported by a domain-specific memory manager that is responsible for garbage collection during the analysis.

Contents

Chapter 1

Introduction

Static program analysis helps programmers reason about the correctness and the behaviour of their programs without executing them. In some cases, analyzing a program could take several minutes, or even hours, which becomes an inconvenience for the programmer. Static analysis needs to provide the programmer with results within a relatively reasonable time in order to be deemed valuable.

MAF ("Modular Analysis Framework") is a framework in which modular static analyses can be created. The framework is implemented in Scala. Therefore, analyses that are created in MAF are, by default, compiled to Java bytecode, which means that they run on a Java Virtual Machine.

The goal of this dissertation is to investigate the possibilities to improve the performance of modular analyses created in MAF by rewriting some performance-critical parts of the framework in an optimised, low-level implementation. An analysis uses for example lattices to represent values and operations in the analyzed programs. Rewriting the lattices in low-level code might improve the performance of the analysis, given that their values and operations are very frequently used in the analysis, since they form the building blocks of that analysis. Profiling analyses helped us identify that a significant amount of time of an analysis is spent in the implementation of lattices. Therefore, optimising this implementation might have the most noticeable performance gain.

Since the framework is written in Scala, Scala Native can be used for this purpose. Using Scala Native, Scala code gets compiled to native code that runs natively on the machine. Scala Native also offers features for integrating low-level constructs, such as C types and manual memory management, directly into Scala, which can be used for the low-level implementation.

This dissertation makes the following contributions:

- We present a low-level implementation for the constant propagation abstract domain for the function-modular analysis of Scheme programs that is created in MAF.
- We present a domain-specific approach for managing the memory of the low-level abstract domain elements by using specific properties of the analysis, allowing elements to be efficiently deallocated.
- We evaluate the performance of the low-level abstract domain and the memory manager and compare it with several implementations, including one that runs on a Java Virtual Machine.

Before describing the implementation of the abstract domain, we explain the background needed to grasp the concepts of this dissertation.

Chapter 2

Background

2.1 Static Analysis

Static program analysis aims at reasoning about the runtime behaviour of a program without executing that program. This behaviour can be for example reaching an error state such as a buffer overflow, where a program writes outside the bounds of an array [11].

2.1.1 Abstract Domain

Static program analysis does not usually work with concrete values due to uncomputability [10, 11]. It is impossible to reason about all the states that a program can reach during its execution. For instance, if a program takes an integer as input, testing input values would require considering *all* possible input values, which include all possible integers, as well as all possible strings, real numbers, etc. Therefore, static program analysis uses an abstract representation for the values that are used in a program. In the context of the example of the integer input, the input could be represented using 4 types of abstract elements: an element that represents all positive integers, an element that represents all negative integers, an element that represents all integers (i.e., positive or negative integers), and an element that represents non-integer values. This is called an abstract domain. An abstract domain is a set of abstract values that are used to represent concrete values with common properties.

2.1.2 Join Semilattice

In abstract domains, abstract values are usually represented using join semilattices [8]. A join semilattice is a partially ordered set with a least upper bound for each two elements of that set, which means that each two elements can be joined together using a join operator [3]. The join operator joins two elements of the lattice and produces the least upper bound. That is the "smallest" upper bound of two elements in that lattice. In the context of the example in Section 2.1.1, a lattice can be used to represent the abstract elements of that abstract domain. Joining the element representing all positive integers and the element representing all negative integers results in the element that represents all integers. In static analysis, the join operator is used in order to combine results obtained from analyzing different states of the program, which allows the analysis to produce more accurate information on the program's behaviour [8].

A lattice has a top element and a bottom element. A top element usually represents an element that does not have any useful information, such as the element that represents all

integers in our previous example. A bottom element usually represents a contradictory element to that set [4]. In our previous example, the element that represents all non-integer values would be the bottom element. In the context of static analysis, a bottom element usually represents the set of program states that are unreachable during the analysis of that program. For example, this could be an input argument of a different type than the required input type [8].

A Hasse diagram can be used to represent the lattice elements and their relationships with one another [4]. Figure 2.1 shows an example of such diagrams. This diagram represents the elements of the example in Section 2.1.1. The top element is represented by $\top$ and the bottom element is represented by $\bot$. The element that represents all positive integers is represented in this Hasse diagram by $+$, and the element that represents all negative integers is represented by $-$.

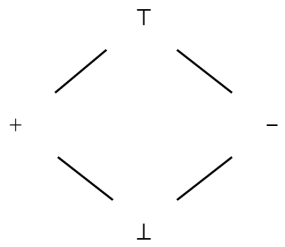

Figure 2.1: An example of a Hasse diagram.

2.2 Modular Static Analysis

Modular static analysis analyzes a program by dividing it into components and analyzing each component in isolation from the other components [2, 14]. These components can be for example modules, classes, or function calls. The analysis combines the results of analyzing the components in order to produce results for the whole program. The function-modular analysis is an example of modular static analysis. In function-modular analyses, components are function calls.

A modular static analysis consists of two parts: Intra-component analysis, and inter-component analysis. In the following subsections, both analyses are explained.

2.2.1 Intra-Component Analysis

An intra-component analysis analyzes a single component in isolation from the other components. Components can also depend on other components, however. A function may call another function in its body. Therefore, an intra-component analysis also keeps track of dependencies that it encounters during the analysis, in order to register them at the end of the intra-component analysis to the global analysis state.

2.2.2 Inter-Component Analysis

The inter-component analysis is responsible for keeping track of the components that need to be analyzed. These components are stored in a worklist. The inter-component analysis is also responsible for deciding which component needs to be analyzed first by an intra-component

analysis. For example, using a Last-In-First-Out worklist algorithm, the most recently added component to the worklist will be the first component to be analyzed by an intra-component analysis.

The inter-component analysis also stores the global analysis state, which consists of results of analyzing components by an intra-component analysis, and results of combining results of analyzed components. The inter-component analysis keeps track of components that have a dependency on parts of the global analysis state.

2.3 Modular Analysis Framework

MAF is a framework in which modular static analyses can be created for higher-order languages [14]. `ModF` is an example of such analyses. `ModF` is a function-modular analysis for Scheme created in MAF. This dissertation aims at optimising analyses created by this framework. Specifically, this dissertation attempts at optimising `ModF` in MAF. In this section, the main components of MAF and `ModF` are explained.

2.3.1 ModAnalysis

`ModAnalysis` is the base class of all modular analyses that are created in MAF. This class is used to implement the inter-component analysis as explained in Section 2.2.2. Therefore, it stores a worklist of the components that need to be analyzed by the intra-component analysis. It also keeps track of which components depend on which parts of the analysis state.

2.3.2 IntraAnalysis

```
abstract class IntraAnalysis(val component: Component):

  /** Set of dependencies read by this intra-component analysis. */
  var R: Set[Dependency] = Set()
  /** Set of dependencies written (triggered) by this intra-component analysis. */
  var W: Set[Dependency] = Set()
  /** Set of components discovered by this intra-component analysis. */
  var C: Set[Component] = Set()
  /** Registers a read dependency. */
  def register(dep: Dependency): Unit = R += dep
  /** Triggers a written dependency. */
  def trigger(dep: Dependency): Unit = W += dep
  /** Spawns a discovered component. */
  def spawn(cmp: Component): Unit = C += cmp
  /**
   * Performs the intra-component analysis of the given component.
   * Important: should only update the *local* analysis state, and must
   * not modify the global analysis state directly.
   */
  def analyzeWithTimeout(timeout: Timeout.T): Unit

  /** Pushes the local changes to the global analysis state. */
  def commit(): Unit =
      R.foreach(inter.register(component, _))
      W.foreach(dep => if doWrite(dep) then inter.trigger(dep))
```

```
        C.foreach(inter.spawn(_, component))
```

`IntraAnalysis` represents an intra-component analysis in the modular analysis. It analyzes a single component in isolation from the other components. As previously mentioned in Section 2.2.1, the intra-component analysis keeps track of different dependencies and components it encounters. Concretely, it keeps track of the following:

- If the currently analyzed component reads a part of the global analysis state, then it has a dependency on that part. These dependencies are stored during the intra-component analysis in the `R` set.
- If the currently analyzed component updates a part of the global analysis state that is a dependency for other components, then the components that depend on that part of the global analysis state need to be (re)analyzed by the intra-component analysis. These parts of the global analysis state are stored during the intra-component analysis in the `W` set.
- If new components are discovered in the currently analyzed component (e.g., a function call in the body of the component that is being analyzed), then these components need to be added to the worklist of the inter-component analysis in order to be analyzed later by the intra-component analysis. These components are stored during the intra-component analysis in the `C` set.

Once the intra-component analysis finishes analyzing the component, all the elements that were locally stored during the intra-component analysis need to be registered in the inter-component analysis. This happens using the `commit` method.

2.3.3 GlobalStore and ReturnValue

When analyzing programs, it is often necessary to emulate the heap memory. Therefore, `GlobalStore` is a trait that provides an analysis with a global store in which the analysis can store results of analyzed components, such as function calls, with their unique address. To manipulate the store during the analysis, methods such as `writeAddr`, to write a value to a memory address, and `readAddr`, to read a value from a specific address, are provided in the trait.

Components, such as function calls, can also have return values. Therefore, the `ReturnValue` trait extends the capabilities of `GlobalStore` in order to store return values, accompanied by their address.

In the context of `ModF`, the global store, with its extended capabilities to store return values, is one of the central components of the analysis, and it is an integral part of the global analysis state, as well as the result of the analysis of a program.

Components can have dependencies on addresses in the store. If an address in the global store is updated (e.g., a return value of a function call is registered), then all components that have a dependency on that address, i.e., components that have previously read the value of that address, need to be reanalyzed.

2.3.4 SequentialWorkListAlgortihm

```
trait SequentialWorklistAlgorithm[Expr <: Expression] extends ModAnalysis[Expr]:
    def emptyWorkList: WorkList[Component]
    var workList: WorkList[Component] = emptyWorkList.add(initialComponent)
    def addToWorkList(cmp: Component) = workList = workList.add(cmp)
    def finished: Boolean = workList.isEmpty
```

```
def step(timeout: Timeout.T): Unit =
    val current = workList.head
    workList = workList.tail
    val intra = intraAnalysis(current)
    intra.analyzeWithTimeout(timeout)
    if timeout.reached then
        addToWorkList(current)
    else
        intra.commit()
def run(timeout: Timeout.T): Unit =
    while !finished && !timeout.reached do step(timeout)
```

`SequentialWorkListAlgorithm` is a trait that can be used to implement worklist algorithms for analyses of which the components are analyzed sequentially. Using the `run` method, the inter-component analysis of the program is initiated. Inside that method, a leading decision loop calls the `step` method, which will perform an intra-component analysis on the first component of the worklist ("first" depends on which worklist is used. The first component of a Last-In-First-Out worklist would be the most recently added component to the worklist, which is different than a First-In-First-Out worklist). Once the intra-component analysis of that component is finished, all local changes of that analysis will be registered in the global analysis state using the `commit` method of `intraAnalysis`.

2.3.5 AbstractDomain and Lattice

As previously explained in Section 2.1.1, analyses use an abstract domain to approximate values in the analysis. `AbstractDomain` represents an abstract domain that is used in an analysis. The values of an abstract domain are represented by the `Lattice` trait, which is a join semilattice structure that, as explained in Section 2.1.2 has a top element, bottom element, and a join operator to produce the least upper bound.

2.3.6 SchemeLattice and SchemeDomain

The `SchemeLattice` trait is an interface that represents the base lattice for Scheme that can be used in an abstract domain for `ModF`, augmented with Scheme-specific operations. These operations can be for instance injecting primitive values, such as strings and integers, into the lattice (i.e., creating an element in the lattice from the provided primitive value), or testing whether a given value should be considered true in a conditional.

The `SchemeDomain` trait is an interface that represents an abstract domain for Scheme programs in ModF.

