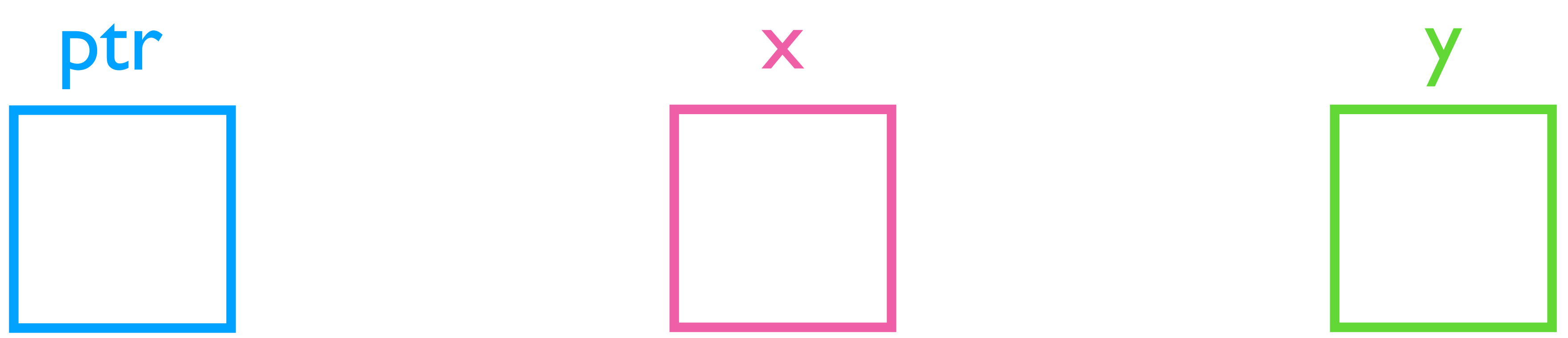
Pointer Analysis

analyzing programs with pointers

- Where is x defined?

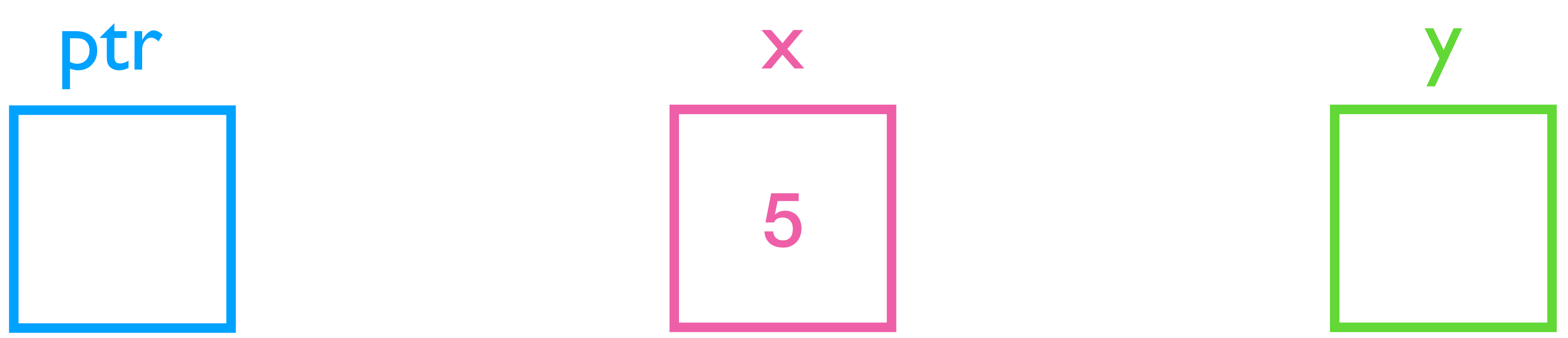
```
x    = 5
ptr  = &x
*ptr = 9
y    = x
```



analyzing programs with pointers

- Where is x defined?

