

Formal Analysis and Verification

FAV 2011/2012

Tomáš Vojnar, Lukáš Holík

`{vojnar,holik}@fit.vutbr.cz`

**Brno University of Technology
Faculty of Information Technology
Božetěchova 2, 612 66 Brno**

Data Flow Analysis

Introduction

- ❖ Data flow analysis is a special form of static analysis.
- ❖ Data flow analyses compute abstract information about the states that a program can reach at each control point via solving in an iterative way data flow equations describing in an abstract way the behaviour of a program.
- ❖ In forward data flow analyses, the data flow equations describe
 - how a statement (block of statements) transforms some abstract value at its input to an abstract value at its output, and
 - how multiple abstract values at the input of a statement (block of statements) are combined into a single value.
- ❖ Backward data flow analyses are analogical.
- ❖ Initial abstract states for particular control points must also be specified: typically, they have non-trivial values for entry/exit program points.

Introduction

- ❖ The data flow equations for a given data flow analysis are **created manually**.
- ❖ Data flow analyses can have a different scope:
 - **Local data flow analysis**: performed within maximal sequences of statements with no control transfer into their middle/from their middle (basic blocks).
 - **Global data flow analysis**: performed within a single function/procedure.
 - **Interprocedural data flow analysis**: performed across functions/procedures.
- ❖ **Possible applications** of data flow analysis include **optimisation, verification, program understanding**, e.g., for debugging.

A Motivating Example: Code Optimisation

A Motivating Example: Code Optimisation

❖ Consider the following program:

```
Procedure: quicksort(int m, int n)
if  $n < m$  then return;
 $i = m - 1; j = n; v = a[n];$                                 /*  $v$  is the pivot. */
while true do                                             /* Move values smaller */
    repeat  $i = i + 1$  until  $a[i] < v;$                     /* than  $v$  to the left */
    repeat  $j = j - 1$  until  $a[j] > v;$                     /* of the split point (sp) */
    if  $i \geq j$  then break;                                /* and other values */
     $x = a[i]; a[i] = a[j]; a[j] = x;$                         /* to the right of sp. */
end
 $x = a[i]; a[i] = a[n]; a[n] = x;$                             /* Move pivot to sp, and sort */
quicksort( $m, i$ ); quicksort( $i+1, n$ );                        /* the partitions independently. */
```

Intermediate Code

❖ When the program is compiled in a naïve way, many redundancies can arise (the initial test of $n < m$ is left out):

1: $i = m - 1$	12: $t5 = a[t4]$	23: $a[t4] = x$
2: $j = n$	13: if $t5 > v$ goto 10	24: goto 6
3: $t1 = 4 \times n$	14: if $i \geq j$ goto 25	25: $t2 = 4 \times i$
4: $t6 = a[t1]$	15: $t2 = 4 \times i$	26: $t3 = a[t2]$
5: $v = t6$	16: $t3 = a[t2]$	27: $x = t3$
6: $i = i + 1$	17: $x = t3$	28: $t2 = 4 \times i$
7: $t2 = 4 \times i$	18: $t2 = 4 \times i$	29: $t1 = 4 \times n$
8: $t3 = a[t2]$	19: $t4 = 4 \times j$	30: $t6 = a[t1]$
9: if $t3 < v$ goto 6	20: $t5 = a[t4]$	31: $a[t2] = t6$
10: $j = j - 1$	21: $a[t2] = t5$	32: $t1 = 4 \times n$
11: $t4 = 4 \times j$	22: $t4 = 4 \times j$	33: $a[t1] = x$

❖ Data flow analysis—e.g., in the form of deriving available expressions—can help us detect such redundancies (and then get rid of them).

Basic Blocks and CFGs

❖ To decrease the granularity of the code to be analysed, one often works with the so called basic blocks instead of particular statements.

❖ A basic block is a maximal sequence of statements that is

- always entered via its first statement and
- always left via its last statement,

i.e., there is no transfer of control into the middle/from the middle of the sequence of statements.

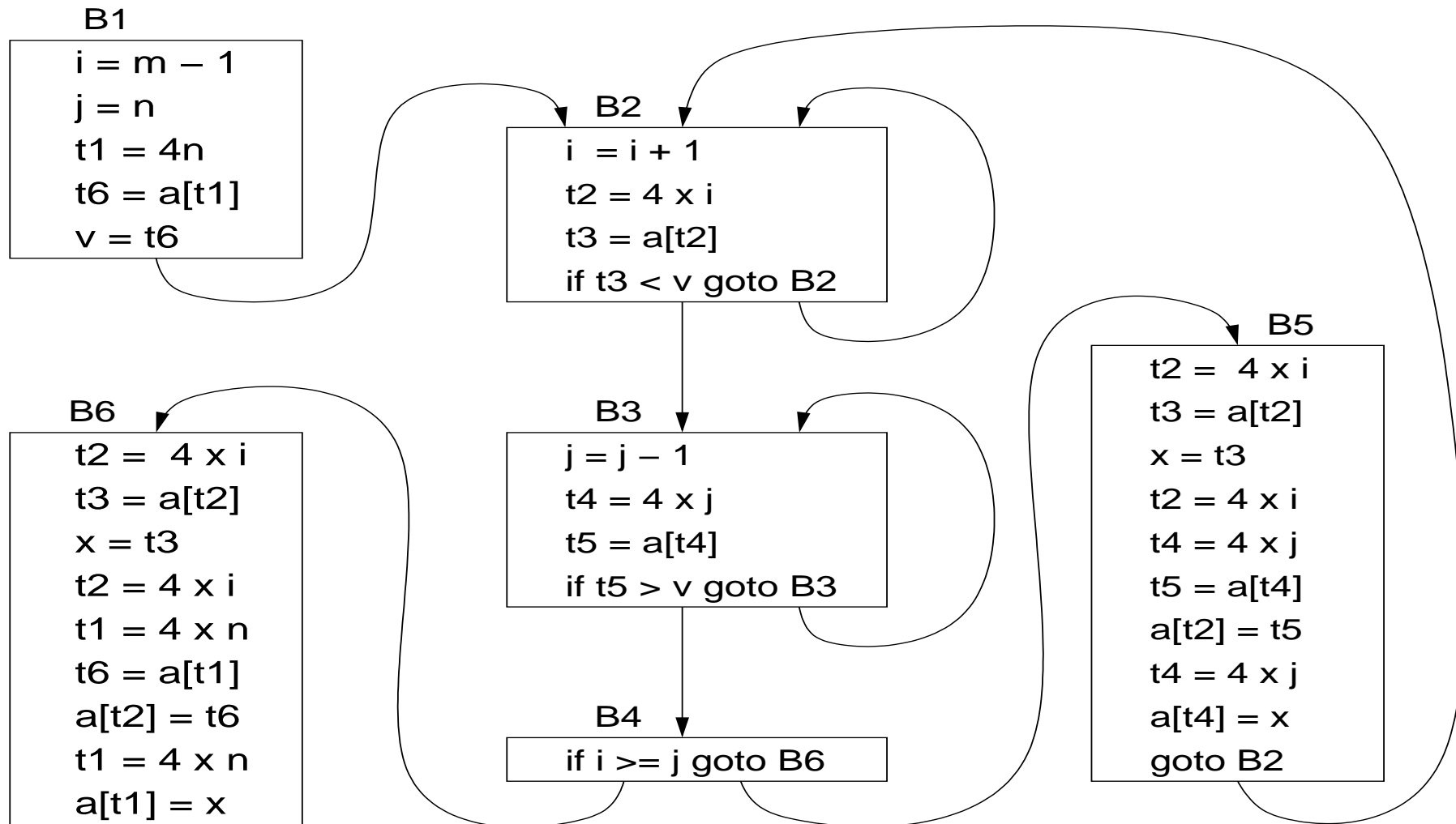
❖ A control flow graph (CFG) is an oriented graph where nodes are basic blocks, and edges follow the transfer of control.