2.3.7 Core Lattices of MAF

MAF provides lattices to represent some elements that are frequently used in analyses as well as their operations. These lattices represent for instance boolean values, strings, integers, and real values.

- `BoolLattice` represents boolean values in an analysis, and offers an interface for common operations that are used on booleans, such as `isTrue` and `not`.
- `IntLattice` represents integer values in an analysis, and offers an interface for common operations that are used between integers, such as `plus` and `modulo`.

- `RealLattice` represents real values in an analysis, and offers an interface for real operations, such as `sqrt`, `ceiling` and `log`.
- `CharLattice` represents character values in an analysis, and offers an interface for operations that are used on characters, such as `isLower` and `upCase`.
- `StringLattice` represents string values in an analysis, and offers an interface for operations that are used on strings, such as `append`, `substring`, and `length`.
- `SymbolLattice` represents symbol values, which are commonly used values in Scheme. Its interface offers the `toString` operation, which converts a symbol to an element of `StringLattice`.

2.3.8 ConstantPropagationlattice

```
object ConstantPropagation:
    sealed trait L[A]

    case object Top extends L[Nothing]

    case class Constant[A](x: A) extends L[A]

    case object Bottom extends L[Nothing]
    ...
    type I = L[BigInt]

    object L:
        ...
        implicit val intCP: IntLattice[I] = new IntLattice[I] {
        ...
        private def binop(
        op: (Int, Int) => Int,
        n1: I,
        n2: I
        ) = (n1, n2) match
            case (Top, Top) => Top
            case (Top, Constant(_)) => Top
            case (Constant(_), Top) => Top
            case (Constant(x), Constant(y)) => Constant(op(x, y))
            case _ => Bottom
        def plus(n1: I, n2: I): I = binop(_ + _, n1, n2)
        ...
        }
        ...
```

Constant propagation is an evaluation technique in which variables are replaced by their constant values. For the given variables $x = 10$ and $y = x + 5$, propagating x in y means that x will be replaced by its constant value, thus $y = 10 + 5$.

Constant propagation is an example of how the operations of the core lattices in Section 2.3.7 could be implemented. For instance, the `plus` operation of `IntLattice` can be implemented to evaluate the addition of two arguments if both arguments are known constants. If at least one of both arguments is $\top$, meaning that there is no useful information known about that argument, then the result would also be $\top$.

Another example is appending strings. In the constant propagation domain of the string lattice, appending two strings means creating a new constant value with the result of the concatenation of both strings, if they are both known constants.

2.3.9 ModularSchemeLattice and Value

```
class ModularSchemeLattice[A <: Address, S: StringLattice, B: BoolLattice, I: IntLattice, R:
    RealLattice, C: CharLattice, Sym: SymbolLattice]
  extends Serializable
  with SchemeLattice:
  ...
  trait Value
  ...
   case class Int(i: I) extends Value:
     override def toString: String = IntLattice[I].show(i)
  ...
     object Value:
       ...
       def op(op: SchemeOp)(args: List[Value]): Value =
       op match
         ...
         case Plus =>
           (args(0), args(1)) match
                 case (Int(n1), Int(n2)) => Int(IntLattice[I].plus(n1, n2))
             ...
       ...
```

The `ModularSchemeLattice` trait is a wrapper that is used to combine lattices of Scheme primitive values, such as a string lattice and an integer lattice. `ModularSchemeLattice` is modular in the sense that the analysis programmer can choose which implementation for each core lattice to use in `ModularSchemeLattice` (e.g., constant propagation).

The `Value` trait and its companion object are used to implement the primitive values and operations of the modular Scheme lattice. The `Int` case class for example is a wrapper around elements from `IntLattice`, and thus represents integers inside the `ModularSchemeLattice`. There are operations that could be performed on integers, such as `Plus` and `Minus`. These operations originate from the respective lattice and can be implemented using `ConstantPropagationlattice` that is explained in Section 2.3.8 for example.

2.4 Abstract Domains as a Candidate for Optimisation

The function-modular analysis for Scheme was profiled in order to determine the parts of the analysis that consume a substantial amount of time. This could help us identify the parts of the analysis that could possibly be optimised. The initial hypothesis is that abstract domains and their lattices are a good candidate for optimisation. Abstract domains and their operations form the building blocks for analyses. Therefore, an analysis is very likely spending a lot of time in primitive operations and values.

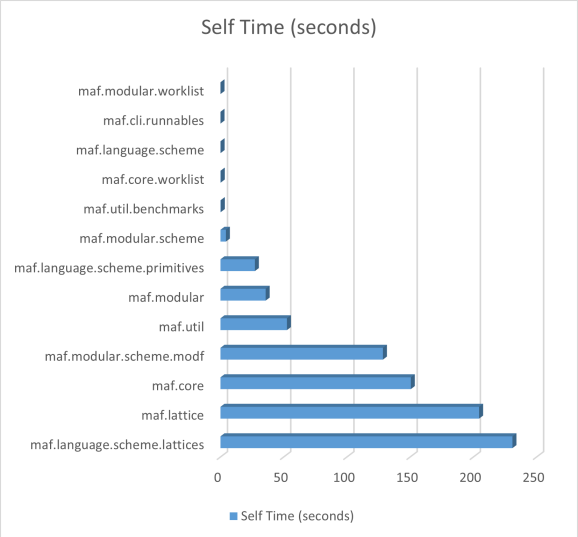

Figure 2.2: CPU profiling results aggregated by the package.

The profiling results confirmed our initial hypothesis. Indeed, as illustrated in Figure 2.2, it is clearly observed that an analysis spends a significant amount of time in lattices and their operations [12]. Therefore, this dissertation aims at investigating potential performance improvements in analyses if abstract domains are implemented using an optimised, low-level implementation.

2.5 Scala Native

By default, Scala code is compiled into Java bytecode that runs on a Java Virtual Machine. Using Scala Native, it is also possible to compile Scala code to machine code. Scala Native is an ahead-of-time compiler that compiles Scala code to machine code (using LLVM), which will then run natively on the host machine, instead of running on a Java Virtual Machine [5]. Scala Native has low-level constructs, such as C types, and manual memory management. Scala Native also offers an interoperability layer in order to interoperate with low-level languages such as C.

Scala Native is the technology used in this dissertation. Since it offers low-level features, it is possible to write and integrate a low-level implementation in Scala. It is important to take into consideration, however, that Scala Native is still an experimental technology, and no functionalities of Scala Native are guaranteed to perform as expected.

2.5.1 Scala Native Memory Management

Scala Native includes a garbage collector, which is Immix GC by default, that automatically manages memory for objects and class instances [5]. Similar to Scala Native programs, this garbage collector also runs natively on the host machine.

Memory can also be manually allocated for C types [5]. This could be useful to interact with external C functions that have pointers as arguments, or to improve the performance of a Scala Native program by minimizing the amount of work the garbage collector has to do. For example, encoding frequently used elements in a byte sequence, and manually managing their memory, reduces the need for the garbage collector.

This is different from the Scala memory model, which relies on the Java Virtual Machine memory model. Given that Scala is compiled with the just-in-time compiler to Java bytecode which will run on a Java Virtual Machine, the Java Virtual Machine will load classes dynamically at run time to the heap memory. Generally, the Java Virtual Machine will store all objects on the heap memory. Variables are references to those objects, and can be stored on the stack, e.g., when calling a function that has local variables [6]. The garbage collection is taken care of by the Java Virtual Machine. Scala has no way of manually allocating memory.

2.5.1.1 Manual Memory Allocation for C Types

Scala Native offers multiple kinds of manual memory management: heap memory management, stack memory management, and Zone memory management (which is a technique that is used to perform semi-automatic memory management) [5]. In order to avoid the overhead of automatic memory management, we use manual heap memory management. Heap memory can be manually managed in Scala Native using the bindings that Scala Native offers for the C functions that perform dynamic heap memory manipulation [5]. These functions are `malloc`, `free`, and `realloc`. Using these functions, memory from the heap can be dynamically allocated and freed once it is no longer needed. It is the responsibility of the programmer to free the dynamically allocated heap memory. If the allocated memory is not freed, the program will suffer from memory leaks.

2.5.2 Features of Scala Native

2.5.2.1 Low-level Primitives

Scala Native provides built-in equivalents of C types that are frequently used. A complete list can be found in the documentation of the `scala.scalanative.unsafe` package[1]. Table 2.1 shows a list of the most popular ones, alongside their Scala Native equivalent.

C type	Scala Native
`void`	`Unit`
`int`	`CInt`
`int*`	`Ptr[CInt]`
`long`	`CLong`
`char`	`CChar`
`char*`	`CString`
`struct { int x, char *y, uint8_t z; }*`	`Ptr[CStruct3[CInt, CString, Byte]]`

Table 2.1: Popular C types and their Scala Native equivalent.

Scala Native also supports pointer arithmetic operations in order to manipulate memory pointers, such as dereferencing a pointer variable. These operations and their Scala Native equivalents can be found in Table 2.2 [5].

Having low-level primitives with pointer operations integrated in Scala Native allows us to directly write low-level code in Scala, instead of relying on an implementation from a low-level language like C.

[1] `https://javadoc.io/doc/org.scala-native/nativelib_native0.4_3/latest/scala/scalanative/unsafe.html`

Operation	C syntax	Scala Native syntax
Load a value from an address	`*ptr`	`!ptr`
Store a value to a given address	`*ptr = value`	`!ptr = value`
Pointer to an index	`ptr + i; &ptr[i]`	`ptr + i`
Load a value from an index	`ptr[i]`	`ptr(i)`
Store a value to an index	`ptr[i] = value`	`ptr(i) = value`
Pointer to a field of a struct	`&ptr->name`	`ptr.atN`
Load a value from a field of a struct	`ptr->name`	`ptr._N`
Store a value to a field of a struct	`ptr->name = value`	`ptr._N = value`

N is the index of that field in the struct.

Table 2.2: Pointer operations in C and their Scala Native equivalent.

2.5.2.2 Native Code Interoperability

As mentioned earlier, Scala Native makes it possible to interoperate with low-level programming languages such as C. In order to do so, `extern` objects can be used. `extern` objects declare methods that are defined elsewhere in a native library in the library path, or in the Scala Native resources. Inside an `extern` object, bindings of the methods, with which Scala Native interoperates, should be defined. This binding is the Scala Native equivalent signature of the signature of that method in its native implementation. Below is an example of such objects. The `test` method takes two arguments, a C integer, and a pointer to a null-terminated C string. This method is defined elsewhere in native code.

```
import scalanative.unsafe._

@extern
object myapi {
    def test(a: CInt, b: CString): Unit = extern
}
```

2.5.3 Scala Native Configurations

Scala Native includes an optimiser that offers a lot of optimisations for the code during the compilation phase in order to achieve a good performance [13]. The following Scala Native optimisation modes were used:

- `debug` mode: This is the default mode for Scala Native. It offers the shortest compilation time, but performs fewer optimisations on the code. This mode was employed during the implementation of the work presented in this dissertation.
- `release-full` mode: This mode performs all optimisations on the code in order to achieve the best runtime performance. This mode was employed during the performance evaluation that is explained in Chapter 6.

Additionally, Scala Native also offers some extra optimisations that are applied at link time. Specifically, the `thin` link time optimisation mode is used during the performance evaluation, which uses LLVM's `ThinLTO` optimisations to optimise C code with which the Scala Native program is interoperating.

The heap size that the garbage collection of Scala Native uses can also be configured. This could be achieved by setting the environment variables `SCALANATIVE_MAX_HEAP_SIZE` and

`SCALANATIVE_MIN_HEAP_SIZE` in order to respectively update the maximum heap size, and the initial heap size of the garbage collector. This will be used during the performance evaluation so that programs that run natively have the same heap size as programs that run on the Java Virtual Machine.

2.5.4 MAF Cross-compilation using Scala Native

MAF is by default compiled to Java bytecode that runs on a Java Virtual Machine. Using cross-compilation, MAF already has a secondary build option to compile to JavaScript using Scala.js. For this dissertation, cross-compilation support for Scala Native is added to MAF. That is, using Scala Native, MAF is compiled to native code that runs natively on the machine.