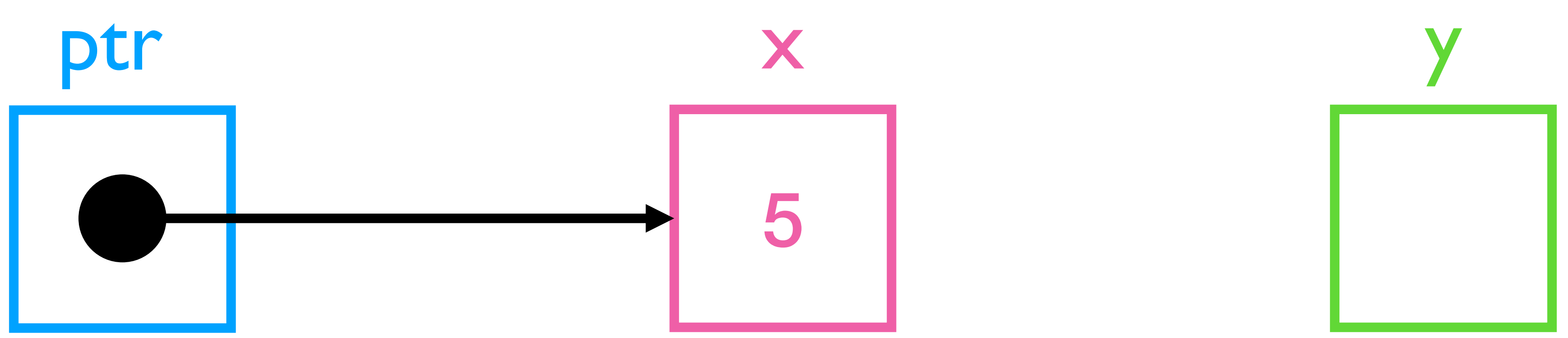
```
x    = 5
ptr  = &x
*ptr = 9
y    = x
```



analyzing programs with pointers

- Where is x defined?

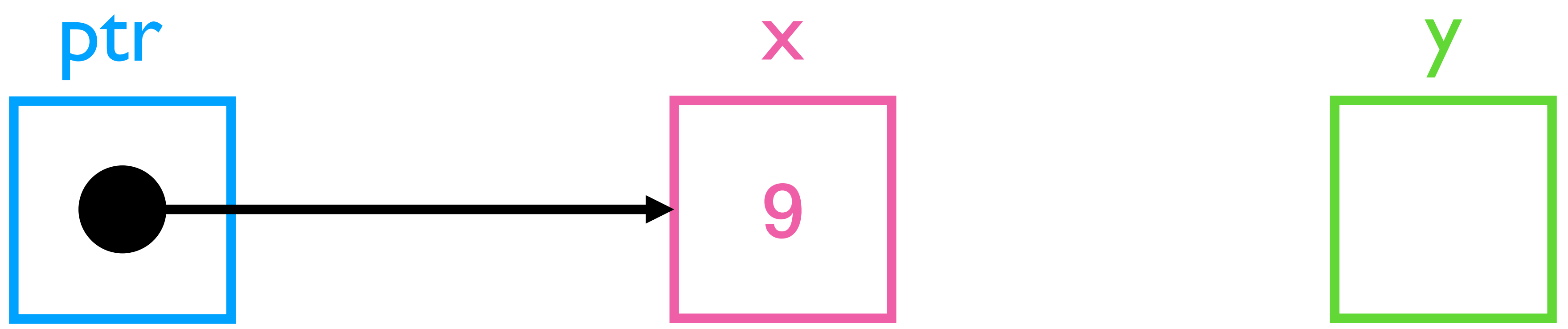
```
x    = 5
ptr  = &x
*ptr = 9
y    = x
```



analyzing programs with pointers

- Where is x defined?

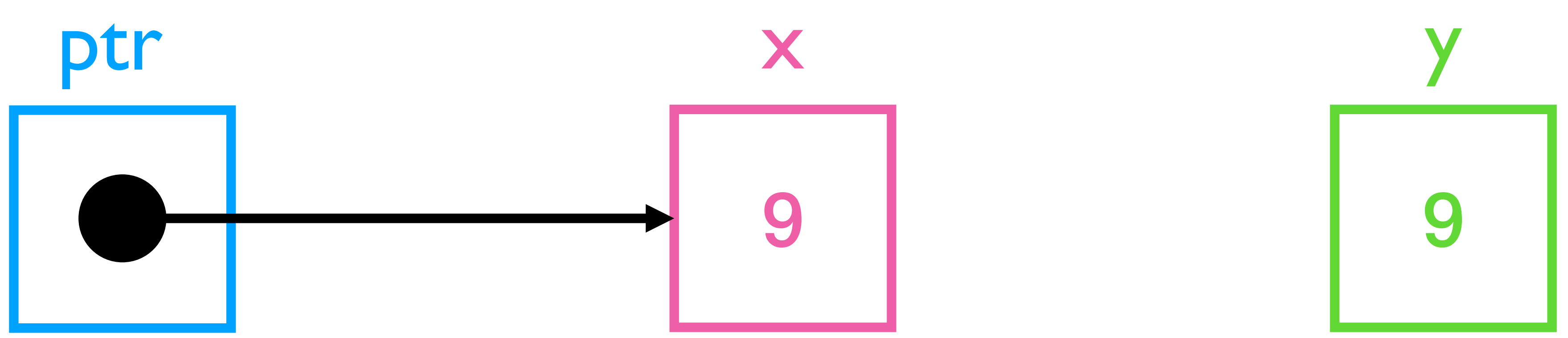
```
x    = 5
ptr  = &x
*ptr = 9
y    = x
```



