Bidirectional flow analysis for a concurrent reversible programming language

Shoji Yuen (joint work with Shun’ya Oguchi)
Graduate School of Informatics, Nagoya University

Contents

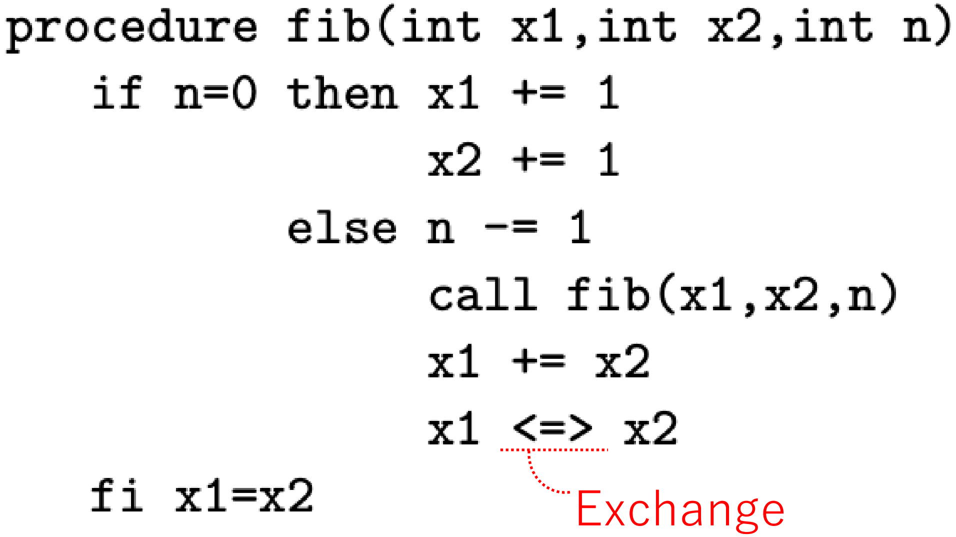
- Background & Motivation
- CRIL: Concurrent Reversible Intermediate Language
- The controlled semantics of CRIL
- Reversibility Properties
- Bidirectional Data Flow Analysis in CRIL
- Constant Propagation in CRIL
- Concluding Remarks

Reversibility in CS What/Why

- Energy/Computation Relation
  Landauer’s principle
- Fundamental computation paradigm
  Reversible Turing Machine
- Reversible PL/Software Analysis
  JANUS, R-FUN
- Reversibility in Communication and Concurrency
  RCCS/CCSK, Hoey’s Language, CRIL
- Low-energy circuits
- Quantum computation/circuits

Reversible Programming Language

- Languages for computation where the control flows both forward and backward.
- Janus [Lutz+, 80] is a reversible programming language.



```
procedure fib(int x1,int x2,int n)
   if n=0 then x1 += 1
               x2 += 1
          else n -= 1
               call fib(x1,x2,n)
               x1 += x2
               x1 <=> x2
   fi x1=x2
```

Exchange

A Janus program [Yokoyama, 10]

```
x1 += x2

x1 <=> x2
```

```
x1 -= x2

x1 <=> x2
```

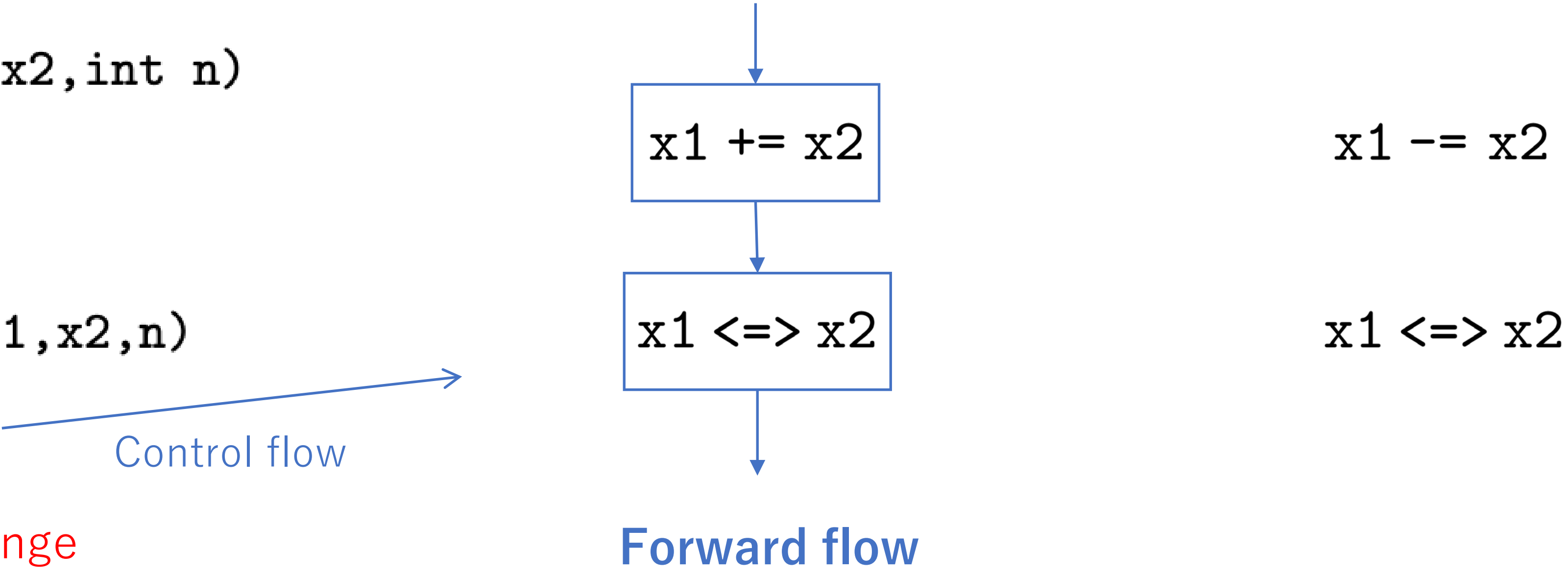
Reversible Programming Language

- Languages for computation where the control flows both forward and backward.
- Janus [Lutz+, 80] is a reversible programming language.

```
procedure fib(int x1,int x2,int n)
   if n=0 then x1 += 1
               x2 += 1
          else n -= 1
               call fib(x1,x2,n)
               x1 += x2
               x1 <=> x2
   fi x1=x2
```

Exchange

A Janus program [Yokoyama, 10]



Forward flow

Reversible Programming Language

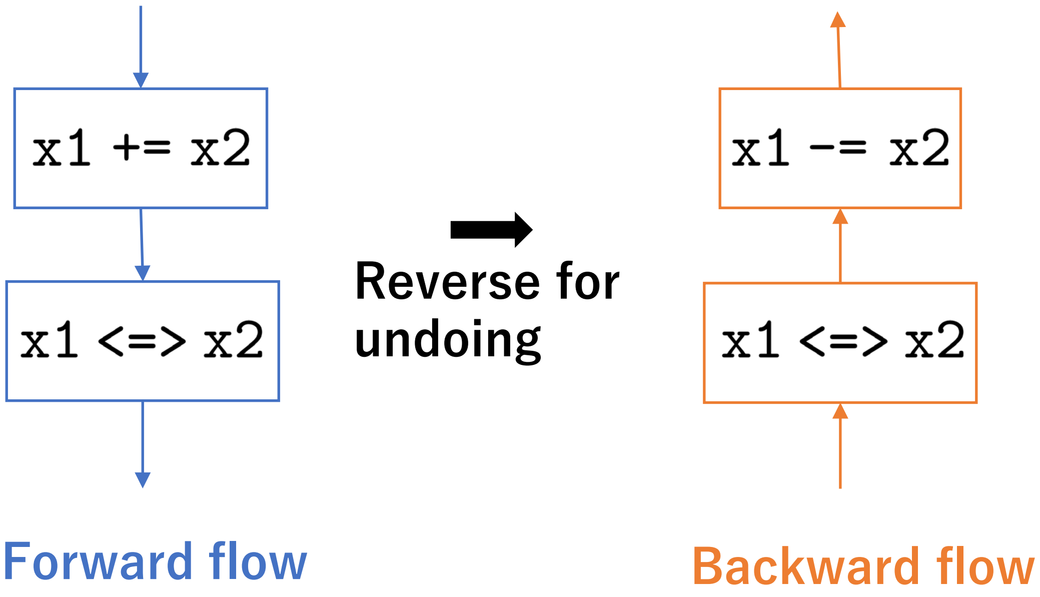
- Languages for computation where the control flows both forward and backward.
- Janus [Lutz+, 80] is a reversible programming language.

```
procedure fib(int x1,int x2,int n)
   if n=0 then x1 += 1
               x2 += 1
          else n -= 1
               call fib(x1,x2,n)
               x1 += x2
               x1 <=> x2
   fi x1=x2
```

Exchange

Control flow

A Janus program [Yokoyama, 10]

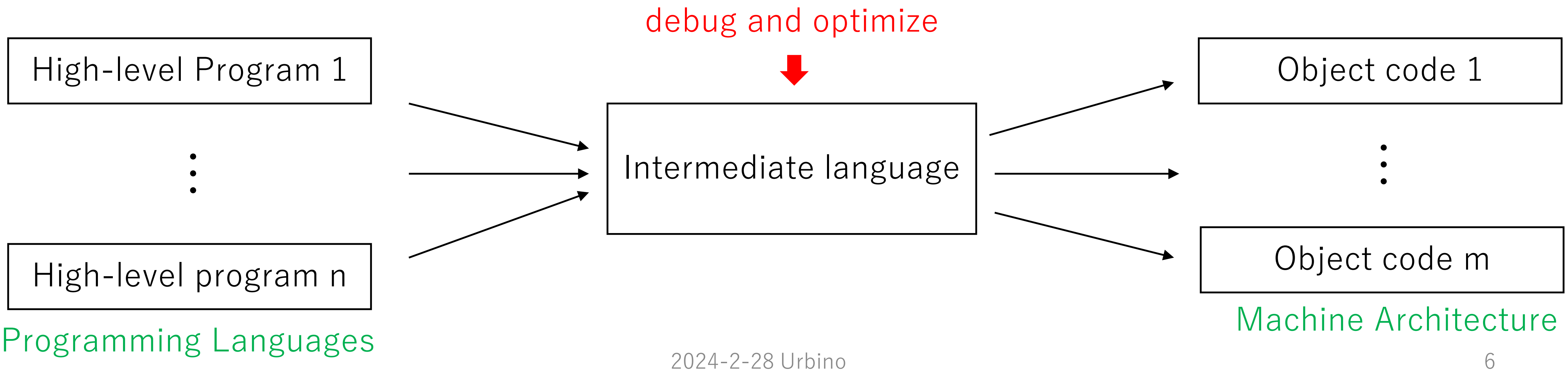


Reversiblity in program executions

- **Reversibility keeps all information of the execution** at any points.
- Reversibility enables behaviour analysis, such as **debugging**, **without replay** by **undoing execution**.
- Reversibility is useful for the analysis of programs where **replay is difficult**, such as **concurrent programs**.

Intermediate Language

- Intermediate languages, such as LLVM, are used to translate high-level language programs into machine-oriented forms, such as three-address codes.
- Intermediate languages are used for debugging and optimization.



RIL : Reversible Intermediate Language [Mogensen 16]

```
begin $main$
$x$ += $0$
$\rightarrow entry$
$entry; loop \leftarrow x$ == $0$
$x$ += $1$
$x < 10 \rightarrow loop; exit$
$exit \leftarrow$
$x$ += $3$
end $main$
```

A RIL program[Mogensen 16]

- RIL is a reversible intermediate language to implement a heap manager for garbage collection and analyse the memoy usage for functional reversible language [Mogensen 16].

RIL : Reversible Intermediate Language [Mogensen 16]

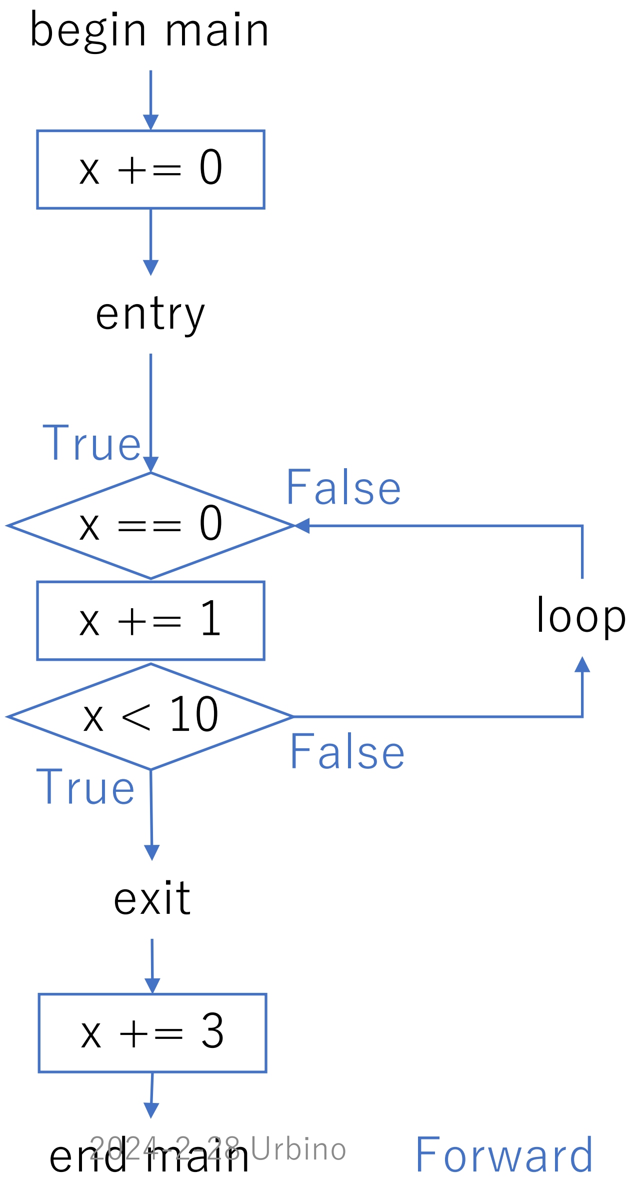
b1	$\mathbf{begin}\ \mathit{main}$	Entry point
	$x\ \mathbf{+=}\ 0$	Instruction
	$\rightarrow \mathit{entry}$	Exit point
b2	$\mathit{entry}; \mathit{loop} \leftarrow x\ \mathbf{==}\ 0$	Conditional entry point
	$x\ \mathbf{+=}\ 1$	Instruction
	$x < 10 \rightarrow \mathit{loop}; \mathit{exit}$	Conditonal exit point
b3	$\mathit{exit} \leftarrow$	Entry point
	$x\ \mathbf{+=}\ 3$	Instruction
	$\mathbf{end}\ \mathit{main}$	Exit point

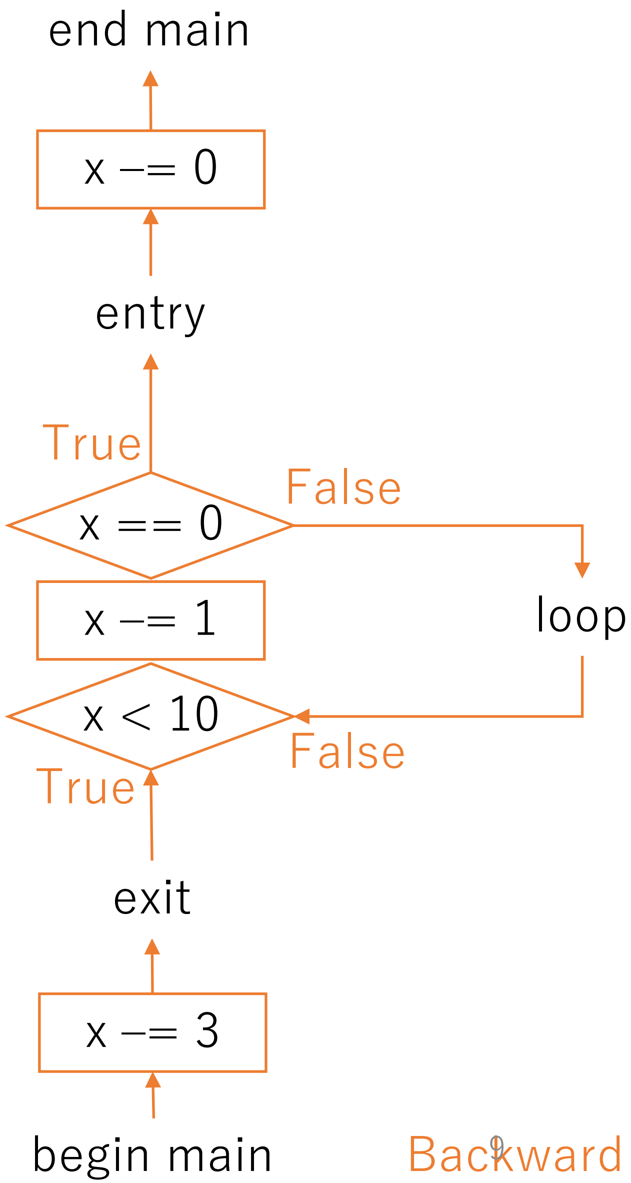
A RIL program[Mogensen 16]

RIL : Reversible Intermediate Language [Mogensen 16]

$$\mathbf{begin}\ \mathit{main}$$
$$x \mathrel{\mathbf{+=}} 0$$
$$\rightarrow \mathit{entry}$$
$$\mathit{entry};\ \mathit{loop} \leftarrow x \mathrel{\mathbf{==}} 0$$
$$x \mathrel{\mathbf{+=}} 1$$
$$x < 10 \rightarrow \mathit{loop};\ \mathit{exit}$$
$$\mathit{exit} \leftarrow$$
$$x \mathrel{\mathbf{+=}} 3$$
$$\mathbf{end}\ \mathit{main}$$

A RIL program[Mogensen 16]





Contents

- Background & Motivation
- CRIL: Concurrent Reversible Intermediate Language
- The controlled semantics of CRIL
- Reversibility Properties
- Bidirectional Data Flow Analysis in CRIL
- Constant Propagation in CRIL
- Concluding Remarks

CRIL : Concurrent RIL

- We propose CRIL, a concurrent reversible intermediate language by extending RIL.
- CRIL serves as a tool for **undoing concurrent programs**.
- CRIL enjoys the basic properties of reversibility.

$$\boxed{\text{RIL}} + \boxed{\text{Concurrency}} \rightarrow \boxed{\text{CRIL}}$$

Syntax of CRIL

$$
\begin{array}{rcl}
\mathit{Pg} & ::= & b^* \\
b & ::= & \mathit{instb} \mid \mathit{callb} \\
\mathit{instb} & ::= & \mathit{entry}\ \mathit{inst}\ \mathit{exit} \\
\mathit{entry} & ::= & l\ \texttt{<-} \mid l\,\texttt{;}\,l\ \texttt{<-}\ e \mid \texttt{begin}\ l \\
\mathit{exit} & ::= & \texttt{->}\ l \mid e\ \texttt{->}\ l\,\texttt{;}\,l \mid \texttt{end}\ l \\
\mathit{inst} & ::= & \mathit{left} \oplus\!\texttt{=}\ e \mid \mathit{left}\ \texttt{<->}\ \mathit{left} \\
 & & \mid \underline{\texttt{V}\ x \mid \texttt{P}\ x} \mid \texttt{assert}\ e \mid \texttt{skip} \\
\mathit{callb} & ::= & \underline{l\ \texttt{<-}\ \texttt{call}\ l(,\ l)^*\ \texttt{->}\ l} \\
e & ::= & \mathit{right} \odot \mathit{right} \mid \texttt{!}\,\mathit{right} \\
\mathit{left} & ::= & x \mid \texttt{M[}x\texttt{]} \\
\mathit{right} & ::= & k \mid \mathit{left} \\
\oplus & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{\^{}} \\
\odot & ::= & \oplus \mid \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \\
 & & \mid \texttt{>=} \mid \texttt{\&\&} \mid \texttt{||}
\end{array}
$$

- CRIL allows multiple blocks to run concurrently and synchronization primitive.

- A program is **well-defined** if labels specify deterministic bi-directional control flow.
- Well-definedness is similar to RIL.

Operational Smantics

Forward direction

$$(Pg,\rho,\sigma,Pr) \xrightleftharpoons[\text{prog}]{p,Rd,Wt} (Pg,\rho',\sigma',Pr')$$

Backward direction

Operational Smantics

Forward direction

$$(Pg,\boxed{\rho},\boxed{\sigma},\boxed{Pr}) \xrightleftharpoons[\text{prog}]{p,Rd,Wt} (Pg,\rho',\sigma',Pr')$$

Backward direction

- Pg is a program (never changed).
- ρ maps a variable to its value.
- σ maps a heap memory address to its value.
- Pr maps a process id to a process configuration.

Operational Smantics

Forward direction

$$(Pg,\boxed{\rho},\boxed{\sigma},\boxed{Pr}) \xrightleftharpoons[\mathbf{prog}]{p,Rd,Wt} (Pg,\rho',\sigma',Pr')$$

Backward direction

- Pg is a program (never changed).
- ρ maps a variable to its value.
- σ maps a heap memory address to its value.
- Pr maps a process id to a process configuration.

A sequence of numbers.

$p \cdot i$ is the i-th subprocess of $p \in \mathbb{N}^*$.

$(l, stage)$, where l is the current label of the basic block and $stage$ is either **begin**, **run**, or **end**.

Transition Label

→ Forward direction

$$(Pg,\rho,\sigma,Pr) \xrightleftharpoons[\text{prog}]{\boxed{p},\boxed{Rd},\boxed{Wt}} (Pg,\rho',\sigma',Pr')$$

Backward direction ←

- p is the process id.
- Rd is the set of variables read.
- Wt is the set of variables updated.

Operational Semantics for Processes

$$\frac{\mathsf{isleaf}(Pr_{act}, p) \quad b \in Pg \qquad \mathsf{entry}(b) \vdash (\rho, \sigma, l, stage) \quad \mathsf{inst}(b) \rhd (\rho, \sigma) \rightharpoonup (\rho', \sigma') \quad \mathsf{exit}(b) \dashv (\rho', \sigma', l', stage')}{(Pg, \rho, \sigma, Pr[p \mapsto (l, stage)]) \xrightleftharpoons[\text{prog}]{p,\mathsf{read}(b),\mathsf{write}(b)} (Pg, \rho', \sigma', Pr[p \mapsto (l', stage')])}\ \textbf{Inst}$$

$$\frac{\mathsf{isleaf}(Pr_{act}, p) \qquad (l' \ \texttt{<-},\ \texttt{call}\ l_1\,,\cdots,l_n,\ \texttt{->}\ l'') \in Pg}{(Pg, \rho, \sigma, Pr[p \mapsto (l', \mathtt{run})]) \xrightleftharpoons[\text{prog}]{p,\varnothing,\varnothing} (Pg, \rho, \sigma, Pr[p \mapsto (l'', \mathtt{run}), \underbrace{p \cdot 1 \mapsto (l_1, \mathtt{begin}), \cdots, p \cdot n \mapsto (l_n, \mathtt{begin})}_{n \text{ subprocesses}}])}\ \textbf{CallFork}$$

$$\frac{\mathsf{isleaf}(Pr_{act}, p) \qquad (l' \ \texttt{<-},\ \texttt{call}\ l_1\,,\cdots,l_n,\ \texttt{->}\ l'') \in Pg}{(Pg, \rho, \sigma, Pr[p \mapsto (l'', \mathtt{run}), \underbrace{p \cdot 1 \mapsto (l_1, \mathtt{end}), \cdots, p \cdot n \mapsto (l_n, \mathtt{end})}_{n \text{ subprocesses}}]) \xrightleftharpoons[\text{prog}]{p,\varnothing,\varnothing} (Pg, \rho, \sigma, Pr[p \mapsto (l'', \mathtt{run})])}\ \textbf{CallMerge}$$

Synchronization

V x **waits until** $x = 0$ and sets $x = 1$.

Symmetric

P x **waits until** $x = 1$ and sets $x = 0$.

$$\frac{}{\texttt{V}\ x \rhd (\rho[x \mapsto 0], \sigma) \rightharpoonup (\rho[x \mapsto 1], \sigma)}\ \textbf{V-op}$$

$$\frac{}{\texttt{P}\ x \rhd (\rho[x \mapsto 1], \sigma) \rightharpoonup (\rho[x \mapsto 0], \sigma)}\ \textbf{P-op}$$

Expressions:

$$\frac{k \text{ is a constant}}{(\rho,\sigma) \rhd k \rightsquigarrow k}\ \textbf{Con} \qquad \frac{}{(\rho[x \mapsto m], \sigma) \rhd x \rightsquigarrow m}\ \textbf{Var} \qquad \frac{}{(\rho[x \mapsto m_1], \sigma[m_1 \mapsto m_2]) \rhd M[x] \rightsquigarrow m_2}\ \textbf{Mem}$$

$$\frac{(\rho,\sigma) \rhd \mathit{right}_1 \rightsquigarrow m_1 \qquad (\rho,\sigma) \rhd \mathit{right}_2 \rightsquigarrow m_2 \qquad m_3 = m_1 \odot m_2}{(\rho,\sigma) \rhd \mathit{right}_1 \odot \mathit{right}_2 \rightsquigarrow m_3}\ \textbf{Exp1}$$

$$\frac{(\rho,\sigma) \rhd \mathit{right} \rightsquigarrow 0}{(\rho,\sigma) \rhd\ !\mathit{right} \rightsquigarrow 1}\ \textbf{Exp2} \qquad \frac{(\rho,\sigma) \rhd \mathit{right} \rightsquigarrow m \qquad m \neq 0}{(\rho,\sigma) \rhd\ !\mathit{right} \rightsquigarrow 0}\ \textbf{Exp3}$$

Instructions:

$$\frac{(\rho,\sigma) \rhd e \rightsquigarrow m_3 \qquad m_2 = m_1 \oplus m_3}{x \oplus\!= e \rhd (\rho[x \mapsto m_1], \sigma) \rightharpoonup (\rho[x \mapsto m_2], \sigma)}\ \textbf{AssVar}$$

$$\frac{(\rho,\sigma) \rhd e \rightsquigarrow m_3 \qquad m_2 = m_1 \oplus m_3}{\texttt{M[}x\texttt{]} \oplus\!= e \rhd (\rho[x \mapsto m_4], \sigma[m_4 \mapsto m_1]) \rightharpoonup (\rho[x \mapsto m_4], \sigma[m_4 \mapsto m_2])}\ \textbf{AssArr}$$

$$\frac{}{x \texttt{ <-> } y \rhd (\rho[x, y \mapsto m_1, m_2], \sigma) \rightharpoonup (\rho[x, y \mapsto m_2, m_1], \sigma)}\ \textbf{SwapVarVar}$$

$$\frac{}{x \texttt{ <-> M[}y\texttt{]} \rhd (\rho[x, y \mapsto m_1, m_3], \sigma[m_3 \mapsto m_2]) \rightharpoonup (\rho[x, y \mapsto m_2, m_3], \sigma[m_3 \mapsto m_1])}\ \textbf{SwapVarArr}$$

$$\frac{}{\mapsto m_1, m_3], \sigma[m_3 \mapsto m_2]) \rightharpoonup (\rho[x, y \mapsto m_2, m_3], \sigma[m_3 \mapsto m_1])}\ \textbf{SwapArrVar}$$

$$\frac{}{m_4], \sigma[m_3, m_4 \mapsto m_1, m_2]) \rightharpoonup (\rho[x, y \mapsto m_3, m_4], \sigma[m_3, m_4 \mapsto m_2, m_1])}\ \textbf{SwapArrArr}$$

$$\frac{}{\rightharpoonup (\rho[x \mapsto 1], \sigma)}\ \textbf{V-op} \qquad \frac{}{\texttt{P}\ x \rhd (\rho[x \mapsto 1], \sigma) \rightharpoonup (\rho[x \mapsto 0], \sigma)}\ \textbf{P-op}$$

$$\frac{}{) \rightharpoonup (\rho, \sigma)}\ \textbf{Skip} \qquad \frac{(\rho,\sigma) \rhd e \rightsquigarrow m \qquad m \neq 0}{\texttt{assert}\ e \rhd (\rho, \sigma) \rightharpoonup (\rho, \sigma)}\ \textbf{Assert}$$

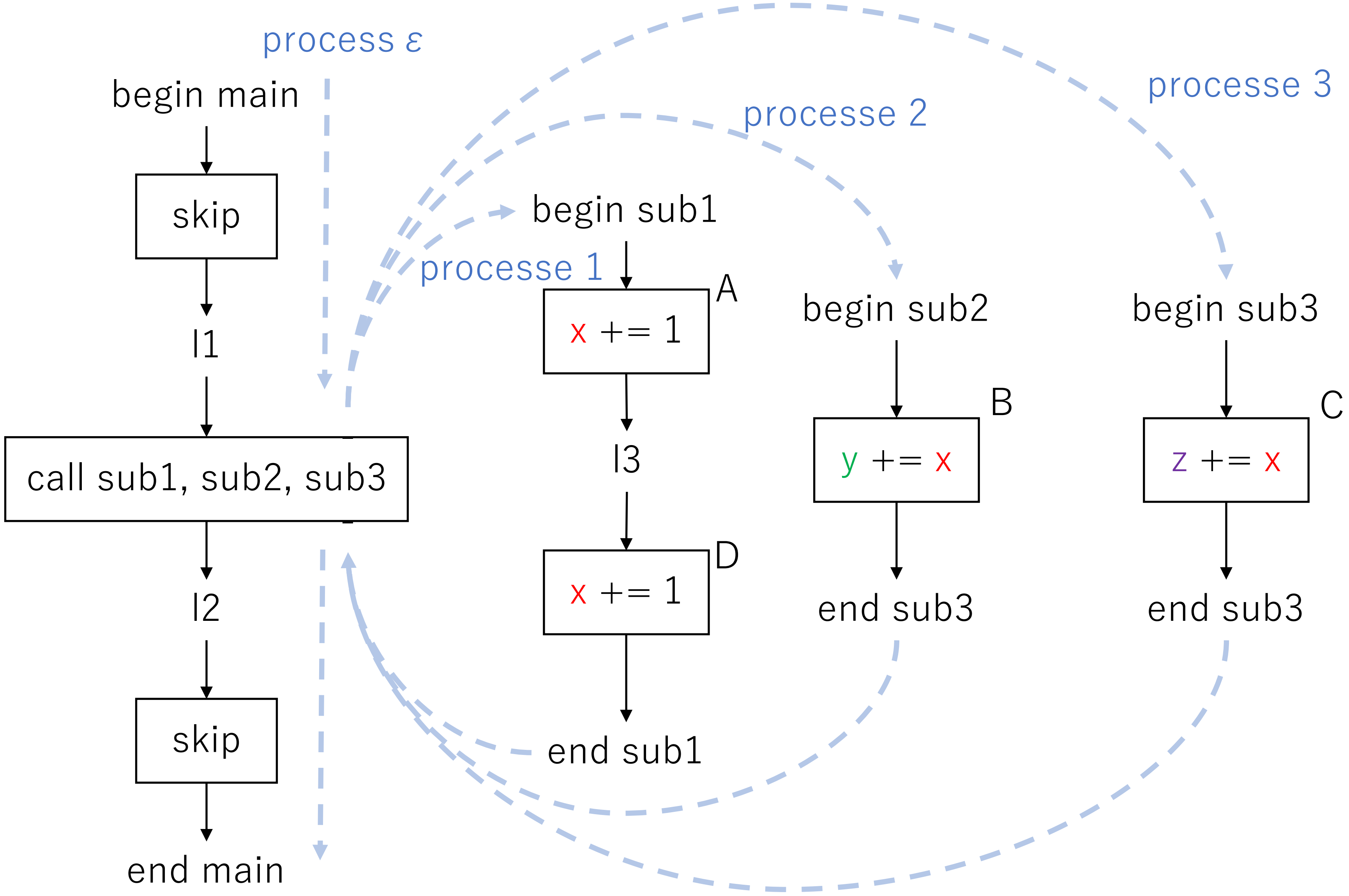
Uncontrolled CRIL Behavior

- $\xrightleftharpoons[\text{prog}]{p,Rd,Wt}$ is reversible in one-step.

- However, the operational semantics does not make a well-defined CRIL program reversible.

- We shall see **what happens** if **undo** is performed **without controling backward exeution**.

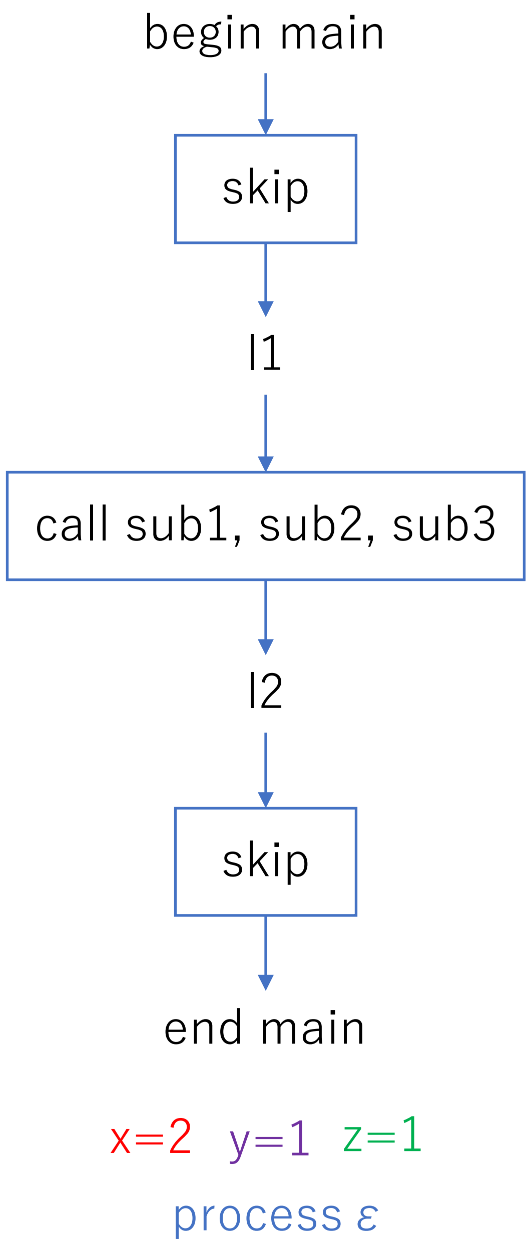
```
  begin main
  skip
  -> l1
  l1 <-
  call sub1, sub2, sub3
  -> l2
  l2 <-
  skip
  end main

  begin sub1
A x += 1
  -> l3
  l3 <-
D x += 1
  end sub1

  begin sub2
B y += x
  end sub2

  begin sub3
C z += x
  end sub3
```

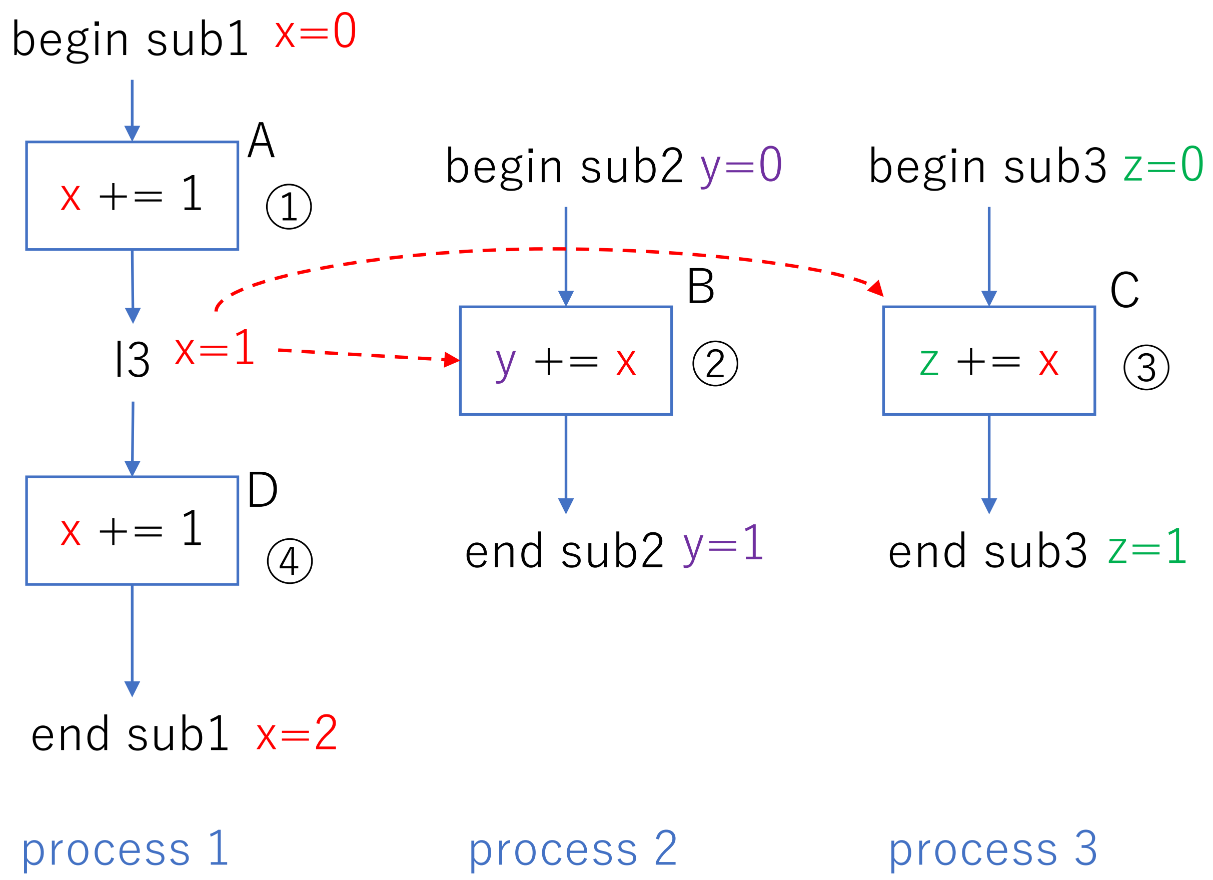


A → B → C → D in the **forward** direction.