❖ For simplicity, we assume that a control flow graph always contains

- a *Start* block with no incoming edges and
- an *End* block with no outgoing edges.

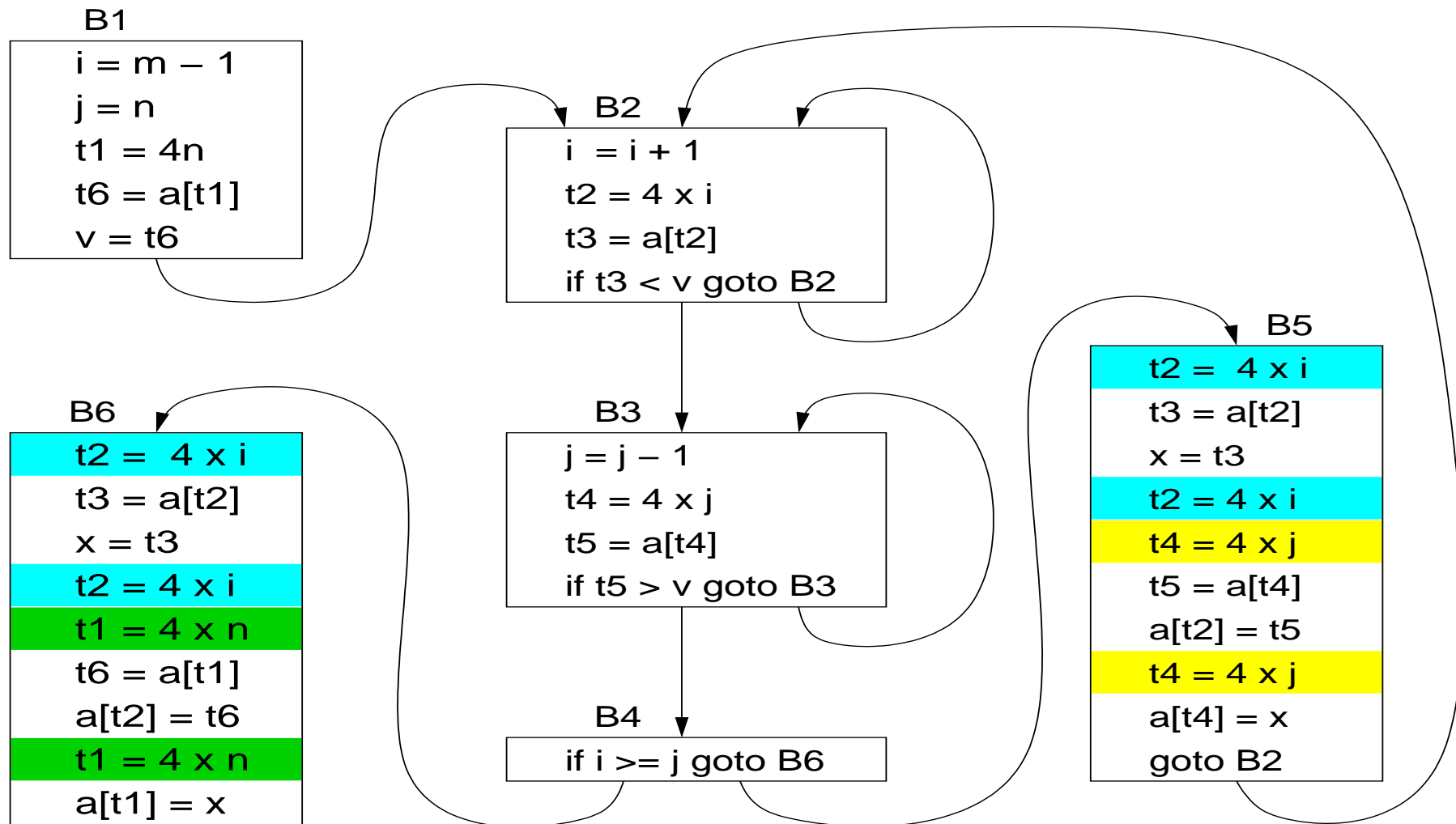
An Example of a CFG

❖ Here is the CFG for our quicksort example (without the initial test of $n < m$):



