Advanced Static Analysis Technique

Woosuk Lee

CSE 6049 Program Analysis

Hanyang University, Korea

Some slides are borrowed from https://github.com/prosyslab-classroom/is593-2020-spring/blob/master/slides/lecture8.pdf

Goal of This Lecture

- Learn advanced techniques for more **precise** and **efficient** analysis

Combination of Multiple Abstractions

- Verifying a program often requires reason over multiple kinds of properties.

- We can construct abstract domains that can handle such cases.

Combination of Multiple Abstractions

- Suppose we have two abstract domains $\mathbb{A}_0, \mathbb{A}_1$ with concretization function $\gamma_0 : \mathbb{A}_0 \longrightarrow \mathscr{P}(\mathbb{M}), \gamma_1 : \mathbb{A}_1 \longrightarrow \mathscr{P}(\mathbb{M})$
- The *product abstraction* is defined by
  - $\mathbb{A}_\times = \mathbb{A}_0 \times \mathbb{A}_1$
  - Concretization function:

    $\forall (a_0, a_1) \in \mathbb{A}_\times,\ \gamma_\times(a_0, a_1) = \gamma_0(a_0) \cap \gamma_1(a_1)$

Combination of Multiple Abstractions

- Two abstract domains can characterize distinct kinds of properties.

- E.g., $\mathbb{A}_0$: interval domain, $\mathbb{A}_1$: parity domain

- Abstract value `([0, 100], Even)` describes even values between 0 and 100.

Synergistic Combination of Multiple Abstractions

- Multiple abstract domains may exchange information to derive stronger properties.
- e.g., Abstract value

  `([1,5], Even)`

  is concretized to `{2,4}`.

  The most precise abstract information is

  `([2,4], Even)`

Reduced Product

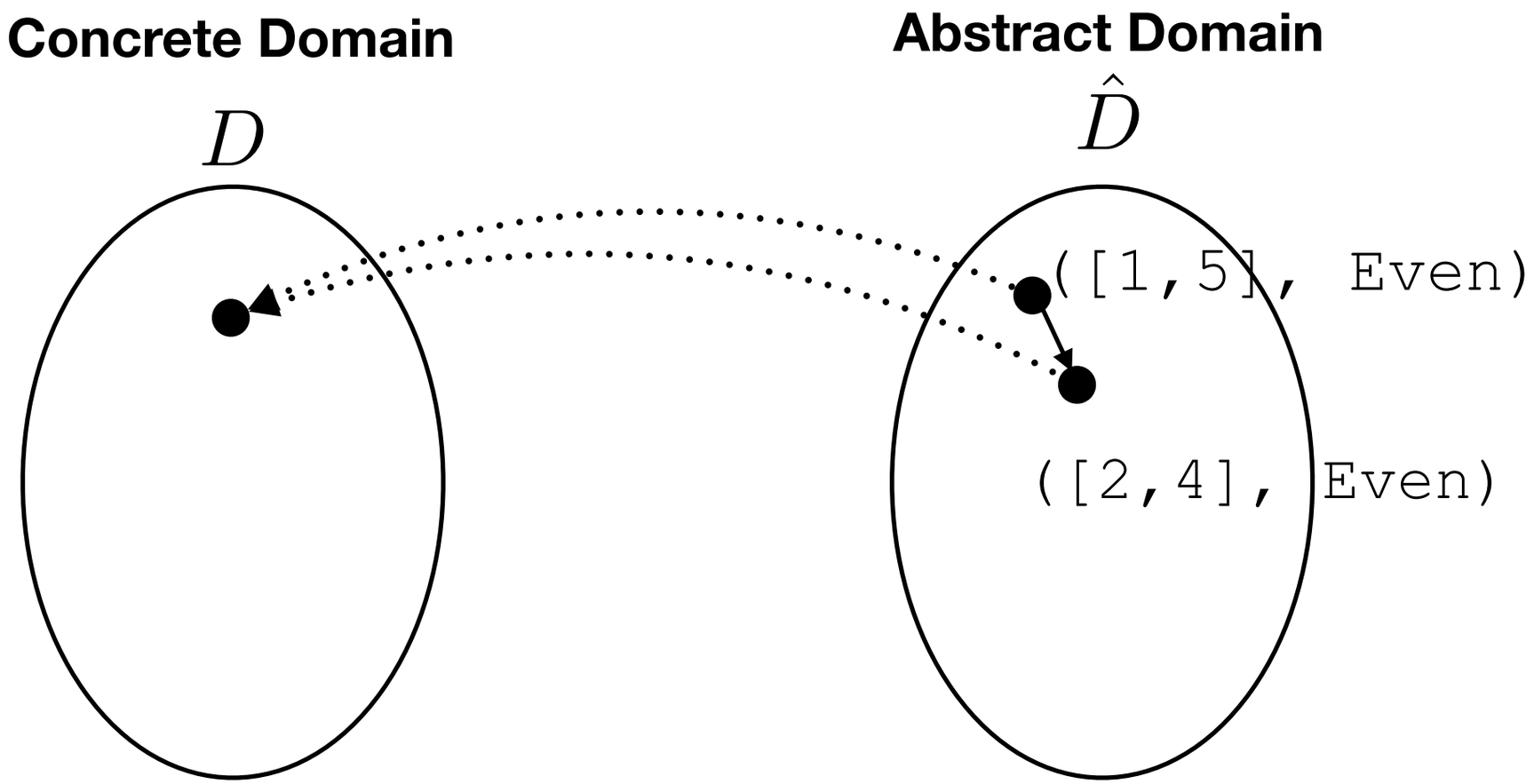
- The reduced product of abstract domains $\mathbb{A}_0, \mathbb{A}_1$ is the set $\mathbb{A}_{\bowtie}$ of equivalence classes of $\mathbb{A}_0 \times \mathbb{A}_1$ for the relation $\equiv$ defined by

$$(a_0, a_1) \equiv (a'_0, a'_1) \iff \gamma_\times(a_0, a_1) = \gamma_\times(a'_0, a'_1)$$

- **e.g.,** `([1,5], Even) ≡ ([2,4], Even)`

Reduction Operator

Reduction operator turns any abstract element into its optimal representation.



$$\rho : \hat{D} \to \hat{D} \text{ s.t. } \rho(d) \sqsubseteq d \wedge \gamma(\rho(d)) = \gamma(d)$$

$$\text{Ideally, } \rho(d) = \alpha(\gamma(d)) \quad (\because \alpha \circ \gamma \sqsubseteq id)$$

Reduction Operator

- Optimal reduction operator ($\alpha \circ \gamma$) is often very expensive.

- In practice, a sound but approximate reduction operation ($\alpha \circ \gamma \sqsubseteq \rho$) is often preferable.

- Reduction is typically performed only at specific points where it is known that it can provide a useful precision gain.

Advanced Iteration Techniques

- Loop invariant inference: sequences of abstract iterates

- Computes weaker and weaker abstract states until stabilization, hence a common source of imprecision (e.g., widening)

- There are various techniques to improve the precision of the static analysis of loops.

Review: Static Analysis of `while`$(B)\{C\}$

- Concrete semantic function: $F = [\![C]\!]_{\mathscr{P}} \circ \mathscr{F}_B$

- Sound abstract semantic function: $F \circ \gamma \subseteq \gamma \circ F^\sharp$

- $M^\sharp$: abstract pre-condition for the loop

- Semantics of the loop:

$$M_{\text{loop}} = \bigcup_{i \geq 0} F^i(\gamma(M^\sharp)) = \mathbf{lfp}\ G$$

where $G(X) = \gamma(M^\sharp) \cup F(X)$

Review: Static Analysis of Loops

- To ensure termination of the analysis, we use widening. To compute the converging sequence

$$\begin{aligned} M_0^\sharp &= M^\sharp \\ M_{k+1}^\sharp &= M_k^\sharp \triangledown F^\sharp(M_k^\sharp) \end{aligned}$$

- $M_{\text{lim}}^\sharp$: the limit of the sequence (after finitely many iterates)

- The analysis returns the sound loop post-condition $\mathscr{F}_{\neg B}^\sharp(M_{\text{lim}}^\sharp)$

Techniques for Improving Precision of Analysis of Loops

- Loop unrolling
- Delayed widening
- Widening with thresholds

Loop Unrolling

- Motivation: hasty join

```
x = ?; // any value
i = 1;
while (i > 0) {
  if(x < 0 || x > 1000) {
    x = 0;
  } else {
    x = x + 1;
  }
  input(i);
}
// actually, x is in [0, 1001]
```

Initialization step

- The abstract value for x with a naive approach would be [-∞, +∞]
- Idea: **detach** the first iteration from the rest

Loop Unrolling

```
x = ?; // any value
i = 1;
while (i > 0) {
  if(x < 0 || x > 1000) {
    x = 0;
  } else {
    x = 1 + x;
  }
  input(i);
}
// actually, x is in [0, 1001]
```

```
x = ?; // any value
i = 1;
if(x < 0 || x > 1000) {
  x = 0;
} else {
  x = 1 + x;
}
input(i);
// x is in [0, 1001]
while (i > 0) {
  if(x < 0 || x > 1000) {
    x = 0;
  } else {
    x = 1 + x;
  }
  input(i);
}
// x is in [0, 1001]
```

first iter.

rest

Loop Unrolling

- The previous sequence for the loop

$$\begin{aligned} M_0^\sharp &= M^\sharp \\ M_{k+1}^\sharp &= M_k^\sharp \mathbin{\triangledown} F^\sharp(M_k^\sharp) \end{aligned}$$