analyzing programs with pointers

- Where is x defined?

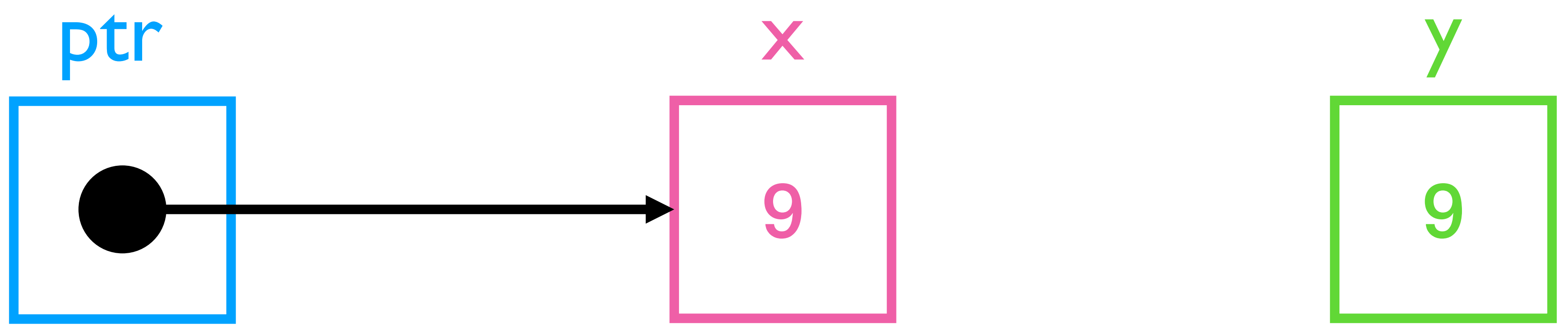
```
x    = 5
ptr  = &x
*ptr = 9
y    = x
```



analyzing programs with pointers

- Where is x defined?

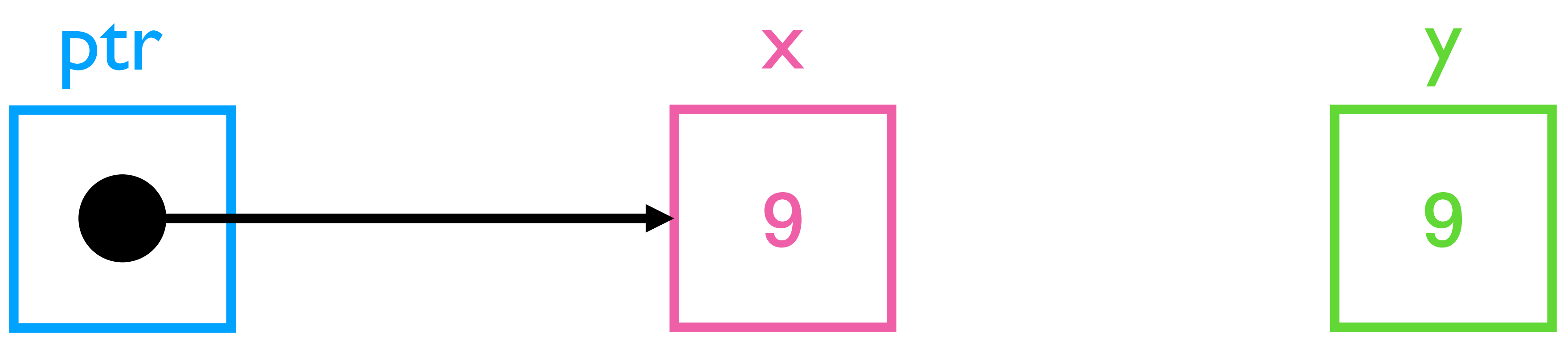
```
x    = 5
ptr  = &x
*ptr = 9
y    = x
```