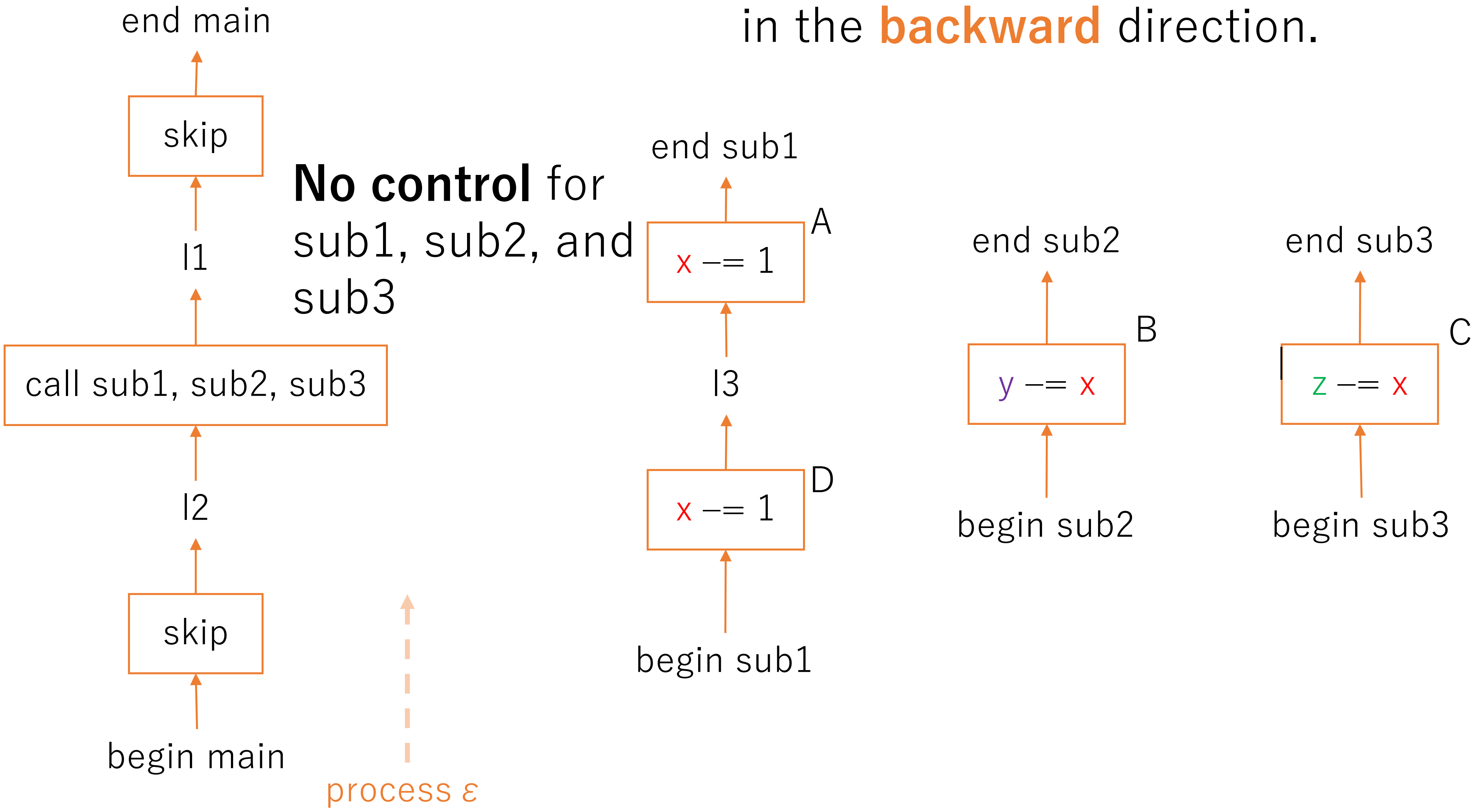
All the variables are initially 0.

x=0 y=0 z=0





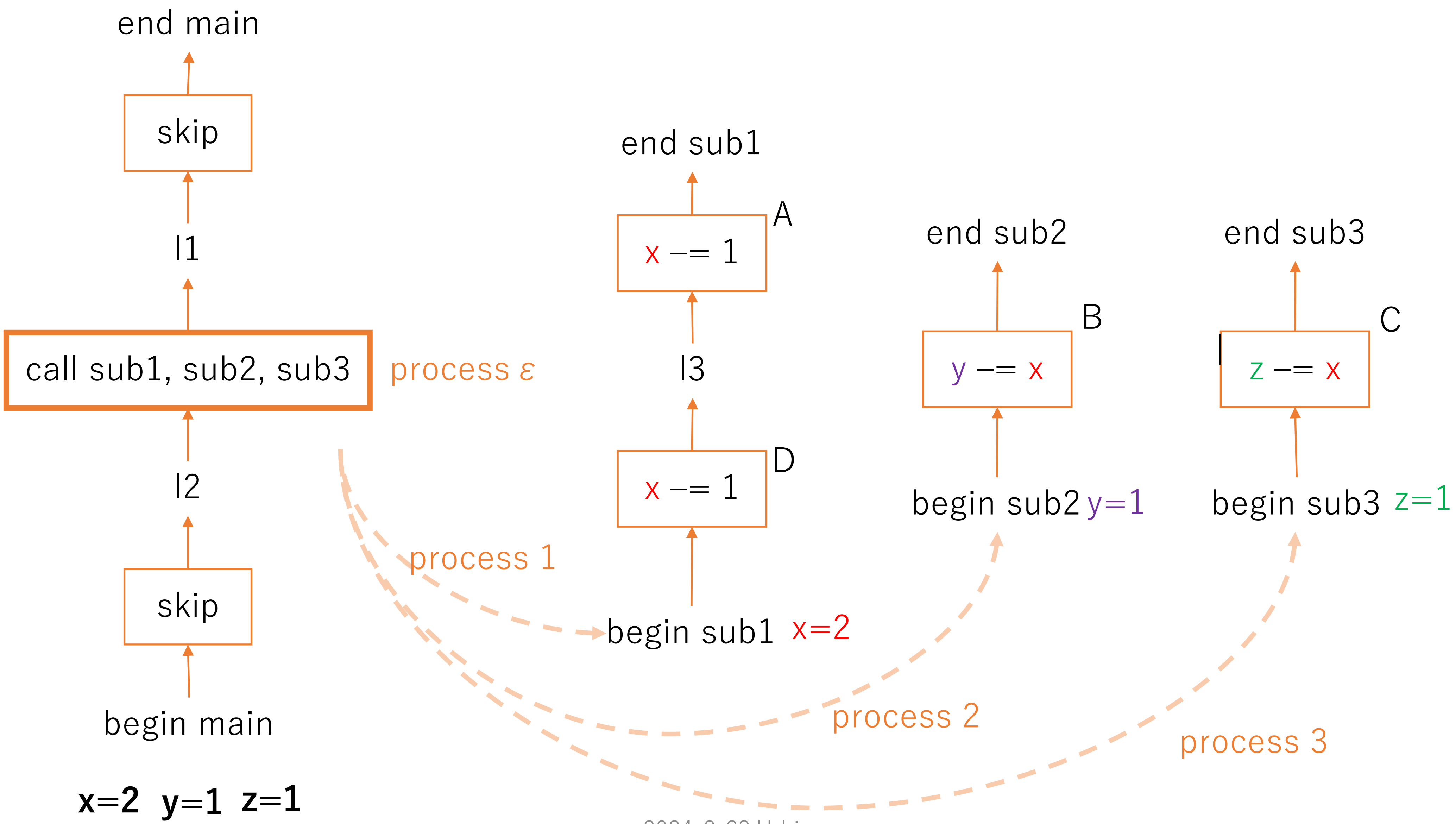
If the blocks are executed in the order C → B → D → A (not D → C → B → A) in the **backward** direction.



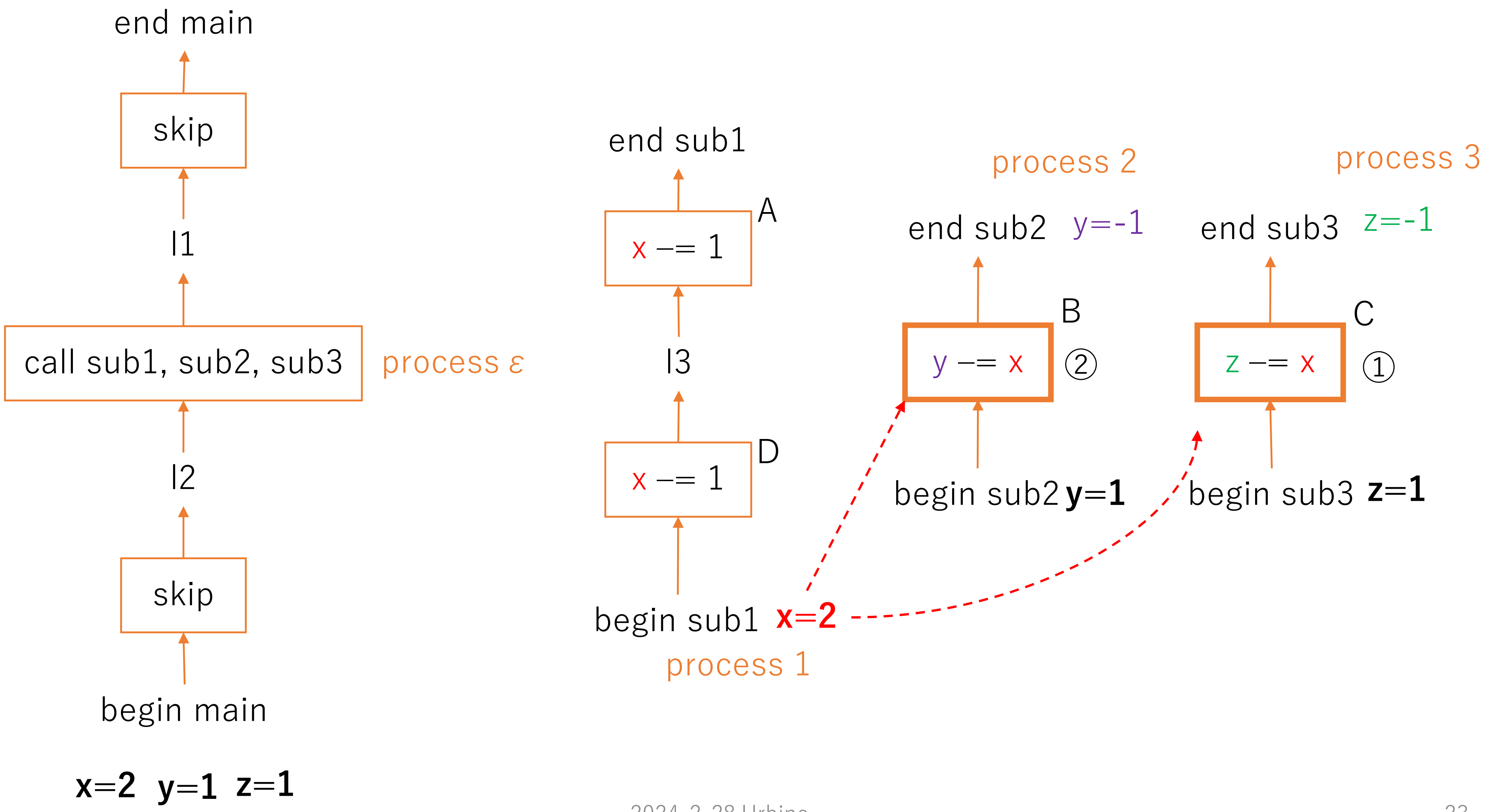
No control for sub1, sub2, and sub3

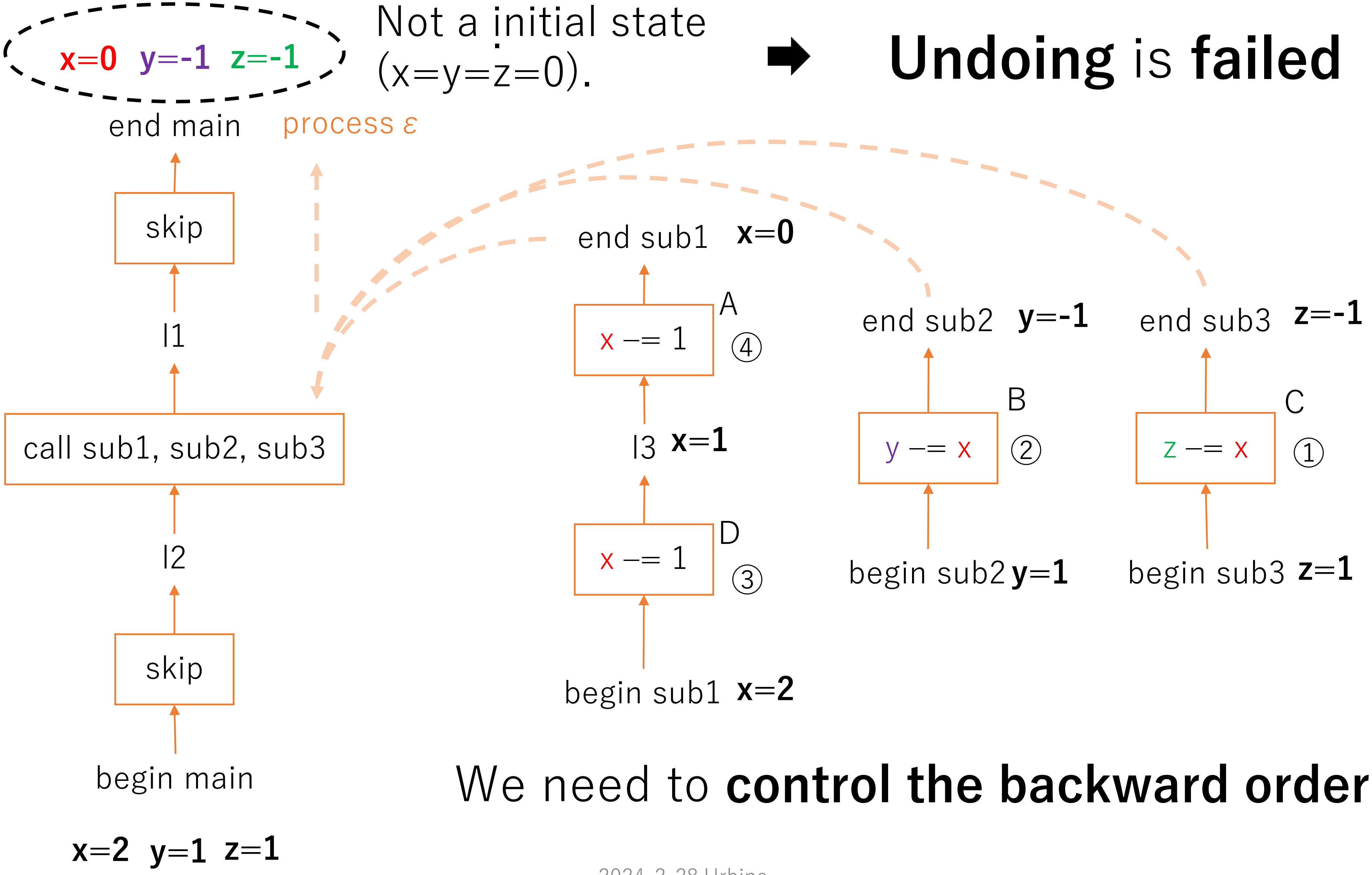
x=2 y=1 z=1

If C → B → D → A in the backward direction.



If C → B → D → A in the backward direction.





Not a initial state (x=y=z=0). ➡ **Undoing** is **failed**

We need to **control the backward order.**

Contents

- Background & Motivation
- CRIL: Concurrent Reversible Intermediate Language
- The controlled semantics of CRIL
- Reversibility Properties
- Bidirectional Data Flow Analysis in CRIL
- Constant Propagation in CRIL
- Concluding Remarks

Controled semantics with Annotation DAG

- We require the information about the causal relationship among basic blocks (≒ the order of reading and updating for each variable).
- We introduce a data structure called annotation DAG (Directed Acyclic Graph).

Program configuration Transition **Annotation DAG Transition**

$$\frac{C \overset{p,Rd,Wt}{\underset{\mathrm{prog}}{\rightleftharpoons}} C' \qquad \boxed{A \overset{p,Rd,Wt}{\underset{\mathrm{ann}}{\rightleftharpoons}} A'}}{\boxed{(C,A) \overset{p,Rd,Wt}{\rightleftharpoons} (C',A')}}\ \textbf{ProgAnn}$$

A denotes an annotation DAG.

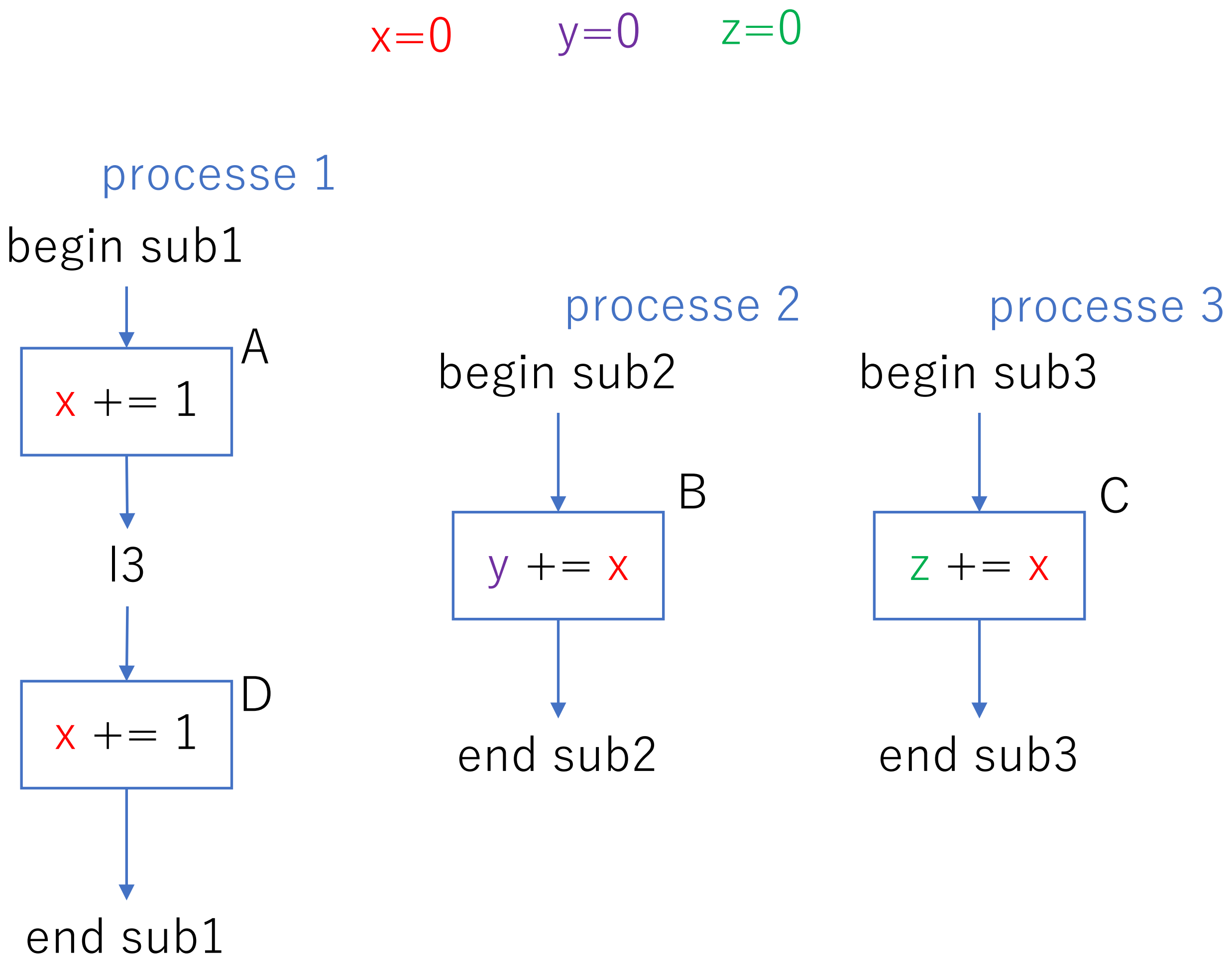
Whole Transition

$$C = (Pg, \rho, \sigma, Pr)$$
$$C' = (Pg, \rho', \sigma', Pr')$$

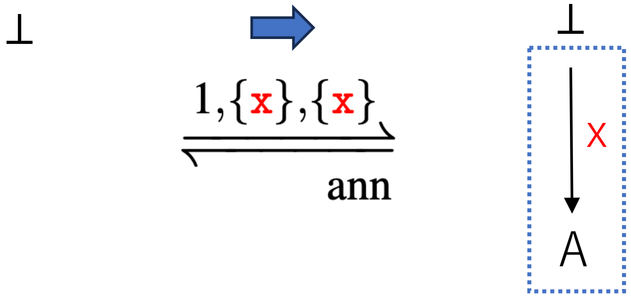
Forward accumulation of causality

In **forward** direction, the annotation DAG **memorizes** the **causality** among the basic blocks.

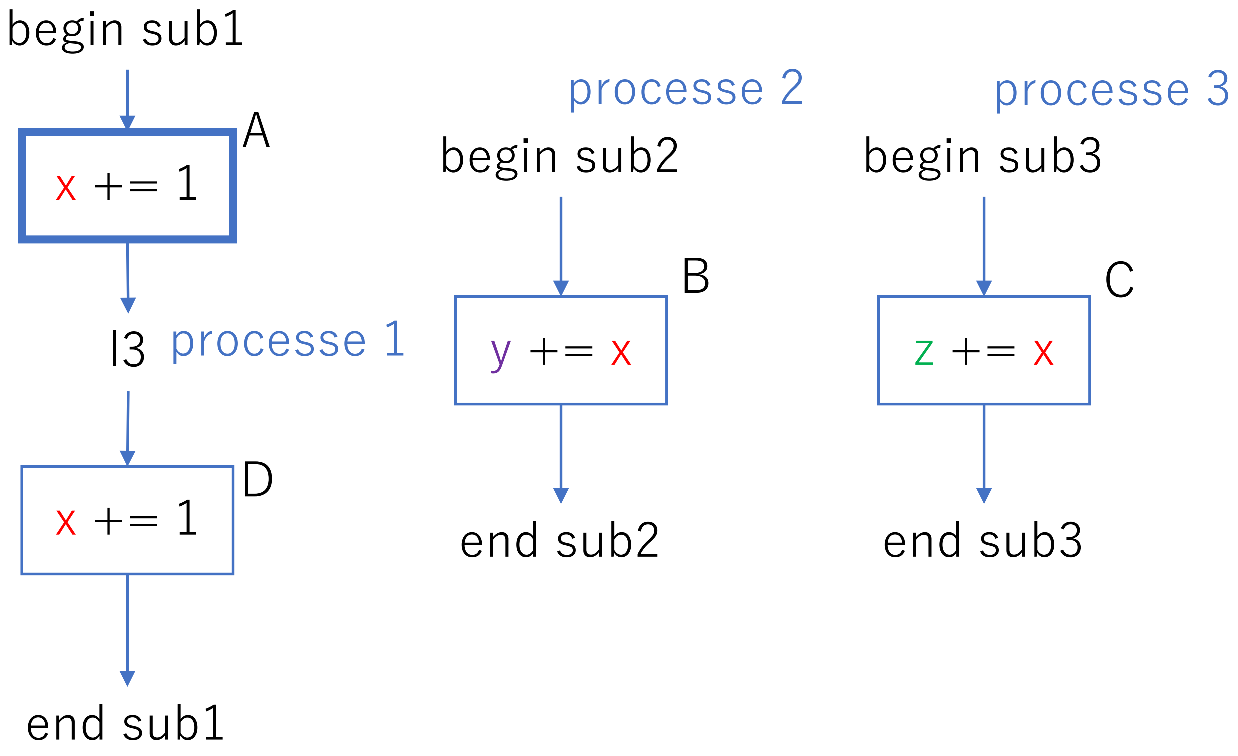
⊥



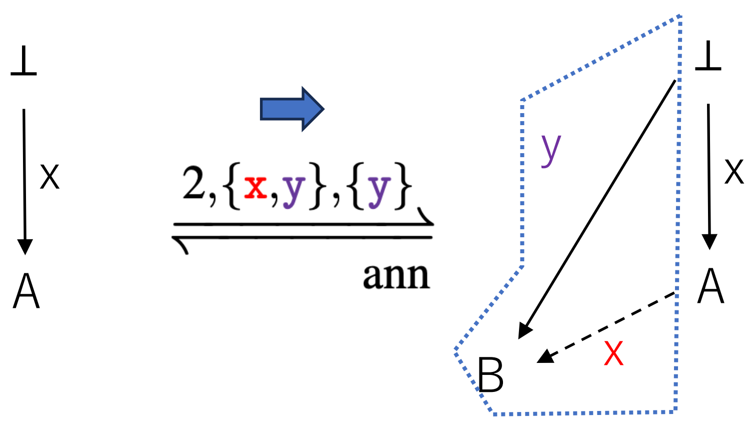
Forward accumulation of causality



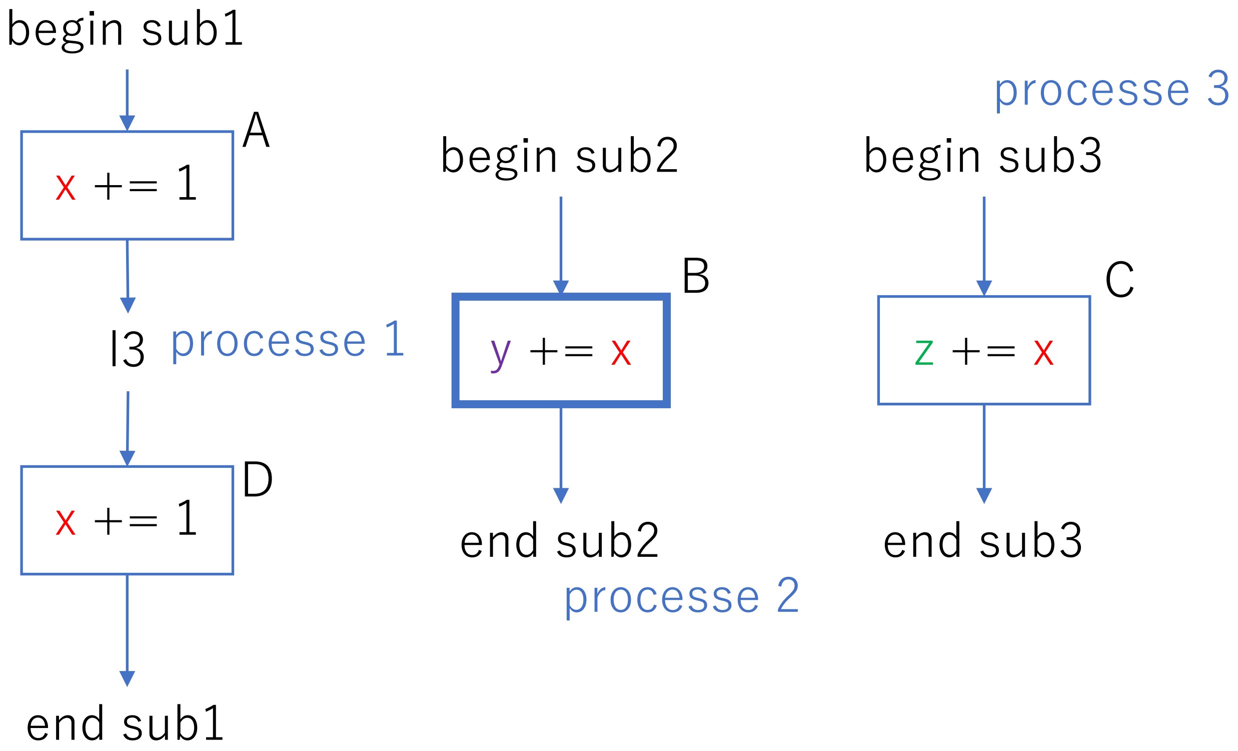
$$x=0 \quad y=0 \quad z=0 \quad \xrightarrow[\text{prog}]{1,\{x\},\{x\}} \quad x=1 \quad y=0 \quad z=0$$



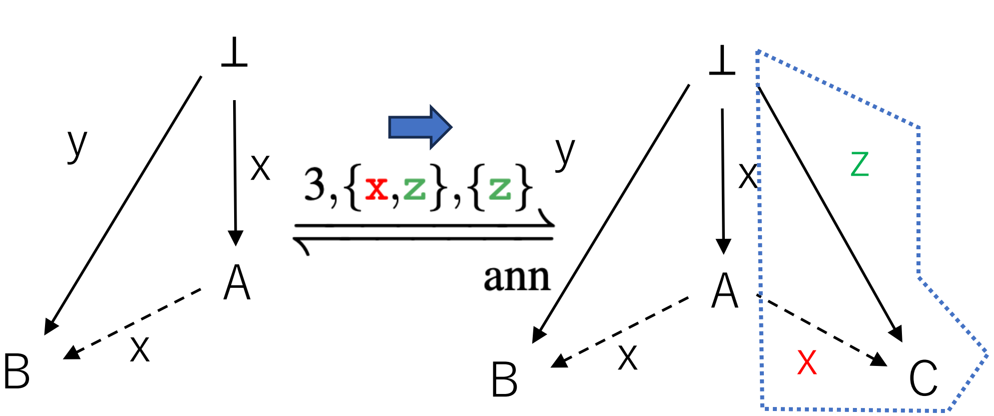
Forward accumulation of causality

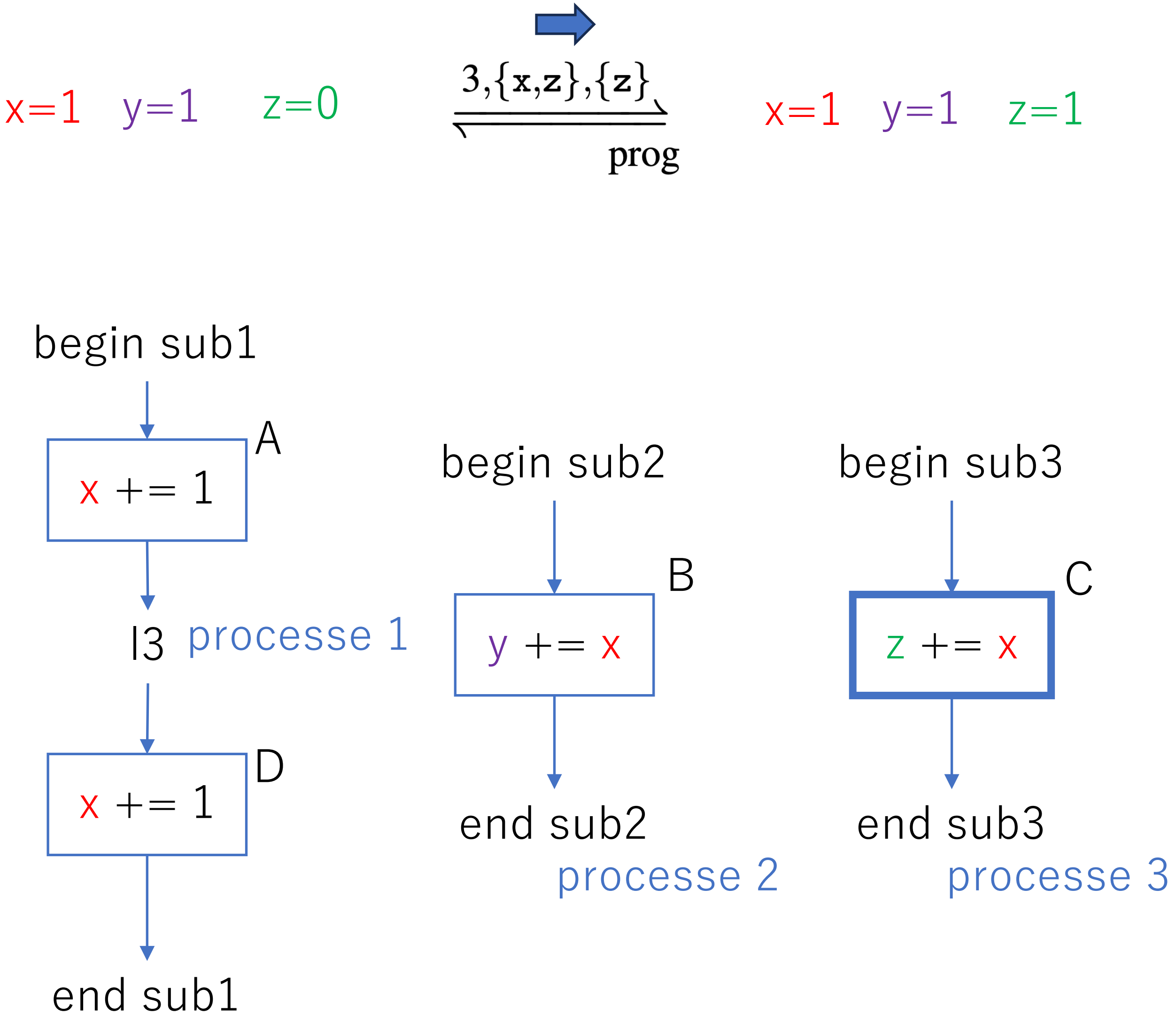


$x=1 \quad y=0 \quad z=0 \;\; \underset{\textbf{prog}}{\overset{2,\{x,y\},\{x\}}{\rightleftharpoons}} \;\; x=1 \quad y=1 \quad z=0$

