UTRECHT UNIVERSITY

MASTER THESIS

Automatic Memory Reuse of Immutable Data in Kotlin

Author:
Gideon OGILVIE

Supervisors:
Dr. Wouter Swierstra
Dr. Alejandro Serrano Mena

December 17, 2024

UTRECHT UNIVERSITY

Abstract

Faculty of Science
Department of Information and Computing Sciences

Master of Science

Automatic Memory Reuse of Immutable Data in Kotlin

by Gideon OGILVIE

Immutable data brings many advantages to programming, including safer concurrency and easier debugging, but it comes with a drawback: immutability means the data cannot be altered. Operations on it require allocating additional memory, but when it is known that the original data will no longer be used, it becomes possible to perform in place updates instead, thus saving memory.

This thesis presents a method to identify such cases by analysing the control flow graph of a program to determine if a variable has a only single usage. The only use of a variable is also guaranteed to be its last usage, which provides a safe approximation for when in place updates can be applied. This approach results in a lookup table that maps each variable to its usage, which provides important information to any program transformation that reduces allocations.

To facilitate these in place updates, this thesis introduces an API-like pattern that allows functions to either allocate new data or update arguments in place, depending on the usage of the relevant argument. This approach is highly extensible, as it can be readily implemented by any library developer, making it a versatile solution for managing immutable data efficiently.

Acknowledgements

I want to start this off by thanking all the people who supported me throughout my studies, especially during my thesis. First of all, my parents and siblings. Without your support I would not have gotten here and I want to sincerely thank you all for it. Second, I want to thank all the people who motivated me to keep going, especially everyone who studied together with me in the library. Without you this project would have either taken twice as long, or I would have never finished it. And last but not least, I want to thank my supervisors. You helped a lot during the process, both on a technical level and a motivational level. You gave great advice, and greatly helped me with the writing.

My thesis would not have been possible without all of you, and I am sincerely thankful to all of you.

Contents

Chapter 1

Introduction

Immutable data brings many advantages in programming, such as creating more predictable behavior, easier sharing of objects between threads or functions, without having to worry about the data changing, easier debugging and reasoning, to name a few. There is however one glaring downside of using immutable data when programming: you cannot change it, so if you do any operation on it, you have to allocate new memory to store this result. For smaller data types this can result in less optimal processor caching, and when using larger data structures, such as lists, this can lead to very high memory usage quickly. Take for example this piece of code:

```
fun foo() {
    val xs = (1 .. 1000000).toList() // create large list
    val ys = xs.map{ it + 1} // increment elements
    val zs = ys.filter{ it % 2 == 0} // filter out all uneven elements
    print(zs)
}
```

If this code is written with immutable data structures, for both the `map` and `filter` operation, new memory has to be allocated for each of the results. If, however, this code was written with mutable data structures, the map and filter function could instead modify the original list. Doing so saves the allocation of two lists, significantly decreasing the memory footprint of the program, without changing its result.

This example clearly demonstrates the overhead that is created when using immutable data, because the original data is kept in memory. This makes sense, because it is not known if the original data might still be needed elsewhere. If, however, the program knows that the data is no longer needed, instead of allocating new memory, the memory of the original list can be reused to store the new list. This allows for the use of immutable data, without the potentially massive overhead it brings.

To be able to do this, two pieces of information are required: is the data involved in this computation no longer needed after, and how can we reuse this memory. Both of these pieces of information are not trivial to acquire, so this thesis will be looking for a way to acquire them both.

The first problem comes down to figuring out when the last use of the data is. If we know that this computation is the last time the data will be used, we know that we can safely reuse its memory. The last use of a piece of data can be very complex to figure out, and is usually done with a solution like reference counting. In this thesis, however, we will tackle this problem statically, during the compilation of the program, instead of during the runtime of the program. This means that the last use of a variable cannot always be determined ahead of time, and as such, we will look at a simplified metric: is this variable used at most once. A variable being used at

most once, means that the memory will always be available for reuse after that one use. This is a safe approximation that can be figured out during compile time, while keeping it manageable within the time-frame of a master thesis.

With this information present, it is now known when memory will be freed up at compile time. This memory can then be either freed up, or reused. Devising a strategy to reuse this memory is not a trivial matter. It is sometimes fairly trivial to reuse memory, for example in the case where a data class is passed to a function and a data class of the same type is returned. Because classes do not vary in memory size, we know the memory of the old class is guaranteed to fit the new data class. Determining this in other instances is not easy, so this thesis will be looking into instances where it is possible to figure this out during the compilation of the program.

After finding solutions for the two previous problems, quantifying the effect of those optimizations is important. To do so, this thesis will look at how often different parameters will trigger this optimization. These parameters will vary in how strict the usage analysis is, and under what circumstances the compiler will trigger the optimization. Measuring the potential difference in impact for each of these parameters, will allow for an informed choice of what to implement for these optimizations.

This thesis will tackle these problems within the Kotlin programming language. Kotlin is a functional programming language with built in support for immutable data and records. Kotlin includes other functional staples, such as higher order functions, which will allow for some interesting optimizations. Along with this, it also already has a robust analysis system in place. This analysis system provides the tools needed to implement the proposed analyses and can be easily extended by creating a plugin. These things combined make Kotlin a very good target for this research.

1.1 Research Questions

So in total this thesis explores the three following research questions:

1. How can we statically determine, during compilation, if a piece of immutable data is used at most once? By understanding if a variable is used at most once, we have a safe approximation of when a variable is used for the last time, since if a variable only has one use, this must be its last use. This information can then later be used to determine if memory can be safely reused. We discuss how we can determine this at compile time in Chapter 3.

2. Under what circumstances can memory from immutable data structures be reused safely to store the result of computations using them? After it is identified that memory has the potential to be reused safely, it needs to be understood when it can actually be done. By identifying patterns within programs where this is possible, we can create a method to do this consistently. Our strategy for this can be found in Chapter 4.

3. What impact does reusing memory of immutable data structures have in terms of memory usage? Quantifying the impact of this research is important for understanding if it is worth it to implement an optimization like this in the compiler. The time it might take to develop an error free and sound solution has to be worth it. If the optimization is not triggered often enough, or the gains are too small, it might not be worth it to continue this endeavor. This question is discussed in Chapter 5.

Chapter 2

Literature and background knowledge

In this chapter we introduce Kotlin, along with key concepts on how it is represented internally by the compiler and how its analysis framework functions. Along with this, scientific research relevant to this thesis is discussed.

2.1 Kotlin

Kotlin is a programming language that is designed to be easy to write, work natively on multiple platforms and have features from both object oriented and functional programming paradigms. A good example of the combination of these two paradigms is the way Kotlin implements the `map` function. Map in a purely functional language like Haskell is defined as a recursive function, whereas Kotlin defines it in a functional but mutable approach. Below you can see a simplified implementation of `map`. In the implementation you can see a mutable list is created to store the result, instead of the recursive approach.

```
inline fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R> {
    val destination: MutableList<R> = mutableListOf()
    for (item in this) {
        destination.add(transform(item))
    }
    return destination
}
```

Kotlin's design[8] enables interoperability with existing Java libraries, allowing developers to seamlessly use Java libraries within Kotlin projects and vice versa. This compatibility largely stems from Kotlin's ability to compile to JVM (Java Virtual Machine) code. The JVM is a virtual machine enables the execution of Java bytecode by providing an abstraction of the underlying hardware and operating system. As a result, any program compiled to Java code can run on any platform with a JVM implementation. Because of the widespread adaptation of Java, nearly every platform has a JVM implementation, including Linux, web browsers and even smart fridges. Along with handling compatibility between projects, the JVM also handles the memory management of its programs.

2.1.1 Kotlin's type system

Like many other languages, Kotlin processes data as typed values or objects. These types contain information about what data can be represented in a value or object

and what the expected behavior is. A value can also be empty, which is represented with a special null object. Kotlin allows any type to be designated as either nullable or non-nullable. For non-nullable types, values cannot be null, ensuring that operations on these types avoid runtime errors due to null references.

Kotlin defines a unified supertype for all types, called `kotlin.Any?`, allowing any value to be cast to it. Additionally, Kotlin defines a unified subtype for all types, called `kotlin.Nothing`.

2.1.2 Kotlin as a functional programming language

Kotlin's design enables functional programming, incorporating well-known functional features like higher-order functions, lambda expressions, and immutable data. For example, lists are immutable by default, requiring explicit modification to make them mutable[7]. Along with this, Kotlin still boasts many OOP features, making it a hybrid language. This brings many benefits to the user, but the mixing of imperative and functional programming aspects makes analyzing the language more tricky.

2.1.3 Lambda expressions

Lambda expressions provide a way to represent anonymous functions. In Kotlin, functions can be stored in variables and called as such. Usually function declarations take multiple lines with many declarations and statements, but lambda functions are often neat one-liners that can be passed around easily. The code below shows such an example, which takes two integers as input and returns their sum.

```
val sum: (Int, Int) -> Int = { x: Int, y: Int -> x + y }
```

2.1.4 Higher-order functions

Kotlin enables higher-order functions, which either take functions as arguments or return them. Higher-order functions are useful, because a lot of things that are commonly done can be expressed using them by only altering the function that is used in such an operation. Common examples of higher-order functions are map, filter, and fold.

Map is a high-order function that applies a singular function to every element of a list and can be defined as follows:

```
fun <A, B> map(list: List<A>, transform: (A) -> B): List<B> {
    val result = mutableListOf()
    for (element in list) {
        result.add(transform(element))
    }
    return result
}
```

Filter is a function which name explains the function quite well. It takes a list and filters out elements that do not match the given predicate. It can be defined as follows:

```
fun <A, B> filter(list: List<A>, predicate: (A) -> Boolean): List<A> {
    val result = mutableListOf()
    for (element in list) {
```

```
        if (predicate(element)) result.add(element)
    }
    return result
}
```

There are many other common higher-order functions, but these examples will suffice for now.

2.2 Control Flow Graphs

Analyzing the relationships between expressions and data within a program requires understanding control flow. The primary reasons for control flow analysis in a program is to enable compiler optimizations and to acquire interesting facts about the program, such as unreachable code or unused variables. One very common example of control flow analysis is lifting an expression outside of a loop, to avoid recalculating it every time the loop iterates.

Control-flow analyses for Kotlin rely on a classic model known as the control-flow graph, or CFG. A CFG of a program is a graph that loosely defines all possible execution paths a program's flow can take[1]. In a control flow graph the control flow relationships are expressed in a directed graph[12].

Control flow graphs provide many advantages over other forms of control flow analysis methods, such as:

- **Visualization**: Control flow graphs can be visualized easily, and those visualizations are fairly similar to the original program structure, helping with understanding.
- **Automated analysis**: Because control flow graphs can be easily encoded as structured data, it is possible to create algorithms that automatically analyze the graph to give useful insights or allow for optimizations.
- **Optimizations**: Control flow graphs can be used as a basis for program optimizations. By analyzing the structure of the program, several improvements can be made, such as eliminating redundant computations or reducing the number of branches.
- **Static analysis**: Because a control flow graph is a stateless representation of a program, the analysis can be constantly updated when the program is being written. This feedback can be passed directly back to the programmer, giving them hints about things that are possibly wrong with the code, allowing them to write better code.

