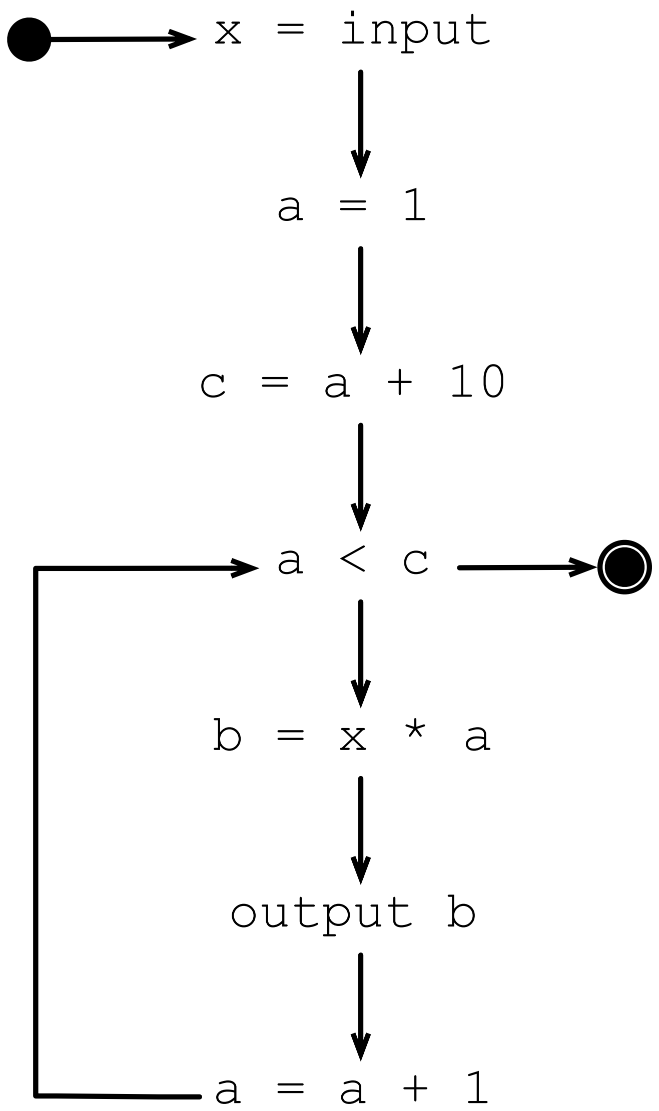
SPARSE ABSTRACT INTERPRETATION

PROGRAM ANALYSIS AND OPTIMIZATION – DCC888

Fernando Magno Quintão Pereira
fernando@dcc.ufmg.br

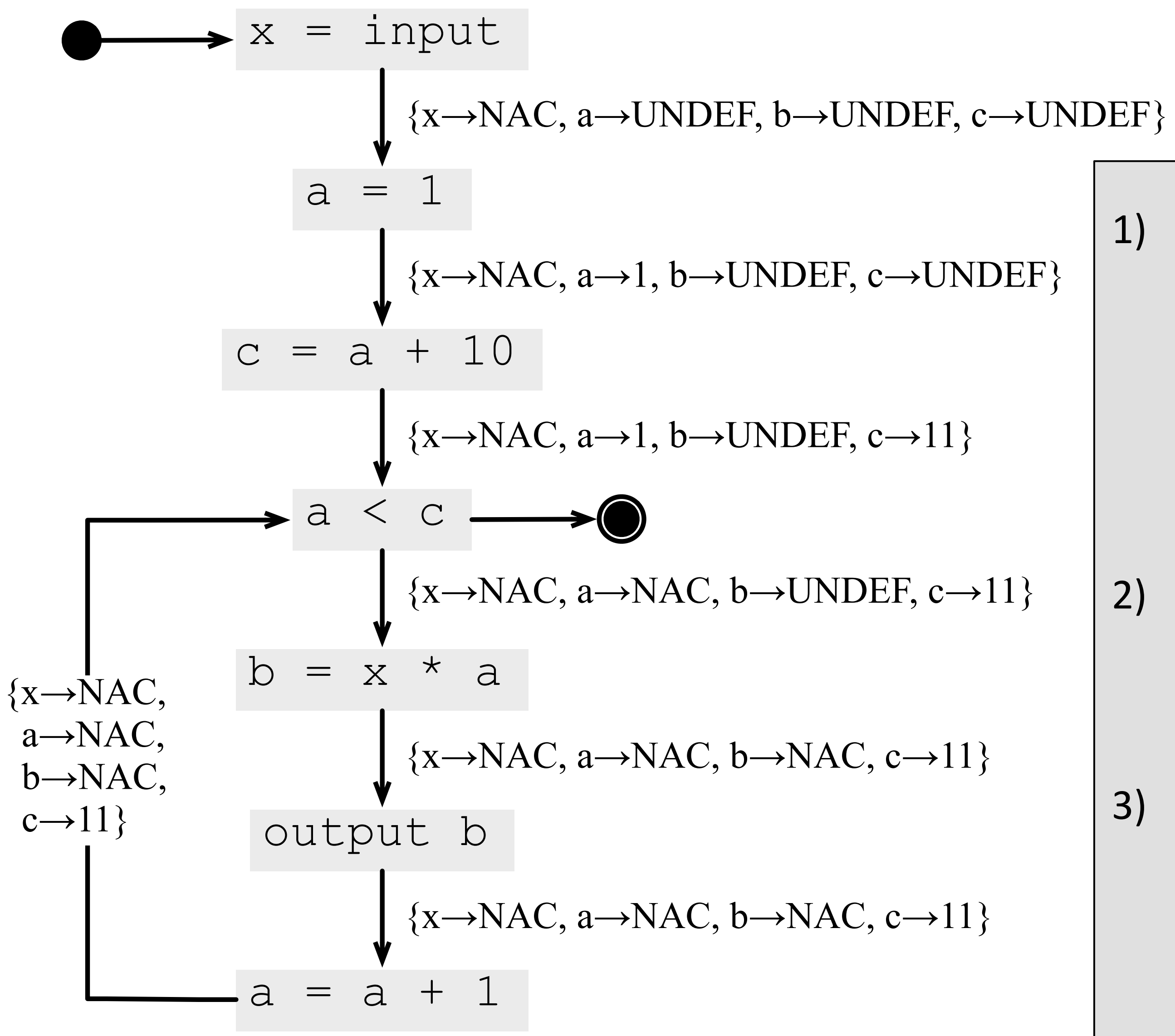
The material in these slides have been taken from the paper "Parameterized Construction of Program Representations for Sparse Dataflow Analyses", by Tavares *et al*. which is available in the course webpage.

Constant Propagation Revisited



What is the solution that our constant propagation analysis produces for this program?

Constant Propagation Revisited



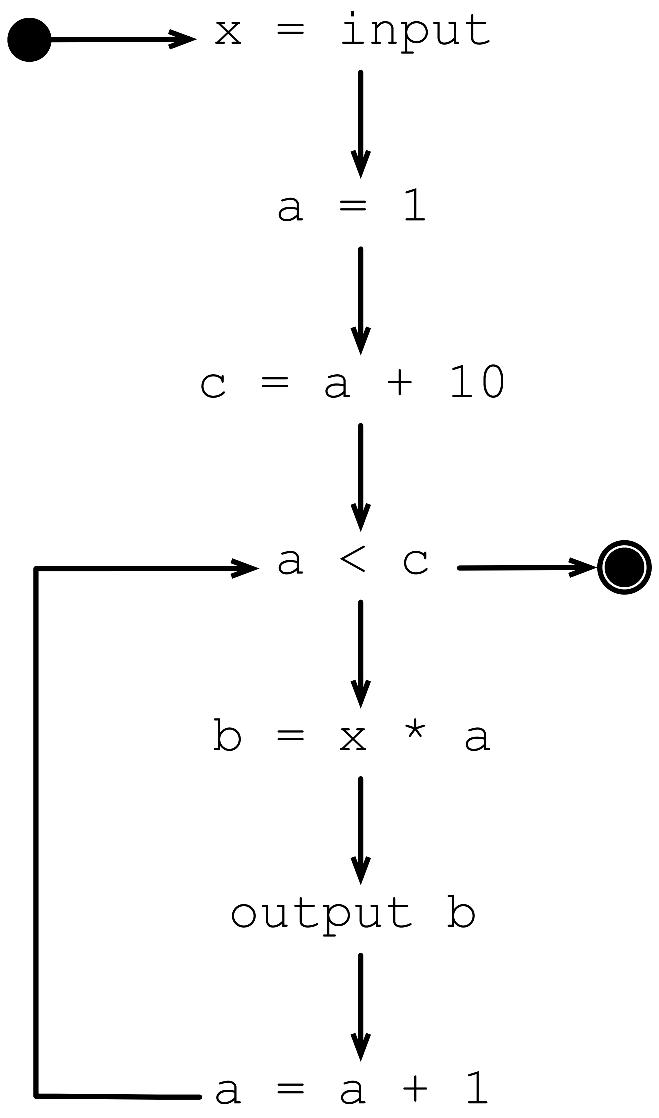
x = input
{x→NAC, a→UNDEF, b→UNDEF, c→UNDEF}
a = 1
{x→NAC, a→1, b→UNDEF, c→UNDEF}
c = a + 10
{x→NAC, a→1, b→UNDEF, c→11}
a < c
{x→NAC, a→NAC, b→UNDEF, c→11}
b = x * a
{x→NAC, a→NAC, b→NAC, c→11}
output b
{x→NAC, a→NAC, b→NAC, c→11}
a = a + 1
{x→NAC, a→NAC, b→NAC, c→11}

1) How much space does this solution requires? Answer in terms of I, the number of instructions, and V, the set of variables.

2) Is there any redundancy in the solution given in the code on the left?

3) Why can't we simply bind the information directly to the *variable name*, i.e., the variable is constant or not?

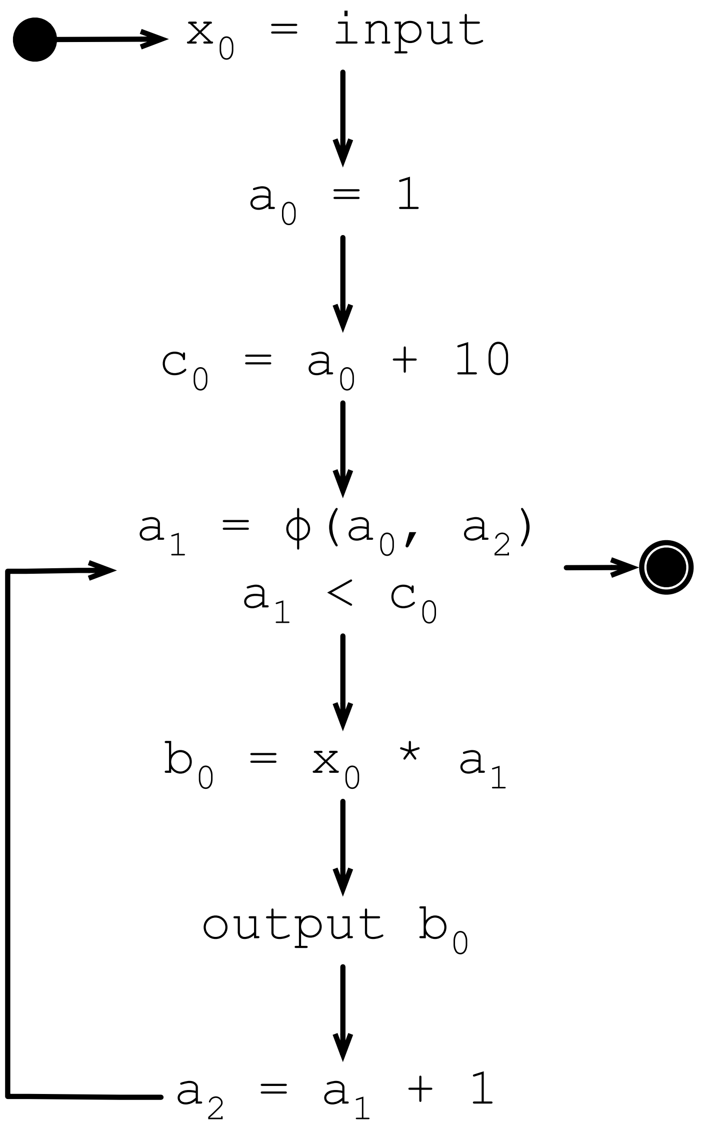
Binding Information to Variables



- Associating the information with a variable name does not work in constant propagation, because the same variable name may denote a variable that is constant at some program points, and non-constant at others.

How could we solve this problem, so that the abstract state of a variable, i.e., constant or not, be invariant along this variable's entire live range?

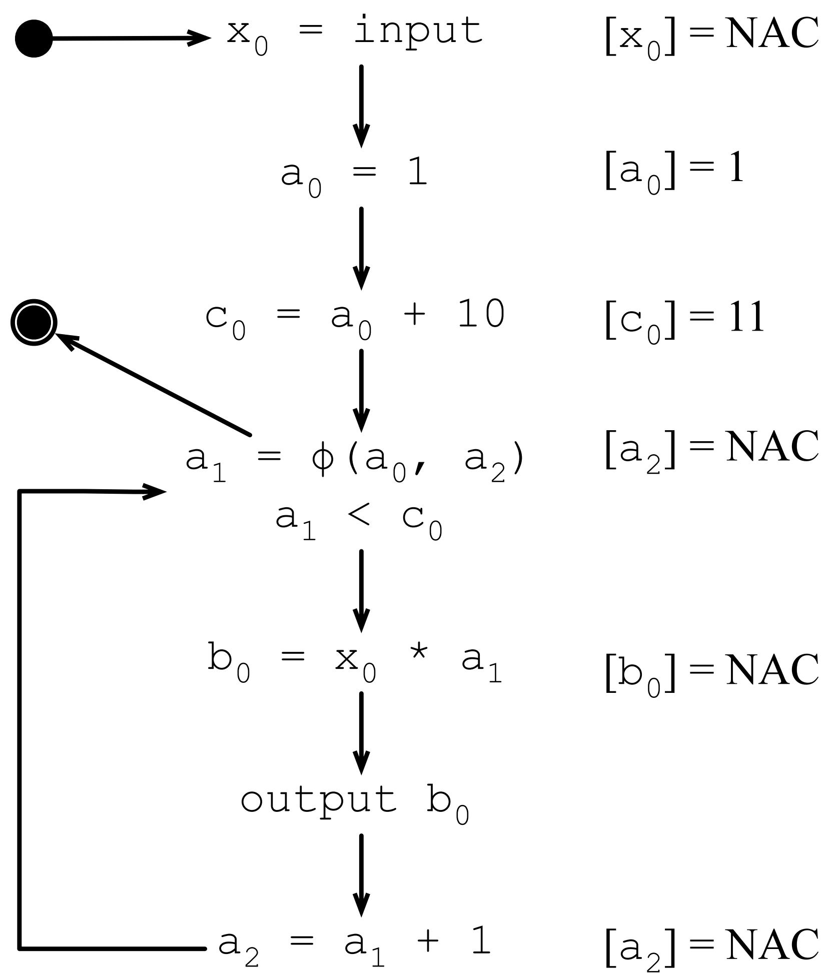
Constant Propagation in SSA Form Programs



- If we convert the program to Static Single Assignment form, then we can guarantee that the information associated with a variable name is invariant along the entire live range of this variable.

Why can we provide this guarantee about constant propagation, if the program is in Static Single Assignment form?

Sparse Constant Propagation



- The only instruction that determines if a variable is constant or not is the assignment.
- We have split live ranges at every point where new information is acquired.
- Thus, we can bind the information, i.e., the abstract state of a variable, to its live range.
- Whenever this is possible, we call the analysis **sparse**.

Sparse Analyses

- If a data-flow analysis binds information to pairs of program points × variable names, then we call it **dense**.
- If the data-flow analysis binds information to variables, then we call it **sparse**.
- But there is one more requirement for a data-flow analysis to be sparse: it must not use *trivial transfer functions*.
- We say that a transfer function is trivial if it does not generate new information.
  - In other words, the transfer function is an identity.

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO
EM CIÊNCIA DA COMPUTAÇÃO
DCC-UFMG

Trivial Transfer Functions

The transfer function associated with the assignment "a = 1" is non-trivial, because it changes the abstract state of the variable name "a".

$$\begin{tikzcd} {} \arrow[d, "{\{x \to NAC, a \to UNDEF, b \to UNDEF, c \to UNDEF\}}"] \\ \texttt{a = 1} \arrow[d, "{\{x \to NAC, a \to 1, b \to UNDEF, c \to UNDEF\}}"] \\ {} \end{tikzcd}$$

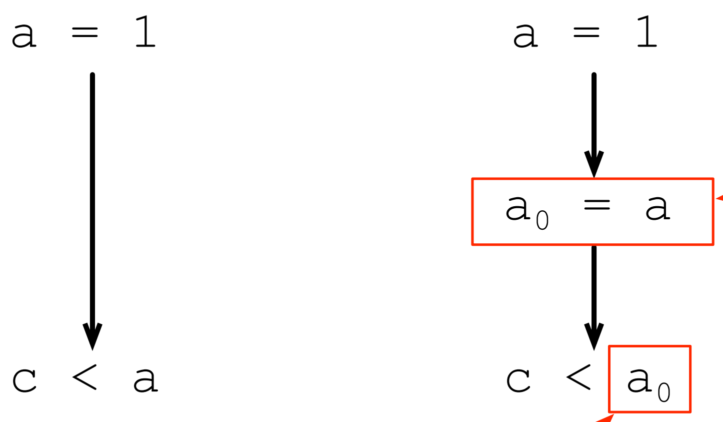
On the other hand, the transfer function associated with the instruction "output b" is trivial, as it does not change the abstract state of any variable.

$$\begin{tikzcd} {} \arrow[d, "{\{x \to NAC, a \to NAC, b \to NAC, c \to 11\}}"] \\ \texttt{output b} \arrow[d, "{\{x \to NAC, a \to NAC, b \to NAC, c \to 11\}}"] \\ {} \end{tikzcd}$$

A trivial transfer function *f* is an identity, *i = f(i)*. In other words, it receives some data-flow information i, and outputs the same information.

Building Sparse Data-Flow Analyses

- We want to build program representations that "sparsify" data-flow analysis.
- We will do it via live *range splitting*.
- We split the live range of a variable via copy instructions.

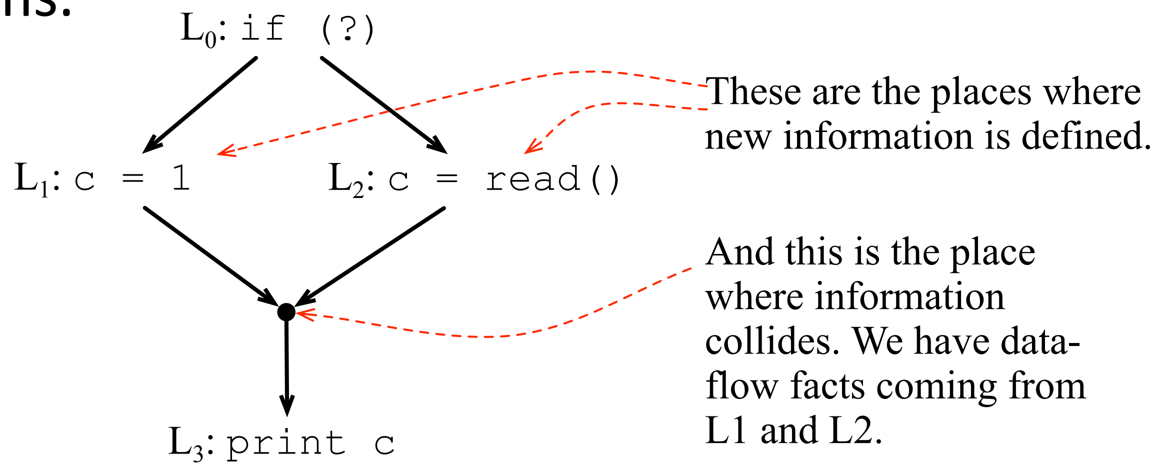


This is the splitting point: the program point where we have inserted a new copy of variable a_0, thus splitting its live range.

We have renamed **every use** of variable a, that is dominated by the splitting point, to a_0.

Splitting Points

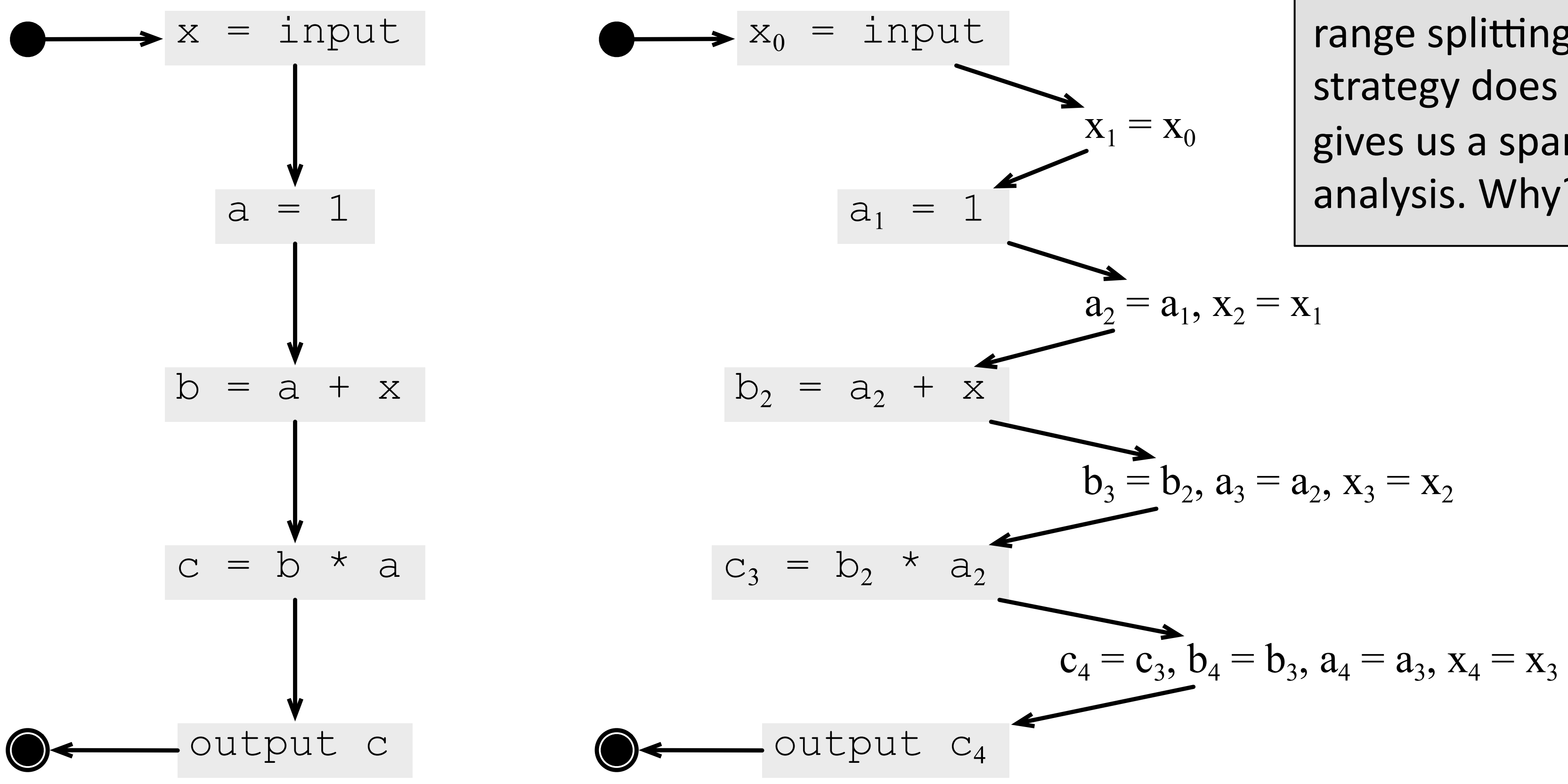
- We split live ranges at the places where new information is generated.
  - In the case of constant propagation, we split at the definition point of variables.
- We split live ranges at the places where different information may collide.
  - In the case of constant propagation, we split at the join points of programs.