- With loop unrolling

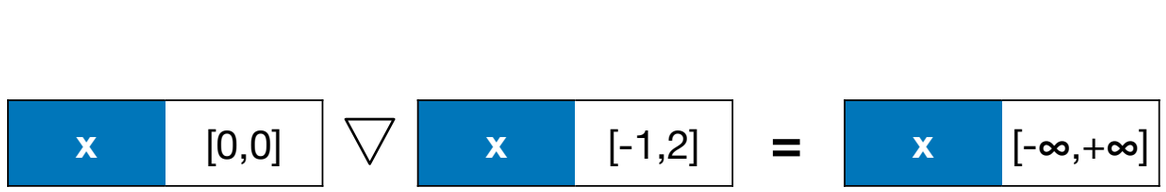
$$\begin{aligned} M_0^\sharp &= M^\sharp \\ M_{k+1}^\sharp &= \begin{cases} F^\sharp(M_k^\sharp) & \text{if } k < N \\ M_k^\sharp \mathbin{\triangledown} F^\sharp(M_k^\sharp) & \text{otherwise} \end{cases} \end{aligned}$$

and the analysis returns $\mathscr{F}^\sharp_{\neg B}(M^\sharp_{\text{lim}})$

Delayed Widening

- Motivation: hasty widening

```
x = 0;
while (rand()) {
  if(rand()) {
    x = -1;
  } else {
    x = x + 2;
  }
}
// x >= -1
```

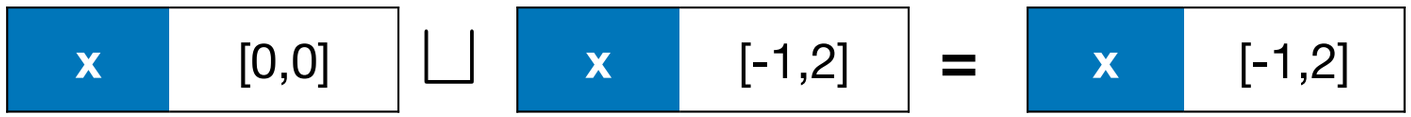


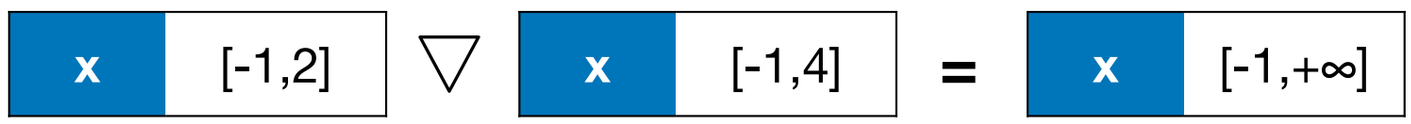
- The abstract value of x with a naive approach would be [-∞, +∞]
- Idea: **delay** the application of widening for the first N iterations

Delayed Widening

```
x = 0;
while (rand()) {
   if(rand()) {
      x = -1;
   } else {
      x = x + 2;
   }
}
// x >= -1
```

Delayed widening where N = 1





Fixed Point!

Delayed Widening

- The sequence with delayed widening:

$$
\begin{array}{lcll}
M_0^\sharp & = & M^\sharp & \\
M_{k+1}^\sharp & = & M_k^\sharp \sqcup^\sharp F^\sharp(M_k^\sharp) & \text{if } k \leq N \\
M_{k+1}^\sharp & = & M_k^\sharp \triangledown F^\sharp(M_k^\sharp) & \text{if } k > N
\end{array}
$$

- Loop unrolling: postpones **join** first N iterations

- Delayed widening: postpones **widening** first N iterations

Widening with Thresholds

- Motivation: the standard widening is too conservative!

```
x = 0;
while (x <= 100) {
  if(x >= 50) {
    x = 10;
  } else {
    x = x + 1;
  }
}
// actually, x is in [0, 50]
```

- The abstract value of x with a naive approach is [0, +∞]
- Idea: use a **slower and more precise** widening

Widening with Thresholds

- Take several small steps and stops at pre-defined threshold values
- For example, consider only one threshold *B*:

A naive widening operator

$$[n,p] \triangledown [n,q] = \begin{cases} [n,p] & \text{if } p \geq q \\ [n,+\infty] & \text{if } p < q \end{cases}$$

A widening with thresholds

$$[n,p] \triangledown [n,q] = \begin{cases} [n,p] & \text{if } p \geq q \\ [n,B] & \text{if } p < q \leq B \\ [n,+\infty] & \text{if } B < q \end{cases}$$

*only the right bounds, for brevity

Widening with Thresholds

```
x = 0;
while (x <= 100) {
   if(x >= 50) {
      x = 10;
   } else {
      x = x + 1;
   }
}
```

Thresholds = {50}

Fixed Point!

Scalability Challenge

The worklist algorithm often does not scale to large complex programs.

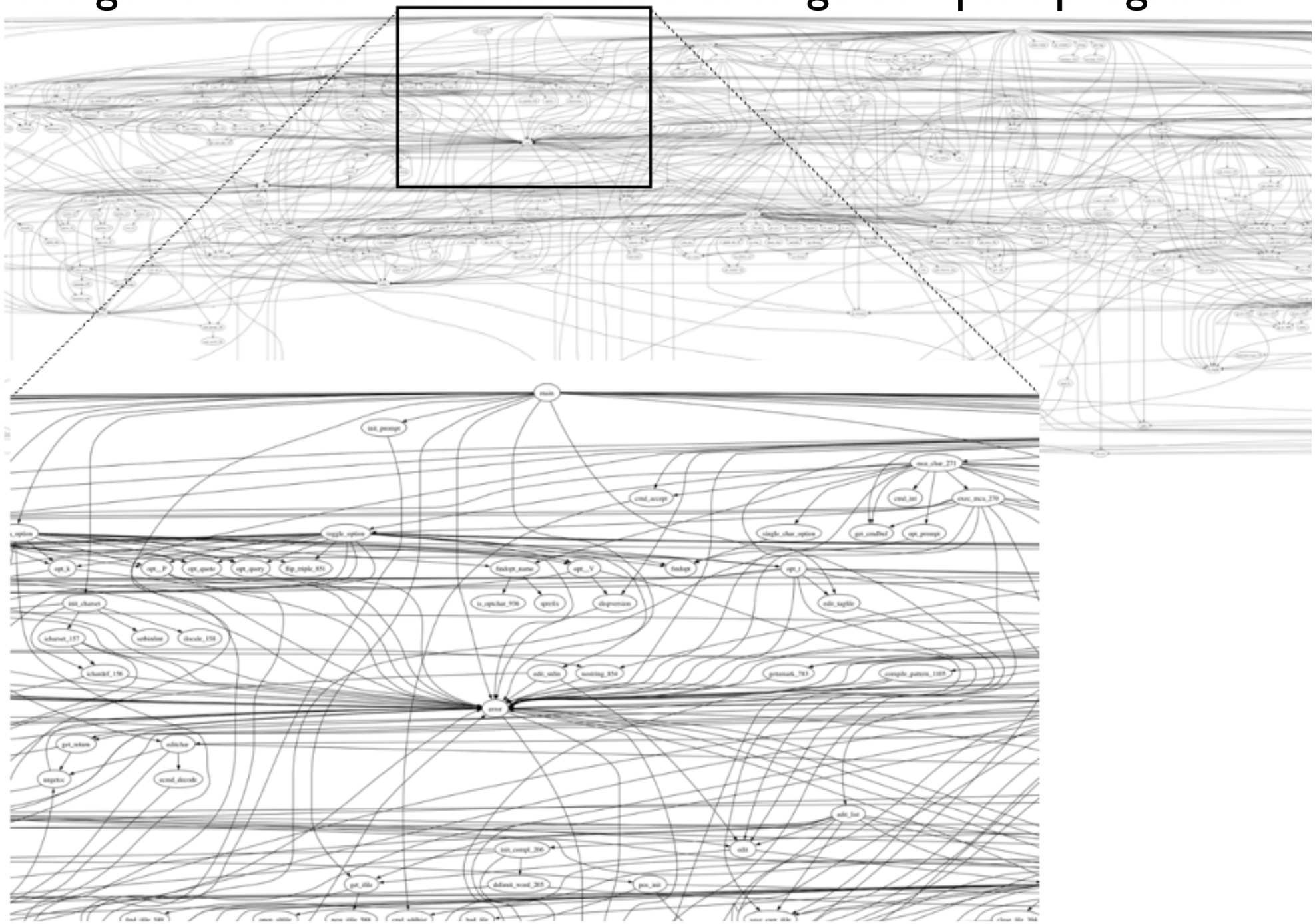


Figure: Call graph of `less-382` (23,822 lines of code)

Sparse Analysis

- The worklist algorithm is often still not efficient enough to analyze large programs.

- Sparse analysis: exploit the semantic sparsity of the input program to analyze

- Spatial sparsity & temporal sparsity

Right part at right moment

Example Performance Gain by Sparse Analysis

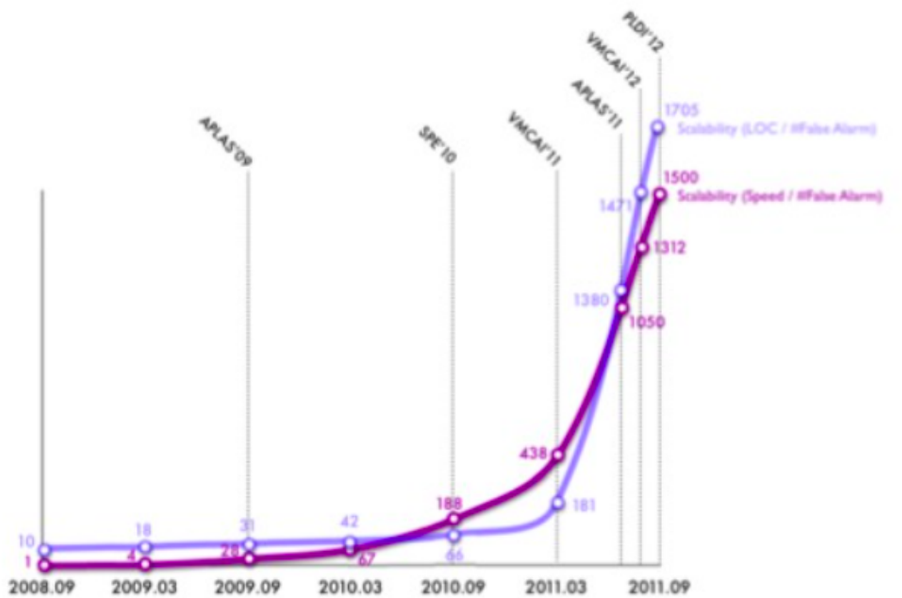
- Sparrow: a "sound", global C analyzer for the memory safety property (no overrun, no null-pointer dereference, etc.)