2.3 CFGs in Kotlin

CFGs in Kotlin[3] are complex because they must represent every language construct. For this reason, mentioning all language constructs in this thesis is impractical, nor are all of them relevant. It is, however, important to mention some of them as they are essential to this thesis. To illustrate those examples, pieces of code along with their representing CFG will be shown, along with a brief explanation. For clarity, some CFGs have had elements removed or merged. These alterations have no impact on our explanation, as the removed or merged elements bear no significance here.

2.3.1 Variables

Kotlin CFGs generally represent variables in three ways: declaration, assignment, and access. In the following figure you can see a small example function which does all three. The variable declaration is marked in red in the figure. Notably, in code first you declare a variable, and then assign its contents after. In the CFG the right side of the assignment appears first followed by the declared variable. This makes sense, as the right side is evaluated first, before that result is stored in a variable. The same goes for regular assignment, marked in green.

The CFG represents literals directly, with their value shown in literal expression nodes, marked in blue and purple. Yellow indicates variable access. In this instance the variable is accessed for the return value of the function. Interestingly enough there is no explicit return node, but this is instead implied with a direct connection to the function exit.

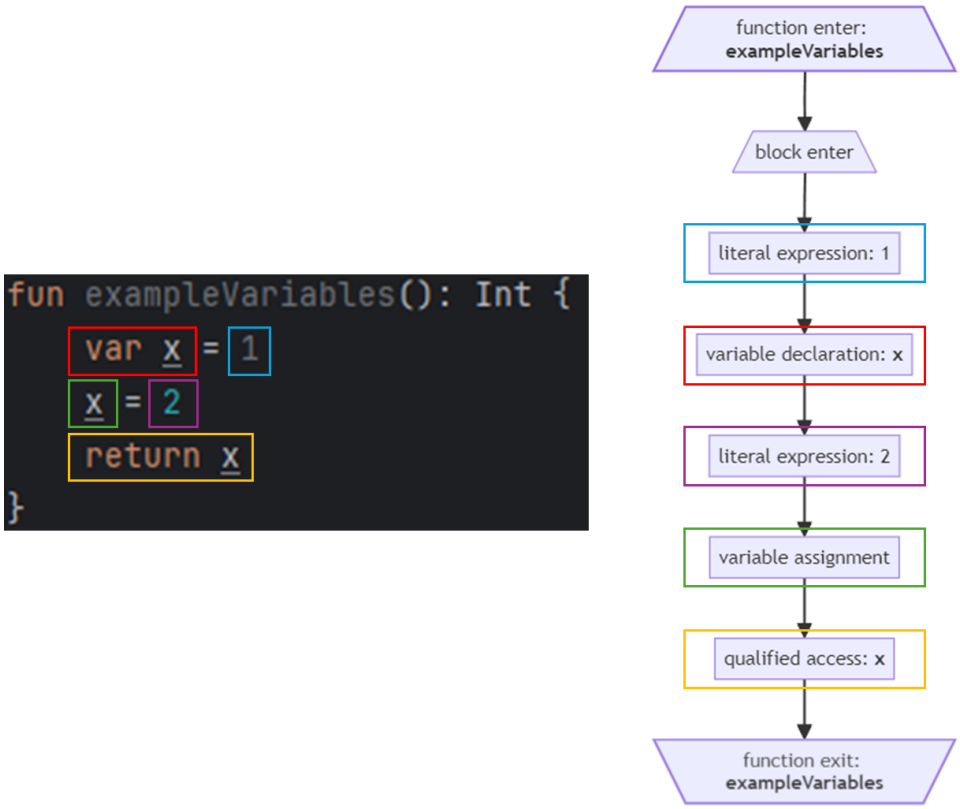

FIGURE 2.1: Variable declaration and assignment in CFGs

2.3.2 Function Arguments

Function calls and their parameters are handled in a specific order to ensure that each argument is evaluated before the function itself is executed. The example below shows a simple function call with two arguments.

As we can see in blue, the arguments are indeed evaluated first in the CFG. In this case the arguments are just two literals, so they are both represented by a single node, but they can be their own expressions with multiple nodes. The start and end of the arguments of function arguments have specific nodes to mark them. These arguments are then passed to the actual function call, marked in red.

```
fun exampleParameters(): List<Int> = listOf(1, 2)
```

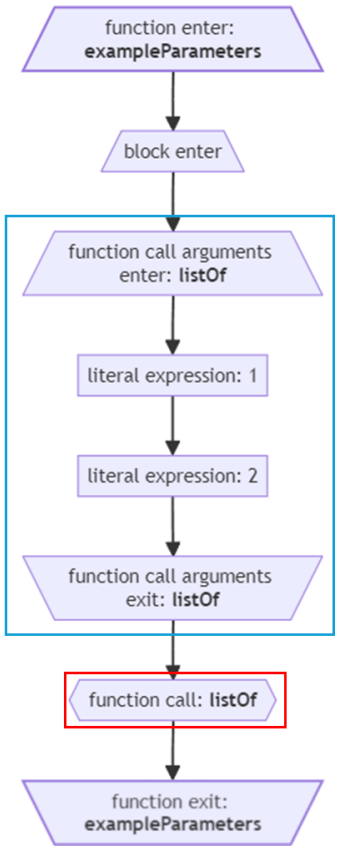

FIGURE 2.2: CFG representation of a function call

2.3.3 Loops

A loop in Kotlin consists of three parts: The loop condition, the loop body and the loop exit. The picture below shows a while loop with its CFG representation, but a for loop looks almost identical. The first part of the loop is the loop condition, marked in red. This part is executed to see if the loop should run or be terminated. If the loop should terminate, the code goes to the loop exit block, after which the CFG can continue with the code after it, in this case a simple function exit.

If the loop should be run again, the loop body is executed, marked in blue. After this the control flows back to the loop condition to see if the loop should terminate. This flow back to an earlier point is done with a special type of edge, called a **back-edge**, marked in green. Back-edges indicate loops within the CFG and are important to this thesis.

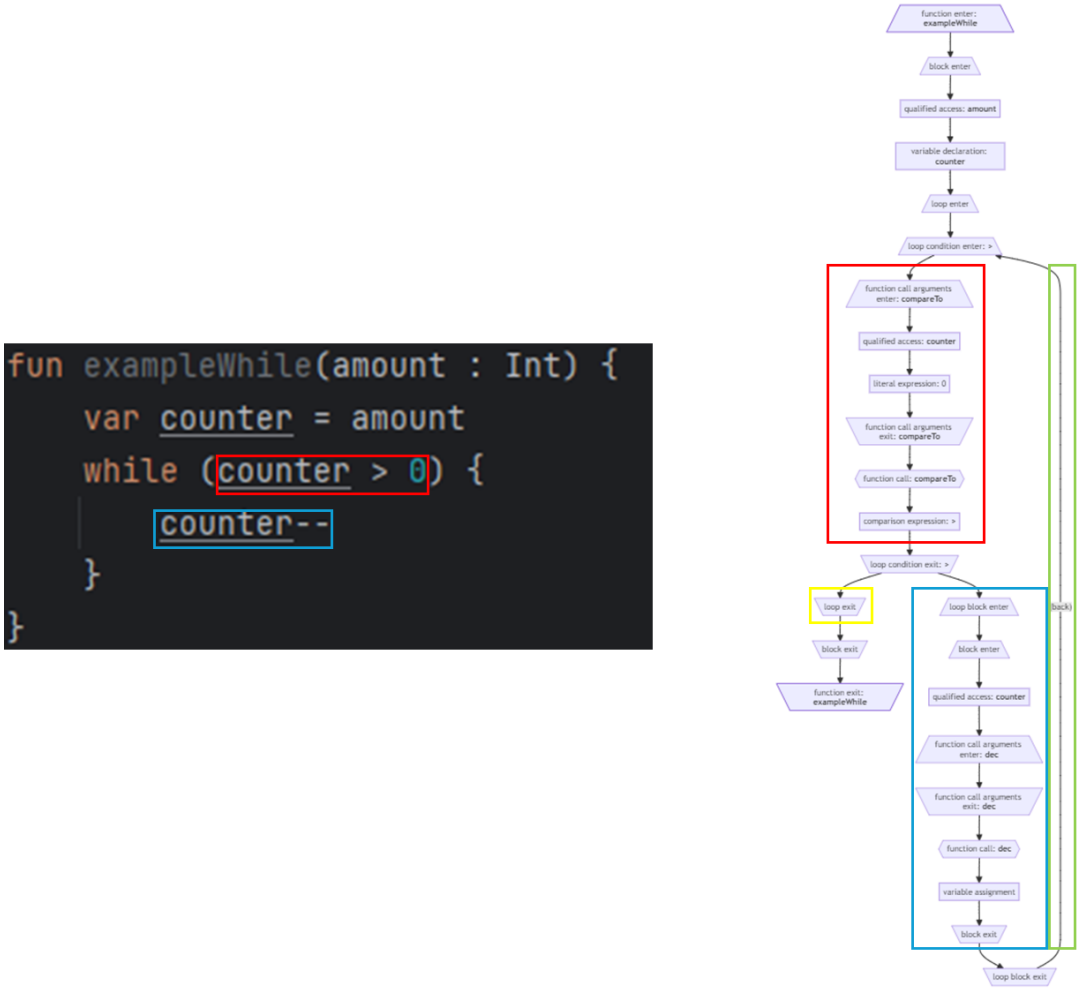

FIGURE 2.3: CFG representation of a while loop

2.3.4 Branching

Branching in Kotlin mainly happens with when expressions, of which if-else expressions are a specialized variant. Below you can see how they are represented in CFGs in Kotlin. They consist of three parts. The first part, marked in red, is the evaluation of the of the branch condition. Since this is an if-else expression, only two possible branches exist. The branch in which case `'decider'` is true, and the one in which it is false. If there were more branches, such as in a normal when expression, this will be desugared to multiple when expressions with two branches in a row. The first branch tests if the value is equal to the branch condition, and the second branch contains all the remaining branches, which will then be branched off one by one in the same way.