L0: if (?)
L1: c = 1
L2: c = read()
L3: print c
These are the places where new information is defined.
And this is the place where information collides. We have data-flow facts coming from L1 and L2.

Splitting All Over is not What we Want

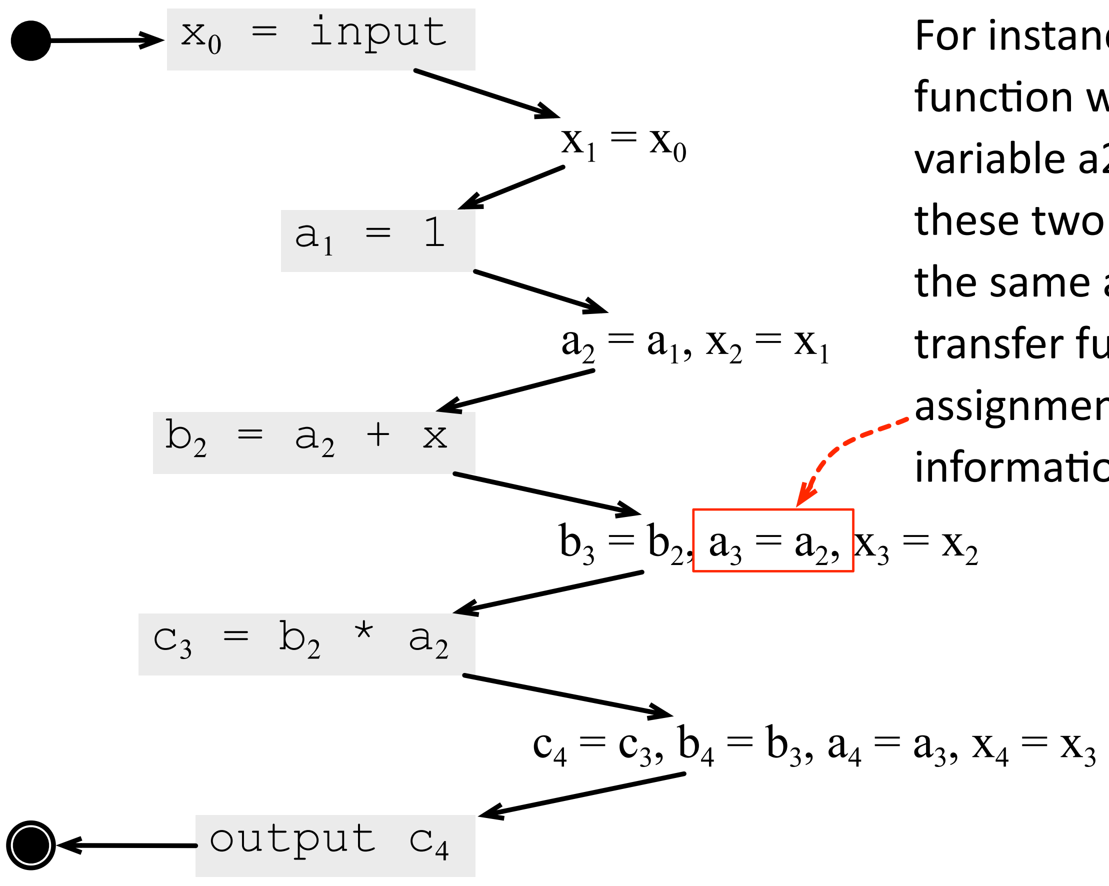
- If we split live ranges at every program point, then, naturally the information associated with every live range will be invariant.



However, this live range splitting strategy does not gives us a sparse analysis. Why?

We do not want trivial transfer functions

- We do not have a sparse analysis because many of the transfer functions associated with the copies are trivial.



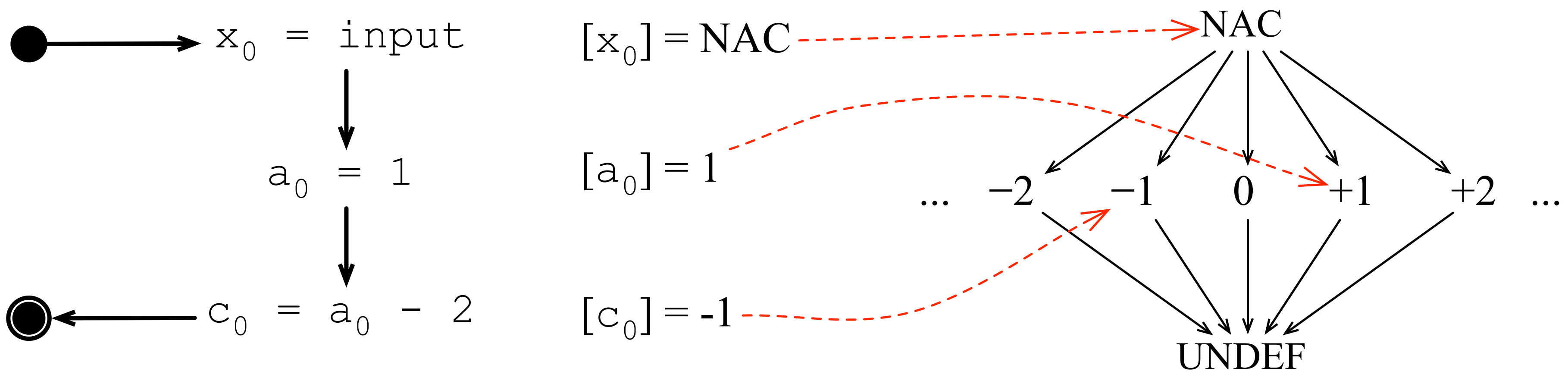
For instance, here our transfer function will copy the abstract state of variable a2 into variable a3. Thus, these two variables will always have the same abstract state, and the transfer function that we have for assignments is not generating any new information.

$$\frac{[\![v']\!] = a}{[\![v]\!] = a}$$

But, in this case, where do we split live ranges?

The Single Information Property

- The information that a data-flow analysis compute is a set of points in a lattice. A dense analysis maps pairs of (program points × variables) to these points.
- We say that a data-flow analysis has the *single information property* for a given program if we can assign a single element of the underlying lattice to each variable, and this information is invariant along the entire live range of the variable.



EXAMPLES OF LIVE RANGE SPLITTING

DCC 888

Class Inference Analysis

- Some languages, such as Python, JavaScript and Lua, store the methods and fields of objects in hash-tables.
- It is possible to speedup programs written in these languages by replacing these tables by virtual tables[♧].
- A class inference engine tries to assign a virtual table to a variable *v* based on the ways that *v* is used.

```
def test(i):
  v = OX()
  if i % 2:
    tmp = i + 1
    v.m1(tmp)
  else:
    v = OY()
    v.m2()
  print v.m3()
```

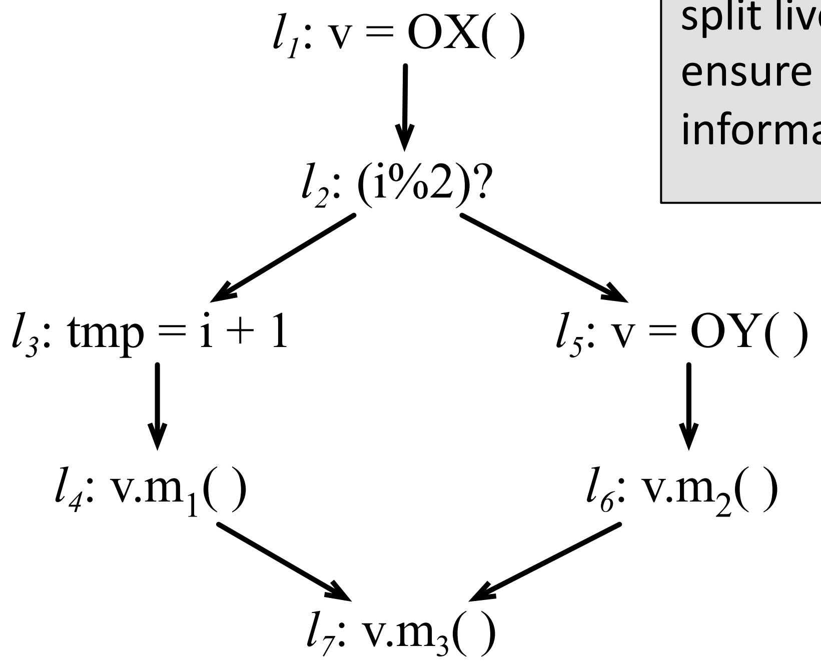
$$\begin{tikzcd}
 & l_1\text{: v = OX( )} \arrow[d] & \\
 & l_2\text{: (i\%2)?} \arrow[dl] \arrow[dr] & \\
l_3\text{: tmp = i + 1} \arrow[d] & & l_5\text{: v = OY( )} \arrow[d] \\
l_4\text{: v.m}_1\text{( )} \arrow[dr] & & l_6\text{: v.m}_2\text{( )} \arrow[dl] \\
 & l_7\text{: v.m}_3\text{( )} &
\end{tikzcd}$$

Consider the Python program on the right. Where is class inference information acquired?

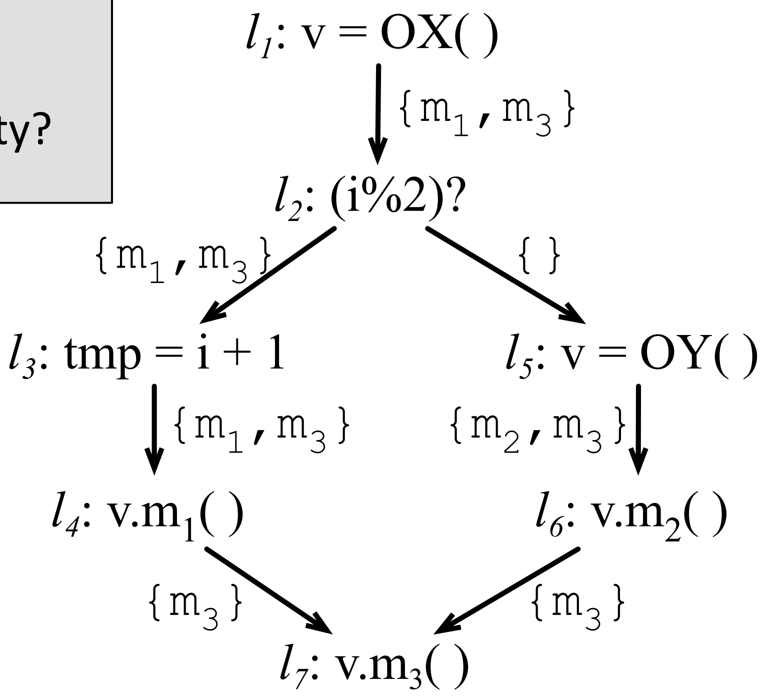
[♧]: Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language

Class Inference Analysis

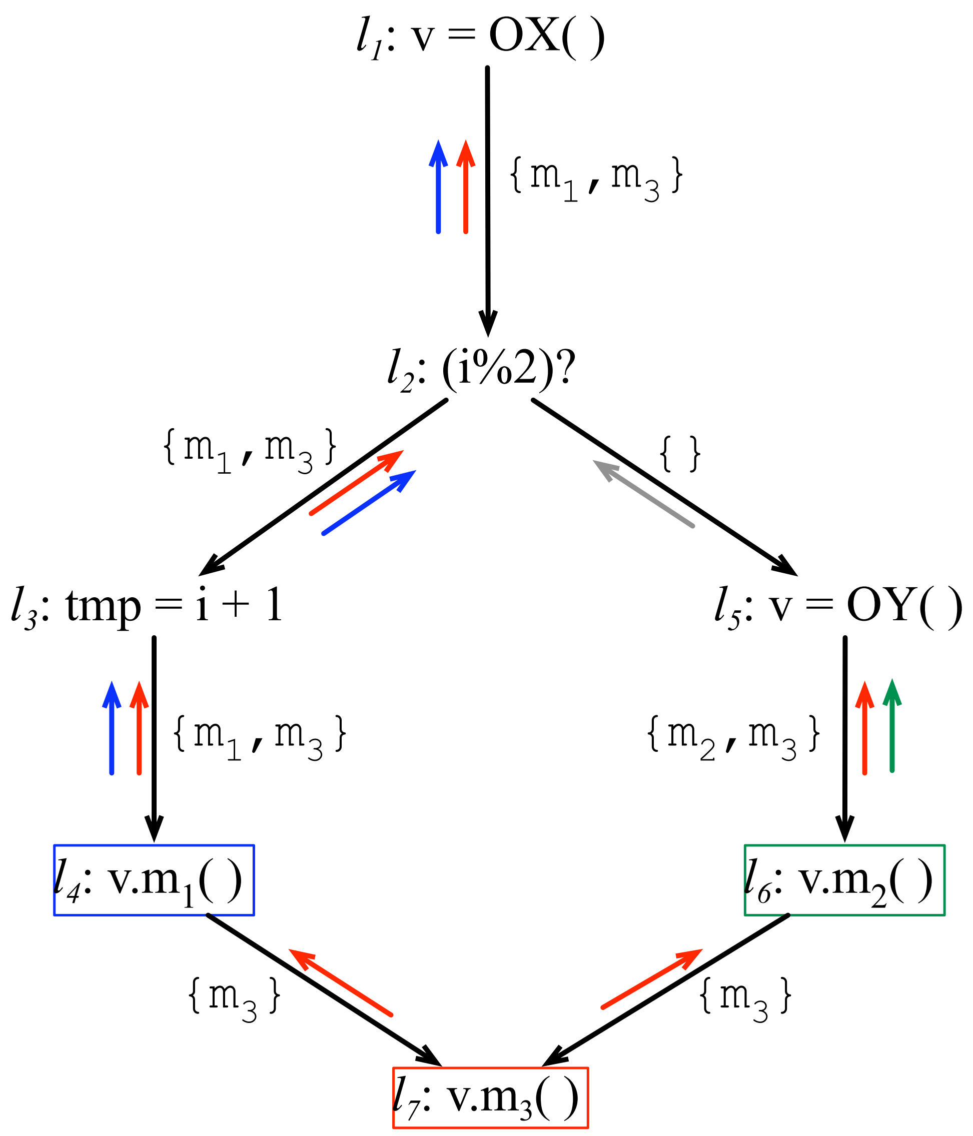
- We discover information based on the way an object is used. If the program calls a method *m* on an object *o*, then we assume that *m* is part of the class of *o*.



So, where should we split live ranges, to ensure the single information property?



The Points where We Acquire Information

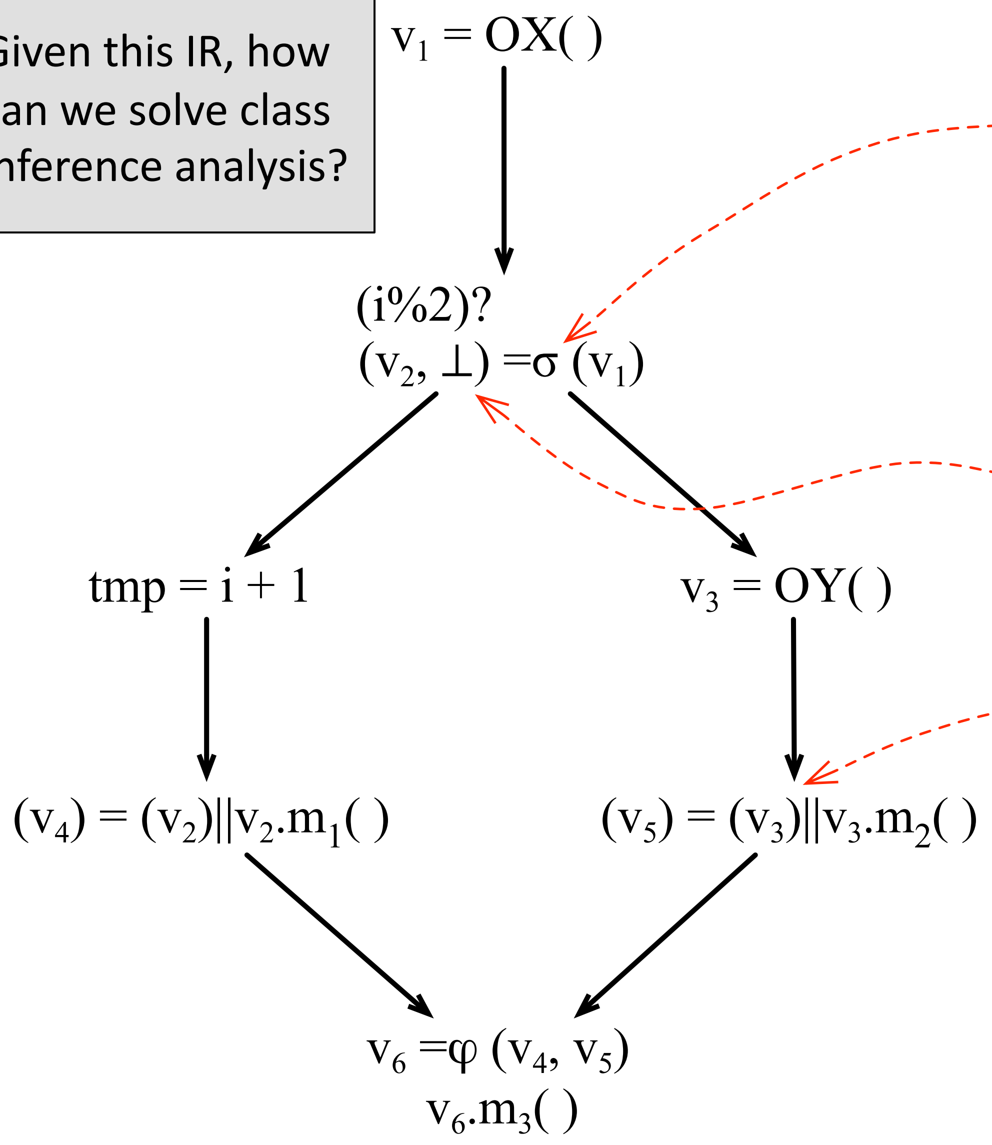


In the class inference analysis, information propagates backwards: if a method m is invoked on an object o at a program point p, we know that o must have the method m everywhere before p.

1) So, in the end, where is information acquired?
2) And where is information colliding?
3) How many variable names would we have in the transformed program?

Backward Live Range Splitting

Given this IR, how can we solve class inference analysis?

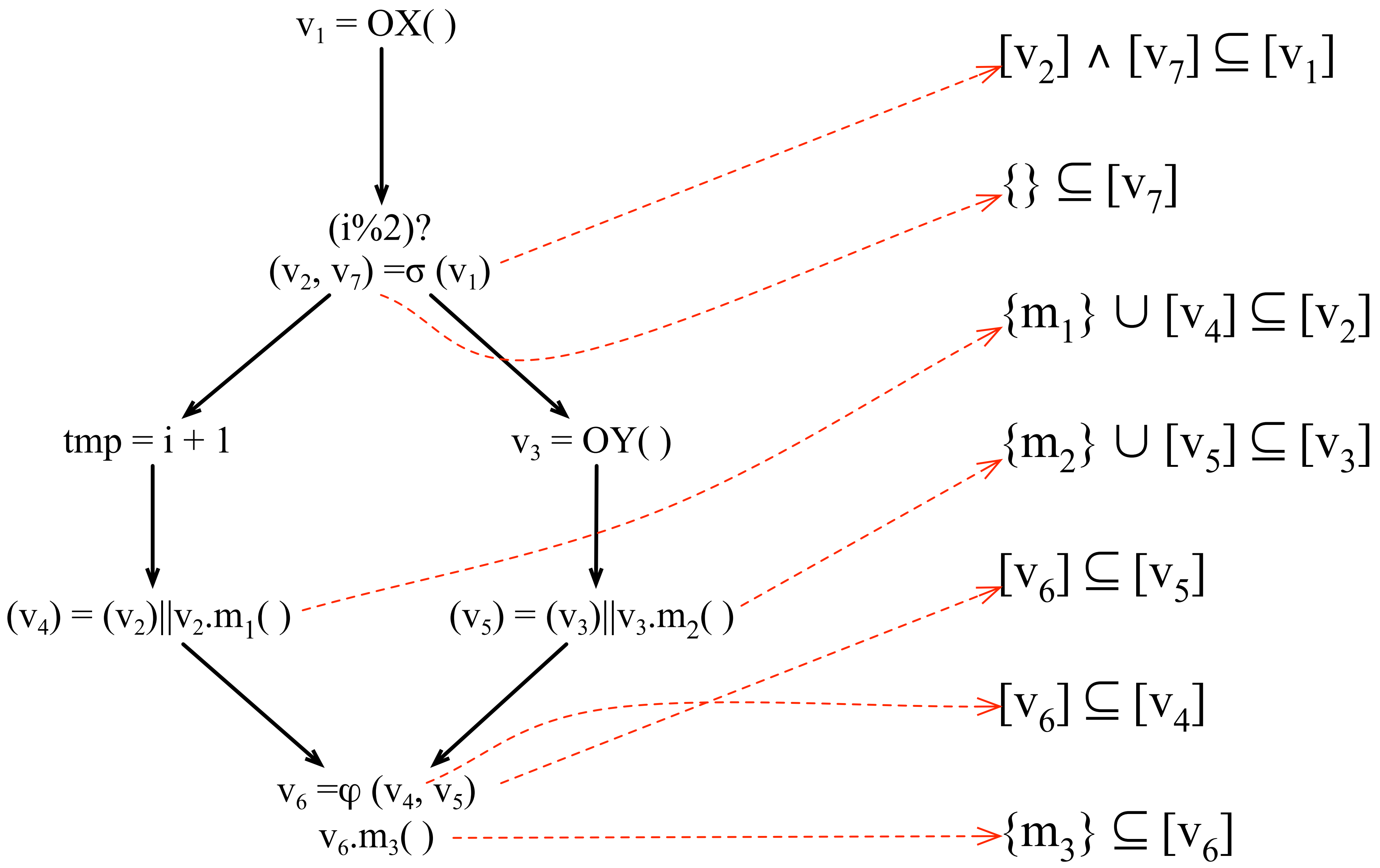


We represent these backward merge points using **these** special instructions, that we call sigma-functions, to distinguish them from the phi-functions used in the forward merge points.

