## The problem

Kotlin is a programming language that implements many features of functional programming languages. One of those features is immutable types. Immutable types bring many advantages (insert advantages), but they do have a major downside: memory usage. Whenever you do any operation on an immutable type, new memory must be allocated to store the result, as you cannot modify the previous memory to store the new result.

The upsides of immutable types generally outweigh the extra use of memory, and if memory is a still a major issue, Kotlin also has support for mutable types. But in some cases, the memory usage of immutable types can still be optimized. For example, if you know a piece of memory is only used once in a computation, this memory can be safely reused after that computation.

Deciding how often a variable is used is not a trivial problem, however. [insert example showing it is not just counting how often a variable occurs in the code].

## Generalizable case

The general idea of this thesis is to reuse memory when it is freed up. This could be done by introducing a contract to a function that says that the memory usage of the operation is less than or equal to the input memory. This could be captured in a Kotlin contract. This contract in combination with frequency analysis can then be used to implement the general reuse of memory.

# Kotlin

Kotlin is a programming language that was designed to be easy to write, work natively on multiple platforms and have features from both object oriented and functional programming paradigms. The language is being developed by JetBrains, with active work being done constantly.

Kotlin is interoperable with existing Java libraries by design, allowing for seamless use of existing Java libraries in Kotlin projects and vice versa. This is in large part thanks to the fact that Kotlin compiles to JVM (Java Virtual Machine) code. The JVM is a virtual machine that allows for the execution of Java bytecode by providing and abstraction of the underlying hardware and operating system. Because of this, any program that can be compiled to Java code can run on any platform that has 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.

## Section on Kotlin language constructs

[General explanation of the language and all its constructs. How in depth should this be?]

## Kotlin as a functional programming language

Kotlin was designed to allow for functional programming, implementing many of the well-known functional features, such as higher-order functions, lambda expressions and pattern matching.

## Section on Kotlin’s type system

Like many other languages, Kotlin operates on data in the form of values or objects, which have types. 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.

Any type in Kotlin can either be nullable or non-nullable. If a type is non-nullable, its value can never be null, meaning all operations on such a type cannot result in a runtime error caused by null.

Kotlin has a unified supertype for all types, called kotlin.Any?, meaning that any value can be cast to it. There is also a unified subtype for all types, called kotlin.Nothing.

### Lambda expressions

A lambda expression is 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. [example showing lambda expression]

### Higher-order functions

Kotlin allows for higher-order functions, which are functions that 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: [insert definition map snippet]

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: [insert definition filter snipper]

Fold is an operation that takes a list and combines all the elements of that list to one with the function passed as an argument. It does this by taking two values of the array, passing it into the function and using that result and the next element of the array for the next operation and so on.

It can be defined as follows: [insert definition fold snippet]

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

## Section on smart casts in Kotlin

## Section on control flow graphs in Kotlin

## Section on function contracts [12.2.5 of reference manual]

Some standard-library functions in Kotlin are defined to adhere to specific call contracts. These contracts affect how the control flow graph of the calling function is constructed. A function’s call contracts consists 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.

### Calls-in-place effect

The calls-in-place effect of a function F for a function-type parameter P specifies that parameter P will be invoked as a function. There are three invocation types:

* At-least-once, meaning that P will be invoked at least once.
* Exactly-once, meaning that P will be invoked exactly once.
* At-most-once, meaning P will be invoked at most once.

These effects change the call graph, because without guarantees of the function being executed, the call graph cannot be created as if it would. With that guarantee, corresponding incoming and outgoing edges can be added.

### 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 could be null without it

}

Here it is automatically deduced that a cannot be null anymore when it is used in the calculation of c, because of 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.

## Section on concurrency in Kotlin

Experiment ideas

Motivating examples

When chaining higher order functions on a list, every higher order function allocates new memory for the intermediary result, but those results are then never used again. Take this piece of code, for example:

fun example1() : List<Int>

{

val result = exampleList.map{it + 1}.filter{ it % 2 == 0}

return result

}

The memory usage of this code is as follows:

exampleList is stored somewhere in memory and in this context we do not know anything else about it.

Then the first higher order function, map, allocates new memory to store the result of its computations.

Then the next higher order function, filter, again allocates new memory to store the result of its computations.

If the chain of higher order functions were extended, the memory usage would grow linearly, even though the only memory that would be reused, would be that of the final result.

If you can be certain that the intermediary results are not used again, it is safe to reuse this memory, or do in place updates as it is called.

To be certain memory can safely be reused, some analysis needs to be done.

Planning

Literature survey

Section on control flow graphs

Subsection on graph construction

Subsection on lattice theory

Subsection on analyses using lattice theory, eg. how to express the possible states of an analysis as a lattice and how to work over the control flow graph with the least-upper bound and greatest-lower bound operations.

Section on frequency analysis

Write something on function scope analysis vs global scope analysis.