`http://github.com/ropas/sparrow`

- ~ 10 hours in analyzing million lines of C

Sparrow
The Early Bird

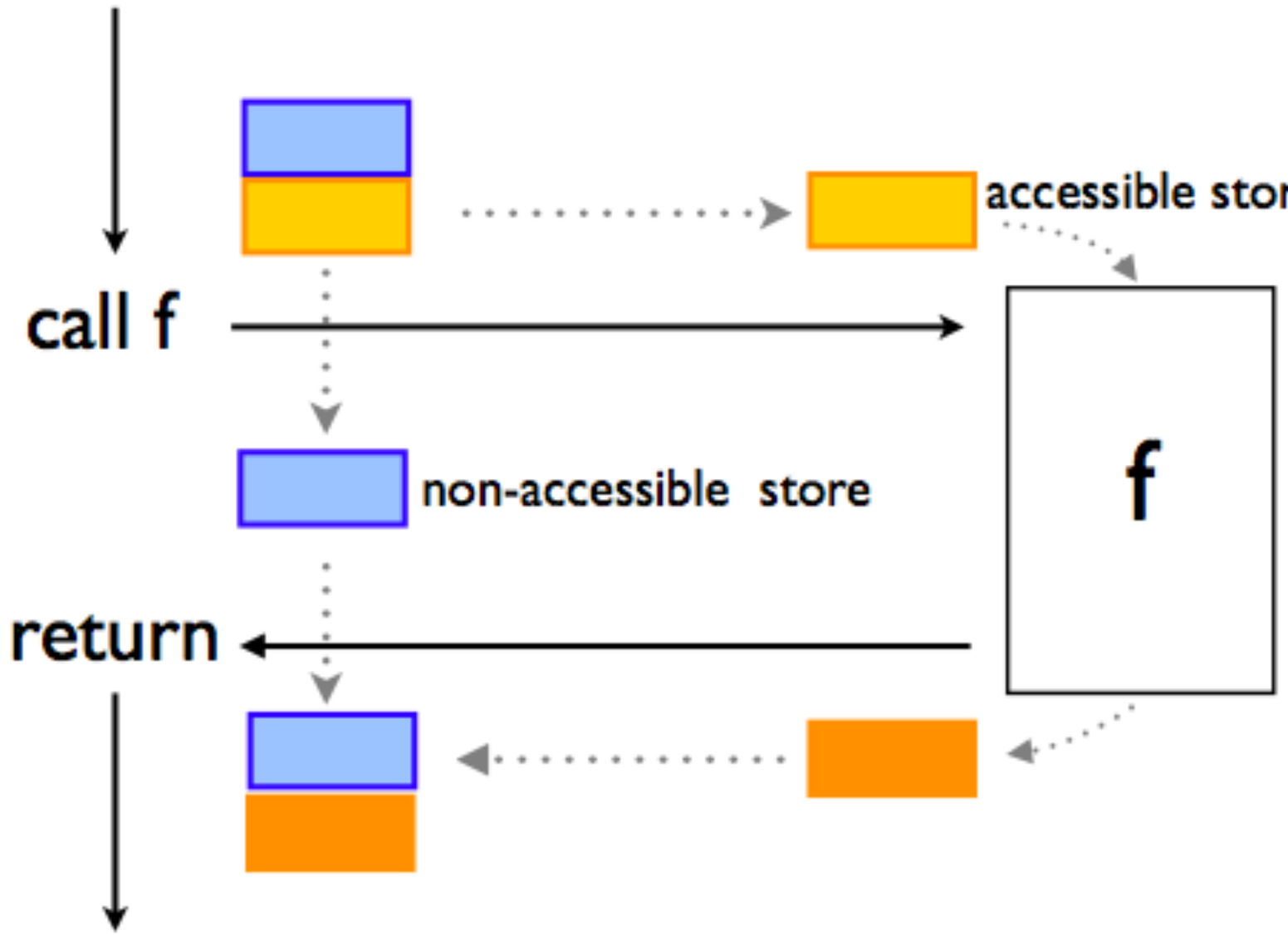
sound-&-global version



- $<$ 1.4M in 10hrs with intervals
- $<$ 0.14M in 20hrs with octagons

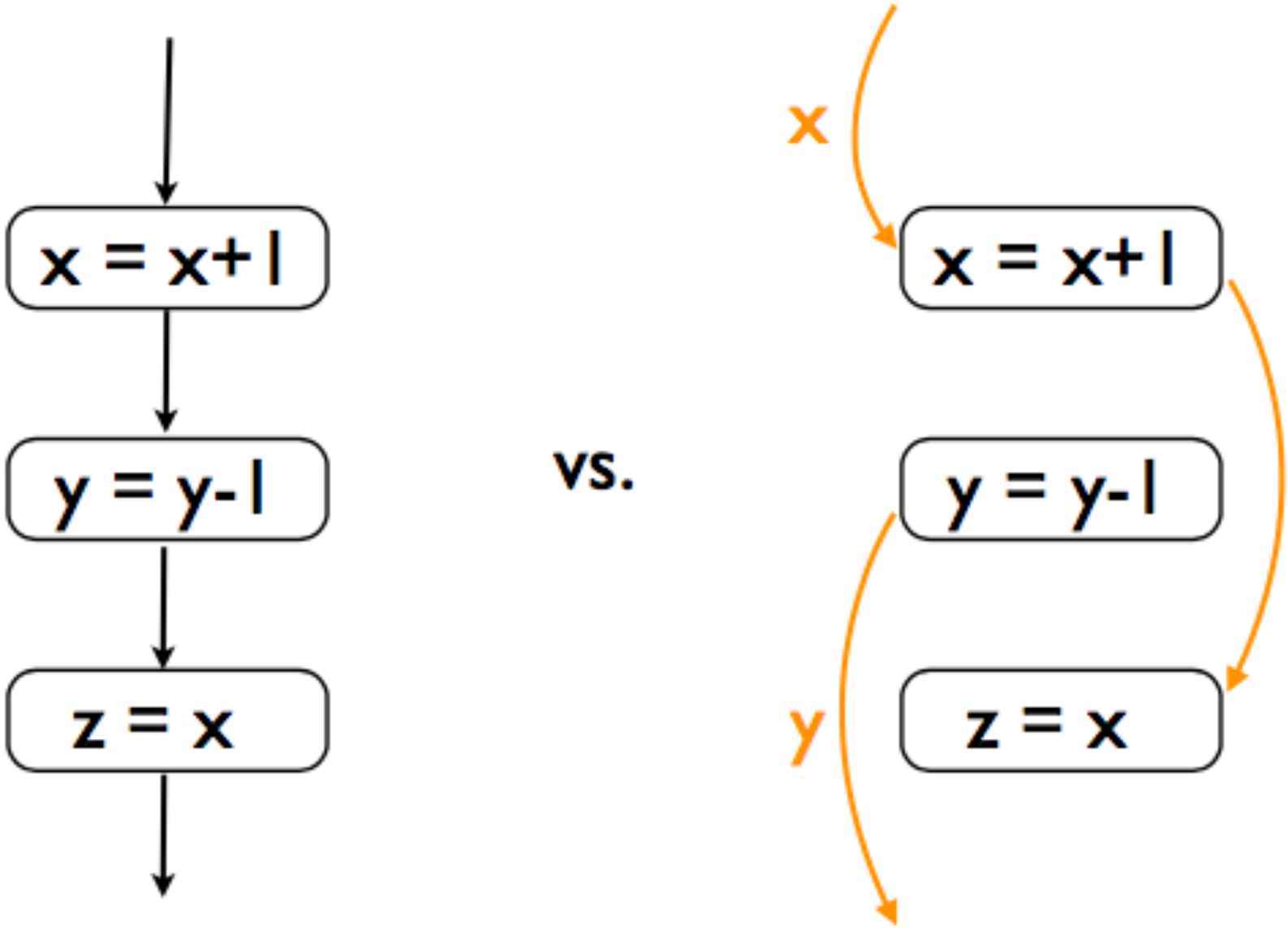
Spatial Sparcity

Each program portion accesses only a small part of the memory.



Temporal Sparcity

After the def of a memory, its use is far.

