1 Interprocedural analysis

Some analyses are not sufficiently precise when done one procedure at a time, because there is not enough information about what other procedures (either callers or callees) might do. Pointer analysis is one place where an interprocedural analysis often yields significant benefits.

1.1 Example

Accurate pointer analysis depends on being able to identify the allocation sites that generate new pointers. However, it is common for programs to define their own abstractions for allocation as in the following code:

```
f() = {
    a: int[] = newArray(5);
    b: int[] = newArray(5);
    ...
    a[0] = b[0] + b[1]  // are a[0] and b[0] aliases?
}
newArray(n: int): int[] = {
    x: int[n];
    ... initialize x somehow...
    return x;
}
```

To do a good job of optimizing function `f`, the compiler may need to reason accurately about the possible aliasing of `a` and `b` with each other and with other heap locations. With no interprocedural analysis, however, there is no information about what the result of `newArray` might point to. So `a` and `b` must be treated as potential aliases for each other and for anything else of the same type.

2 Context-insensitive analysis

A simple strategy for getting more information is to combine the control flow graphs for the various functions, connecting them according to their calls and returns. The control flow graphs are connected according to the *call graph* for the program. How this strategy works is shown in Figure 1 for the example program.

The analysis is *context-insensitive*: it analyzes the function `newArray` for all possible calling contexts at once. The result of this combined analysis is then sent back to all call sites via the `return` node.

This will at least allow the compiler to determine that `a` and `b` are both generated by allocation *alloc1*, and therefore don't alias other variables. However, it won't be able to determine that `a` and `b` also don't alias each other.

The combined CFG for Figure 1 is an abstraction of the real *call tree* that happens at run time. The call tree has one node for each actual procedure invocation, and therefore contains two distinct nodes for `newArray`. The context-insensitive analysis effectively collapses the CFGs for these two distinct invocations into one CFG.

One way of recovering precision is to connect the CFGs according to the possible call trees instead of the call graph, cloning the `newArray` CFG and creating two separate copies with distinguished allocation sites *alloc1* and *alloc2*. Then we'd know that `a` can point only to *alloc1* and `b` only to *alloc2*: no aliasing possible. The result of context-sensitive analysis using cloning is shown in Figure 2

However, cloning only works if the call tree is finite (and not huge). In general there can be an infinite number of possible call trees, and therefore an infinite number of calling contexts in which a procedure needs to be analyzed. We need a way to *abstract* over possible contexts.

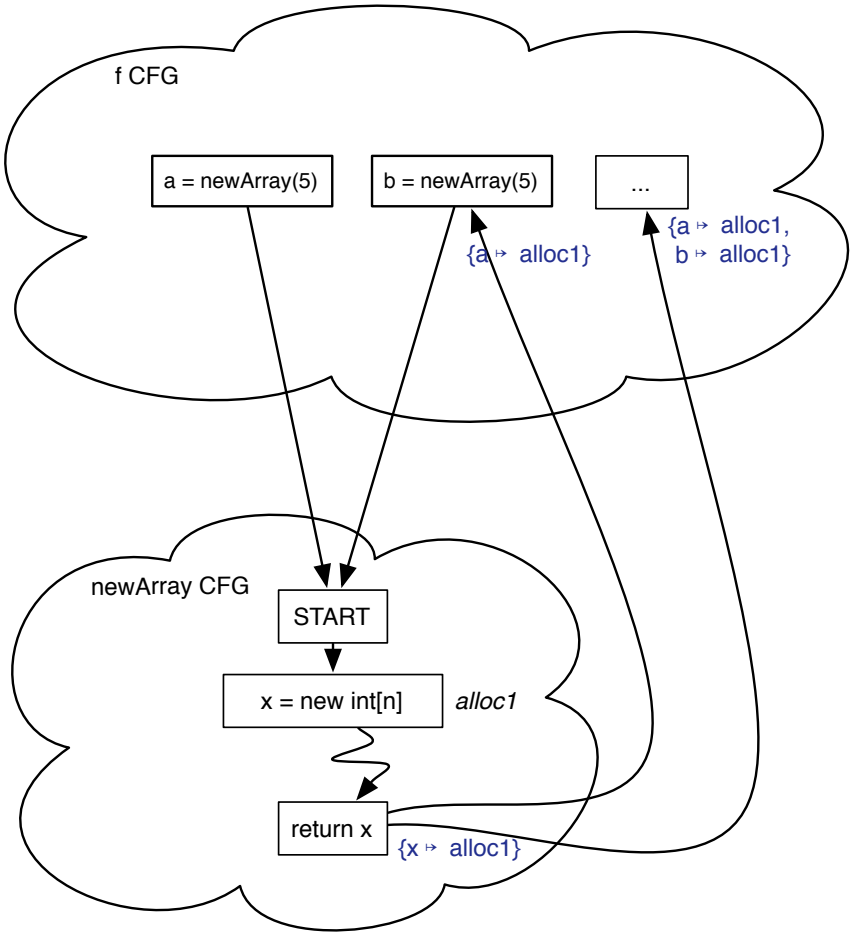

Figure 1: Context-insensitive analysis (dataflow values shown in blue)