analyzing programs with pointers

- Where is x defined?

```
x    = 5
ptr  = &x
*ptr = 9
y    = x
```



- Problem: just looking at variable names does not give you the right answer
    - Both *ptr and x talk about the same memory location (ptr points to x)
- Must know (or estimate) this points to information for correct analysis

program model

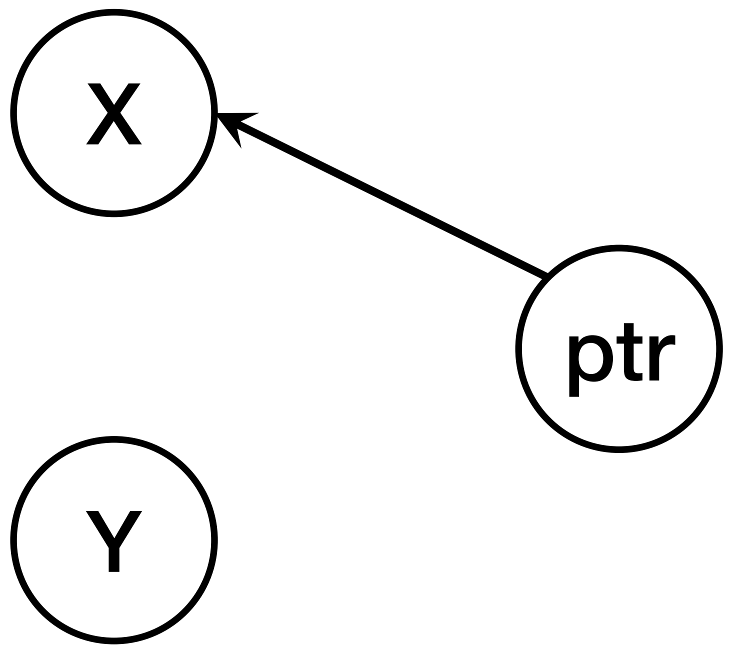
- For now, types are simple: base type is int, or pointer (*) to another type
- No function calls, no pointer arithmetic
- Statements using pointer variables

```
Address of:   x = &y
Copy:         x = y
Load:         x = *y
Store:        *x = y
```

- Arbitrary computations involving ints

points-to graph

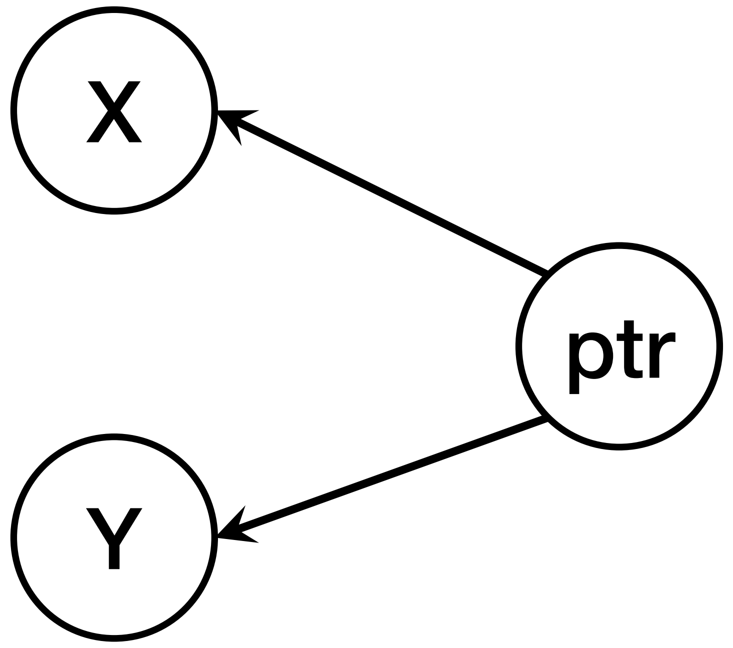
- What information do we track? **points-to graphs**
  - Nodes are program variables
  - Edges say “a points to b”
  - Can use a special node for NULL, a special node for “somewhere in the heap”
- Points-to graph can be different at different points



points-to graph

- Out-degree of a node can be more than one
  - Node with multiple outgoing edges says “a **may** point to b *or* c”
  - Represents uncertainty in the analysis
    - e.g., if more than one way to reach a program point

```
if (q)
   ptr = &x
else
   ptr = &y
//what does ptr point to?
```