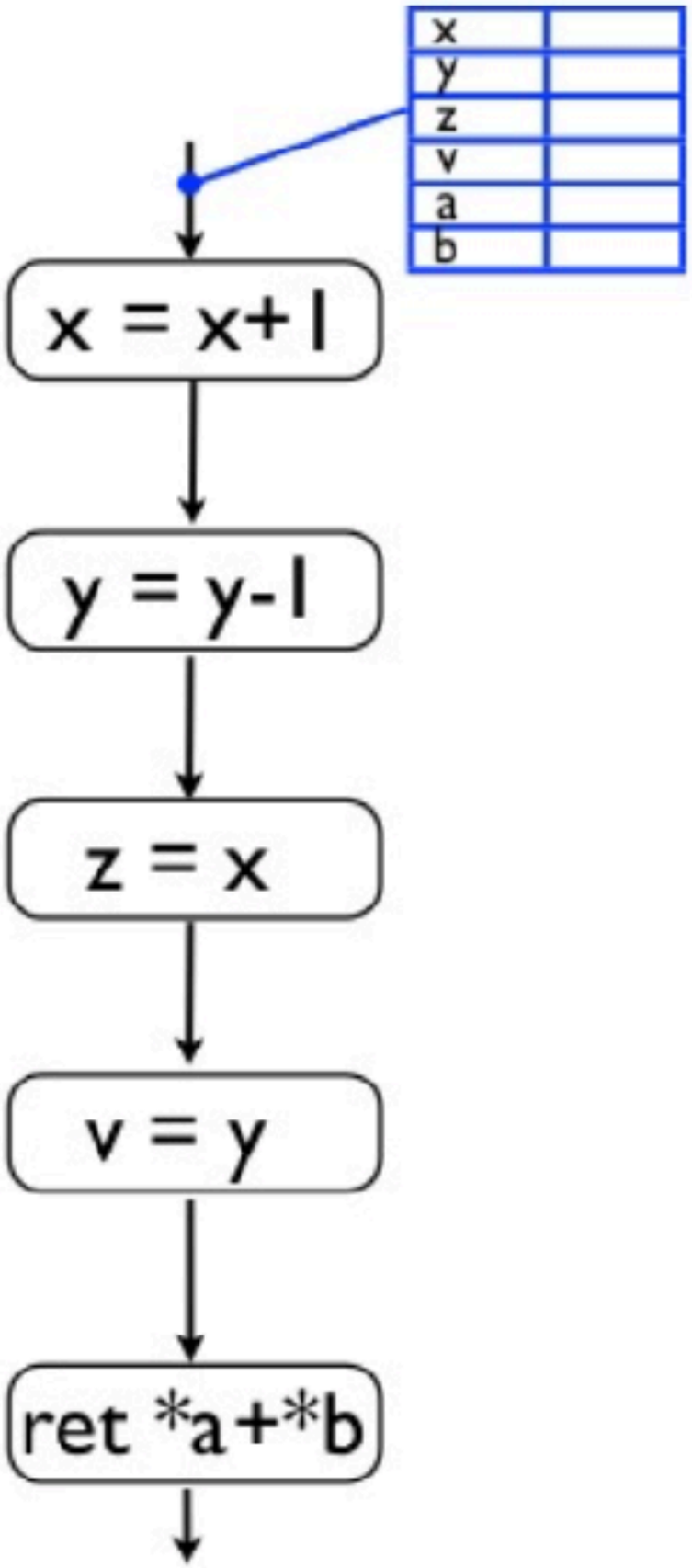


Example

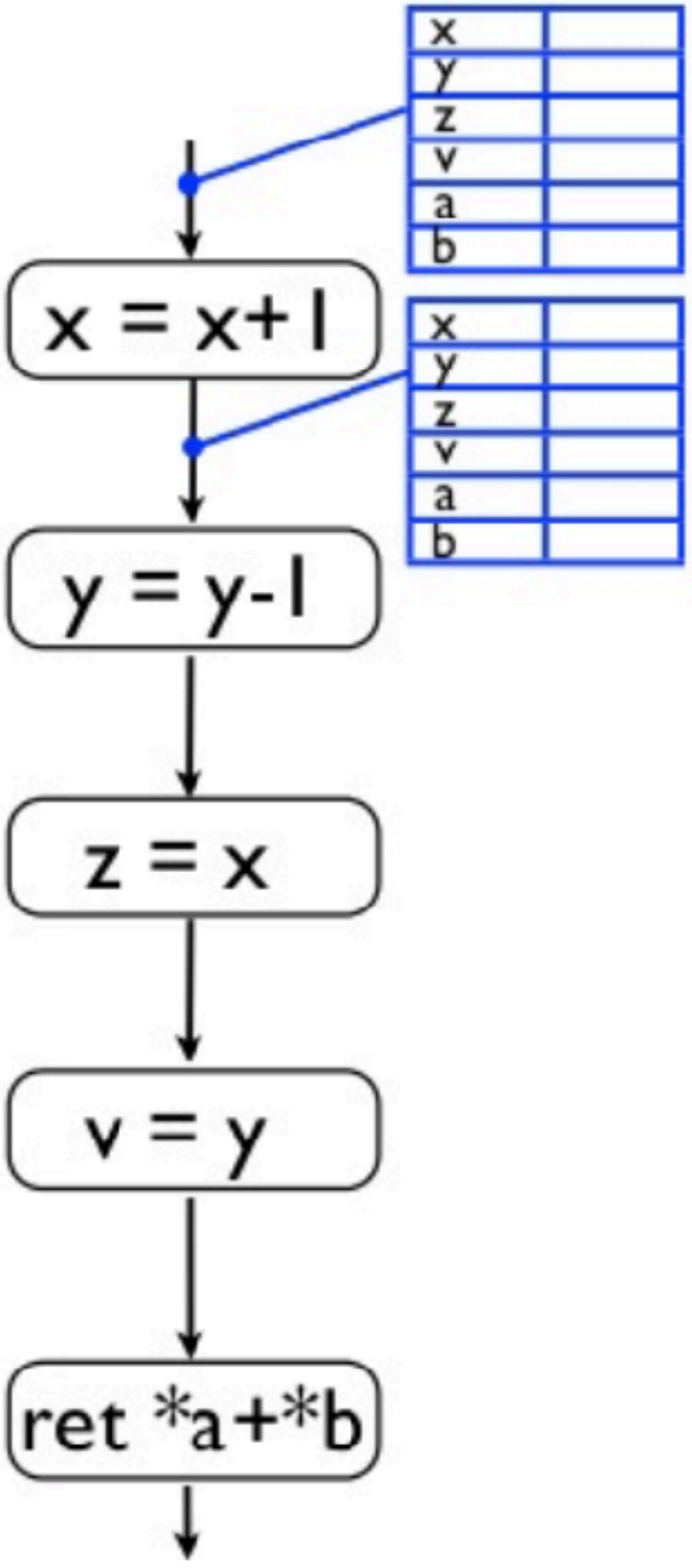
Example (Code fragment)

```
x = x + 1;
y = y - 1;
z = x;
v = y;
ret *a + *b
```

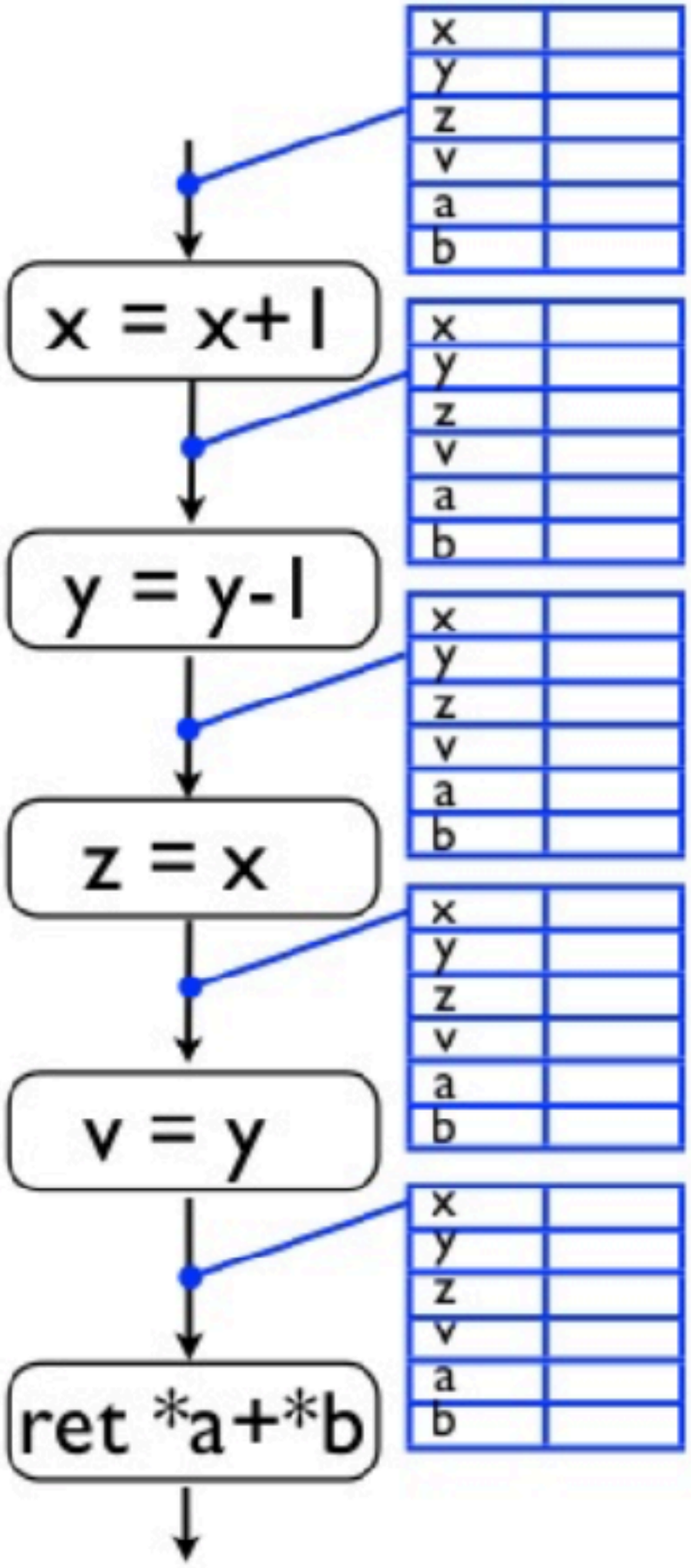
Assume that `a` points to `v` and `b` to `z`.