No information is arriving at the second parameter of the sigma-function, thus we represent it as an undefined name, denoted by ⊥

We use **this notation** of parallel copies, e.g., ||, to indicate that there is some live range splitting happening at that program point. We shall talk more about this notation soon.

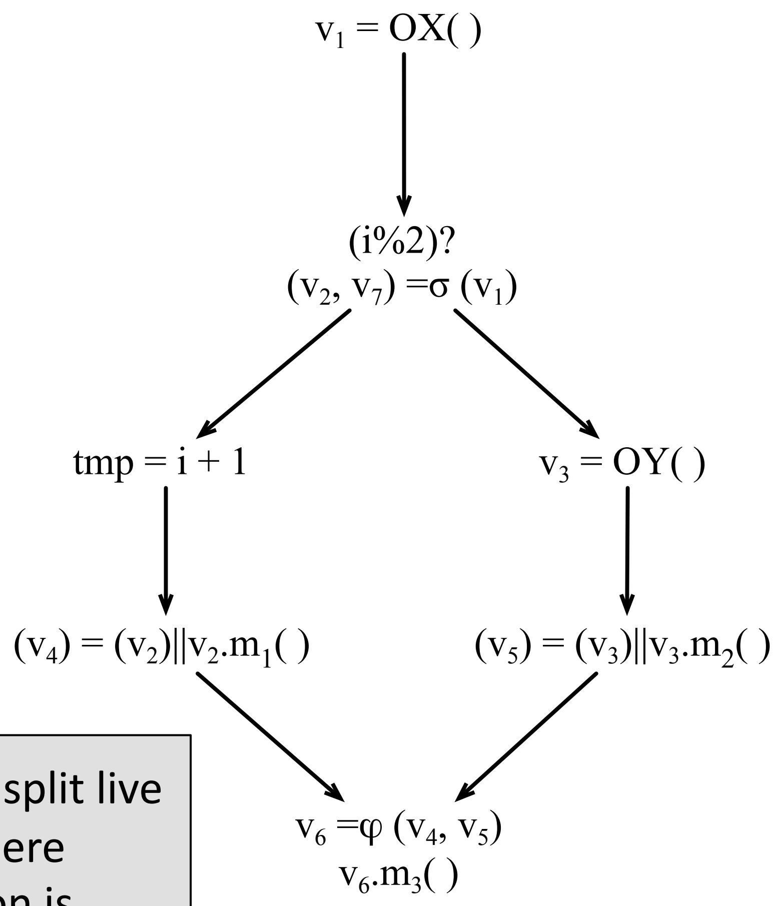
Class Inference as a Constraint System



v_1 = OX( )
(i%2)?
$(v_2, v_7) = \sigma (v_1)$
tmp = i + 1
v_3 = OY( )
$(v_4) = (v_2) \| v_2.m_1( )$
$(v_5) = (v_3) \| v_3.m_2( )$
$v_6 = \varphi (v_4, v_5)$
$v_6.m_3( )$
$[v_2] \wedge [v_7] \subseteq [v_1]$
$\{\} \subseteq [v_7]$
$\{m_1\} \cup [v_4] \subseteq [v_2]$
$\{m_2\} \cup [v_5] \subseteq [v_3]$
$[v_6] \subseteq [v_5]$
$[v_6] \subseteq [v_4]$
$\{m_3\} \subseteq [v_6]$

Is SSA Form Really Necessary?

- The representation that we have produced is in SSA form.
- It is very convenient to keep the program in this representation:
  - Helps other analyses.
  - Compiler already does it.
  - Fast liveness check♧.
- Yet, we do not really need SSA form for this particular backward analysis.



v_1 = OX( )
(i%2)?
$(v_2, v_7) = \sigma(v_1)$
tmp = i + 1
v_3 = OY( )
$(v_4) = (v_2) \| v_2.m_1(\,)$
$(v_5) = (v_3) \| v_3.m_2(\,)$
$v_6 = \varphi(v_4, v_5)$
$v_6.m_3(\,)$

If we only split live ranges where information is produced, how would our IR be?

♧: Fast Liveness Checking for SSA-Form Programs, CGO (2008)

A More Concise Representation

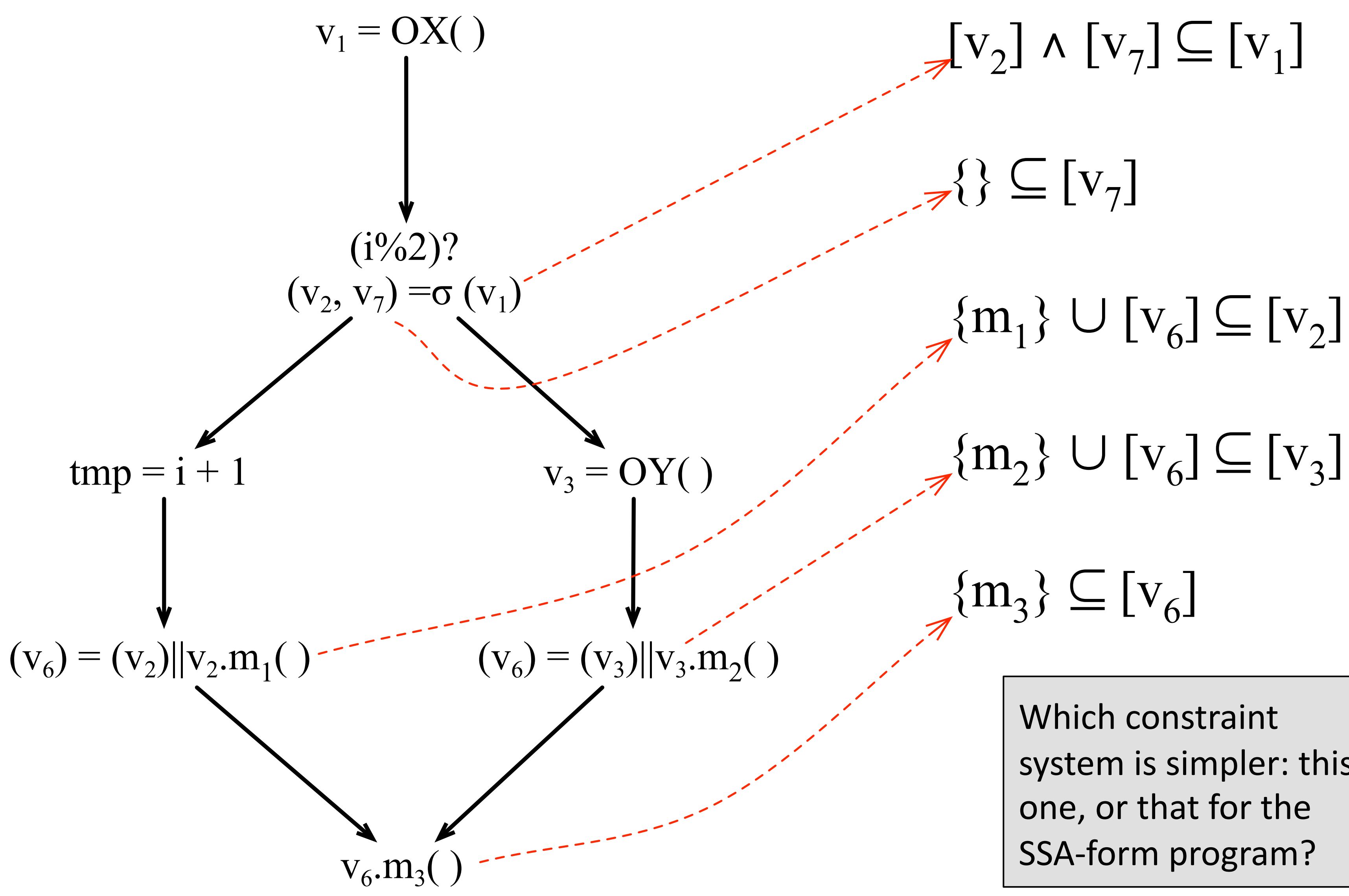
The program on the right is no longer in SSA form. It has fewer variable names.

How would be the constraint system for this new program representation?

$$\begin{tikzcd}
 & v_1 = OX(\,) \arrow[d] & \\
 & \begin{array}{c}(i\%2)? \\ (v_2, v_7) = \sigma\,(v_1)\end{array} \arrow[dl] \arrow[dr] & \\
tmp = i + 1 \arrow[d] & & v_3 = OY(\,) \arrow[d] \\
(v_4) = (v_2) \| v_2.m_1(\,) \arrow[dr] & & (v_5) = (v_3) \| v_3.m_2(\,) \arrow[dl] \\
 & \begin{array}{c}v_6 = \varphi\,(v_4, v_5) \\ v_6.m_3(\,)\end{array} &
\end{tikzcd}$$

$$\begin{tikzcd}
 & v_1 = OX(\,) \arrow[d] & \\
 & \begin{array}{c}(i\%2)? \\ (v_2, v_7) = \sigma\,(v_1)\end{array} \arrow[dl] \arrow[dr] & \\
tmp = i + 1 \arrow[d] & & v_3 = OY(\,) \arrow[d] \\
(v_6) = (v_2) \| v_2.m_1(\,) \arrow[dr] & & (v_6) = (v_3) \| v_3.m_2(\,) \arrow[dl] \\
 & v_6.m_3(\,) &
\end{tikzcd}$$

A Constraint System for the Non-SSA form Program


$v_1 = OX(\,)$
$(i\%2)?$
$(v_2, v_7) = \sigma\ (v_1)$
$tmp = i + 1$
$v_3 = OY(\,)$
$(v_6) = (v_2)\|v_2.m_1(\,)$
$(v_6) = (v_3)\|v_3.m_2(\,)$
$v_6.m_3(\,)$
$[v_2] \wedge [v_7] \subseteq [v_1]$
$\{\} \subseteq [v_7]$
$\{m_1\} \cup [v_6] \subseteq [v_2]$
$\{m_2\} \cup [v_6] \subseteq [v_3]$
$\{m_3\} \subseteq [v_6]$

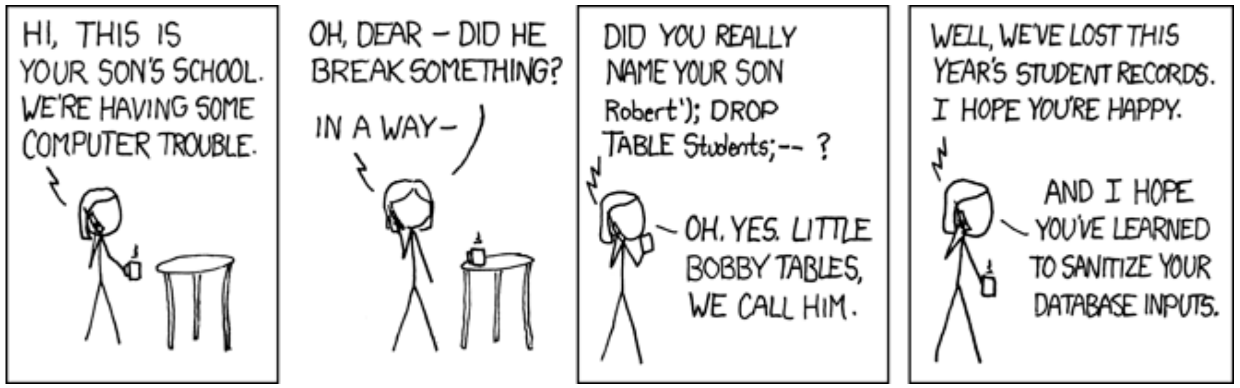
Which constraint system is simpler: this one, or that for the SSA-form program?

Tainted Flow Analysis

- The goal of the tainted flow analysis is to find out if there is a path from a source of malicious information to a sensitive operation.

1) Does the program below contain a tainted flow vulnerability?

2) To which kind of attack?


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

$$\begin{tikzcd}
l_1\text{: v = input( )} \arrow[d] & & l_2\text{: v = "Hi!"} \arrow[d] \\
l_3\text{: echo v} \arrow[dr] & & l_4\text{: echo v} \arrow[dl] \\
& l_5\text{: is v Clean?} \arrow[dl] \arrow[dr] & \\
l_7\text{: echo v} & & l_6\text{: echo v}
\end{tikzcd}$$

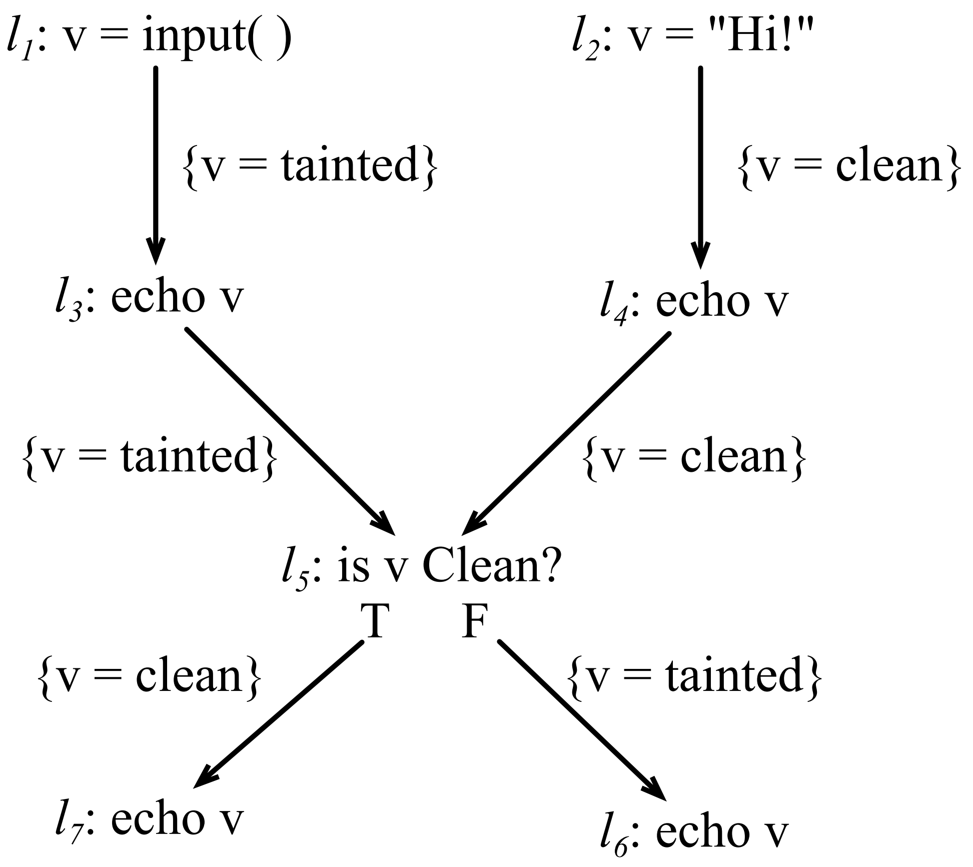
3) Where is new information acquired in this type of analysis?

4) How does this information propagate along the CFG?

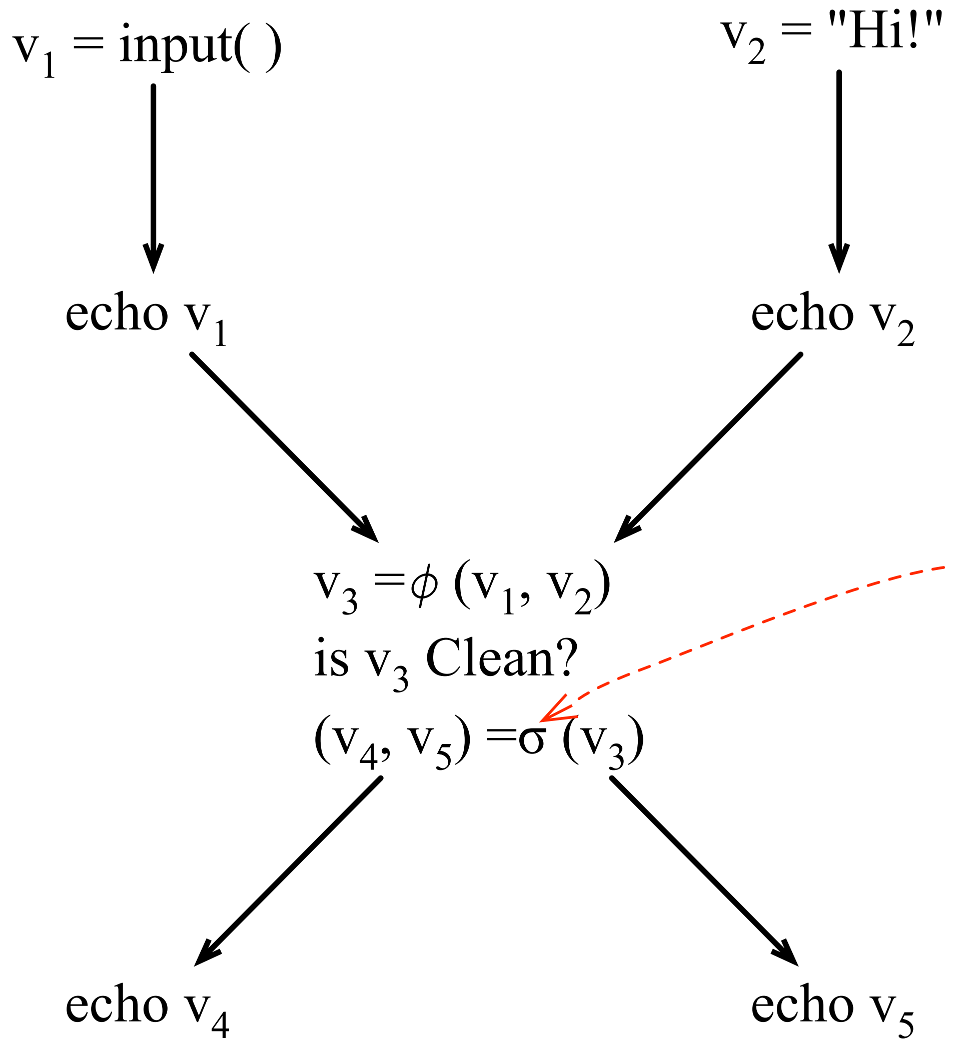
Conditional Tests

- Information in the tainted flow analysis propagates forwardly.
- And we can learn new facts from conditional tests.

So, where should we split live ranges, to ensure the single information property?



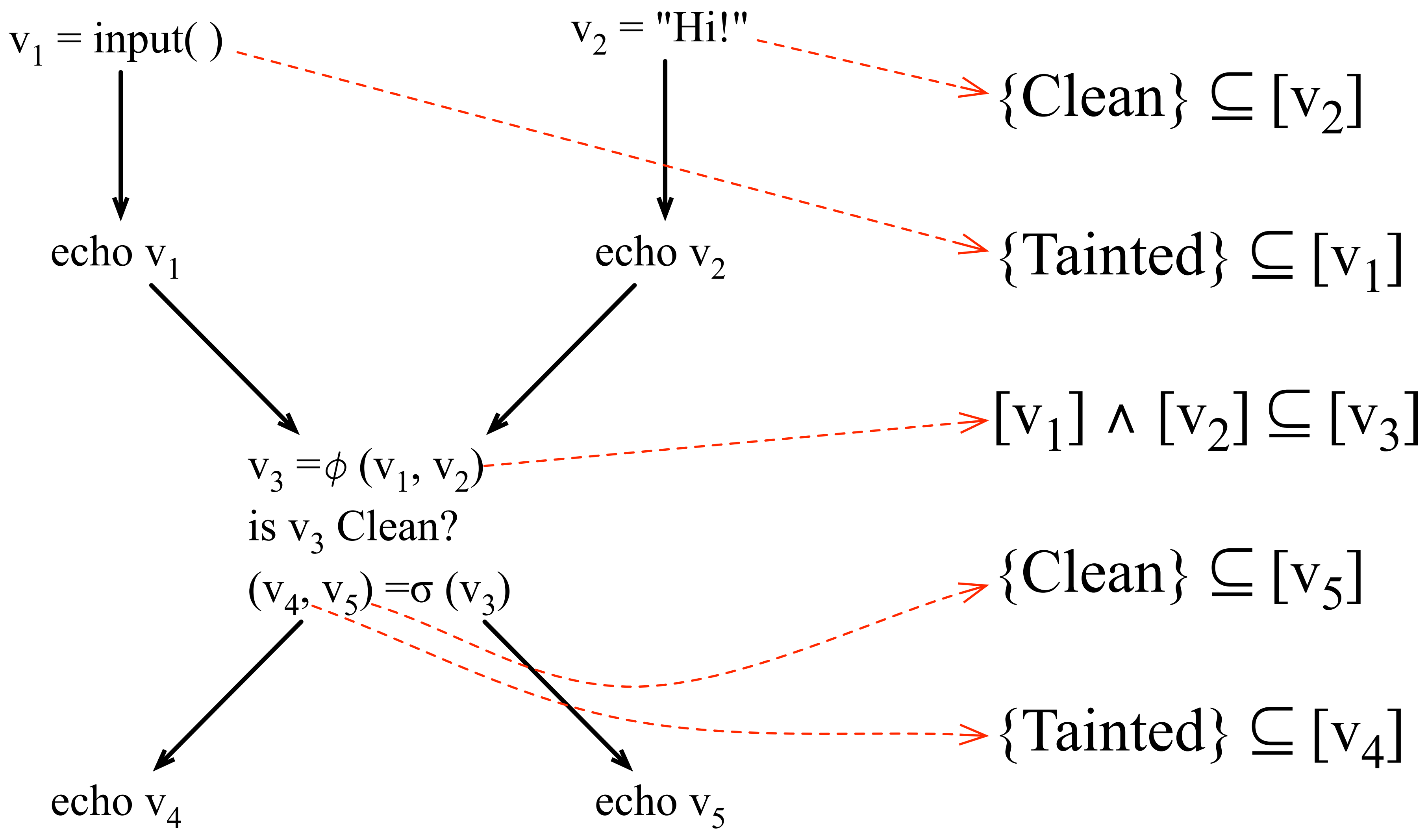
Splitting After Conditionals



And which constraints can we extract from this program representations?

We use sigma-functions after conditional tests, to split live ranges. In fact, we use sigma-functions whenever we need to copy the contents of a variable to multiple locations.