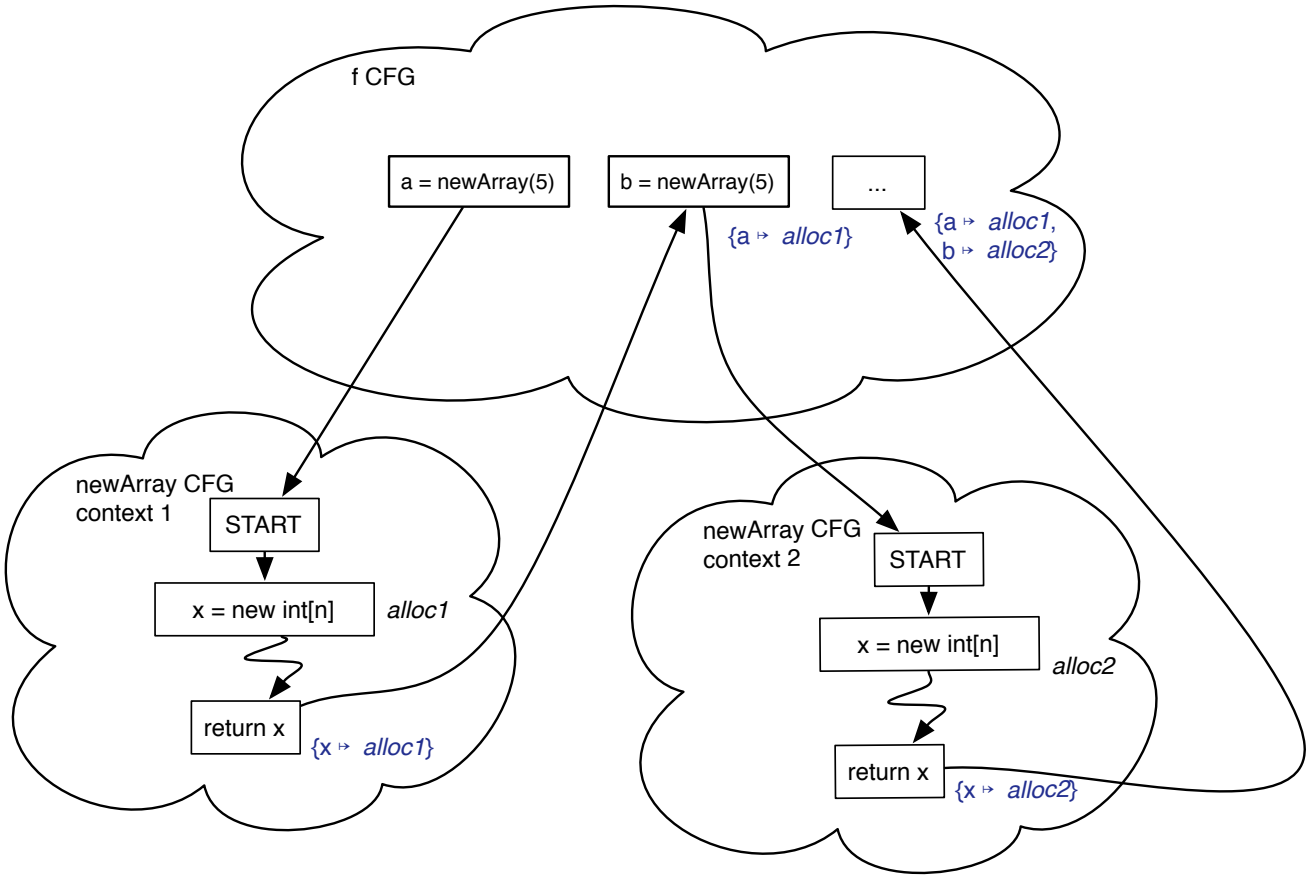

Figure 2: Context-sensitive analysis with cloning

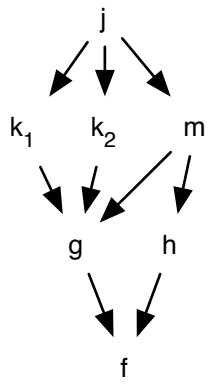

Figure 3: Calling contexts example

3 Calling contexts

Thinking backward from a given call to a procedure f, there is some number of distinct callers, and in turn, these callers may have multiple distinct callers, as depicted in Figure 3. This diagram expands the call graph for the program into a tree in the backward direction. For example, there are two distinct calls to procedure g from procedure k, labeled k_1 and k_2. Here, at least four calling contexts for f are shown, which can be enumerated in terms of the call paths from the root to f. These paths can be named using their *call strings*, which are the sequences of names visted along the path: jk_1gf, jk_2gf, $jmgf$, and $jmhf$.

Each of these contexts could potentially provide and receive different information from their respective calls to f, so additional precision might be gained by making a separate copy of the f CFG for each of these contexts. But in general the number of such contexts will be infinite, in particular if there is any recursion.