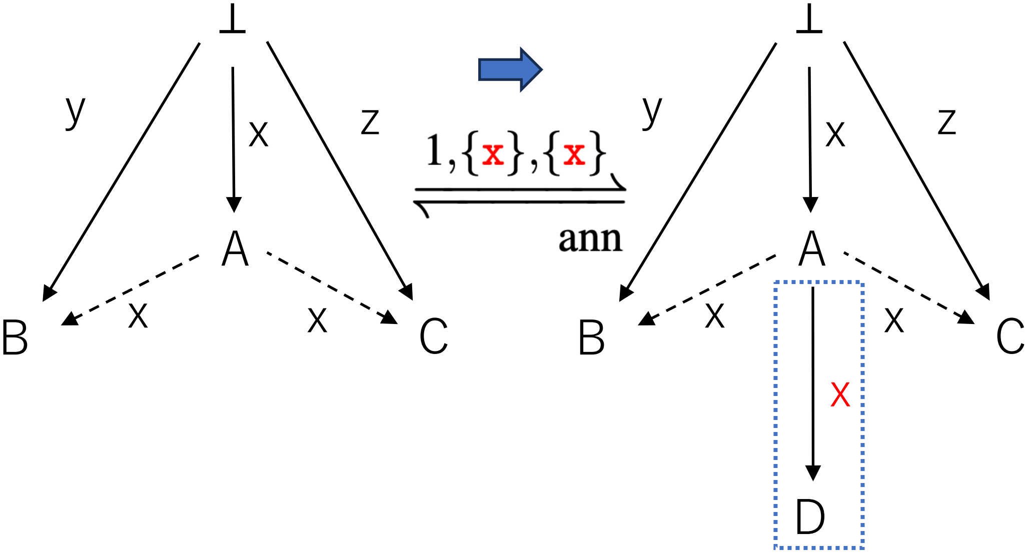


Forward accumulation of causality

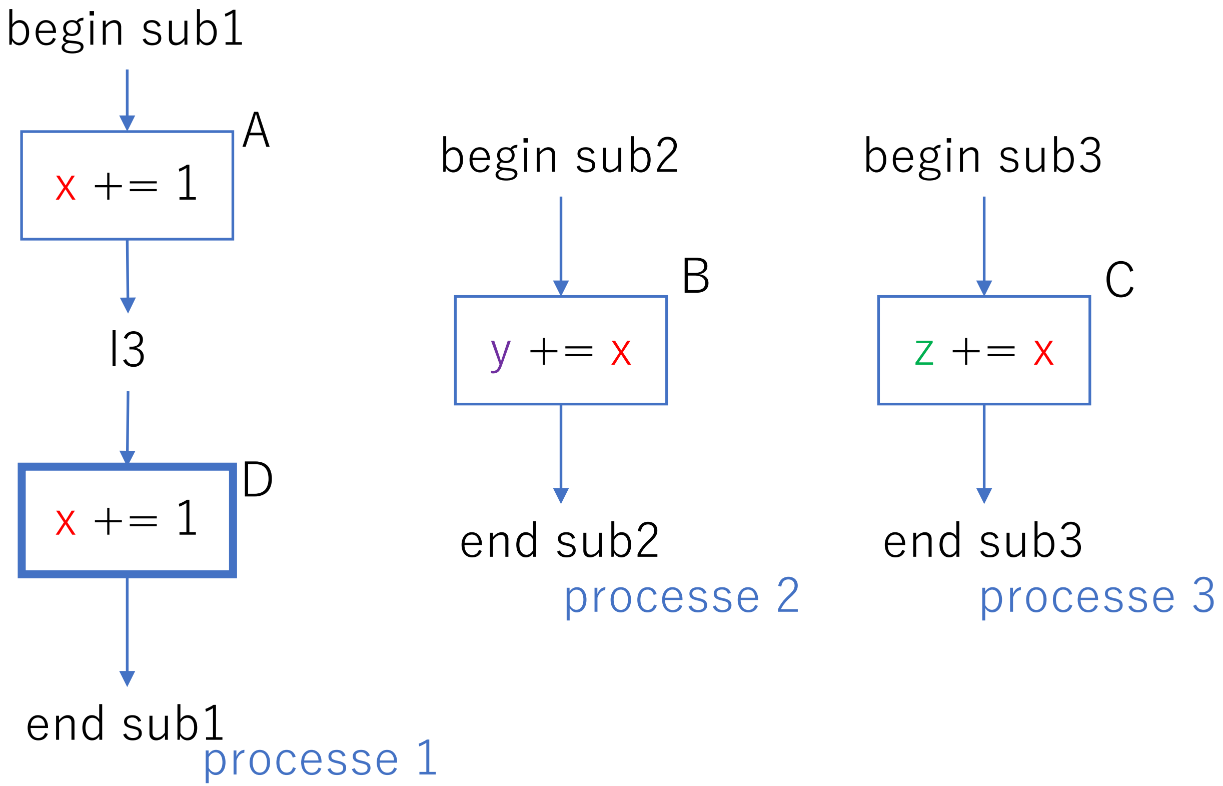




Forward accumulation of causality

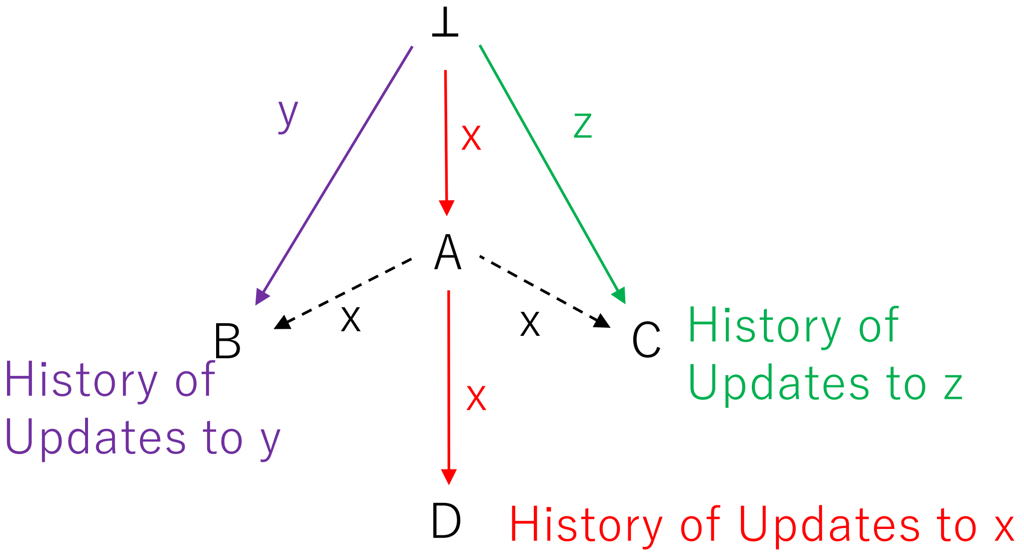


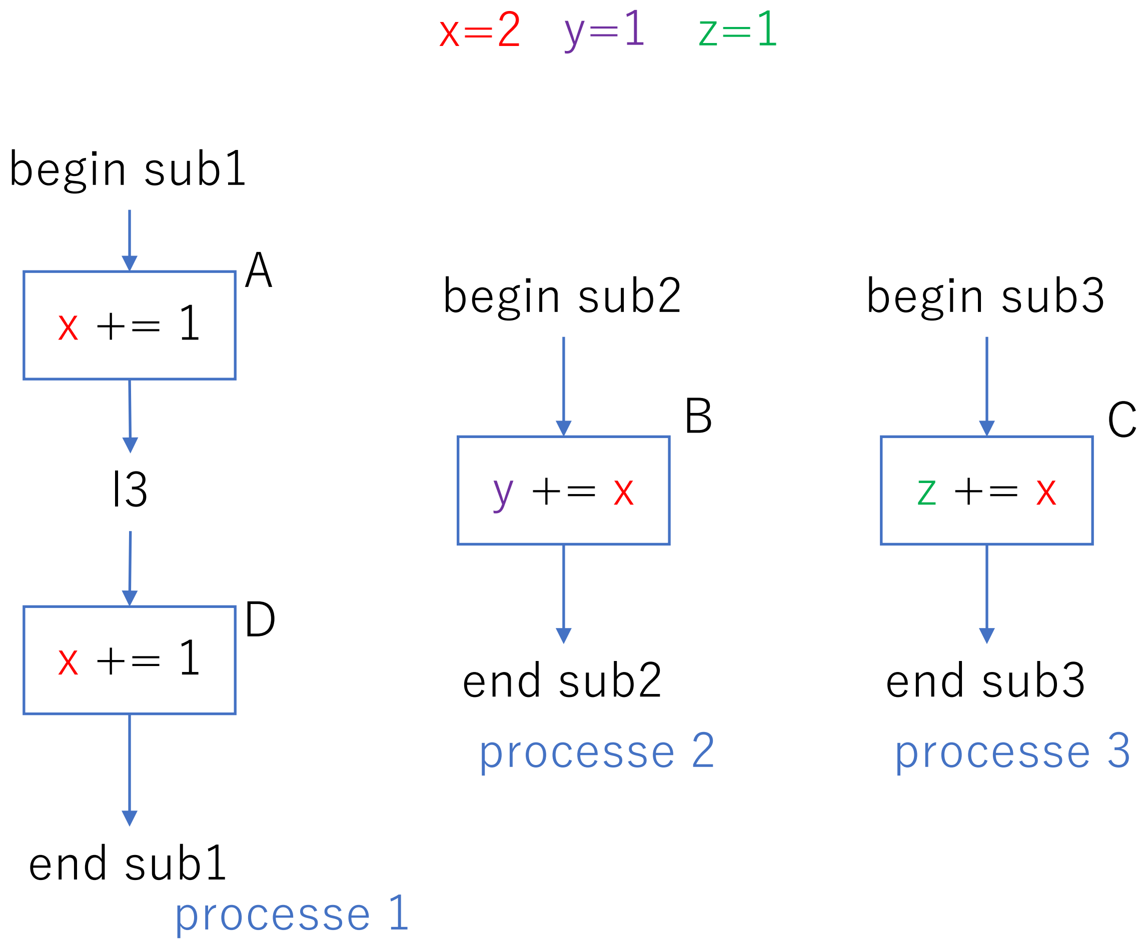
$$\text{x=1}\quad \text{y=1}\quad \text{z=1}\quad \overset{1,\{x\},\{x\}}{\underset{\text{prog}}{\rightleftharpoons}}\quad \text{x=2}\quad \text{y=1}\quad \text{z=1}$$



Forward accumulation of causality

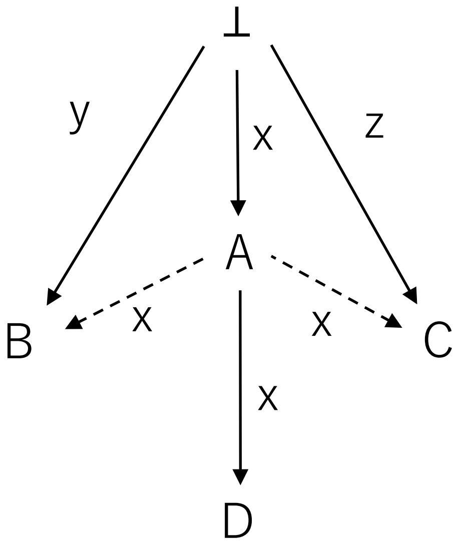
Note: an annotation DAG **does not remember** the **values of variables**.



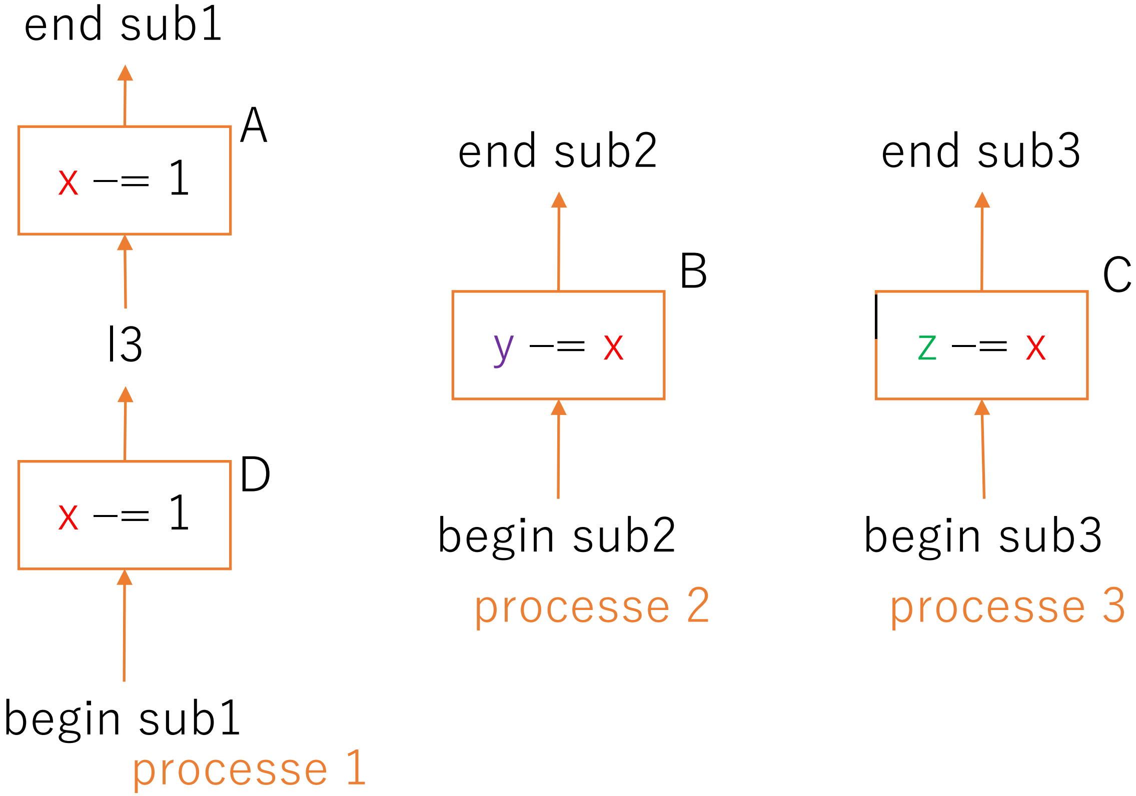


Backward rollback of causality

In backward direction, the Annotation DAG **controls** the execution by **matching the causality**.

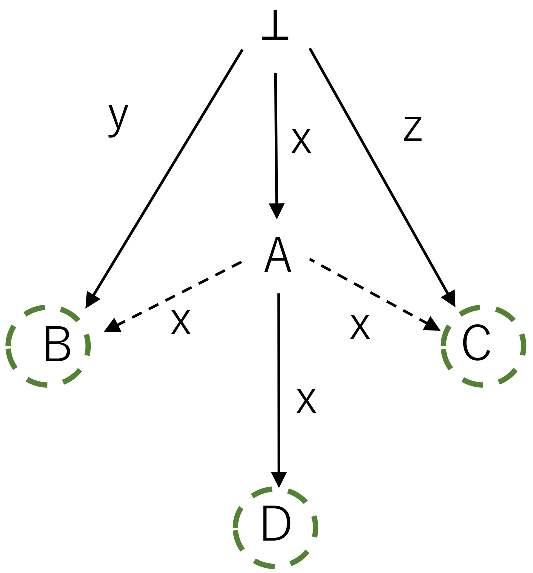


x=2 y=1 z=1



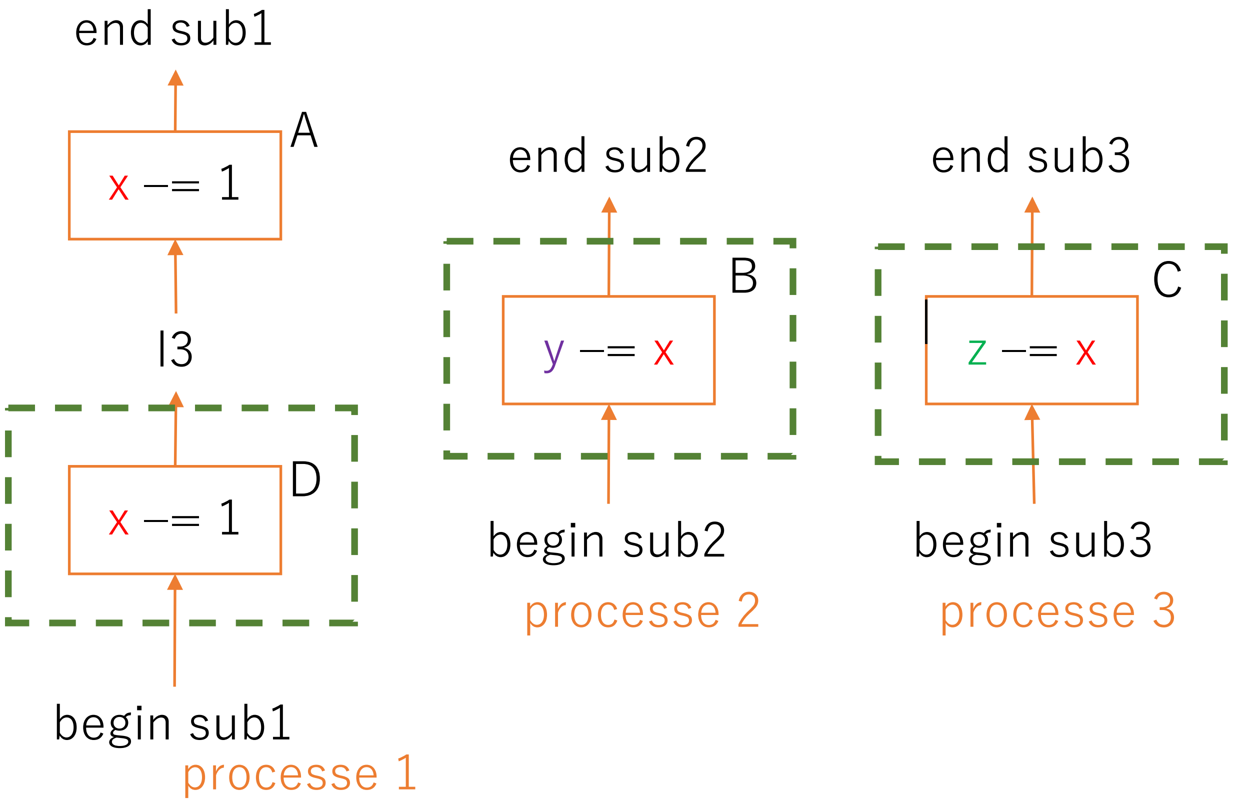
Backward rollback of causality

B and C receive the value of x from



B, C, or D may be executed backward.

x=2 y=1 z=1



end sub1

A

x -= 1

l3

D

x -= 1

begin sub1

processe 1

end sub2

B

y -= x

begin sub2

processe 2

end sub3

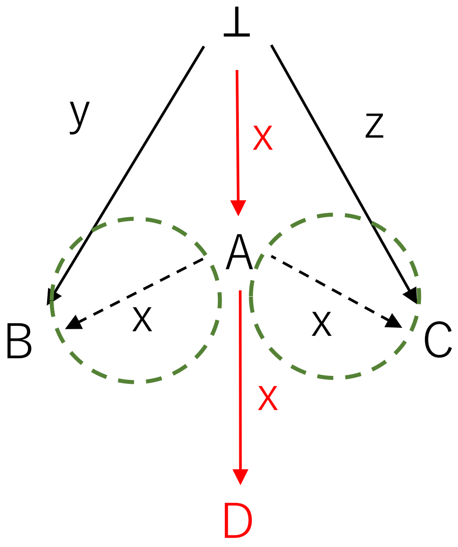
C

z -= x

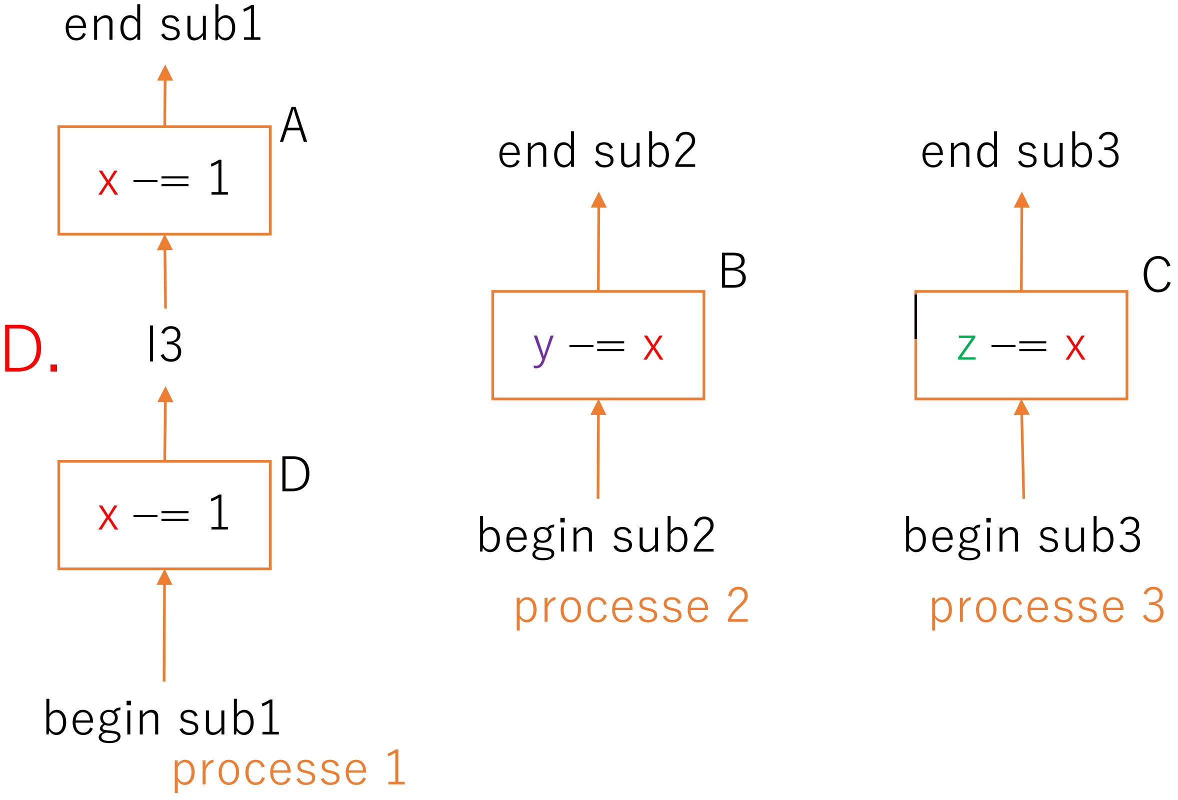
begin sub3

processe 3

Backward rollback of causality



x=2 y=1 z=1



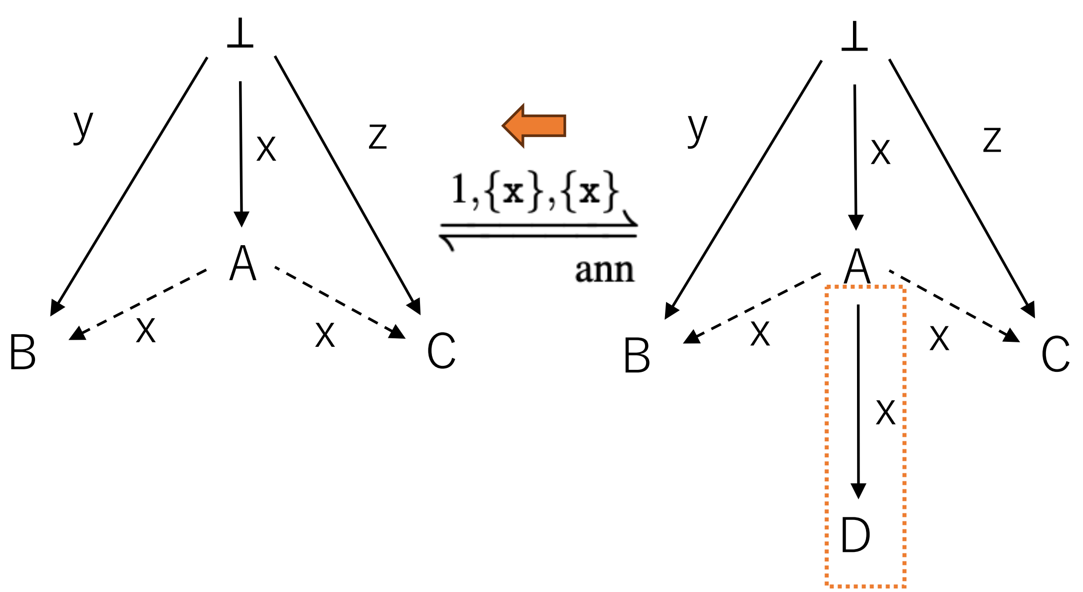
The current value of x is defined by D.

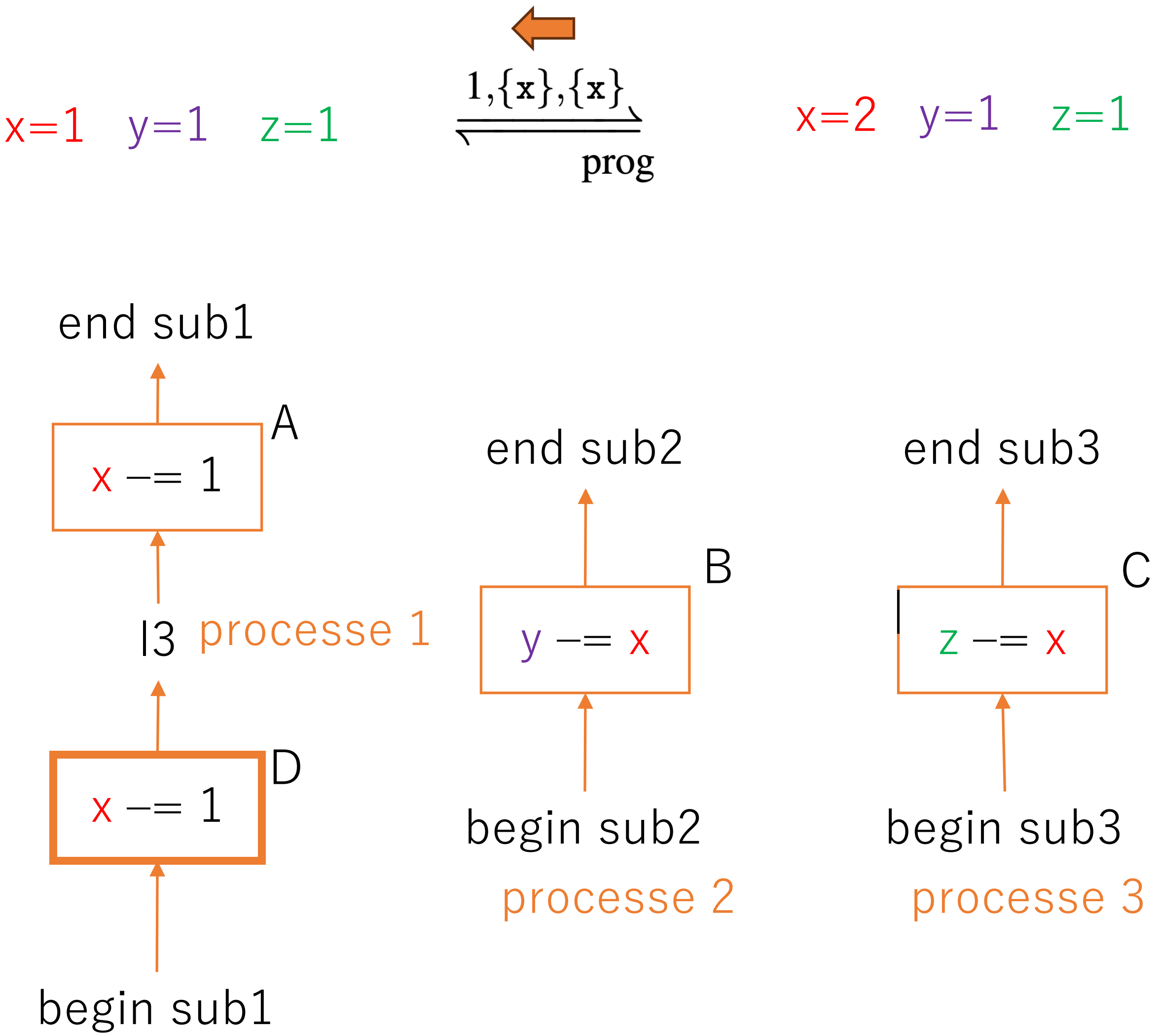
However, B and C do not receive the current value of x .

⬇

B and C **cannot** be reversed.

Backward rollback of causality

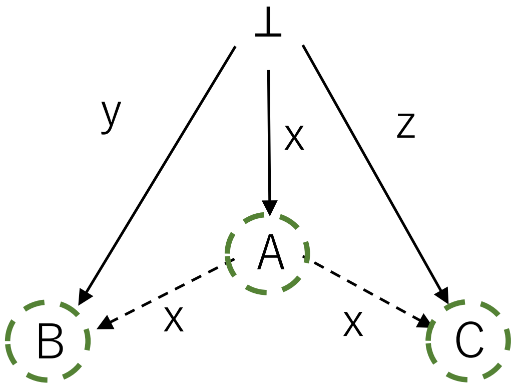




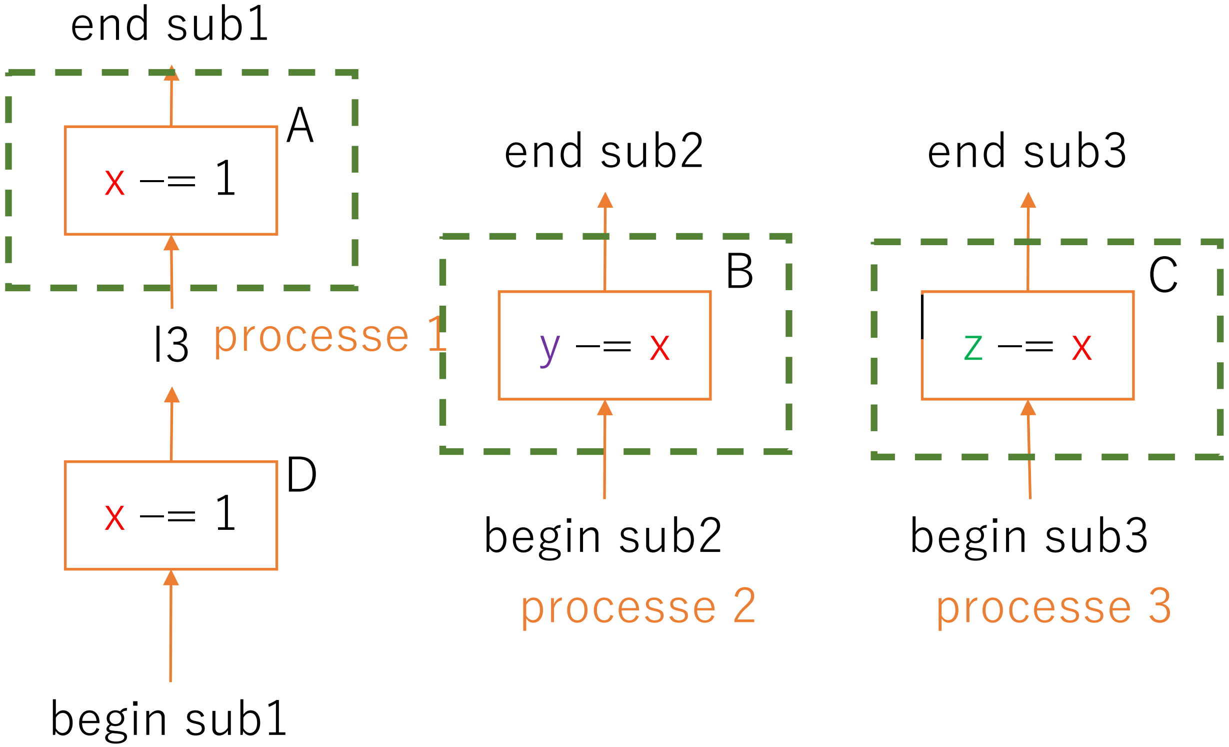
Only D **can** be reversed.

D is removed from the annotation DAG

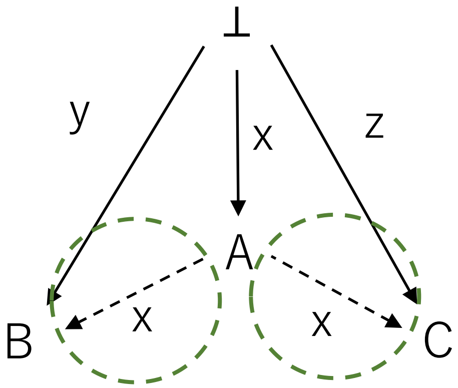
Backward rollback of causality



B, C, or D may be executed backward.