Tainted Analysis as a Constraint System



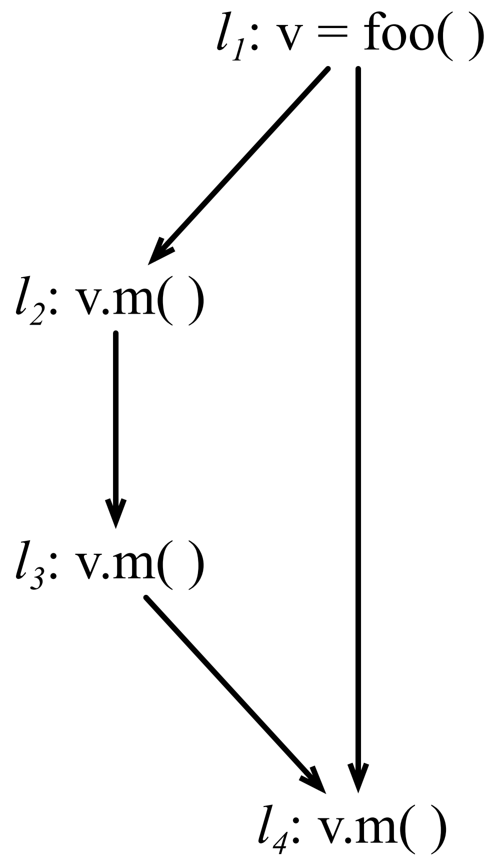
v_1 = input( )
v_2 = "Hi!"
echo v_1
echo v_2
$v_3 = \phi(v_1, v_2)$
is v_3 Clean?
$(v_4, v_5) = \sigma(v_3)$
echo v_4
echo v_5
$\{Clean\} \subseteq [v_2]$
$\{Tainted\} \subseteq [v_1]$
$[v_1] \wedge [v_2] \subseteq [v_3]$
$\{Clean\} \subseteq [v_5]$
$\{Tainted\} \subseteq [v_4]$

Null Pointer Analysis

- Type safe languages, such as Java, prefix method calls with a test that checks if the object is null. In this way, we cannot have segmentation faults, but only well-defined exceptions.

```
Exception in thread "main"
           java.lang.NullPointerException
   at Test.main(Test.java:7)
```

Consider the program on the right. Which calls can result in null pointer exceptions, and which calls are always safe?


$$\begin{tikzcd}
 & l_1\text{: v = foo( )} \arrow[dl] \arrow[ddd] \\
l_2\text{: v.m( )} \arrow[d] & \\
l_3\text{: v.m( )} \arrow[dr] & \\
 & l_4\text{: v.m( )}
\end{tikzcd}$$

Null Pointer Analysis

1) Can the call at l_4 result in a null pointer exception?

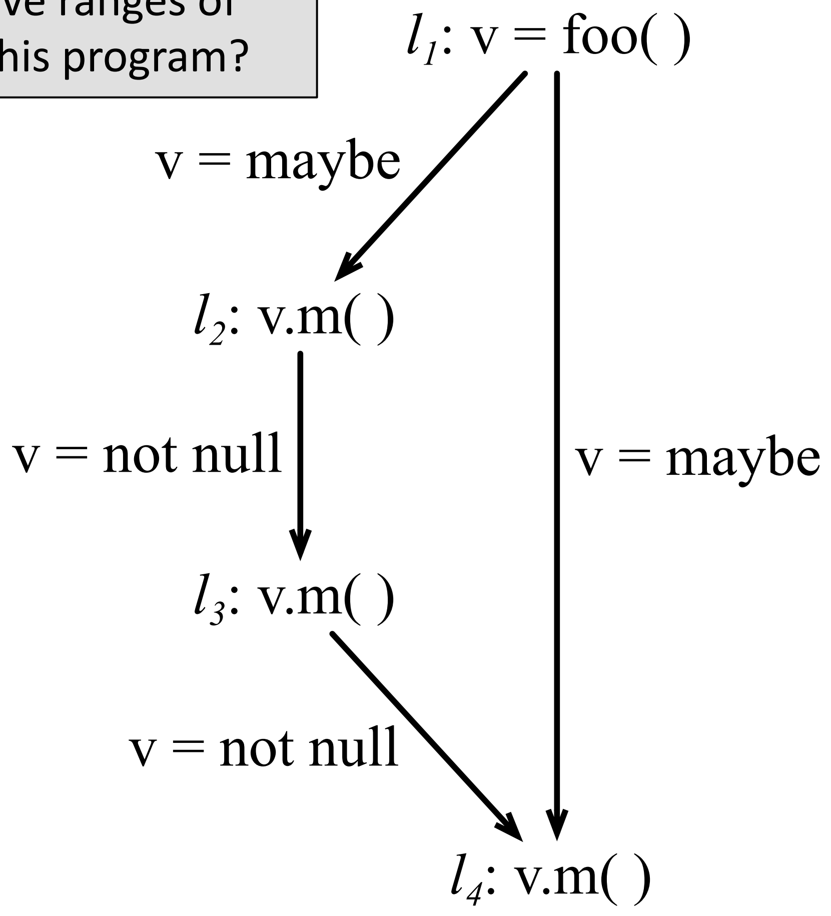
2) Where is information produced?

Java, being a programming language that is type safe, must ensure – dynamically – that every call to an object that is null fires an exception.

3) How is a null pointer check implemented?

4) What do we gain by eliminating these tests?

5) Can you split the live ranges of this program?

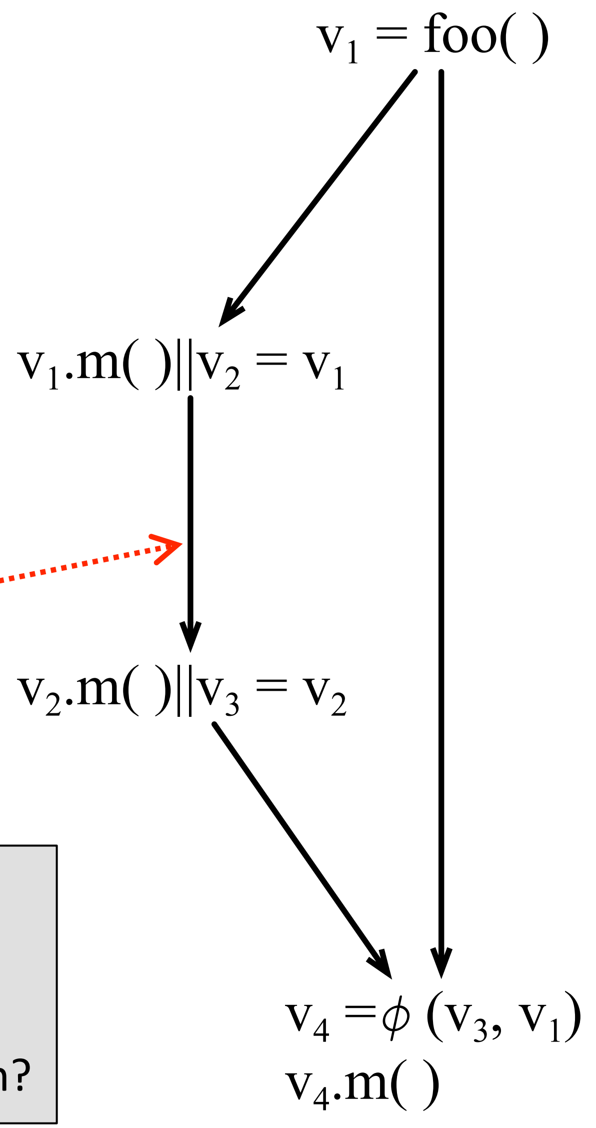


$$\begin{tikzcd}
 & l_1: v = foo() \arrow[dl, "v = maybe"'] \arrow[ddd, "v = maybe"] \\
l_2: v.m() \arrow[d, "v = not null"'] & \\
l_3: v.m() \arrow[dr, "v = not null"'] & \\
 & l_4: v.m()
\end{tikzcd}$$

Splitting After Uses

In the case of null check elimination, we split live ranges after uses. If we invoke a method *m* on an object *o*, and no exception happens, then we know that *m* can be safely invoked on *o* in any program point after the use point.

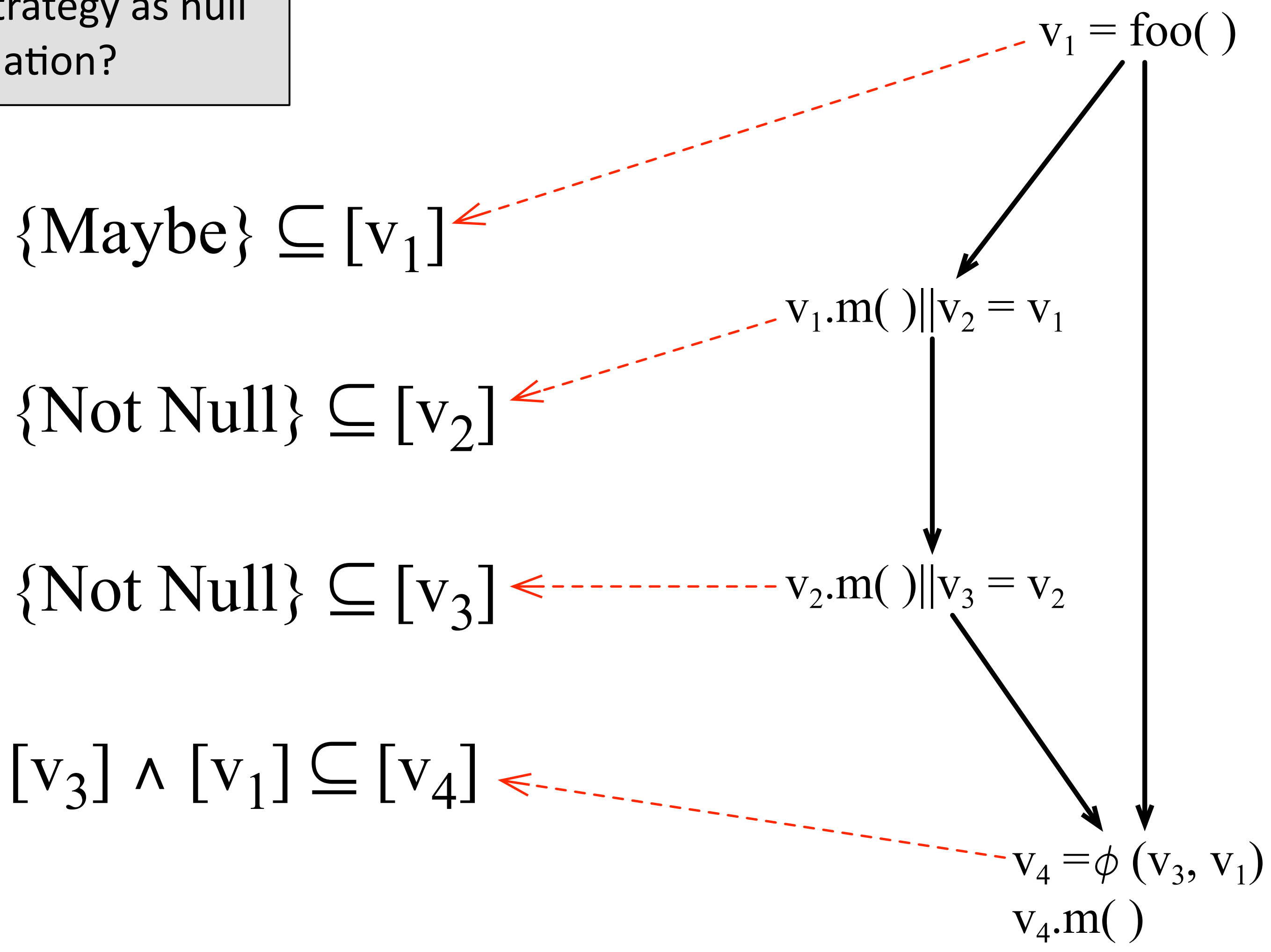
As an example, an exception can never happen at this point, because otherwise it would have happened at the previous use of v_1, e.g., v_1.m()



How can we transform this problem in a constraint system?

Null Pointer Check Elimination as Constraints

Can you think about other analyses that would use the same live range splitting strategy as null pointer check elimination?



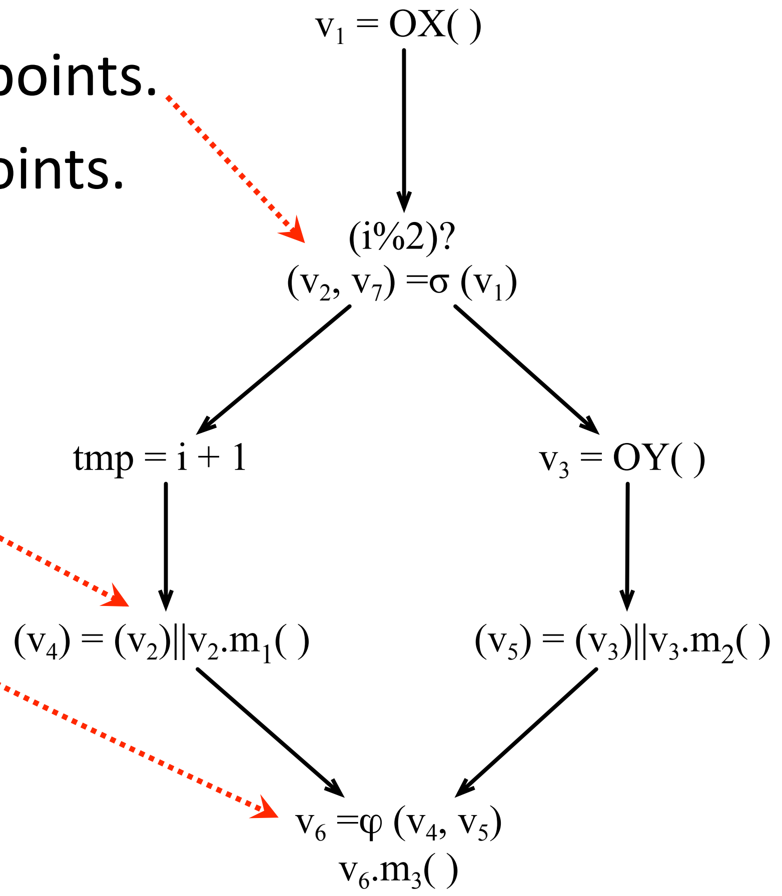
$v_1 = foo()$
$\{Maybe\} \subseteq [v_1]$
$v_1.m() \| v_2 = v_1$
$\{Not\ Null\} \subseteq [v_2]$
$v_2.m() \| v_3 = v_2$
$\{Not\ Null\} \subseteq [v_3]$
$[v_3] \wedge [v_1] \subseteq [v_4]$
$v_4 = \phi(v_3, v_1)$
$v_4.m()$

LIVE RANGE SPLITTING STRATEGIES

DCC 888

Live Range Splitting Notation

- We use three special notations to indicate that we are splitting live ranges.
  - Phi-functions split at join points.
  - Sigma-functions split at branch points.
  - Parallel copies split at interior points.

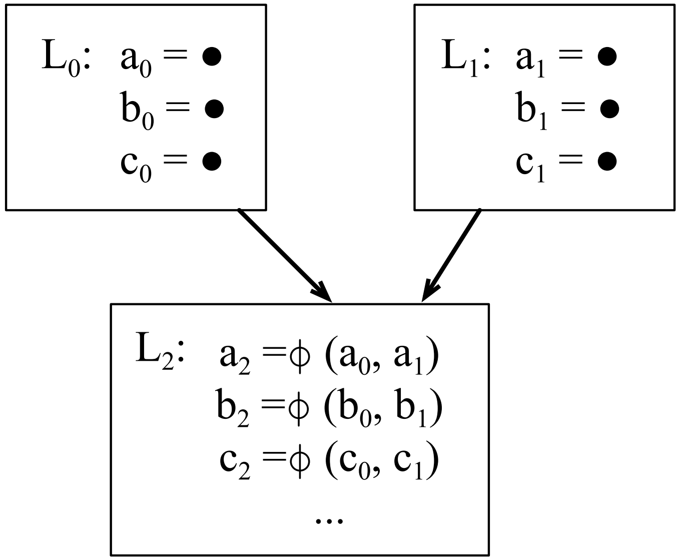

v_1 = OX( )
(i%2)?
$(v_2, v_7) = \sigma(v_1)$
tmp = i + 1
v_3 = OY( )
$(v_4) = (v_2) \| v_2.m_1()$
$(v_5) = (v_3) \| v_3.m_2()$
$v_6 = \varphi(v_4, v_5)$
$v_6.m_3()$

Parallel Copies

- We split live ranges at interior nodes via parallel copies.
- Usually there is no assembly equivalent of a parallel copy; nevertheless, we can implement them with move or swap instructions.
- Semantically, we have a program point p like: $(a_1 = a_0) \mid\mid (b_1 = b_0) \mid\mid x_1 = f(y_0, z_0)$, we read all the variables used at p, e.g., a_0, b_0, y_0, z_0, then write all the variables defined at p, e.g., a_1, b_1, x_1.
  - All the reads happen in parallel.
  - All the writes happen in parallel.

Phi-Functions

- Phi-functions work as multiplexers: they choose a copy based on the program flow.
- We may have multiple phi-functions at the beginning of a basic-block.
- If we have N phi-functions with M arguments at a basic block, then we have M parallel copies of N variables.

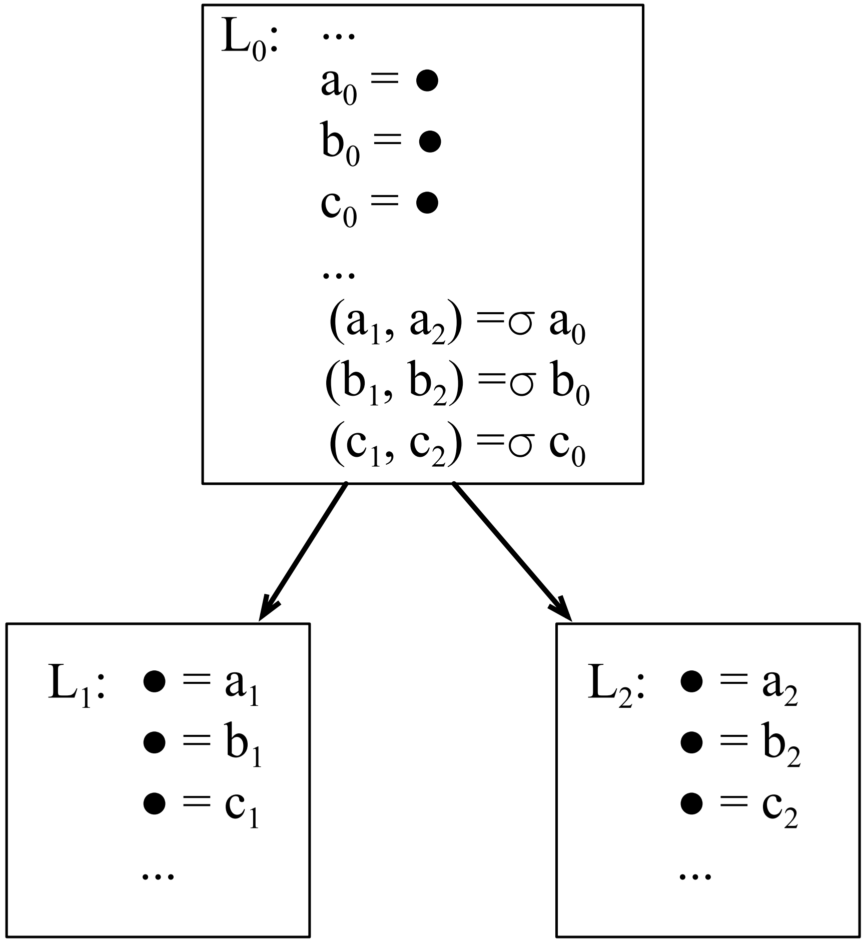


In this example, if control reaches L_2 from L_0, then we have the parallel copy $(a_2 = a_0)||(b_2 = b_0)||(c_2 = c_0)$; otherwise we have the parallel copy $(a_2 = a_1)||(b_2 = b_1)||(c_2 = c_1)$

Sigma-Functions

- Sigma-Functions work as demultiplexers: they choose a parallel copy based on the path taken at a branch.

In this example, if we branch from L_0 to L_1, then the following parallel copy takes place: $(a_1 = a_0)||(b_1 = b_0)||(c_1 = c_0)$; otherwise we have the parallel copy $(a_2 = a_0)||(b_2 = b_0)||(c_2 = c_0)$



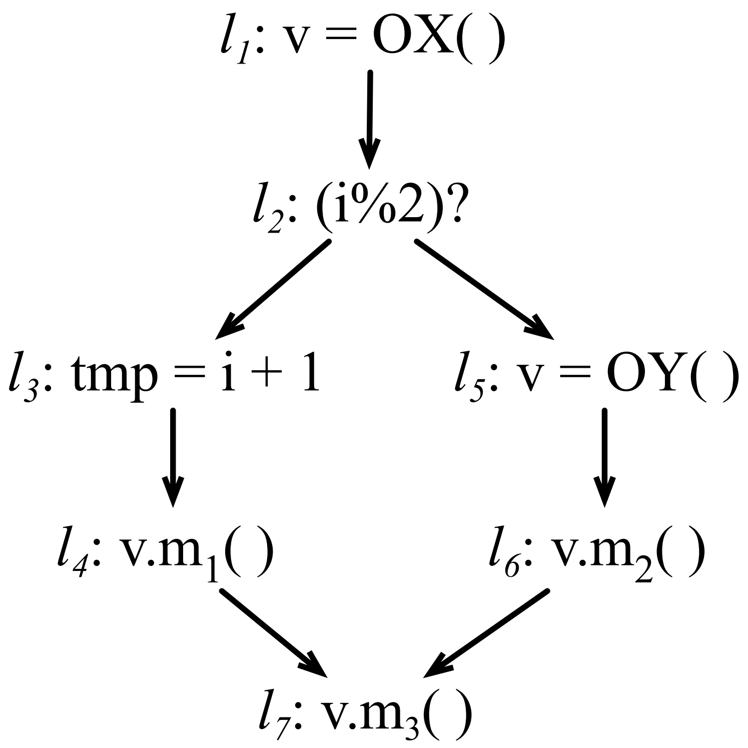
Live Range Splitting Strategy

- A live range splitting strategy for variable v, is a set $P_v = I_\uparrow \cup I_\downarrow$ of "oriented" program points.
- The set $I_\downarrow$ denotes points that produce information that propagates forwardly.
- The set $I_\uparrow$ denotes points that produce information that propagates backwardly.
- Live ranges must be split at least in every point of P_v.

And each split point may cause further splitting, as we will see soon.

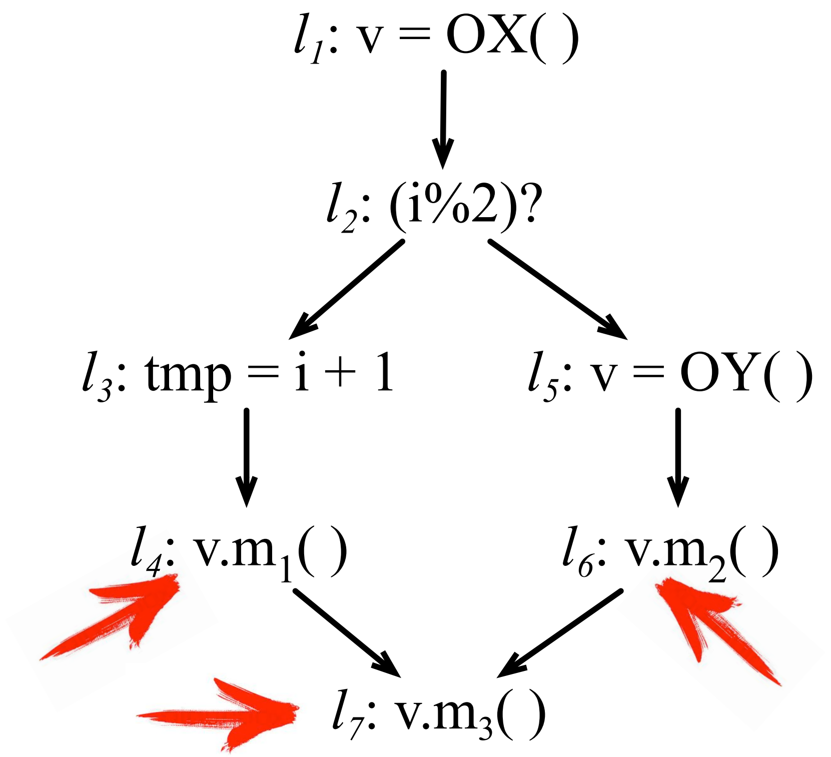
We have seen **this** kind of cascading behavior, when converting a program to SSA form. Do you remember the exact situation?