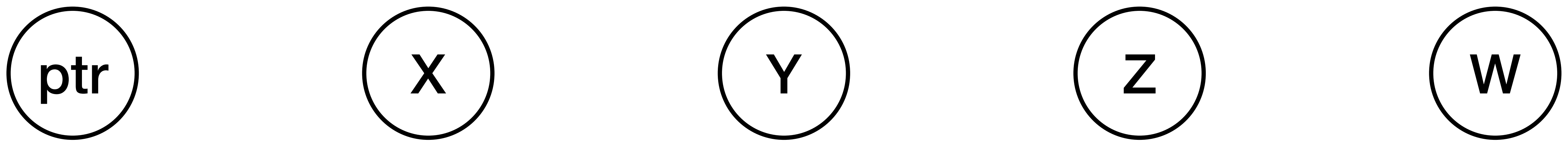
making a lattice

- To create a lattice, we need a $\bot$, a $\top$ and a $\sqsubseteq$
  - $\bot$ is "graph with no edges"
  - $\top$ is "graph with all nodes pointing to all other nodes"
- $G_1 \sqsubseteq G_2$ if and only if G_2 has all of the edges G_1 has, and maybe some more
- What about join ($\sqcup$) and meet ($\sqcap$)?
  - $G_1 \sqcup G_2$ = graph with the *union* of the edges in both graphs
  - $G_1 \sqcap G_2$ = graph with the *intersection* of the edges in both graphs

gameplan

- Two different kinds of pointer analyses
  - flow-sensitive: standard dataflow analysis --- what is the points-to graph at each point in the program?
  - flow-insensitive: simplification --- what if we construct a single points-to graph that is valid at all points in the program? (Overapproximates flow-sensitive result)

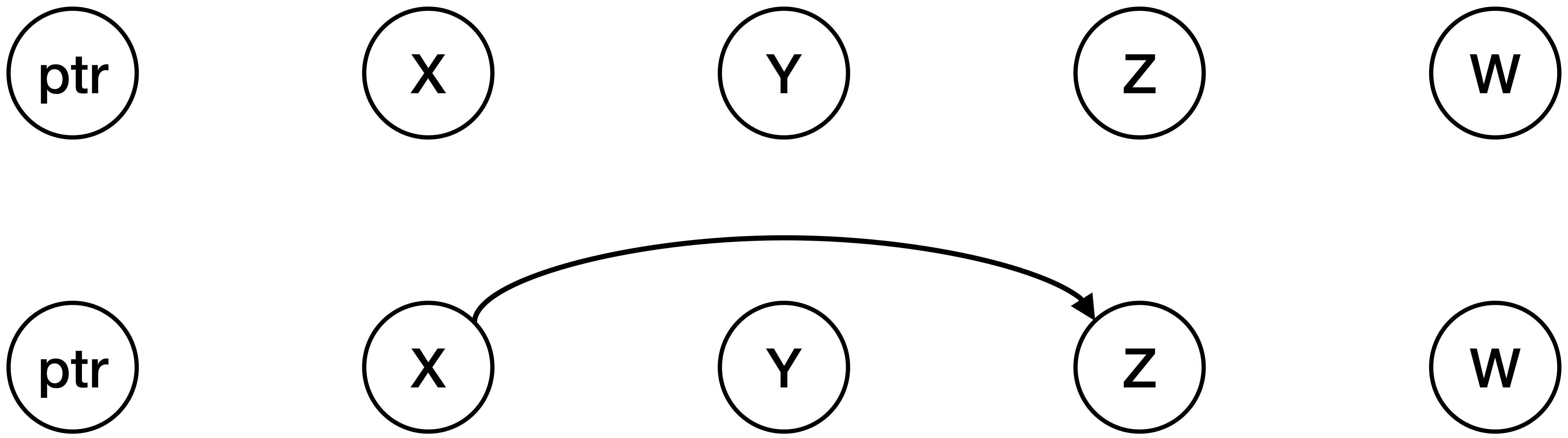
example: flow-sensitive




```
x   = &z

ptr = &x

y   = &w

ptr = &y
```