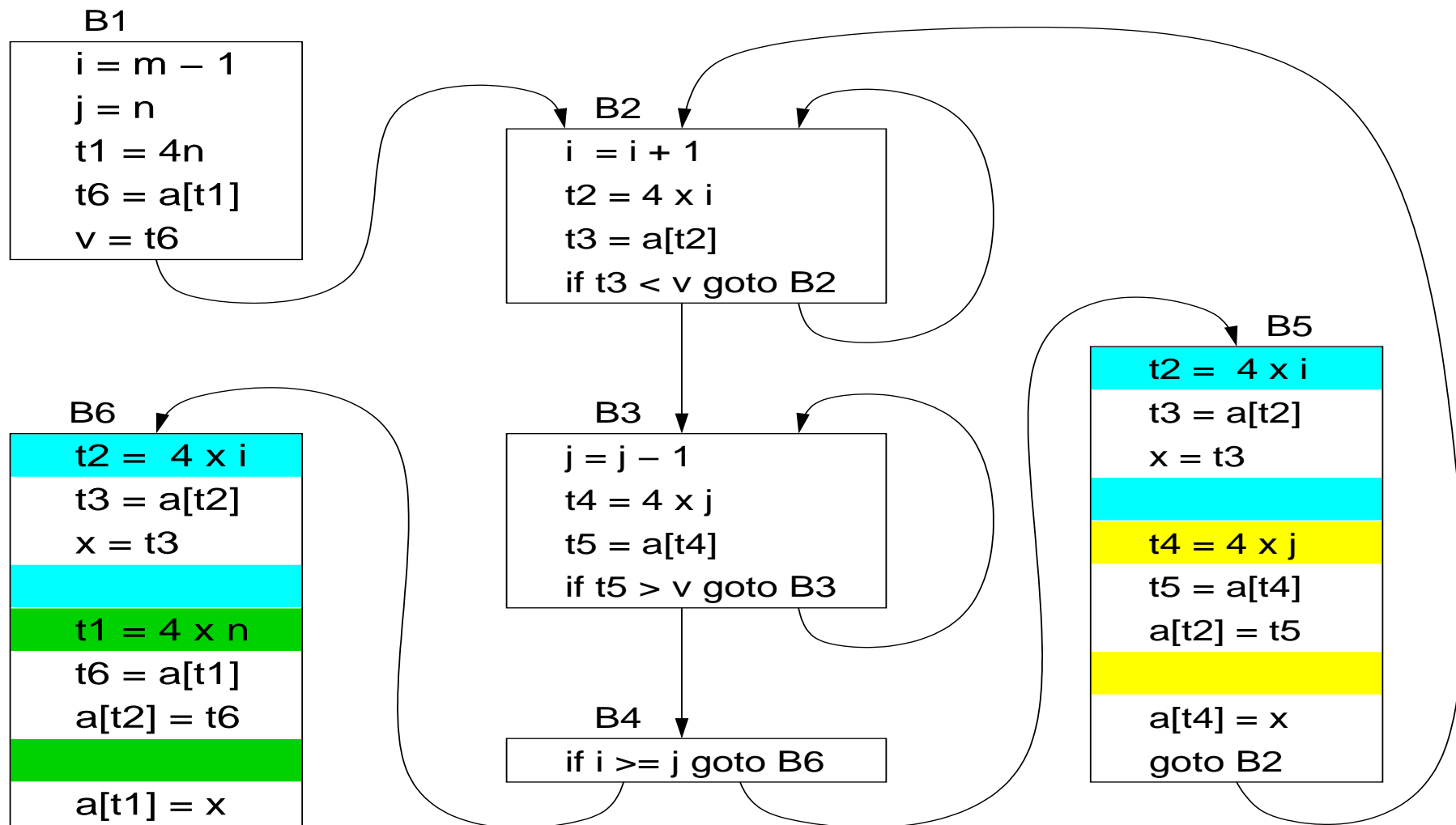
❖ We will show some (simplified) optimisations on this example.

Local Common Subexpression Elimination



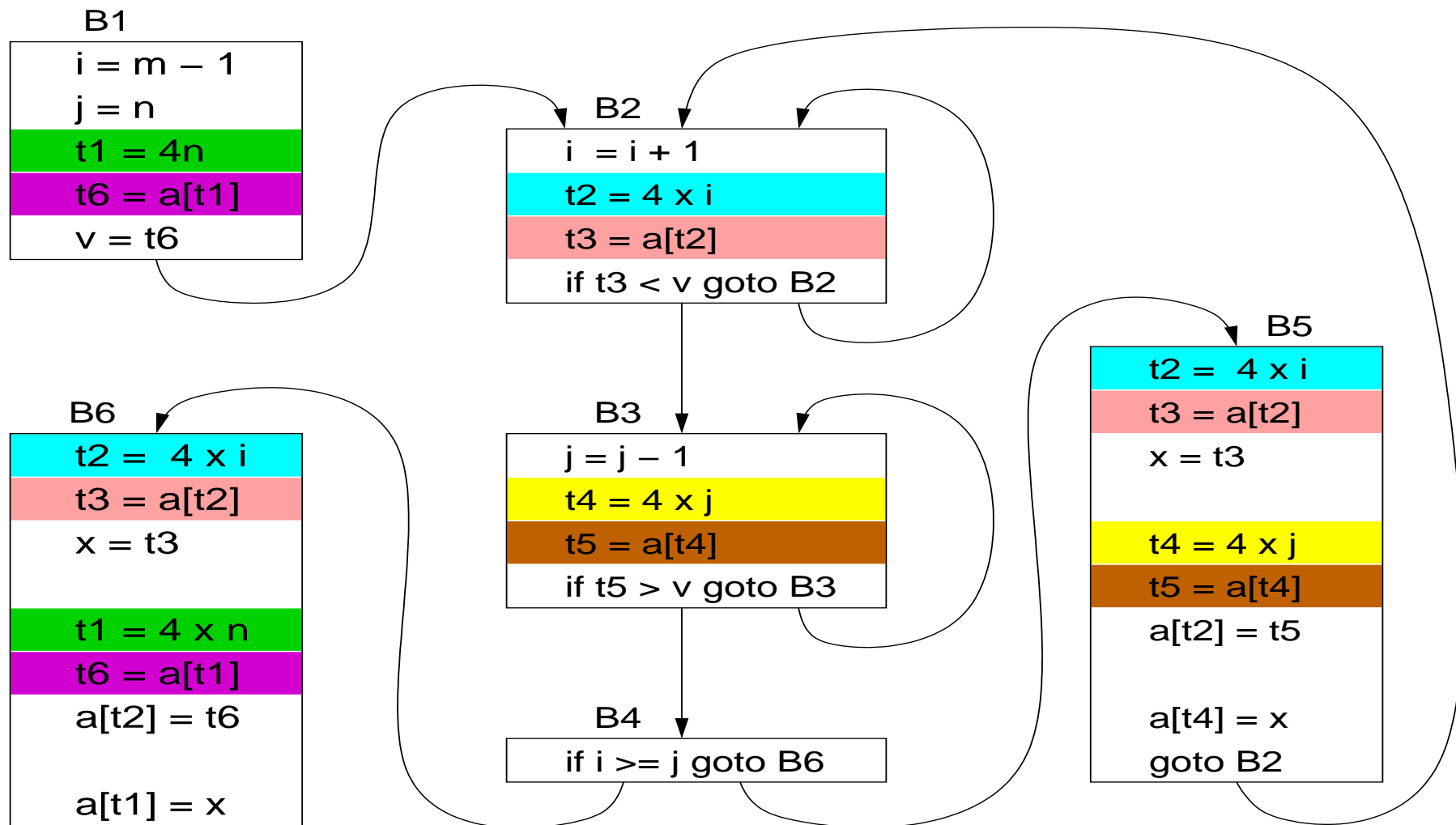
❖ An evaluation of an expression whose value is available from within a basic block at a program point p can be eliminated.