Quiz Time: Class Inference


$$\begin{tikzcd}
 & l_1\text{: v = OX( )} \arrow[d] & \\
 & l_2\text{: (i\%2)?} \arrow[dl] \arrow[dr] & \\
l_3\text{: tmp = i + 1} \arrow[d] & & l_5\text{: v = OY( )} \arrow[d] \\
l_4\text{: v.}m_1\text{( )} \arrow[dr] & & l_6\text{: v.}m_2\text{( )} \arrow[dl] \\
 & l_7\text{: v.}m_3\text{( )} &
\end{tikzcd}$$

What is the live range splitting strategy for the class inference analysis of our original example?

Quiz Time: Class Inference


$$\begin{tikzcd}
 & l_1\text{: v = OX( )} \arrow[d] & \\
 & l_2\text{: (i\%2)?} \arrow[dl] \arrow[dr] & \\
l_3\text{: tmp = i + 1} \arrow[d] & & l_5\text{: v = OY( )} \arrow[d] \\
l_4\text{: v.m}_1\text{( )} \arrow[dr] & & l_6\text{: v.m}_2\text{( )} \arrow[dl] \\
 & l_7\text{: v.m}_3\text{( )} &
\end{tikzcd}$$

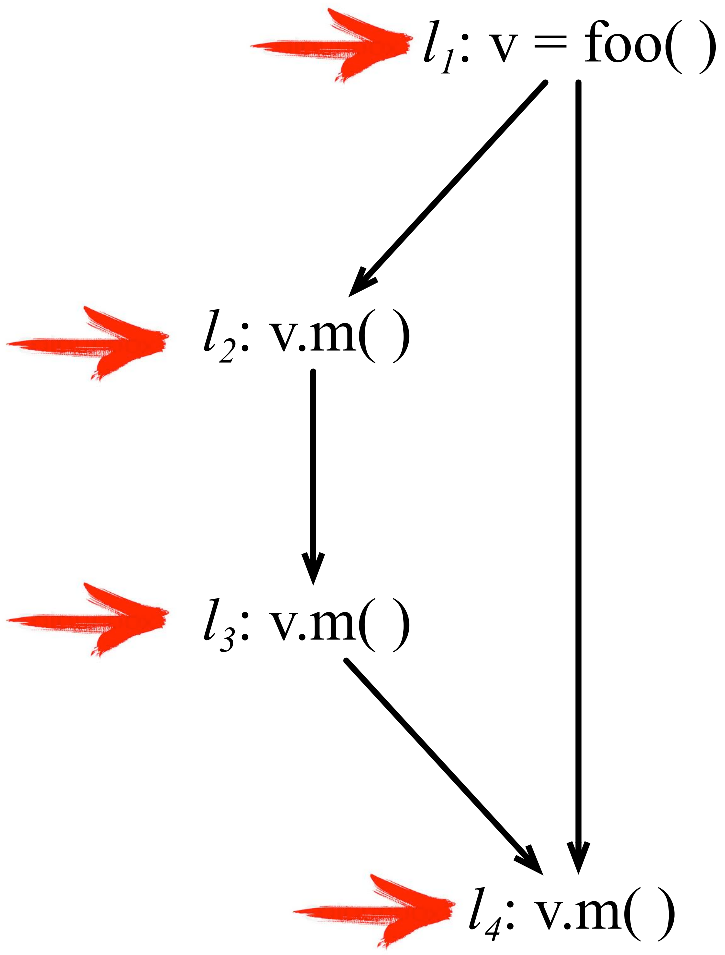
Class inference acquires information from uses, and propagates it backwardly. This gives us the following live range splitting strategy: $P_v = \{l_4, l_6, l_7\}\uparrow$

Quiz Time: Null Pointer Check Elimination

$$\begin{tikzcd}
 & l_1\text{: v = foo( )} \arrow[dl] \arrow[ddd] \\
l_2\text{: v.m( )} \arrow[d] & \\
l_3\text{: v.m( )} \arrow[dr] & \\
 & l_4\text{: v.m( )}
\end{tikzcd}$$

What is the live range splitting strategy for our "null pointer check elimination" analysis?

Quiz Time: Null Pointer Check Elimination


$$\begin{tikzcd} & l_1\text{: v = foo( )} \arrow[dl] \arrow[ddd] \\ l_2\text{: v.m( )} \arrow[d] & \\ l_3\text{: v.m( )} \arrow[dr] & \\ & l_4\text{: v.m( )} \end{tikzcd}$$

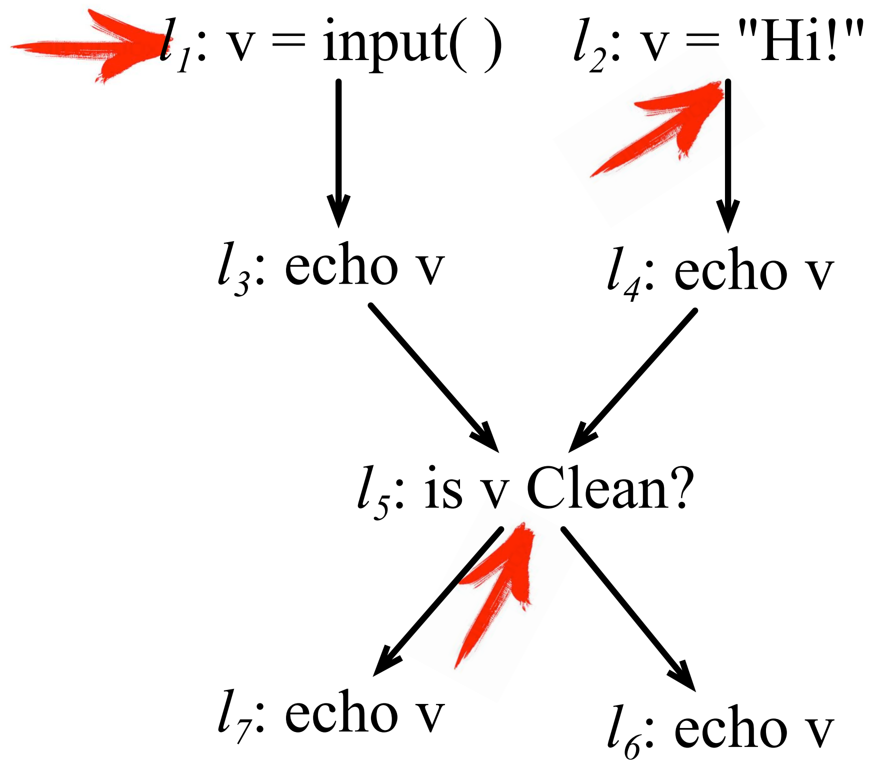
Null pointer check elimination takes information from the uses, and from the definition of variables, and propagates it forwardly. This gives us the following live range splitting strategy: $P_v = \{l_1, l_2, l_3, l_4\}_\downarrow$

Quiz Time: Tainted Flow Analysis

$$\begin{tikzcd}
l_1\text{: v = input( )} \arrow[d] & l_2\text{: v = "Hi!"} \arrow[d] \\
l_3\text{: echo v} \arrow[dr] & l_4\text{: echo v} \arrow[d] \\
 & l_5\text{: is v Clean?} \arrow[dl] \arrow[d] \\
l_7\text{: echo v} & l_6\text{: echo v}
\end{tikzcd}$$

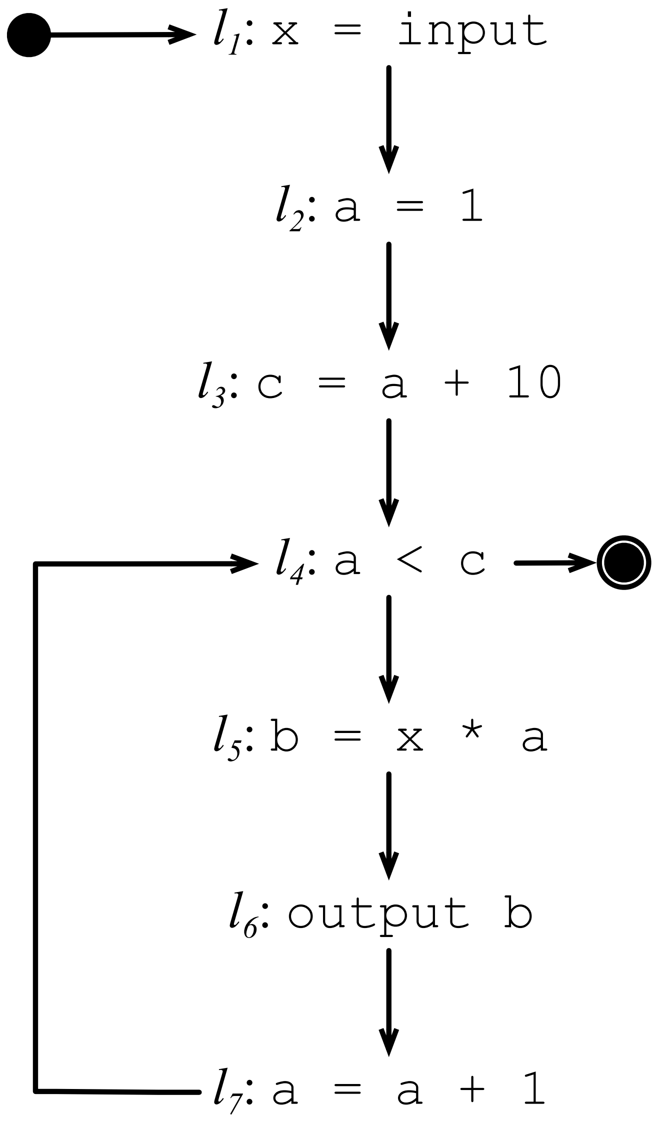
What is the live range splitting strategy for our tainted flow analysis, considering our example?

Quiz Time: Tainted Flow Analysis



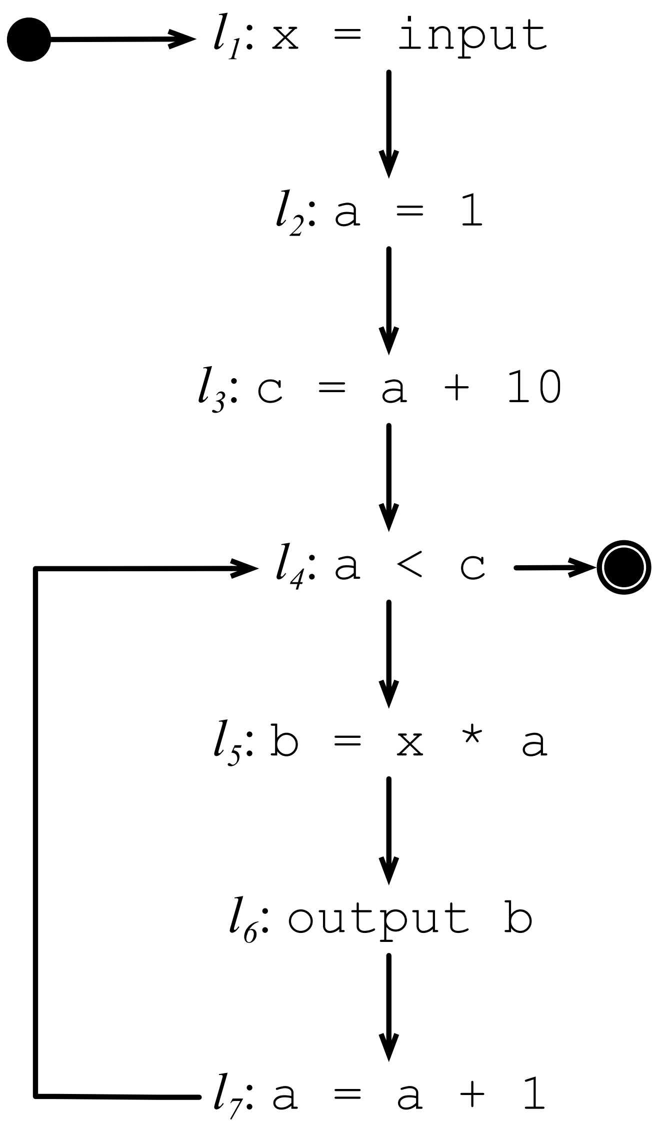
Tainted flow analysis takes information from definitions and conditional tests, and propagates this information forwardly. This gives us the following live range splitting strategy: $P_v = \{l_1, l_2, out(l_5)\}_\downarrow$. We let $out(l_5)$ denote the program point after l_5.

Quiz Time: Constant Propagation



Finally, what is the live range splitting strategy if we decide to do constant propagation in this example on the left?

Quiz Time: Constant Propagation

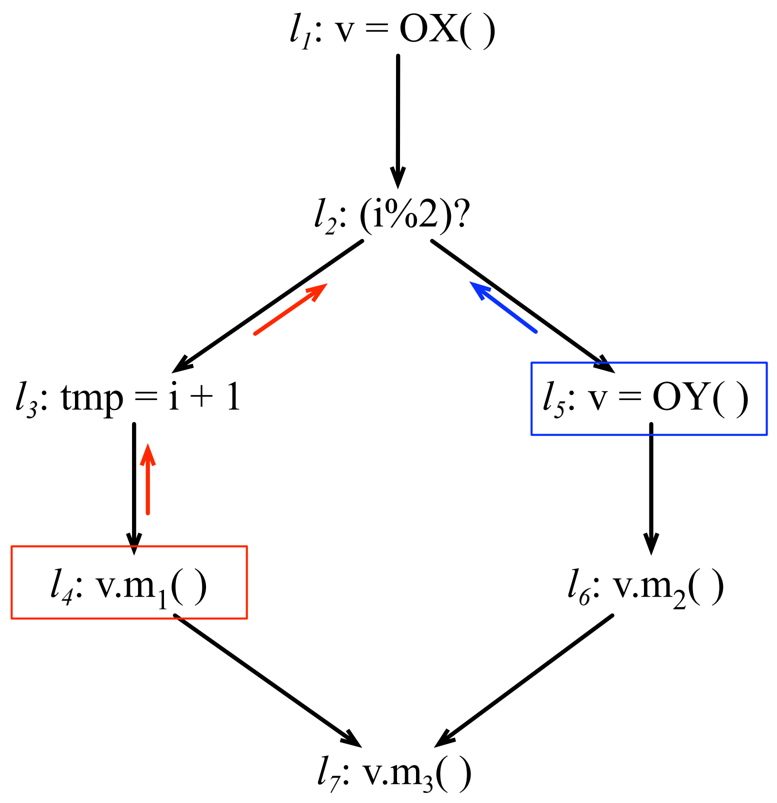


There are two main implementations of constant propagation. The simplest one just takes information from definitions. The more involved one♤ uses also the result of equality tests to find out if a variable is constant or not. Assuming the simplest kind, we would have the following strategies: $P_x = \{l_1\}_\downarrow$, $P_a = \{l_2, l_7\}_\downarrow$, $P_c = \{l_3\}_\downarrow$ and $P_b = \{l_5\}_\downarrow$. Notice that we have to split live ranges for each different variable in our program.

♤: Constant propagation with conditional branches

Meet Nodes

- Consider a forward (resp. backward) data-flow problem where S(p, v) is the maximum fixed point solution for variable v and program point p.
- We say that a program point m is a meet node for variable v if, and only if, m has n predecessors (resp. successors), $s_1, \ldots, s_n$, $n > 2$, and there exists i, j, $1 \leq i < j \leq n$, such that $S(s_i, v) \neq S(s_j, v)$.



Notice that there is no cheap way to compute the meet nodes by just considering the structure of the program. Why?

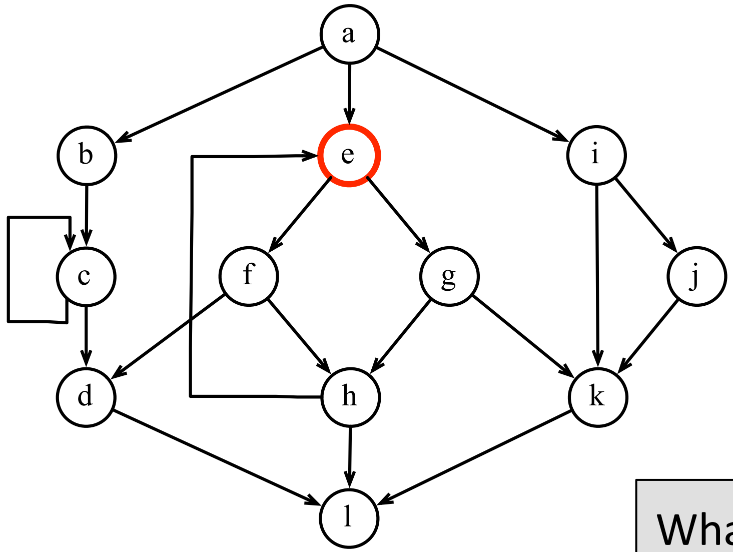
Computing Meet Nodes

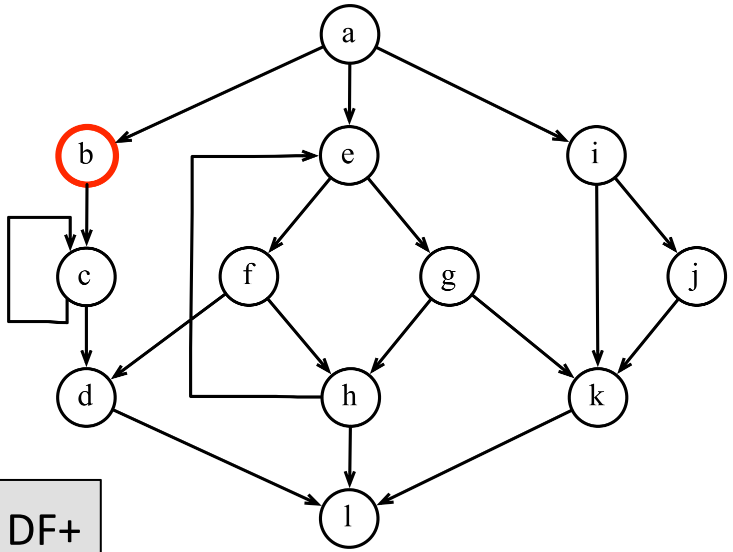
- In order to know the meet nodes, we must solve the analysis.
  - But we need them to solve the analysis!
- So we will just approximate the set of meet nodes, using only structural properties of the program's CFG:
  - **Dominance**: a CFG node n dominates a node n' if every program path from the entry node of the CFG to n' goes across n. If n ≠ n', then we say that n strictly dominates n'
  - **Dominance frontier** (DF): a node n' is in the dominance frontier of a node n if n dominates a predecessor of n', but does not strictly dominate n'.
  - **Iterated dominance frontier** (DF+): the iterated dominance frontier of a node n is the fixed point of the sequence:

$$DF_1(n) = DF(n)$$

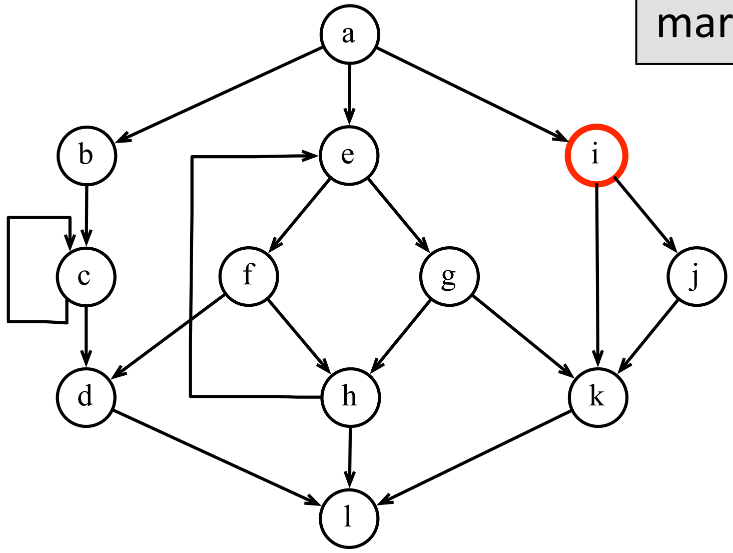
$$DF_{i+1}(n) = DF_i(n) \cup \{DF(z) \mid z \in DF_i(n)\}$$

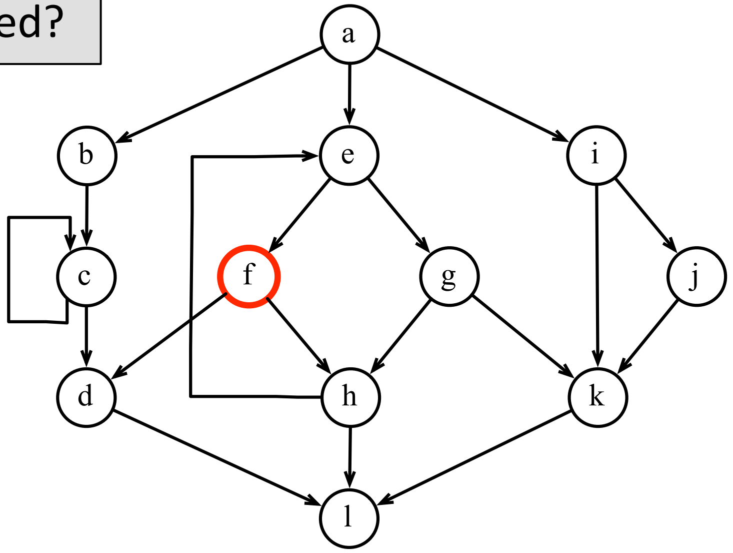
Iterated Dominance Frontier



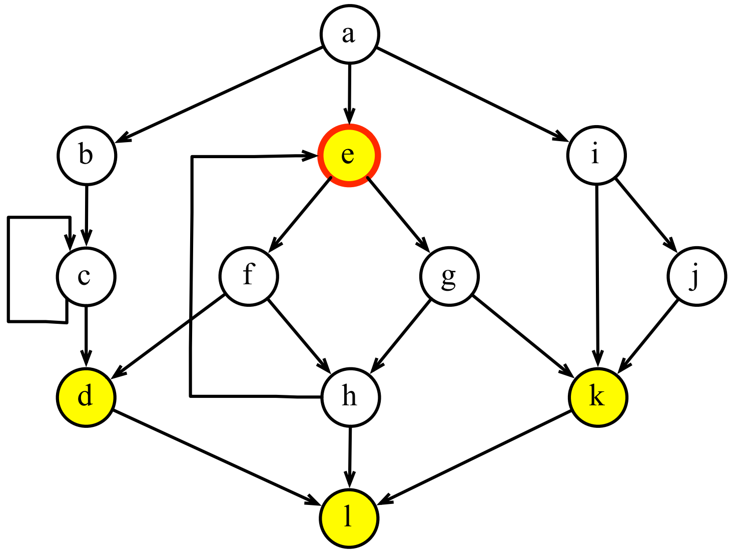


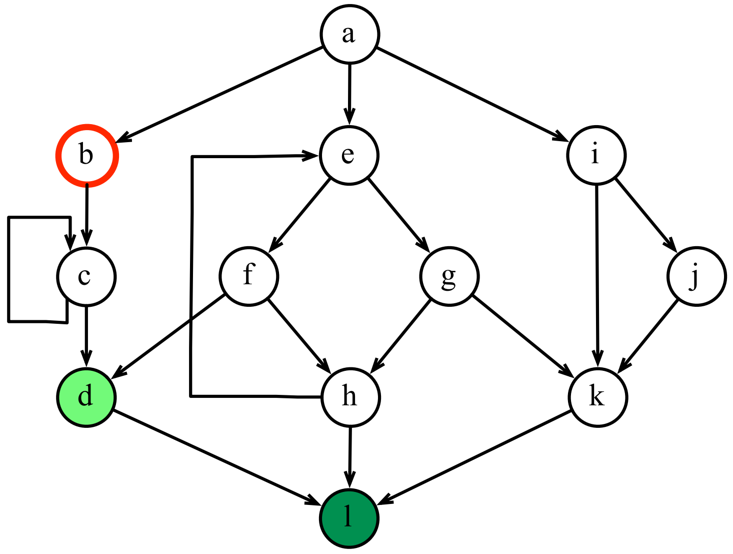
What is the DF+ of each node marked in red?