Chapter 3

Overview of the Native Lattice

For this dissertation, we developed a low-level implementation of the different lattices that are used in the abstract domain of `ModF`, which we refer to as the *Native Lattice*. This chapter provides a comprehensive explanation of the Native Lattice implementation. The main idea is that high-level constructs are eliminated, and optimised low-level constructs are used instead, in order to gain some additional performance.

The Native Lattice is based on the Constant Propagation Lattice that was explained earlier in this dissertation. Lattice elements can be either a top element, a bottom element, or a constant element. In the Native lattice, we no longer use case classes and objects to represent the lattice elements. Instead, each lattice uses its own representation of a top element, a bottom element, and constant elements depending on how these elements can be efficiently encoded using a low-level implementation.

3.1 BoolLattice

The `BoolLattice` is used to represent boolean values and their operations inside an abstract domain. Since there are only 4 different sorts of elements inside this lattice, it would make sense to encode them in bits. Concretely, the elements are encoded as follows:

Element	**Bitwise Representation**
`false`	`0000`
`true`	`0001`
`bottom`	`0010`
`top`	`0100`

Table 3.1: The bitwise representation of the `BoolLattice` elements.

Figure 3.1 illustrates a Hasse diagram representing the `BoolLatice` elements and the relationships among these elements. Using this bitwise representation, booleans can now be encoded in a 4-bit data type. Since a byte is the smallest data type that can be used for values, this is used to store the values of the `BoolLattice`.

The bitwise representation is not arbitrarily chosen, but rather to make it easier and more efficient for the operations of this lattice to be implemented. Indeed, if we consider the implementation of the `isTrue` operation as an example, we can easily check whether the given element is a constant (i.e., true or false), the top element, or the bottom element.

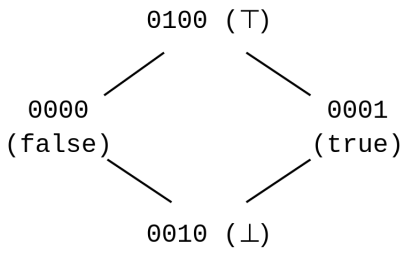

Figure 3.1: A Hasse diagram for `BoolLattice`.

If the given element is a constant, then we can apply a bitwise `and` with the `0001` mask, which will be `0000` if the element is false, or `0001` if the element is true.

If the element is a top or bottom element, however, we can obtain the results by simply, and efficiently shifting the byte 2 times to the right.

```
uint8_t boolIsTrue(uint8_t b)
{
    if(b < 2) {
        return b & 1;
    }
    return b >> 2;
}
```

The other operations of the `BoolLattice` are similarly implemented.

The operations were implemented in C and imported in the `BoolLattice` implementation using `extern` objects that were explained in Section 2.5.2.2. Some operations in `BoolLattice` need to return a Scala boolean (i.e., `true` or `false` from the `Boolean` data type of Scala). `isTrue` and `isFalse` are examples of such functions. Therefore, an additional function is written in order to convert the results from the C functions that implement these lattice operations, which return `0` or `1`, to Scala booleans. This is done by simply checking whether the return value is equal to one (i.e., `i == 1.toByte`).

3.2 IntLattice and RealLattice

The elements of the `IntLattice` and `RealLattice` are both represented using literals without any additional wrappers. Concretely, elements of the `IntLattice` are represented using `CInt`, and elements of the `RealLattice` are represented using `CDouble`. For the `IntLattice`, the top and bottom elements are respectively the maximum value for an integer, and the minimum value for an integer. For the `RealLattice`, the top and bottom elements are respectively the maximum value for a double, and the minimum value for a double. The elements of both lattices are illustrated in a Hasse diagram in Figure 3.2, where `Int.MinValue` $<$ `n` $<$ `Int.MaxValue` and `Double.MinValue` $<$ `m` $<$ `Double.MaxValue`.

The simple operations of these lattices such as addition and subtraction are implemented using the underlying system. More advanced operations, such as logarithmic and trigonometric operations, are implemented using the math library of the Standard C Library. The functionalities of the library may be less safe, since there are no `try catch` constructs in C for instance,

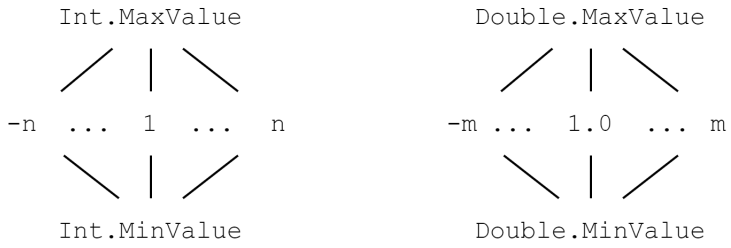

Figure 3.2: Hasse diagrams for respectively the `IntLattice` and the `RealLattice`.

but the performance of these functionalities might be better, due to less safety.

Provided below is the implementation of the `sin` operation from the `ReaLattice` as an example.

```
val top = Double.MaxValue
val bottom = Double.MinValue
...
def sin(n: R): R =
    if(n == top || n == bottom) then n
    else scalanative.libc.math.sin(n)
```

3.3 NativeString

Implementing strings and symbols in a low-level fashion is harder than other data types, since they consist of a sequence of characters. One could use a null-terminated string of which the memory is manually managed. These are the strings that are usually used in a low-level language like C.

Scala Native supports this kind of strings. In the domain-specific language of Scala Native, these are called `CString`, which is essentially a pointer to an array of `CChar` characters.

Some problems might arise, however, when using `CString`: we do not want to calculate the length of the string each time it is requested. To this end, a `struct` can be used to store the length of the string, as well as the pointer to the string.

Below is the `struct` that will be used to store the relevant information of strings in our implementation.

```
type Sn_struct = CStruct3[CInt, CString, CChar]
type S = Ptr[Sn_struct]
```

`CStruct3` is the equivalent primitive of a `struct` C type with three fields in the domain-specific language of Scala Native. The first element of the `Sn_struct` is the length of the string. The second element is the pointer to the string. The third element is the reference counter that will be used by the domain-specific memory management, which will be explained in Chapter 4.

Another problem that we may encounter when using `CString` is checking for equality. The `==` operator between `Ptr` elements will check for referential equality, meaning that two `CString` pointers are equal when they point to the same string in memory. It might also be necessary to check for string equality. That is, if two strings contain the same characters, but they are not

necessarily pointing to the same array of characters in memory, then they are equal. We would also want to implement the necessary string operations, such as appending strings, in a concise way.

To this end, a new class `NativeString` is used to implement the string operations. The `NativeString` is implemented as a value class, meaning that it extends the `AnyVal` class of Scala. `AnyVal` is the root class for all value classes, such as integers, in Scala. Value classes in Scala are used for their efficiency advantages and low runtime overhead [9].

The `NativeString` class stores a pointer to a `Sn_struct` that is mentioned above. It implements different methods that are needed for the string elements, such as referential equality, string equality, appending strings, getting a substring, etc.

Since lattice elements cannot be imperatively modified, some optimisations can be made. When comparing two strings using string equality, for instance, referential equality of both strings can be first compared. If they both point to the same array of characters, then they are equal.

Comparing equality between a `NativeString` and either `top` or `bottom` should only happen via referential equality, since both are unique objects that never get altered. In order to make sure that string equality fails when one of the arguments is `top` or `bottom`, their length is set to respectively `Integer.MaxValue`, and `Integer.MinValue`. This ensures that the string equality fails when the lengths of both arguments are compared, so that the character sequences of the input arguments do not get tested for equality.

```
class NativeString(val underlying: S) extends AnyVal:
    ...
    // assuming lengths are equals
    @tailrec
    private def cstrEquals(s1: CString, s2: CString): Boolean =
        if(!s1 == 0.toByte) then true
        else if (!s1 != !s2) then false
        else cstrEquals(s1 + 1, s2 + 1)

    def ==(other: NativeString): Boolean =
        val l1 = underlying._1
        val l2 = other.underlying._1
        val s1 = underlying._2
        val s2 = other.underlying._2
        eq(other) || (l1 == l2 && cstrEquals(s1, s2))

    // Referential equality.
    // two values are referring to the same object
    def eq(other: NativeString): Boolean =
        this.underlying == other.underlying
```

Since `NativeString` objects are represented in their underlying representation by null-terminated strings, some operations, such as copying and concatenating a string, could be efficiently implemented without the need for additional variables or counters in order to know when a loop is finished. As an example, the `++` implementation is provided below.

```
@tailrec
private def strcpy(dest: CString, src: CString): Unit =
    if(!src == 0.toByte) then
        !dest == 0.toByte
    else
```

```
        !dest = !src
        strcpy(dest + 1, src + 1)

@tailrec
private def strcat(dest: CString, src: CString): Unit =
    if(!src == 0.toByte) then
        !dest = 0.toByte
    else
        !dest = !src
        strcat(dest + 1, src + 1)

def ++(other: NativeString): NativeString =
        val struct = malloc(sizeof[Sn_struct]).asInstanceOf[S]
        val stringLength = this.length + other.length
        struct._1 = stringLength
        struct._2 = malloc(stringLength.toULong + 1.toULong).asInstanceOf[CString]
        struct._3 = 0
        strcpy(struct._2, underlying._2)
        strcat(struct._2 + underlying._1, other.underlying._2)
        val ns = new NativeString(struct)
        NativeString.allocatedStrings += ns
        ns
```

`strcpy` and `strcat` are helper functions that are implemented using tail recursion. The optimiser bears the burden of ensuring that tail-recursive functions are optimised efficiently and correctly by transforming them into loops when compiling the code. In the `strcpy` function for instance, the current pointer of `src`, which is what `strcpy` is copying from, will be dereferenced to check whether the end of the string is reached. If that is the case, then the null-terminator can be added to the destination. Otherwise, the current character of `src` will be copied to `dest`, and both pointers will be incremented for the next iteration.

The `NativeString` companion object creates two elements that are treated differently in some cases: `top` and `bottom`. As the names suggest, these elements will be used in lattices that use `NativeString` to represent their elements. These elements are special in the sense that they only need to be allocated and deallocated once. Specifically, they need to be allocated when the analysis of a program starts, and they need to be deallocated upon completion of the analysis (or, if there will be multiple analyses running in the same program, when the program that runs the analyses starts, and when the program finishes). Since those are unique objects that do not get modified during analyses, it would be inefficient to keep allocating and deallocating top and bottom elements.

The `NativeString` companion object stores all created `NativeString` objects in an `hashSet` for easy retrieval when those need to be deallocated. It also has an `apply` method, which creates new `NativeString` objects from a Scala string.

```
object NativeString:
    var allocatedStrings : mutable.HashSet[NativeString] = mutable.HashSet.empty

    private val ignoreTopLength = Int.MaxValue
    private val ignoreBottomLength = Int.MinValue
    var top: NativeString =
        val temp = NativeString("top")
        temp.underlying._1 = ignoreTopLength
        temp.underlying._3 = 1
        temp
```

```
var bottom: NativeString =
    val temp = NativeString("bottom")
    temp.underlying._1 = ignoreBottomLength
    temp.underlying._3 = 1
    temp

def apply(x: String): NativeString =
    val struct = malloc(SnSize).asInstanceOf[S]
    val stringLength = x.length()
    struct._1 = stringLength
    struct._2 = malloc(stringLength.toULong + 1.toULong).asInstanceOf[CString]
    struct._3 = 0
    var i = 0
    while(i < stringLength) do
        !(struct._2 + i) = x(i).toByte
        i = i +1
    !(struct._2 + stringLength) = 0.toByte
    val ns = new NativeString(struct)

    allocatedStrings += ns
    ns
...
```

Creating a new `NativeString` from a Scala string requires allocating enough memory for the `Sn_Struct` as well as the character sequence for the null-terminated string. After that, the character sequence of the Scala string can be copied to the null-terminated string.

