1 Why alias analysis?

Two memory locations $[e_1]$ and $[e_2]$ are aliases if e_1 and e_2 evaluate to the same value. If we have two memory operations, where at least one of them is a write, they can be safely reordered if the memory operands are not aliases. There are various reasons, we want to be able to determine whether two locations might be aliases:

- *Instruction selection*. If we can identify possible aliases, we can convert quadruples into larger expression trees, because we can change the order in which expressions accessing memory are computed. This simplifies optimization.
- *Instruction scheduling*. Memory accesses tend to be slow, so we would like to schedule them early in the instruction stream if possible. This will work well only if we can rule out aliasing.
- *Redundancy elimination*. Elimination of redundant memory operands is more effective if we can conservatively identify possible aliases.

These problems all involve solving the *may-alias* problem. A solution will be considered if all actual aliases are identified as possible aliases.

It is also possible to do a *must-alias* analysis, which enables other optimizations such as replacing one memory operand with another one, but this is not of as general utility.

2 Aliasing heuristics

Some simple heuristics are fairly effective at identifying possible aliases. Many memory operands are stack locations. Since the frame pointer register fp is constant throughout a given function, operands $[fp + i]$ and $[fp + j]$ are aliases only if $i = j$. A stack location also can never alias a non-stack location, and in many languages, stack locations and non-stack locations can be identified statically.

Some locations are immutable, such as the memory location that stores the length of an array. If we know there can't be any writes to such a location, we don't have to worry about aliases. The compiler can keep track of which locations are immutable and propagate that information to lower-level representations such as IR or abstract assembly.

3 Points-to analysis

The usual way to solve the general alias analysis problem is as a *points-to* analysis, and the terms *alias analysis* and *points-to analysis* are sometimes used interchangeably, though points-to analysis is really one way to solve alias analysis.

The idea is analyze which locations each pointer can point to. Locations can be on the heap or the stack. Since we can't in general predict at compile which locations will exist at run time, we abstract the set of storage locations in some way, typically as the set of allocation sites in the program. For example, if the program contains an expression `new C(...)`, all locations allocated by this expression might be mapped onto a single abstract storage location. Two pointers that can point to different allocations made using this expression will according to the analysis point to the same abstract location, and will be treated as aliases. (We will see later that taking interprocedural context into account can improve the precision of this analysis.)

3.1 Inclusion-based vs. Unification-based

One basic choice in designing a pointer analysis is whether it should be *inclusion-based* or *unification-based*. In an inclusion-based analysis, a pointer can point to a set of abstract locations, and two pointers may be aliases if they both can point to some abstract location. In a unification-based analysis, pointers are placed into equivalence classes; if pointer p can point to something that pointer q can, they are both in the same equivalence class. Unification-based analyses more directly solve alias analysis, but they lose precision. With an inclusion-based analysis it is possible for p to alias q and for q to alias r, but have p not alias r. In an unification-based analysis, this is not possible.

3.2 Flow-sensitive vs. flow-insensitive analyses

Another choice for pointer analysis (and, indeed, for other analyses) is whether the analysis should be *flow-sensitive* or *flow-insensitive*. In a flow-sensitive analysis, different points-to information is computed for each program point, whereas for a flow-insensitive analysis, points-to information is merged across all program points. The dataflow analyses we have been looking at are almost all flow-sensitive. Two flow-insensitive analyses we saw were sparse conditional constant propagation (using SSA form) and loop-invariant expressions analysis. Another example of a flow-insensitive program analysis is type checking, because it computes a single type for each variable that does not change at different program points. It's possible to have a flow-sensitive type systems, and in fact, the Xi type system has a flow-sensitive flavor in the way a statement generates an updated typing context Γ.

Flow-sensitive analyses can compute more precise information about the program, which is important for many analyses. A flow-insensitive liveness analysis, for example, would be useless. However, flow-sensitive analyses are also more expensive and, particularly if run as whole-program analyses, may not scale up to large programs, though there has been progress in recent years at making flow-sensitive analyses more scalable.

3.3 Analysis as abstraction

For simplicity, let's consider a flow-*insensitive* analysis, so we will compute just one heap that abstracts not only over all objects allocated by each allocation point, but also over all program points. The points-to analysis is a kind of *abstract interpretation* in which we imagine running the program, but with program state projected according to the abstraction onto a simpler, finite representation. This projection allows us to simplify the infinite number of heap structures that can happen in real executions down to a finite heap representation that conservatively represents all possible run-time heaps, yet contains enough information to be useful.

3.4 Example

Figure 1 shows a small example that can create aliases. There are three variables, `x`, `y`, and `z`, which can point to various heap-allocated objects. There are three allocation points, creating three abstract locations that we will call *alloc1*, *alloc2*, and *alloc3*. The loop on the right constructs an arbitrarily long linked list two elements at a time, so the possible run-time heap structures are infinite. Which of `x`, `y`, and `z` can be aliases after this code executes?

The analysis of this CFG is depicted in Figure 2. After executing the first two blocks, we know that `x`, `y`, and `z` can point to *alloc2*, *alloc3*, and *alloc1* respectively. Now consider the loop. It makes `alloc2.n` point to `y`, which means we add edges from `alloc2.n` to everything `y` might be pointing to (only *alloc3*). Then, `y.n=z` makes `alloc3.n` point to *alloc1*. And `z=x` means that `z` now points to the same things that `x` can point to: *alloc2*. Repeating the loop, `y.n = z` makes `alloc3.n` point to to *alloc2*, as shown in blue. At this point the loop analysis converges.

