CS 4120/5120 Lecture 29 Abstract interpretation and pointer analysis 12 April 2019
Lecturer: Andrew Myers

1 Abstract interpretation

We've seen that we can analyze the behavior of programs at compile time by using use elements of various lattices to represent facts about the execution of a program at run time. Sometimes the lattice elements have been sets ordered by relations $\subseteq$ and $\supseteq$; sometimes they have been more exotic structures. An example of the latter is the lattice elements used in conditional constant propagation (CCP). These lattice elements map each program variable to either a constant or the special values $\top$ and $\bot$.

As we've discussed program analyses, the tricky part has always been coming up with the correct transfer function F_n for each node n. The transfer function represents the change in information associated with executing n in a given program state.

Abstract interpretation, a technique pioneered by Patrick and Radhia Cousot in the late '70s, offers a principled way to design both the lattice over which dataflow analysis is performed and the transfer functions that operate on these lattice elements.

1.1 Abstraction and concretization

The key idea is to maintain a mapping between a more concrete representation of program execution, which corresponds directly to how programs execute, and a more abstract one that is amenable to compile-time program analysis. Both the concrete and abstract representations are lattices.

A common choice for the more concrete representation is a set of possible states of the program (such as the values of all variables and the structure of the heap), ordered by the superset relation $\supseteq$.[1] Thus, the most informative value $\top$ corresponds to the empty set of program states, and the least informative value $\bot$ corresponds to all possible states. It is also possible to include the execution history of the program as part of the concrete representation, which can be useful if we are trying to analyze using the past or future behavior of the program.

For example, consider CCP. For each possible program state, there is a "best" lattice element that describes that state. For example, suppose we have a program with two variables x and y. The state in which $x = 3$ and $y = 4$ is best represented by the lattice element $\{x \mapsto 3, y \mapsto 4\}$. The best lattice element to describe a *set* of states is simply the lattice meet of the best lattice representations of those states. For example, if we merge with a second control flow path in which $x = 3$ and $y = 5$, the corresponding lattice element is:

$$\{x \mapsto 3, y \mapsto 4\} \sqcap \{x \mapsto 3, y \mapsto 5\} = \{x \mapsto 3, y \mapsto \bot\}$$

Thus, the mapping to lattice elements in general loses information about the program. The mapping from the more concrete domain (e.g., sets of states) to the more abstract domain is called the *abstraction function* α. The *concretization* function γ maps from the abstract domain to the concrete domain, with the property that the concretization of the abstraction of a concrete element S is no more informative than the original element and the abstraction of the concretization of an abstract element a is the same abstract element:

$$\gamma(\alpha(S)) \sqsubseteq S$$
$$\alpha(\gamma(a)) = a$$

Further, both α and γ must be monotonic, so $S_1 \sqsubseteq S_2 \Rightarrow \alpha(S_1) \sqsubseteq \alpha(S_2)$ and $a_1 \sqsubseteq a_2 \Rightarrow \gamma(a_1) \sqsubseteq \gamma(a_2)$. In other words, making the input to either of these functions more precise cannot harm the precision of their output.

[1]In the abstract interpretation literature, the lattices are turned upside down, so that the "most informative dataflow value" is $\bot$, corresponding to the empty set of program states, and the least informative value is $\top$, corresponding to all possible states. We will keep the ordering consistent with the dataflow analyses discussed thus far.