In blue is marked the branch for if `'decider'` is true, and in green if false. Those branches are then unified in the 'when exit' node. If the when expression contains more than two branches, they are still all merged into a single node.

```
fun exampleElif(decider: Boolean): Int =
    if (decider) 1 else 2
```

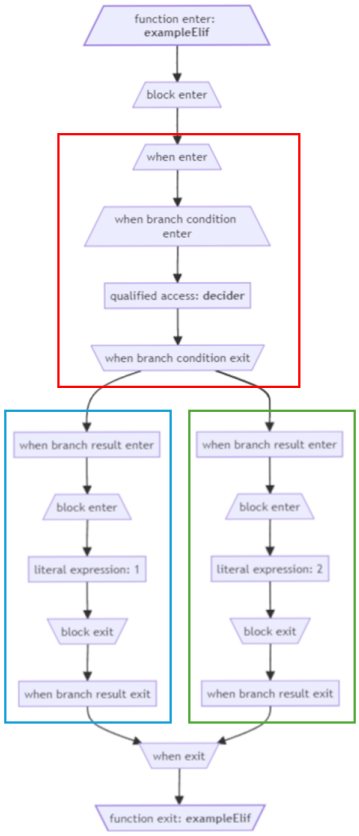

FIGURE 2.4: CFG representation of an if-else expression

Lamba Functions

Lambda functions are a type of function that is usually passed as an argument to a higher-order function. Below is an example of a lambda function that is used as the argument to map. The lambda function in this case just takes the input argument and returns it plus one.

This lambda goes to the map function, along with the argument `'input'`. Though Kotlin usually analyzes each function separately, it makes an exception for lambda functions. They are considered part of the function in which they are created, and are analyzed together with it.

This part of the graph is the part marked in yellow below. It looks just like how a normal function would, with some minor additions. The most important one is the edge that is marked with `(back)`. This back-edge represents that this lambda function can be called potentially more than once. Not all lambda functions have this back-edge, since it is dependent on the contracts of the specific function. Function contracts can be used to specify how often a lambda will be executed. If a contract specifies that the function will potentially more than once, this back-edge will be present. For more information on this, see the section 2.4.2 on function contracts.

```
fun exampleLambda(input : List<Int>) : List<Int> {
    return input.map{it + 1}
}
```

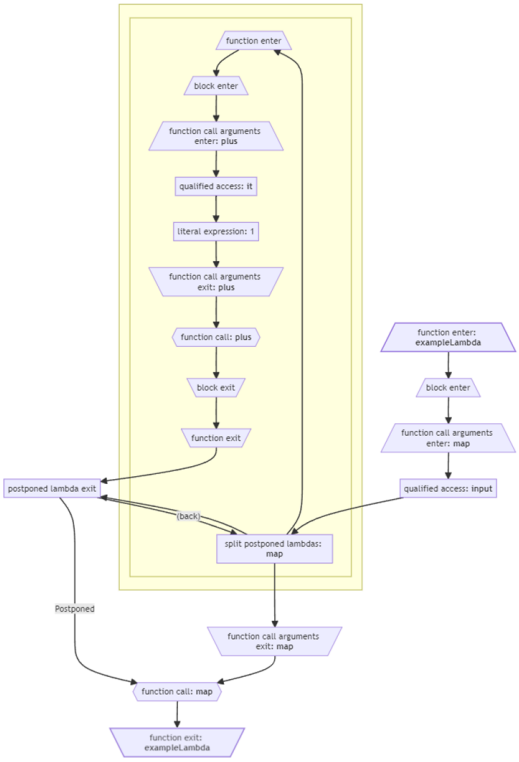

FIGURE 2.5: CFG representation of a lambda function

2.4 Lattice-based Control Flow Analysis

Lattice analysis[10] is a common way to do control flow analysis, by modeling abstract program states as elements of lattices and joining them using lattice operations. The lattices in this context are used as a mathematical model for properties of variables, such as values, definitions or usage.

A lattice is a partially ordered set where each pair of elements has both a greatest lower bound and a least upper bound. These operations, referred to as **join** and **meet**, are used to merge analysis states for different program paths. The join operation combines what is true for each path and merges that information. This represents an information gain, or a union of possibilities. The join is often used to calculate an overestimation, or safe approximation of a fact. The meet operation calculates what is true in each path and is used for underestimations or refinements.

To calculate the abstract state of a CFG node, a transfer function is required. This function calculates the abstract state either directly, or by using the abstract state of other nodes. The change of state is based is based on the nature of the operation. For example, if the node currently being analyzed is an assignment node, this could have a different effect than if it was a variable access node. The transfer functions can be different for each lattice analysis.

Lattice analyses are guaranteed to terminate. Since the lattices are finite, or at least bounded in practice, repeated application of the join or meet operation will eventually reach a fixed point. At this point, the data flow analysis stops and the resulting lattice elements summarize the variable properties at each program point.

Lattice analysis is widely used in in compiler optimization and error detection. Common use cases are constant propagation, live variable analysis and reaching definitions. Since lattice analyses share a common structure, implementing this structure means it is simple to implement any lattice based analysis.

2.4.1 Analyses Kotlin does

Kotlin already performs several types of analyses on the CFG. One that is commonly used in many programming languages is variable initialization analysis. Variable initialization analysis makes sure a variable definitely has a value before its first usage. It operates on abstract values from the assignedness lattice, a flat lattice over the set {Assigned, Unassigned}. The analysis propagates these abstract values forward, using the standard join operation to merge states across different paths.

Along with variable initialization analysis, Kotlin employs many other common data flow analyses, including, but not limited to reachability analysis, constant propagation, and unused variable detection. Many of these algorithms are described in Program Analysis (an Appetizer)[11].

Smart casting analysis

One of Kotlin's more unique analyses is smart cast analysis[15]. Smart casting is a form of flow-sensitive typing, meaning that some expressions may introduce changes to the compile-type of variables. This allows the programmer to avoid unneeded explicit casting of values when their runtime types are guaranteed to conform to the expected compile-time types.

A concrete example of smart casting in action:

```
fun smartCastInInference() {
    val user: User? = getUserById(3)
    if (user == null) return
    // at this point the compiler knows user: User
    println(user.name)
}
```

It achieves this by using a map lattice as its data-flow domain. This analysis maps any expression to a Type x Type sublattice: the first element identifies the type the expression definitely has, and the second specifies a type it definitely does not have.

In the example above, without smart casts, `user` has the type `User?`. However, because of the earlier line `if (user == null) return`, we can infer that `user` is no longer null. Kotlin thus 'smart casts' `user` to type User instead of User?. This allows us to print `user.name` without needing to check if `user` is null. Several methods can introduce smart casts, including conditional expressions (if), cast expressions (as), type-checking expressions (is), and others. Another key mechanism supporting smart casting is function contracts.

2.4.2 Kotlin function contracts

Some standard-library functions in Kotlin are defined to adhere to specific call contracts[16]. These contracts affect how the control flow graph of the calling function is constructed. A function's call contracts consist of one or more effects. There are several kinds of effects:

- Calls-in-place.
- Returns-implies-condition.
- Particular implementations may introduce other types of effects.

These effects alter the call graph. Without a guarantee that the function executes, the call graph cannot represent it as if it will. With that guarantee, corresponding incoming and outgoing edges can be added.

Calls-in-place effect

The calls-in-place effect is used to specify how often a lambda function will be called when it is passed to a function as an argument. These contracts modify the CFG's structure, making them relevant to this research. There are currently three different types of calls-in-place effects:

- At-least-once
- Exactly-once
- At-most-once

If no contract is present for a lambda function passed as a function argument, the corresponding CFG fragment will look as seen in 2.6. Since there are no guarantees about how often the lambda function will be executed, the analysis cannot add any edges going out from the lambda to the rest of the CFG.

If a function is executed at least once, the CFG will be modified to reflect this. This is done by adding an outgoing edge from the lambda to later in the function and by adding a back-edge. How this looks like can be seen in 2.7

If a function is executed exactly once, this is shown by adding an outgoing edge from the lambda to the rest of the function, to signify that control can pass from the lambda directly to there. How this looks like can be seen in 2.8.

Lastly, if a function has an argument saying that the argument is executed at most once, an edge showing that control can skip over the argument to the rest of the function is added, as seen in 2.9.

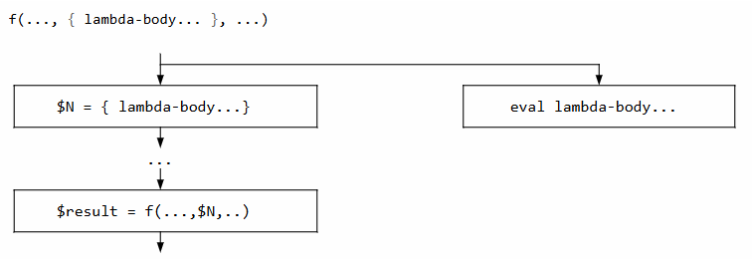

FIGURE 2.6: A lambda argument with no contract

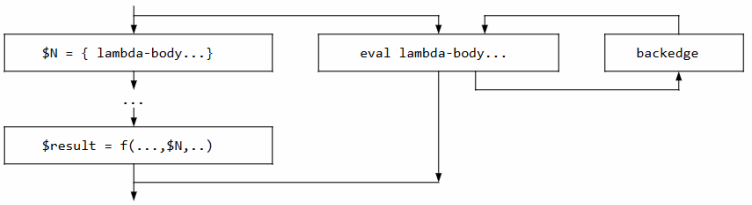

FIGURE 2.7: A lambda argument with an at-least-once contract

Returns-implies-condition effect

The returns-implies-condition effect specifies that when function F is called, that Boolean condition P will be assumed to be true, if the call to F returns. This allows for the addition of an 'assume node' in the control flow graph, which can be used to make deductions about the state of the program and its variables.

An example of this is with the Kotlin smart cast system. Take the following Kotlin code:

```
fun example()
{
        val a : Int? = someFunction() // a functions that returns a nullable integer
        val b = 5
        require(a != null)
        val c = a + b // would be illegal without the contract,
        // as a is a nullable type, and could thus be null
}
```

Here, the compiler automatically deduces that `a` cannot be null when calculating `c`, thanks to the `require` function, which has a returns-implies-condition effect. After the require function, because the value of `a` is statically known to be non-null, it is automatically smart casted to be a non-nullable integer.

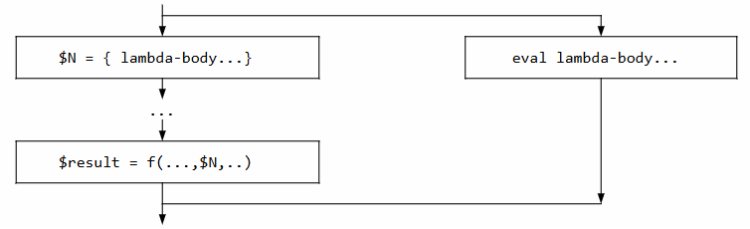

FIGURE 2.8: A lambda argument with an exactly-once contract

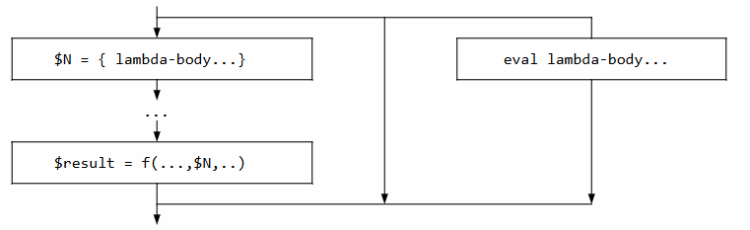

FIGURE 2.9: A lambda argument with an at-most-once contract

2.5 Related works

This section explores three key areas relevant to addressing these challenges: uniqueness analysis, memory management for immutable data structures, and aliasing analysis. It reviews research each domain, highlighting their potential relevance and limitations for Kotlin's multi-paradigm nature. Furthermore, it provides a basis for understanding how existing methods can be extended or adapted to the unique requirements of Kotlin and similar languages.