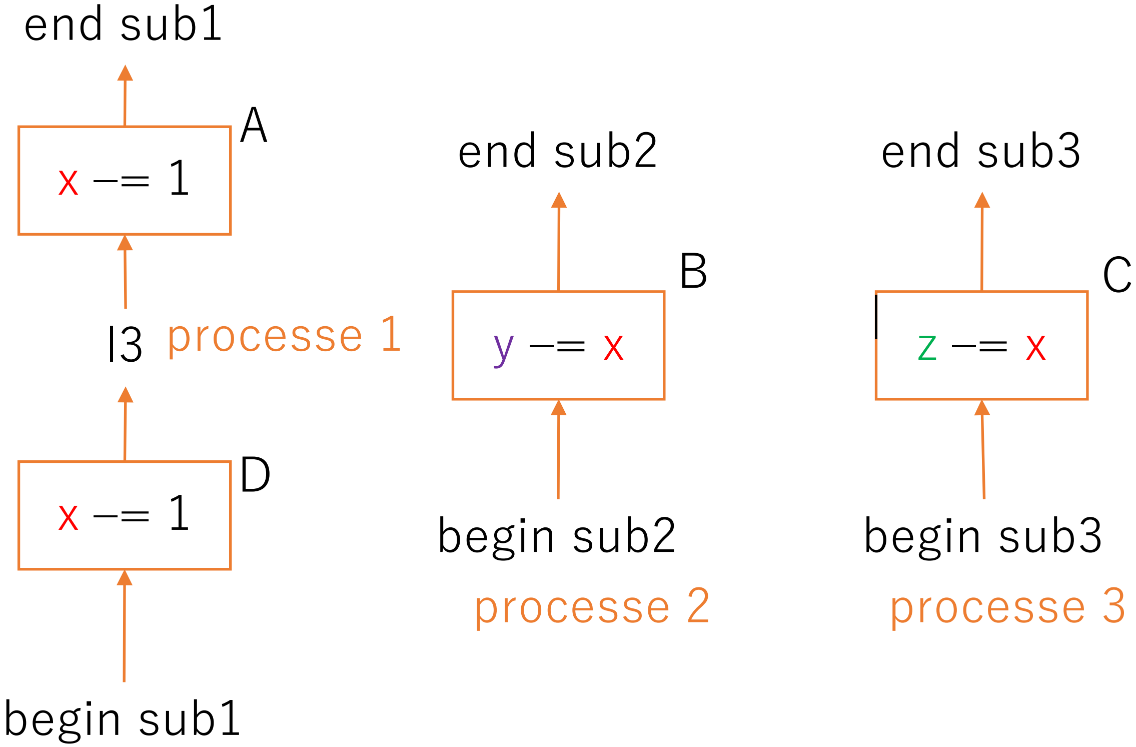
Backward rollback of causality



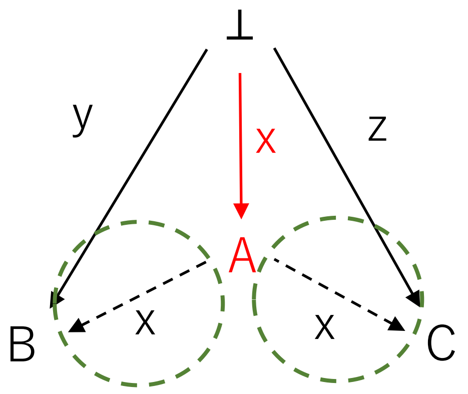
There is still causality from A to B and C.

⬇

A **cannot** be reversed



Backward rollback of causality

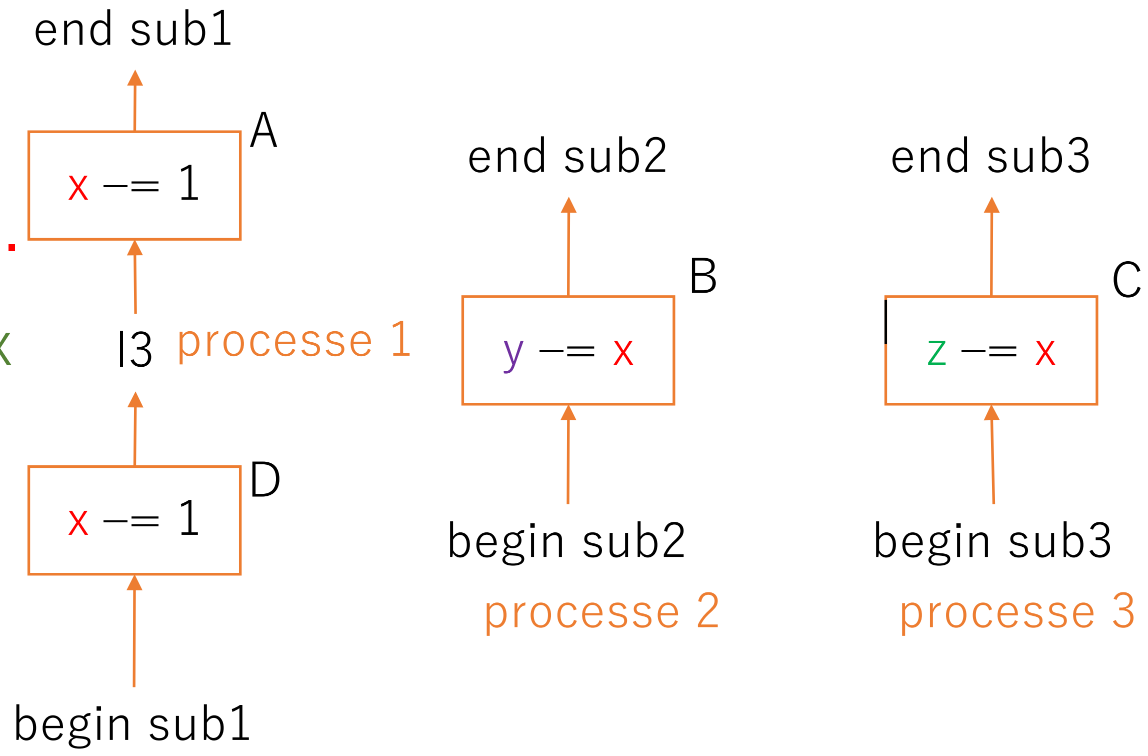


The current value of x is defined by D.

B and C receive the current value of x

🡇

B and C **can** be reversed



end sub1

A

x −= 1

l3 processe 1

D

x −= 1

begin sub1

end sub2

B

y −= x

begin sub2

processe 2

end sub3

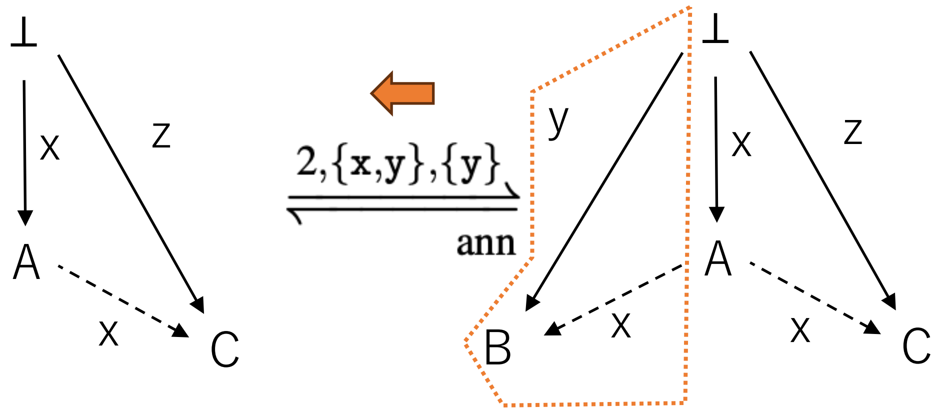
C

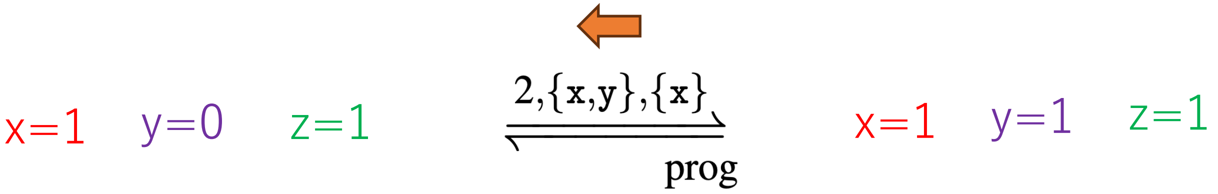
z −= x

begin sub3

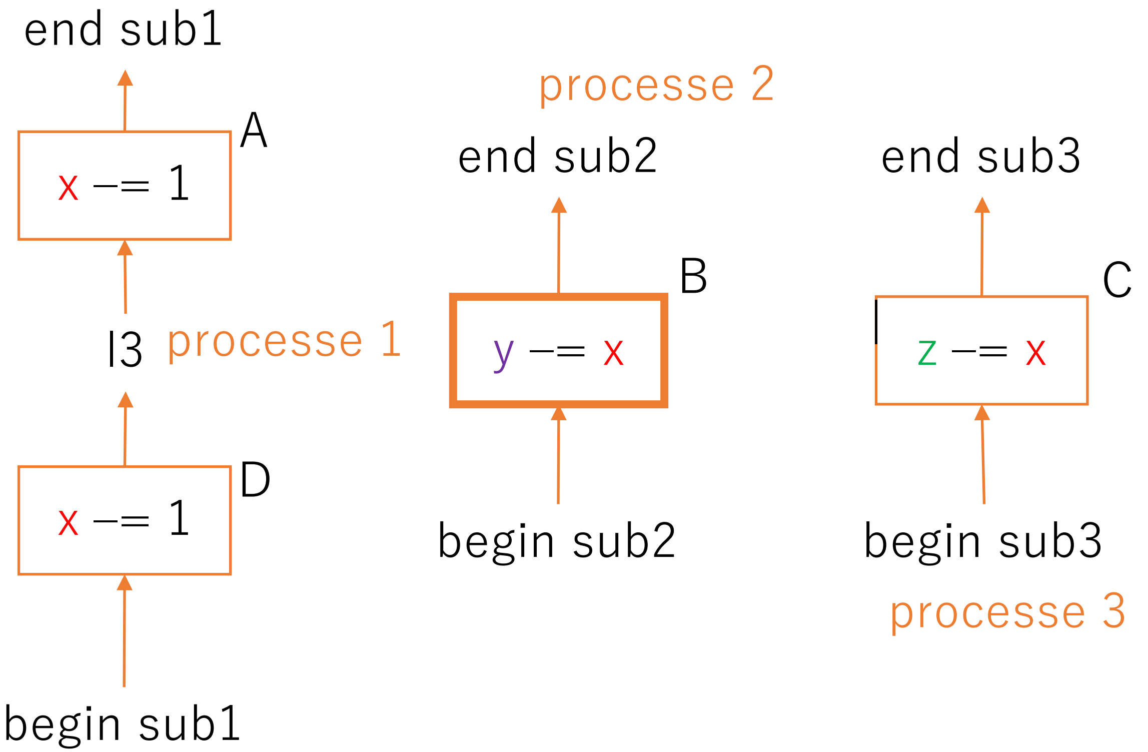
processe 3

Backward rollback of causality



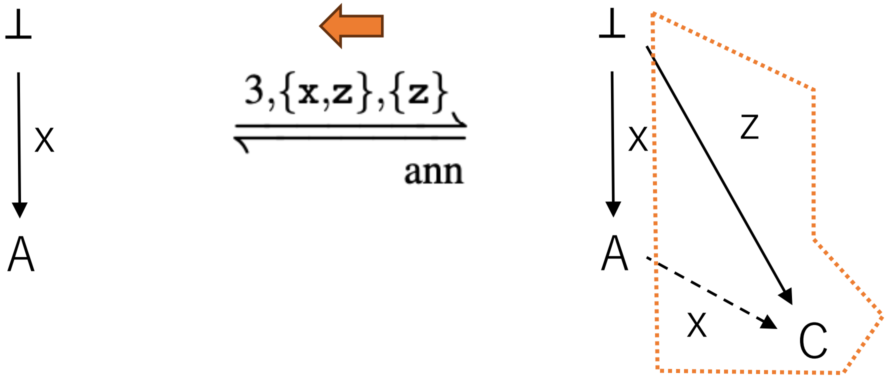


We can reverse **either B or C first.**

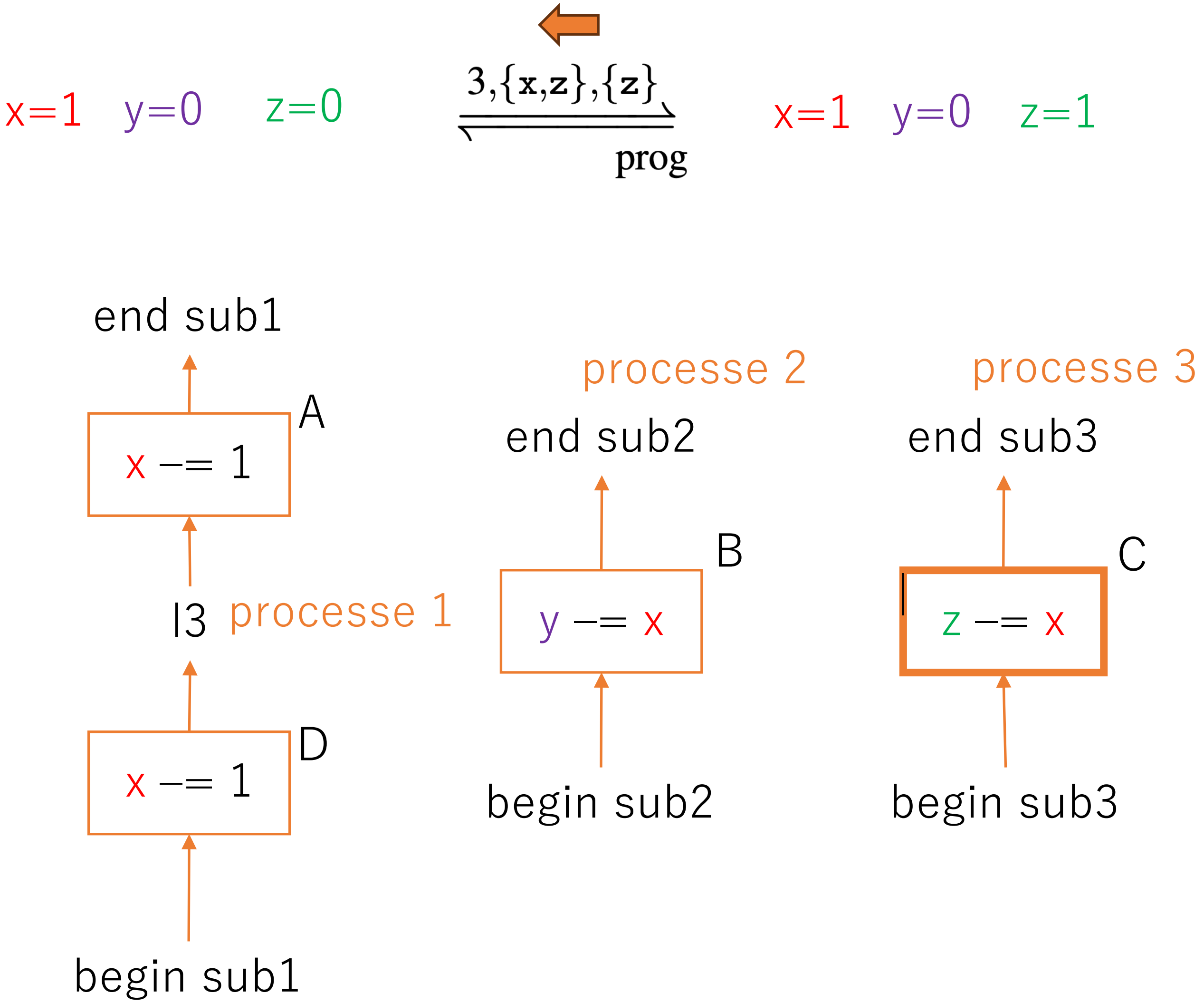


B is reversed first.

Backward rollback of causality

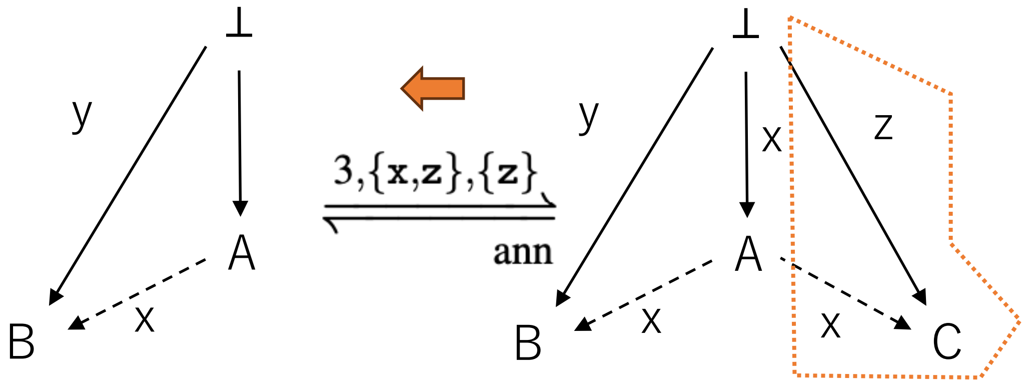


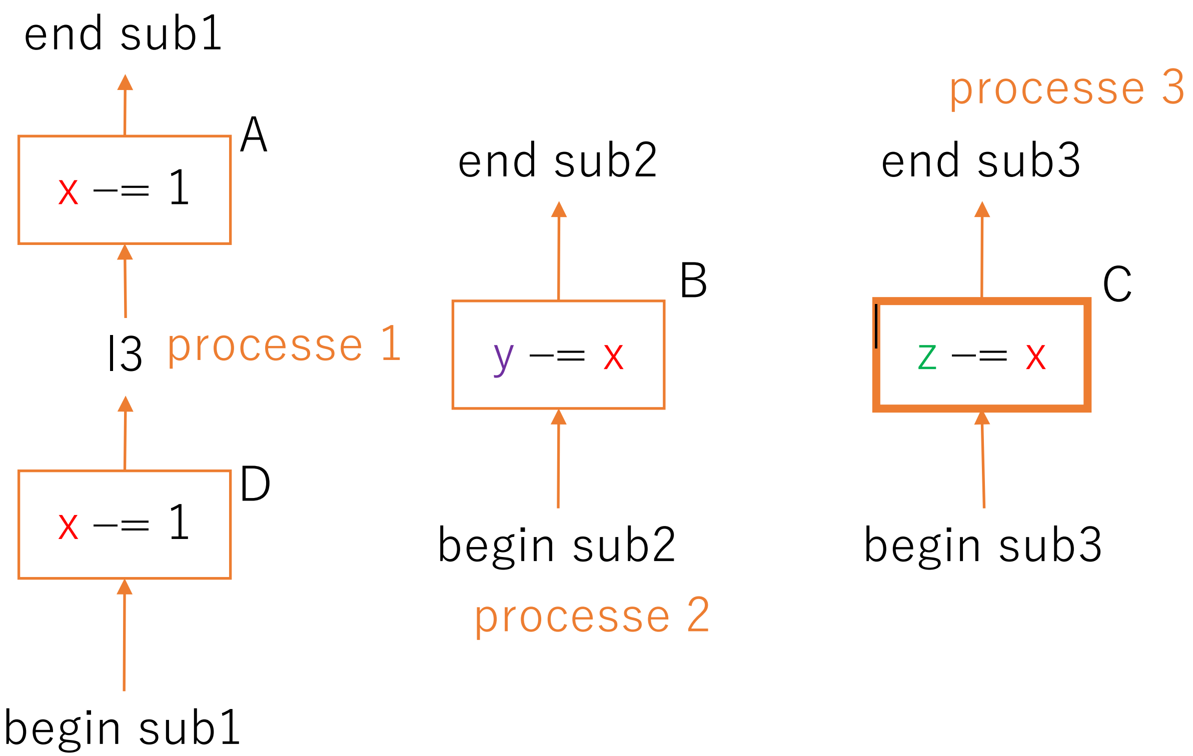
Next, C is reversed.



Backward rollback of causality

$$x=1 \quad y=1 \quad z=0 \xleftharpoondown[\text{prog}]{3,\{x,z\},\{z\}} x=1 \quad y=1 \quad z=1$$

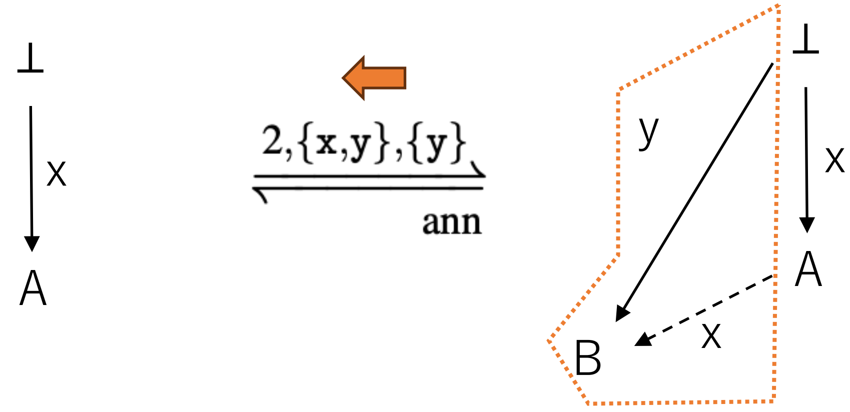


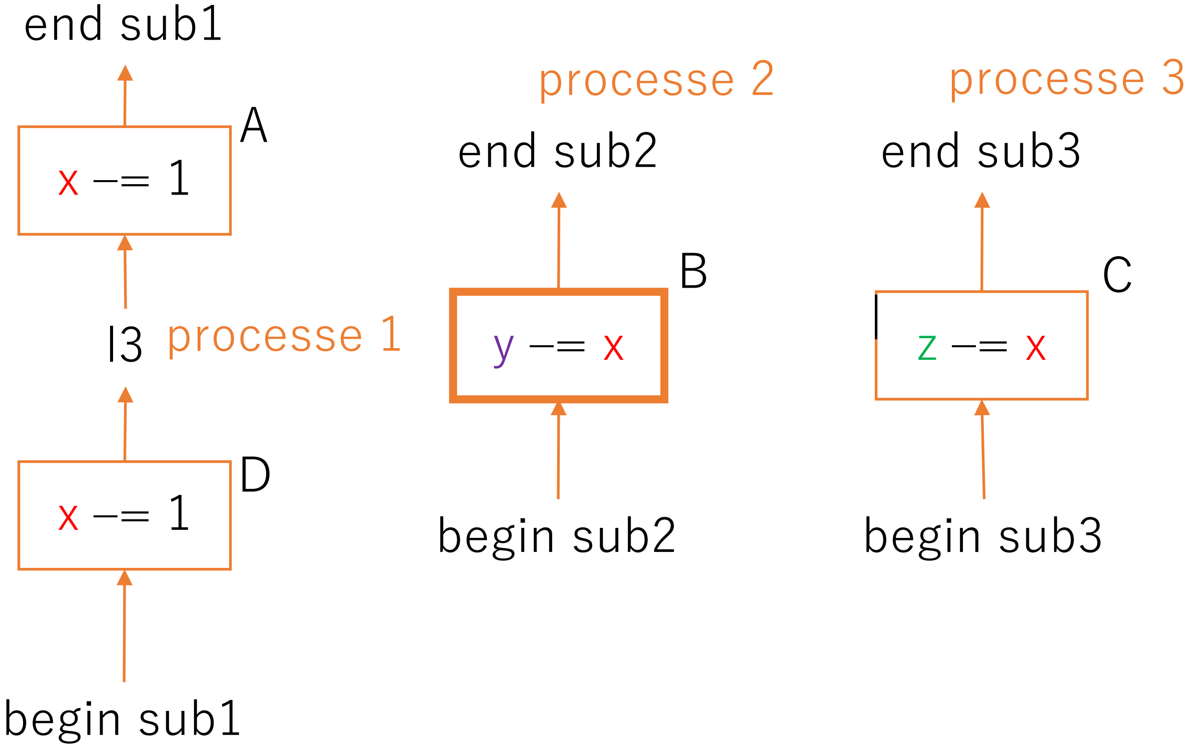


C is reversed first.

Backward rollback of causality

$$x=1 \quad y=0 \quad z=0 \quad \xrightleftharpoons[\text{prog}]{2,\{x,y\},\{x\}} \quad x=1 \quad y=1 \quad z=0$$

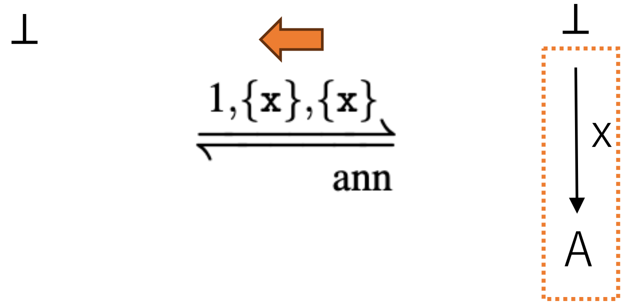




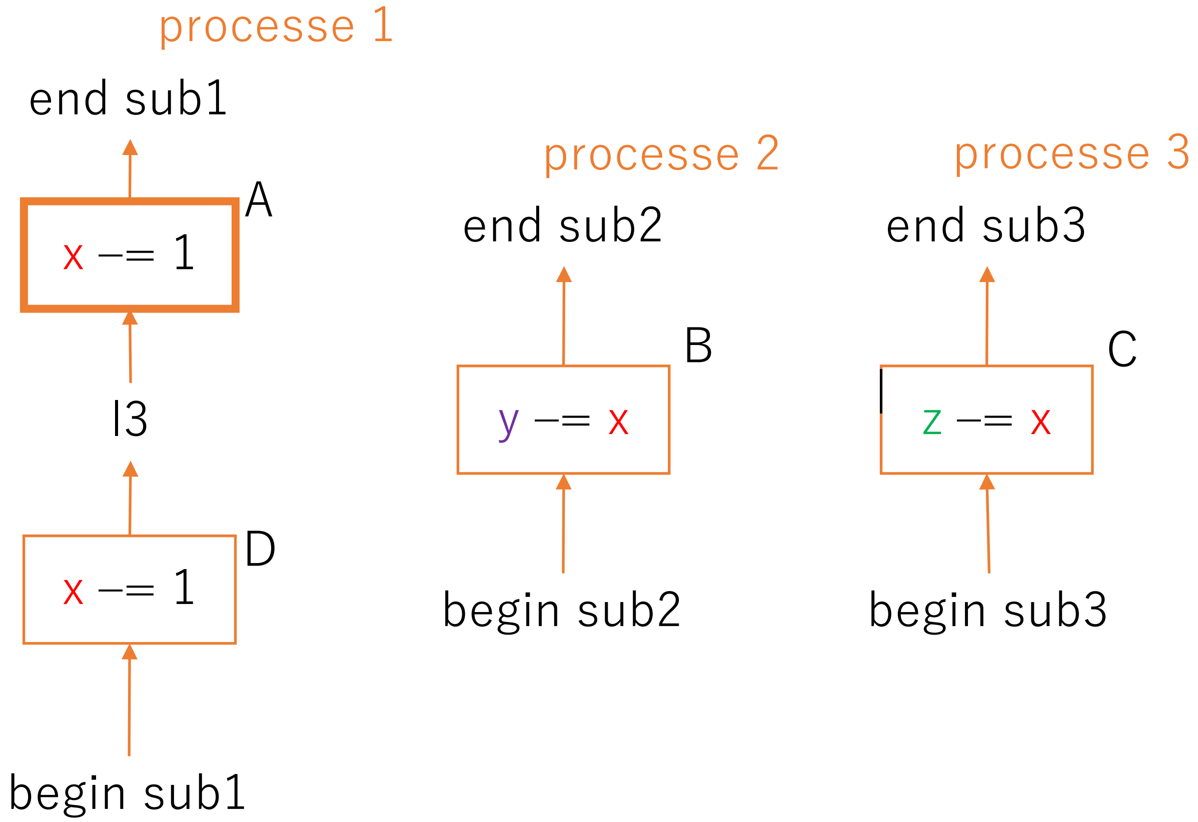
Next, B is reversed.

Whether B or C is executed first,
we get the same annotation DAG.

Backward rollback of causality



$x=0 \quad y=0 \quad z=0 \quad \xleftarrow[\text{prog}]{1,\{x\},\{x\}} \quad x=1 \quad y=0 \quad z=0$



No one receives a value from A.

⬇

A **can** be reversed.

Contents

- Background & Motivation
- CRIL: Concurrent Reversible Intermediate Language
- The controlled semantics of CRIL
- Reversibility Properties
- Bidirectional Data Flow Analysis in CRIL
- Constant Propagation in CRIL
- Concluding Remarks

Causal Safety and Causal Liveness

We show CRIL has the Causal Safety and the Causal Liveness[Lanese + 2020].

Causal Safety (CS)：An action can not be reversed until any actions caused by it have been reversed

Causal Liveness (CL)：We should allow actions to reverse in any order compatible with Causal Safety, not nessearily the exact inverse of the forward order.

In CRIL,

- **CS** ensures that the **same value** is used **in both forward and backward** direction.
- **CL** ensures that a program **does not deadlock** without returning to its initial state in backward direction.

Independence Relation

CS and **CL** hold in an LTS with independence if some axioms are valid in it [Lanese + 20].

Two transitions are independent if

1. two processes are **not** in a **parent-child relationship**; and
2. one of them **does not read** a variable **written by the other**.

1.

$$\underline{p_1 \not\leq p_2} \;\land\; p_2 \not\leq p_1$$

p_2 is not the subprocess of p_1

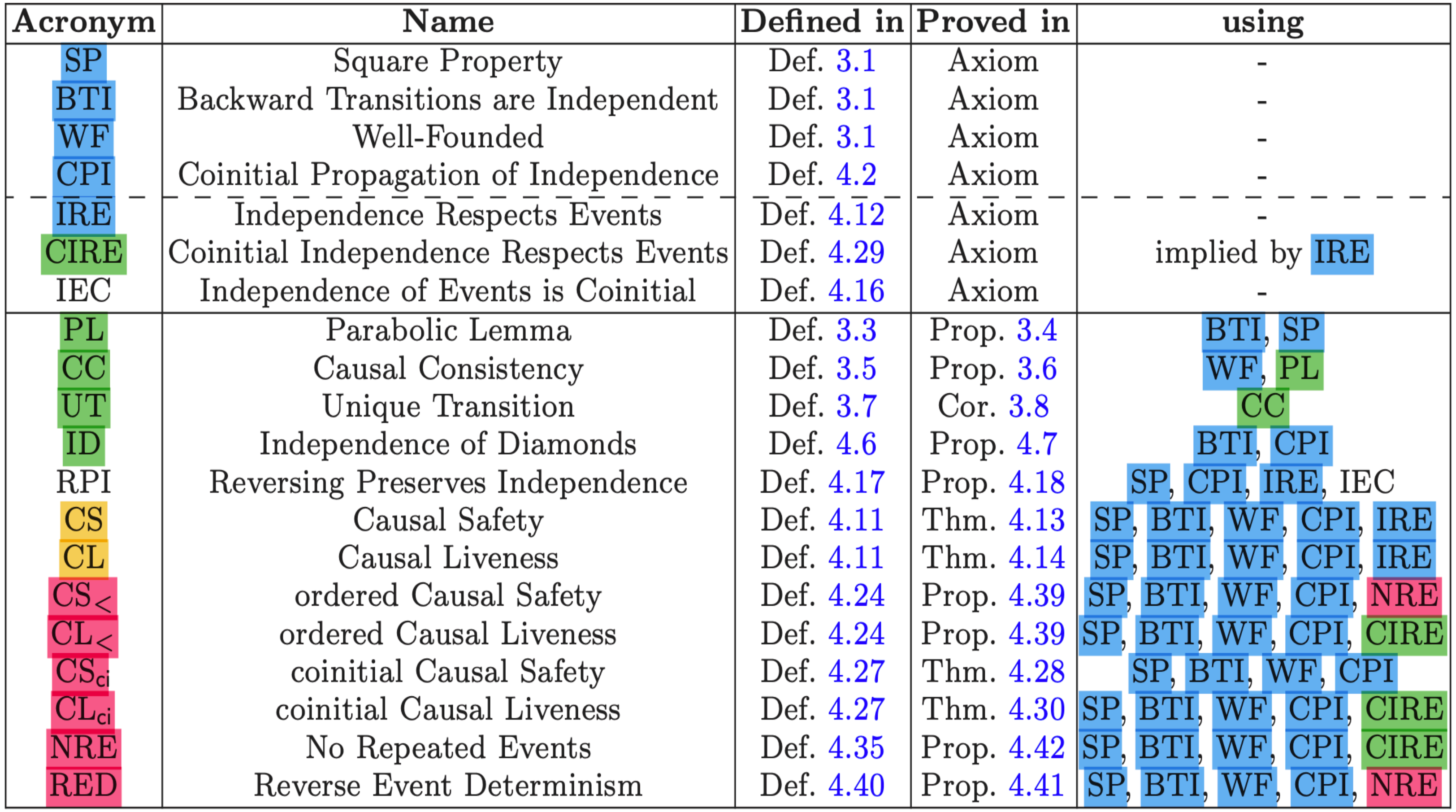
2.

$$Rd_1 \cap Wt_2 = \varnothing \;\land Rd_2 \cap Wt_1 = \varnothing$$

Axiomatic Approach [Lanese+ 20]

SP, BTI, WF, CPI, and IRE hold for LTSI of CRIL

CS & CL

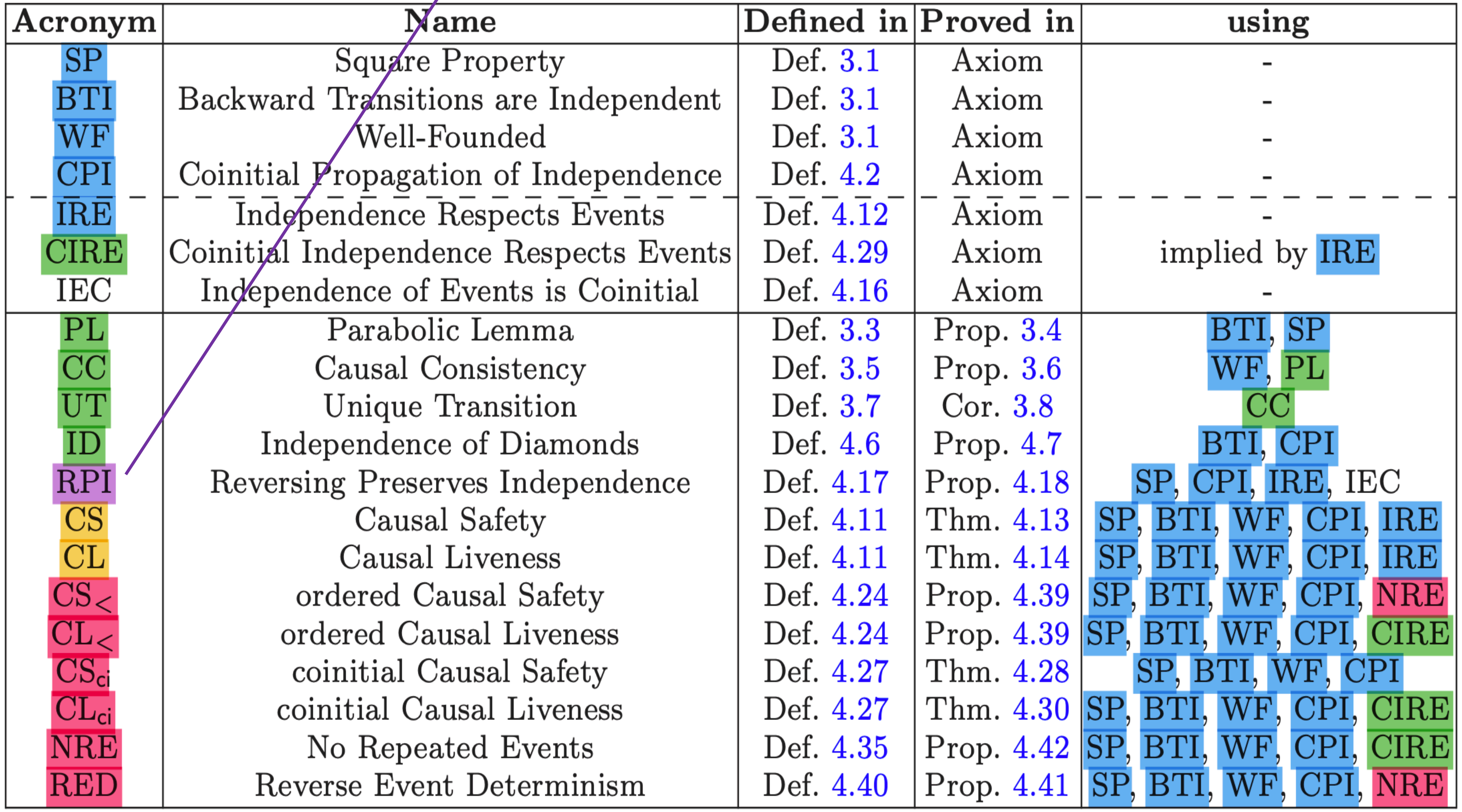


Acronym	Name	Defined in	Proved in	using
SP	Square Property	Def. 3.1	Axiom	-
BTI	Backward Transitions are Independent	Def. 3.1	Axiom	-
WF	Well-Founded	Def. 3.1	Axiom	-
CPI	Coinitial Propagation of Independence	Def. 4.2	Axiom	-
IRE	Independence Respects Events	Def. 4.12	Axiom	-
CIRE	Coinitial Independence Respects Events	Def. 4.29	Axiom	implied by IRE
IEC	Independence of Events is Coinitial	Def. 4.16	Axiom	-
PL	Parabolic Lemma	Def. 3.3	Prop. 3.4	BTI, SP
CC	Causal Consistency	Def. 3.5	Prop. 3.6	WF, PL
UT	Unique Transition	Def. 3.7	Cor. 3.8	CC
ID	Independence of Diamonds	Def. 4.6	Prop. 4.7	BTI, CPI
RPI	Reversing Preserves Independence	Def. 4.17	Prop. 4.18	SP, CPI, IRE, IEC
CS	Causal Safety	Def. 4.11	Thm. 4.13	SP, BTI, WF, CPI, IRE
CL	Causal Liveness	Def. 4.11	Thm. 4.14	SP, BTI, WF, CPI, IRE
$\mathrm{CS}_<$	ordered Causal Safety	Def. 4.24	Prop. 4.39	SP, BTI, WF, CPI, NRE
$\mathrm{CL}_<$	ordered Causal Liveness	Def. 4.24	Prop. 4.39	SP, BTI, WF, CPI, CIRE
$\mathrm{CS}_{\mathrm{ci}}$	coinitial Causal Safety	Def. 4.27	Thm. 4.28	SP, BTI, WF, CPI
$\mathrm{CL}_{\mathrm{ci}}$	coinitial Causal Liveness	Def. 4.27	Thm. 4.30	SP, BTI, WF, CPI, CIRE
NRE	No Repeated Events	Def. 4.35	Prop. 4.42	SP, BTI, WF, CPI, CIRE
RED	Reverse Event Determinism	Def. 4.40	Prop. 4.41	SP, BTI, WF, CPI, NRE

CRIL also has PL, CC, UT, ID, and RPI.