4 Abstracting contexts

The solution to the problem of too many contexts, as with the problem of alias analysis, is abstraction: the analysis needs to project classes of calling contexts into single abstracted contexts, and do an analysis that conservatively captures the analyses that would be done on all contexts in the class.

Context-insensitive analysis can be viewed an instance of this approach, in which all contexts are abstracted to a single context. However, more fine-grained abstractions can give more precise results.

4.1 k-limiting context-sensitive analysis

One approach to abstraction is to collapse together all contexts that agree on the last k procedure calls. This is called *k-limiting* context-sensitive analysis, also known as k-CFA.

4.2 Abstracting cycles

Because of recursion, the last k calls may include a recursive cycle. The precise number of recursive calls typically doesn't add information, so another approach to context sensitivity is to consider all call strings to be equivalent if they agree when repeated recursive cycles are removed. So the call strings $hgf, hggf, hgggf$ would all be considered as one context hg^*f. Analysis in this context entails having the CFGs of h, g, and f wired together in a way that loses information about whether calls to g are coming from h or from g itself.

5 Using summaries

We would like to avoid rerunning a full program analysis of a given procedure for each calling context. This may be accomplished by computing a *summary* of the way the procedure affects information. If output information can be summarized as a simple function of the input information, the summary can be used instead of the corresponding dataflow analysis.

The summary is typically parameterized on the input information, which often requires making the underlying analysis more general so that it can be parameterized.

For example, summarizing the behavior of the `newArray` procedure, we can see that when it is called in a context c, it performs an allocation which we might name $alloc_{\texttt{newArray}/c}$. The result of analyzing the procedure is the mapping $\{x \mapsto alloc_{\texttt{newArray}/c}\}$. Note that to summarize the procedure accuately, we generalize the points-to analysis to allow heap locations to be indexed by contexts. Given the two calls to `newArray` from program points we might name f_1 and f_2, applying the summary to these context yields the mappings $\{\texttt{x} \mapsto alloc_{\texttt{newArray}/f_1}\}$ and $\{\texttt{x} \mapsto alloc_{\texttt{newArray}/f_2}\}$. Therefore the analysis of f determines points-to information for `a` and `b` as $\{\texttt{a} \mapsto alloc_{\texttt{newArray}/f_1}, \texttt{b} \mapsto alloc_{\texttt{newArray}/f_2}\}$, so the two variables are not aliases.

6 Object-sensitive analysis

The idea of object-sensitive analysis is to characterize the past history in terms of the chain of allocation dependencies. For object-oriented code, in particular, object sensitivity tends to be more effective than call-site sensitivity. Instead of representing the context as a sequence of call sites, the context is represented as a sequence of allocation sites.

7 Solving equations with fixed points

Many problems can be solved iteratively, and we have seen some of them during this course: dataflow analysis, loop-invariant expressions, LL(1) grammar construction. Type inference and information flow analysis are some other examples.

The common feature of all these problems is that they can be expressed as a system of n equations of n variables $x_1, \ldots, x_n$, whose values are elements drawn from some lattice L with finite height h and top element $\top$. The equations come in one of the following two forms:

$$\begin{aligned} x_1 &= f_1(x_1, \ldots, x_n) \\ x_2 &= f_2(x_1, \ldots, x_n) \\ &\vdots \\ x_n &= f_n(x_1, \ldots, x_n) \end{aligned} \qquad\qquad \begin{aligned} x_1 &\sqsubseteq f_1(x_1, \ldots, x_n) \\ x_2 &\sqsubseteq f_2(x_1, \ldots, x_n) \\ &\vdots \\ x_n &\sqsubseteq f_n(x_1, \ldots, x_n) \end{aligned}$$

where, in addition, the functions f_i are monotonic. Given these constraints, we know that we can use iterative analysis to find maximal solutions to equations of either kind. Because the solution is maximal, the solution iterative analysis finds to one of these sets of equations (using $=$ or $\sqsubseteq$) must also solve the corresponding set of equations of the other kind ($\sqsubseteq$ or $=$, respectively). Therefore we will simply assume the equations use equality.

The simple iterative analysis algorithm uses the equations to evaluate x_1, x_2, x_3, and so on up to x_n, and then repeats until convergence. However, for many problems, we can speed up convergence asympotically by evaluating the equations for the x_i in a different order.

The key to speeding up analysis is to observe that for a given system of equations, not all the functions f_i depend on all the variables $x_1, \ldots, x_n$. For any system of equations, we can draw a dependency graph on variables, where each variable is a node and there is an edge from x_i to x_j if f_j depends directly on x_i. For dataflow analysis, this dependency graph is essentially the same as the control flow graph.

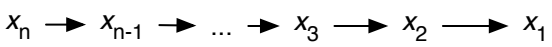

Figure 4: A simple dependency graph

7.1 Example

Consider the dependency graph shown in Figure 4. When the equations are applied in the order $1, \ldots, n$, it can take $n - 1$ iterations to propagate information all the way from x_n to x_1, or total time $O(n^2)$.

On the other hand, if the variables were evaluated in reverse order, convergence would be achieved after just one pass, or $O(n)$ time.