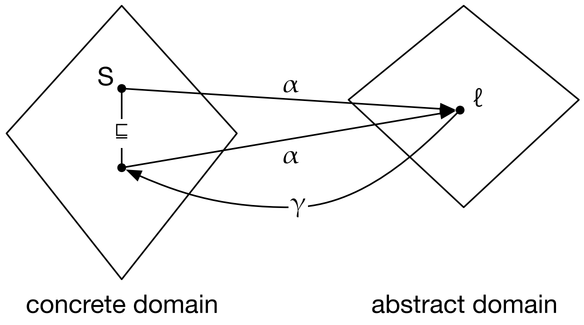

Figure 1: Abstraction and concretization functions

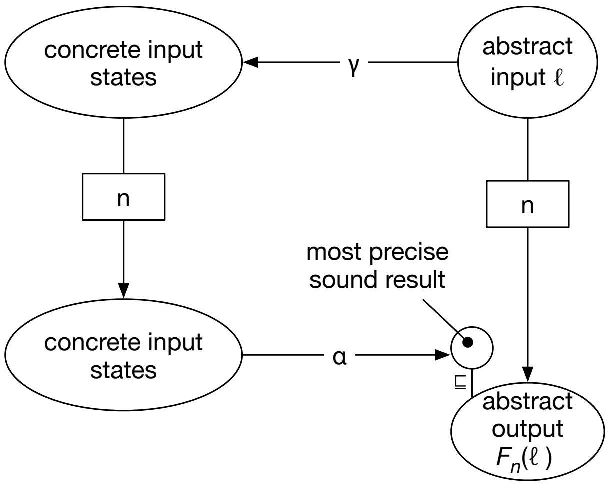

Figure 2: Transfer function as an abstraction of execution

These relationships make α and γ a *Galois insertion*, depicted in Figure 1.[2]

For example, both the set of all states where $x = 3$, and the set of states where $x = 3$ and either $y = 4$ or $y = 5$, map via α to the abstract element $\{x \mapsto 3, y \mapsto \bot\}$. To preserve the property $\gamma(\alpha(S)) \sqsubseteq S$, the concretization of this element must include at least the set of all states where $x = 3$, and the most precise concretization would be exactly this set; that is, we choose the concretization $\gamma(a) = \bigcap\{S \mid \alpha(S) \sqsupseteq a\}$

1.2 Designing transfer functions

We'd like to construct transfer functions that capture conservatively what the original program would have done, but by "computing" on abstract elements. We can construct the most precise such transfer function by using the abstraction function α to map the inputs and outputs of the actual computation into the abstract space. Suppose that the actual computation at node n maps a given set of states S to a set $S' = f_n(S)$. Then, given an input element ℓ, we can map it to a set of states $\gamma(\ell)$ and apply the effect of the CFG node n to it to get a resulting set of states $f_n(\gamma(\ell))$. Applying the abstraction function to this gives us a sound transfer function: $\alpha(f_n(\gamma(\ell)))$. Intuitively, this transfer function, which operates downward on the right-hand side of the figure, is constructed by instead following arrows around the figure counter-clockwise.

If γ is defined to be the most precise concretization describe above, this transfer function will also be as precise as possible. We don't necessarily need this most precise transfer function in order to have an effective program analysis, however. All that is required is that the result $F_n(\ell)$ is no more informative than

[2]It is also possible to define α and γ as a *Galois connection*, which weakens the second statement to an inequality, but by only adding useless elements to the abstract domain.

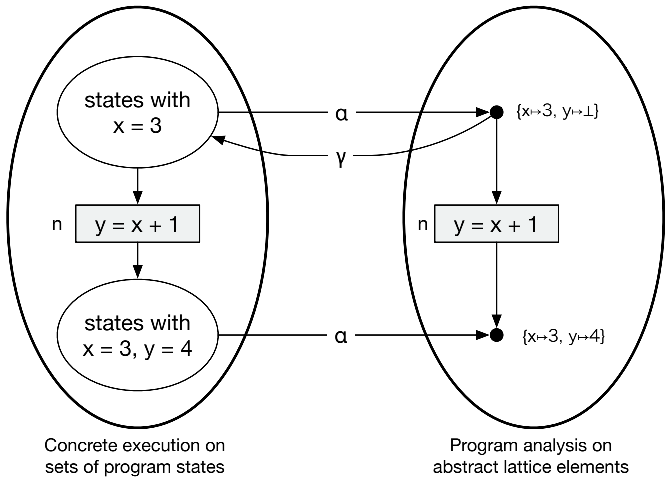
states with
x = 3
α
γ
{x↦3, y↦⊥}
n
y = x + 1
n
y = x + 1
states with
x = 3, y = 4
α
{x↦3, y↦4}
Concrete execution on sets of program states
Program analysis on abstract lattice elements