CRIL is Label-Generated [Lanese + 23].

RPI is derived from LG (not from IEC.)



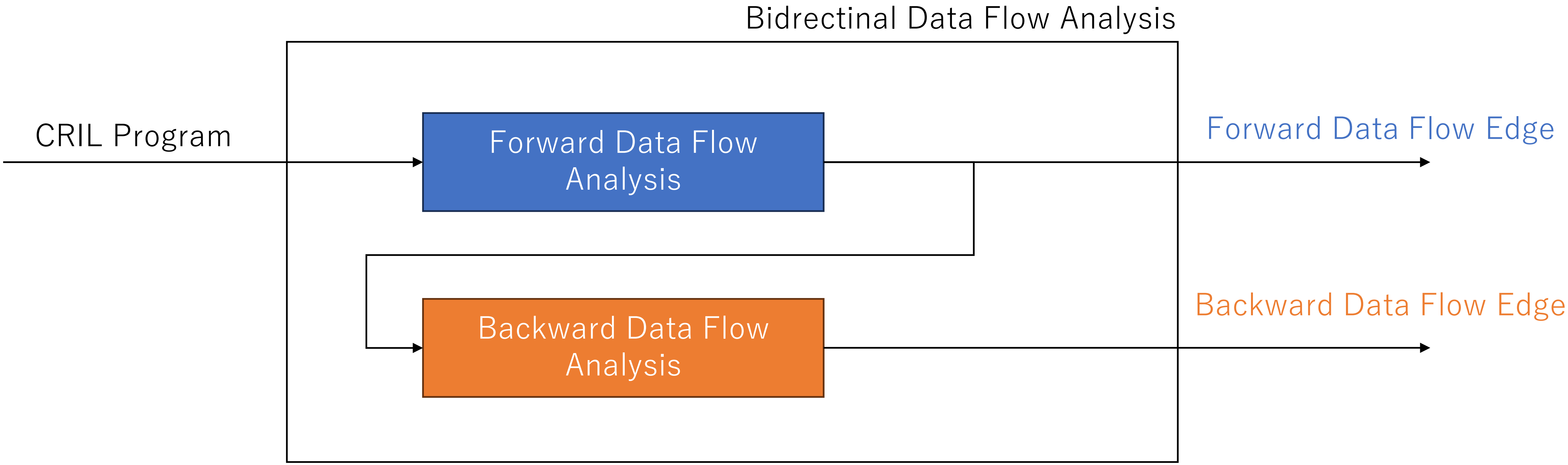
Acronym	Name	Defined in	Proved in	using
SP	Square Property	Def. 3.1	Axiom	-
BTI	Backward Transitions are Independent	Def. 3.1	Axiom	-
WF	Well-Founded	Def. 3.1	Axiom	-
CPI	Coinitial Propagation of Independence	Def. 4.2	Axiom	-
IRE	Independence Respects Events	Def. 4.12	Axiom	-
CIRE	Coinitial Independence Respects Events	Def. 4.29	Axiom	implied by IRE
IEC	Independence of Events is Coinitial	Def. 4.16	Axiom	-
PL	Parabolic Lemma	Def. 3.3	Prop. 3.4	BTI, SP
CC	Causal Consistency	Def. 3.5	Prop. 3.6	WF, PL
UT	Unique Transition	Def. 3.7	Cor. 3.8	CC
ID	Independence of Diamonds	Def. 4.6	Prop. 4.7	BTI, CPI
RPI	Reversing Preserves Independence	Def. 4.17	Prop. 4.18	SP, CPI, IRE, IEC
CS	Causal Safety	Def. 4.11	Thm. 4.13	SP, BTI, WF, CPI, IRE
CL	Causal Liveness	Def. 4.11	Thm. 4.14	SP, BTI, WF, CPI, IRE
$\mathrm{CS}_<$	ordered Causal Safety	Def. 4.24	Prop. 4.39	SP, BTI, WF, CPI, NRE
$\mathrm{CL}_<$	ordered Causal Liveness	Def. 4.24	Prop. 4.39	SP, BTI, WF, CPI, CIRE
$\mathrm{CS}_{\mathrm{ci}}$	coinitial Causal Safety	Def. 4.27	Thm. 4.28	SP, BTI, WF, CPI
$\mathrm{CL}_{\mathrm{ci}}$	coinitial Causal Liveness	Def. 4.27	Thm. 4.30	SP, BTI, WF, CPI, CIRE
NRE	No Repeated Events	Def. 4.35	Prop. 4.42	SP, BTI, WF, CPI, CIRE
RED	Reverse Event Determinism	Def. 4.40	Prop. 4.41	SP, BTI, WF, CPI, NRE

Contents

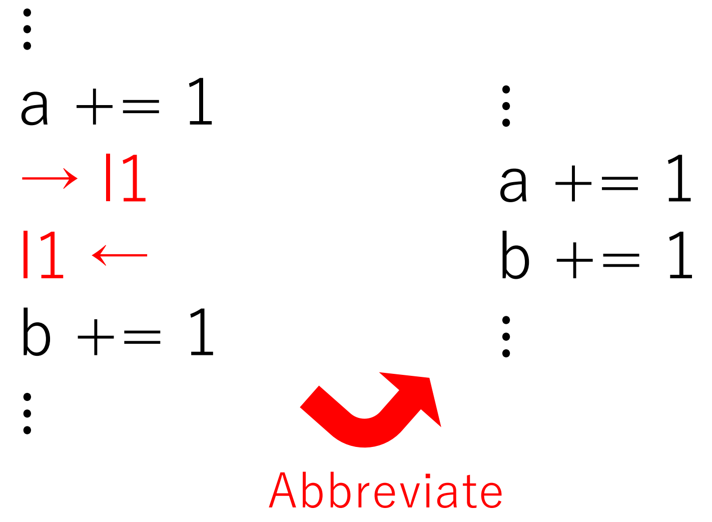
- Background & Motivation
- CRIL: Concurrent Reversible Intermediate Language
- The controlled semantics of CRIL
- Reversibility Properties
- Bidirectional Data Flow Analysis in CRIL
- Constant Propagation in CRIL
- Concluding Remarks

Bidirectional Data Flow Analysis

- Calculate backward data flow using the forward data flow.


Bidrectinal Data Flow Analysis
CRIL Program
Forward Data Flow Analysis
Backward Data Flow Analysis
Forward Data Flow Edge
Backward Data Flow Edge

Example for Data Flow Analysis



```
⋮
a += 1
→ l1
l1 ←
b += 1
⋮
```

Abbreviate

```
⋮
a += 1
b += 1
⋮
```

- sub1 and sub2 manipulate a and b.

```
begin main
a += 1
b += 1
call sub1, sub2
b -= 3
a -= 4
end main
```

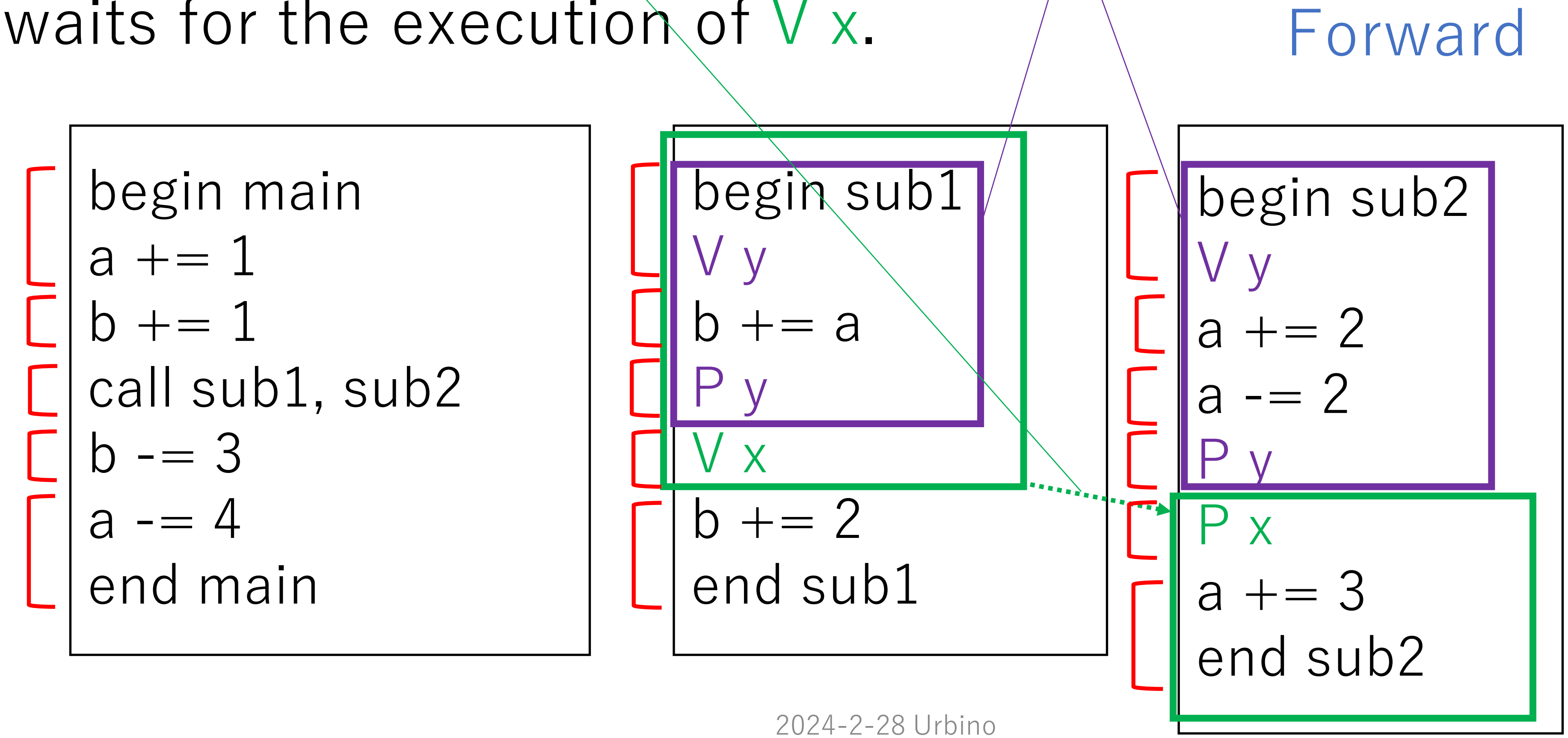
```
begin sub1
V y
b += a
P y
V x
b += 2
end sub1
```

```
begin sub2
V y
a += 2
a -= 2
P y
P x
a += 3
end sub2
```

Control by P-V operations

Note that x and y start from and end to 0

- V y and P y makes two critical regions.
- The ordering relationship occurs since P x waits for the execution of V x.



Forward

```
begin main
a += 1
b += 1
call sub1, sub2
b -= 3
a -= 4
end main
```

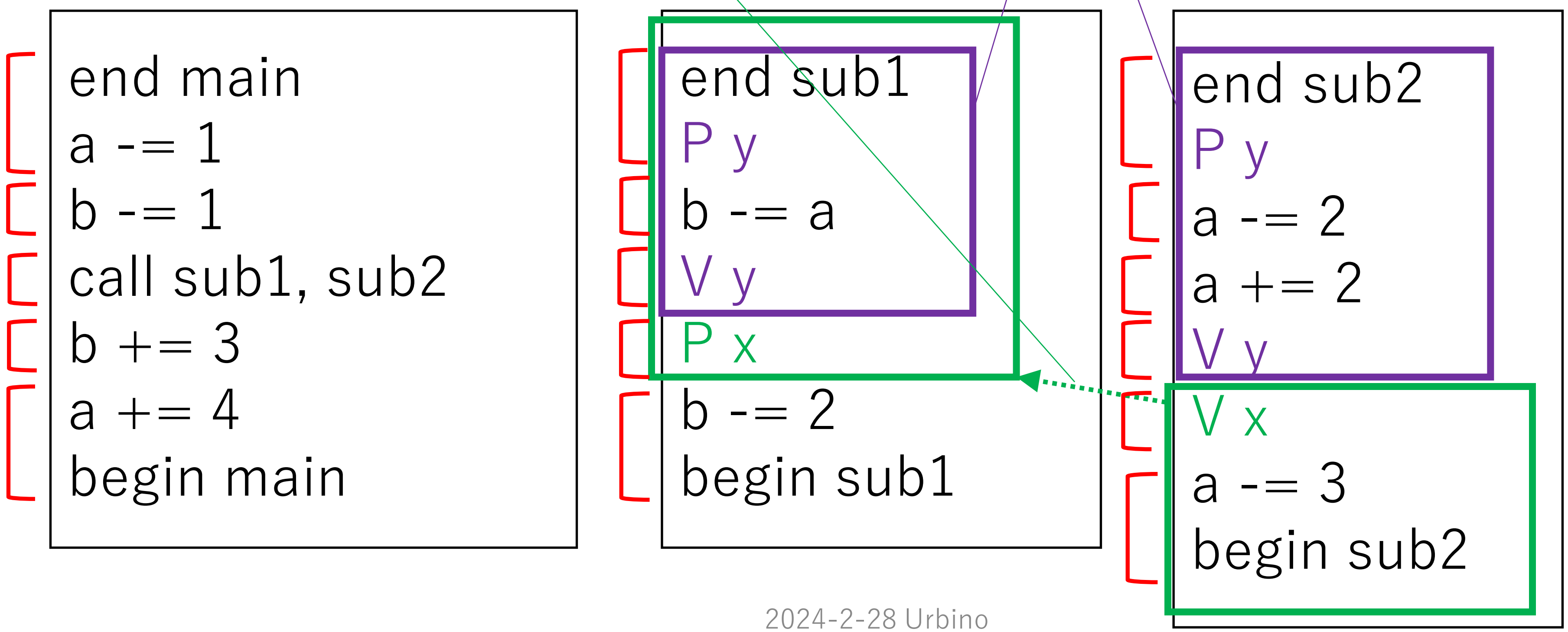
```
begin sub1
V y
b += a
P y
V x
b += 2
end sub1
```

```
begin sub2
V y
a += 2
a -= 2
P y
P x
a += 3
end sub2
```

P-V operations in Backward Direction

- V y and P y makes two critical regions.
- The ordering relationship occurs since P x waits for the execution of V x.

Backward



```
end main
a -= 1
b -= 1
call sub1, sub2
b += 3
a += 4
begin main
```

```
end sub1
P y
b -= a
V y
P x
b -= 2
begin sub1
```

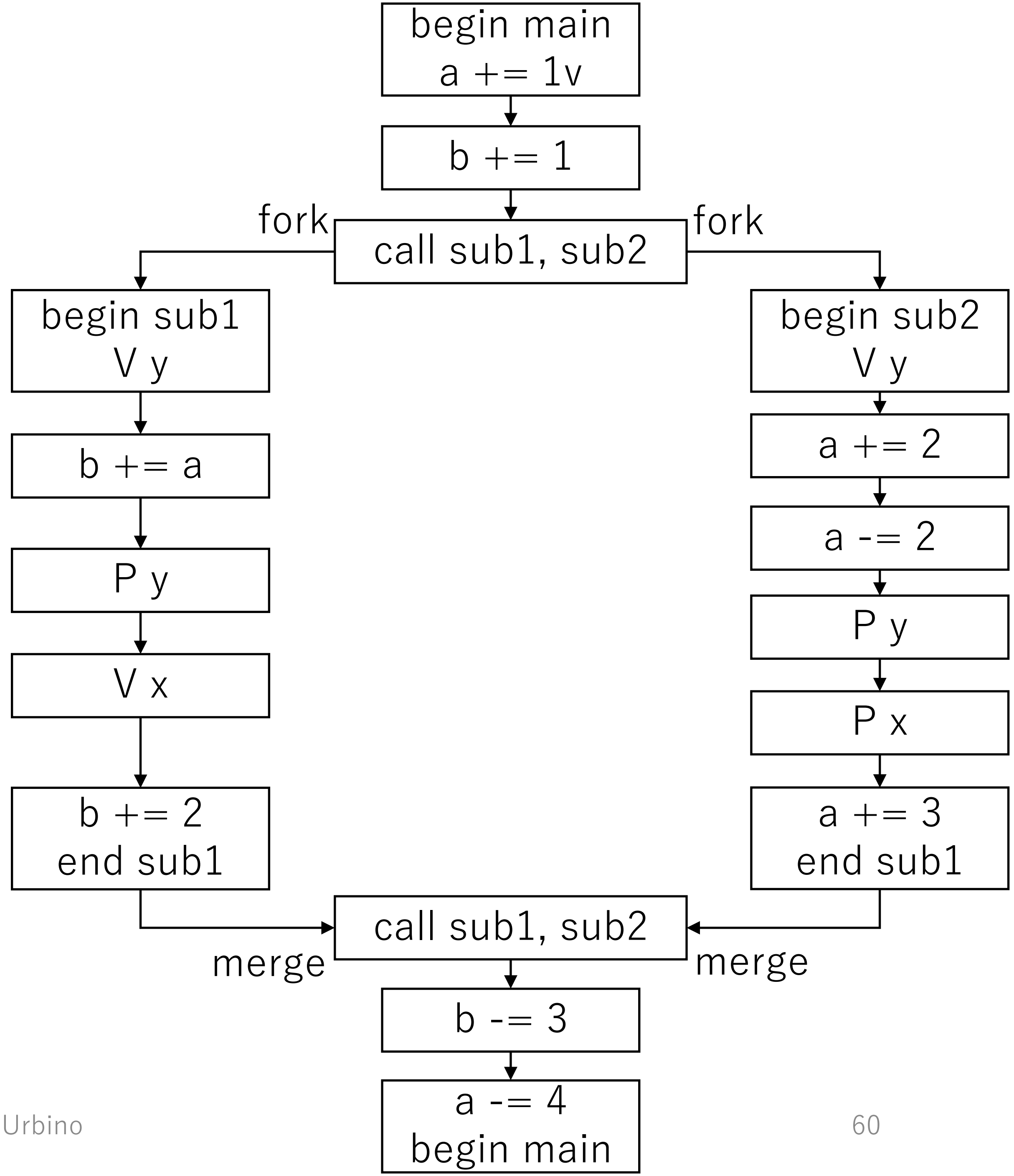
```
end sub2
P y
a -= 2
a += 2
V y
V x
a -= 3
begin sub2
```

Control Flow Graph

```
begin main
a += 1
b += 1
call sub1, sub2
b -= 4
a -= 4
end main
```

```
begin sub1
V y
b += a + 1
P y
V x
b += 2
end sub1
```

```
begin sub2
V y
a += 2
a -= 2
P y
P x
a += 3
end sub2
```



begin main
a += 1v

b += 1

fork

call sub1, sub2

fork

begin sub1
V y

b += a

P y

V x

b += 2
end sub1

begin sub2
V y

a += 2

a -= 2

P y

P x

a += 3
end sub1

call sub1, sub2

merge

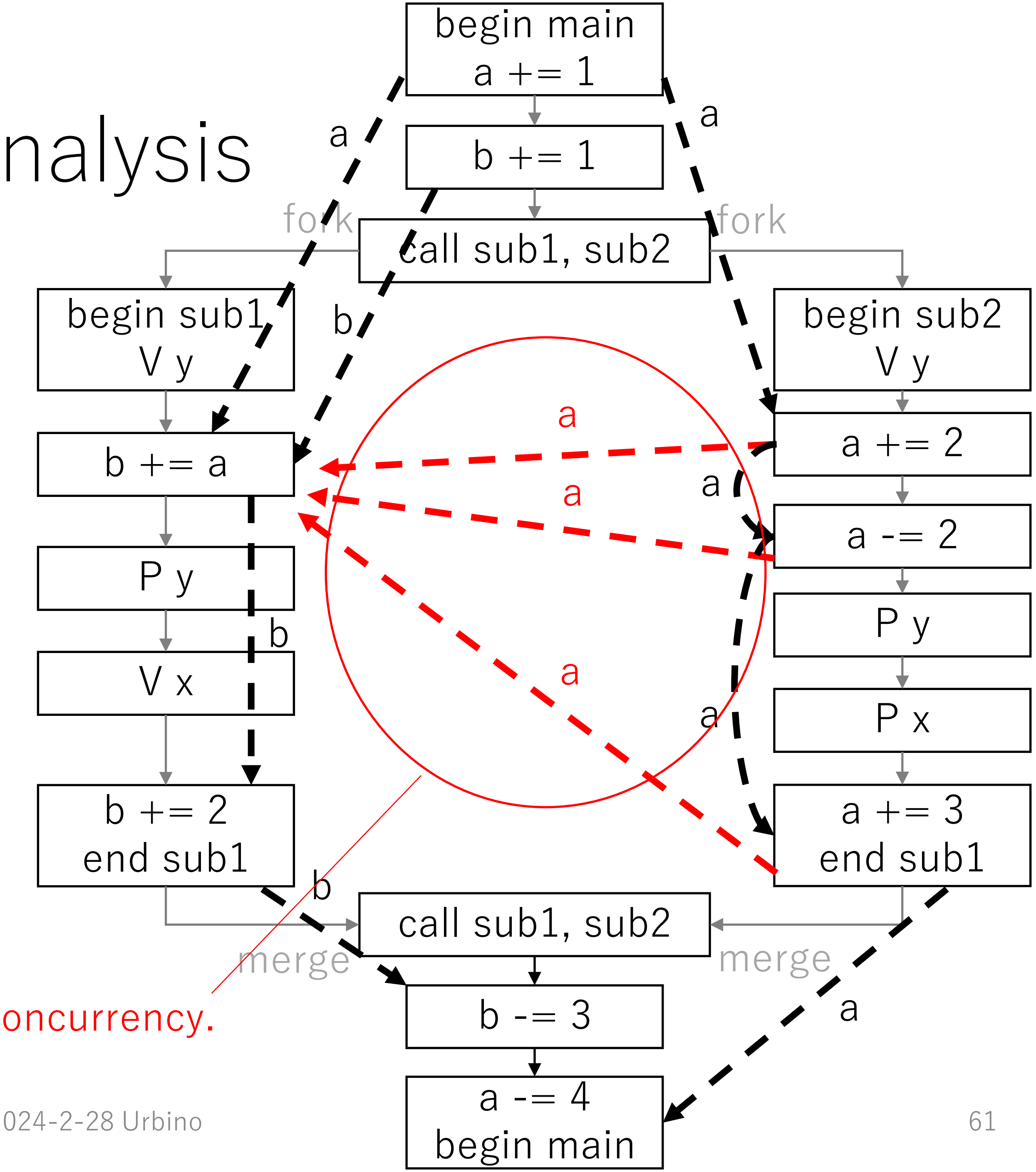
merge

b -= 3

a -= 4
begin main

Forward Data Flow Analysis

- We represents data flow by **edges labeled variables** among basic blocks.
- We apply use-def analysis like forward-only sequential program.

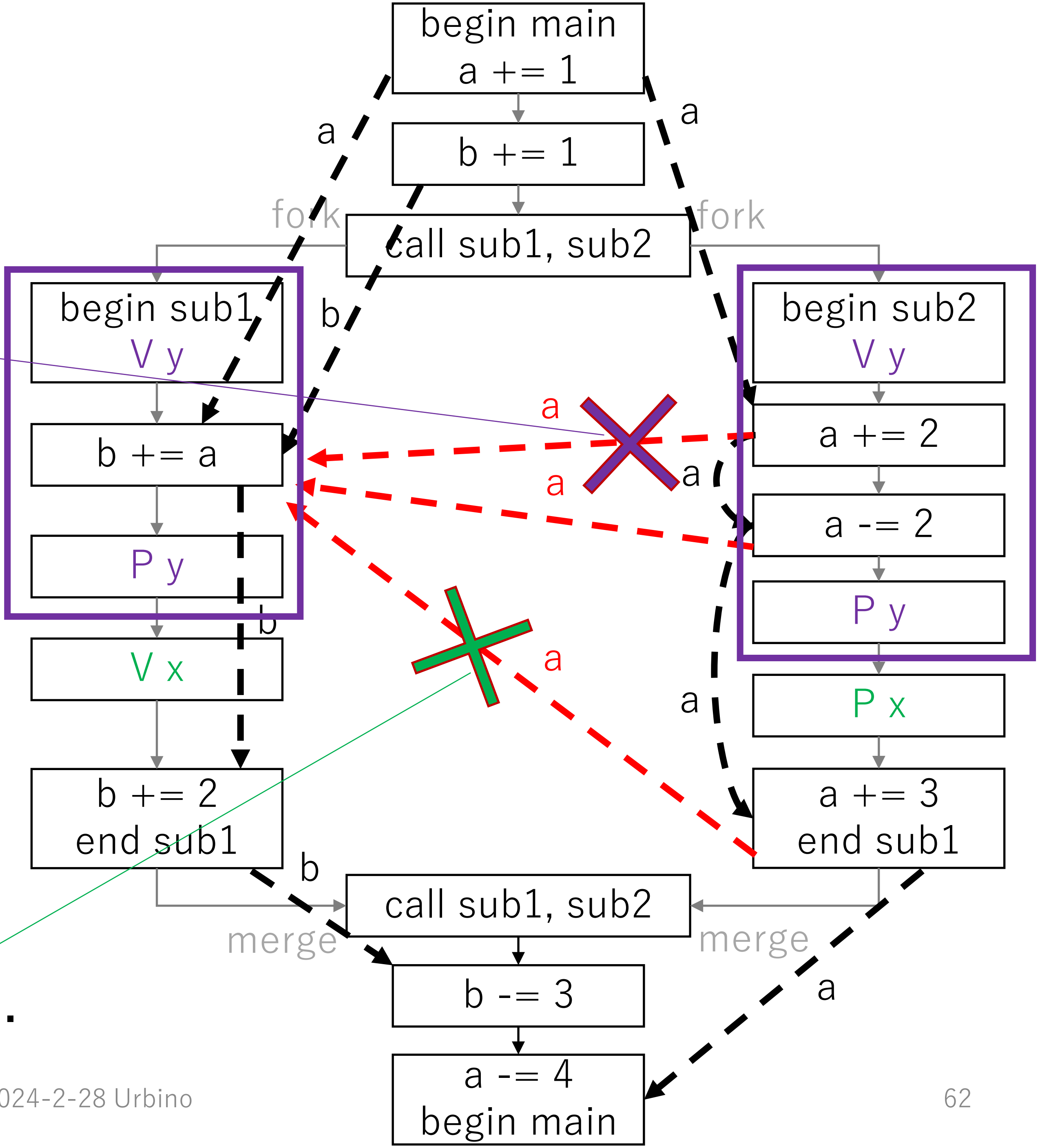


Generated by concurrency.

Edge Elimination

[Critical reagion analysis]

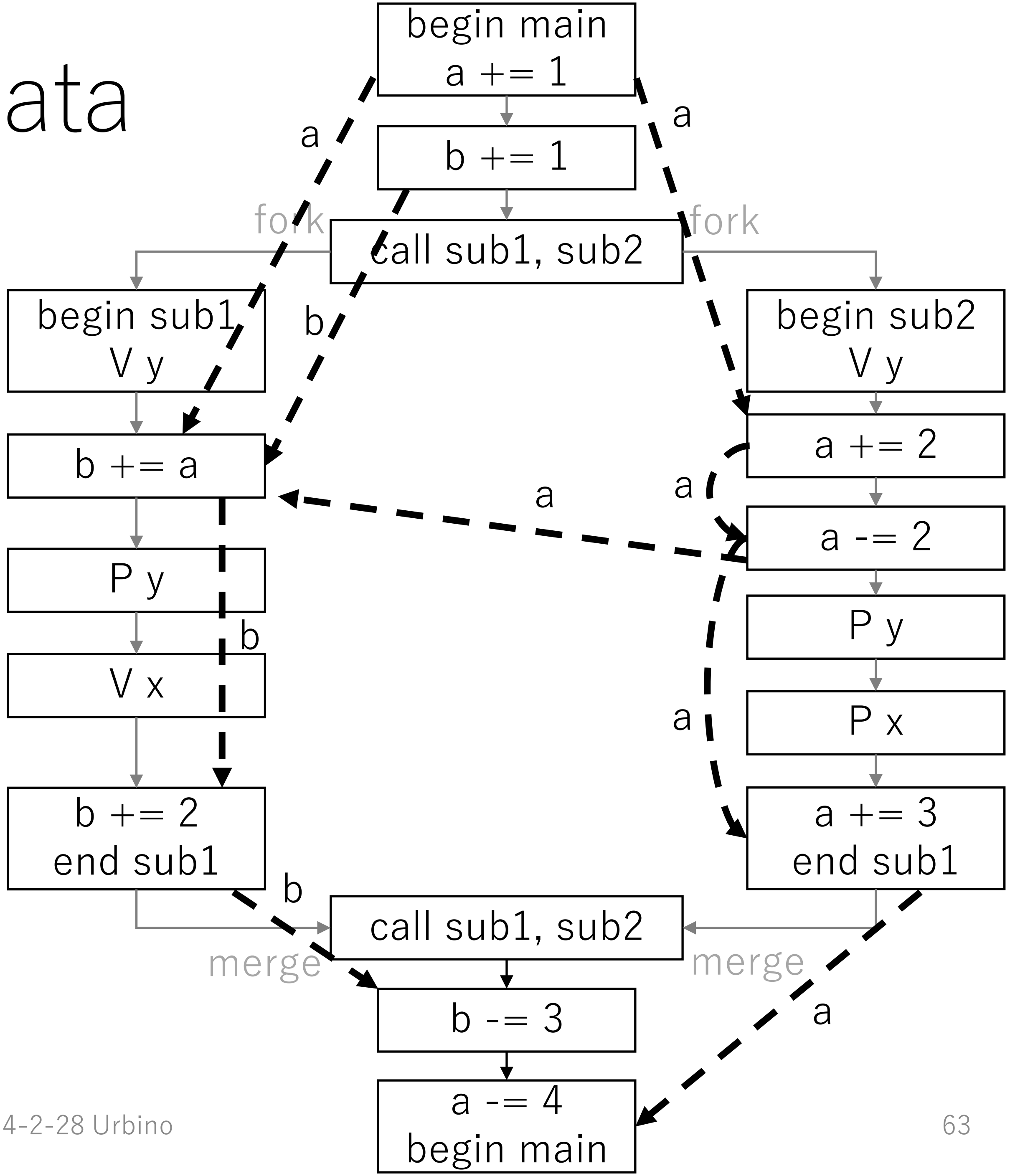
The definition of'a-=2' is killed by 'a-=2'.



[Synchronization analysis]

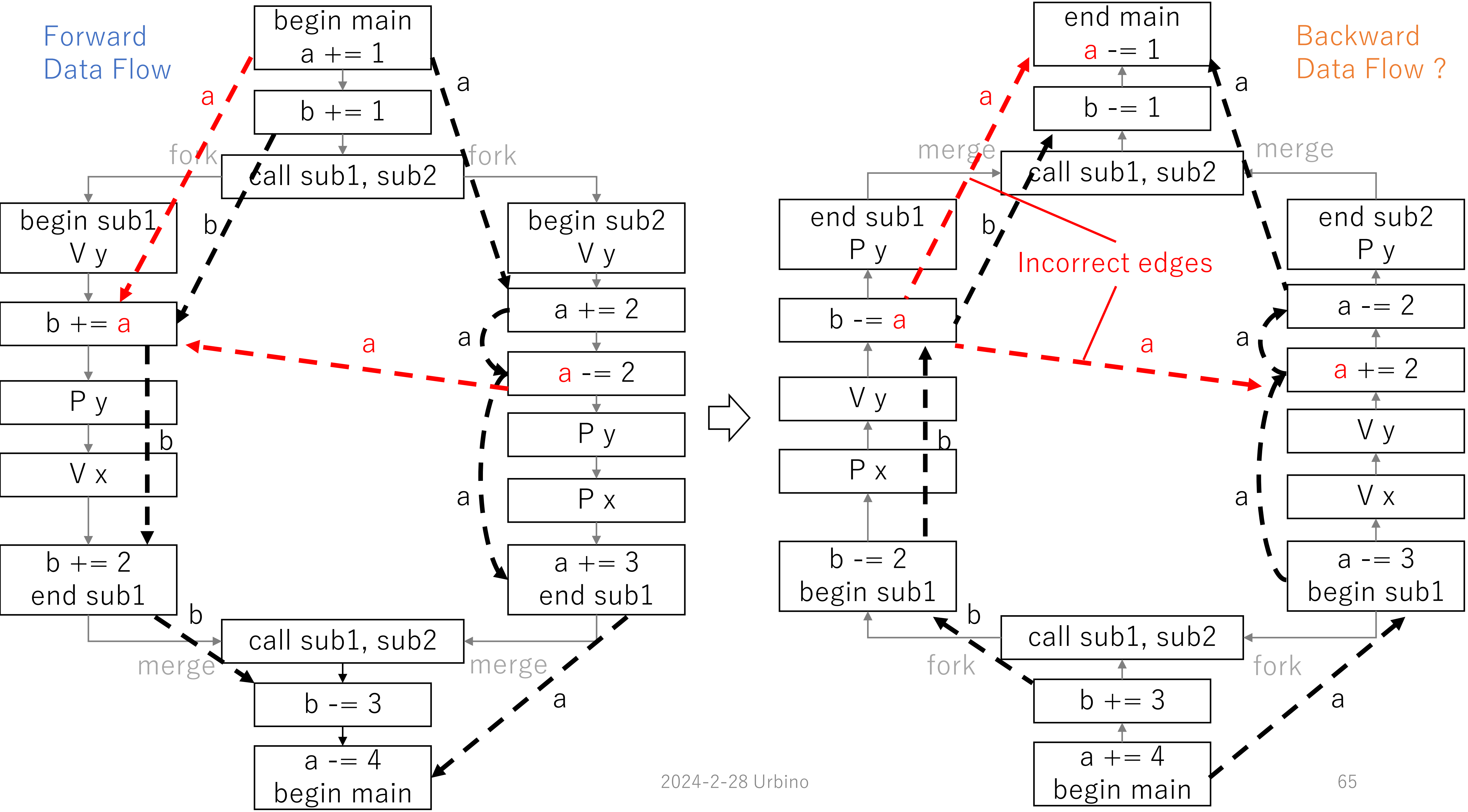
'a+=3' is executed after 'b+=a'.