3.4 StringLattice and SymbolLattice

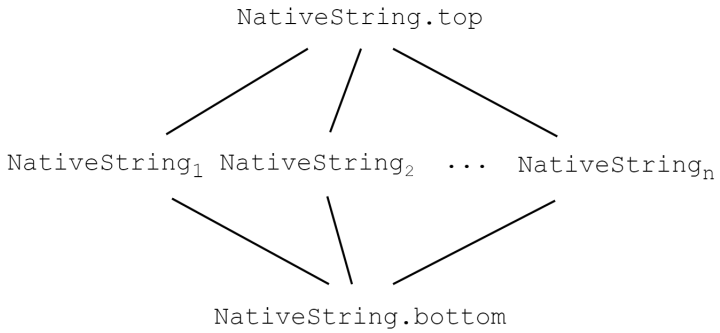

Figure 3.3: A Hasse diagram for the `StringLattice` and the `SymbolLattice`.

Figure 3.3 illustrates a Hasse diagram of the `StringLattice` and the `SymbolLattice`. Elements of the `StringLattice` and `SymbolLattice` are implemented using `NativeString`. $\texttt{NativeString}_m$ indicates a lattice constant represented by a `NativeString` instance.

When applying one of the operations of those lattices, such as `append`, we first need to check whether one of both arguments is the `top` or the `bottom` element. This can be efficiently done using the `eq` method of `NativeString` that checks for referential equality.

Below is the implementation of the `append` operation from the `StringLattice` as an example.

```
def append(s1: S, s2: S): S =
  if(s1.eq(bottom) || s2.eq(bottom)) then bottom
  else if(s1.eq(top) || s2.eq(top)) then top
  else s1 ++ s2
```

First, both arguments are tested to check whether one of them is the top element or the bottom element. If both elements are constants, then the `++` operator will be applied, which will return a `NativeString` containing the concatenation of both strings.

3.5 CharLattice

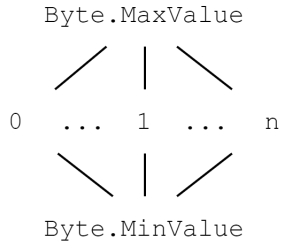

Figure 3.4: A Hasse diagram for the `CharLattice`.

The `CharLattice` is used to represent characters in an abstract domain. Similar to the `IntLattice` and the `RealLattice`, elements of the `CharLattice` are represented by literals, namely: a byte.

The top and bottom elements of this lattice are respectively the maximum value of a byte and the minimum value of a byte. The elements of this lattice are illustrated using a Hasse diagram in Figure 3.4.

Operations of this lattice, such as `toUpper` and `toLower`, are implemented using the functionalities of the Standard C library.

```
override def charEqCI(c1: C, c2: C): BoolLattice =
  if(c1 == top || c2 == top) then BoolLattice.top
  else if(c1 == bottom || c2 == bottom) thenBoolLattice.bottom
  else BoolLattice.inject(scalanative.libc.ctype.toupper(c1) == scalanative.libc.ctype.toupper(c2))
```

3.6 Alternative Implementations for Some Lattices

Some lattices have an alternative implementation. In this section, these alternative solutions are briefly motivated and described. The performance of these implementations are also evaluated in Chapter 6.

3.6.1 StringLattice and SymbolLattice

3.6.1.1 An Implementation using Scala Strings

Instead of manually allocating memory for lattice elements that rely on strings for their representation, another option is to use Scala strings. The memory manager of Scala Native takes care of allocating and deallocating memory for those strings. To this end, an alternative implementation for `StringLattice` and `SymbolLattice` is also tested. This implementation uses Scala strings to represent the lattice elements. To prevent the need for additional wrappings around the strings, which are present in the constant propagation implementation that is explained in Section 2.3.8, a randomly generated string can be used for the top and the bottom elements. This is to avoid using any wrappings around strings. This approach avoids the need for creating additional wrapping objects.

Figure 3.5 illustrates the lattice elements in this alternative implementation using a Hasse diagram. $\texttt{Scala.String}_\texttt{m}$ indicates an instance of `Scala.String` that is used as a lattice element in `StringLattice` or `SymbolLattice`.

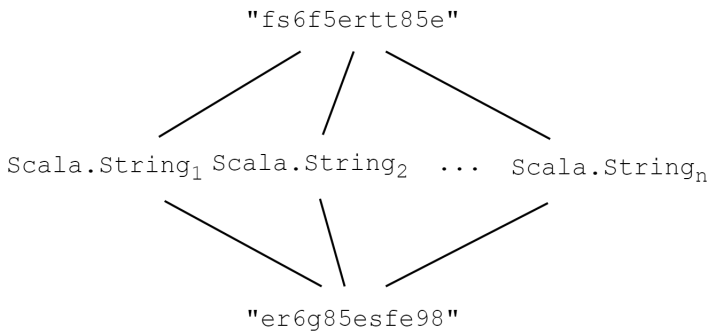

Figure 3.5: A Hasse diagram for the alternative implementation of `StringLattice` and `SymbolLattice`.

Since Scala strings are used, the interface of the Scala strings can now be used to implement the lattice operations. Below is the implementation of `append` as a demonstration.

```
val top: S2 = "fs6f5ertt85e"
val bottom: S2 = "er6g85esfe98"

def append(s1: S2, s2: S2): S2 =
    if(s1 == bottom || s2 == bottom) then bottom
    else if(s1 == top || s2 == top) then top
    else
        inject(s1 ++ s2)
```

3.6.1.2 An Implementation without AnyVal

The `NativeString` class is implemented using the `AnyVal` class of Scala, which is the base class for all value classes in Scala, such as integers. Value classes are intended to be efficient at runtime. However, in some cases, such as when storing value class instances in an array, some

inefficiencies might arise [1]. Therefore, an alternative implementation for the lattice elements of `StringLattice` and `SymbolLattice` was also tested. In this implementation, we do not use any wrappers around the `struct` that was discussed earlier. The operations between the strings are implemented in a `NonClassNativeString` object. They have an identical implementation as the operations of the `NativeString`, with the exception that they now take an additional argument, being the string that was previously the instance of `NativeString` to which messages were sent.

```
object NonClassNativeString:
    type Sn_struct = CStruct3[CInt, CString, CChar]
    type S = Ptr[Sn_struct]
    def ++(first: S, second: S): S =
        /* Same implementation as the NativeString */
```

3.6.2 BoolLattice

Since the `BoolLattice` only has 4 elements, it might also be an option to implement its operations without using any bitwise manipulation. Therefore, an alternative implementation of the `BoolLattice` is provided that directly implements the lattice operations in Scala by simply testing whether the given argument is the top element, the bottom element, or a constant (i.e., true or false). This way, we do not need to calculate the results using bitwise manipulation first and then convert these results to Scala booleans. The elements of this lattice still have the same representation as the implementation described in section 3.1. The implementation of `isTrue` is provided as an example.

```
val top = 4.toByte
val bottom = 2.toByte
def isTrue(b: B): Boolean =
    if(b == top) then true
    else if(b == bottom) then false
    else b == 1.toByte
```

Chapter 4

Domain-Specific Memory Management

As we have seen in section 3.4, the memory of the `StringLattice` and the `SymbolLattice` elements is manually managed. A question that arises: when should the memory for the allocated strings and symbols be deallocated. Naively, all strings and symbols could be deallocated when the analysis is finished, but this might result in a big overhead, since all strings and symbols have to be kept in memory, even the ones that are no longer needed by the analysis. When analyzing a program, a significant number of garbage values can be generated, since some parts of the program need to be repeatedly analyzed.

To this end, some kind of memory management is required in order to deallocate the elements that are no longer needed during the analysis.

4.1 Idea and Pseudocode

Each intra-component analysis keeps its own store in which it stores all the elements that are needed for that analysis. When an intra-component analysis is initiated, that analysis stores a copy of the global analysis state in its store to which it writes results during the analysis. Once an intra-component analysis finishes, the global analysis state needs to be updated with the results from that intra-component analysis (using the `commit` method, as explained in Section 2.3.2). For instance, if there was a dependency on the return value of the component that the intra-component analysis was analyzing, then that dependency should be updated in the global analysis state with the newly calculated results, in case those are different from the previously stored return value for that component in the global analysis state. There are also values in the store of the intra-component analysis that will not be written to the global analysis state. These are values that were needed during the intra-component analysis, but did not have any known dependants for them in the global analysis state. Values of local variables are an example of such values. When a component is analyzed, those local values are no longer needed, and thus can be deallocated. Several questions arise:

- How can we determine which values need to be deallocated?
- When should the garbage values be deallocated?

The answer to the first question is *reference counting*. The idea is as follows: for each value, a counter is stored that keeps track of how many times that value is being stored in the global

analysis state as a dependency. If that counter is zero, it means that the value is no longer needed in the global analysis state, and can be deallocated. The reference counter of a given value gets incremented each time the value is written to the global analysis state.

As mentioned earlier, the global analysis state needs to be updated at the end of an intra-component analysis. This means that garbage values produced by an intra-component analysis need to be deallocated after the global analysis state is updated by that intra-component analysis. This answers the second question. Indeed, after an intra-component analysis commits its changes to the global analysis state, all reference counters are updated, and the values of which the reference counter is equal to zero, are not needed in the global analysis state, nor in the current intra-component analysis, since it is already finished analyzing the component. These values can thus be deallocated.

When updating the global analysis state, we should also take the values that were previously stored in the state into consideration. If a certain part of the global analysis state is updated with a new value that does not equal the old value, then that old value is no longer needed in that part of the global analysis state, which means that its reference counter needs to be decremented, since the new value replaces that old value for that part of the global analysis state.

Provided below is a simplified pseudocode of the algorithm divided in three parts: memory management, intra-component analysis and inter-component analysis. Only the parts that are relevant for the domain-specific memory management are included in the pseudocode.

Algorithm 1 Domain-specific memory management.

```
AllocatedElements ← EmptyDict        ▷ A dictionary containing each allocated value and its reference counter.
procedure CollectGarbage()
    for each (element, referenceCounter) ∈ AllocatedElements do
        if referenceCounter = 0 then
            Deallocate(element)
            AllocatedElements ← AllocatedElements − element
        end if
    end for
end procedure
```

Pointers to all allocated elements will be stored in a dictionary together with their reference counter. The garbage collection iterates over all allocated elements, and deallocates those of which the reference counter is equal to zero.

Algorithm 2 Domain-specific memory management in the intra-component analysis.

```
Dependencies ← EmptySet
Store ← Copy(GlobalAnalysisState)
procedure Analyze(Component)
    /* Intra-component analysis for Component */
    Commit()
end procedure
procedure Commit()
    WriteToGlobalState(Dependencies)
    CollectGarbage()
end procedure
procedure WriteToGlobalState(Dependencies)
    for each dependency ∈ Dependencies do
        UpdateGlobalState(dependency)
        IncreaseReferenceCounter(AllocatedElements, dependency)
    end for
end procedure
```

During an intra-component analysis of a component, elements that were discovered during the analysis will be allocated and thus added to `AddedElements`. When the analysis finishes, `CollectGarbage` will be called in order to deallocate all elements that are no longer needed. When the local changes of the intra-component analysis are committed, some allocated values need to be stored in the global analysis state. Specifically, those are values of addresses in the global analysis state that have dependencies on them. For example, if a component has a dependency on an address of the return value of the currently analyzed component (e.g., because it is also its return value), then this value needs to be updated in the global analysis state, in case it is different from the previous value.

Algorithm 3 Domain-specific memory management in the inter-component analysis.

```
GlobalAnalysisState ← EmptyDict
procedure UpdateGlobalState(dependency)
    if dependency already exists then
        OldDependency ← GetOldDependency(GlobalState)
        DecreaseReferenceCounter(AllocatedElements, OldDependency)
        SetDependency(GlobalState, dependency)
    end if
end procedure
procedure RunAnalysis()
    PrepareMemory()
    Analyze()
end procedure
procedure EmptyAnalysisMemory()
    for each Element ∈ AllocatedElements do
        Deallocate(Element)
    end for
    AllocatedElements ← EmptyDict
end procedure
```

`PrepareMemory` is called before analyzing a given program by the inter-component analysis. This procedure makes sure that the storage for the allocated elements is ready to store elements. If `AddedElements` is a low-level implementation of a dictionary for instance that requires some initialisation (e.g., memory allocation), then this can happen in `PrepareMemory`.