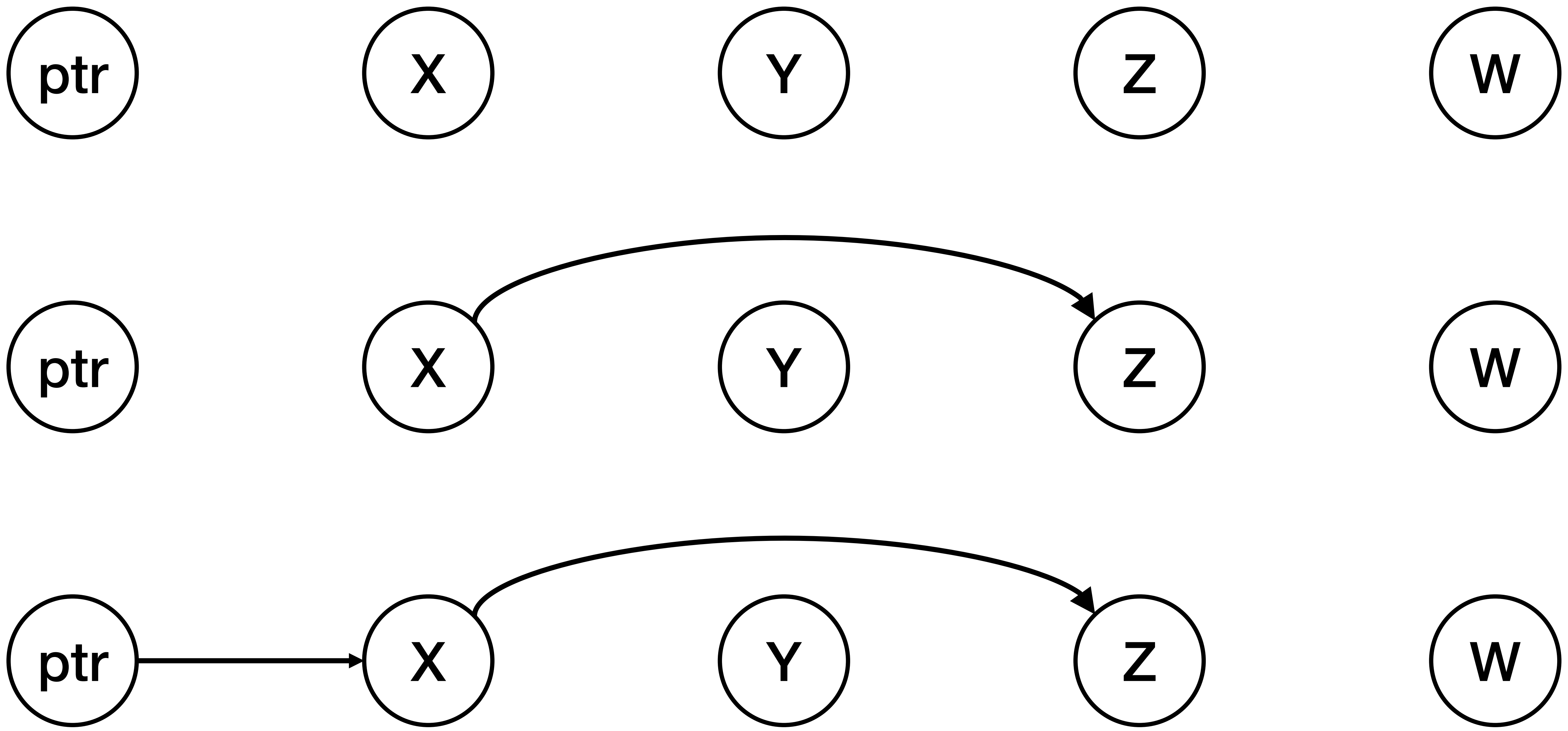
example: flow-sensitive

```
x   = &z

ptr = &x

y   = &w

ptr = &y
```