Result of Forward Data Flow Analysis



Backward Data Flow Analysis

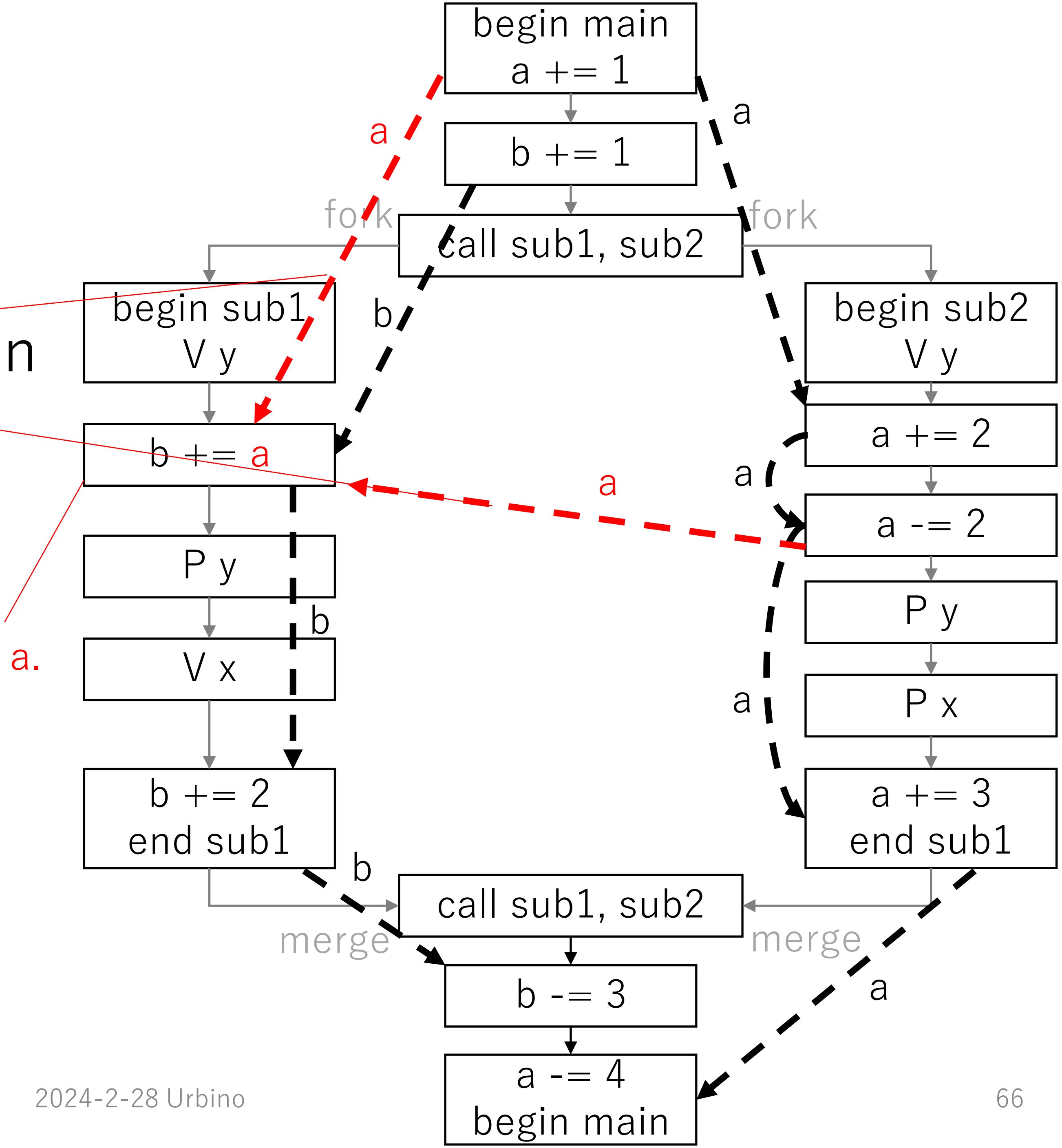
- Calculate backward data flow by **transforming** the result of forward data flow analysis.
- **Simply reversing edges** does **not** yield backward data flow.


Forward
Data Flow
begin main
a += 1
b += 1
call sub1, sub2
fork
fork
begin sub1
V y
b += a
P y
V x
b += 2
end sub1
begin sub2
V y
a += 2
a -= 2
P y
P x
a += 3
end sub1
call sub1, sub2
merge
merge
b -= 3
a -= 4
begin main
a
a
b
b
b
a
a
a
a
Backward
Data Flow ?
end main
a -= 1
b -= 1
call sub1, sub2
merge
merge
end sub1
P y
b -= a
V y
P x
b -= 2
begin sub1
end sub2
P y
a -= 2
a += 2
V y
V x
a -= 3
begin sub1
call sub1, sub2
fork
fork
b += 3
a += 4
begin main
Incorrect edges
a
a
b
b
b
a
a
a
a

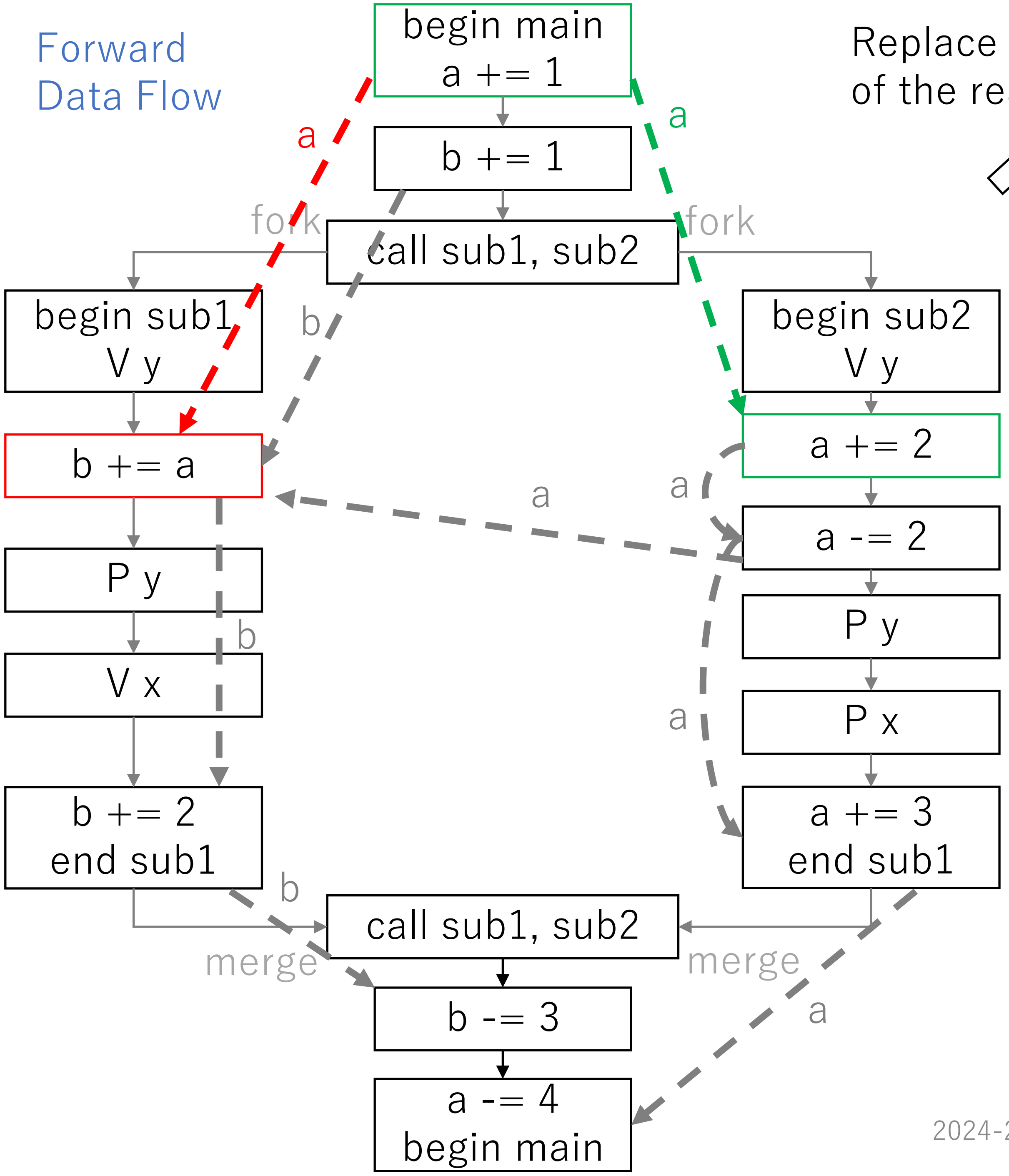
Read Only Edge

- A read only edge refers to an edge where the destination does not write to variables labeled.

- We **replace** the **source** of read only edges.



Forward
Data Flow



begin main
a += 1

b += 1

call sub1, sub2

fork

fork

begin sub1
V y

b += a

P y

V x

b += 2
end sub1

begin sub2
V y

a += 2

a -= 2

P y

P x

a += 3
end sub1

merge

merge

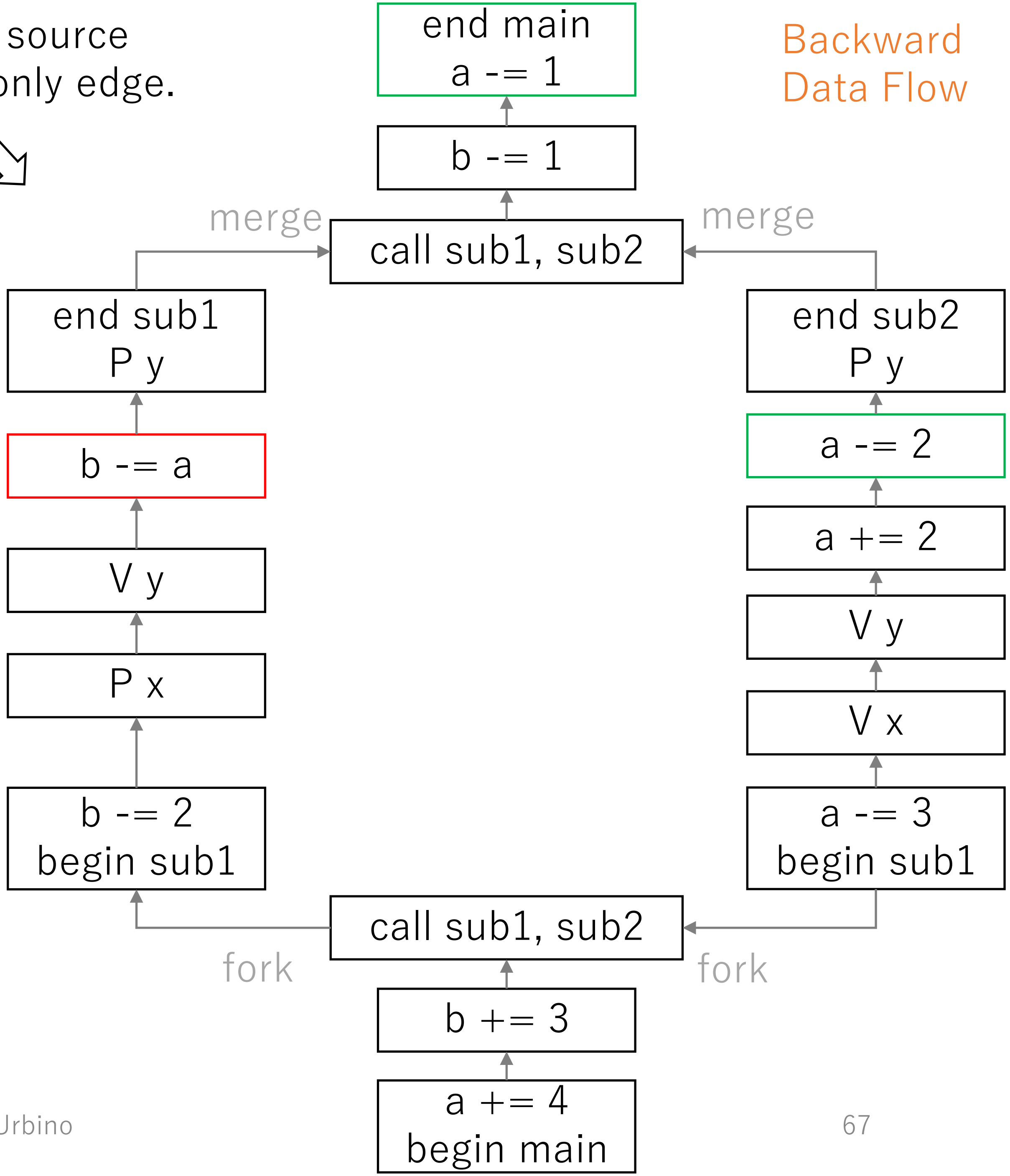
call sub1, sub2

b -= 3

a -= 4
begin main

Replace the source
of the read only edge.

Backward
Data Flow



end main
a -= 1

b -= 1

merge

merge

call sub1, sub2

end sub1
P y

b -= a

V y

P x

b -= 2
begin sub1

end sub2
P y

a -= 2

a += 2

V y

V x

a -= 3
begin sub1

call sub1, sub2

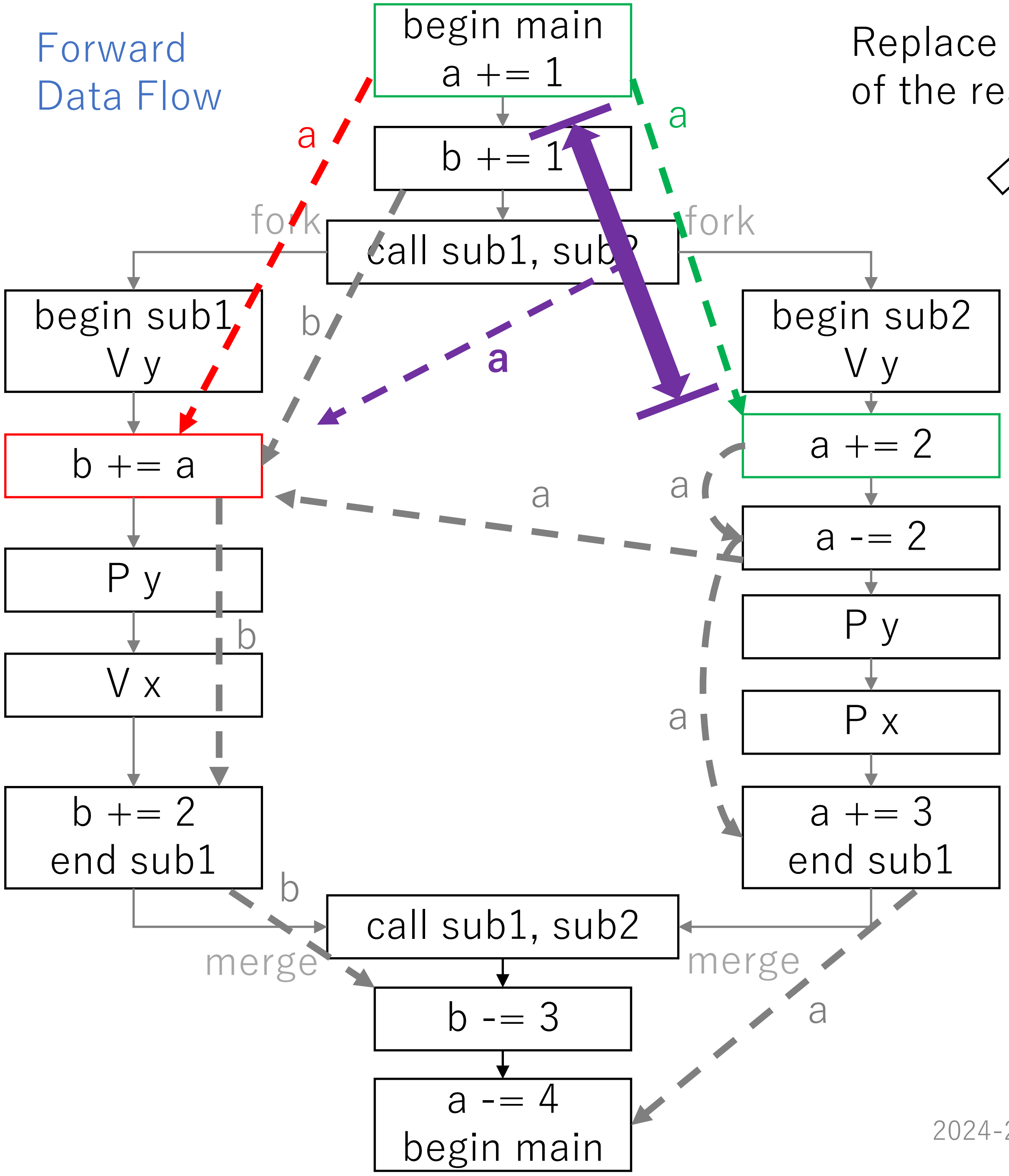
fork

fork

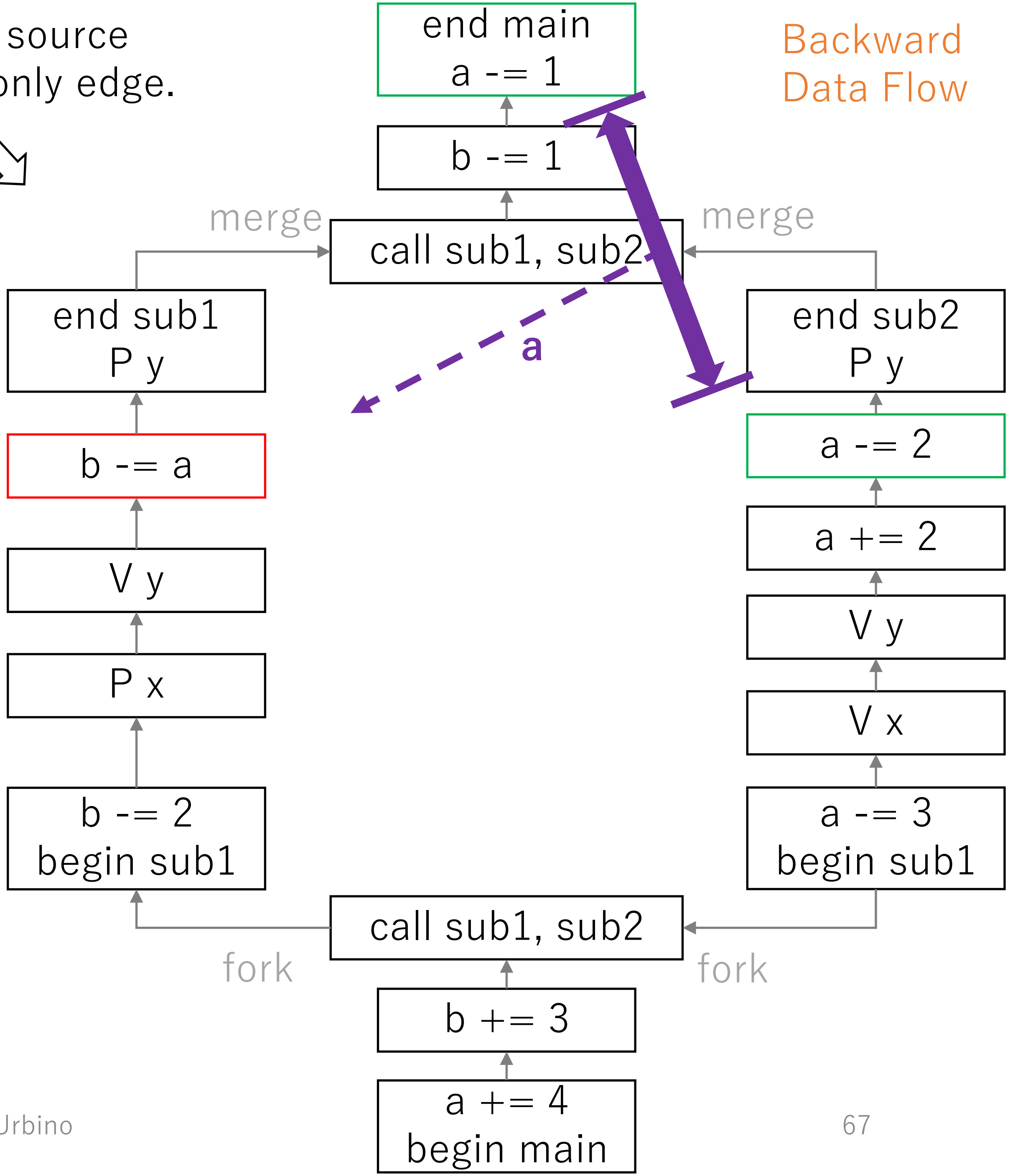
b += 3

a += 4
begin main

Forward
Data Flow

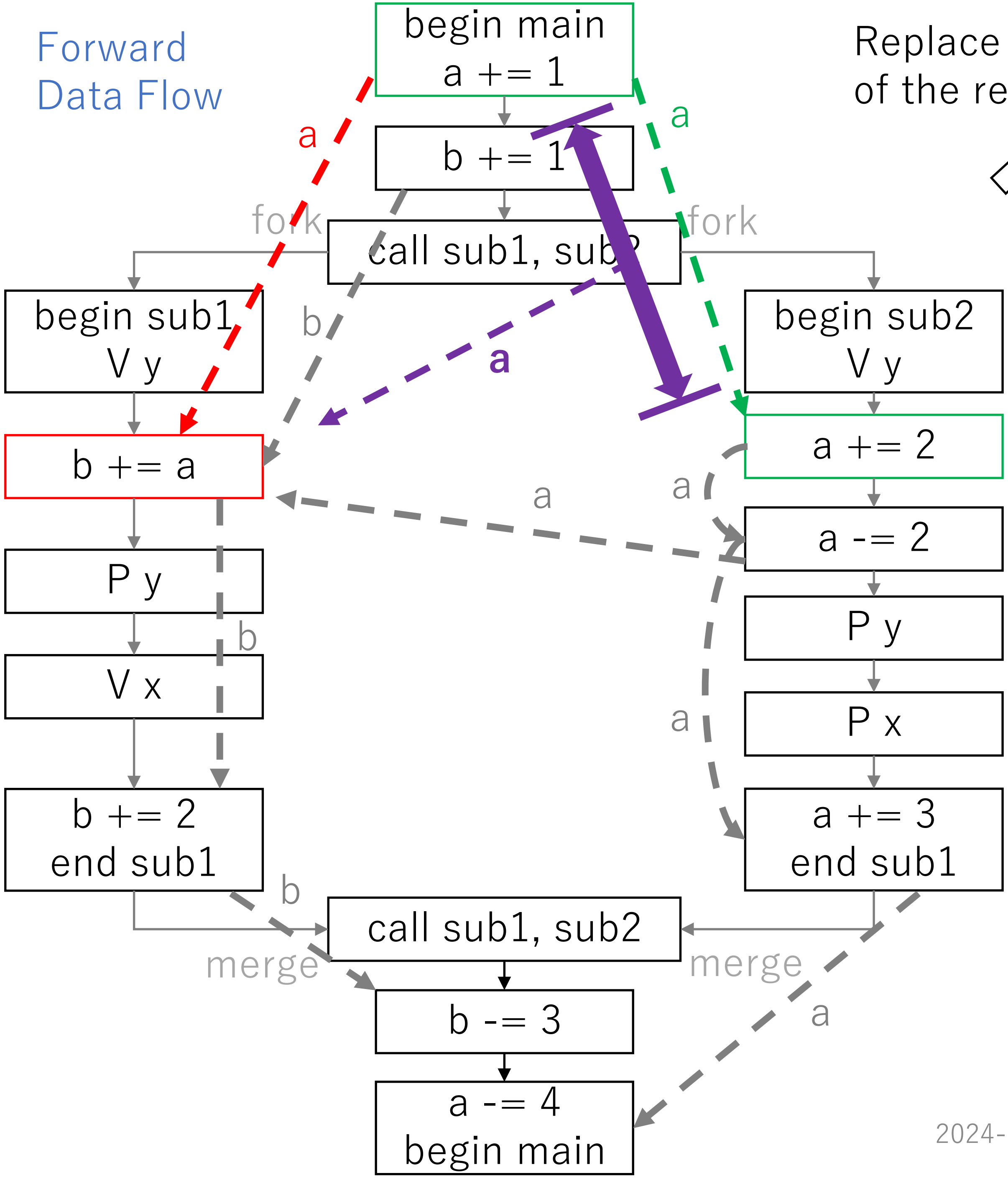


Replace the source
of the read only edge.



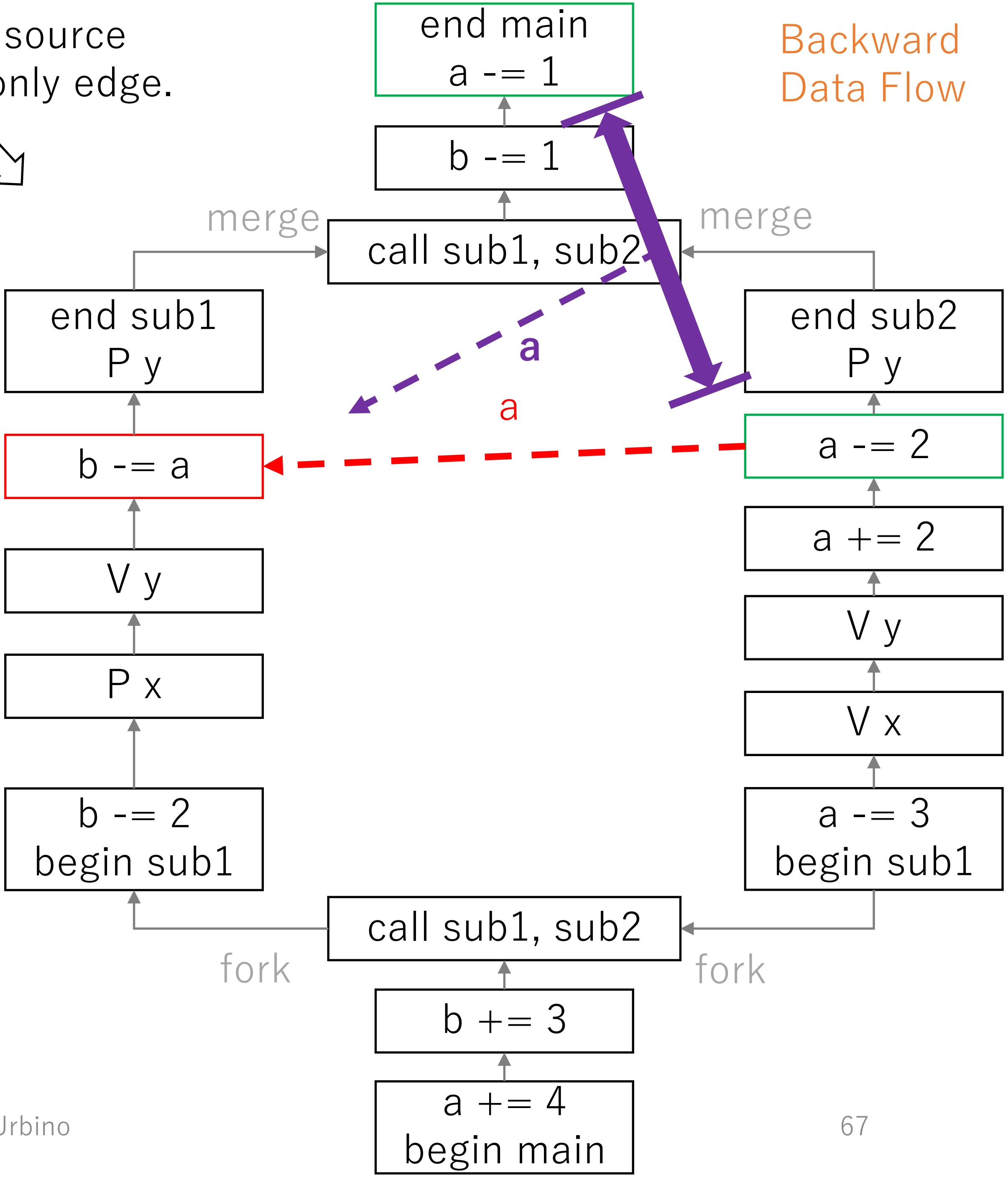
Backward
Data Flow

Forward Data Flow



Replace the source of the read only edge.

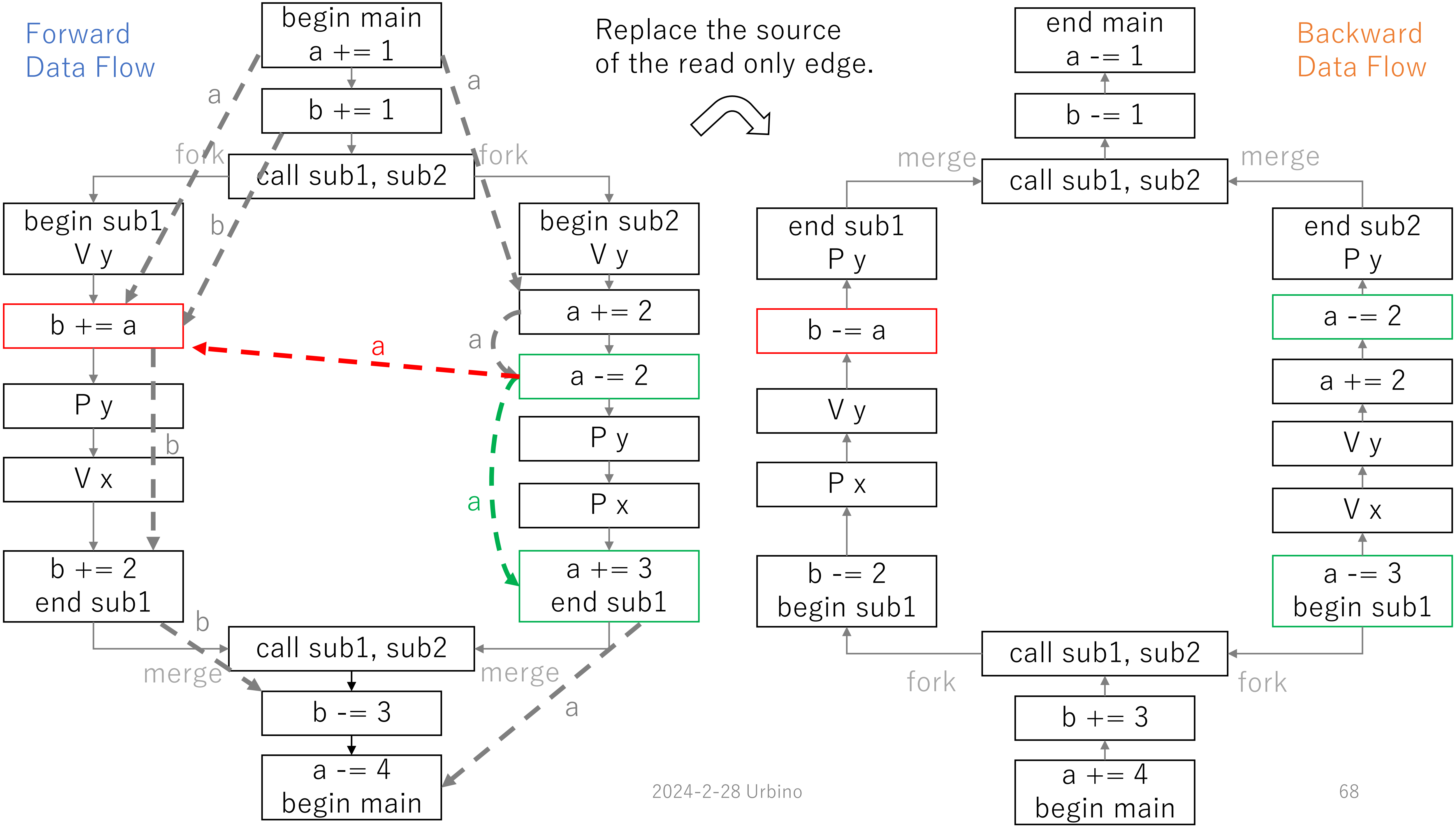
Backward Data Flow



Forward
Data Flow

Replace the source
of the read only edge.