Local Common Subexpression Elimination



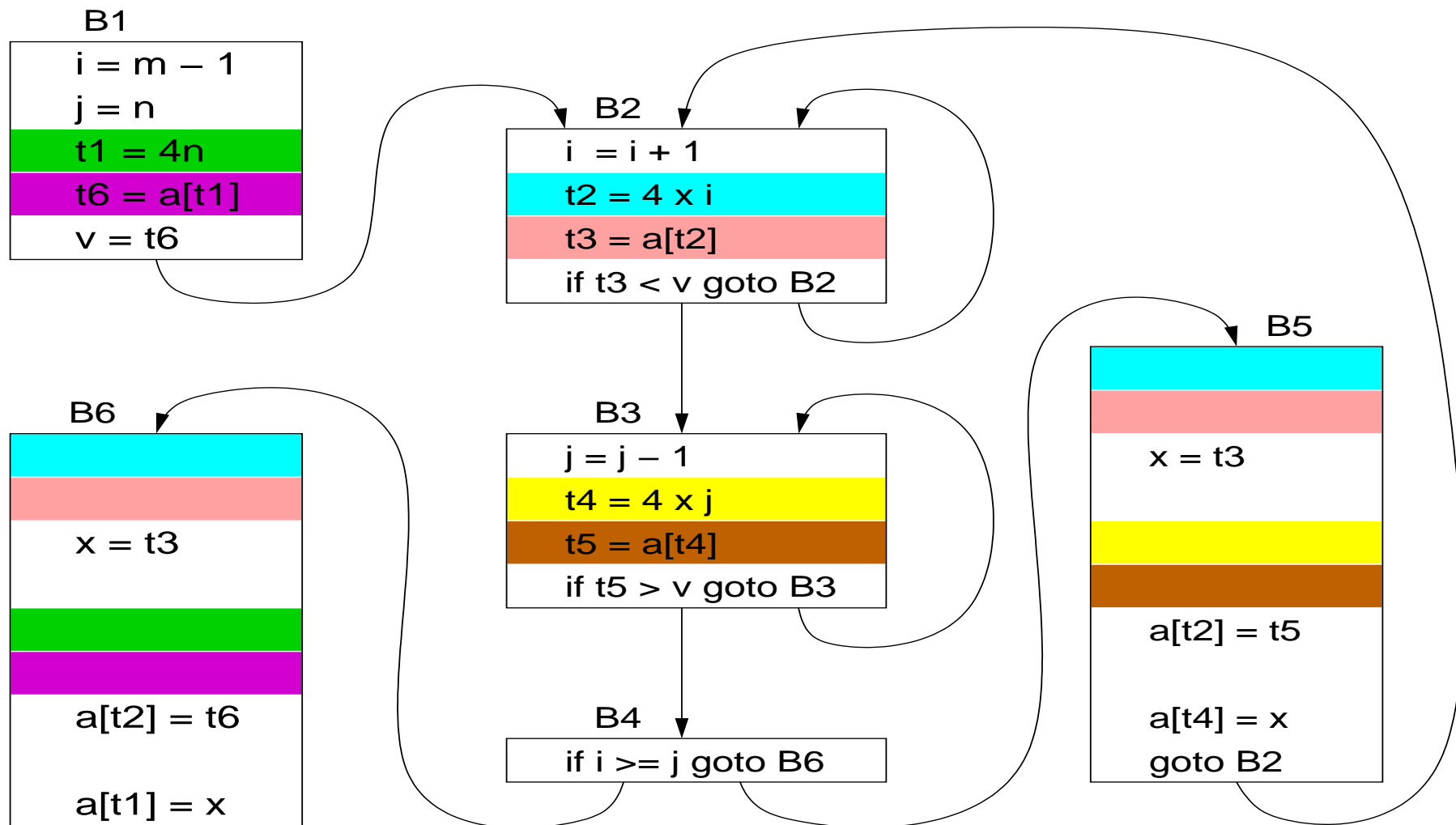
❖ An evaluation of an expression whose value is available from within a basic block at a program point p can be eliminated.