When the global analysis state is no longer needed, the user can free the remaining memory of the analysis by calling `EmptyAnalysisMemory`.

4.2 Example

Consider the following Scheme program:

```
(define (greet name)
  (let ((prefix "hello, ")
        (suffix "!"))
    (string−append prefix name suffix)))

(define (main)
  (greet "Alice"))
```

We assume the entry point to the program is the `main` function. Therefore, this is the initial component of the analysis. The analysis is using a LIFO worklist in this example, which means that the component most recently added to the worklist will be the first component to be analyzed by the intra-component analysis. The worklist will be denoted in this example by a $\mathbb{W}$. Components will be denoted by a $C_{\texttt{f}}$, `f` being the function that is called. This example is a simplified version of how a program in MAF is analyzed. The concrete execution of the analysis in MAF can deviate from the provided steps.

- $\mathbb{W} = \{C_{\texttt{main}}\}$

  The `main` component is first added to the worklist. The component will be then removed from the worklist and analyzed by the intra-component analysis. The intra-component analysis discovers a call to `greet`. Therefore, a component for `greet` is created, and will be registered in the inter-component analysis, in order to be added to the worklist such that it can be analyzed later by the intra-component analysis. Since the return value of the call to `greet` is also the return value of `main`, a dependency between $C_{\texttt{main}}$ and the return value of the call to `greet` is registered.

- $\mathbb{W} = \{C_{\texttt{greet}}\}$

  Next, $C_{\texttt{greet}}$ is analyzed by the intra-component analysis. The argument of the call to `greet` is injected into the string lattice, and therefore, a lattice element of the string lattice is allocated. Inside the body of `greet`, two local strings are defined, `prefix` and `suffix`. Both strings are injected to the string lattice, and lattice elements of the string lattice are thus allocated for them.

  A call to `string-append` is found in the body, which is a primitive function. Therefore, the function is evaluated using the `append` operator from the string lattice. The return value for $C_{\texttt{greet}}$ is updated in the local store of its intra-component analysis. The intra-component analysis for this component is now finished, and the local changes need to be registered in the global analysis state.

  Since there is a dependency on the return value of $C_{\texttt{greet}}$, it needs to be written to the global analysis state. The reference counter of the return value is thus incremented, which means that it is not garbage and should not be deallocated.

As for the other strings, these are no longer needed in the global analysis state, nor in the analysis of other elements. Therefore, their reference counters do not get incremented, and thus stay at zero. This means that these values are garbage and need to be deallocated during this garbage collection cycle.

The return value of $C_{\texttt{greet}}$ is now registered in the global analysis state. All components that have a dependency on this return value need to be reanalyzed by the intra-component analysis. Therefore, $C_{\texttt{main}}$ is added to the worklist in order to be analyzed by the intra-component analysis.

- $\mathbb{W} = \{C_{\texttt{main}}\}$

  The `main` component is reanalyzed by the intra-component analysis. Since the return value of the call to `greet` is now known, the return value of $C_{\texttt{main}}$ can now be updated with the same return value of $C_{\texttt{greet}}$. If the analysis and its global state and results are no longer needed, the programmer can call `emptyAnalysisMemory`, which frees the remaining allocated memory for that analysis.

4.3 Implementation

In this section, the concrete implementation for the above explained approach to manage the memory of lattice elements is described. Some technical parts of the code are abstracted away, in order to make the explanation of the code in this dissertation comprehensible.

In our implementation, elements of `StringLattice` and `SymbolLattice` are allocatable. The other elements are implemented as literal C types, such as `CInt`, and the memory of which could be managed using the Scala Native memory manager with minimal overhead. The implementation of the domain-specific memory manager could be further expanded to other lattices and their elements if needed.

4.3.1 NativeString

Below is the memory management part of the `NativeString` implementation. `NativeString` has `increaseReferenceCounter` and `DecreaseReferenceCounter` methods. The definition of these methods can be found on lines 3 until 11.

The `increaseReferenceCounter` and `DecreaseReferenceCounter` methods respectively increment and decrement the reference counter of a `NativeString` object, if it is not a top or bottom element, since those are treated differently, as explained in Section 3.3. The `isGarbage` method indicates whether a `NativeString` object is garbage, i.e., whether its reference counter is equal to zero.

The `makeGarbage` method sets the reference counter of a `NativeString` at zero in order for the element to be considered garbage. This method will be used when the remaining memory for the analysis is emptied, and the analysis is no longer needed.

Deallocating a `NativeString` requires deallocating the underlying character sequence, as well as the surrounding `struct` that is explained in Section 3.3.

```
1 class NativeString(val underlying: S) extends AnyVal:
2     ...
3     def increaseReferenceCounter() =
4         if(!(eq(top) || eq(bottom))) then
5             underlying._3 = (underlying._3 + 1).toByte
6
```

```
7       def decreaseReferenceCounter() =
8           val ctr = underlying._3
9           if(ctr > 0) then
10              if(!(eq(top) || eq(bottom))) then
11                  underlying._3 = (ctr − 1).toByte
12
13      def isGarbage: Boolean =
14          underlying._3 == 0
15
16      def makeGarbage(): Unit =
17          if(!(eq(top) || eq(bottom))) then
18              underlying._3 = 0.toByte
19
20      def deallocate(): Unit =
21          if(!(eq(top) || eq(bottom))) then
22              free(underlying._2.asInstanceOf[Ptr[Byte]])
23              free(underlying.asInstanceOf[Ptr[Byte]])
```

For the implementation of the domain-specific memory management, we do not use a dictionary to store and update the reference counters. This would not be very efficient, since the program needs to search for the element and its reference counter in the dictionary first. To this end, we store the reference counter of a given element inside the `struct` that contains the pointer to that element. This way, it is not needed to search for the counter before increasing or decreasing it. A `HashSet` is used to store all the allocated elements. This makes it easier to retrieve the allocated elements in order to deallocate them when needed.

Initializing the memory storage for the `NativeString` objects in our implementation just empties `allocatedStrings`. In implementations where that storage is manually managed, initializing the memory requires allocating memory for the storage.

The `gc` method frees all the `NativeString` elements that are no longer needed, and removes them from `allocatedStrings`. `deallocateAllStrings` frees all the allocated strings.

```
1   object NativeString:
2       ...
3       var allocatedStrings : mutable.HashSet[NativeString] = mutable.HashSet.empty
4
5       def initializeMemory(): Unit =
6           allocatedStrings = mutable.HashSet.empty
7
8       def gc(): Unit =
9           val garbage = allocatedStrings.filter(_.isGarbage)
10          garbage.foreach(s =>
11              s.deallocate()
12              allocatedStrings −= s)
13
14
15      def deallocateAllStrings(): Unit =
16          allocatedStrings.foreach(_.deallocate())
```

4.3.2 The GC and IntraGC Trait

The `GC` trait defines the main memory management algorithm using the template method pattern. It offers an interface of abstract methods that will be used for the memory management, as well

as public methods that can be used by the programmer to empty the analysis memory once the analysis is no longer needed. Implementing a memory manager requires extending this trait and implementing the abstract methods. The methods that are left abstract depend on the concrete implementation and representation of the lattice elements.

The implementation directly follows the pseudocode of the algorithm. Indeed, we can see in `updateAddr`, which updates the value of an address in the global analysis state, that if the old value does not equal the new value, then its reference counter is decremented, and the address is updated with the new value. This happens in lines 18 to 22.

```
1  trait GC[Expr <: Expression] extends ModAnalysis[Expr] with SequentialWorklistAlgorithm[Expr]
       with GlobalStore[Expr] with ReturnValue[Expr] { inter =>
2      override def run(timeout: Timeout.T): Unit =
3          initializeMemory()
4          super.run(timeout)
5
6      def initializeMemory(): Unit
7      def emptyAnalysisMemory(): Unit
8      def increaseReferenceCounter(value: Value): Unit
9      def decreaseReferenceCounter(value: Value): Unit
10     def gc(): Unit
11
12     def writeAddr(addr: Addr, value: Value): Boolean =
13         updateAddr(inter.store, addr, value)
14             .map(updated => inter.store = updated)
15             .isDefined
16
17     override def updateAddr(store: Map[Addr, Value], addr: Addr, value: Value): Option[Map[Addr,
           Value]] =
18         store.get(addr) match
19             case None if lattice.isBottom(value) => None
20             case None => Some(store + (addr -> value))
21             case Some(oldValue) =>
22                 val newValue = lattice.join(oldValue, value)
23                 if newValue == oldValue then
24                     None
25                 else
26                     decreaseReferenceCounter(oldValue)
27                     Some(store + (addr -> newValue))
```

In `IntraGC`, `doWrite` updates the value of a dependency in the global analysis state when the local changes of an intra-component analysis need to be committed to the global analysis state. If the value of the dependency is updated, which means that this dependency was not yet known in the global analysis state, or the old value of that dependency did not equal the new value, then `true` is returned, which indicates that the global analysis state is updated. This means that the reference counter for that value needs to be incremented, which happens at lines 8 and 9.

Once the local changes are committed to the global analysis state, the garbage values can be deallocated, which happens by calling `gc` in line 19.

```
1      trait IntraGC extends GlobalStoreIntra with ReturnResultIntra {
2          intra =>
3
4          override def doWrite(dep: Dependency): Boolean =
5              dep match
```

```
6                   case AddrDependency(addr) =>
7                       val value = intra.store(addr)
8                       val isGlobalStateUpdated = inter.writeAddr(addr, value)
9                       if isGlobalStateUpdated then
10                          increaseReferenceCounter(value)
11                      isGlobalStateUpdated
12                  case _ => super.doWrite(dep)
13
14
15          override def commit(): Unit =
16              R.foreach(inter.register(component, _))
17              W.foreach(dep => if doWrite(dep) then inter.trigger(dep))
18              C.foreach(inter.spawn(_, component))
19              gc()
20      }
```

4.3.3 The NativeGC Trait

The `NativeGC` trait provides a concrete implementation for the memory management based on the `GC` trait that was explained in Section 4.3.2. It implements the abstract methods of that trait using the implementation of the `NativeString` that is described in Section 4.3.1. This trait is mainly technical glue code between `GC` and `NativeString`. Therefore, it will not be explained in detail in this dissertation.

Chapter 5

Implemenation Validation

5.1 Testing the Native Lattice Implementation

It is important to test the correctness of the Native Lattice implementation. MAF comes with a set of generator-driven, property-based tests for the core lattices (called `LatticeTest`). These tests check several properties for each lattice for at least 100 different instances that get injected into the lattice. For example, a test for the `BoolLattice` checks whether injecting an element to the `BoolLattice` preserves the truthiness of that element.

We extended these tests in order to test the Native Lattice implementation. Therefore, we implemented generators that generate elements for each lattice according to the Native Lattice implementation. When testing strings, the generator generates Scala strings, and injects them to the `StringLattice` implementation in the Native Lattice. An important difference between Scala strings, and null-terminated strings in C, however, is that the latter uses the ASCII character set, while the former uses the Unicode character set. This means that the `StringLattice` is limited to only the ASCII character set. Therefore, the string generator in the tests is also modified to only generate strings from the ASCII character set.

Additionally, we implemented unit tests written in order to test the correctness of the `NativeString` implementation. These tests check whether the `NativeString` methods produce correct results. For example, appending an empty `NativeString` to a non-empty `NativeString`, checking for referential or string equality between different `NativeString` instances and the other `NativeString` methods are tested.

Additional unit tests were added in order to test the correctness of the C implementation of the operations of `BoolLattice` that is explained in Section 3.1 in isolation.