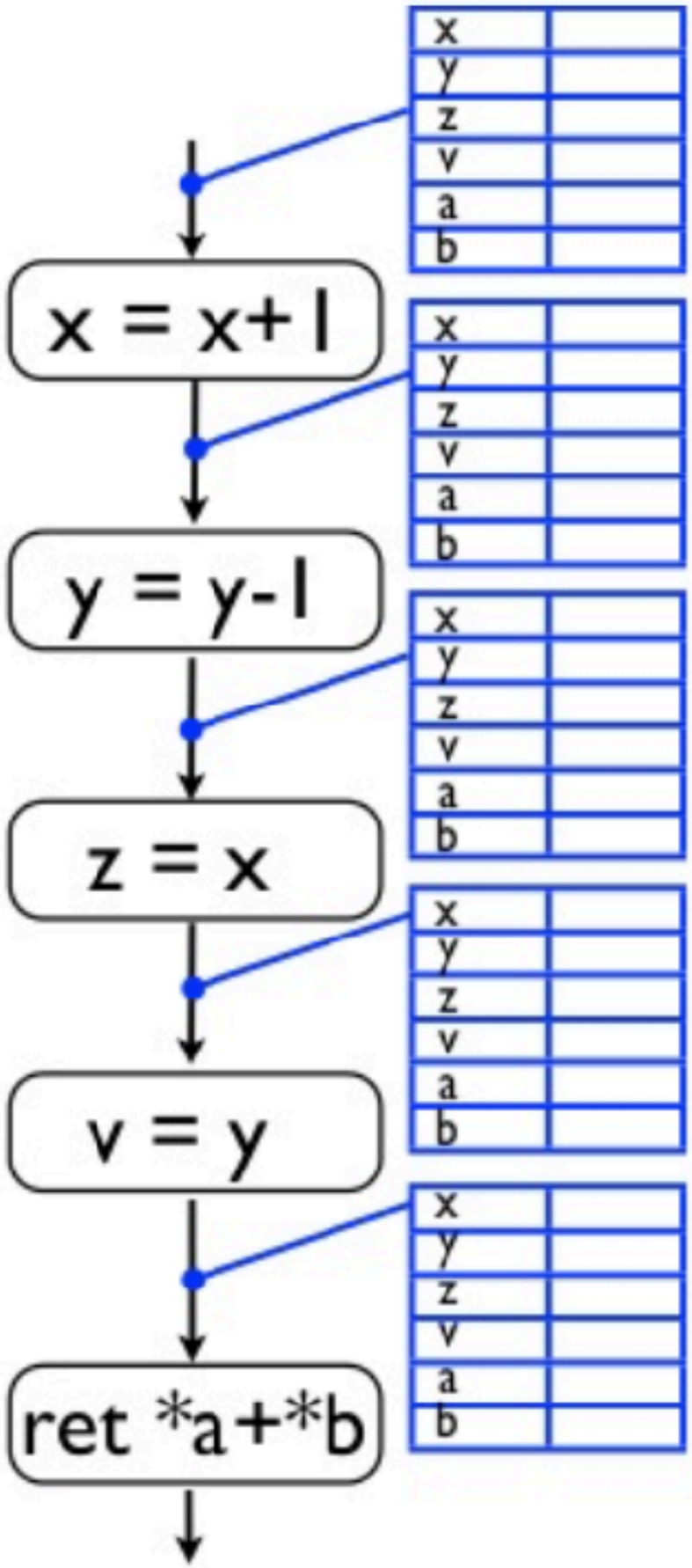
Consider this program and suppose that we analyze the program with the initial abstract state.



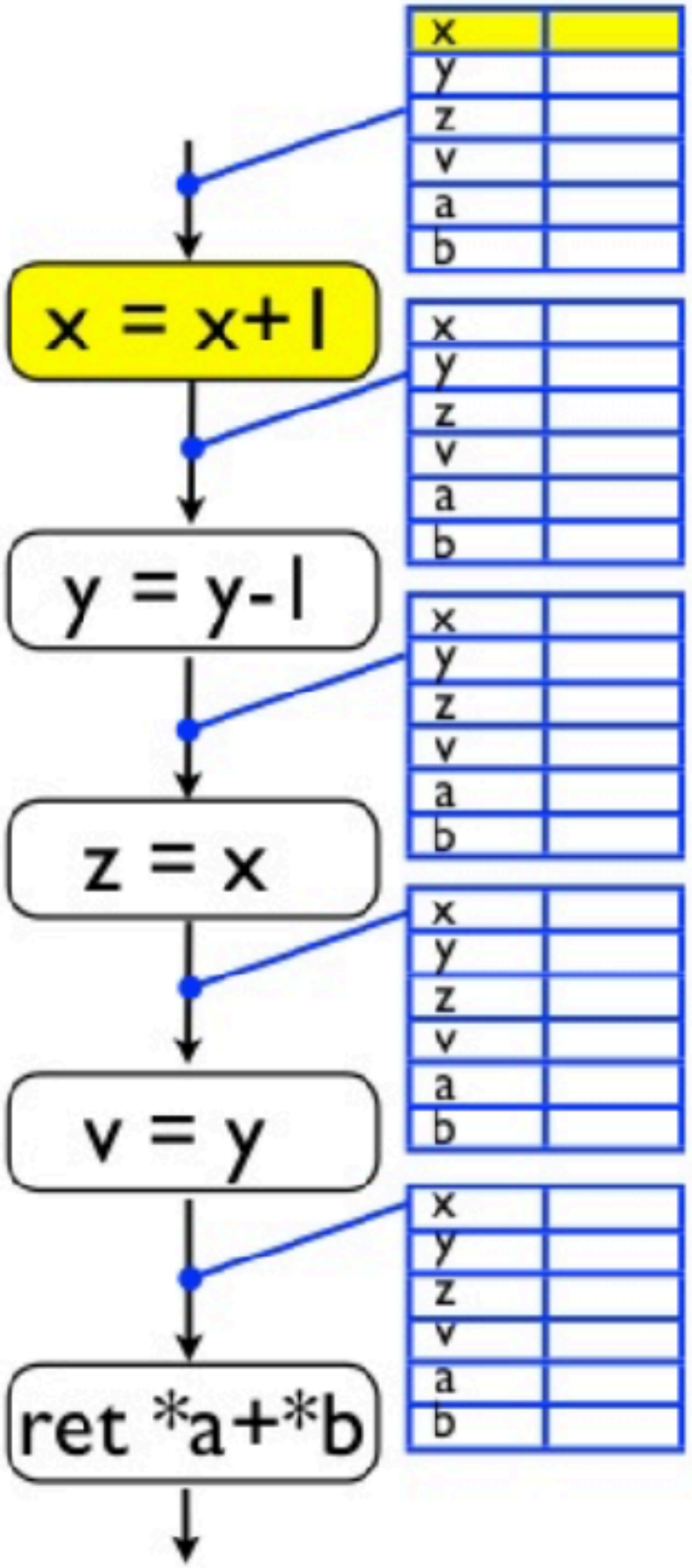
Conventional static analysis propagates the entire abstract states along control flows of the program.



So, after the analysis is terminated, each program point is associated with the entire abstract states.

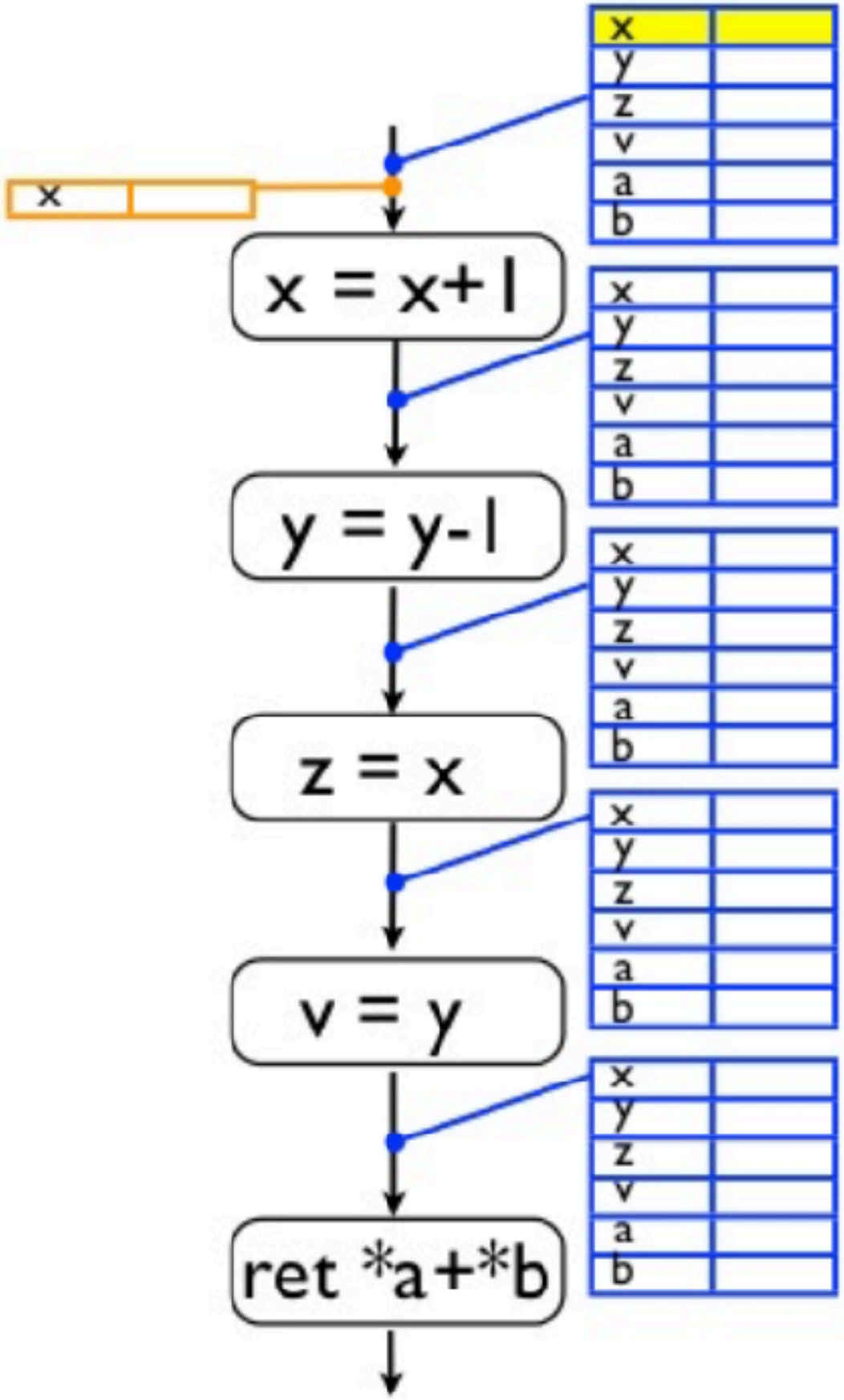


Sparse analysis aims to optimize this conventional static analysis based on two observations.