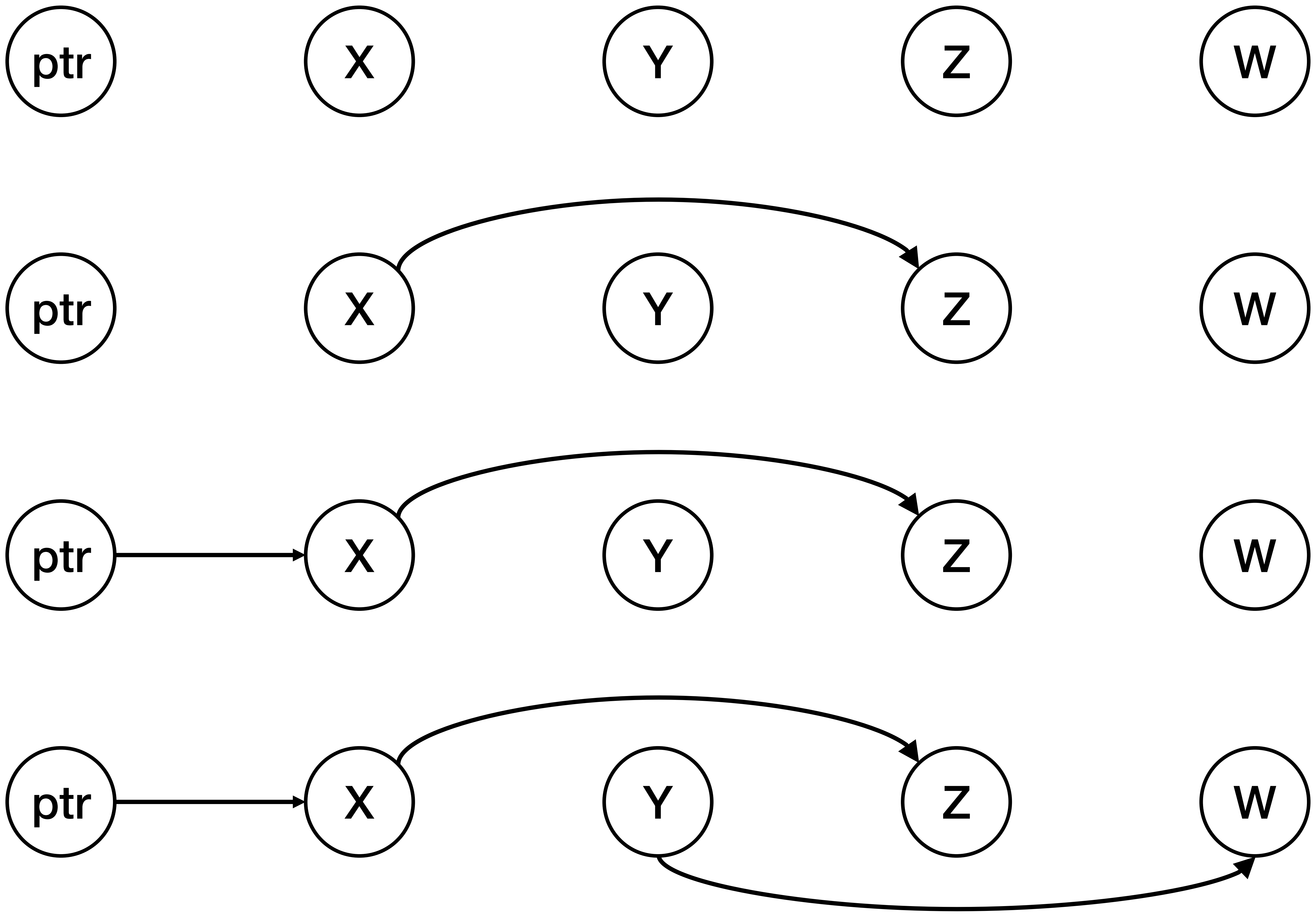
example: flow-sensitive

```
x   = &z

ptr = &x

y   = &w

ptr = &y
```