Figure 3: Example transfer function

this most precise result:

$$F_n(l) \sqsubseteq \alpha(f_n(\gamma(\ell))$$

For example, suppose the abstract input is $\ell = \{x \mapsto 3, y \mapsto \bot\}$ and the node n is the assignment y=x+1. The input set of states $\gamma(\ell)$ is all states for which $x = 3$. Running this statement on these states, as depicted in Figure 3 gives us all states for which $x = 3$ and $y = 4$. Mapping this back via α, we have $F_n(\ell) = \{x \mapsto 3, y \mapsto 4\}$, which is exactly what we'd want. On the other hand, if we defined $F_n(\ell)$ to be $\{x \mapsto \bot, y \mapsto 4\}$ in this case, that would still be sound—just less precise that we might like.

The program analyses associated with alias analysis are a good chance to think about program analysis in terms of abstract interpretation.

2 Why alias analysis?

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. For 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.

2.1 Aliasing heuristics

Some simple heuristics are fairly effective at identifying possible aliases. Many memory operands are stack locations. Since the frame pointer fp is constant throughout a given function, operands $[\mathtt{fp} + i]$ and $[\mathtt{fp} + j]$

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 Pointer analysis

The usual way to solve the general alias analysis problem is as a *pointer analysis*, and the terms *alias analysis* and *pointer analysis* are sometimes used interchangeably, though pointer analysis can be used as a way to solve alias analysis but has other uses as well.

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, introduced by Andersen, 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, introduced by Steensgard, 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, and are more efficient, 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 is flow-sensitive: each statement generates an updated typing context Γ that is used by the next statement.

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

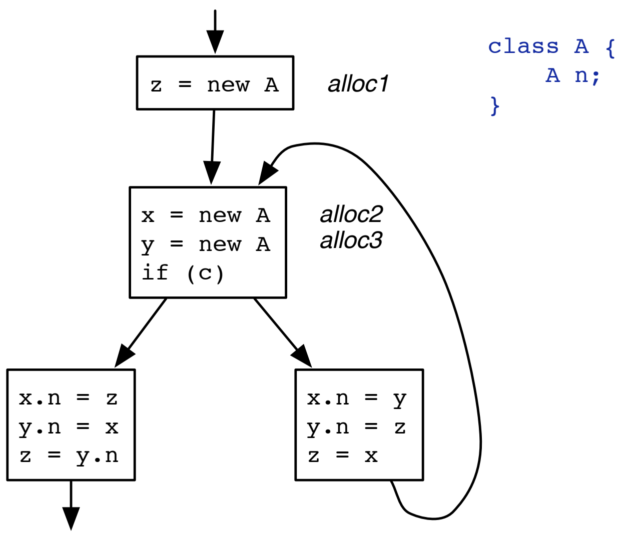

Figure 4: An example with aliasing

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 4 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 5. 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 arrows from `x.n` to everything that `z` can point to. This gives us the green arrows to *alloc1* and *alloc2*. 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*.

Thus, though there are an infinite number of possible run-time heap structures, the analysis computes a finite representation of the heap that includes all the pointers that are in any possible run-time structure.

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.