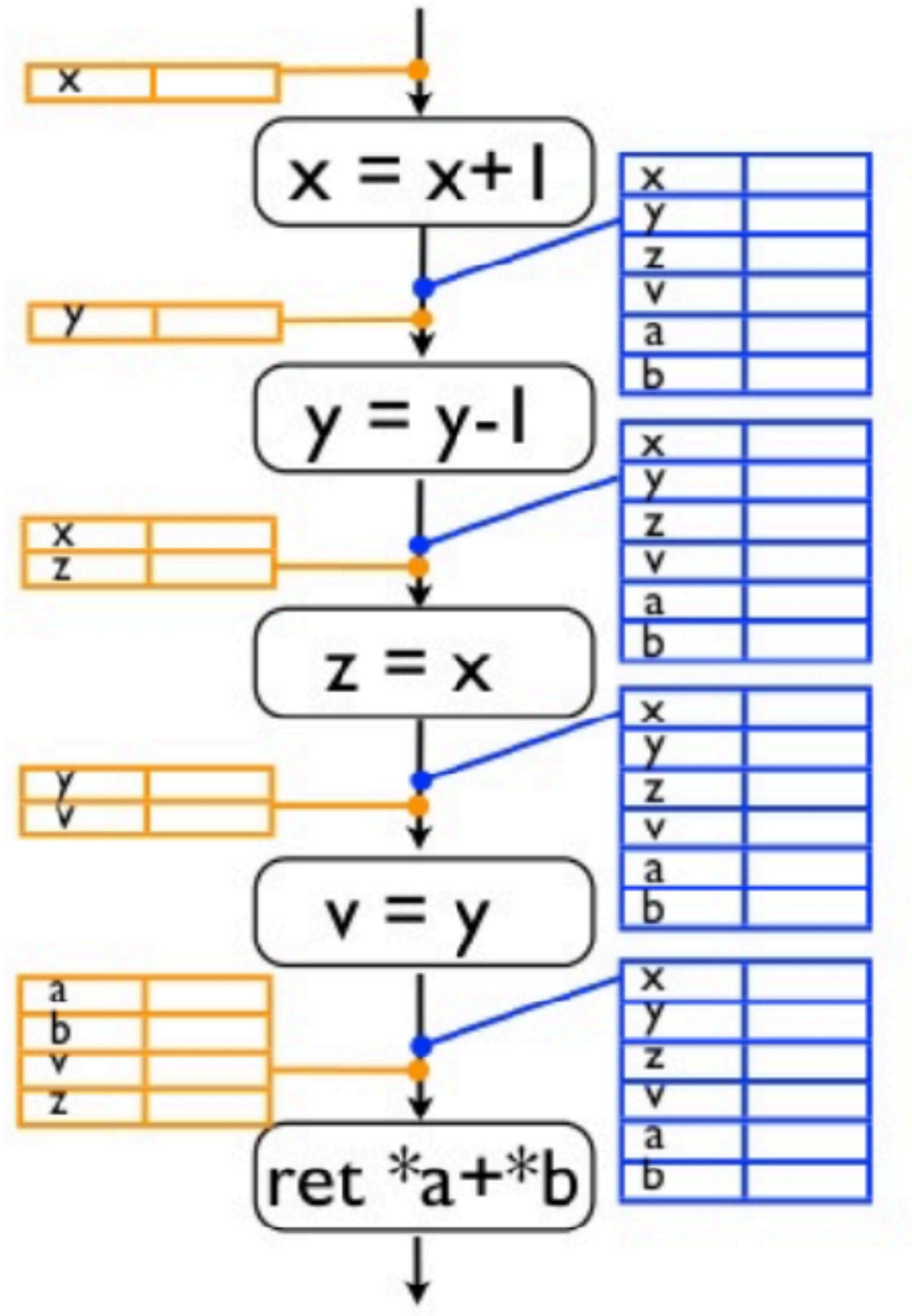


First, in the analysis of each statement, only a small subset of the state is actually used.

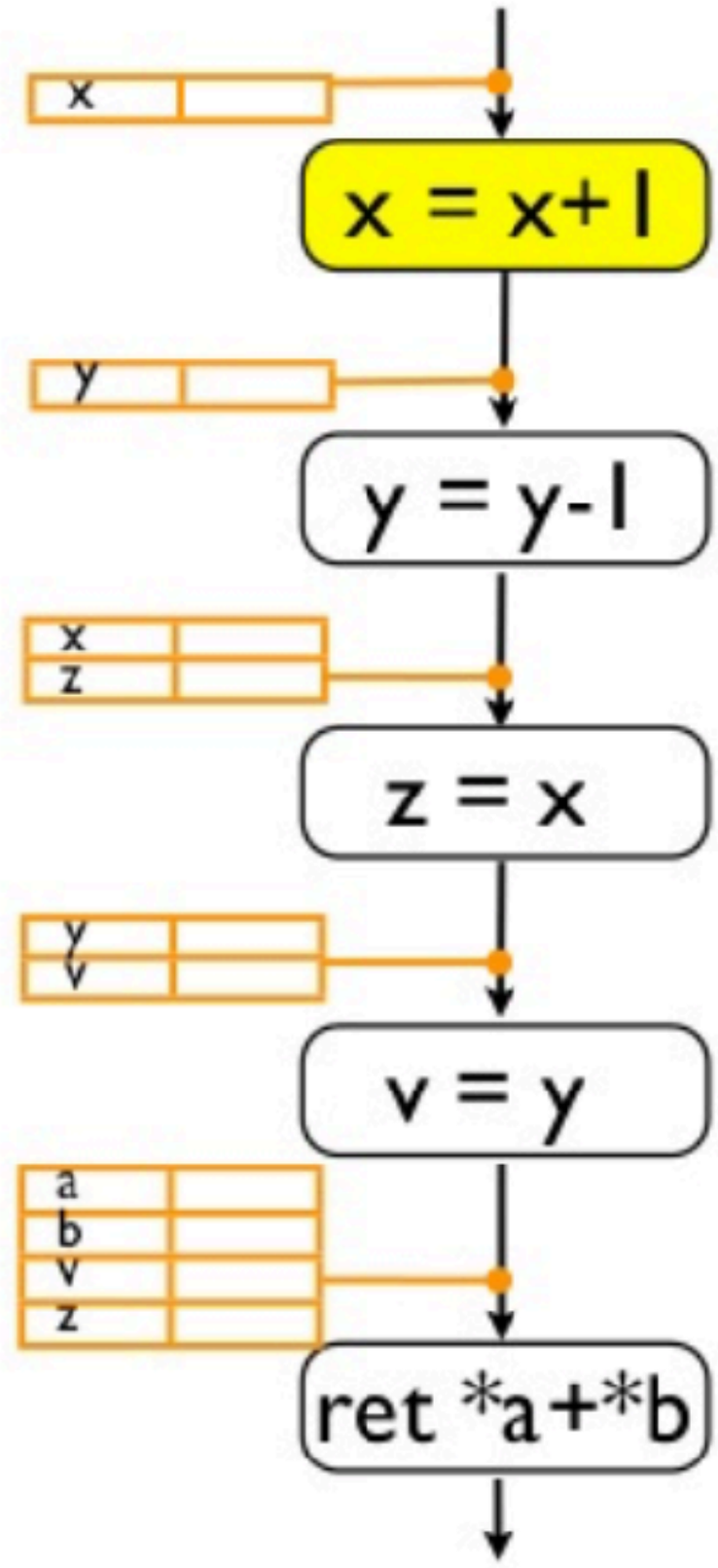
For example, only the value of x is necessary to analyze the first statement.



So, in sparse analysis, we keep x here and remove other values.

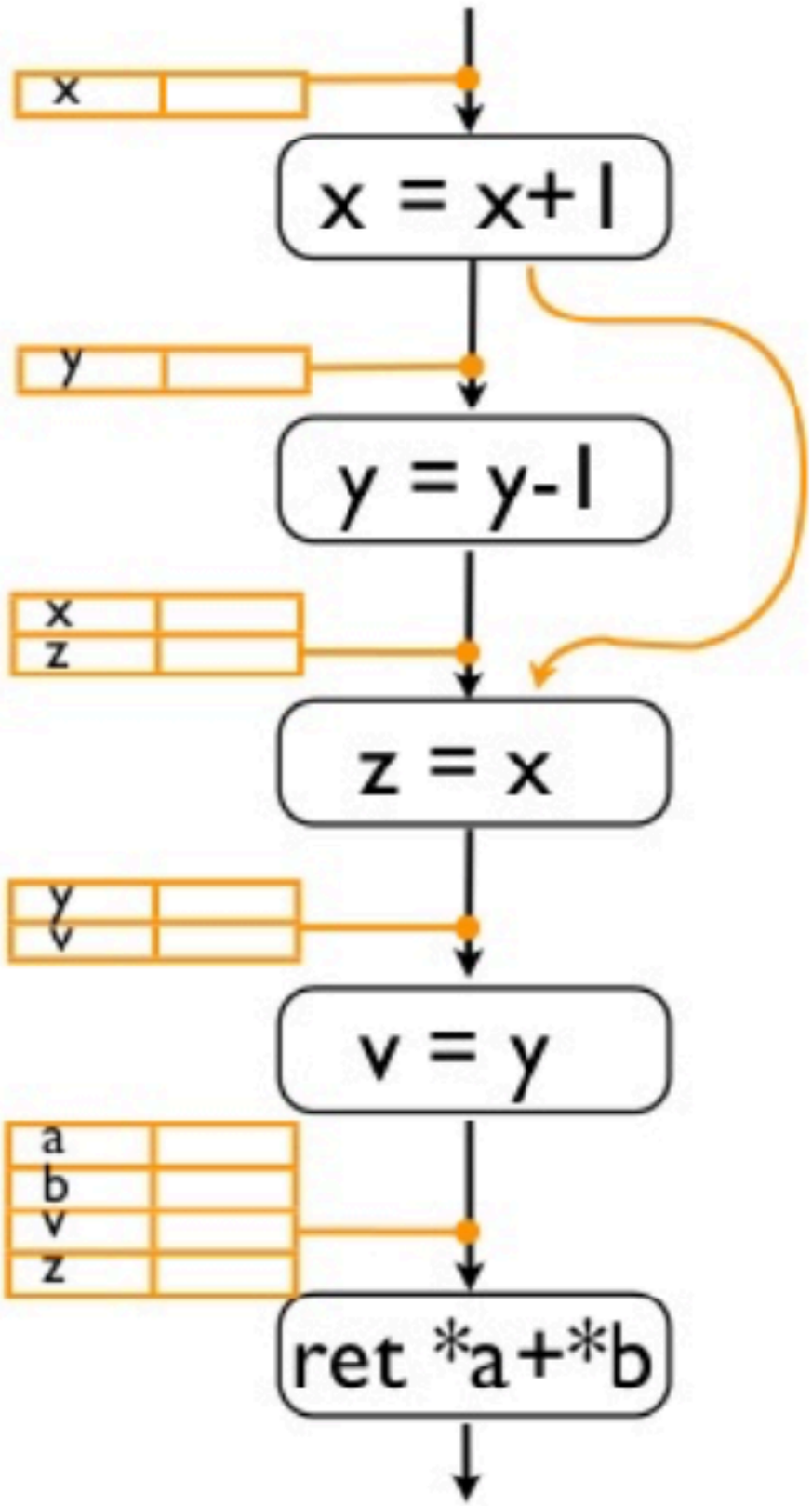


In this way, at each program point, sparse analysis stores only the values that will be used in the analysis.