Global Common Subexpression Elimination



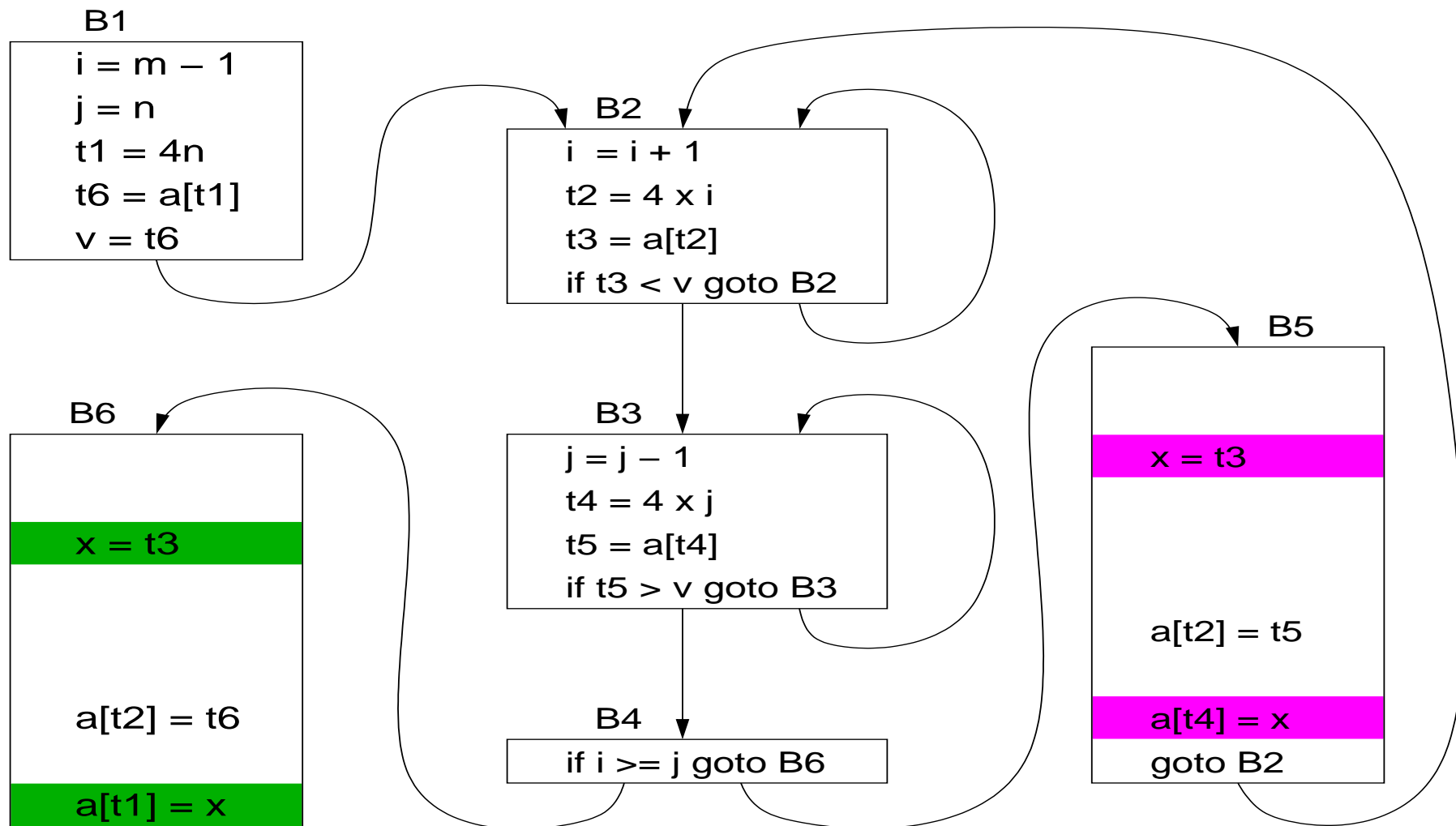
❖ An expression whose value is available from within the entire control flow graph at a program point p can be eliminated.

Global Common Subexpression Elimination



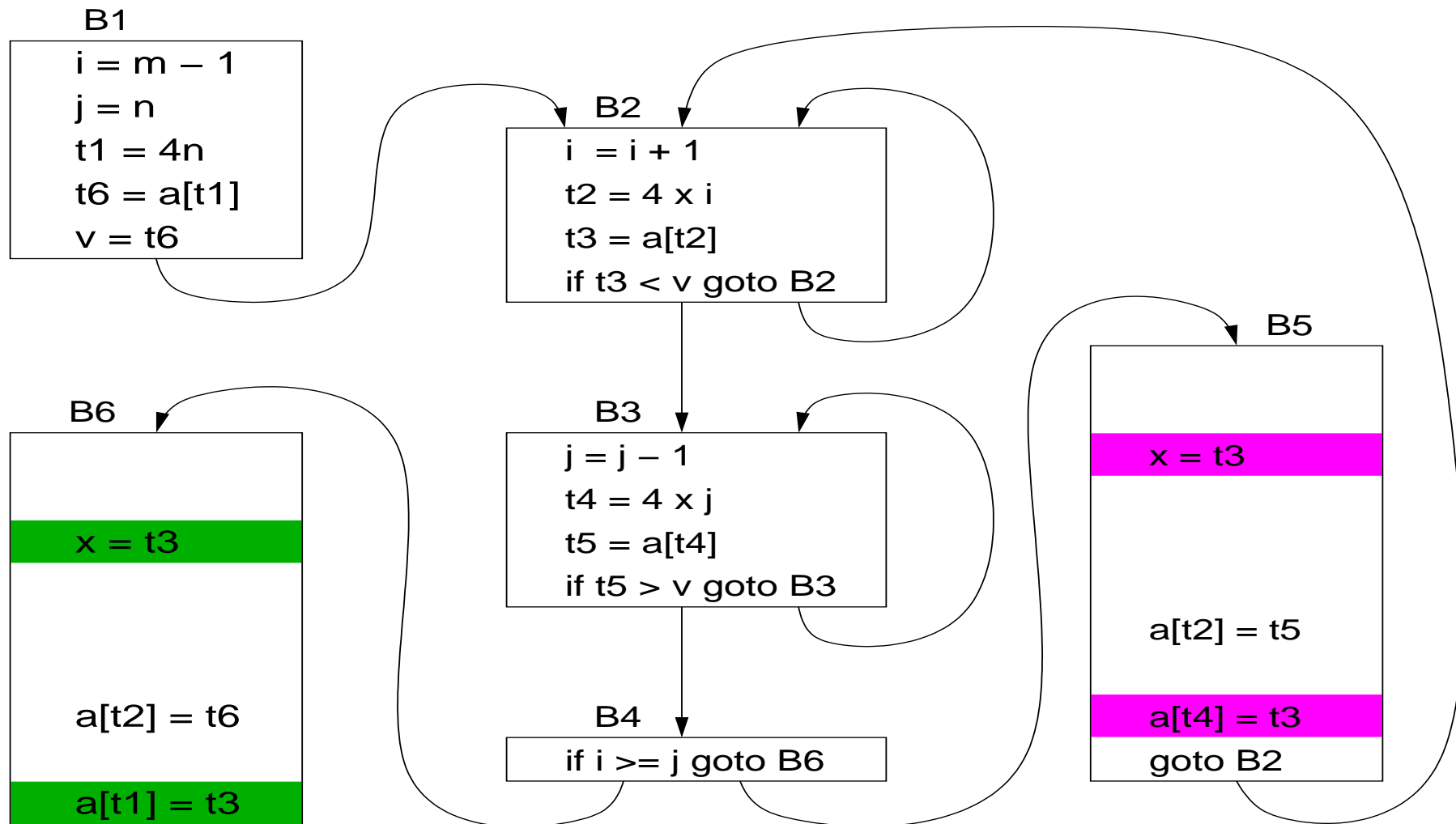
❖ An expression whose value is available from within the entire control flow graph at a program point p can be eliminated.