2.5.1 Uniqueness

Immutable data in programming offers several advantages, such as enabling referential transparency, where the compiler can replace a function call with its return value without affecting the program. A downside of immutable data, however, is updating it. Updating an immutable value requires allocating new memory for the data and storing its reference in a new variable. This often makes programs with immutable data less memory efficient, because chaining operations requires new memory allocations for each operation. While this additional memory generally must be allocated, it is not always the case. When you know that the original memory will not be used again, you can safely store the new data in the old memory, without breaking the immutability of your data elsewhere in the program[18].

Determining whether a value is accessed at most once, however, is not a trivial matter, and a lot of research on how to do it has been put into it. Natural reduction types[20] are a way to tackle this problem. Another type-based approach by Smetsers, J. and Barendsen E.[17], is one that focuses on rewriting the program graph to create more efficient space behavior and on interfacing with non-functional operations, such as IO. Extending the Hindley-Milner type system[18] is another approach that works for higher-order functions and data structures. This analysis can also be enabled to determine when a value is used exactly once, instead of at most once, making it suitable for call-by-value evaluation.

All those papers are, however, based on purely functional programming languages, but our target language, Kotlin, is a multi-paradigm language[19], which

includes, but is not limited to, functional programming. Because of this, it is unsure if those approaches in the previous papers will work. Because of that, approaches for determining uniqueness are also important[4][9].

2.5.2 Memory management for immutable data structures.

Since programs usually only have a limited amount of memory available to them, it is usually a good idea to optimize a program to use as little memory as possible. Since immutable data structures introduce memory overhead, by needing fresh allocations for any operation done on them, optimizing their use can have large benefits for the memory usage of a program.

Reference counting[6] is a way to optimize the memory usage of immutable data structures. It is a runtime solution that works by automatically keeping track how many live references to an object exist. When an object is created, its reference count is set to one and every time a new reference to the object is created, it is incremented by one. When a reference goes out of scope or is explicitly removed, the count is decremented by one. When the count drops to zero, this means that no references to the object remain, and its memory can thus be safely deallocated. Using this greatly decreases the memory usage of immutable data structures, since they are only kept in memory if they might still be used.

Perceus[13] is a fresh look at reference counting in functional programming languages. By combining reference counting with an ownership system, where the ownership of references is passed down into functions, it achieves a memory environment that is garbage-free, meaning that any reference that is retained is live. This reference counting system also includes reuse analysis, which matches the size of constructors and destructors. If they match and the destructed object has a reference count of one, the new object overwrites the old, instead of being allocated new memory.

A common challenge faced reusing memory in functional programming is that it breaks referential transparency. Heap recycling[5] is a method that achieve this under specific circumstances. This paper introduces a lightweight language construct that enables user-directed in place updates. Other methods to achieve in place updates in functional programming languages usually require altering the program style in a significant way, such as Haskell's monads, but heap recycling preserves the declarative and clean nature of functional languages.

2.5.3 Aliasing

Aliasing in computer science happens when two references point to the same memory location. During static analysis of programs, the aliasing of variables can complicate the process. This is because any observation about a variable can also affect any variable that can be aliased to it. To combat this, it is crucial to correctly identify and handle any alias that might occur.

An example of why aliasing can create problems for analysis can be seen below in the C code fragment. Since both 'p' and 'q' point to the same memory location, any judgment made about 'q' also has to be made about 'p'. If aliases are not kept in mind when performing static analysis, variables might have different judgments made about them, despite being the same variable.

```
int a = 5;
int *p = &a, *q = p;
*q = 10;  // Both `p` and `q` alias the same memory location.
```

There are several approaches to finding all the aliases within a program and choosing the most suitable approach for your use case is important. A more complete overview of all aspects can be found in [14]. The first distinction to make when calculating aliases is whether you want the set of potential aliases or the certain aliases. The set of potential aliases are the complete set of every other variable a variable can potentially point to. This is different from certain aliases, which are only aliases that always occur, not only in some of the paths. In our case, since we want to determine the upper bound of the usage of a variable, we want the set of potential aliases of variables. The analysis framework of Kotlin already does an alias analysis, but this analysis calculates the certain aliases.

The second distinction to make is whether you want to do pointer analysis or reference analysis. Pointer analysis focuses on the memory addresses a variable can point to, and is generally more useful for lower level languages, since higher level languages often do not have direct access to pointers. Pointer analysis works directly with abstract addresses and can also include pointer arithmetic, whereas reference analysis abstracts away the memory management and instead focusing on the relation between objects, and the object they point to.

In [10] several methods to calculate pointer aliases are discussed. The two most prominent ones are Steengaard's algorithm and Andersen's algorithm. They both provide a method for calculating the set of potential aliases. Steengaard's algorithm is a unification-based approach, where any assignment of a variable to another merges both sets of aliases they might have. This is a fast, but less accurate method of determining aliases. Andersen's algorithm, on the other hand, is a more precise and computationally intensive approach. It works by tracking inclusion constraints between variables, where the points-to set of one variable can be included in another's. This allows for finer granularity in alias analysis, distinguishing between different potential alias relationships. While Andersen's algorithm provides more accurate results, it comes at the cost of higher computational complexity, making it less suitable for large-scale analysis compared to Steensgaard's algorithm. Both methods are foundational in static analysis of pointer aliasing, with Steensgaard being favored for performance and Andersen for precision.

Lastly, it is important to differentiate between interprocedural and intraprocedural analysis. Intraprocedural analysis provides a more complete picture of all possible aliases, but it is more complicated to calculate. The Kotlin analysis framework itself does not have the built-in ability to do intraprocedural analysis.

Chapter 3

Usage Analysis

One of the criteria we set for reusing memory was that we need to be sure that the memory of a variable will not be used again after altering it. To safely approximate this, we decided to only reuse memory of variables with only one use. To determine which variables can and cannot be reused, we need to be able to analyze our code.

Because it is quite tricky to understand what being used means, we must first clearly define what constitutes as being used. There are different ways to approach this, but the most straight forward way is to simply count any access of a variable as a use. This allows the use of the variable access node of the Kotlin control flow graph to signify usage. Calculating the usage of a variable is not as simple as just counting how often a variable can be found in variable access nodes. This is mainly due to scoping and aliasing. If a variable is called from an outer scope, or a variable is a direct alias of another, we have to keep this in mind when calculating the usage of these variables.

How often a variable is used is represented with a subset lattice over the following set: $\{0, 1, \infty\}$. The elements of this lattice are chosen to be only the numbers zero, one and infinity, because more fine-grained counting complicates this problem a lot while not having the greatest return. The value inf is used to represent any amount of usage larger than once.

Initially, the algorithm assigns each tracked variable a value of zero. Then, if a variable is used, we move to the next right element in the lattice. So a variable with a usage value of zero becomes 1 and 1 becomes inf. This also applies for a variable that is used at most once. If a value is used at most once, that means the value is either zero or one. We can do this, because increasing both those values by one stage gives us once and inf, which is equivalent to the set of at least once.

3.1 The algorithm

The goal of our algorithm is to find out how often a variable is used. To achieve this, the CFG of every function in the program will be analyzed. In this analysis, the declaration of all the variables will be tracked, along with when they are used and in what context. After the algorithm has run, we will have a mapping from every variable in the analyzed code to how often it is used. This will allow us to easily look up how often a variable is used when we later want to transform the code.

Analysis in the Kotlin compiler is intraprocedural, meaning that functions are analyzed separately from another. Because of this, the analysis of every function starts at the root node of a function, and then recurses in a dfs-like manner from child to child. Along with this, there is a requirement to enter a node. This requirement is that all the parent nodes that are connected by a forward edge are already visited.

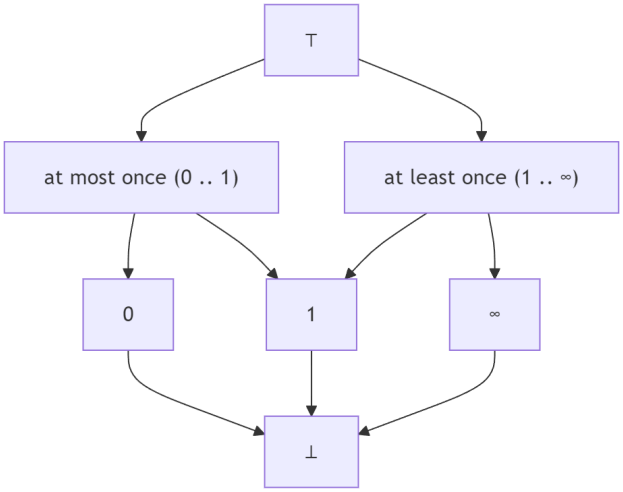

FIGURE 3.1: The lattice for usage

This is to make sure that when a path splits and meets back up again, both paths are already analyzed separately and can be merged correctly.

3.1.1 Simple usage analysis

In simpler cases, the algorithm boils down to how often an access to a variable occurs in the CFG. There are complications with variables from outer scopes and branching, but for this first part we are showing how this works with just a single scope with no branching in the CFG. Figure 3.2 shows such an example, which is produced by the following code:

```
fun exampleUsage() : Int {
    val x = 5
    val y = x + x
    return y
}
```

We start with an analysis domain that contains all the function arguments with an initial usage value of zero. Later we expand this when we encounter variable declaration nodes. When we encounter such a node, we store that this variable is created in the currently active scope, with an initial usage value of zero.

Then, if we encounter a qualified access node for this variable, we increase the usage value of this variable to 'once'. Any subsequent use changes the usage value to 'used more than once,' which remains constant for all further uses.

At the end of the scope, we know that any variable declared within it can no longer be accessed, and thus it can no longer be used. We therefore know that the usage value of any variable created within it is final, so we can store the usage of the variables in a map from variable to usage amount. In our example the final result of the analysis is that `'x'` is used more than once, during the calculation of `'y'`. `'y'` is used only once, as return value for the function.