Backward
Data Flow



begin main
a += 1

b += 1

call sub1, sub2

begin sub1
V y

b += a

P y

V x

b += 2
end sub1

begin sub2
V y

a += 2

a -= 2

P y

P x

a += 3
end sub1

call sub1, sub2

b -= 3

a -= 4
begin main

end main
a -= 1

b -= 1

call sub1, sub2

end sub1
P y

b -= a

V y

P x

b -= 2
begin sub1

end sub2
P y

a -= 2

a += 2

V y

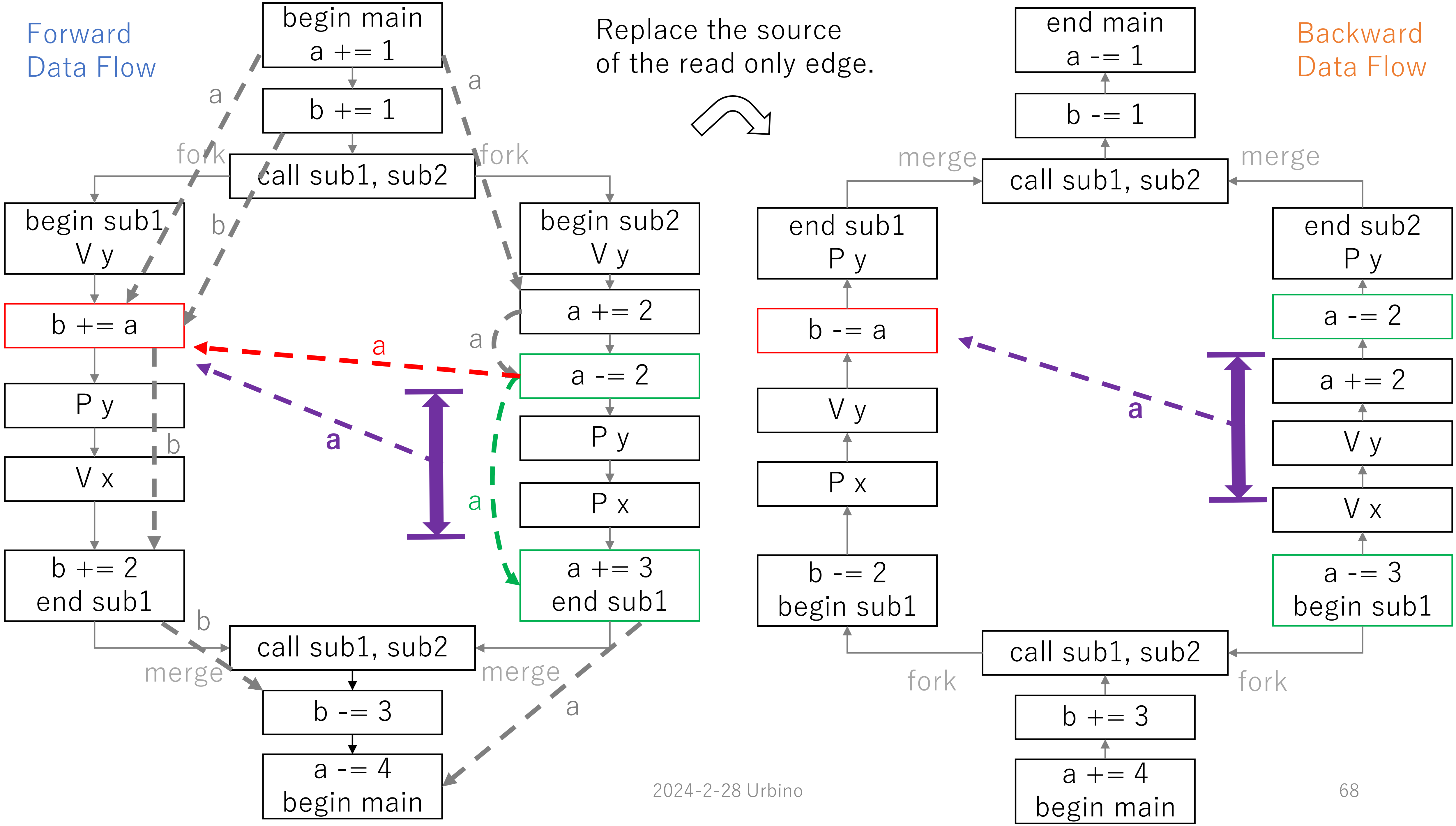
V x

a -= 3
begin sub1

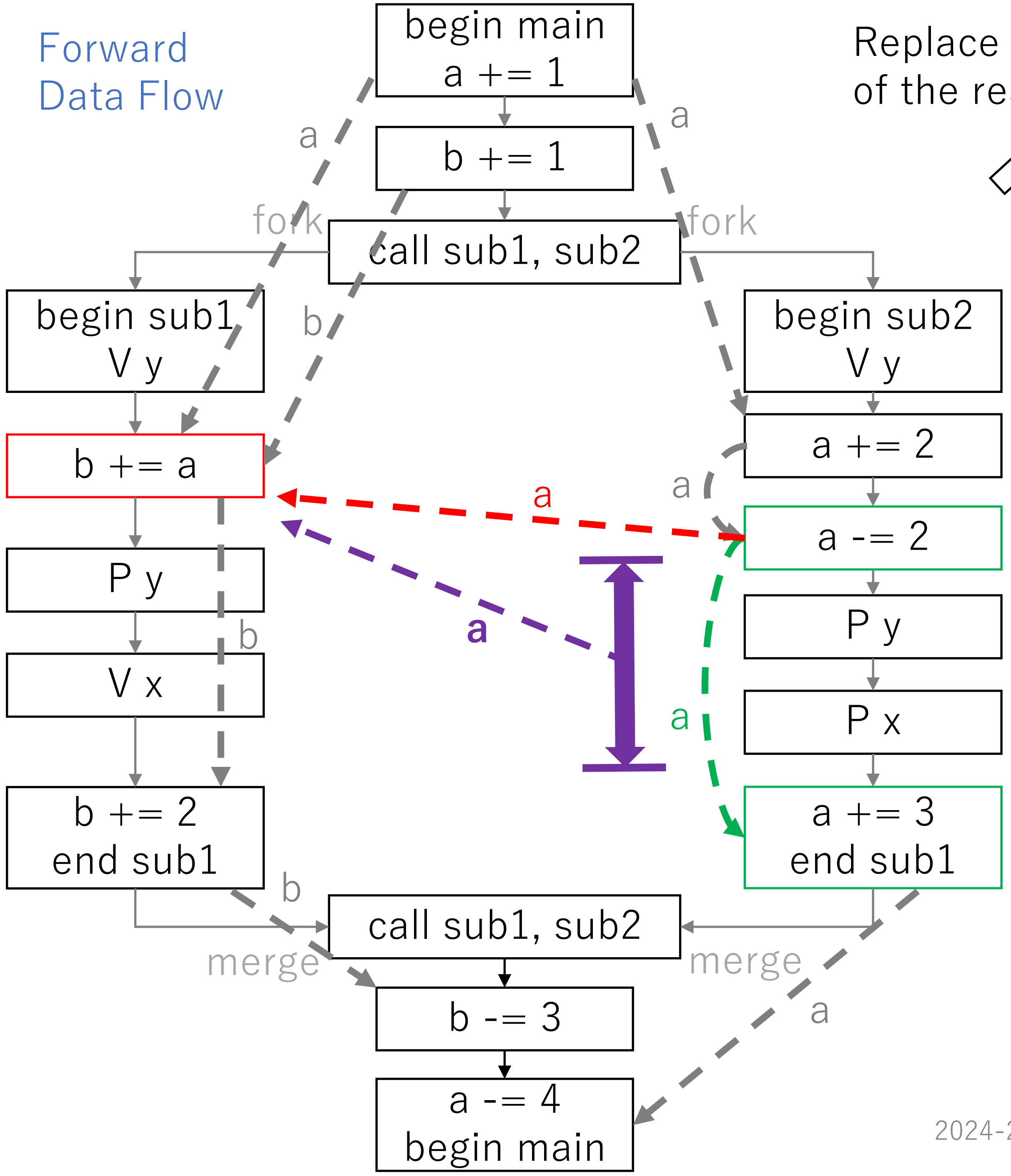
call sub1, sub2

b += 3

a += 4
begin main


Forward
Data Flow
begin main
a += 1
b += 1
call sub1, sub2
fork
fork
a
a
b
begin sub1
V y
begin sub2
V y
b += a
a += 2
a -= 2
a
a
a
P y
P y
b
V x
P x
a
b += 2
end sub1
a += 3
end sub1
b
call sub1, sub2
merge
merge
b -= 3
a
a -= 4
begin main
Replace the source
of the read only edge.
end main
a -= 1
b -= 1
merge
merge
call sub1, sub2
end sub1
P y
end sub2
P y
b -= a
a -= 2
a
a += 2
V y
V y
P x
V x
b -= 2
begin sub1
a -= 3
begin sub1
call sub1, sub2
fork
fork
b += 3
a += 4
begin main
Backward
Data Flow

Forward
Data Flow



begin main
a += 1

b += 1

call sub1, sub2

begin sub1
V y

b += a

P y

V x

b += 2
end sub1

begin sub2
V y

a += 2

a -= 2

P y

P x

a += 3
end sub1

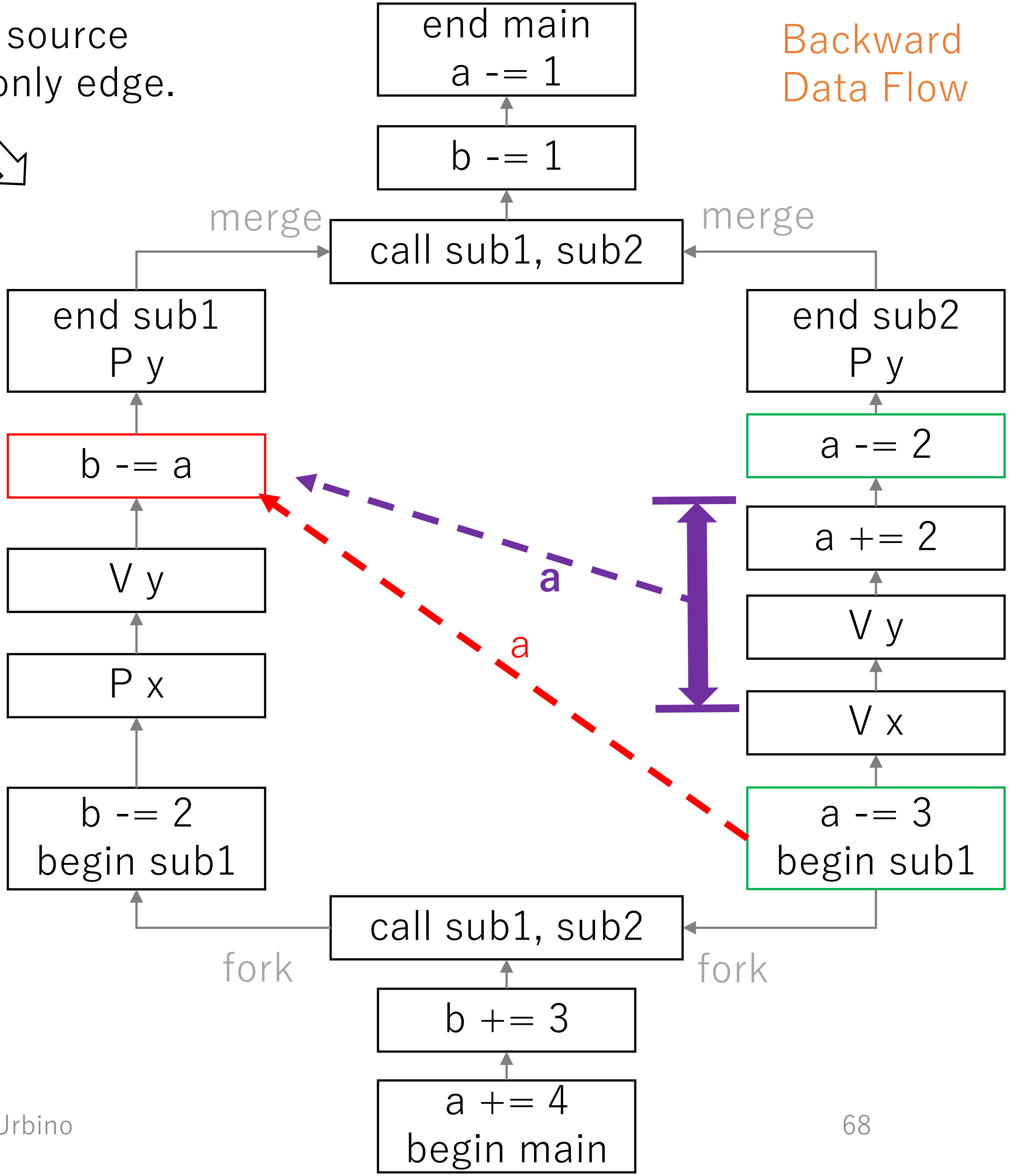
call sub1, sub2

b -= 3

a -= 4
begin main

Replace the source
of the read only edge.

Backward
Data Flow



end main
a -= 1

b -= 1

call sub1, sub2

end sub1
P y

b -= a

V y

P x

b -= 2
begin sub1

end sub2
P y

a -= 2

a += 2

V y

V x

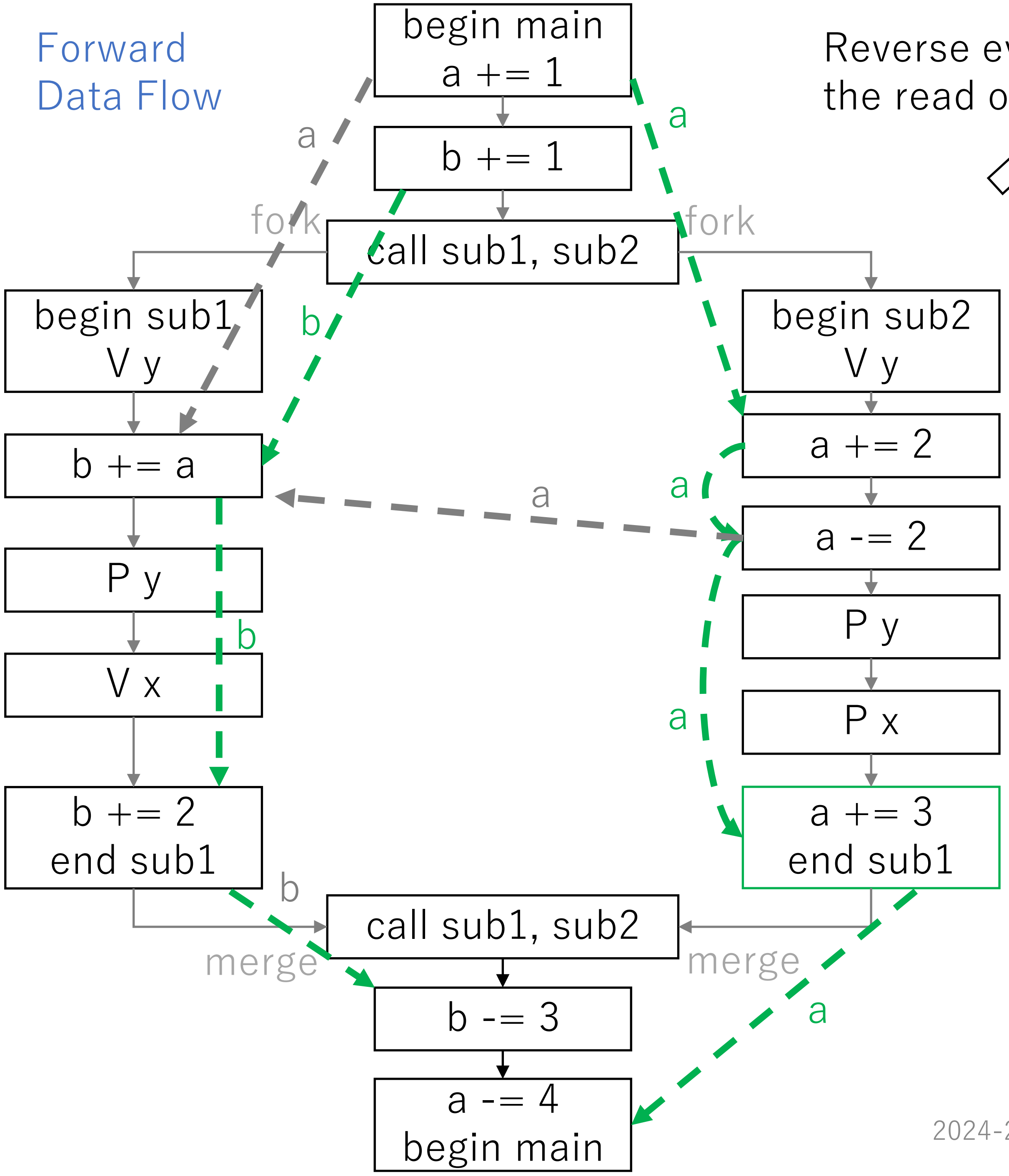
a -= 3
begin sub1

call sub1, sub2

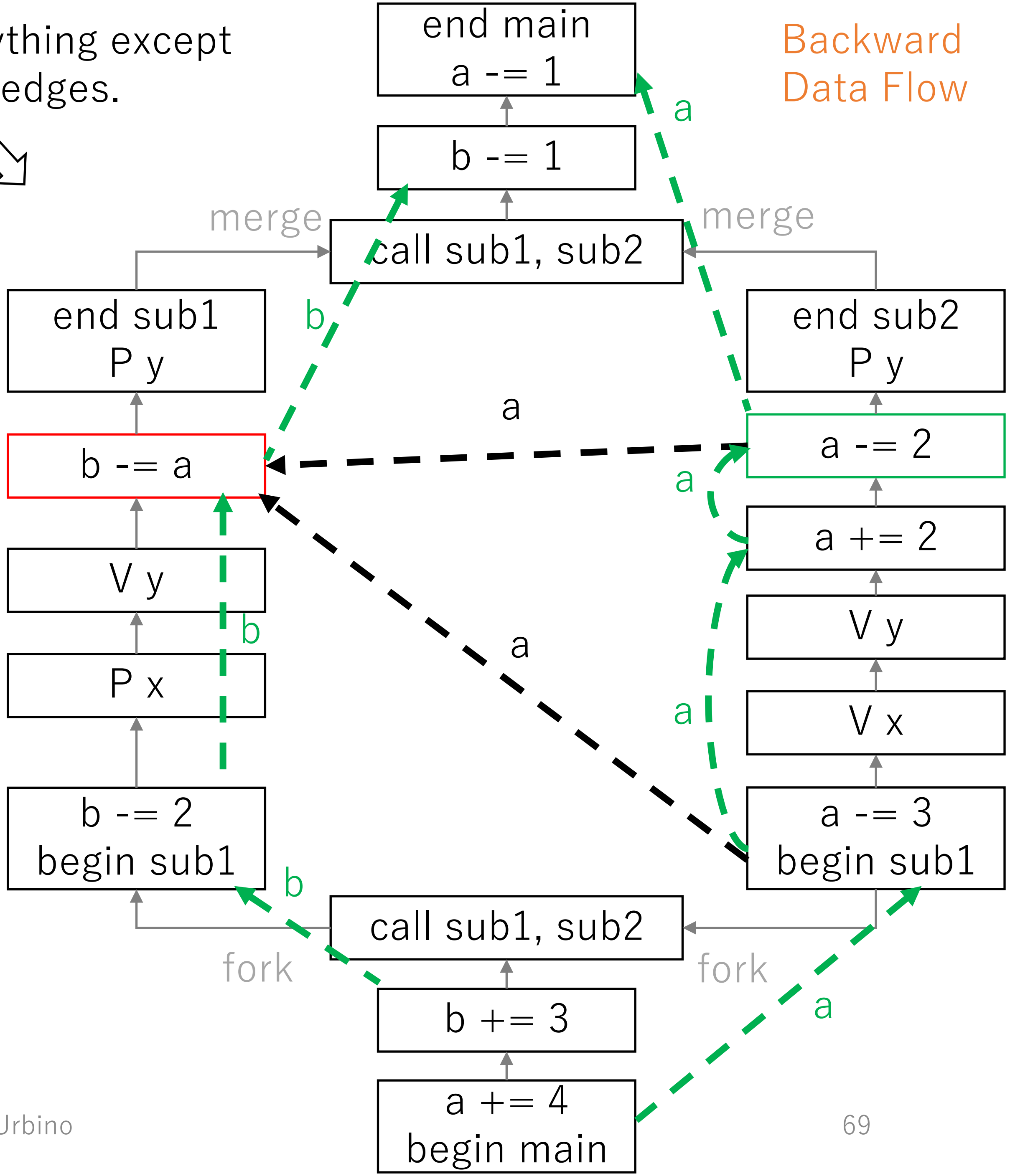
b += 3

a += 4
begin main

Forward
Data Flow

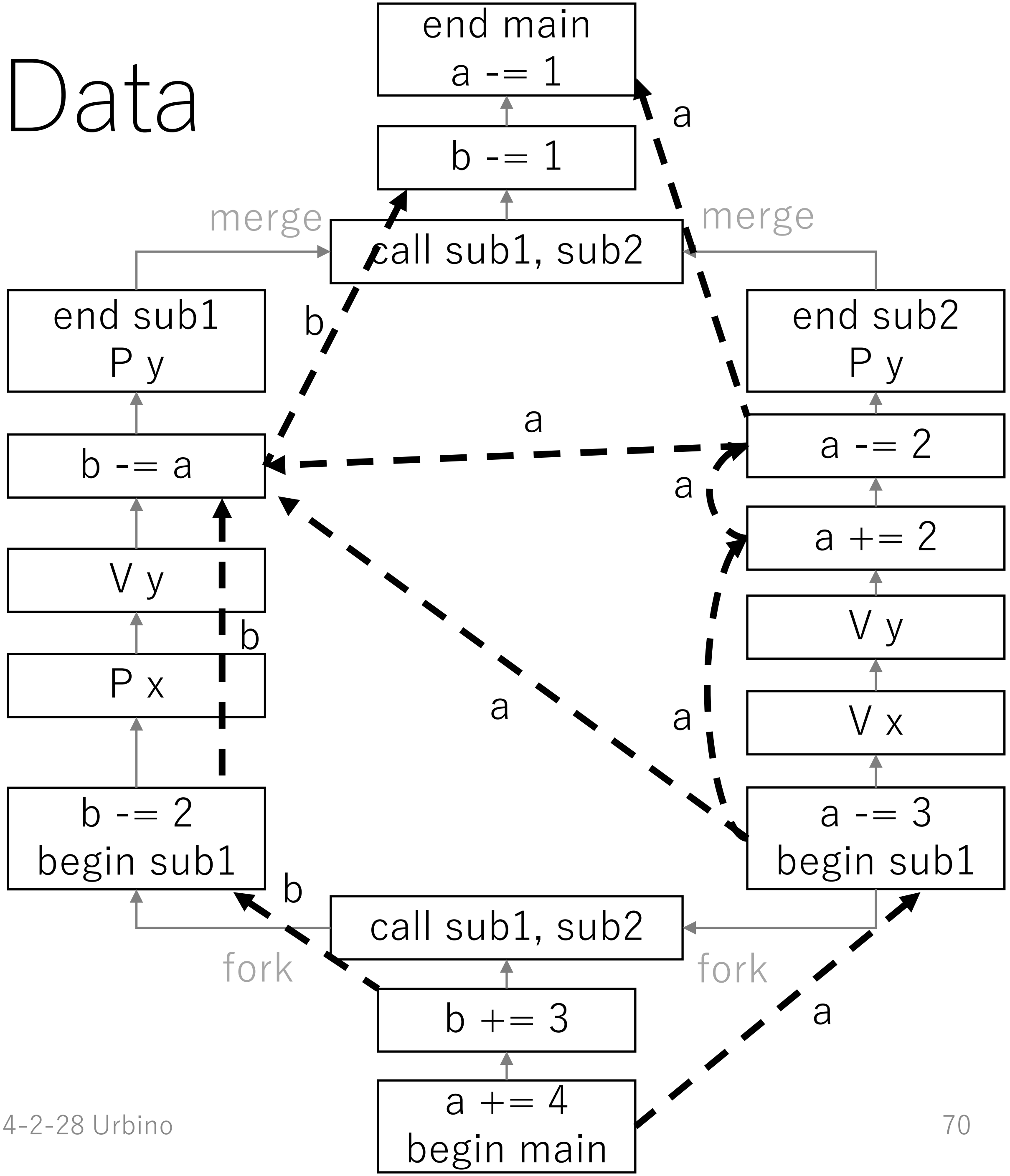


Reverse everything except
the read only edges.

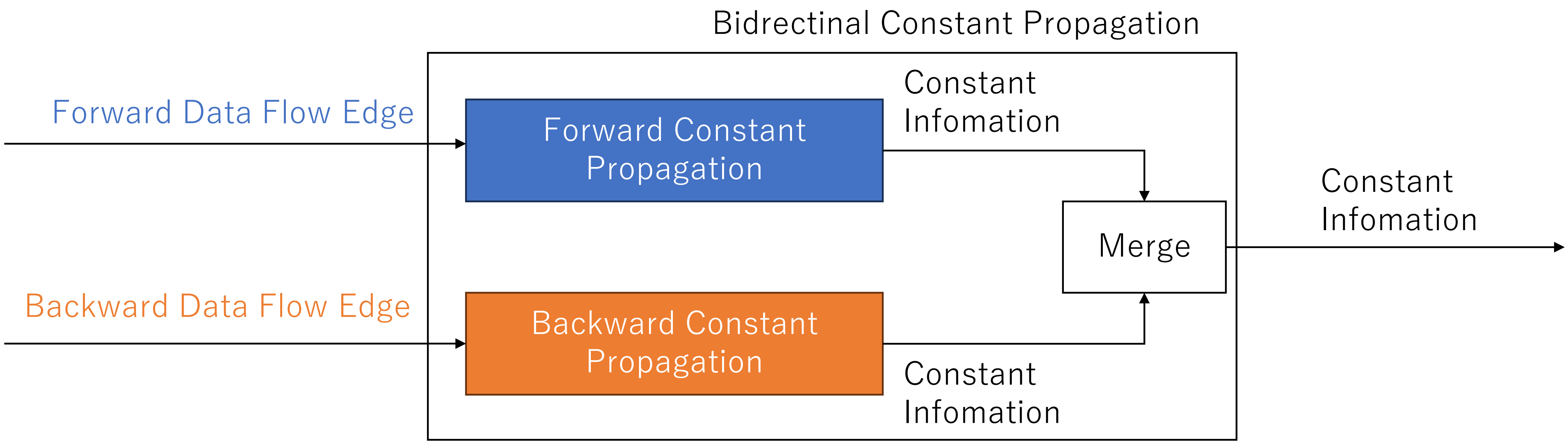


Backward
Data Flow

Result of Backward Data Flow Analysis



Constant Propagation in CRIL



Bidrectinal Constant Propagation

Forward Data Flow Edge

Forward Constant Propagation

Constant Infomation

Backward Data Flow Edge

Backward Constant Propagation

Constant Infomation

Merge

Constant Infomation

Example for Constant Propagation

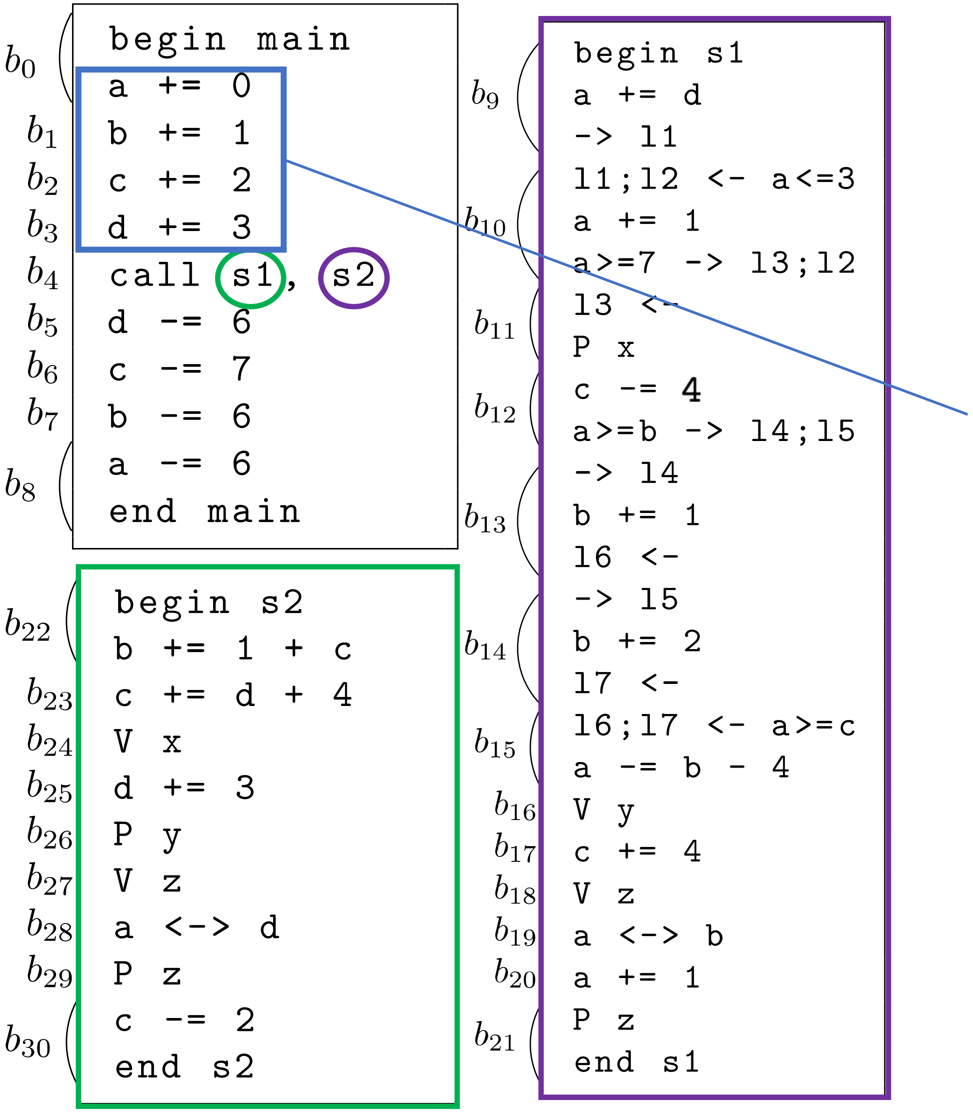
```
b0   begin main
     a += 0
b1   b += 1
b2   c += 2
b3   d += 3
b4   call s1, s2
b5   d -= 6
b6   c -= 7
b7   b -= 6
b8   a -= 6
     end main
```

```
b22  begin s2
     b += 1 + c
b23  c += d + 4
b24  V x
b25  d += 3
b26  P y
b27  V z
b28  a <-> d
b29  P z
b30  c -= 2
     end s2
```

```
b9   begin s1
     a += d
     -> l1
b10  l1;l2 <- a<=3
     a += 1
     a>=7 -> l3;l2
b11  l3 <-
     P x
b12  c -= 4
     a>=b -> l4;l5
b13  -> l4
     b += 1
     l6 <-
b14  -> l5
     b += 2
     l7 <-
b15  l6;l7 <- a>=c
     a -= b - 4
b16  V y
b17  c += 4
b18  V z
b19  a <-> b
b20  a += 1
b21  P z
     end s1
```

- We assume all the values of variables starts from and ends to 0 (common in reversible programming).

Example for Constant Propagation



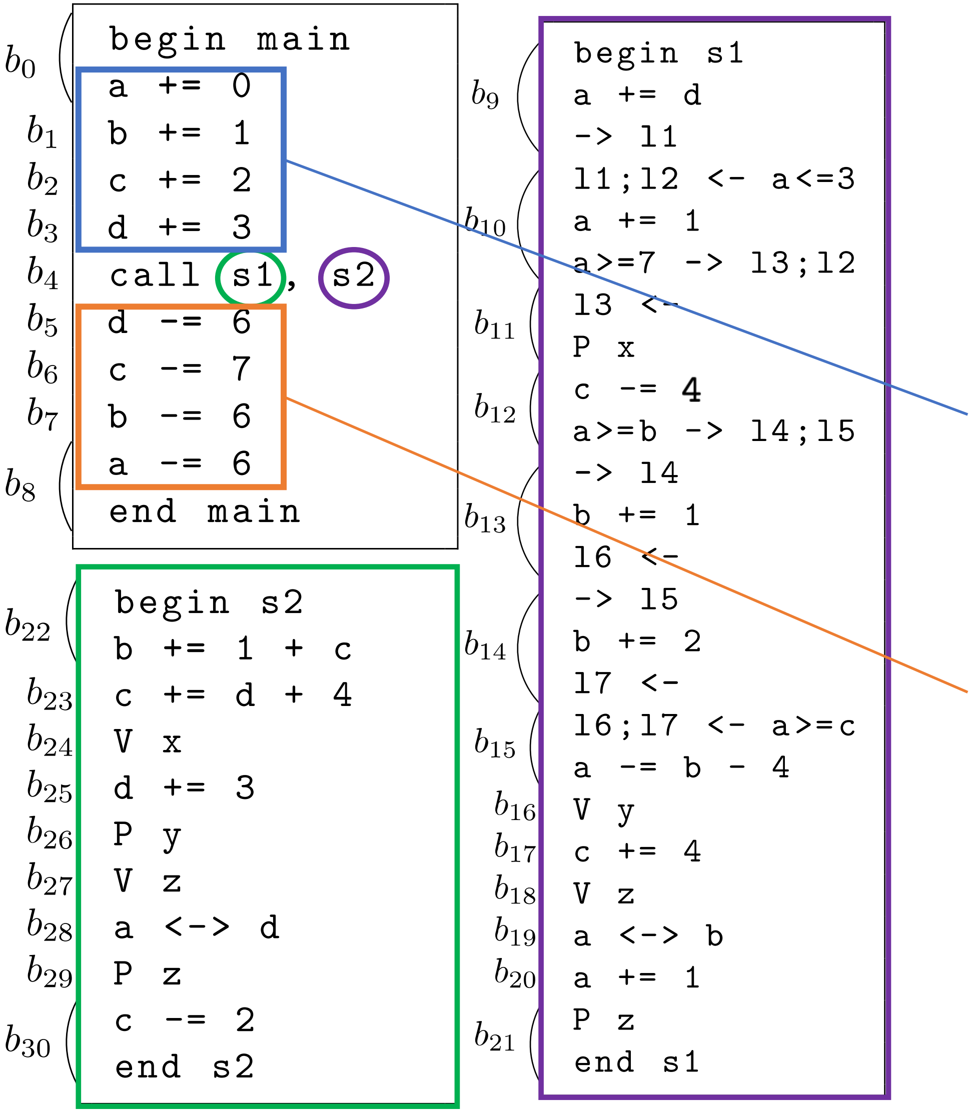
```
begin main
a += 0
b += 1
c += 2
d += 3
call s1, s2
d -= 6
c -= 7
b -= 6
a -= 6
end main
```

```
begin s2
b += 1 + c
c += d + 4
V x
d += 3
P y
V z
a <-> d
P z
c -= 2
end s2
```

```
begin s1
a += d
-> l1
l1;l2 <- a<=3
a += 1
a>=7 -> l3;l2
l3 <-
P x
c -= 4
a>=b -> l4;l5
-> l4
b += 1
l6 <-
-> l5
b += 2
l7 <-
l6;l7 <- a>=c
a -= b - 4
V y
c += 4
V z
a <-> b
a += 1
P z
end s1
```

- We assume all the values of variables starts from and ends to 0 (common in reversible programming).
- In forward direction, the variables are set to a=0, b=1, c=2, d=3 at first.