Considering the left branch, assigning `x.n=z` adds the green arrow. The assignment `y.n=x` adds no edges because `y.n` already points to everything `x` can. And the assignment `z=y.n` similarly adds no edges. The end result is that `x` can point to *alloc2*, `y` can point to *alloc3*, and `z` can point to either *alloc1* or *alloc2*.

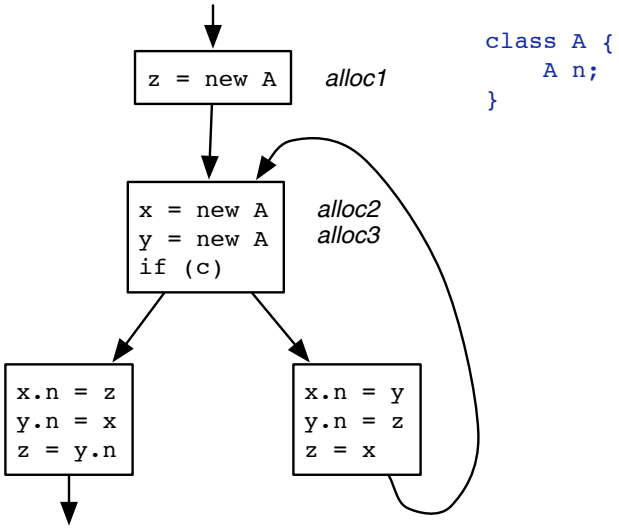

Figure 1: An example with aliasing

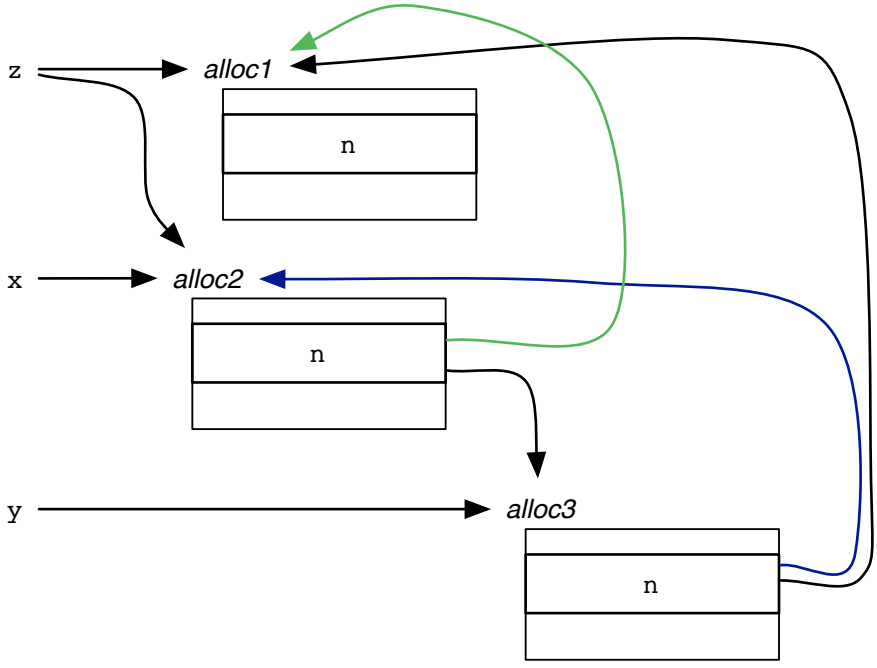

Figure 2: A flow-insensitive points-to analysis

For each variable v, the analysis computes a set of locations $Ptr(v)$ that v might point to. A memory location $[v_1]$ does not alias $[v_2]$ as long as $Ptr(v_1) \cap Ptr(v_2) = \emptyset$. So `x` and `z` might be aliases, since they both can point to abstract location *alloc2*, but no other pair of variables can be.

3.5 Transfer functions

Now let's formalize a flow-sensitive inclusion-based analysis. There is a set of abstract objects (or heap locations) $o \in H$. These may be allocation points, addresses taken of variables (e.g., with the C `&` operator), or parameters that are inputs to the function. [1]

There is also a set of variables V, which include both regular variables like `x` but also fields such as `o.n` where `o` is an abstract object.

The dataflow values are possible bipartite graphs in which all edges go from a variable to a heap location. That is, the values are binary relations, subsets of $V \times H$. Given a binary relation $l \subseteq V \times H$, we give the notation $l(v)$ the (standard) meaning $\{o \mid (v, o) \in l\}$. The top value is the empty relation $\emptyset$, and the ordering $\sqsubseteq$ is $\supseteq$.

A points-to relationship exists after a node if it existed before the node and isn't killed by the node, or if it is generated by the node:

$$out(n) = (in(n) - kill(n)) \cup gen(n, in(n))$$

To complete this transfer function, the functions $kill()$ and $gen()$ are defined as follows:

n	$kill(n)$	$gen(n, \ell)$
$x = q$	$\{x\} \times H$	$\{x\} \times \ell(q)$
$x = q.f$	$\{x\} \times H$	$\{x\} \times \ell(q.f)$
$x = q[i]$	$\{x\} \times H$	$\{x\} \times \ell(q.elem)$
$x =$ `new`$\ldots(alloc_i)$	$\{x\} \times H$	$\{(x, alloc_i)\}$
$x = \&y$	$\{x\} \times H$	$\{(x, alloc_y)\}$
$x.f = y$	$\emptyset$	$\{(o.f, \ell(y)) \mid o \in \ell x\}$
$x = f(\ldots)$	$\{x\} \times H$	$\{x\} \times H'$ (where H' is any location f can return)

This transfer function is monotonic but not distributive, because we can have a loss of information when merging graphs.

[1] Lack of knowledge about possible aliasing in function inputs will be a significant limitation on precision of the analysis, but the remedy is interprocedural analysis, which we have not yet covered.