Copy Propagation and Dead Variable Elimination



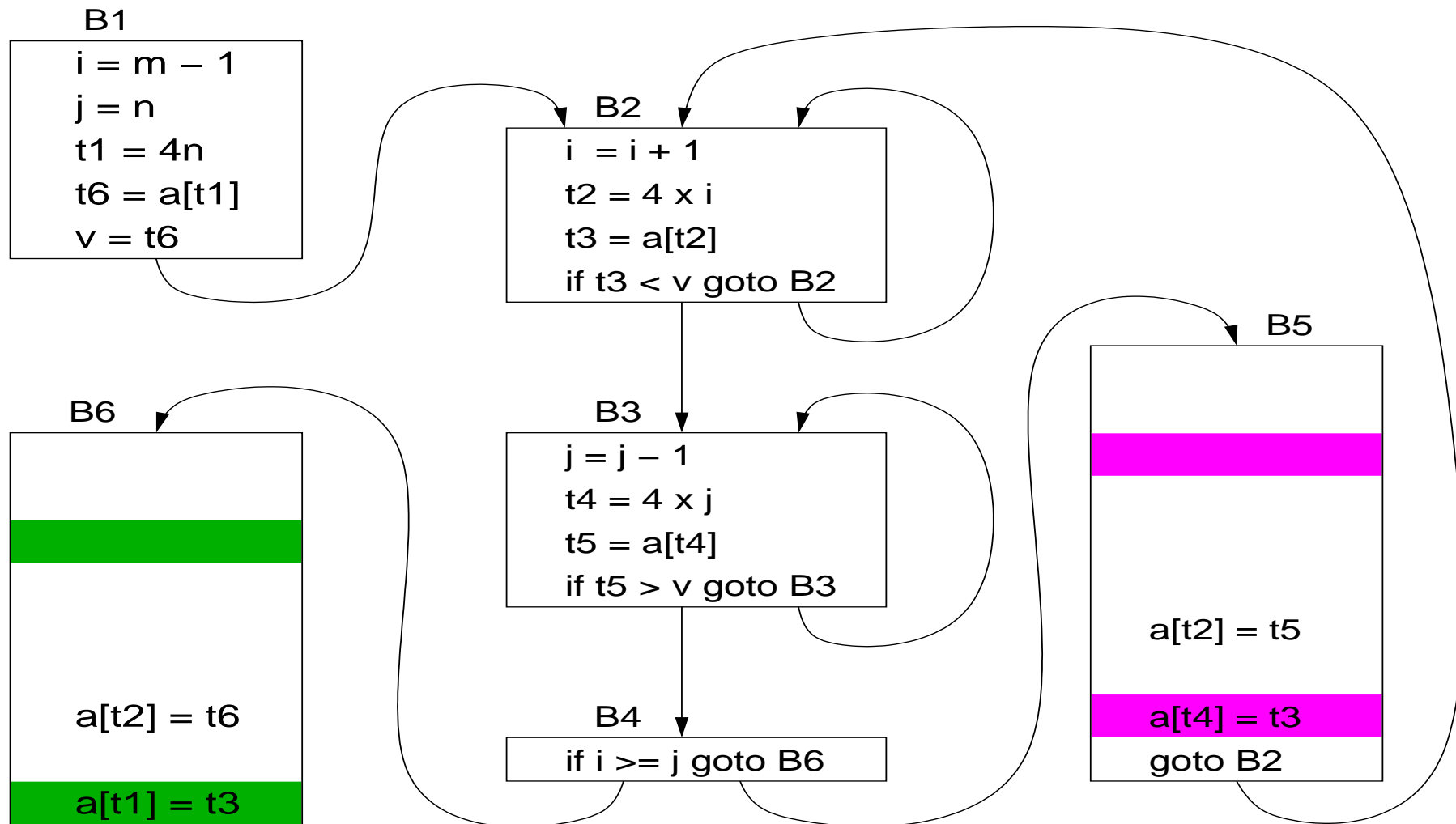
- ❖ Repeated copying of the value of an expression that stays constant in some part of a program can be avoided by **propagating the constant value** of this expression.
- ❖ An assignment of a value to a **variable such that the assigned value is never used** can be eliminated.