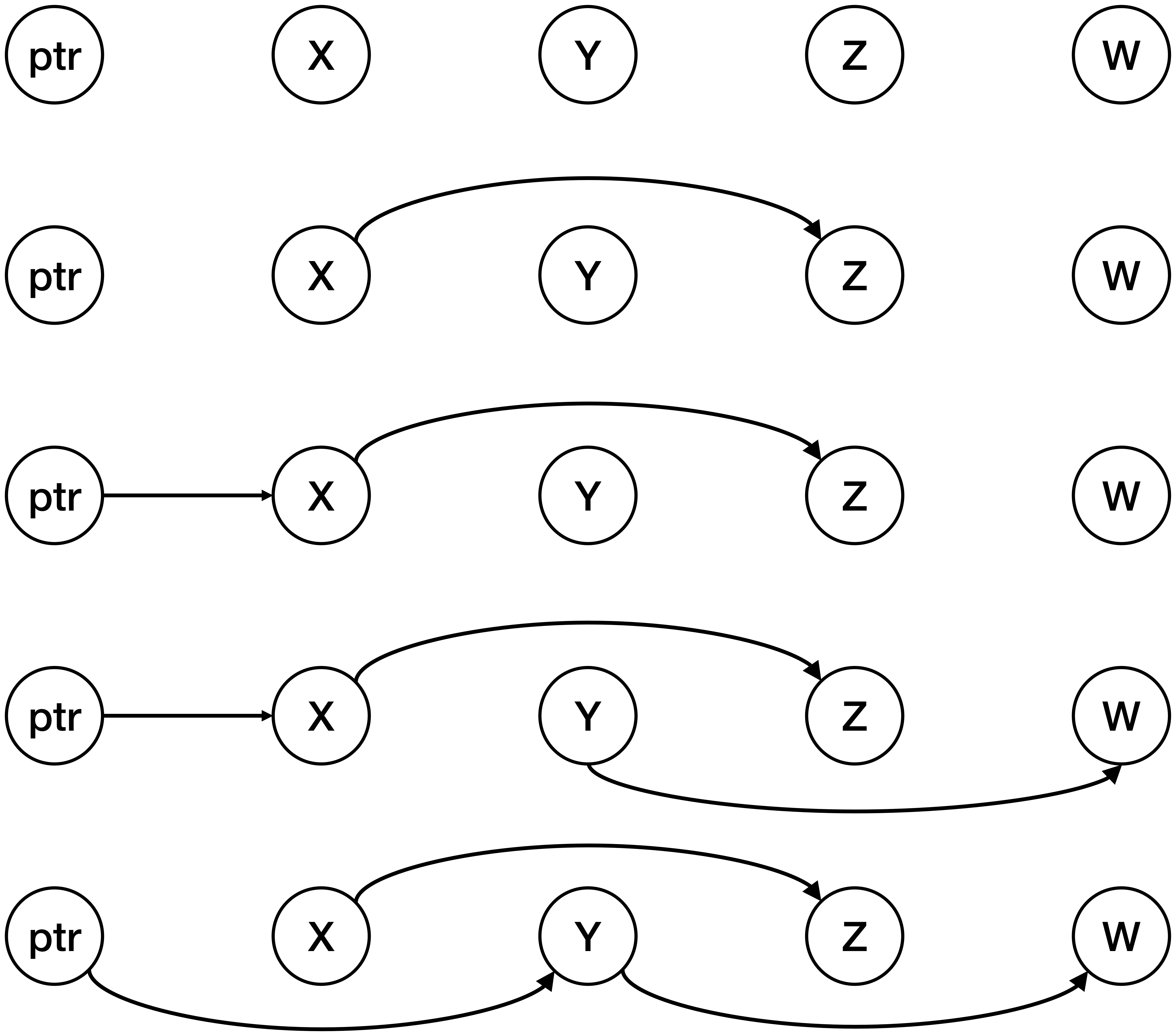
example: flow-sensitive

```
x   = &z

ptr = &x

y   = &w

ptr = &y
```



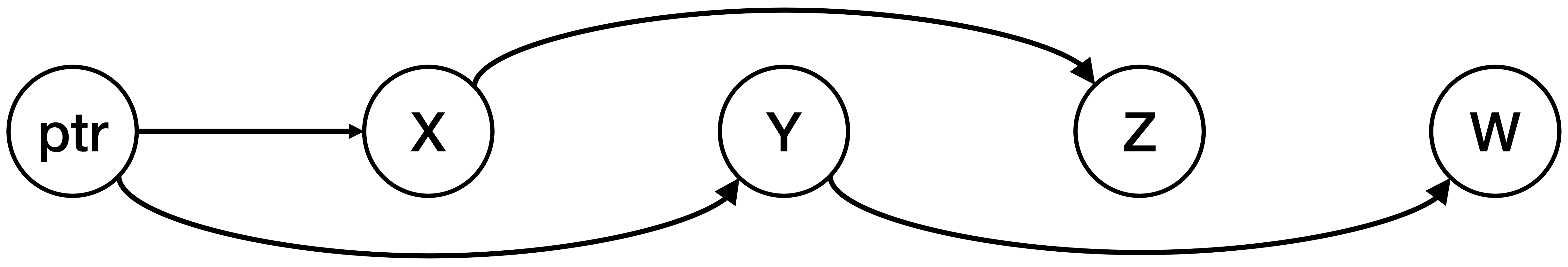
example: flow-sensitive

```
x   = &z
ptr = &x
y   = &w
ptr = &y
```



example: flow-insensitive

```
x   = &z

ptr = &x

y   = &w

ptr = &y
```



ptr points to x or y because we only have one points-to graph

next: flow-sensitive pointer-analysis