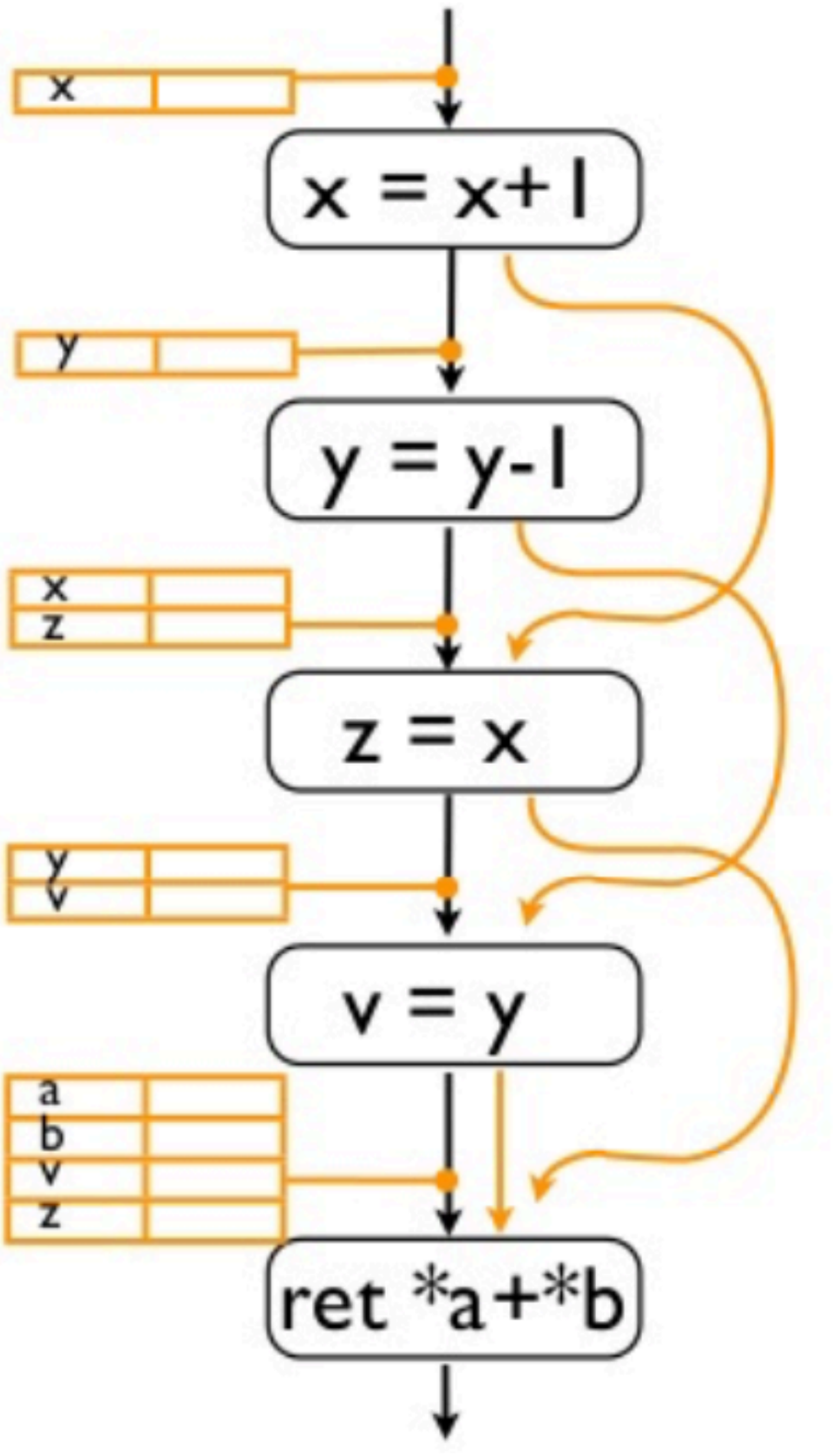


The second observation is that the semantic dependencies among statements are usually sparse.

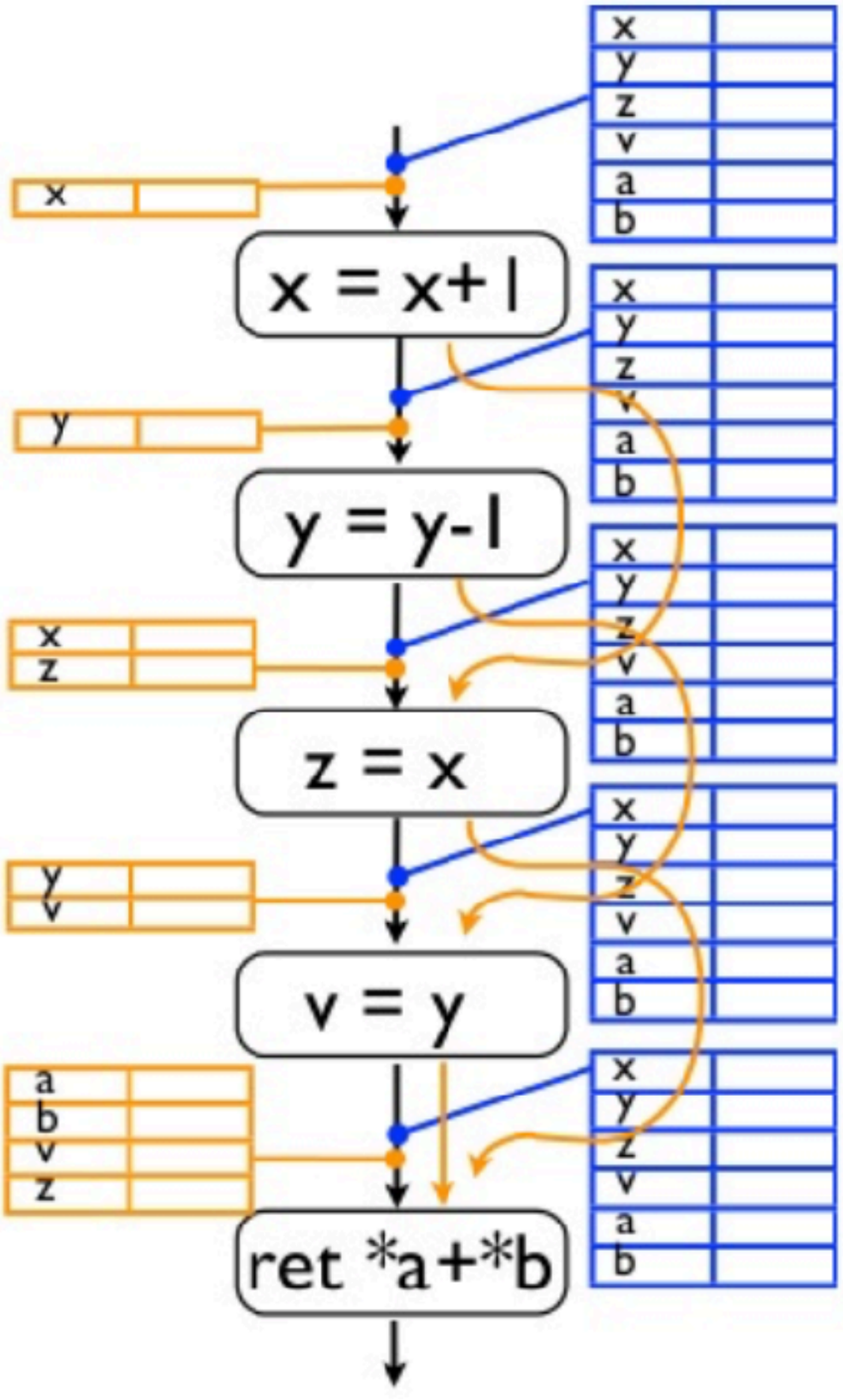
For example, the value of x in the first statement is not used at the next statement but used at the third statement.



So, sparse analysis propagates the value directly to the use point.



Similarly, other values are also propagated along semantic dependencies of the program.



These two "localizations" significantly improves the scalability of the non-sparse analysis.

Exploiting Spartial Sparcity

$$F^\sharp : (\mathbb{L} \to \mathbb{M}^\sharp) \to (\mathbb{L} \to \mathbb{M}^\sharp)$$

becomes

$$F^\sharp_{sparse} : (\mathbb{L} \to \mathbb{M}^\sharp_{sparse}) \to (\mathbb{L} \to \mathbb{M}^\sharp_{sparse})$$

where

$$\mathbb{M}^\sharp_{sparse} = \{M^\sharp \in \mathbb{M}^\sharp \mid dom(M^\sharp) = Access^\sharp(l), l \in \mathbb{L}\} \cup \{\bot\}.$$

Variables used at label l of the input program

Exploiting Temporal Sparcity

Need the def-use chain information as follows.