Copy Propagation and Dead Variable Elimination



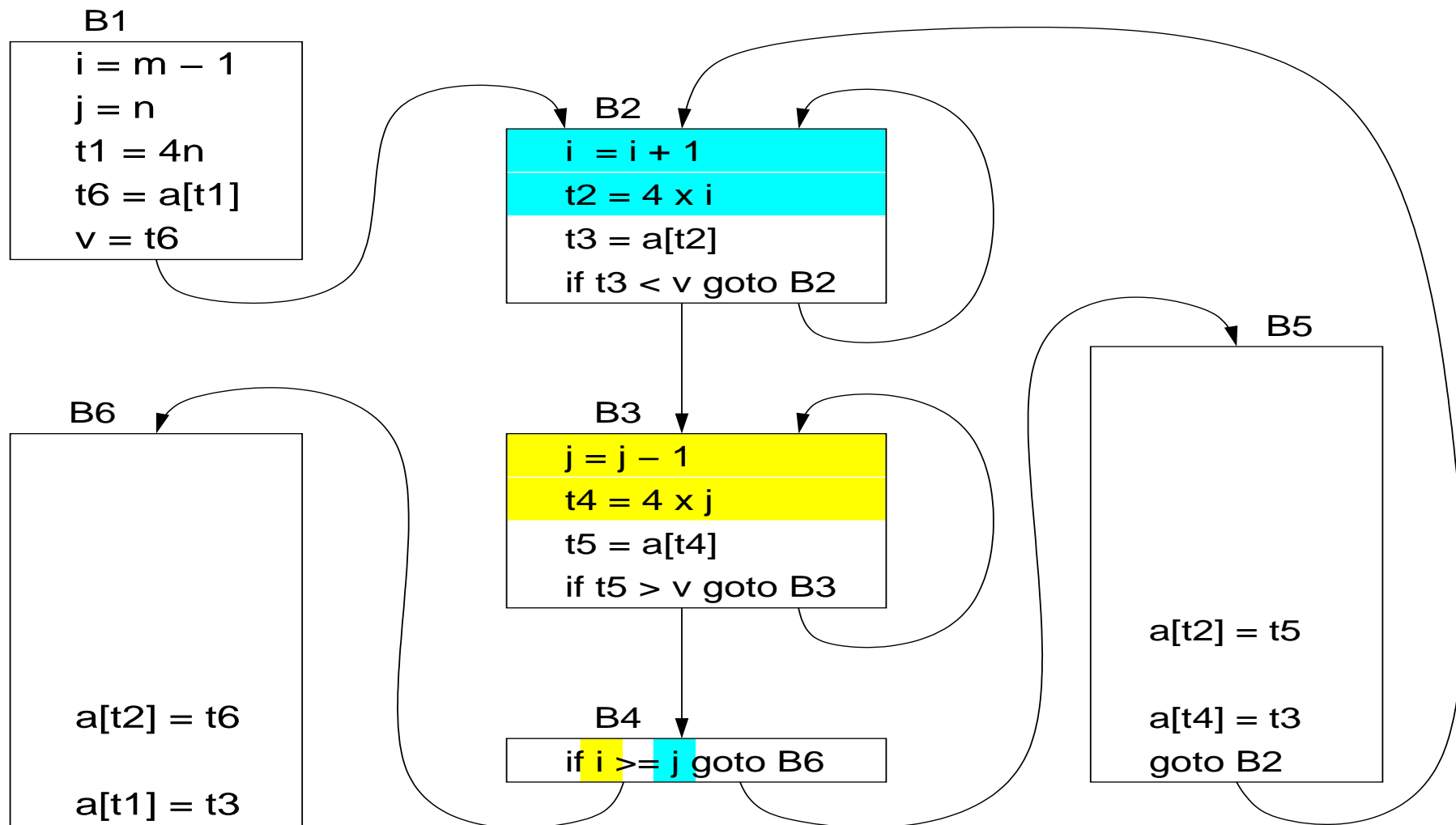
- ❖ Repeated copying of the value of an expression that stays constant in some part of a program can be avoided by **propagating the constant value** of this expression.
- ❖ An assignment of a value to a **variable such that the assigned value is never used** can be eliminated.

Copy Propagation and Dead Variable Elimination



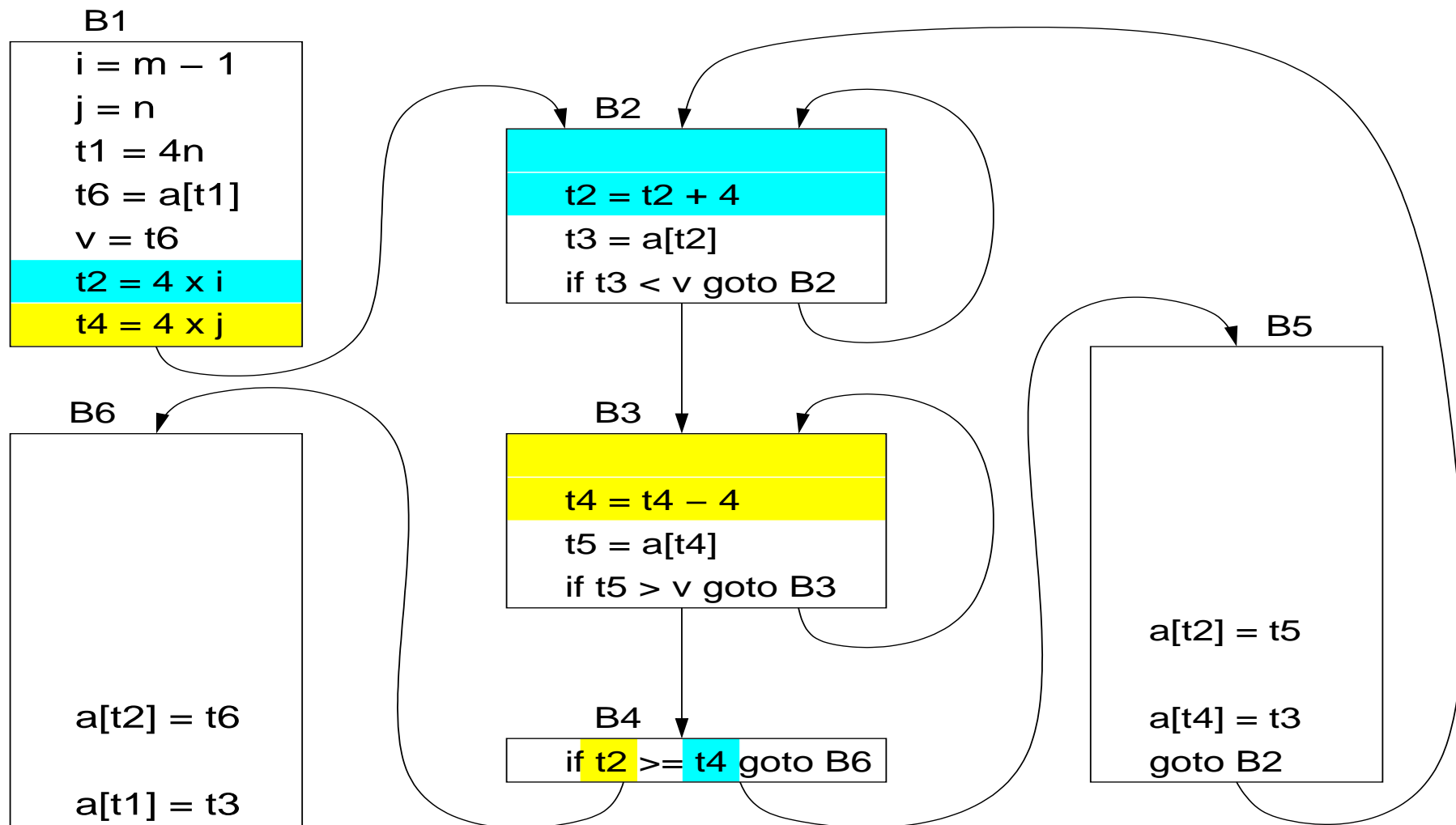
- ❖ Repeated copying of the value of an expression that stays constant in some part of a program can be avoided by **propagating the constant value** of this expression.
- ❖ An assignment of a value to a **variable such that the assigned value is never used** can be eliminated.