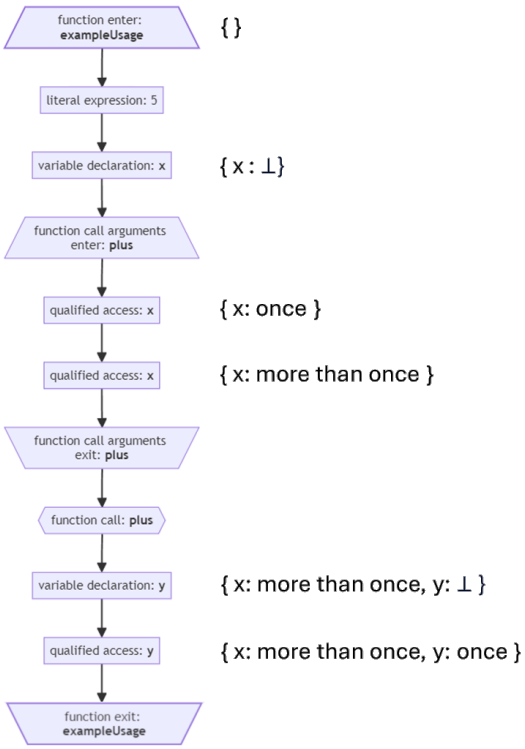

FIGURE 3.2: Usage analysis in the simplest case

3.1.2 Tracking scopes

This algorithm seems straightforward initially, but complications arise when the usage scope of a variable differs from its declaration scope Because of this, it is important to keep track of what scope we are currently in. We do this by creating a recursive data structure to store our usage information. On the first level of the data structure, we store the information for the current scope, i.e. the variables declared in this scope. Then we also store a reference to the information of the scope above it, which in turn holds a reference of the scope above it and so on. A visual example of this can be seen in figure 3.3.

When we enter a new scope, we create a new scope information object and set the previously active scope as its parent. Because Kotlin automatically annotates any node that signifies the start of a new scope with a so called enter node marker, we can very easily find out when a new scope is created. In this example any node that signifies the start of a new scope is marked green.

When a scope is closed, it is also marked with an exit node marker. When such a marker occurs, we simply set the parent scope of the previously active scope as the currently active scope. These nodes are marked red in this example.

3.1.3 Branching

The first complication arises with branching in the CFG. Branches require more complex handling, as a variable may have different usages in each of the branches. To accommodate for this, the algorithm will separate the usage values for each branch. This is done by creating a copy of the active scope (and all the scopes above it) before a branch is entered. Each branch then sets its parent as their respective copy. During the analysis of the branch, any increase in usage for a variable from an outer scope is only updated in this copy. When the branches meet again, the algorithm merges scopes by combining each variable's usage values using a lattice join operation

In the example below, we can see how this works in practice. The variable 'example' is declared in the function scope (marked in blue). It is then only used later in the if branch (marked in orange). In the else branch, it is not used, however. This means that during the analysis of both of those branches, the usage value for 'example' holds two different values. These two branches meet again in the when exit node. At this point, the copies that exists for each branch are merged. The scopes inner scope for each branch is already closed, so they do not have to be merged. Then the information for the when scope from both copies is merged, but since no variable was declared in that scope, nothing changes here. Then the function scope of both copies is merged. Since one copy has the variable 'example' as being used once, and the other has the variable being used zero times, we have to perform a lattice join on those two values. Looking at 3.1, when we merge zero and one, we get a new value of at most once. This means that the analysis from this point onward will continue with a value of 'at most once' for the variable 'example'.

```
fun exampleBranching(decider : Boolean) {
    val example = 10
    if (decider) {
        val x = example
        ...
    }
    else {
        ...
    }
}
```

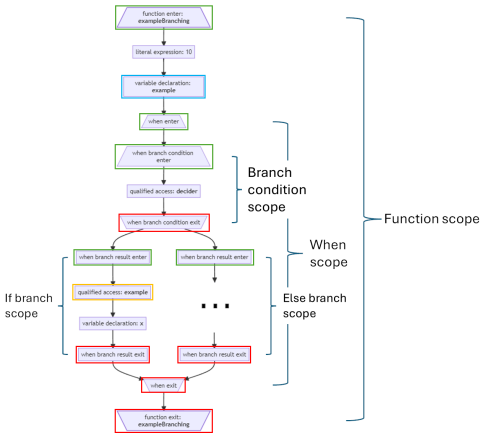

FIGURE 3.3: Scoping of an if-else statement

3.1.4 Loops

The second complication arises when a node containing a variable from an outer scope is executed multiple times, relative to the scope it is declared in. This is because a variable node, from an inner scope, might be executed more than once, relative to the scope it was declared in. Figure 3.4 illustrates this with an example, for which the code is shown below.

```
fun exampleRelativeScopeExecution() {
    var sum = 0
    val base = 5
    while (sum < 100) {
        sum += base
    }
}
```

In this example we can see the variables `'sum'` and `'base'` being declared inside the outer function scope. Each of them has a single access node within the loop block. Because each loop execution uses `'sum'` and `'base'` once, we cannot predict how often these nodes will execute. This is because this loop iterates an unknown amount of times.

To determine if such a thing occurs, we also need to track how often a scope is potentially executed. We can find out how often a scope is executed, by inspecting the incoming edges at the start of the scope. If there are any incoming backwards edges, this means that the scope contains some sort of loop and is thus potentially executed more than once. If no backwards incoming edge exists, we know that the current scope is executed at most once. We will then store if a scope is executed at most once in our scope information object.

To now determine how often a variable from an outer scope is used relative to the scope it was declared in, we can bubble up the scope information structure until we get to the scope a variable was declared in. For each level we check if that level is executed potentially more than once. If any level, including the level the variable is used in, but not the level it was declared in, is potentially executed more than once, this means that a variable node is executed an unknown amount of times, compared to the scope it was declared in. Thus the variable is used an unknown amount of times. If all the scope levels are executed at most once, this means that the variable is also only used a single time.

3.1.5 Usage of function arguments

Since the Kotlin compiler only supports intraprocedural analysis, this limits the ability to see how often a variable is used, if it is passed as an argument to a function. If you could do interprocedural analysis, you could extend the analysis to make each function provide information about how often it uses its variables, but this is not possible with only intraprocedural analysis. To mimic this, this implementation provides annotations that can be placed at function arguments, to signify how often they will be used. This then allows usage information to be updated accordingly, whenever a variable is passed as a function argument. The values this annotation can have are identical to the values of the usage lattice.

Along with this, the usage of the argument is verified, by comparing the calculated usage value at the end of the analysis with the provided annotation. If the two values do not match, a compiler warning is emitted, to notify the user that they have

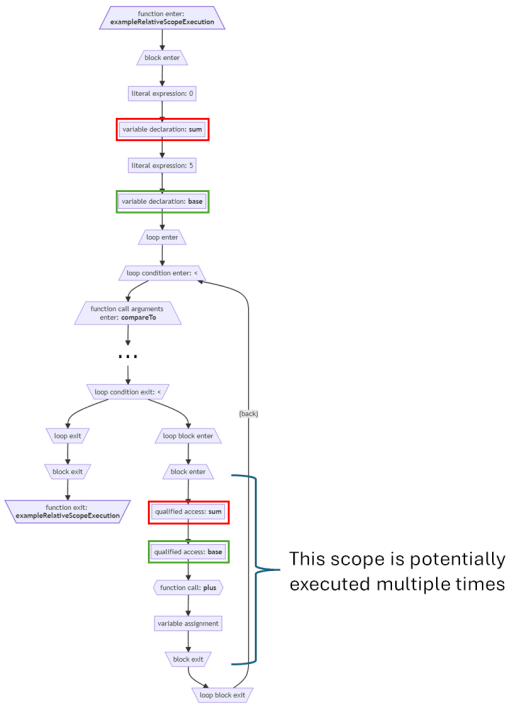

FIGURE 3.4: How often a scope is executed relative to outer scopes

made a mistake with their annotation. If this were implemented in the actual Kotlin compiler like this, it likely is a good idea to change this to a compiler error instead, since incorrect usage information can lead to erroneously updating immutable data, which can lead to incorrect program output.

By adding an annotation to an argument you could create a function header like this:

```
fun example(@Usage(UsageAmount.ONCE) decider : Boolean) : List<Int> {
    ...
    }
```

The argument `'decider'` has been annotated to say it is used once in this function. To verify this, at the end of the analysis the calculated usage can simply be compared to the value provided by the annotation.

3.2 Limitations of this algorithm

While this algorithm for determining the usage of all variables within a function usually provides the correct result, it is still limited, both in that it sometimes provides an overestimation of the usage, and rarely it can even produce incorrect results.

3.2.1 Inter- vs intraprocedural analysis

The first, and biggest limitation is that only intraprocedural analysis is performed. This makes it impossible to track how data is passed through functions, meaning that values that are passed through multiple functions, and then are only used once, cannot be updated in place. This can be partially circumvented by the annotations that provide usage, but this still requires a lot of additional work on the part of the programmer, which might not be worth it.

3.2.2 Aliasing

While the Kotlin compiler already has a system built in to determine whether two variables are an alias of each other, this method of determining aliases returns an underestimation of potential aliases. This means that any alias that this method finds is guaranteed to be an alias of another variable. For this algorithm, however, an overestimation would be required, since we want to determine the upper bound of the usage of a variable. This means that when any variable that can potentially be an alias of another is used, this should also increase the usage of that potential alias. Because of the mismatch here, with one algorithm providing an overestimation and the other an underestimation, there is potential for errors here.

3.3 Algorithm Sketch

In this section we will formalize how the algorithm works. This algorithm works by analyzing the CFG of a function. It does this with a function that analyzes a node and its incoming and outgoing edges. This is a recursive function that calls itself for each child of a node, as long as all the parents of the node have been visited already. This is to ensure that the usage analysis of all previous paths have been completed.

The function itself takes three arguments: what node is being analyzed, and map from node to the usage information from when that node was analyzed, and a map from variable to usage. The second argument also doubles as a visited list for CFG nodes in the function currently being analyzed.

There are three big steps in the analysis of a node, with step 2 differing, depending on what node is being analyzed:

1. Determining if the current scope is executed at most once.

2a. Updating what the currently active scope is.

2b. Processing uses of variables in this node.

3. Deciding which children should be analyzed now.

Step 1

The first step of this function is to determine how often the scope this node is in is executed.

- If the current node opens a scope and one of the incoming edges is a backedge: This scope is potentially executed more than once.
- Otherwise: This scope is executed at most once.

Step 2a

If the node we are currently analyzing, either opens or closes a node, we execute this step, otherwise we choose step 2b. Step two checks if a node opens or closes a node, or does neither.

- This node opens a scope:
  - If this is the entry node of the function we are analyzing, create a new scope information object and add all the function arguments to this scope with an initial usage value of zero.

  - Otherwise: Create a new scope information object and set the parent as the scope information object of the previous node.

- This node closes a scope: Every variable that was declared in this scope is now final, so add their usage value to the variable to usage amount map.
  - If there is only one incoming forward edge to this node: set the scope information object of the parent as the active scope.
  - If there are multiple incoming forward edges to this node: Merge the scope information objects of all the previous nodes into one and set the parent of that object as the currently active scope.

Step 2b

The other option for this step of the analysis of a node is to process any uses and declarations done in this node.

- If this node is a variable declaration node: Add the variable that is declared to the currently active scope information object.
- If this node is a variable access node:
  - If the variable being accessed is from the current scope: increase the usage value of this variable according to the rules described in the start of this chapter.
  - If the variable being accessed is not from the current scope: Check if the variable is from the parent scope, or any of its parents and keep track if any of the scopes are executed more than once.
    - If any scope is executed more than once: set the usage value of the variable to unknown.
    - If no scope is executed more than once: update the usage of the variable according the rules described at the start of this chapter.
- If this node is neither: do nothing.

