Formalization of usage analysis

The Kotlin compiler does analyses by transforming the user written code to a control flow graph (CFG) and analysing these graphs. Every piece of code has a representation as a control flow graph, and every element a representing CFG node. For usage analysis, the two most important fragments are the Qualified Access Node, which indicates a variable is used and the Variable Declaration Node, which creates a variable.

The goal of usage analysis is to determine for any variable ‘x’ how often it is used in a CFG fragment. The naive method to do this would be to simply add any variable ‘x’ to your analysis domain when it is declared in a Variable Declaration Node and set the current usage to 0 and increase that by 1 every time a variable is used in a Qualified Access Node. This approach however does not always yield accurate results, because things such as branching, loops, aliasing, and object fields are not considered.

To start:

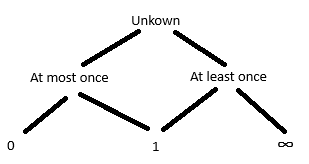
Our algorithm starts with an empty analysis domain to which variables can be added. The state of the variable is represented with a lattice over the following set: {0, 1, ∞}. The elements of this lattice are chosen to be only the numbers 0, 1 and infinity, because more fine-grained counting complicates this problem a lot while not having the greatest return.

A variable is added to the domain when it is used in a Variable Declaration Node. It is added with the initial value 0.

When a variable is seen in a Qualified Access Node, this means a variable is “used” once, so its usage goes up, meaning if a variable currently has 0 uses, its value in the analysis domain is changed to 1, if it is already at 1 it is changed to ∞, and if it is already at ∞ it stays there.

Next up is branching. Whenever one of the following CFG nodes occurs, this means that there will be a branch in the CFG: WhenBranchConditionExitNode, LoopConditionExitNode, TryExpressionEnterNode, TryMainBlockExitNode, BinaryAndExitLeftOperandNode, BinaryOrExitLeftOperandNode, JumpNode.

When such a node is encountered, the analysis will also split in two and is joined later when the branches meet again. Each time the analysis is split, the new analyses will both copy the analysis domain of the previous node and continue separately from another. When two branches again, their domains will be joined using the lattice join operator, resulting in the following possible values for each variable:



Because it is fundamentally impossible to determine how often a loop iterates, the usage information of variables that are used within loops is hard or even possible to determine, as the loop could iterate zero times, once, or any other amount. Because of this, if the usage of a variable is different at the backedge at the end of the loop than it was before the loop started, we can change the usage of that variable to unknown, like the KillDataFlow instruction already present in the Kotlin dataflow analysis.

Object fields are part of objects, and to access an object field, first an access of the variable itself must be done. To keep track of this, the variables within the analysis domain will have a tree-like structure. Any variable has its own usage value and every field of that variable has a branch from that variable with again its own usage value, and so on.

Whenever some field ‘y’ of variable ‘x’ is used, such as in val q = x.y + z, the usage for both x and its child y go up. This is done, because if only y’s usage count is incremented without incrementing x’s, problems can arise when x is only used once elsewhere, and the memory for the reference x holds could thus be reused. If any usage of x.y occurs afterwards, this would now point to invalid memory.

Function arguments pose a problem to the analysis, as their initial value is unknown, within the function scope, and they could also be aliases of one another. To combat this, function arguments are handled in a different manner from regular variables. When a function is analysed, all arguments are treated as separate variables, but instead of an absolute usage count, they hold a relative usage. At the end of the analysis of a function, for each argument it will be known how often it is used. This information will be stored, so that when a function that is already analysed is seen in the CFG analysis, the relative usage of each parameter can be added to the absolute usage of the variable that is used as a parameter. This will also allow for aliasing within function calls to be possible. If it is determined that two or more parameters are aliases of each other, the relative usage of those parameters can be added and the total usage can be determined.