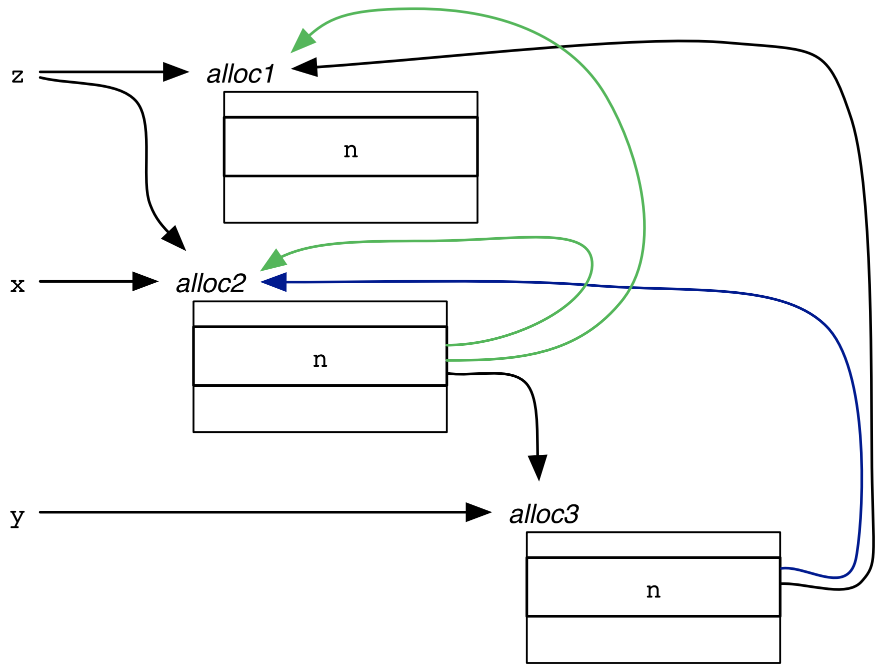

Figure 5: A flow-insensitive points-to analysis

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. [3]

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 and `n` is the name of a field.

The dataflow values are 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 use the notation $l(v)$ the (standard) meaning of the image of v under l, defined as $\{o \mid (v, o) \in l\}$. The top value is the empty relation $\emptyset$, and the ordering $\sqsubseteq$ is $\supseteq$, since the conservative outcome is to find too many points-to relationships.

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. It helps to think about this as an abstract interpretation.

[3]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.

n	$kill(n)$	$gen(n, \ell)$
$x = y$	$\{x\} \times H$	$\{x\} \times \ell(y)$
$x = y.f$	$\{x\} \times H$	$\{x\} \times \{o \mid (y, o') \in \ell \wedge o \in \ell(o'.f)\}$
$x = y[i]$	$\{x\} \times H$	$\{x\} \times \{o \mid (y, o') \in \ell \wedge o \in \ell(o'.\texttt{elem})\}$
$x = \texttt{new} \ldots (alloc_i)$	$\{x\} \times H$	$\{(x, alloc_i)\}$
$x = \&y$	$\{x\} \times H$	$\{(x, addr_y)\}$
$x.f = y$	$\emptyset$	$\{o.f \mid o \in \ell(x)\} \times \ell(y)$
$x = f(\ldots)$	$\{x\} \times H$	$x \times H' \cup \ell'$ (where H' is any location f can return)
		(and ℓ' contains all the pointers that f might create.)

Notice that for simple assignments to a variable x, we can kill all existing pointers from x. But for an assignment to $x.f$, we can't kill anything. This is because the assignment only affects one concrete object, but even if x only points to one abstract object, that abstract object might represent multiple real objects.

The rule for function calls has to be quite conservative. First, without extra knowledge about the function being called, it could return a pointer to any abstract object. Second, it may be able to update the heap in an arbitrary way to add new pointers. There are two ways we may be able to improve the precision. One is to use an *escape analysis* to prove that certain parts of the heap are inaccessible to function f and hence no pointers to or from them can be created. Second, we can use an *interprocedural* analysis to get precise information about f's side effects and return value. We will talk about such analyses soon.

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

It is also possible to use the same rules in a flow-insensitive way. This amounts to abstracting away which edge we're talking about collecting information for. The same rules are used, but we can no longer "kill" pointers; $kill(n)$ is treated as $\emptyset$.