Step 4

After analyzing a node, we need to decide what node to analyze next.
For every child:

- Have all the parents of this child been visited: Call the analysis function with this child as the node being analyzed.
- Otherwise: Do not analyze this node yet.

3.4 Summary

The research question for this chapter was: "How can we statically determine, during compilation, if a piece of immutable data will be used at most once?" To answer this research question, we developed an algorithm that analyzes the CFG of a program and determines for every function how often each variable in it is used. This algorithm provides an over-approximation of how often a variable is used. This is

done, because we want to be certain that we can reuse memory, which can only be done if we are certain a variable is used at most once.

While this algorithm provides a good baseline for determining the usage of variables, it can certainly still be improved. The current approach uses intraprocedural analysis, because the Kotlin compiler does not have interprocedural analysis in place. Adding the option to do interprocedural analysis could allow for big improvements for this algorithm.

This algorithm returns, for every variable declared in a function, if it is used at most once. The reason this is done, is because if a variable is used at most once, you can be certain that that use is also the last use. To improve this algorithm, you could instead create an algorithm that tries to determine the last time a variable is used. This is more complex than just determining if a variable is used once, because of several factors, such as non-linear control flow, capturing of values, and passing variables through functions. Despite this, it might still be possible to create an algorithm that can answer this for some instances.

In conclusion, the algorithm introduced in this chapter provides a method to statically determine if a variable is used at most once, answering the original research question. By using the CFG representation of a program and a simple lattice structure, we can efficiently determine how often all variables are used inside of functions. This can also handle more complex scenarios, such as branching in the CFG and loops. However, due to limitations with interprocedural analysis and aliasing, this algorithm can still be further improved by addressing those issues. Despite this, this algorithm is a good starting point to help provide the information needed to automatically reuse memory of immutable data in Kotlin.

Chapter 4

Transformation

After collecting usage information, we know which variables can and cannot be reused. Armed with this information, we are ready to transform the code so that memory can be reused instead of newly allocated. But first we have to decide how we want to do this transformation. Transforming the code is not as simple checking where there a variables with only a single usage and then assigning this memory to the next variable that gets declared. This is because the old memory can not always hold all the data, or the next assignment

To actually begin transforming the program, we need to decide on a method to do so. This method should give a consistent transformation that automatically decides which immutable objects can be updated in place and which cannot. But before we can do that, we need to decide how to store information about mutating vs non-mutating functions.

4.1 API Design

Before we dive into possible solutions, we first need to establish some criteria our solution should adhere to as best as possible. These criteria will serve as guidelines when designing our solution. By keeping these constraints in mind, we can create a solution that will be as good as possible, given our constraints in time and resources. Below the most important criteria are listed:

- **Ease of use**: An ideal solution will be very easy to use for the programmer. This can either be achieved by making the steps to activate this very simple and straightforward, or by minimizing what the programmer sees of this solution. By keeping this solution easy to use, the impact on the programmer is minimized and will thus likely result in this solution being used as much as possible.
- **Extensibility**: By allowing this solution to be easily extended, anyone wanting to implement this in their own tool or library can do so. This would allow for the Kotlin ecosystem to easily adapt to this solution and make Kotlin a more efficient language overall.
- **Backwards compatibility**: Creating breaking changes in a language is not something that should be done lightly. Because of that, this solution should either completely avoid, or minimize any such changes. By doing so, old libraries can implement this solution while not requiring any updates to the project using them.
- **Performance**: Performance is an important metric for nearly any program, and obviously is for this solution as well. Any solution must minimize or eliminate any negative impact on either metric.

- **Safety**: Lastly, the solution should be safe to use. Any program that runs with this optimization, should be guaranteed to produce the same result as without the optimization. This is a very important metric for any solution to this problem, because an unsound transformation can lead to wrong program results, so this needs to be properly tested and verified.

4.2 Different approaches

With these criteria in mind, there were several approaches that were considered for this thesis. They all performed differently regarding those criteria, but in the end only one was chosen. This solution is not a perfect catch all solution, but it provides a method that is relatively simple, yet powerful.

An ideal solution to this problem would be one that automatically performs any feasible program transformation, without requiring any additional information from the user. To achieve this, however, is not reasonable within the time frame of this thesis. This is because it would require a very thorough analysis of the language, as well as a significant alteration of the Kotlin compiler. Because of this, we will have to limit our work to a solution that achieves as big of an impact possible with the available time.

The general idea we have for our solution is to create a way to switch between mutating and non-mutating code, based on the result of the usage analysis we performed in the previous chapter. We have devised several methods that can work as a potential solution, but due to the limited time available for this thesis, we can only implement one of them. However, it is still important to mention the other approaches, since they can still have value.

Two implementations for the same function

The first approach we looked at for this was one where a function has two different implementations, based on if it mutates or not. Both functions would have identical signatures, so the programmer would only be exposed to a single one. Then during compilation of the program, the function would be chosen, based on the usage of its arguments.

While this method seems like a good solution, it brings some practical concerns with it. Namely having to create two functions with the same signature is not something that is usually possible. Along with this, it is also hard to switch around functions in the program after the type checker has run. This transformation would run after this stage, so a solution for that would also have to be found. Because of those reasons this approach was not picked.

Annotating paired functions

An alternative to having two implementations for a function is creating an annotation that can be attached to functions to pair them together. This annotation contains three pieces of information: the other function it is paired to, which argument mutates and if this is the mutating version. During compilation, the compiler can check if a called function has this type of annotation. If the annotation is present, the correct version can be chosen, based on the usage of the indicated argument.

A slightly different approach to this is to only annotate the mutating variant of a function. This opens up the possibility for anyone to create a mutating version of a function. Doing so allows developers to optimize any library they are using, but it

also brings potential security risks. Since the programmer did not create the original function, they might not completely understand what it does and inadvertently create a mutating version that does slightly different things. Detecting this, both from a compiler perspective and a user perspective can be very difficult. Thus, allowing this can compromise safety and might need to be avoided.

Internal compiler optimization

The second approach is to select already existing functions within Kotlin that will have the most impact when optimized. These functions will get a highly optimized, mutating implementation. The compiler then selects which version is chosen, based on the usage analysis. Since the compiler will handle this directly, we are not restricted by having to write Kotlin code, but we can create these functions with JVM-bytecode. Since this is done internally also allows the usage analysis to be more specialized, since it only needs to look at the variables which are passed to those functions.

Doing the optimization this way scores really well on four of the five criteria. Since everything is handled internally, safety, performance, backwards compatibility and ease of use can all be handled very neatly. The reason this approach was ultimately not chosen, is because it is not extensible at all. We do want our solution to be extensible, so that is why we ultimately did not select this approach. It is however possible to implement this approach along with our chosen approach, so further research into this is likely worthwhile.

4.2.1 Chosen approach

The chosen approach for this thesis is similar to the first approach described in 4.2, but with a few alterations. These alterations are meant to eliminate the issues with the approach, while maintaining its general idea. The main issues with the approach were that it would require a lot of changes in the Kotlin compiler to make it work.

These issues are both solved by the same solution. This solution is to add the possibility to add an additional argument to a function that would work as a flag to differentiate between the mutating and non-mutating version of a function. This value of this flag will then automatically be determined by the compiler, based on the usage analysis in the previous step.

Any function with this flag would have to be created with two operating modes. One where it allocates new memory for the result, and one where the corresponding argument would be updated in place instead. The flag would determine which mode is chosen. This solves the problem where you need two different implementations per function.

The type of the flag could simply be an enum that is standardized and can be defined as follows:

```
enum class Mutate { YES, NO }
```

While adding a new argument seems to break backwards compatibility, one of our important criteria, this is only the case in very niche instances. By defaulting the value of the argument to 'no', in most cases you can call the function with the exact same arguments as before the change. This means that in the vast majority of cases any code that uses a library that now has potentially mutating functions does not need to be changed.

An example of how this could work is shown here with the function 'map'. Map is a higher-order function that applies a function to every element of a list. The function it applies is one that transforms every element from the element type of the list to another, but possibly the same, type. Because lists in Kotlin generally hold reference types, this function always produces another list that uses exactly the same amount of memory as the input list. Because this is a function that is used very commonly in Kotlin code, and the memory usage of the input list is guaranteed to be the same as the memory usage of the output list, this function is an ideal target for a relatively easy optimization.

```
fun <A, B> List<A>.mapMutate(shouldMutate: Mutate = Mutate.NO,
  transform: (A) -> B): List<B> =
    when {
        shouldMutate == Mutate.YES && this is MutableList<*> -> {
            val me: MutableList<A> = this as MutableList<A>
            val result: MutableList<B> = this as MutableList<B>
            for (i in indices) {
                result[i] = transform(me[i])
            }
            this
        }
        else -> this.map(transform)
    }
```

To do this transformation, we would have to slightly alter the original definition of 'map'. First, an additional argument is added, which determines if the behavior of 'map' is the original behavior, creating a new list, or the new behavior, changing the memory in place. Let's call this argument `'shouldMutate'`, and it is of our enum type `'Mutate'`. The argument by default is always set to `'Mutate.NO'`, and if that is the case, the default behavior of map is used. But if the list argument of map is only used once, the argument could be changed, at compile time, from `NO` to `YES`. This makes the changes to the CFG required to implement this optimization very minimal, only having to change the value of an enum to a different entry of the same type.

The definition of map would also have to change. Because Kotlin does not allow you to alter immutable data types, it would seem that there is no easy way to do this. This would complicate the process of implementing this optimization quite a bit, because it would require altering two parts of the Kotlin compiler, instead of one. Fortunately, it is possible to trick the Kotlin compiler into changing an immutable list to a mutable one, because both are compiled to the same list type in JVM byte-code.

By casting the input list to the result type, and also retaining a variable with the original input type, we now have two references to the same original list. The compiler does not see these two references as being the same, so it will have no issue overwriting data in the original list. The reason the compiler does not realize the list is of two types at once, is because generic types are changed to the lowest bounded class. Since 'A' and 'B' are unbounded, they will both be changed to be of type 'Object' during compilation of the program.

4.2.2 Implementation

Besides the usage information, we also need to know what function can mutate and which of their arguments are the ones doing the mutating. Which functions are

mutating is very simple: any function with an enum of the type 'Mutate' is designed to have to possibility to mutate. However, which argument will be mutated is a bit more complex. This is the argument for which we need to check its usage, so we need to know which argument this is. In our example there are three arguments: the flag to indicate if the function mutates, the transform function and the 'this' argument. This is the argument which will be updated in place, so we will have to check if it is used at most once.

While our example is hard-coded to check the usage of this argument, this is not a feasible method in general. To solve this, we need to give the programmer a method to mark which argument is being reused. While this is future work, a way this can be done is by providing an annotation to indicate which argument would be reused.

With this information present, the transformation is fairly simple to apply. Traverse the CFG, check if the current node is a function call node and then check if this function can mutate. If it can, check the usage of the mutating argument. If it is used (at most) once, set the flag to should mutate. Otherwise leave it as non-mutating. This is done for every node in the CFG. The order of traversal is not important, since the changes this algorithm will make to the CFG are all independent.