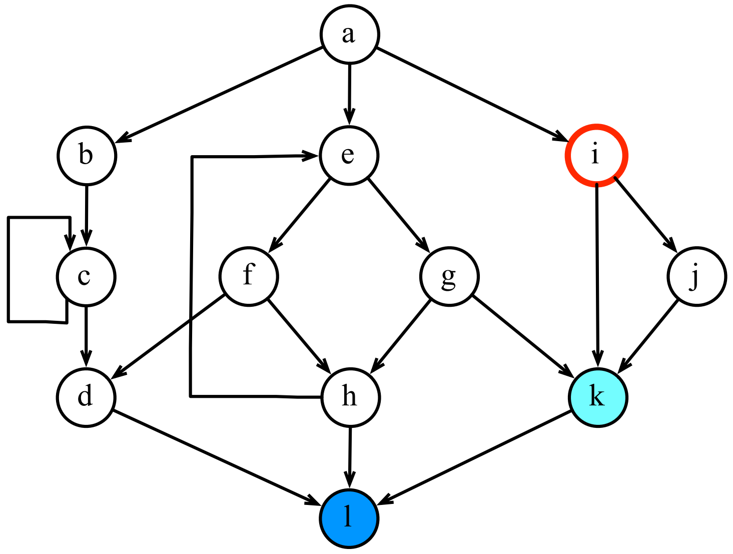


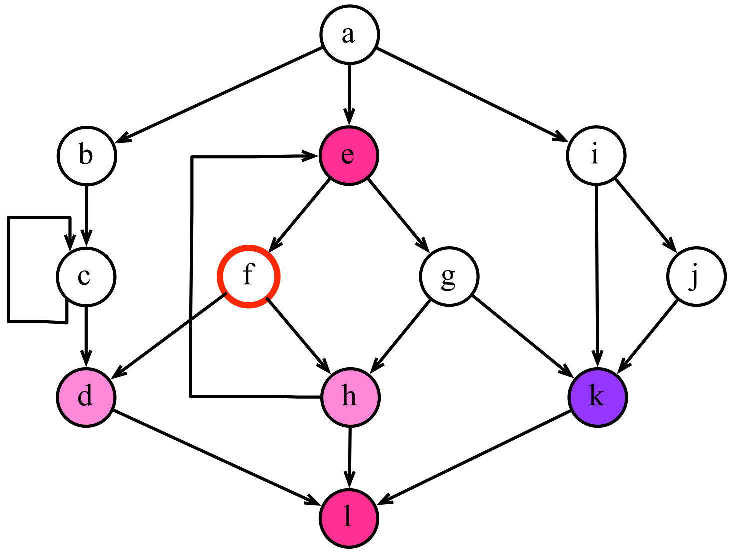


Iterated Dominance Frontier





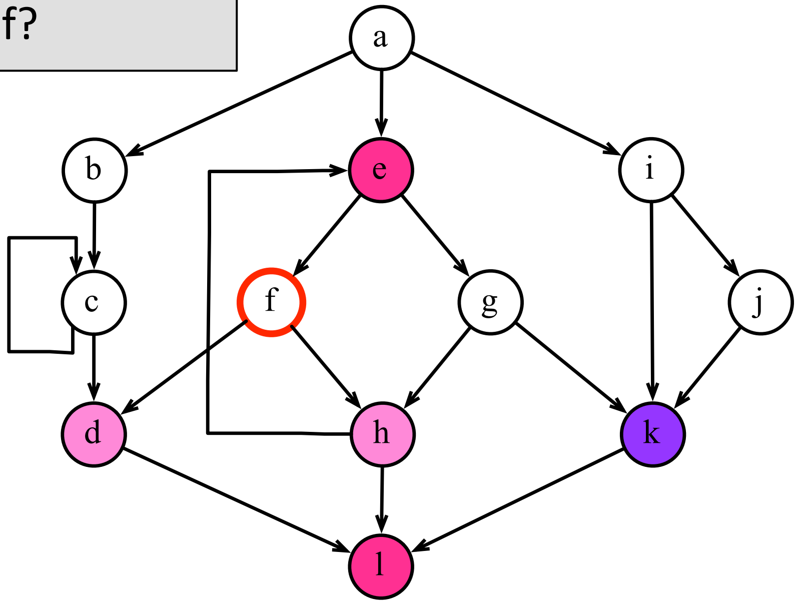




Splitting at Meet Nodes

- If information is produced at a point p, then we must recombine this information at the iterated dominance frontier of *p*.
- In this case, we are approximating meet nodes as the nodes in the DF+(*p*).
- We split the live range of a variable v at meet nodes, for a forward analysis, in the same way we did it when building the Static Single Assignment form:
  - If we have a node *n* with *m* predecessors, then we create a phi-function at *n*, with *m* entries.

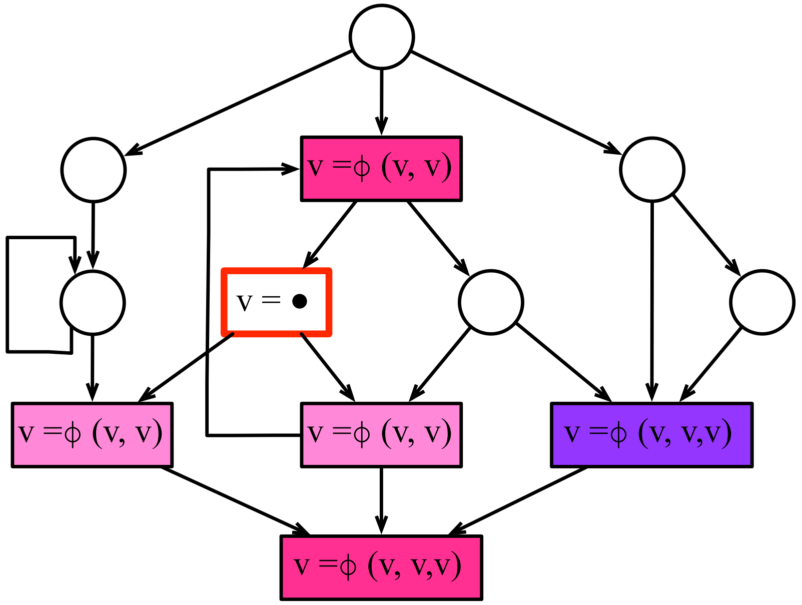
How would we split if we had a definition of v at f?



Splitting at Meet Nodes

- If information is produced at a point p, then we must recombine this information at the iterated dominance frontier of *p*.
- In this case, we are approximating meet nodes as the nodes in the DF+(*p*).

1) When are all these copies really necessary?
2) How (and under which assumptions) could we remove some of these copies?

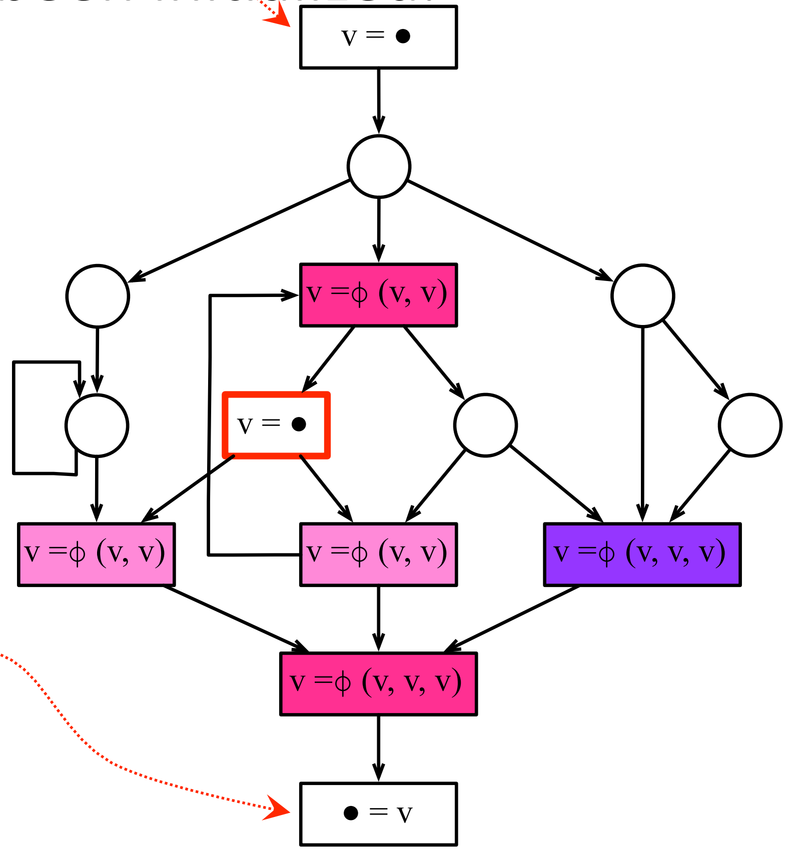


Pseudo-Defs and Pseudo-Uses at Extreme Nodes

- We can assume that we have a pseudo-definition of the variable, at the entry node of the Control Flow Graph.
- This pseudo-definition avoids the problem of paths in which a variable might not have been initialized.

But, in any case, we only need to split live ranges where these live ranges exist, i.g., where the variable is alive.

Nevertheless, it is not wrong to insert phi-functions at places where the variable is not alive. We can simplify matters a little bit assuming a pseudo use at the exit node of the CFG.



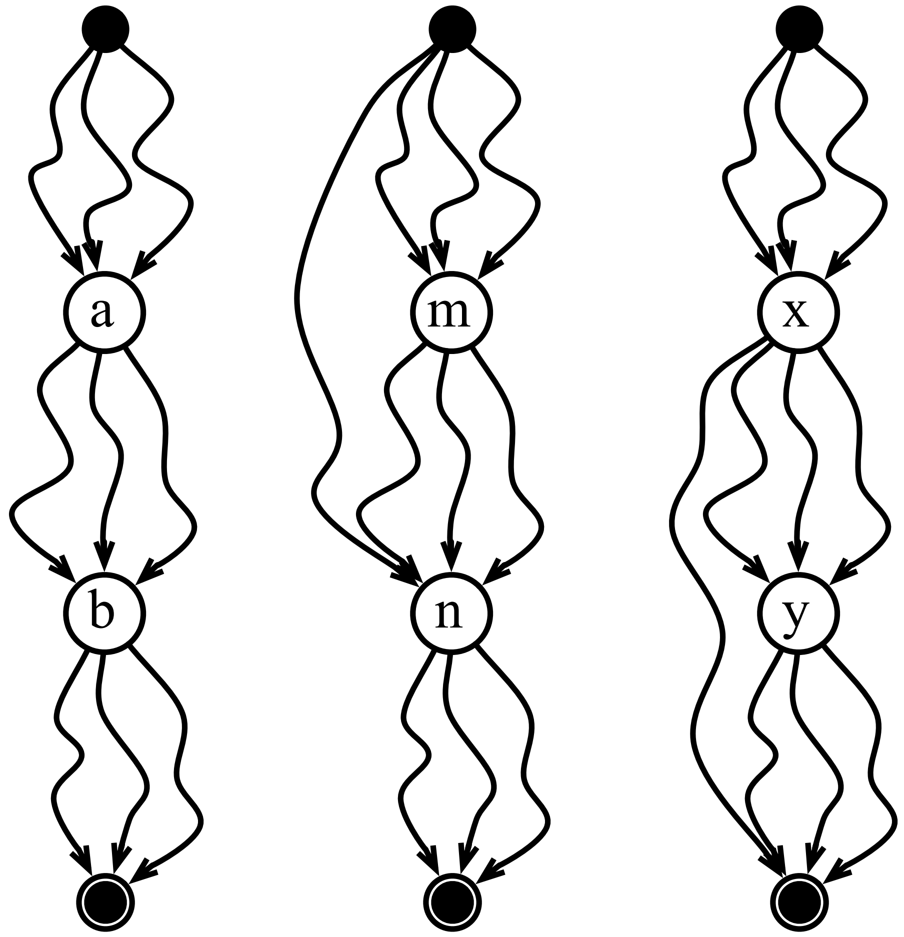
Backward Analyses

- For backward analyses we do everything "backwardly". Instead of the iterated dominance frontier, we consider the iterated **post**-dominance frontier.
  - **Post-Dominance**: a CFG node n post-dominates a node n' if every program path from n' to the exit node of the CFG goes across n. If n ≠ n', then we say that n strictly post-dominates n'
  - **Post-Dominance frontier** (DF): a node n' is in the post-dominance frontier of a node n if n post-dominates a successor of n', but does not strictly post-dominate n'.
  - **Iterated post-dominance frontier** (PDF+): the iterated post-dominance frontier of a node n is the fixed point of the sequence:

$$PDF_1(n) = PDF(n)$$

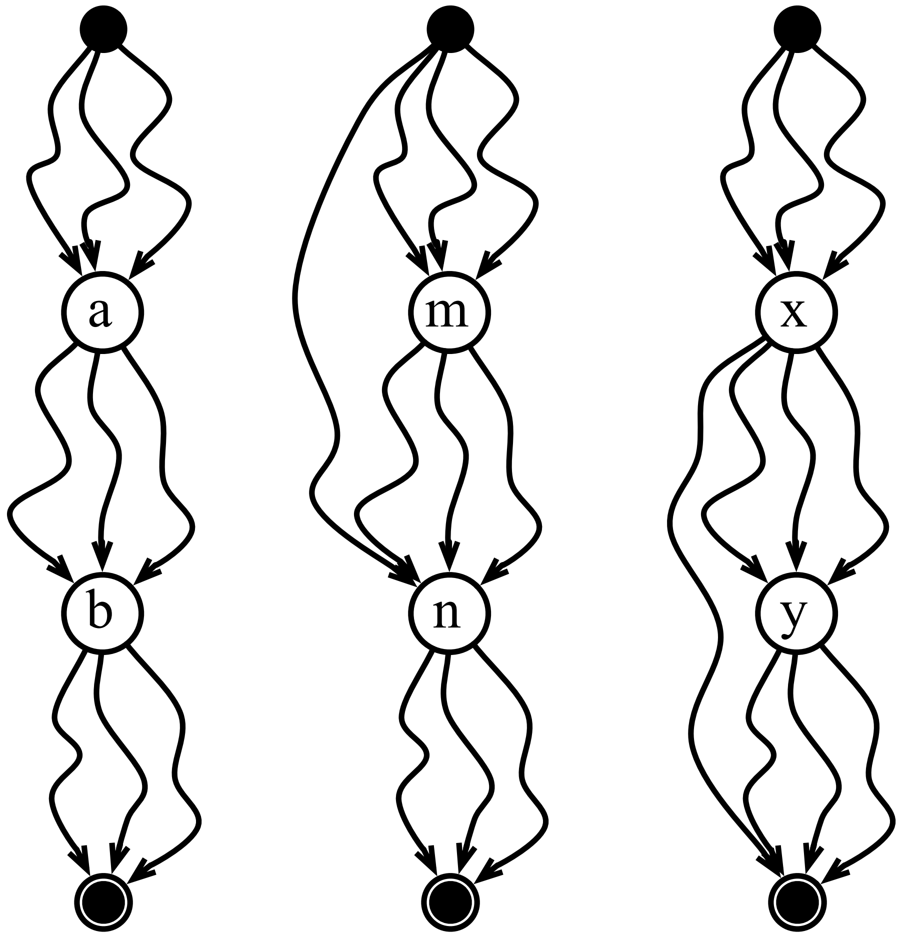
$$PDF_{i+1}(n) = PDF_i(n) \cup \{PDF(z) \mid z \in PDF_i\}$$

Dominators and Post-Dominators



What is the dominance and post-dominance relations between each node in these three graphs?

Dominators and Post-Dominators

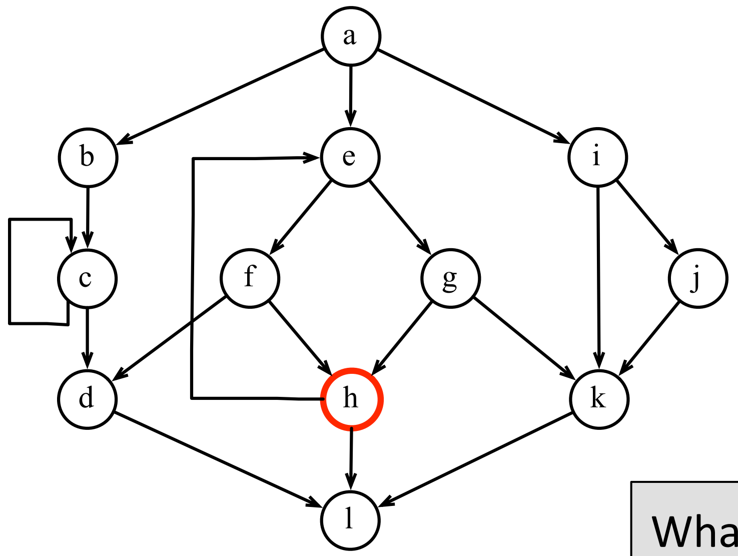


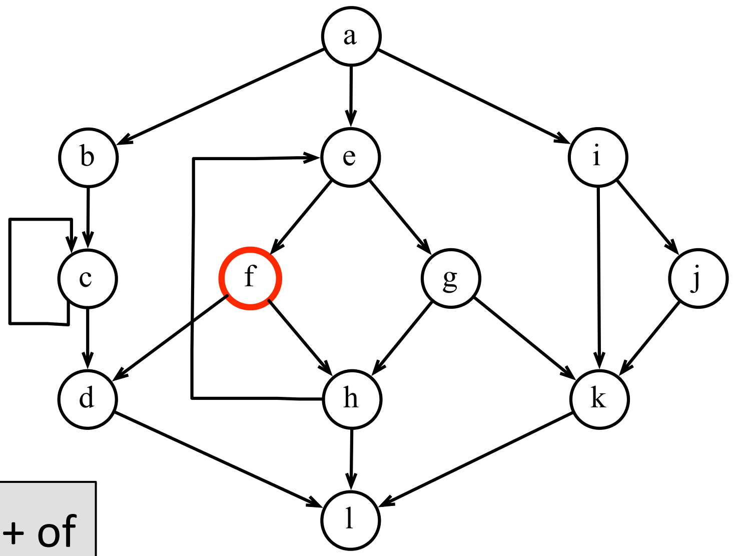
Node 'a' dominates node 'b', for any path from start to 'b' goes across 'a'. Similarly, node 'b' post-dominates node 'a', for any (backwards) path from the end of the CFG to 'a' must go across 'b'.

Node n post-dominates node m, but m does not dominate n.

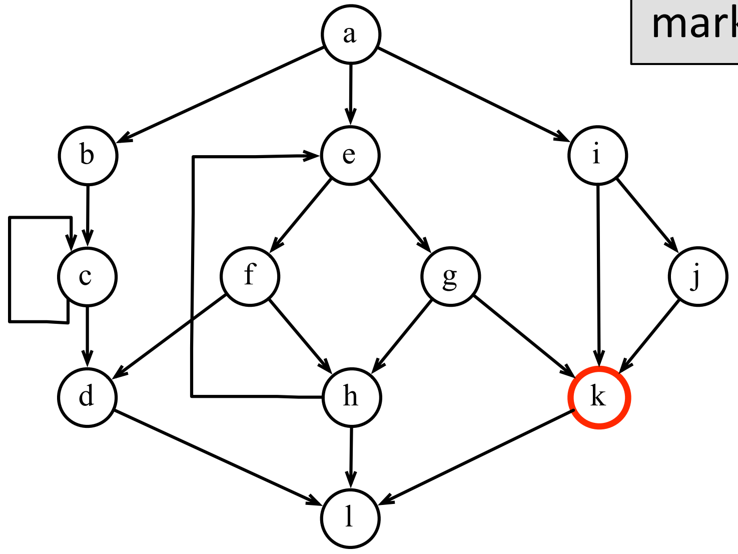
Node x dominates node y, but y does not post-dominate node x.