Example for Constant Propagation



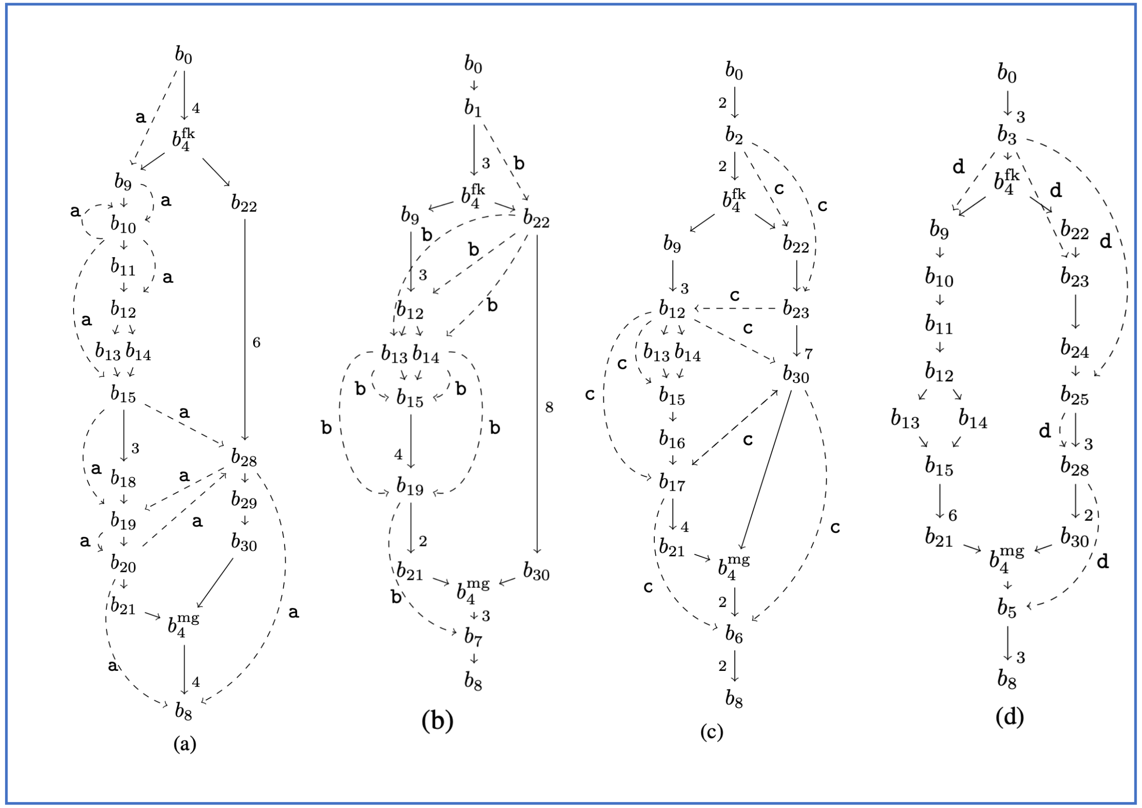
```
begin main
a += 0
b += 1
c += 2
d += 3
call s1, s2
d -= 6
c -= 7
b -= 6
a -= 6
end main
```

```
begin s2
b += 1 + c
c += d + 4
V x
d += 3
P y
V z
a <-> d
P z
c -= 2
end s2
```

```
begin s1
a += d
-> l1
l1;l2 <- a<=3
a += 1
a>=7 -> l3;l2
l3 <-
P x
c -= 4
a>=b -> l4;l5
-> l4
b += 1
l6 <-
-> l5
b += 2
l7 <-
l6;l7 <- a>=c
a -= b - 4
V y
c += 4
V z
a <-> b
a += 1
P z
end s1
```

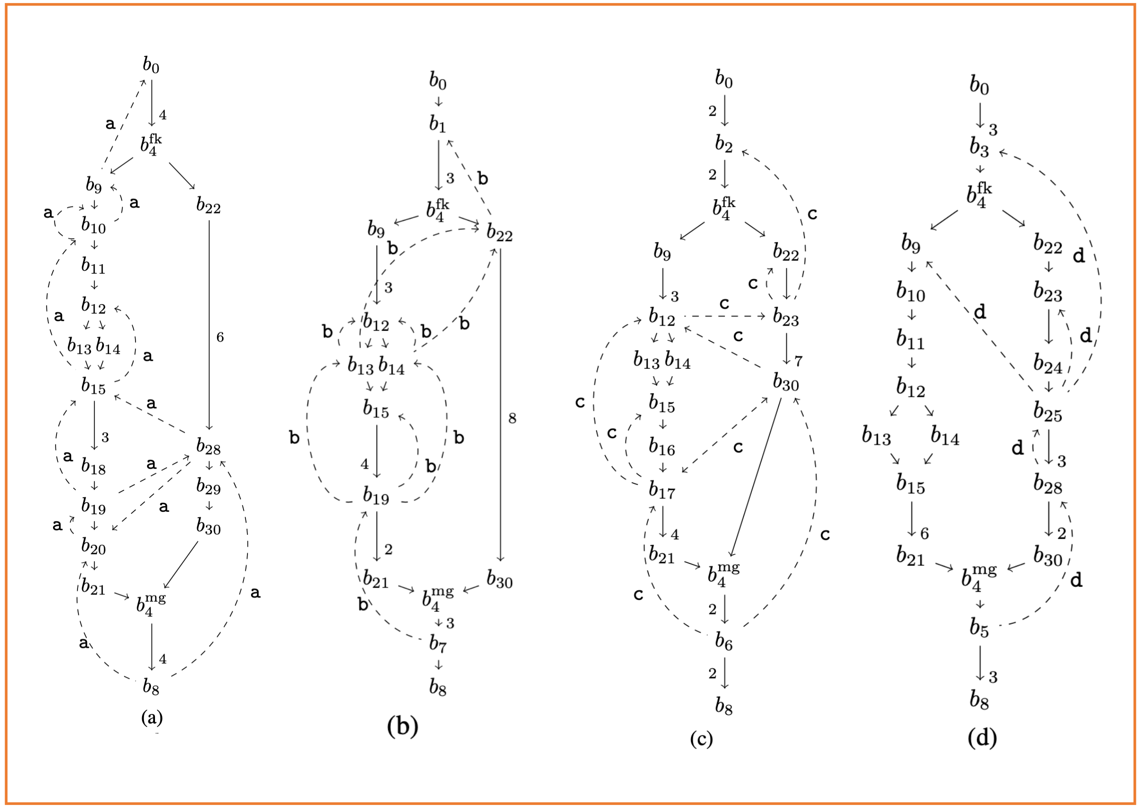
- We assume all the values of variables starts from and ends to 0 (common in reversible programming).
- In forward direction, the variables are set to a=0, b=1, c=2, d=3 at first.
- In backward direction, the variables are set to a=6, b=6, c=7, d=6 at first.

Calculating Data Flow



(a) (b) (c) (d)

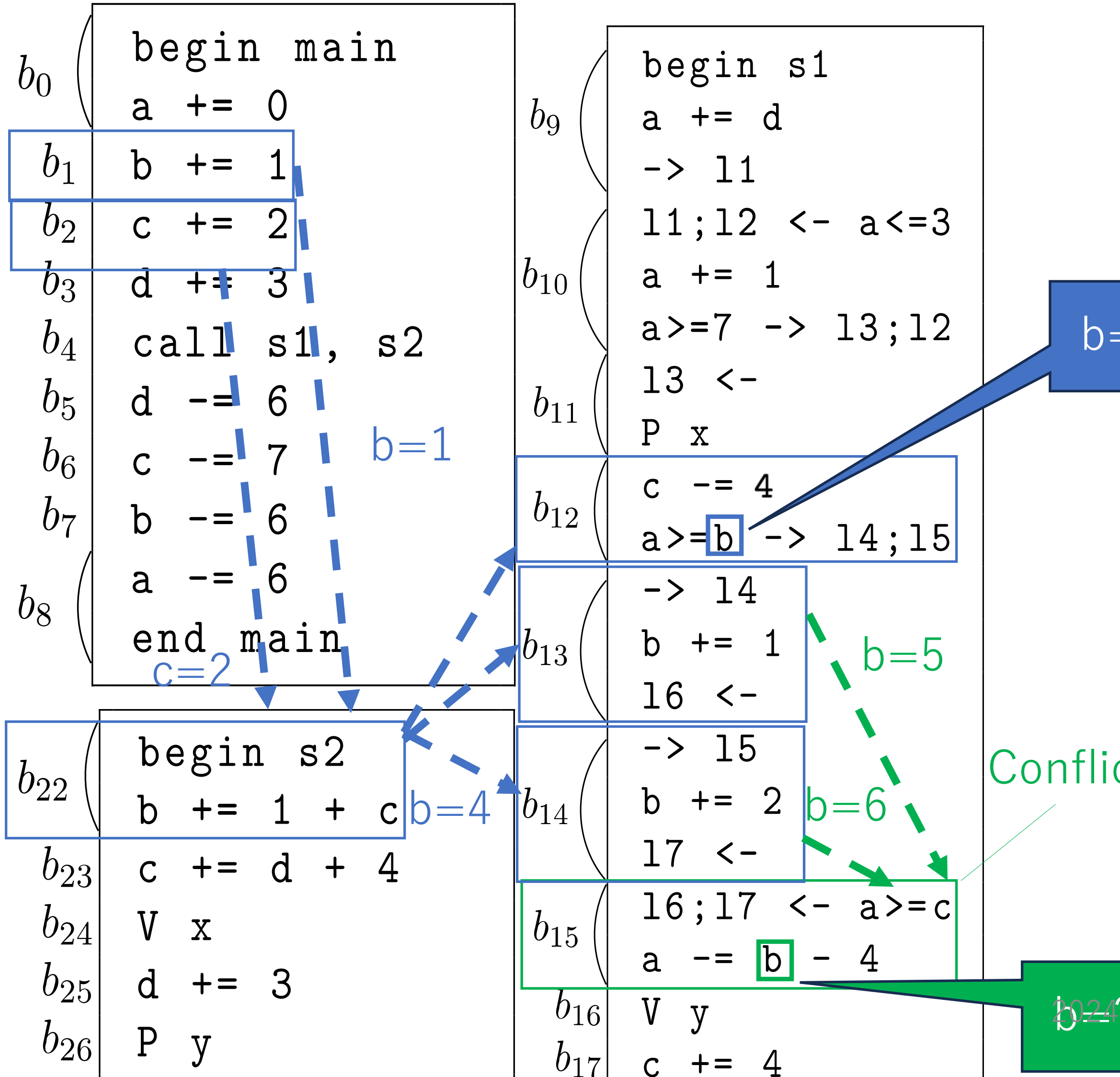
Forward data flow



(a) (b) (c) (d)

Backward data flow

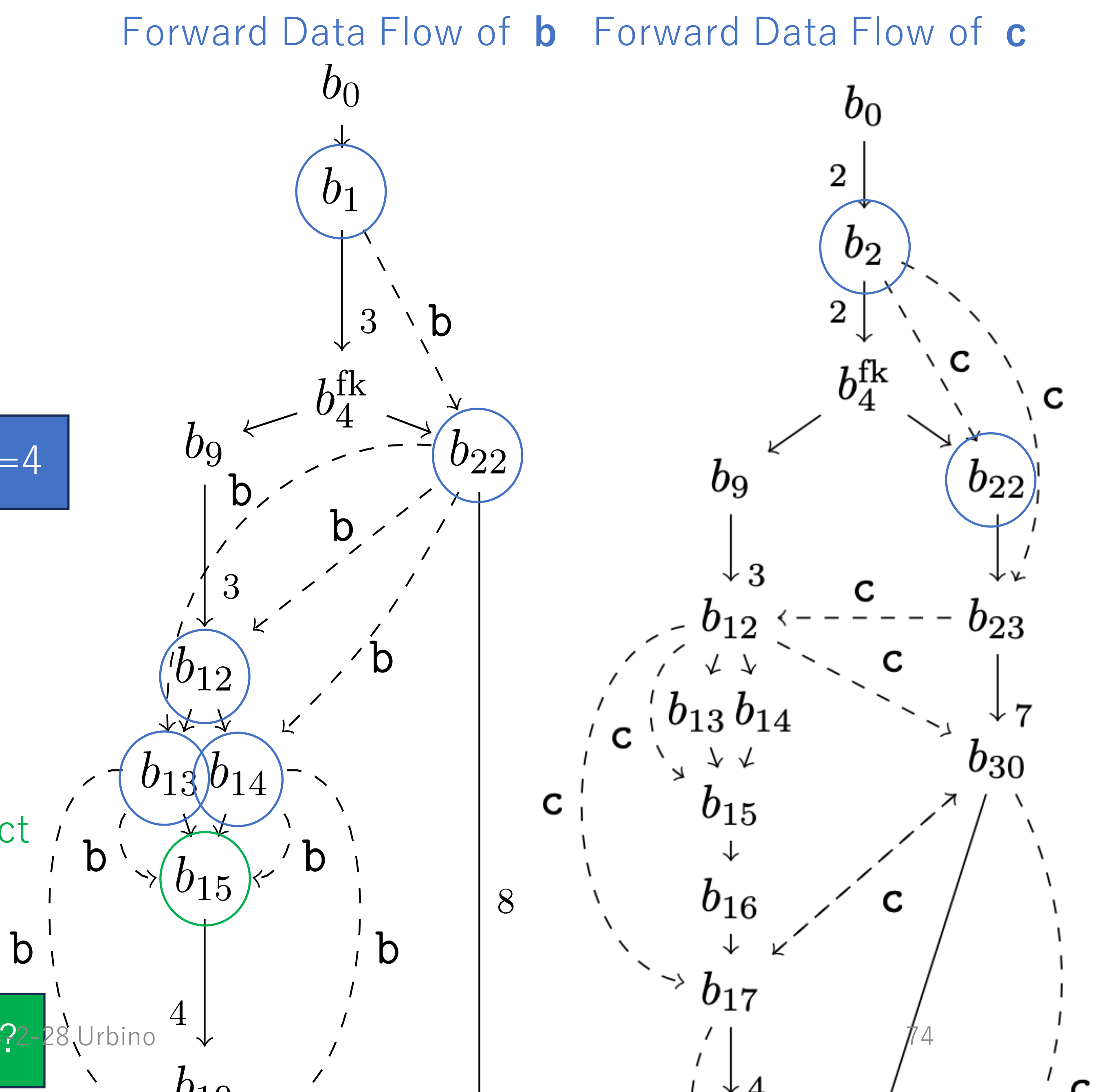
Forward Propagation



```
b0   begin main
     a += 0
b1   b += 1
b2   c += 2
b3   d += 3
b4   call s1, s2
b5   d -= 6
b6   c -= 7
b7   b -= 6
b8   a -= 6
     end main
```

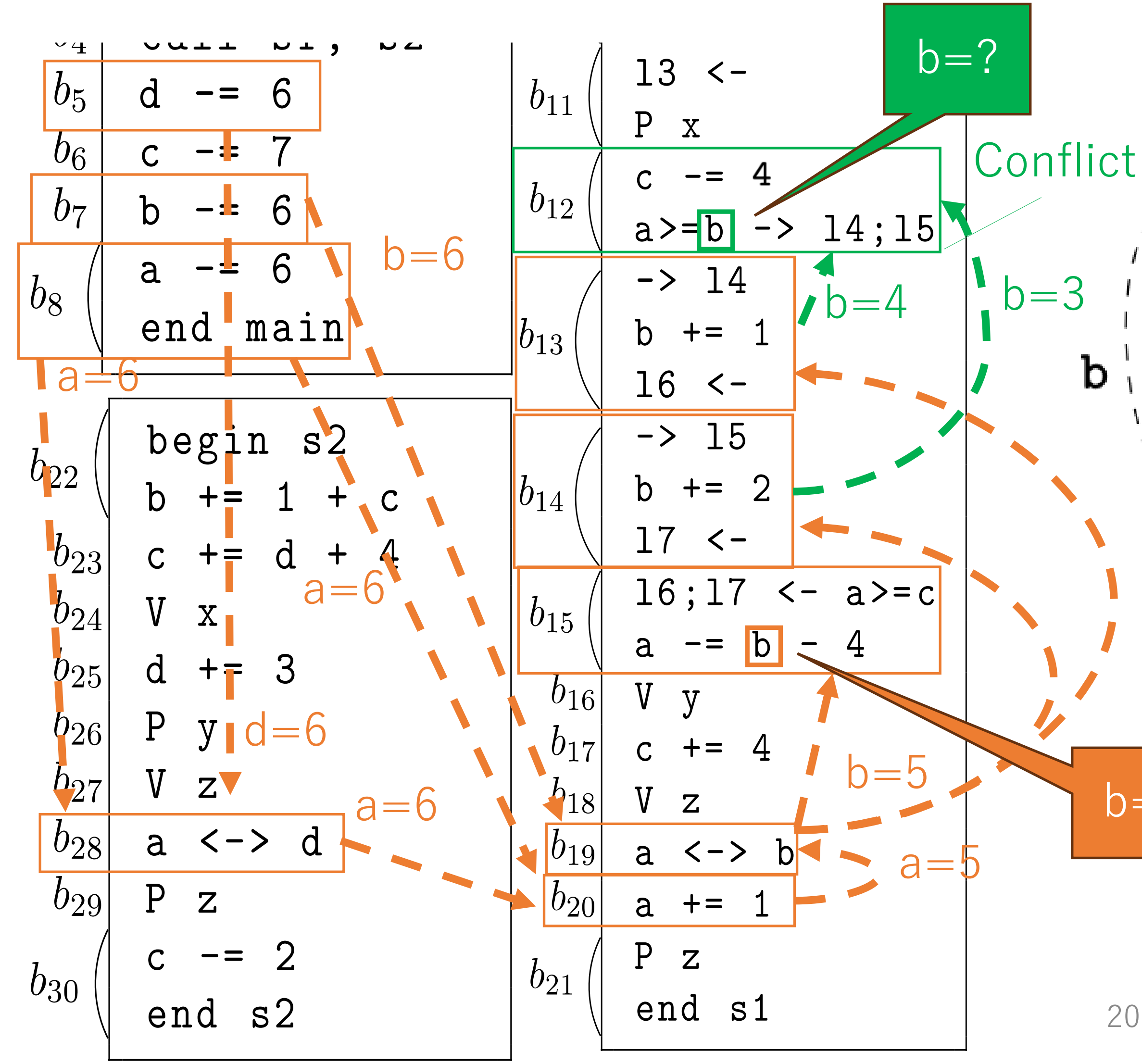
```
b22  begin s2
     b += 1 + c
b23  c += d + 4
b24  V x
b25  d += 3
b26  P y
```

```
b9   begin s1
     a += d
     -> l1
b10  l1;l2 <- a<=3
     a += 1
     a>=7 -> l3;l2
b11  l3 <-
     P x
b12  c -= 4
     a>=b -> l4;l5
b13  -> l4
     b += 1
     l6 <-
b14  -> l5
     b += 2
     l7 <-
b15  l6;l7 <- a>=c
     a -= b - 4
b16  V y
b17  c += 4
```



Forward Data Flow of **b** Forward Data Flow of **c**

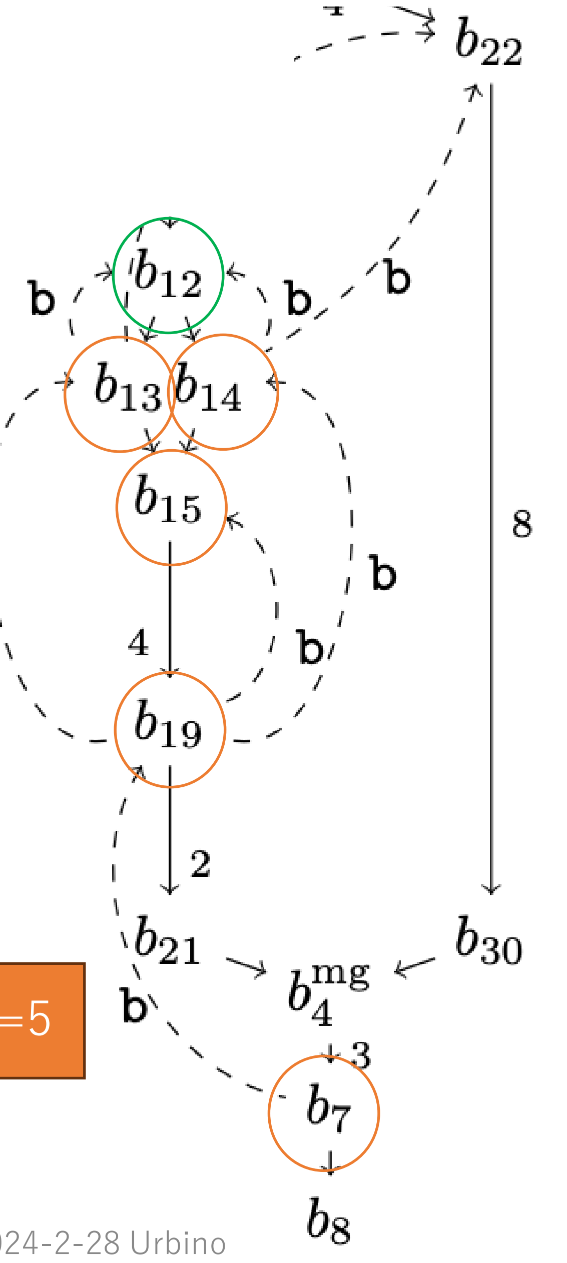
Backward Propagation



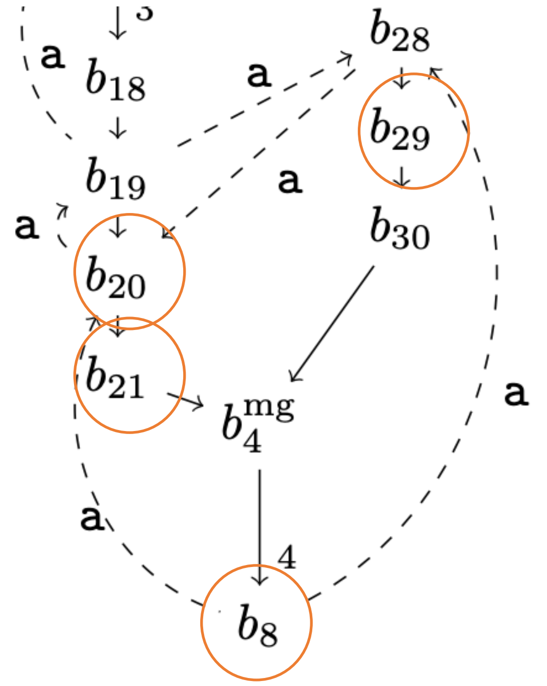
```
b4   call s1, s2
b5   d -= 6
b6   c -= 7
b7   b -= 6
b8   a -= 6
     end main

b22  begin s2
     b += 1 + c
b23  c += d + 4
b24  V x
b25  d += 3
b26  P y
b27  V z
b28  a <-> d
b29  P z
b30  c -= 2
     end s2
```

```
b11  l3 <-
     P x
b12  c -= 4
     a>=b -> l4;l5
b13  -> l4
     b += 1
     l6 <-
b14  -> l5
     b += 2
     l7 <-
b15  l6;l7 <- a>=c
     a -= b - 4
b16  V y
b17  c += 4
b18  V z
b19  a <-> b
b20  a += 1
b21  P z
     end s1
```

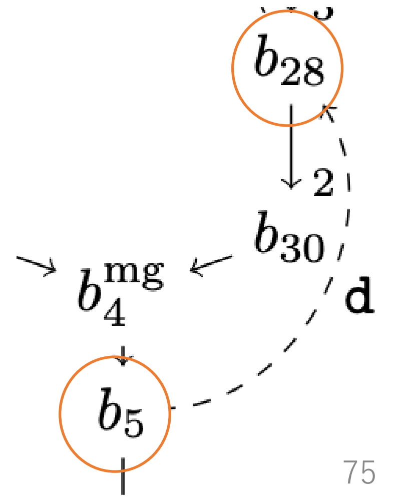
Forward Data Flow of **b**



Forward Data Flow of **a**



Forward Data Flow of **d**



Backward Propagation

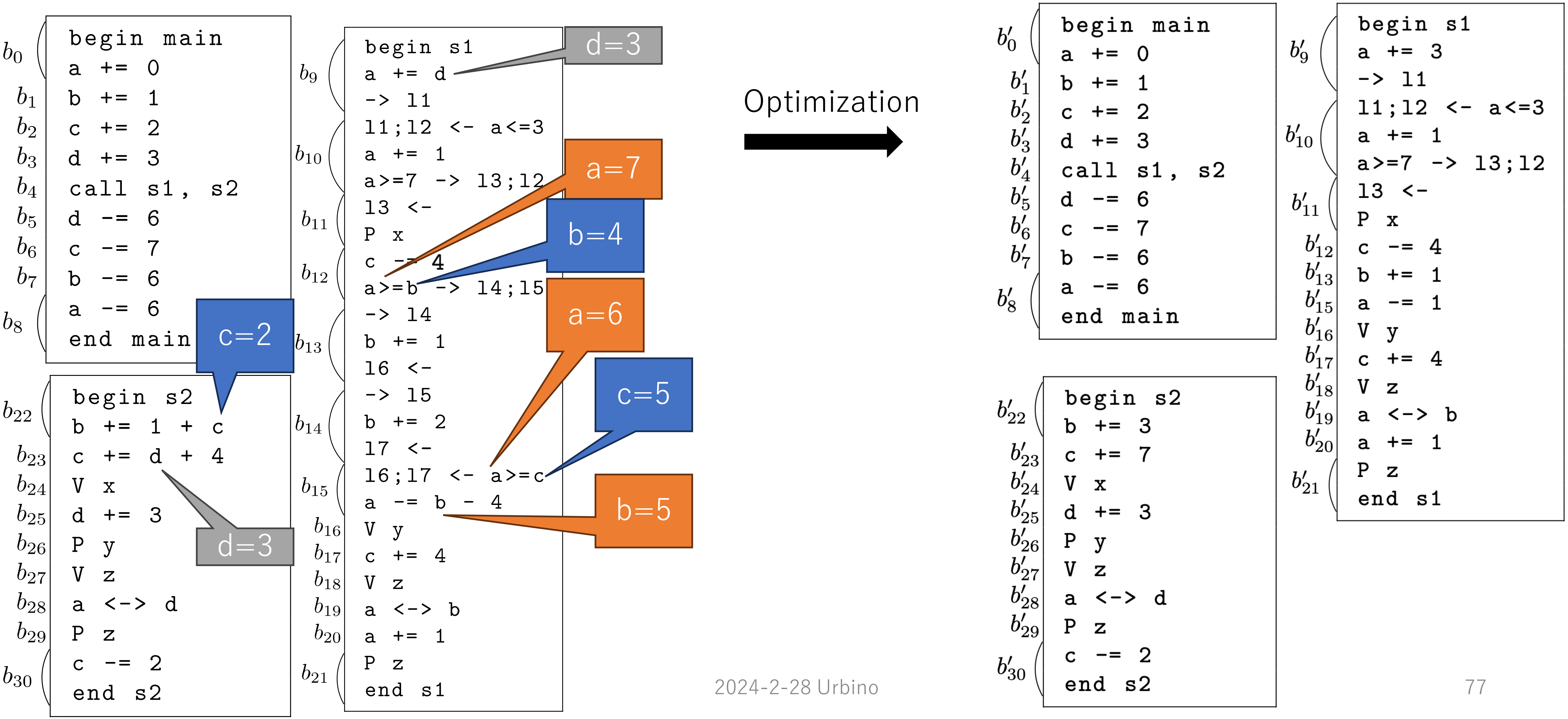
```
b0   begin main
     a += 0
b1   b += 1
b2   c += 2
b3   d += 3
b4   call s1, s2
b5   d -= 6
b6   c -= 7
b7   b -= 6
b8   a -= 6
     end main
```

```
b22  begin s2
     b += 1 + c
b23  c += d + 4
b24  V x
b25  d += 3
b26  P y
b27  V z
b28  a <-> d
b29  P z
b30  c -= 2
     end s2
```

```
b9   begin s1
     a += d
     -> l1
b10  l1;l2 <- a<=3
     a += 1
     a>=7 -> l3;l2
b11  l3 <-
     P x
b12  c -= 4
     a>=b -> l4;l5
b13  -> l4
     b += 1
     l6 <-
b14  -> l5
     b += 2
     l7 <-
b15  l6;l7 <- a>=c
     a -= b - 4
b16  V y
b17  c += 4
b18  V z
b19  a <-> b
b20  a += 1
b21  P z
     end s1
```

Forward Propagation	Backward Propagation	Result
b=4	b=?	b=4
b=?	b=5	b=5

Result of Constant Propagation



```
b0    begin main
      a += 0
b1    b += 1
b2    c += 2
b3    d += 3
b4    call s1, s2
b5    d -= 6
b6    c -= 7
b7    b -= 6
b8    a -= 6
      end main
```

```
b22   begin s2
      b += 1 + c
b23   c += d + 4
b24   V x
b25   d += 3
b26   P y
b27   V z
b28   a <-> d
b29   P z
b30   c -= 2
      end s2
```

```
b9    begin s1
      a += d
      -> l1
b10   l1;l2 <- a<=3
      a += 1
      a>=7 -> l3;l2
b11   l3 <-
      P x
b12   c -= 4
      a>=b -> l4;l5
b13   -> l4
      b += 1
      l6 <-
b14   -> l5
      b += 2
      l7 <-
b15   l6;l7 <- a>=c
      a -= b - 4
b16   V y
b17   c += 4
b18   V z
b19   a <-> b
b20   a += 1
b21   P z
      end s1
```

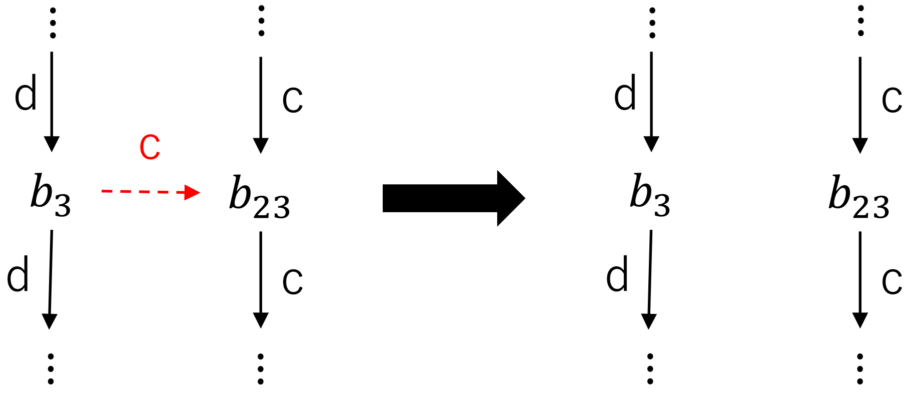
```
b'0   begin main
      a += 0
b'1   b += 1
b'2   c += 2
b'3   d += 3
b'4   call s1, s2
b'5   d -= 6
b'6   c -= 7
b'7   b -= 6
b'8   a -= 6
      end main
```

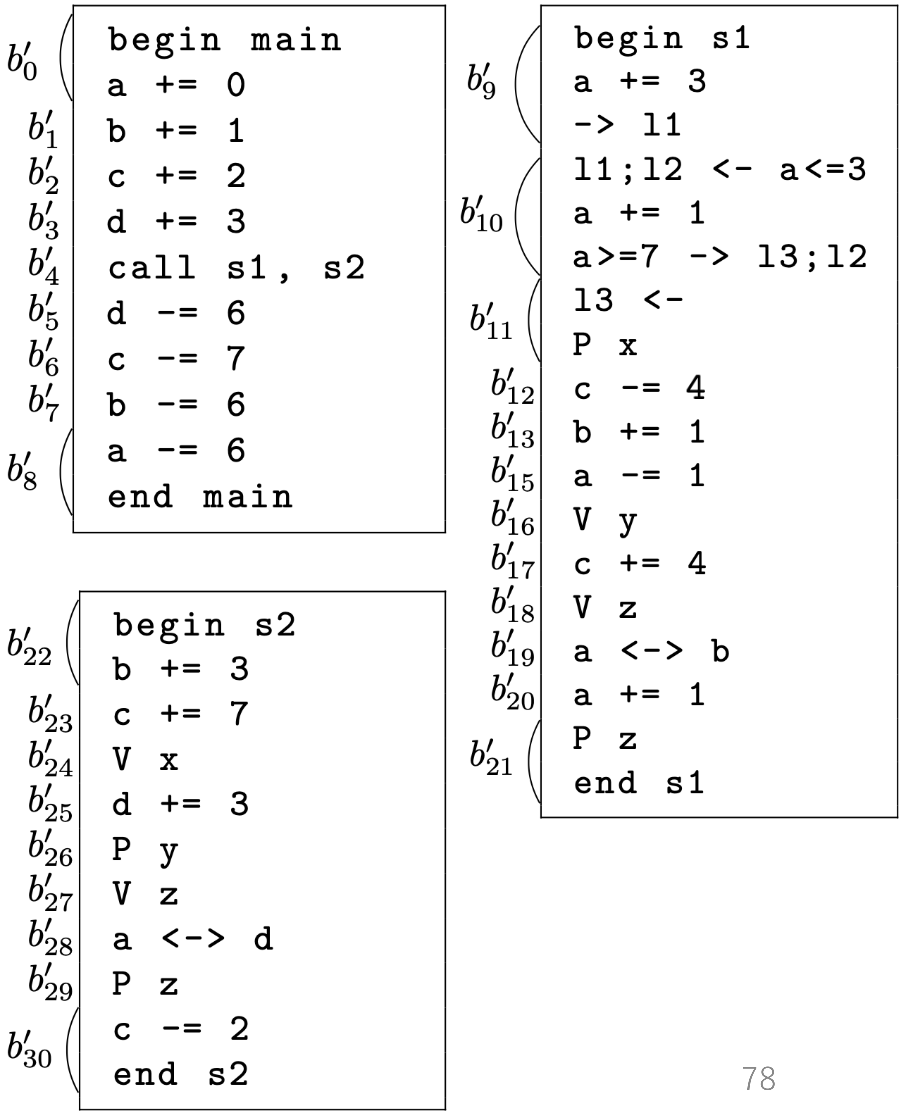
```
b'22  begin s2
      b += 3
b'23  c += 7
b'24  V x
b'25  d += 3
b'26  P y
b'27  V z
b'28  a <-> d
b'29  P z
b'30  c -= 2
      end s2
```

```
b'9   begin s1
      a += 3
      -> l1
b'10  l1;l2 <- a<=3
      a += 1
      a>=7 -> l3;l2
b'11  l3 <-
      P x
b'12  c -= 4
b'13  b += 1
b'15  a -= 1
b'16  V y
b'17  c += 4
b'18  V z
b'19  a <-> b
b'20  a += 1
b'21  P z
      end s1
```

Effect of Constant Propagation

- The program becomes simpler.
- The number of variable reads decrease.
- The annotation DAG after optimization is smaller than the pre-optimized version.





```
begin main
a += 0
b += 1
c += 2
d += 3
call s1, s2
d -= 6
c -= 7
b -= 6
a -= 6
end main
```

```
begin s1
a += 3
-> l1
l1;l2 <- a<=3
a += 1
a>=7 -> l3;l2
l3 <-
P x
c -= 4
b += 1
a -= 1
V y
c += 4
V z
a <-> b
a += 1
P z
end s1
```

```
begin s2
b += 3
c += 7
V x
d += 3
P y
V z
a <-> d
P z
c -= 2
end s2
```

Contents

- Background & Motivation
- CRIL: Concurrent Reversible Intermediate Language
- The controlled semantics of CRIL
- Reversibility Properties
- Bidirectional Data Flow Analysis in CRIL
- Constant Propagation in CRIL
- Concluding Remarks

Concluding Remarks

- **CRIL** as a concurrent reversible intermediate language.
- Contorlled semantics with **annotation DAG**
- The Controlled semantics has **Causal Safety** and **Causal Liveness**,
- **Bidrectional data flow analysis** in CRIL
- **Constant propagation** to CRIL

Future work

- A variant of **SSA** (Static Single Assignment) form for other optimization techniques
  (Work-in-progress) An extension of RSSA
- **Channel-based communication** for the message-passing.
  Application/Extension for Go?
- **Effect of Annotation DAG in Reversibility**
  Less reversible with less dependency structure?