To illustrate, take the following function:

```
fun positiveExample() : List<Int> {
    val list = listOf(1, 2, 3)
    return list.mapMutate(Mutate.NO){it + 1}
}
```

Note that the mutate flag argument to `'mapMutate'` can be omitted, since it has its default value. This is not done for clarity in this example. The usage analysis returns that `'list'` is only used a single time, thus we can apply the transformation. Simply change the mutation flag from `NO` to `YES` and the implementation of `'mapMutate'` does the rest. After the transformation, the code will look as follows:

```
fun positiveExample() : List<Int> {
    val list = listOf(1, 2, 3)
    return list.mapMutate(Mutate.YES){it + 1}
}
```

If the usage analysis returns that the mutating function argument is potentially used more than once, no transformation is applied. Take for for example the following code:

```
fun negativeExample() : List<Int> {
    val list = listOf(1, 2, 3)
    println(list)
    return list.filterMutate{it % 2 == 0}
}
```

Since you can trivially see that list has two uses, according to our heuristic we are not allowed to change the call to filter to a mutating version. This leaves the code as is, with the implicit mutate flag of `filter` at its unaltered default value of `NO`.

Chaining operations

Of note is that this optimization is a sound optimization, provided the usage information and the mutating implementation are correct. This means that this optimization can always be applied, if an instance is found. This is especially useful when

multiple potentially mutating functions are applied in a row. For example, take this function:

```
fun chainingExample() : List<Int> {
    val list = listOf(1, 2, 3)
    return list.map{it + 1}.filter { it % 2 == 0 }
}
```

Running our optimization gives us this result:

```
fun chainingExample() : List<Int> {
    val list = listOf(1, 2, 3)
    return list.mapMutate(Mutate.YES){it + 1}
        .filterMutate(Mutate.YES){ it % 2 == 0 }
}
```

This version saves the allocation of two lists over the not optimized version, since the result of map and filter can both update the initial list in place.

Inlining functions

When it comes to performance, our solution is a bit worse than directly executing the chosen function. This is because our functions need two different execution paths: the mutating and the allocating paths. While this only adds a single if-else evaluation, this can still add up over millions of function calls. There is however a way to circumvent this, and this way lies in the `inline` keyword in Kotlin.

Any function that has the `inline` keyword in Kotlin will have a call to it substituted by the actual code of the function during the compilation process. So if we add the `inline` keyword to our mutating functions, they will be directly inlined. This then enables constant propagation to optimize the the not chosen branch away, since the value for the mutate flag is always known at compile time. Take the first example in 4.2.2, if `mapMutate` has the inline keyword, the function would be inlined as follows:

```
fun positiveExample() : List<Int> {
    val list = listOf(1, 2, 3)
    return if (Mutate.YES == Mutate.YES) {
        ... // mutating variant
    } else {
        ... // allocating variant
    }
}
```

Since `Mutate.YES` is always equal to itself, the compiler deduces that the else branch of the if statement is never chosen, and thus optimizes it away. This also works if the non-mutating branch is chosen. Because inlining large functions is not recommended, it is a good idea to implement mutating functions as an if-else statement where each branch is a function call. This way the correct function will be inlined. This then gives the same performance as implementing this optimization by automatically choosing the correct function.

4.3 Kotlin's Standard Library

A good application of this technique would to implement it in Kotlin's standard library. Standard library functions are often among the most used functions in any programming language, so optimizing them can yield great result. This would result in a good potential improvement to the overall Kotlin ecosystem. Along with this it would also serve as a good example to library developers who want to implement this solution in their own library and it can also be used to benchmark real world performance impact of this solution.

Functions in the Kotlin standard library that can benefit from this include, but are not limited to:

- Transformation functions, such as `filter` and `map`.
- Sorting functions, such as `sort` and `shuffled`.
- Partitioning functions, such as `take`, `drop` and `partition`.
- String processing functions, such as `replace` and `split`.
- Copying functions, such as `toList`, `toMap` and `toSet`.

4.4 Results

In this chapter we set out to answer our second research question: "Under what circumstances can memory from immutable data structures be reused safely to store the result of computations using them?" The approach we discussed provides an underestimation to this question, but it provides a very solid foundation. This is done by providing a method for programmers to create functions that have to option to update its data in place, or allocate new memory.

The approach we discussed, overall, scores well on all the criteria we set. The five criteria we set are:

- Ease of use: It is easy to use for any programmer, since they generally do not interact with the new parameter at all.
- Extensibility: Our implementation provides a way for a creator of a library to create a version of their functions that can update data in place. This allows for the creation of a more efficient Kotlin ecosystem.
- Backwards compatibility: While our approach does not score perfectly in backwards compatibility, it should only break in rarer cases, where functions are called by introspection or through storing functions in variables.
- Performance: The performance of a program with our solution should be the same, or better, than one without it. This is because the execution time is about the same, and memory usage is better.
- Safety: Our approach does not provide a perfect solution when it comes to safety. Having two version of functions adds a new dimension to testing, since both versions always need to produce the same results. Despite this, if the library is properly tested, there should be no general safety concerns with this method.

While this solution is not a perfect solution that works without any additional effort of end users, it is still a step in the right direction. As it stands, immutable data in Kotlin creates large overhead in certain circumstances, and in those instances updating the data in place can circumvent this. Our algorithm provides a reliable way of doing so and most of the egregious waste of memory can be avoided this way.

There are still major improvements to be made, with a method that automatically creates the mutating variant functions instead of relying on the programmer being an amazing outcome. This would provide increased safety for that solution, in comparison to our solution, since the transformation could be done via a sound rule-set.

4.5 Summary

In this chapter we discussed how we are going to transform the program after calculating the usage information of the variables. The way of transforming that will be chosen has to score well on certain criteria, being: ease of use, extensibility, backwards compatibility, performance, and safety. We discussed several approaches that could work and then explain our chosen approach.

Our chosen approach is to provide an API-like interface for libraries that can be used to indicate which function have to option to mutate their arguments in place. This is done by creating a flag that can be set by the compiler, based on the usage of its arguments. If an argument is used at most once, this means that the argument can be mutated in place. This transformation is done by traversing over all nodes in the CFG and checking if it is a function call node. If the node is a function call node, we check if it is a potentially mutating function. If it is, and the mutating argument is used at most once, the flag is changed from its default, non-mutating, to the mutating value.

Chapter 5

Benchmark Design

Now that we have devised a feasible transformation method it is important to measure its impact. Quantifying impact of this research is a complicated task, as we only provide a solution for others to implement. Because of this, it is not possible to take an existing code base, run our optimization and compare the performance and memory usage with and without it. Instead we will setup several benchmarks with varying parameters, to get an understanding of how our solution performs in different circumstances. Because it is hard to get direct performance numbers, we will primarily focus on more indirect measures of performance, but there will also be benchmarks that measure performance more directly.

There will be two types of benchmarks: benchmarks how well our usage analysis works, and benchmarks on how well the program transformation will perform. Both of these benchmarks will give us insights into the weak and strong parts of our solution, as well as give an indication of the impact implementing this in real world applications. Further details are described in the rest of this chapter. Due to time constraints, we were not able to actually run the benchmarks and gather the results. If this research is extended in the future, that is a good place to start.

5.1 Measuring the effectiveness of our usage analysis

The algorithm we devised in Chapter 3 calculates the usage of all variables in a piece of Kotlin code. It does this by providing an overestimation of the usage of variables. Because of this, there are situations where a variable can be reused, but our algorithm will mark it as not reusable. To assess how well our algorithm performs, we want to measure how often this occurs. Assessing our algorithm this way gives us a proxy for its real-world performance.

The way we will do this is by creating a comprehensive set of tests in which we know whether a variable can be reused or not. These tests should cover as many different scenarios we can think of and it should be manually determined for every variable if it is suitable for reuse or not. The next step is to then run the algorithm on every test case and compare its outputs with the manually determined reusability of every variable. This should give a good indication of how good our algorithm is for actually detecting when a variable can be reused.

With this information we can determine how effective our algorithm is at actually detecting opportunities for memory reuse and we can identify the strong and weak-points of our algorithm. If we find any area in which our algorithm is severely lacking, or perhaps erroneously marking variables as reusable, when they are not, we can know where to make improvements to our algorithm.

We expect that variables that are only used within the scope they are declared in are always correctly identified as (non-)reusable and variables from out scopes

are also always correctly identified, if there are no loops involved. Once loops are involved things get more tricky, because the control flow is no longer linear. This might result in certain cases being incorrectly identified as not reusable. One that is likely to be misidentified is when the only use of a variable is during the reassignment of that variable.

```
fun example() {
    var list = (1..1000).toList()
    for (x in 1..10) {
        list = list.drop(x) // List is only used once and then overwritten
    }
}
```

Our algorithm would identify `list` in the above example as being used more than once, since list is from an outer scope and the current scope is potentially executed more than once. Despite this, `list` can be reused in this context, because the reference gets overwritten and thus its usage gets reset.

5.2 Measuring the potential impact of our analysis

Another metric that is very important to assess is how well our algorithm performs in code-bases that are representative of professional Kotlin use. Since our solution does not provide a direct optimization, but instead provides a method for programmers to optimize their own code, we cannot directly quantify the impact of our solution. Instead, we will be taking metrics that show the potential of our solution. These metrics will focus on how often our solution has the potential to trigger in different circumstances.

Because of the previously mentioned limitation where we cannot directly measure the impact, but we have to estimate it. To get this estimation we will measure how often our solution can trigger in the following circumstances:

- All function calls: By measuring what percentage of functions are called with an argument with only a single usage, we get a gross overestimation of how often our solution can be triggered. This overestimation gives a clear upper bound of the potential of our solution.
- Only standard library calls: By measuring how often standard library functions are called with an argument with only a single use, we can get a measure of how effective a potential implementation using our method can be.
- Chained function calls: Chained function calls guarantee that the intermediate result is only used once, and can thus be done without usage analysis. If it turns out that a significant portion of all optimization are of this type, it can be worthwhile to only implement the optimizations here, to save resources that would otherwise be needed for the usage analysis.

5.2.1 Benchmarking all functions

To get a measure of how often our solution has the potential to trigger, we will measure what ratio of functions is called with an argument that only has a single use. Measuring this allows us to get a good indication of how effective our solution can be on any code base, without needing to create an actual implementation for all the functions in it.

To run these experiments, it is important to decide on which code bases they are to be tested. The code needs to be representative of actual Kotlin code in professional development, but also manageable in terms of complexity for the purposes of analysis and benchmarking. Ideally, the selected code bases should include a mix of small to medium-sized projects to facilitate controlled experimentation, as well as larger, industry-scale projects to ensure that the algorithm performs well under realistic conditions. Furthermore, the code bases should utilize various Kotlin features, such as functional programming constructs, higher-order functions, and immutable data structures, to get as realistic as possible a representation of the performance of our algorithm.