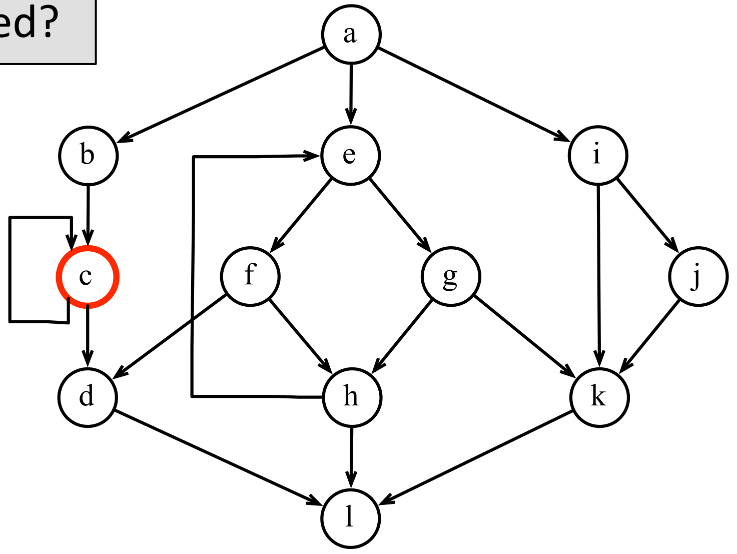
Iterated Post-Dominance Frontier



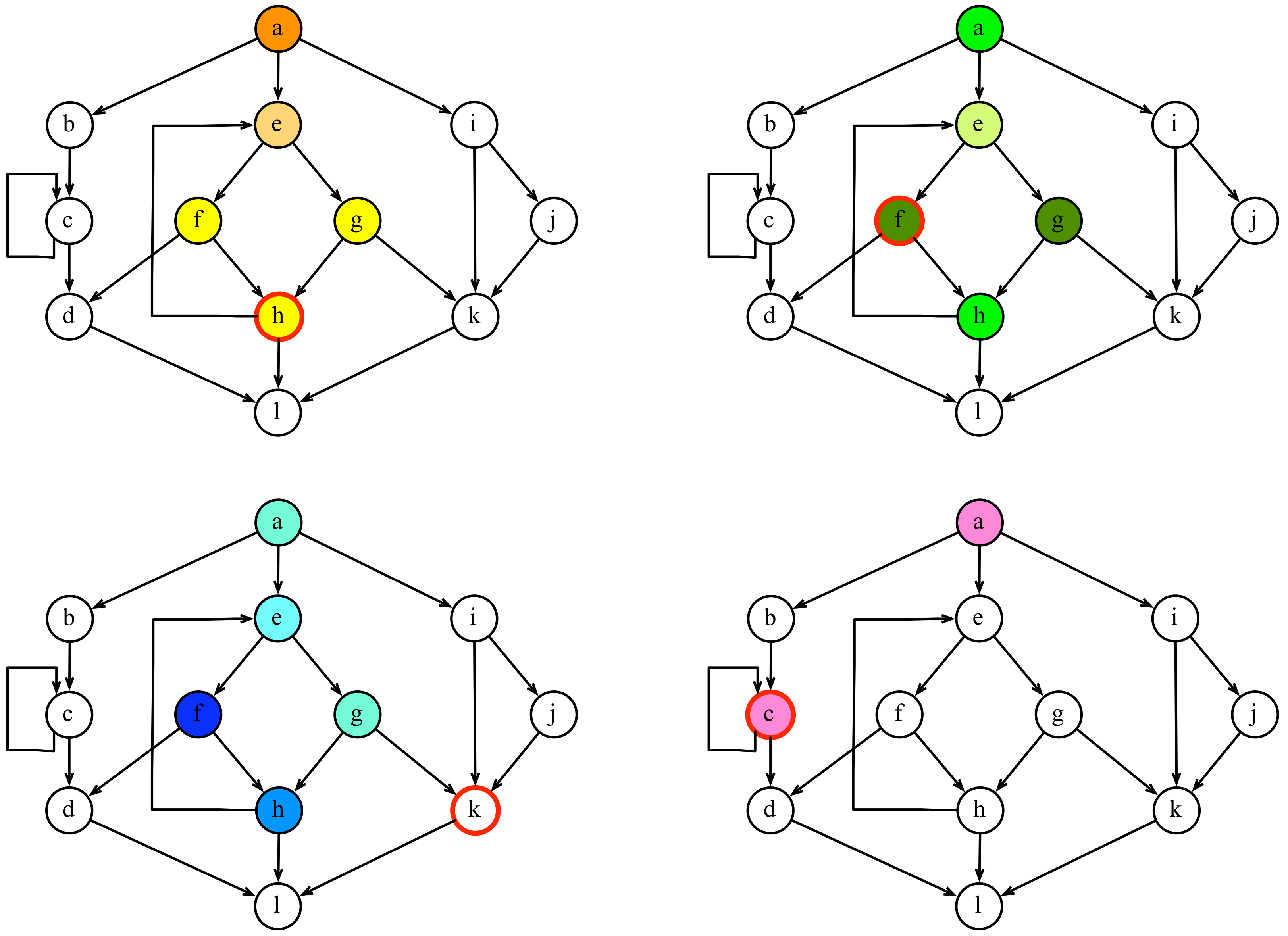


What is PDF+ of each node marked in red?





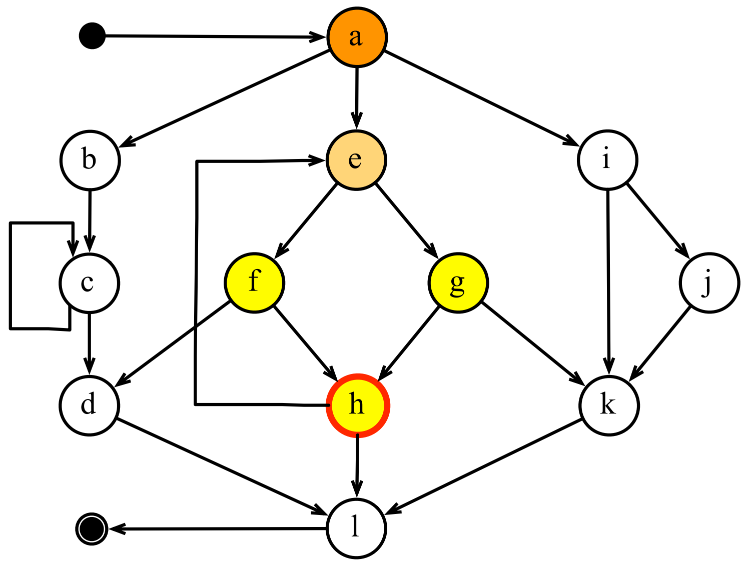
Iterated Post-Dominance Frontier

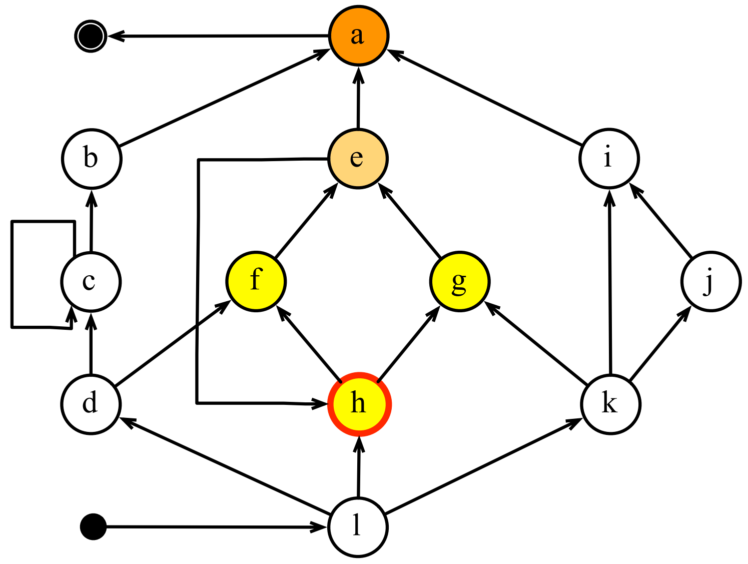


A Cheap Trick

The post-dominance relations are equivalent to the dominance relations if we reverse the edges of the CFG.

As an example, the post-dominance frontier of h, in the CFG on the upper right is the same as the dominance-frontier of h in the CFG in the lower right.

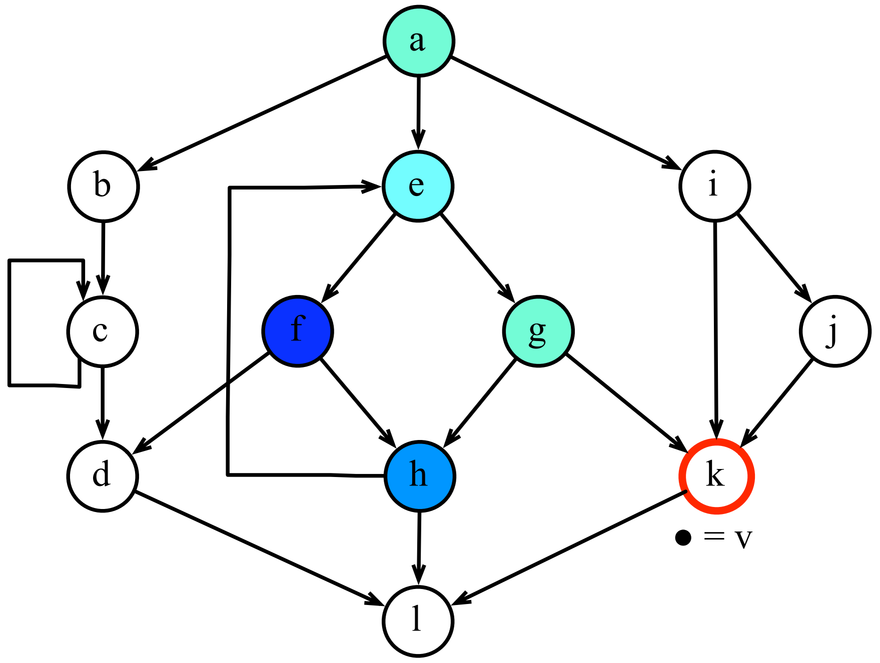




Splitting for Backward Analyses

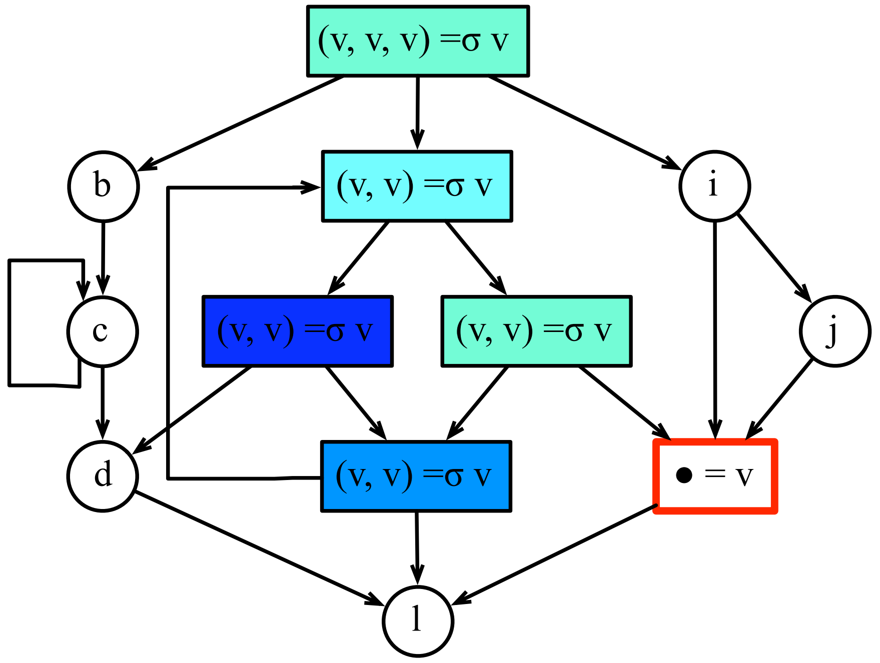
- If we have a definition of variable v at a program point p, then we must split the live range of v at every node that is in the post-dominance frontier of v.
- We do this live range splitting via sigma-functions, which we insert at the nodes in PDF+(v).

How would we split live ranges if we had a definition of variable v at node k in the control flow graph on the right?



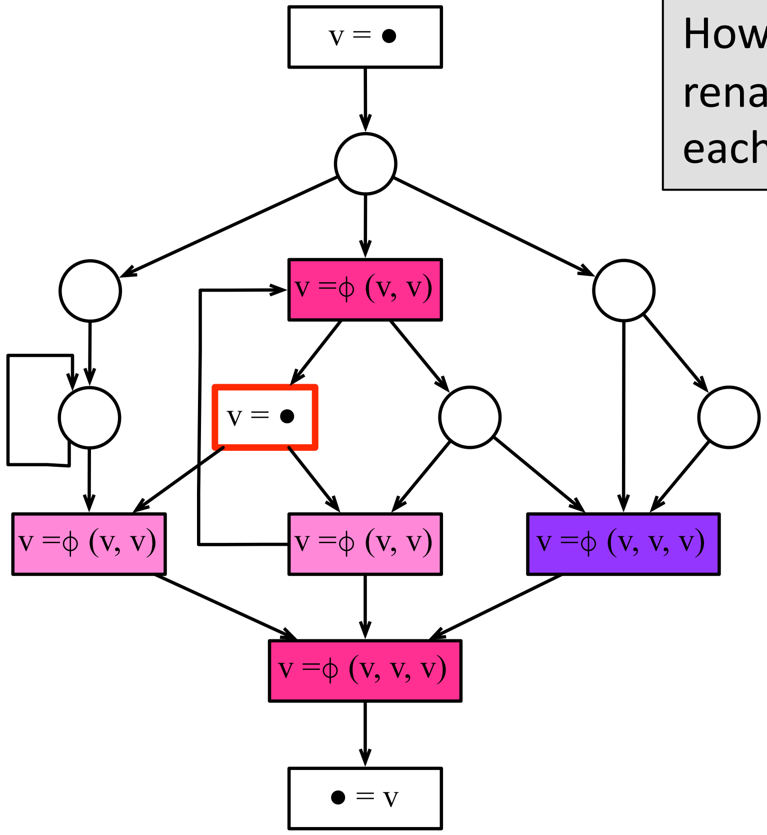
Splitting for Backward Analyses

- If we have a definition of variable v at a program point p, then we must split the live range of v at every node that is in the post-dominance frontier of v.
- We do this live range splitting via sigma-functions, which we insert at the nodes in PDF+(v).
    - If we have a node *n* with *m* successors, then we create a sigma-function at *n*, defining *m* new versions of v.

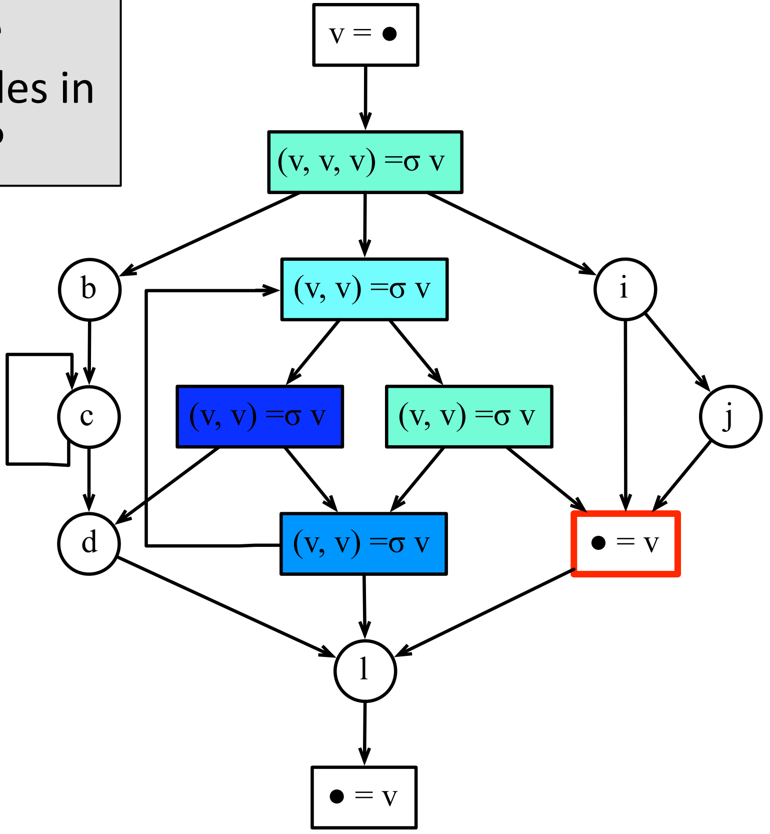


Renaming Variables

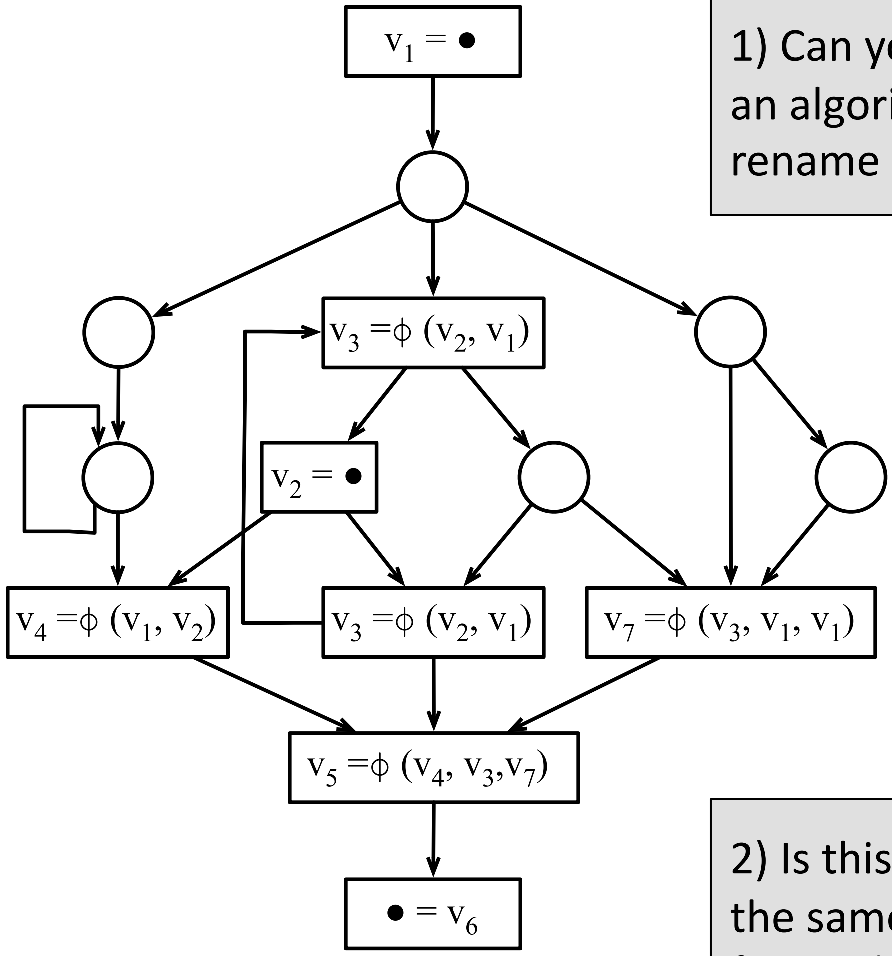
- Once we have split variables, we must rename them, ensuring that each variable is bound to a unique information, which is invariant along its live range.



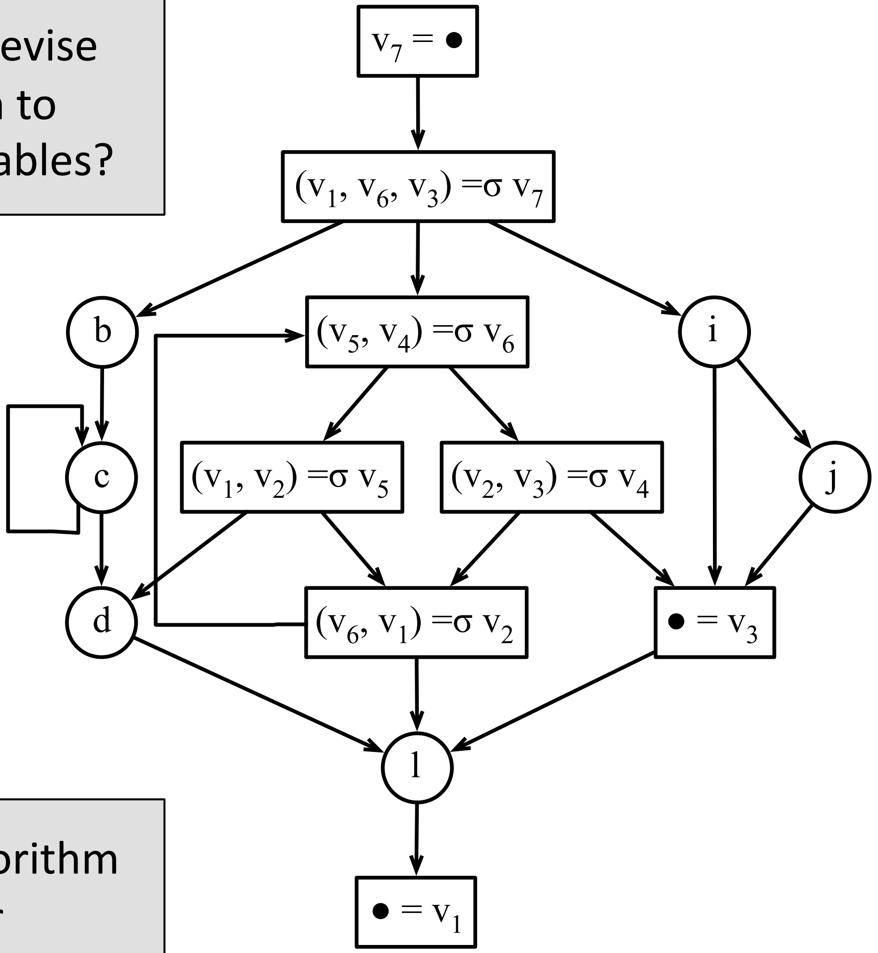
How would we rename variables in each program?



Renaming Variables



1) Can you devise an algorithm to rename variables?


$v_7 = ●$
$(v_1, v_6, v_3) = \sigma\ v_7$
b
$(v_5, v_4) = \sigma\ v_6$
i
c
$(v_1, v_2) = \sigma\ v_5$
$(v_2, v_3) = \sigma\ v_4$
j
d
$(v_6, v_1) = \sigma\ v_2$
$● = v_3$
l
$● = v_1$

2) Is this algorithm the same for forward and backward analyses?

Renaming Variables

- Forward analyses:
  - For each definition site v = • at a program point p, in any order:
    - Find a new name v_i to v;
    - Rename every use of v that is dominated by p to v_i
- Backward analyses:
  - For each use site • = v where information originates, at a program point p, in any order:
    - Find a new name v_i to v;
    - Rename every use of v, and each definition of v, that is post-dominated by p to v_i

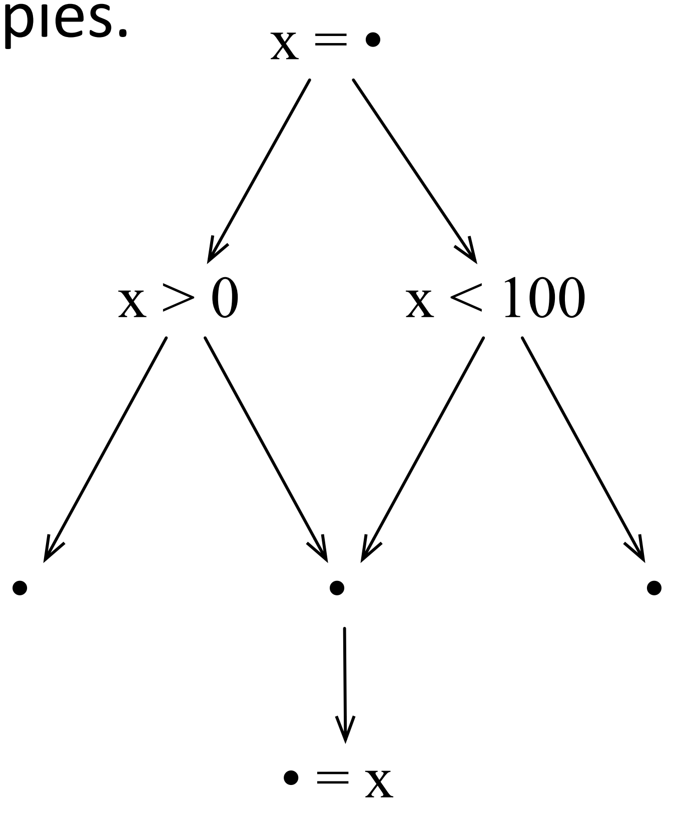
1) Why we do not need to consider definition sites where information originates for the backward analysis?