Finally, in order to check whether analyses that use the Native Lattice implementation produce correct results, additional tests were written that compare the global analysis state of analyses that use the Native Lattice implementation with an analysis that uses the constant propagation domain implementation. The results of analyses that use the alternative implementations that are described in Section 3.6 are similarly tested.

5.2 Testing and Debugging Memory Management

Detecting and diagnosing memory-related bugs can be challenging due to their elusive nature. Therefore, additional debugging infrastructure was implemented in order to debug the memory manager. `DebuggableNativeString` provides the same implementation of `NativeString`, aug-

mented with additional code in order to debug memory-related issues. For instance, the `free` and `malloc` functions are replaced with functions that collect data when memory for strings is allocated and freed, in order to detect memory leaks. These data are counters that are used to detect whether there are more `free` calls than `malloc` calls, and objects that were previously freed which could be used to detect whether an object is freed again. The use of `DebuggableNativeString` has greatly enhanced the debugging process for memory-related issues, resulting in the detection of errors that were previously undetected.

Additionally, in order to ensure the effectiveness of the domain-specific memory management, it is necessary to check that the domain-specific memory management does not leave any garbage values deallocated. Therefore, additional tests were written to test whether there are no more remaining allocated strings left than the ones that are present in the global analysis state. Indeed, if there are more strings left, this would indicate that there are garbage values left deallocated. There could be fewer strings left, since the same `NativeString` instance could be used for different parts of the global analysis state.

Finally, it is also crucial to test whether analyses that use the Native Lattice and the accompanying memory manager produce the correct result. In order to do so, an analysis that uses the constant propagation implementation for lattices is compared with an analysis that uses the Native Lattice implementation. Both analyses are used to analyze several non-trivial Scheme programs. At the end of the analysis of each program, the global analysis state of the analysis that uses the constant propagation implementation is compared with the global analysis state of the analysis that uses the Native Lattice implementation. If the values at addresses of the global analysis state of the latter analysis correspond to values at addresses of the former analysis, then the analysis has indeed produced correct results.

Chapter 6

Performance Evaluation

In this chapter, we empirically evaluate the performance of the Native Lattice implementation and the accompanying memory manager. Therefore, we measure the analysis time and compare the analysis times when different settings are used for the analysis. The various analyses configurations are listed and explained in Table 6.1. The deployment target is the environment in which the analysis is benchmarked. JVM indicates that the analysis is benchmarked while running on the JVM, while Native indicates that the analysis is benchmarked while running natively on the host machine.

Benchmarking `JVMCP` helps us estimate how much performance gain or loss was acquired when running analyses natively on the host machine. Benchmarking `NativeCP` and comparing with `JVMCP` helps us compare the performance of analyses under the same settings, but compiled to different targets. Both `JVMCP` and `NativeCP` analyses have identical settings. As for `NativeS`, benchmarking this analysis and comparing it with analyses that use the `NativeString` give us some additional insights on whether using a domain-specific memory manager and a low-level implementation of string elements in analyses helps improve the performance.

Finally, benchmarking `NativeGCB`, `NativeSB` and `NonClassNativeGCB` help us see whether the alternative implementations for the different lattices are any better.

6.1 Implementation Details

In order to perform the analyses and calculate the results, we implement a benchmarking infrastructure. The infrastructure is identical for both the JVM program and the Scala Native program. The only difference is that the Scala Native program has more configurations for which the performance needs to be evaluated.

Several Scala and Java libraries are not supported by Scala Native. Frameworks such as Akka, and several parts of the standard Java Library, such as `Java.util.Calendar` are not supported. Therefore, the code base of MAF had to be adjusted in order to support Scala Native. Parts of the code that have conflicts with Scala Native were removed if they are not needed for the work in this dissertation. If some conflicting code is needed, then an equivalent substitute for that code is implemented.

When benchmarking Scala programs that run on the JVM, *sbt-assembly*[1] is used to build a fat jar that can run on the Java Virtual Machine. For programs that are compiled using Scala Native and Clang, an executable is produced that can run natively on the machine.

[1] `https://github.com/sbt/sbt-assembly`

Analysis name	Deployment target	Lattice implementation used
`JVMCP`	JVM	Constant propagation implementation for lattices, as explained in Section 2.3.8.
`NativeCP`	Native	Constant propagation implementation for lattices, as explained in Section 2.3.8.
`NativeGC`	Native	The Native Lattice as explained from Section 3.1 through 3.5, and the memory manager, which is explained in Chapter 4.
`NativeS`	Native	The Native Lattice with the alternative implementation for `StringLattice` and `SymbolLattice` that uses Scala strings, as explained in Section 3.6.1.1.
`NativeSB`	Native	The Native Lattice, the alternative implementation for `StringLattice` and `SymbolLattice` that uses Scala strings, as explained in Section 3.6.1.1, and the alternative implementation for `BoolLattice`, as explained in Section 3.6.2.
`NativeGCB`	Native	The Native Lattice, the memory manager and the alternative implementation for `BoolLattice`, as explained in Section 3.6.2.
`NonClassNativeGCB`	Native	The Native Lattice, the alternative implementation for `StringLattice` and `SymbolLattice` that do not use `AnyVal`, as explained in Section 3.6.1.2, and the alternative implementation for `BoolLattice`, as explained in Section 3.6.2.

Table 6.1: The benchmarked configurations for the `ModF` analysis.

In order to ensure that all compiler optimisations are effectively utilized when compiling MAF using Scala Native, the project settings were modified. Specifically, the optimisation mode was set to `release-full`, and the link time optimisation mode was set to `thin`.

The heap size settings of Scala Native were used to set the maximal heap size used by the garbage collector of Scala Native to the same maximal heap size used by the Java Virtual Machine. In order to make sure that the heap size settings are correctly set, we also extended Scala Native in a public pull request in order to expose the heap size information[2]. On the Java Virtual Machine, this information could be retrieved using `System.getRuntime().MaxMemory` from the Java Standard Library. These parts of the Java Standard Library are not yet implemented in Scala Native. Our extension for Scala Native can now also be used in order to implement these missing functionalities.

6.2 Benchmarking Methodology

All benchmarks were executed on a machine with the following specifications:

- Hardware:
  - **Manufacturer and model:** Lenovo Yoga 7i 16iah7

[2] `https://github.com/scala-native/scala-native/pull/3275`

  - **Year of manufacture:** 2022
  - **CPU:** Intel® Core™ i7-12700H Processor (14 cores @ 2.30GHz that include 6 performance cores up to 4.70GHz boost and 8 efficiency cores up to 3.50 GHz boost, 20 threads).
  - **RAM:** 32 GB LPDDR5-4800
  - **SSD:** 1 TB SSD M.2 2242 PCIe 4.0x4 NVMe
- Software:
  - **Operating System:** Ubuntu 22.04.2 LTS
  - **Performance mode:** Performance
  - **JVM version:** openjdk version 11.0.18 (Java 11)
  - **Clang version:** 14.0.0-1ubuntu1
  - **Scala version:** 3.2.1
  - **Scala Native version:** 0.4.12

The heap sizes for both the JVM and the garbage collector of Scala Native are set at 8 GB. Each program is analyzed 200 times, preceded by 50 warm-up runs for each program, allowing the JVM to perform optimizations on the code. The 200 runs are cumulated into groups of 20 runs. The reported results are the average of the cumulative runs for each program. The 95% confidence interval is reported as well. The average of the 20 cumulative runs is referred to in this dissertation as *cumulative analysis time*.

To ensure that the benchmarking of previous analyses does not interfere with the accuracy of the results of the current analysis, each execution of the benchmarking infrastructure is dedicated to benchmarking a single analysis.

We benchmark our configurations by analyzing several non-trivial Scheme programs. Concretely, these programs are listed in Table 6.2, and are included in the code base of MAF. The lines of code that each program contains are also reported in Table 6.2. These lines of code are calculated using *cloc* [3], and they do not include the comments nor the blank lines.

File name	LOC
icp/icp_3_leval.scm	334
icp/icp_1c_prime-sum-pair.scm	394
icp/icp_1c_ambeval.scm	379
icp/icp_8_compiler.scm	466
icp/icp_2_aeval.scm	300
icp/icp_5_regsim.scm	396
icp/icp_1c_ontleed.scm	412
icp/icp_1c_multiple-dwelling.scm	402
icp/icp_7_eceval.scm	774
scm/peval.scm	497
scm/sboyer.scm	632
scm/scheme.scm	767

Table 6.2: The programs that are used for benchmarking `ModF` analyses.

[3] `https://github.com/AlDanial/cloc`

6.3 Results

In this section, we present the benchmarking results in bar diagrams. The results are divided into three groups according to the analysis time duration.

Figure 6.1 depicts the analyses with a cumulative analysis time lower than 30000 ms. We can see that all native analyses perform better than the JVM. The `NativeGCB` analysis seems to finish analyzing the programs the fastest. In Figure 6.2, programs are shown for which the cumulative analysis time of the analyses is between 30000 ms and 100000 ms. For these programs, we notice that the performance of all native analyses is similar to `JVMCP`.

Finally, Figure 6.3 shows the analyses with a cumulative analysis time higher than 100000 ms. Interestingly, we can see that `JVMCP` outperforms all native lattices by a big margin when the cumulative analysis time is high. This is likely due to the various optimisations that the JVM performs, given that it utilizes a just-in-time compiler. Therefore, the longer an analysis runs, the more hot code paths are detected and optimized, resulting in better performance. This is not the case in the native analysis, given that all code is compiled ahead of time, and no optimisations on any parts of the compiled code are performed during the execution of these analyses. We can also observe that the confidence interval for `JVMCP` is larger than the other analyses, which indicates more variation in analysis time. This is possibly for the same reason. Further empirical investigations a required to validate that code optimisations of `JVMCP` are the primary factor for the differences in longer analysis times.

6.4 Comparing Relative Analysis Times

In this section, we compare the analysis times of different lattice implementations in relation to one another. This enables us to assess the performance of the alternative implementations for the lattices when employed in the analyses.

The results presented in this section are the relative analysis time. This is computed by dividing the mean of the cumulative analysis time of a given analysis A by the mean of the cumulative analysis time of another analysis B that A is being compared to. Lower relative analysis times mean that the analysis A performed better in the experiments.

Figure 6.4 depicts the analysis time of `NativeGCB` and `NativeSB` relative to the analysis time of respectively `NativeGC` and `NativeS`. These results help us assess whether the alternative implementation for `BoolLattice`, as described in Section 3.6.2, performs better than the implementation of `BoolLattice` that uses bitwise operations, which is described in Section 3.1.

As we can see from the results, analyses that use the alternative implementation for `BoolLattice` perform better than analyses that use the `BoolLattice` implementation with bitwise operations in most cases. These results were to be expected, given that additional code was required for some lattice operations in order to convert the results from the bitwise operations to Scala booleans, which might have added an additional overhead to that implementation.