- we streamline the abstract one-step relation

$$(l, M^\sharp) \hookrightarrow^\sharp (l', M'^\sharp) \quad \text{for } l' \in \mathtt{next}^\sharp(l, M^\sharp).$$

so that the link $\hookrightarrow^\sharp$ should follow the **def-use chain**:

  - from (def) a label where a location is defined
  - to (use) a label where the defined location is read

Precision-Preserving Sparse Analysis

Goal

$$F^\sharp : D^\sharp \to D^\sharp \quad \overset{\text{sparsify}}{\Longrightarrow} \quad F^\sharp_{sparse} : D^\sharp \to D^\sharp$$

$$\mathsf{lfp} F^\sharp \quad \overset{\text{still}}{=} \quad \mathsf{lfp} F^\sharp_{sparse}$$

Step 1: Estimating Accessed Variables at Each Label

Need to safely estimate

$$Access^\sharp(l).$$

Use yet another sound static analysis, a futher abstraction:

$$(\mathbb{L} \to \mathbb{M}^\sharp, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathbb{M}^\sharp, \sqsubseteq_M)$$

(a "flow-insensitive" version of the "flow-sensitive" analysis design)

Flow-insensitive pointer analysis will be introduced in the upcoming lectures

Step 2: Computing Def-Use Information

- Let

$$D^\sharp : \mathbb{L} \to \wp(\mathbb{X}) \text{ and } U^\sharp : \mathbb{L} \to \wp(\mathbb{X})$$

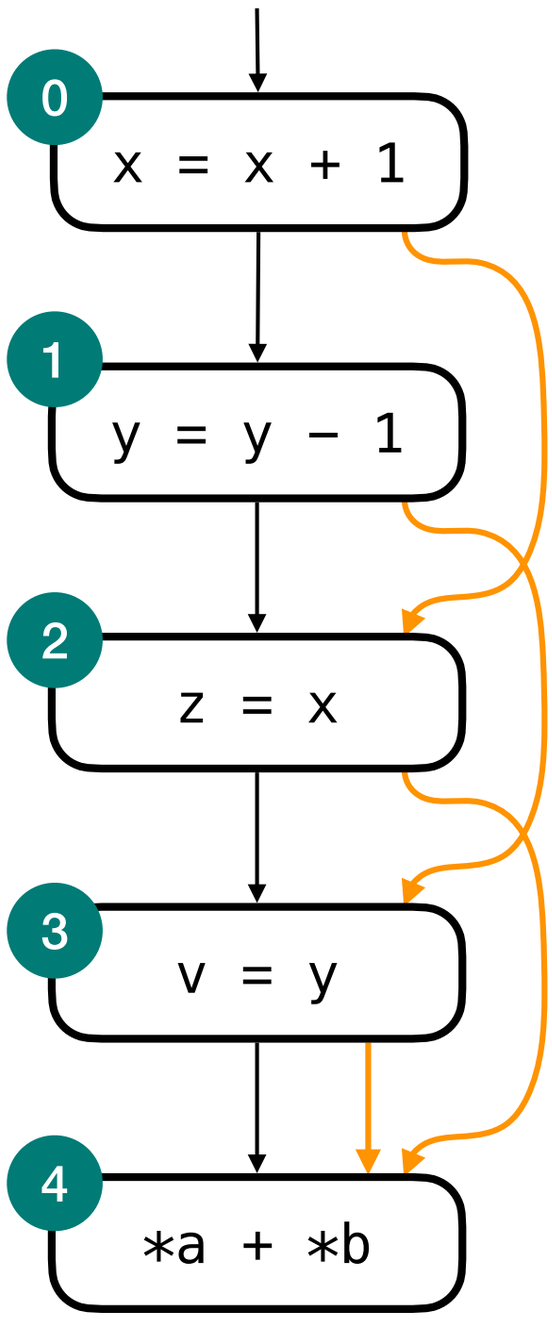
  be the def and use sets from the original analysis.
- Need to safely estimate $D^\sharp$ and $U^\sharp$.
- Use yet another sound static analysis to compute

$$D^\sharp_{pre} \text{ and } U^\sharp_{pre}$$

such that

- $\forall l \in \mathbb{L} : D^\sharp_{pre}(l) \supseteq D^\sharp(l)$ and $U^\sharp_{pre}(l) \supseteq U^\sharp(l)$.
- $\forall l \in \mathbb{L} : U^\sharp_{pre}(l) \supseteq D^\sharp_{pre}(l) \setminus D^\sharp(l)$.

Def and Use Sets



$$D^\sharp(0) = \{x\}$$
$$U^\sharp(0) = \{x\}$$

$$D^\sharp(1) = \{y\}$$
$$U^\sharp(1) = \{y\}$$

$$D^\sharp(2) = \{z\}$$
$$U^\sharp(2) = \{x\}$$

$$D^\sharp(3) = \{v\}$$
$$U^\sharp(3) = \{y\}$$

$$D^\sharp(4) = \{\}$$
$$U^\sharp(4) = \{a, b, v, z\}$$

*Assume a and b point to z and v

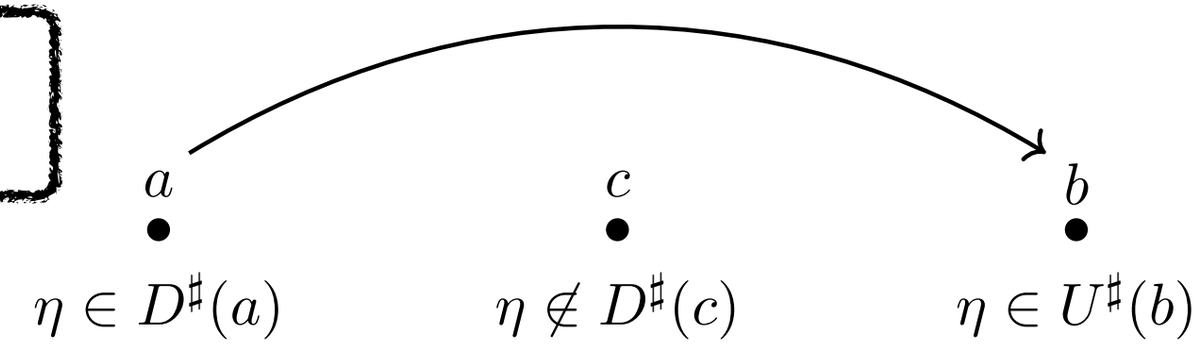
Step 3: Analysis with Def-Use Chains

- **Def-use chain**: Label `a` to label `b` is a def-use chain for a variable `x` whenever `x` is defined at `a` and used at `b`, and `x` is not re-defined in-between `a` and `b`.

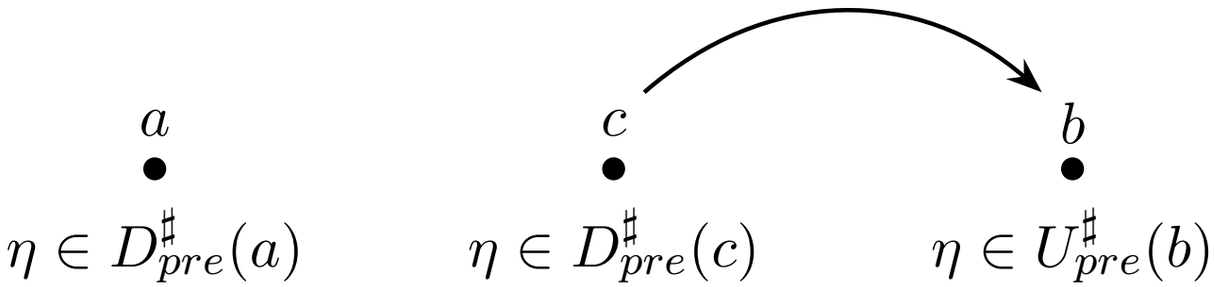
- The resulting sparse analysis with safe def-use chains has the same precision as the original non-sparse analysis.

Need for the Second Condition for $D^{\sharp}_{pre}$ and $U^{\sharp}_{pre}$

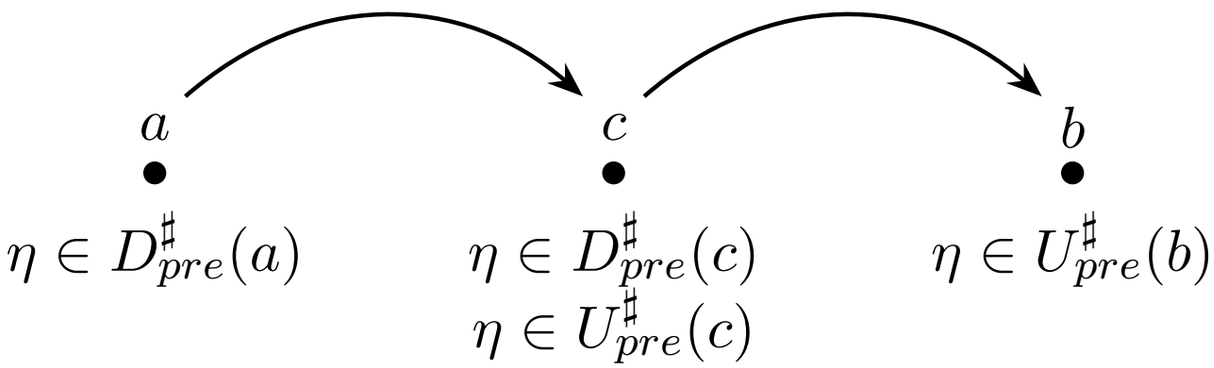
$\forall l \in \mathbb{L} : \hat{U^{\sharp}_{pre}}(l) \supseteq D^{\sharp}_{pre}(l) \setminus D^{\sharp}(l)$.



(d) Original analysis def-use edge for η



(e) Missing def-use edge (a to b) for η because of over-approximate $D^{\sharp}_{pre}(c)$



(f) Recovered def-use edge (a to b via c) for η by safe $U^{\sharp}_{pre}(c)$