Pruning Unused Copies

- Our live range splitting algorithm may insert copies that are not used (nor defined) in the program, if we do not consider the pseudo-definitions and the pseudo-uses.
- We can remove some of these copies.
  - This process is called *pruning*.

1) Consider the program on the right. How would we split live ranges for a forward analysis that extracts information from definitions and conditional tests?

2) Can you name an actual analysis that behaves like this?



Pruning Unused Copies

1) Many new versions of variable v are unnecessary. Can you name a few?

2) How could we simplify the program on the right?

3) What is the complexity of this simplification algorithm?

$$\begin{tikzcd}
 & & x = \bullet \arrow[dl] \arrow[dr] & & \\
 & x > 0 \arrow[dl] \arrow[dr] & & x < 100 \arrow[dl] \arrow[dr] & \\
\bullet & & \bullet \arrow[d] & & \bullet \\
 & & \bullet = x & &
\end{tikzcd}$$

$$\begin{tikzcd}
 & & x_0 = \bullet \arrow[dl] \arrow[dr] & & \\
 & \begin{array}{c} x_0 > 0 \\ (x_4, x_1) =\sigma\ x_0 \end{array} \arrow[dl] \arrow[dr] & & \begin{array}{c} x_0 < 100 \\ (x_2, x_5) =\sigma\ x_0 \end{array} \arrow[dl] \arrow[dr] & \\
\bullet & & x_3\ \varphi = (x_1, x_2) \arrow[d] & & \bullet \\
 & & \bullet = x_3 & &
\end{tikzcd}$$

Pruning by Liveness

We can prune by liveness: if a copy of the variable is not alive, i.e., there is no path from this copy to a use of the variable that does not go across a redefinition of it, then we can eliminate this copy.

$$\begin{tikzcd}
 & & x_0 = \bullet \arrow[dl] \arrow[dr] & & \\
 & \begin{array}{c} x_0 > 0 \\ (x_4, x_1) =\sigma\ x_0 \end{array} \arrow[dl] \arrow[dr] & & \begin{array}{c} x_0 < 100 \\ (x_2, x_5) =\sigma\ x_0 \end{array} \arrow[dl] \arrow[dr] & \\
\bullet & & x_3\ \varphi = (x_1, x_2) \arrow[d] & & \bullet \\
 & & \bullet = x_3 & &
\end{tikzcd}$$

$$\begin{tikzcd}
 & & x_0 = \bullet \arrow[dl] \arrow[dr] & & \\
 & \begin{array}{c} x_0 > 0 \\ (\perp, x_1) =\sigma\ x_0 \end{array} \arrow[dl] \arrow[dr] & & \begin{array}{c} x_0 < 100 \\ (x_2, \perp) =\sigma\ x_0 \end{array} \arrow[dl] \arrow[dr] & \\
\bullet & & x_3\ \varphi = (x_1, x_2) \arrow[d] & & \bullet \\
 & & \bullet = x_3 & &
\end{tikzcd}$$

Birectional Analyses

There exist data-flow analyses[♤] that propagate information both forwardly, and backwardly. Consider, for instance, bitwidth analysis: we want to know the size, in bits, of each variable.

1) Why would we be willing to know the size, in bits, of each variable?

2) Which statements produce new information in this analysis?

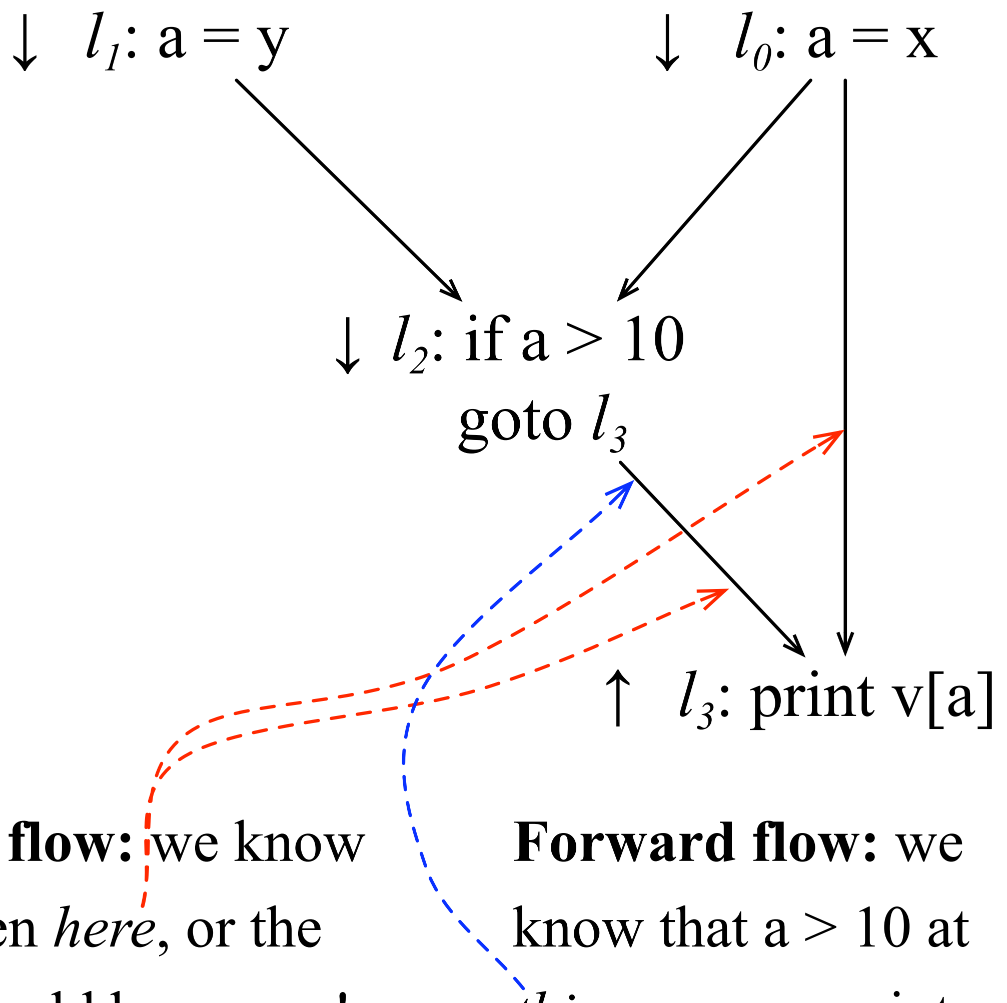
3) How does each of these statements propagate information?

$$\begin{tikzcd}
l_1\text{: a = y} \arrow[dr] & & l_0\text{: a = x} \arrow[dl] \arrow[dd] \\
& l_2\text{: if a > 10 goto } l_3 \arrow[dr] & \\
& & l_3\text{: print v[a]}
\end{tikzcd}$$

[♤]: Bitwidth analysis with application to silicon compilation, PLDI (2000)

Bidirectional Analysis

1) Which information can we infer from the backward constraints?
2) And which information can we learn from the forward constraints?
3) How would be the intermediate program representation that we create for this example?



Backward flow: we know that a < v.len *here*, or the program would be wrong!

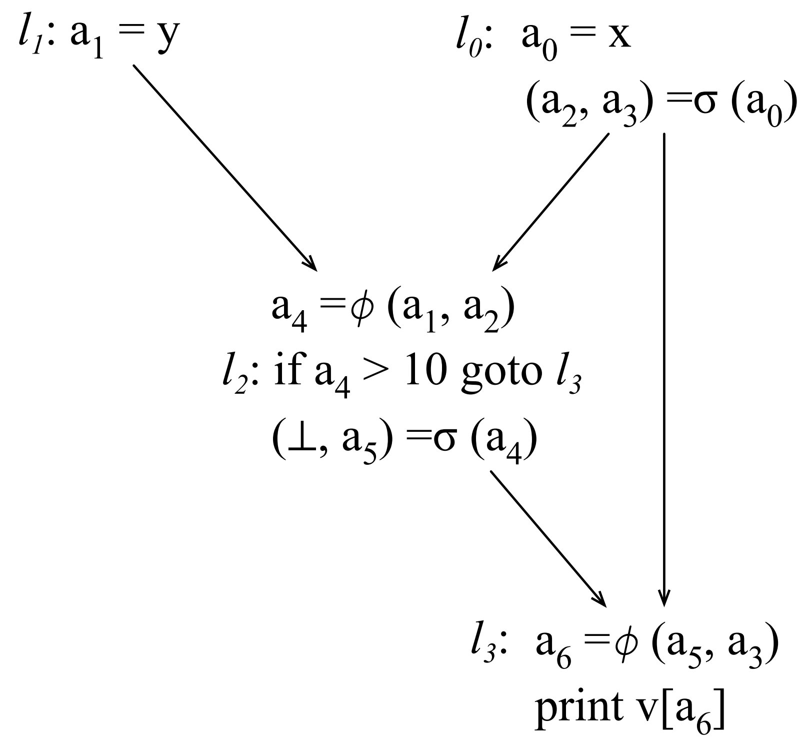
Forward flow: we know that a > 10 at *this* program point.

Bidirectional Analysis

Which constraints can we derive from this new program representation?

$$\begin{tikzcd}
l_1\text{: a = y} \arrow[dr] & & l_0\text{: a = x} \arrow[dl] \arrow[dd] & l_1\text{: } a_1 = y \arrow[dr] & & \begin{array}{l} l_0\text{: } a_0 = x \\ (a_2, a_3) = \sigma\,(a_0) \end{array} \arrow[dl] \arrow[dd] \\
 & \begin{array}{c} l_2\text{: if a > 10} \\ \text{goto } l_3 \end{array} \arrow[dr] \arrow[rrr, thick, bend left] & & & \begin{array}{c} a_4 = \phi\,(a_1, a_2) \\ l_2\text{: if } a_4 > 10 \text{ goto } l_3 \\ (\perp, a_5) = \sigma\,(a_4) \end{array} \arrow[dr] & \\
 & & l_3\text{: print v[a]} & & & \begin{array}{l} l_3\text{: } a_6 = \phi\,(a_5, a_3) \\ \text{print } v[a_6] \end{array}
\end{tikzcd}$$

Bidirectional Analysis



l_1: $a_1 = y$

l_0: $a_0 = x$
$(a_2, a_3) = \sigma(a_0)$

$a_4 = \phi(a_1, a_2)$
l_2: if $a_4 > 10$ goto l_3
$(\perp, a_5) = \sigma(a_4)$

l_3: $a_6 = \phi(a_5, a_3)$
print $v[a_6]$

Forward flow:

$[a_0] \geq [x]$

$[a_1] \geq [y]$

$[a_2] \geq [a_0]$

$[a_3] \geq [a_0]$

$[a_4] \geq \mathrm{MAX}([a_1], [a_2])$

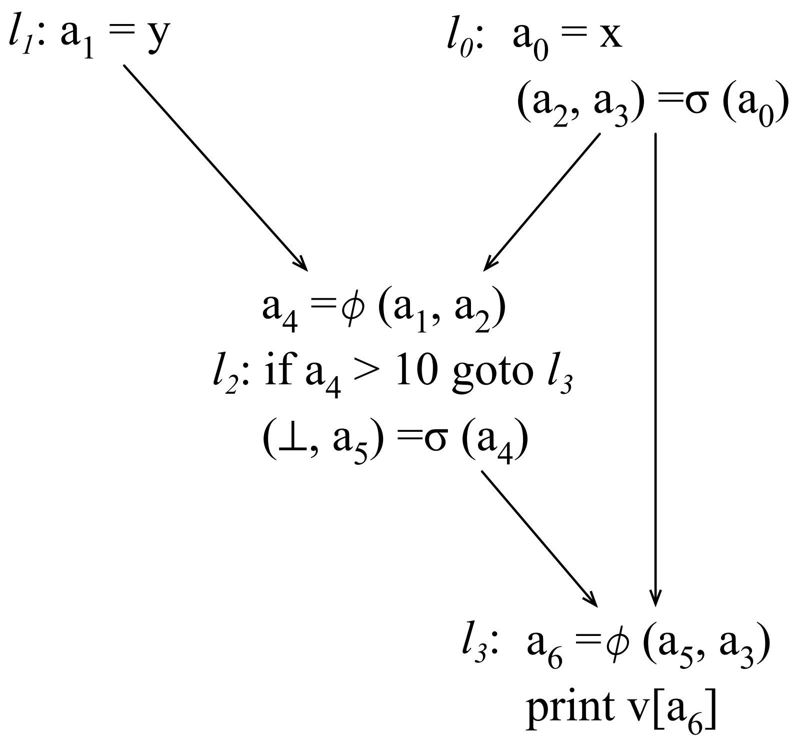
$[a_5] \geq \mathrm{MAX}([a_4], 10)$

$[a_6] \geq \mathrm{MAX}([a_3], [a_5])$

1) Can you explain each one of these constraints?

2) And how are the backward constraints like?

Bidirectional Analysis



l_1: $a_1 = y$
l_0: $a_0 = x$
$(a_2, a_3) = \sigma\ (a_0)$
$a_4 = \phi\ (a_1, a_2)$
l_2: if $a_4 > 10$ goto l_3
$(\perp, a_5) = \sigma\ (a_4)$
l_3: $a_6 = \phi\ (a_5, a_3)$
print $v[a_6]$

Forward flow:

$[a_0] \geq [x]$

$[a_1] \geq [y]$

$[a_2] \geq [a_0]$

$[a_3] \geq [a_0]$

$[a_4] \geq \text{MAX}([a_1], [a_2])$

$[a_5] \geq \text{MAX}([a_4], 10)$

$[a_6] \geq \text{MAX}([a_3], [a_5])$

Backward flow:

$[a_6] < [v.len]$

$[a_5] \leq [a_6]$

$[a_3] \leq [a_6]$

$[a_4] \leq [a_5]$

$[a_1] \leq [a_4]$

$[a_2] \leq [a_4]$

$[a_0] \leq \text{MIN}([a_2], [a_3])$

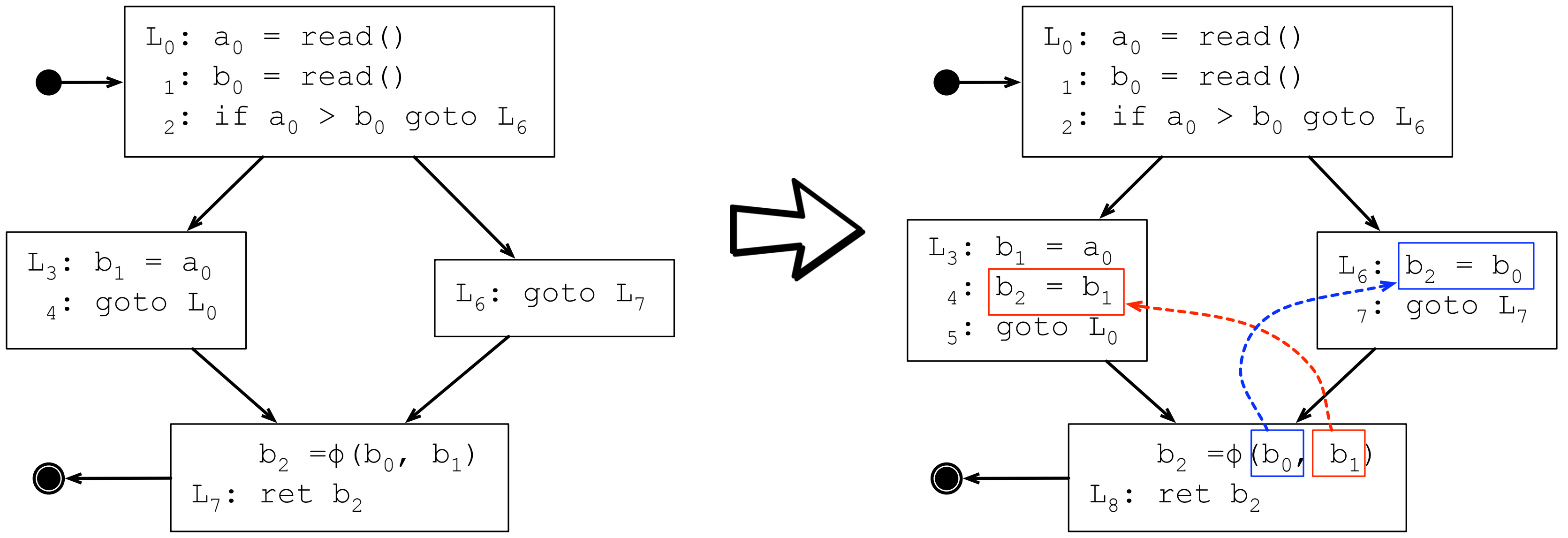
$[x] \leq [a_0]$

$[y] \leq [a_1]$

Can you provide a rationale for the backward constraints?

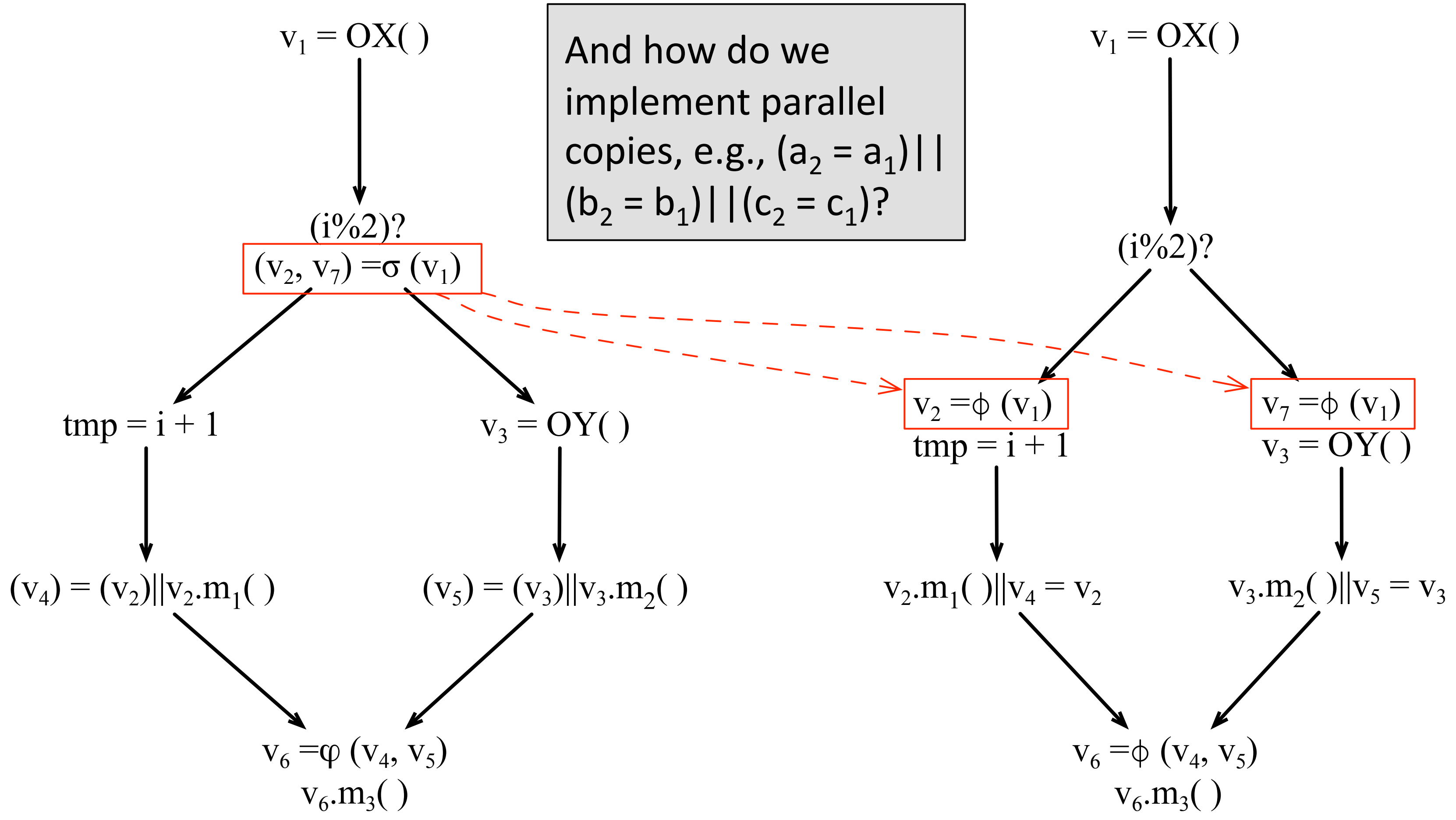
Implementing Parallel Copies

- Actual instruction sets, in general, will not provide parallel copies.
  - We must implement them using more conventional instructions.
- We already know how to implement phi-functions: we can replace these instructions with copies.
  - But, how do we implement sigma-functions, and parallel copies placed at interior nodes?



Dealing with Sigma-Functions

- We can represent sigma-functions as *single-arity* phi-functions, at the beginning of successor basic blocks.



And how do we implement parallel copies, e.g., $(a_2 = a_1) || (b_2 = b_1) || (c_2 = c_1)$?

Implementing Parallel Copies

- Parallel copies can be implemented as sequences of move instructions:

$$(a_2 = a_1) || (b_2 = b_1) || (c_2 = c_1)$$

$$\Downarrow$$

$a_2 = a_1;$
$b_2 = b_1;$
$c_2 = c_1$

This same strategy, of implementing parallel copies as sequences of move instructions, can also be used to implement the parallel copies embedded in phi-functions and in sigma-functions.

Phi-Assignment:

$$\begin{bmatrix} v_1 \\ \vdots \\ v_m \end{bmatrix} =\phi \begin{array}{c} l_1, \dots, l_n \\ \begin{bmatrix} v_{11}, \dots, v_{1n} \\ \vdots \quad\quad \vdots \\ v_{m1}, \dots, v_{mn} \end{bmatrix} \end{array}$$

Sigma-Assignment:

$$\begin{array}{c} l_1, \dots, l_n \\ \begin{bmatrix} v_{11}, \dots, v_{1n} \\ \vdots \quad\quad \vdots \\ v_{m1}, \dots, v_{mn} \end{bmatrix} \end{array} =\sigma \begin{bmatrix} v_1 \\ \vdots \\ v_m \end{bmatrix}$$

Parallel-Assignment:

$$\begin{bmatrix} v_1 \\ \vdots \\ v_m \end{bmatrix} = \begin{bmatrix} u_1 \\ \vdots \\ u_m \end{bmatrix}$$

A Bit of History

- An well-known attempt to implement a framework for Sparse Analyses is due to Choi *et al*.
- There have been many program representations for Sparse Analyses, such as Ananian's SSI and Bodik's e-SSA.
- These representations still use the core ideas introduced by Cytron *et al*. to build the static single assignment form.

- Cytron, R., Ferrante, J., Rosen, B., Wegman, M., and Zadeck, K., "An Efficient Method of Computing Static Single Assignment Form", POPL, p 25-35 (1989)
- Choi, J., Cytron, R., and Ferrante, J., "Automatic Construction of Sparse Data Flow Evaluation Graphs", POPL, p 55-66 (1991)
- Ananian, S., "The Static Single Information Form", MSc Dissertation, Massachusetts Institute of Technology (1999)
- Bodik., R., Gupta, R., and Sarkar, V., "ABCD: eliminating array bounds checks on demand", PLDI, p 321-333 (2000)