Measuring performance like this is an overestimation of how often our solution can trigger, since not all functions can be optimized, or are worth optimizing. But since this is an overestimation, it also gives an upper bound of the usefulness of this solution. Based on how often our solution can trigger, different choices can be made, further in the design process. If it turns out there are generally very few cases where our solution triggers, different methods to optimize the memory usage of immutable data structures can be explored. Or, if it turns out that a very high percentage of function calls are with single use variables, it might be needed to look into a different, more automated transformation strategy.

5.2.2 Benchmarking standard library functions

Within the standard library of Kotlin, there are a lot of functions that benefit from updating its arguments in place. If this solution were to be implemented in the official Kotlin compiler, it is only natural that the standard library of Kotlin uses it. By filtering which functions within the standard library would benefit from having the ability to update its arguments in place and marking what argument would be updated in place, we can create a benchmark that can show the potential of our solution in a realistic implementation.

This experiment can be run on the same code bases as in section 5.2.1. By collecting what percentage of all function calls are of the selected functions, and what percentage of them the relevant argument only has a single use, we get a very clear indication of the potential impact of our solution. These metrics can be used to show that it is worthwhile to add this optimization to libraries and what can be expected of the result.

If this optimization is implemented for the Kotlin standard library, benchmarking performance directly also becomes a possibility. This would result in a more direct measure of the impact of a solution like this. Since this optimization targets the memory use of programs, benchmarking the memory usage of impacted programs with and without the optimization is very logical, but it does not paint a complete picture. To get a more complete picture, metrics such as garbage collection overhead, memory allocations and object count are also very important. Along with those, benchmarking runtime is also important, to ensure our optimization does not aversely affect runtime too much.

Since standard library functions of a language are often among the most used, because they contain functions that are commonly used in programming, this is likely one of the biggest optimization targets within Kotlin, at least when it comes to the total percentage of function calls that would benefit from our optimization. When it comes to what percentage of function calls of this library are worth optimizing, it is more representative of what other libraries might achieve as well.

5.2.3 Chained operations

Chained operations create temporary variables to store the result of intermediate computations. Generally speaking those intermediate results are discarded immediately after they are used for the next computation, so the memory allocated for them is then available for reuse. Finding out where this occurs does not require simple usage analysis and can instead be done by checking if a function is called on the result of a previous function. This simplicity makes chained operations an excellent candidate for targeted optimization, as their patterns are easy to identify and consistent across various use cases.

Because it is straightforward to determine whether a function call is part of a chain, it is worthwhile to measure how frequently these patterns occur and what proportion of optimizable function calls fall into this category. Based on this metric, it can be decided if implementing a full usage analysis is worth it, or if perhaps only adding the optimization for these instances is worth it. During development it is important to make informed decisions on what and what not to do, so this metric is very valuable for deciding if implementing usage analysis is worth it.

While implementing optimizations solely for chained operations might provide a quick and impactful solution, it is important to evaluate whether these cases constitute a large enough percentage of all potential optimizations. If it turns out they are a too small amount, relying solely on those instances leaves a lot of room for optimization. However, if a significant amount of optimizable function calls are of this type, this could justify prioritizing a simpler implementation, rather than a more resource-intensive usage analysis.

If it turns our our usage analysis is not feasible, due to technical or resource constraints, it might still be worth it to implement our solution for just chained operations. Chained operations on potentially very large data types, such as lists, can accumulate memory very quickly, and in these cases it is worthwhile to update them in place, instead of allocating new memory for every temporary result.

Chapter 6

Discussion

In this thesis we have discussed a method to determine which variables are used at most once in a piece of Kotlin code by analyzing its CFG and thus safe to update in place. Along with this we have devised a way to use this information to automatically switch from allocating new memory to updating one of its arguments in place. Both of these have been adapted into a Kotlin compiler plugin as a proof of concept, but a fully fledged implementation would require more work still.

6.1 Usage Analysis

The usage analysis we developed is a data flow analysis algorithm that is based on lattice analysis to determine the usage of each variable within a function. The analysis is an over-approximation of the usage, since it needs to be certain that any variable that is marked as reusable is not used again, because this could lead to wrong program output.

To implement this system in the Kotlin compiler there is still work left to be done. There are still some potential errors that can occur when variables are aliased, so implementing a system to detect any potential alias would be the first step. After this, the system should be further analyzed to ensure that no variable is erroneously reported as reusable. This can be done either through comprehensive testing, or through formal proofs and reasoning.

The currently known biggest flaw of the analysis, is that reassignments of variables within loops do not reset the analysis of a variable. This means that when the only use of a variable is in loop, in the calculation of its own reassignment, this variable would be marked as not reusable, when in fact it can be. Further experimentation is warranted to see if other such limitations exist.

Our solution only looks at variables with only a single use, as a safe approximation for variables that can be reused. This was done because it is harder to determine the last use of a variable, if there are multiple uses, because of things such as sharing and closure capture. There are undoubtedly instances where a variable is used multiple times, yet it is still possible to figure out at compile time which of those uses is its last. Using a lattice of natural numbers instead, along with escape analysis[2] can perhaps achieve this.

Another limitation is the lack of intraprocedural analysis within the Kotlin framework. Intraprocedural analysis could enable the passing of usage information of variables between functions. This can enable more in place updates, such as when a variable is passed through multiple functions, with each pass being the only use in that function. Since there is no intraprocedural analysis, currently the only way to pass usage information between functions is to do so as arguments, like with our program transformation scheme.

A different approach to this solution would be to implement something such as uniqueness typing into Kotlin, and use that information for the program transformation instead. This would however require an overhaul of the Kotlin type system, and not worth it, depending on how big the benefits are, compared to our usage analysis. Still, it might be worthwhile, and further research could explore this.

6.2 Transformation

We developed a method of automatically switching between memory allocating and in place updating variants of functions, based on the usage of their arguments. This method relies on developer input to create a function that can do both, switching depending on the value of an argument added to the function. The current status of this transformation method is that of a proof of concept. To further improve it, some work is still remaining.

The first step would be to generalize the method that is currently being used to determine which functions can update one of their arguments in place. Currently which functions can do this is hard-coded in the plugin, but to allow this method to be implemented by anyone, we need to add universal annotations which tell the compiler that a function has the functionality to update in place instead of allocate. These annotations need to be added to a publicly available package and be properly integrated with the Kotlin compiler.

The next step would be to actually implement this optimization to any applicable function within the Kotlin standard library. This would serve as a good example for any Kotlin developer if they want to implement this in their own library and also reduce the overall memory consumption of Kotlin programs.

A clear limitation of this approach is that it relies heavily on developer input. An alternative solution that is done completely by the compiler is desirable, but was not feasible with the given time-frame of this thesis. In any follow up research, looking into doing this, at least partially, can prove worthwhile. Reasoning about memory layout and if that memory is fit for reusing, during compile time can be difficult, but there are most likely instances where it can be done.

6.3 Benchmarks

Due to a lack time, we were unable to actually complete and run the benchmarks described in Chapter 5. This is the next step in this research, since the results would be guiding for its continuation. Depending on the results, different courses of action can be taken.

The first set of benchmarks, to measure the effectiveness of our usage analysis, also doubles as a test set. Its results indicate if the analysis does a good enough job, or if there are parts that are worth improving. Along with this, any false positives the algorithm might produce can be found, and fixes can be implemented.

The second set of benchmarks will provide a more direct measure of the change in performance this optimization will have. If it turns out that the overall impact of this research is very broad, it is likely worth it to implement this, or a similar solution in the Kotlin compiler. If the results are less promising, it still might be worth it to implement this for just Kotlin's standard library. Doing that would allow for higher levels of optimization, since the scope is suddenly more limited.

Combined, the results of both sets of benchmarks will drive the direction of this research. A thorough evaluation of these benchmarks will also reveal whether additional tools or complementary approaches are needed to fully realize the potential of this optimization. By addressing both theoretical and practical limitations, these benchmarks have the potential to provide not only validation for the proposed solution but also a clear road map for future development. Ultimately, they will determine if this approach can meaningfully enhance Kotlin's handling of immutable data.

Bibliography

[1] Frances E Allen. "Control flow analysis". In: *ACM Sigplan Notices* 5.7 (1970), pp. 1–19.

[2] Bruno Blanchet. "Escape analysis for object-oriented languages: application to Java". In: *Acm Sigplan Notices* 34.10 (1999), pp. 20–34.

[3] *Control- and data-flow analysis*. https://kotlinlang.org/spec/control-and-data-flow-analysis.html. Accessed: 2024-02-28.

[4] Paola Giannini, Marco Servetto, and Elena Zucca. "A type and effect system for uniqueness and immutability". In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 2018, pp. 1038–1045.

[5] Jurriaan Hage and Stefan Holdermans. "Heap recycling for lazy languages". In: *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. 2008, pp. 189–197.

[6] Paul Hudak. "A semantic model of reference counting and its abstraction (detailed summary)". In: *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*. 1986, pp. 351–363.

[7] *Kotlin Lists*. https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list/. Accessed: 2024-02-28.

[8] *Kotlin specification*. https://kotlinlang.org/spec/introduction.html. Accessed: 2024-12-04.

[9] Florian Lanzinger et al. "Scalable and Precise Refinement Types for Imperative Languages". In: *International Conference on Integrated Formal Methods*. Springer. 2023, pp. 377–383.

[10] Anders Møller and Michael I Schwartzbach. "Static program analysis". In: *Notes. Feb* (2012).

[11] Flemming Nielson and Hanne Riis Nielson. "Program Analysis (an Appetizer)". In: *arXiv preprint arXiv:2012.10086* (2020).

[12] Oystein Ore. "Studies on directed graphs, I". In: *Annals of Mathematics* 63.3 (1956), pp. 383–406.

[13] Alex Reinking et al. "Perceus: Garbage free reference counting with reuse". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 96–111.

[14] Barbara G Ryder. "Dimensions of precision in reference analysis of object-oriented programming languages". In: *International Conference on Compiler Construction*. Springer. 2003, pp. 126–137.

[15] *Smart casts*. https://kotlinlang.org/spec/type-inference.html#smart-casts. Accessed: 2024-02-28.

[16] *Smart casts*. https://kotlinlang.org/spec/control--and-data-flow-analysis.html#function-contracts. Accessed: 2024-02-28.

[17] J Smetsers and E Barendsen. "Conventional and uniqueness typing in graph rewrite systems". In: *13th Conf. on Foundations of Software Technology and TCS, LNCS*. 1993.

[18] David N Turner, Philip Wadler, and Christian Mossin. "Once upon a type". In: *Proceedings of the seventh international conference on Functional programming languages and computer architecture*. 1995, pp. 1–11.

[19] *Why teach Kotlin*. https://kotlinlang.org/education/why-teach-kotlin.html. Accessed: 2024-02-28.

[20] David A Wright and Clement A Baker-Finch. "Usage analysis with natural reduction types". In: *International Workshop on Static Analysis*. Springer. 1993, pp. 254–266.