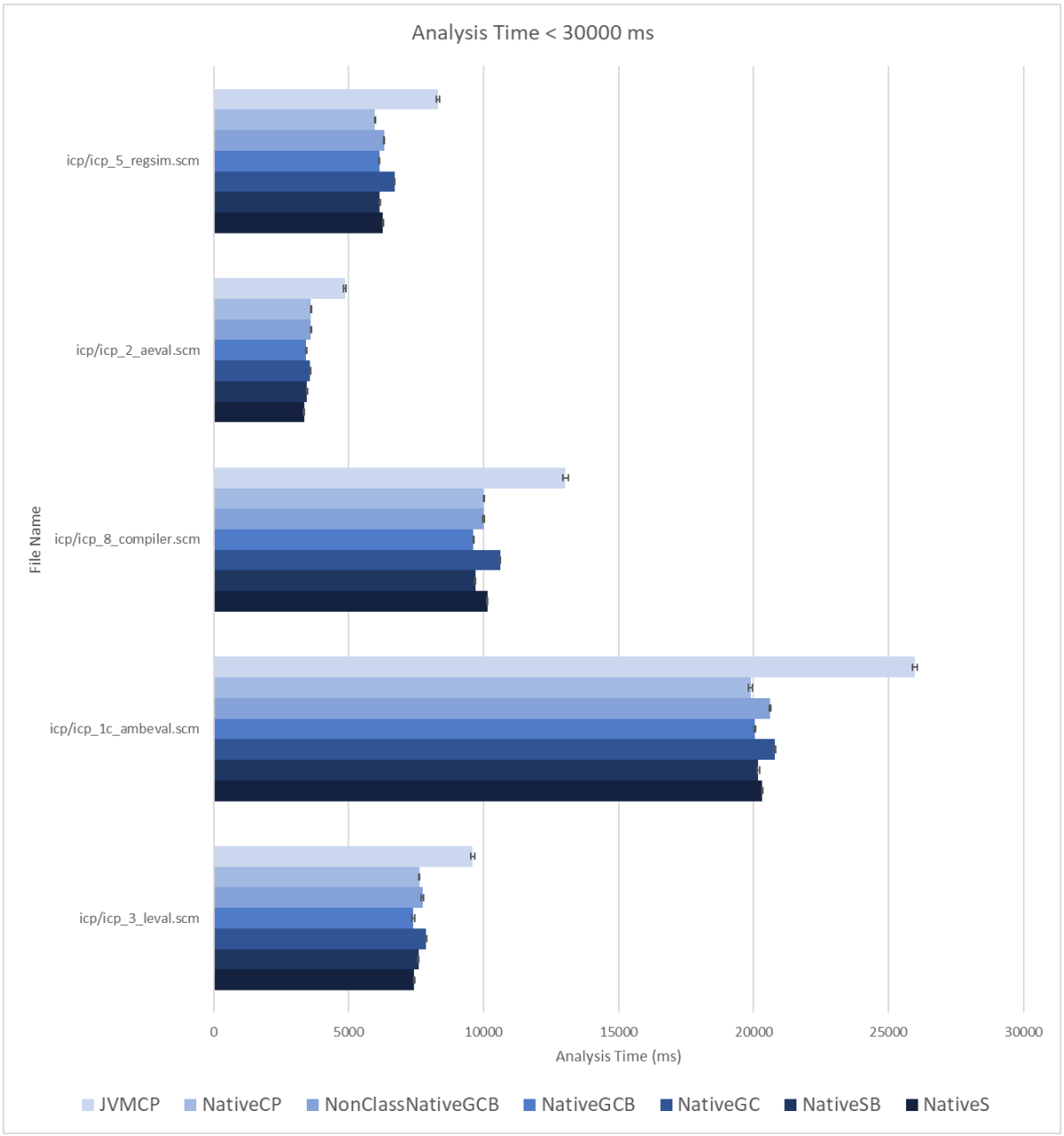

Figure 6.1: Analyses with cumulative analysis time lower than 30000 ms.

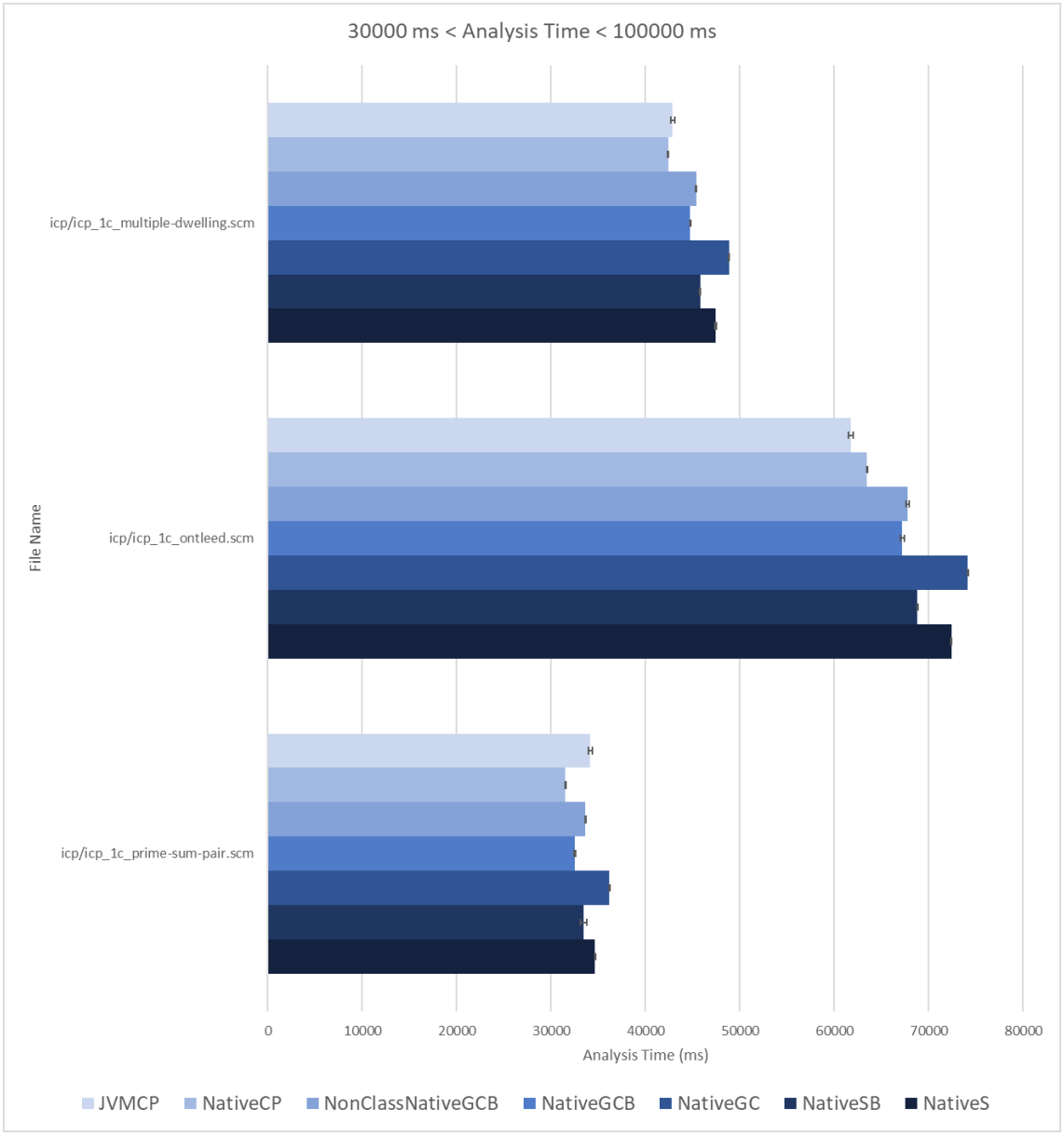

Figure 6.2: Analyses with cumulative analysis time between 30000 ms and 100000 ms.

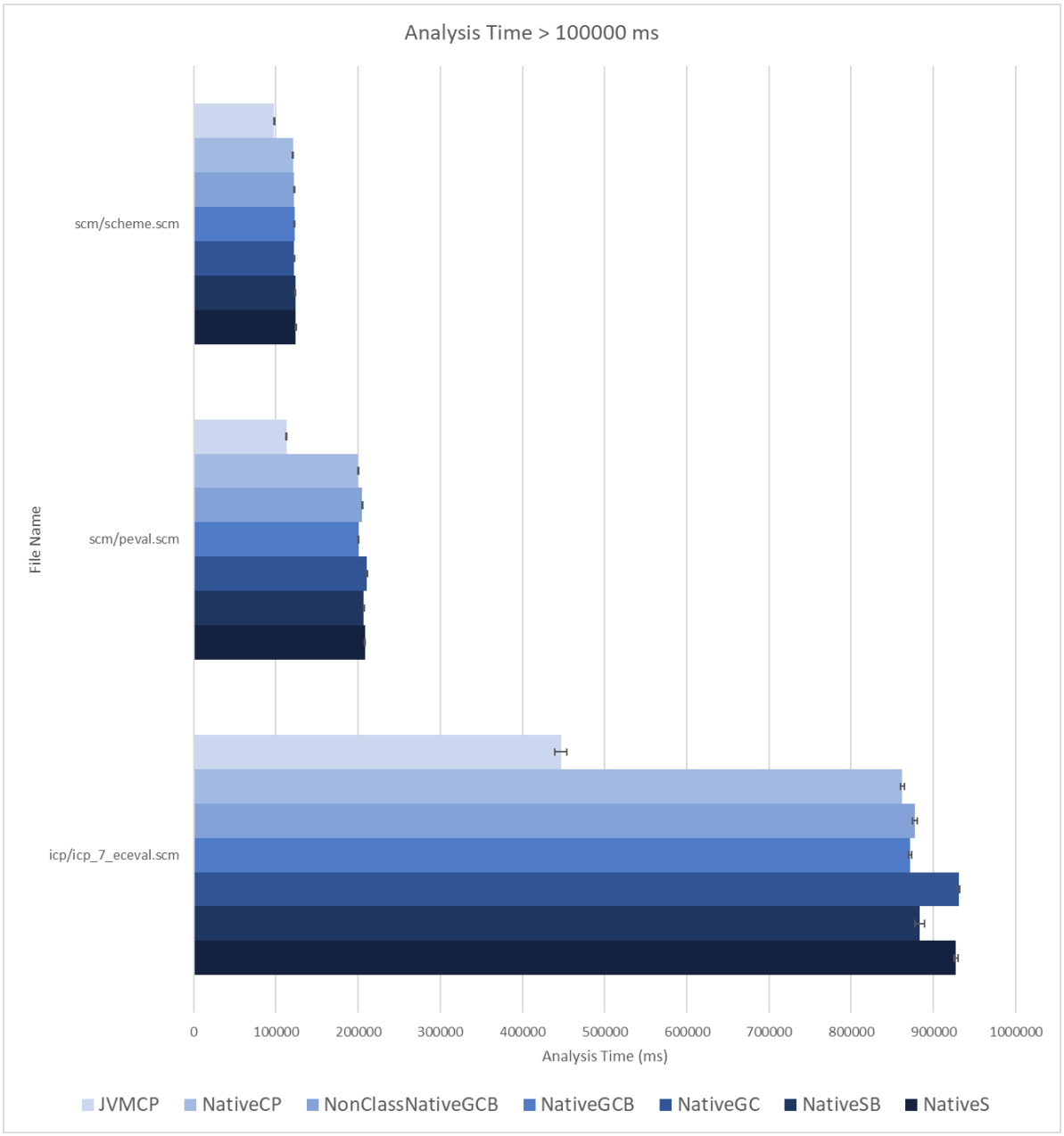

Figure 6.3: Analyses with cumulative analysis time higher than 100000 ms.

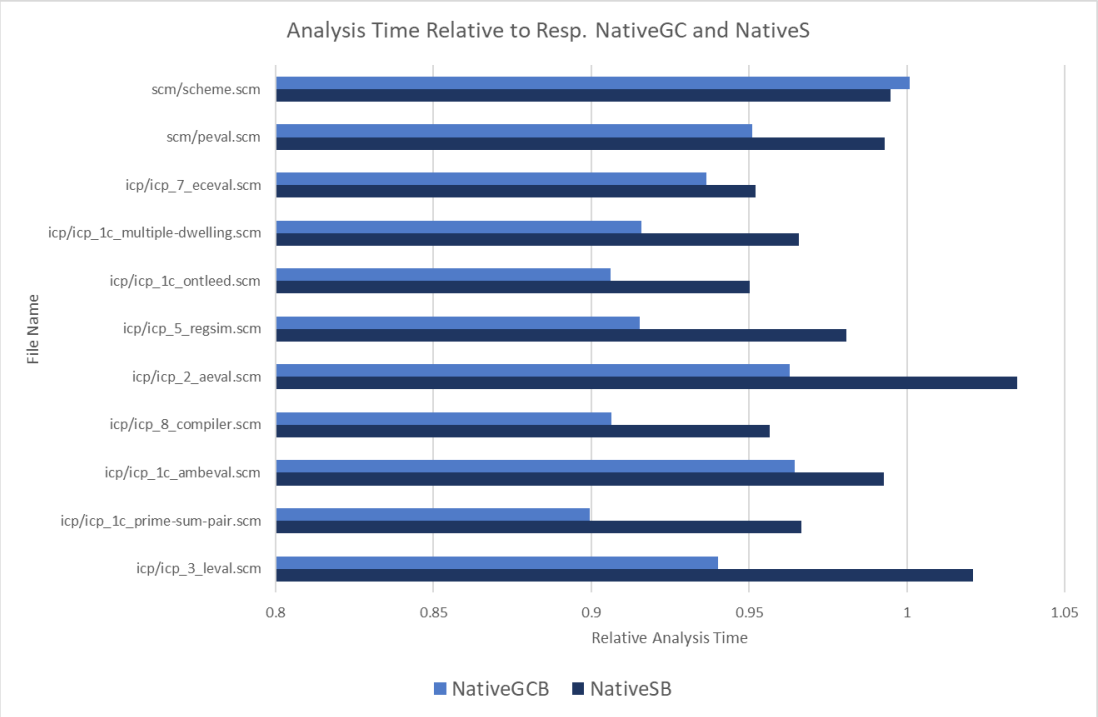

Figure 6.4: Analysis times of `NativeGCB` and `NativeSB` relative to respectively `NativeGC` and `NativeS`.

