Flow-Insensitive Pointer Analysis
drawbacks of flow-sensitivity

• Flow-sensitive pointer analysis is expensive

• Keep track of a different points-to graph at each program point
  
  • Can take a while for analysis to converge

• Large storage requirements for large programs
  
  • Can do flow-insensitive analysis instead
flow-insensitive analysis

• Key idea: ignore control flow

• All statements in a function dumped into a single list

  • No loops, branching

• Compute a single points-to graph that is valid for the entire function regardless of when and how often statements are executed
weak updates

• Because we are computing a single points-to graph, and we do not know when a given statement executes, we never remove information.

• Replace all strong updates with weak updates.

• Update graph in place.

\[ G' = G \text{ with } pt(x) \cup \{y\} \]
Andersen’s algorithm

• Compute points-to graph by formulating this as a series of set constraints

• Solve set constraints (same fix point algorithms we’ve seen before!)

• Only trick: when points-to sets are updated, loads and stores generate new constraints!

```
 address of x = & y  y ∈ pt(x)
 copy x = y           pt(x) ⊇ pt(y)
 load x = * y          ∀ a ∈ pt(y). pt(x) ⊇ pt(a)
 store * x = y         ∀ a ∈ pt(x). pt(a) ⊇ pt(y)
```
• What if you have multiple functions? Need to do \textit{interprocedural analysis}

• Simple approach we’ve seen before: assume a function can do \textit{anything}

• What can you do instead?
  
  • Execute \textit{interprocedurally}
  
  • Propagate points-to information from caller to callee, back from callee to caller
interprocedural analysis

• We won’t really cover this, but there are two basic approaches

• **Context-sensitive**: treat each function call separately, like in a real execution
  - Essentially, inline callee into caller
  - What do we do for recursion? Need to approximate
  - Pros: accurate. Cons: slow

• **Context-insensitive**: merge information from call sites of each function
  - Essentially, represent each function once in a control flow graph
  - Merge information from multiple callers within the callee
  - Pros: faster. Cons: inaccurate (information can flow from one caller to another!)
dealing with the heap

• What about heap allocations?

• Simple approach: one node represents “the heap.”
  • \( x = \text{malloc(…)} \) makes \( x \) point to “the heap.”

• More complicated approach: a different heap node for each malloc site

• Even more complicated: shape analysis to reason about how heap nodes point to each other