Figure 6.5 depicts the analysis time of `NonClassNativeGCB` relative to the analysis time of `NativeGCB`. These results help us assess whether using value classes around the underlying representation for strings introduced additional overhead. As we can see, the differences are fairly minimal, and in most cases in favor of `NativeGCB`.

Lastly, we also compare `NativeGCB` and `NativeSB` to `NativeCP`. This comparison helps us assess the performance of the analyses when low-level constructs are used. This comparison is also valuable for evaluating the domain-specific memory manager, given that `NativeSB` relies on the memory manager of Scala Native in order to manage the memory of the string lattice and the symbol lattice elements, while `NativeGCB` uses the domain-specific memory manager which uses analyses properties to manage the memory of the string lattice and the symbol lattice elements.

Figure 6.6 shows the analysis time of `NativeGCB` and `NativeSB` relative to `NativeCP`. In all programs, `NativeGCB` performed better than `NativeSB`. This might be an indication that using a low-level string representation for the `StringLattice` and `SymbolLattice` elements, accompanied by the domain-specific memory manager, improve the analysis performance, given that symbols are frequently used in `Scheme`. We can also observe that, in some programs for which the analysis time is low, the `NativeGCB` analysis performed better than `NativeCP`. However, in most cases, the performance of both analyses is very similar to `NativeCP`. The compiler optimiser of Scala Native performs a substantial amount of optimisations that are intended for the high-level object-oriented and functional features of Scala [13, 7]. Therefore, it is possible that these optimisations have improved the performance of `NativeCP`, which relies entirely on the object-oriented and functional features that Scala provides. Additional experiments are needed in order to identify the reasons behind the performance similarities between the analyses.

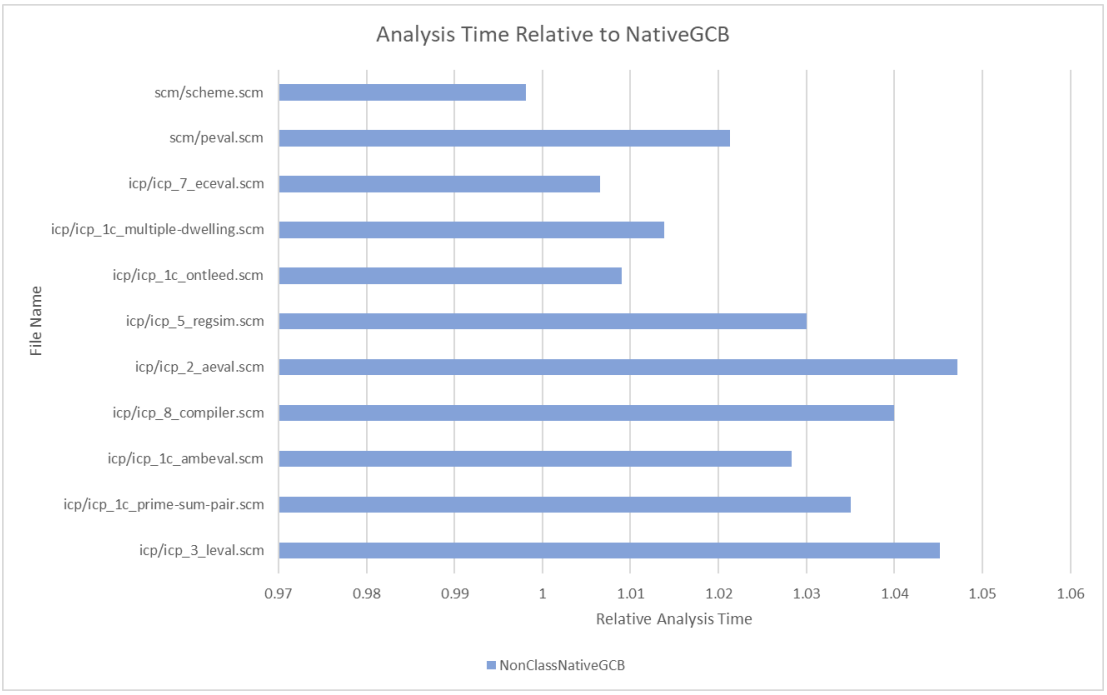

Figure 6.5: Analysis time of `NonClassNativeGCB` relative to `NativeGCB`.

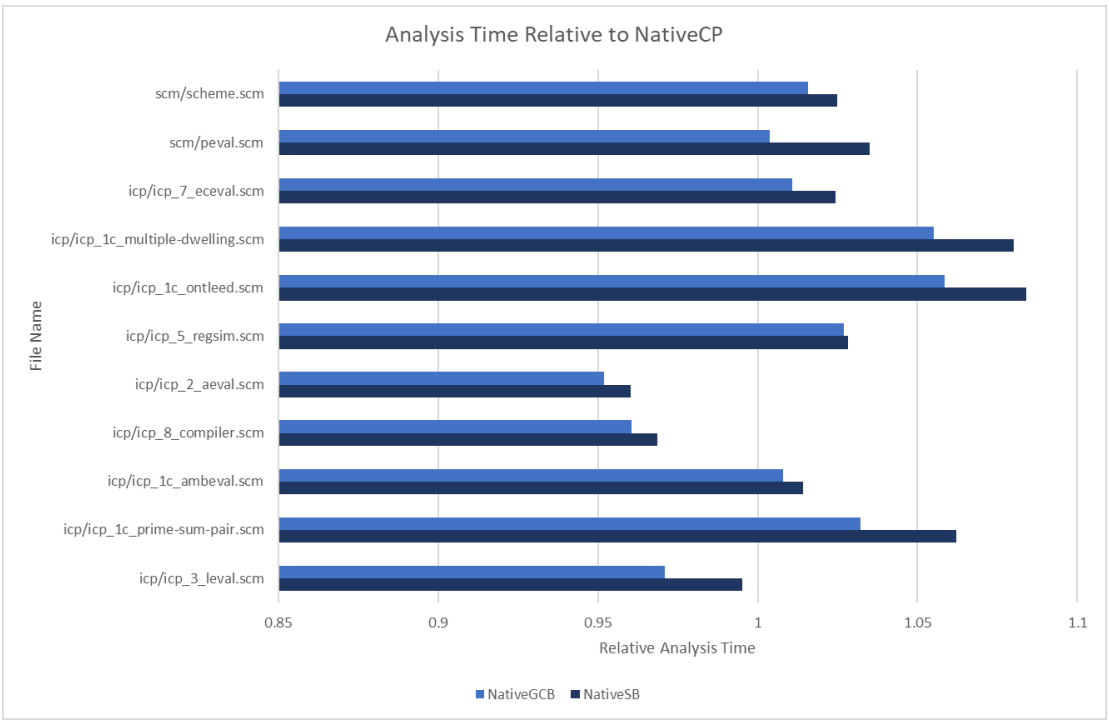

Figure 6.6: Analysis time of `NativeGCB` and `NativeSB` relative to `NativeCP`.

Chapter 7

Conclusion

In this dissertation, we presented a low-level efficient implementation for the constant propagation abstract domain that is used in function-modular analyses for Scheme programs. We presented a domain-specific memory manager, which uses analyses properties in order to manage the memory of some elements of the abstract domain. We demonstrated the correctness of the results that are produced by analyses that use the low-level abstract domain and the domain-specific memory manager.

As we saw in Chapter 6, using a low-level implementation for analyzing programs with a relatively low analysis time improved the performance of analyses. In most cases, running analyses natively on the local machine, whether these analyses use the Native Lattice low-level implementation or the high-level constant propagation domain implementation, closely matches the performance of the Java Virtual Machine whilst still being entirely compiled ahead of time. The Java Virtual Machine performs significantly better when running analyses for programs with a long analysis time, which could be explained by its just-in-time optimisations.

7.1 Future Work

The results of our experiments raised some interesting questions. Analyses that run on the Java Virtual Machine perform better when the analysis time is very high. The low-level implementation of the Native Lattice, augmented with the domain-specific memory manager, closely matches the performance of the high-level constant propagation domain. Therefore, additional experimental studies might be needed in order to identify the main factors that are behind these results. For example, microbenchmarking the various lattice implementations independently and in isolation from the rest of the analysis should provide us with more precise insights. Unfortunately, given the time limitations, it was not feasible to perform these experiments for this dissertation.

Additionally, the presented domain-specific memory manager could be further extended and optimised. An idea worth exploring, for example, might be optimising compound data structures that are used in analyses, such as `cons`-cells and vectors. In this case, it is also a possibility to extend the capabilities of the memory manager in order to also manage the memory of compound data structures. This opens doors for using interesting algorithms, such as an efficient iterative implementation of the Deutsch-Schorr-Waite algorithm.

Bibliography

[1] Stephen Compall. The high cost of anyval subclasses. 2017.

[2] Patrick Cousot and Radhia Cousot. Modular Static Program Analysis. In *International Conference on Compiler Construction*, pages 159–179. Springer, 2002.

[3] Brian A. Davey and Hilary A. Priestley. *Introduction to lattices and order*, chapter Complete Lattices and Galois Connections. Cambridge University Press, Cambridge, 1990.

[4] Brian A. Davey and Hilary A. Priestley. *Introduction to lattices and order*, chapter Ordered Sets. Cambridge University Press, Cambridge, 1990.

[5] Denys Shabalin et al. *Scala Native Documentation*, chapter User's Guide. EPFL, 2022. Release 0.4.9.

[6] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. *The Java Virtual Machine Specification, Java SE 8 Edition*, chapter The Structure of the Java Virtual Machine. Oracle, 2022.

[7] Wojciech Mazur. A Deep Dive into Scala Native Internals. Scala Love conference, 2021.

[8] Anders Møller and Michael I. Schwartzbach. *Static Program Analysis*, chapter Lattice Theory. 2008. Aarhus University, Denmark.

[9] Martin Odersky, Jeff Olson, Paul Phillips, and Joshua Suereth. *SIP-15 - Value classes.* EPFL, 2012.

[10] Henry Gordon Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical society*, 74(2):358–366, 1953.

[11] Xavier Rival and Kwangkeun Yi. *Introduction to Static Analysis: An Abstract Interpretation Perspective*, chapter Pogram Analysis. The MIT Press, 2020.

[12] Abdullah Sabaa Allil, Noah Van Es, Jens Van der Plas, Quentin Stiévenart, and Coen De Roover. Research training: Optimising static program analysis using scala native. 2023.

[13] Denys Shabalin and Martin Odersky. Interflow: Interprocedural Flow-Sensitive Type Inference and Method Duplication. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*, pages 61–71, 2018.

[14] Noah Van Es, Jens Van der Plas, Quentin Stiévenart, and Coen De Roover. MAF: A Framework for Modular Static Analysis of Higher-Order Languages. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 37–42. IEEE, 2020.