Elimination of Induction Variables



- ❖ Reduction of the number of **induction variables** of a loop.
- ❖ Costly operations are replaced by cheaper ones.

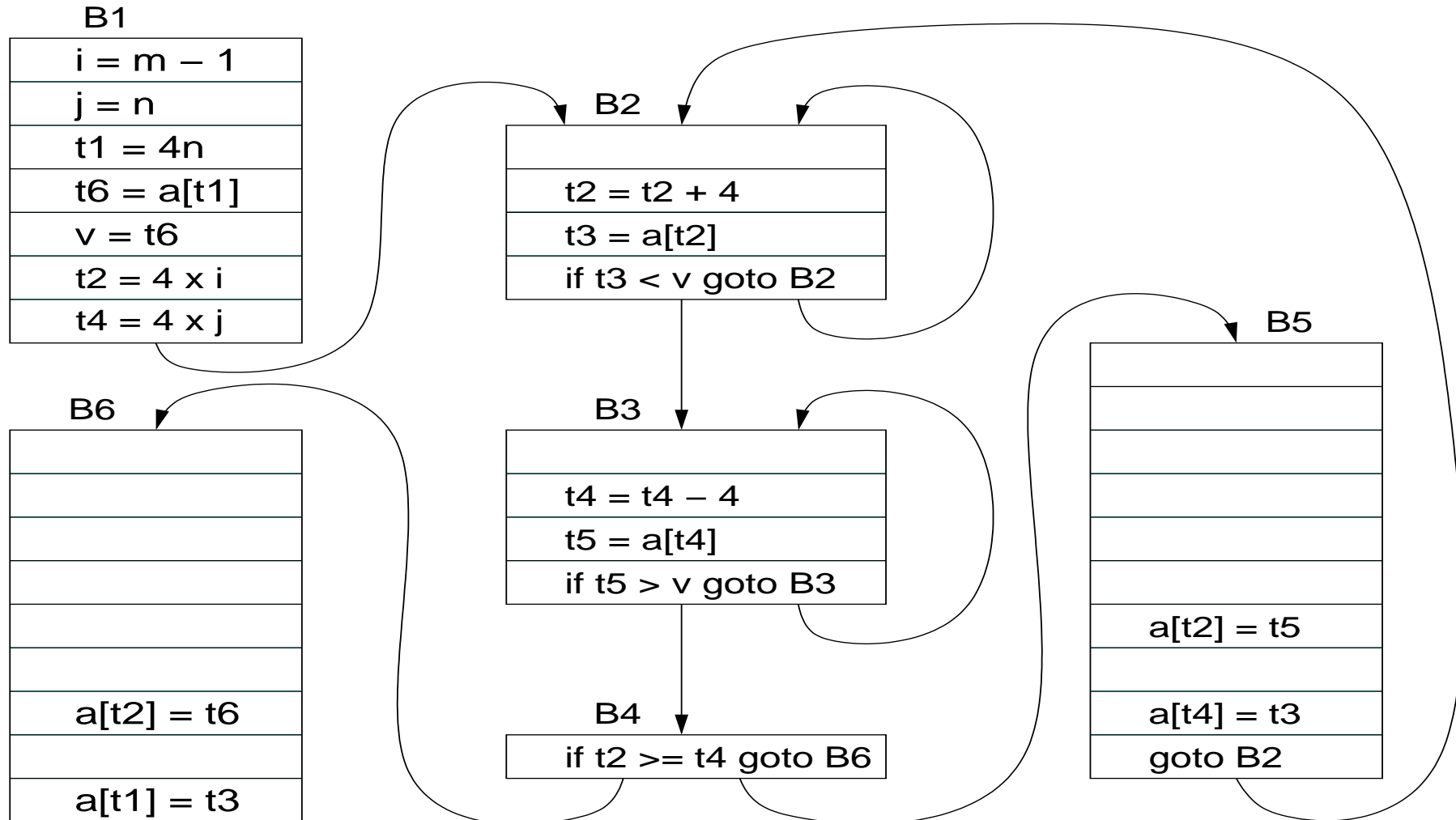
Elimination of Induction Variables



- ❖ Reduction of the number of **induction variables** of a loop.
- ❖ Costly operations are replaced by cheaper ones.

Final Code

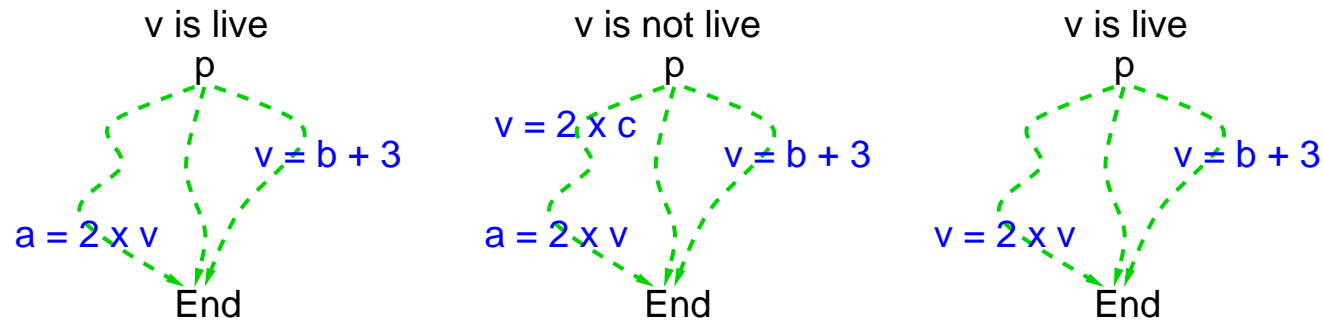
❖ Optimised intermediate code—simpler code, simplified loop bodies:



Classical Data Flow Analysis for Code Optimisation

Live Variables

❖ A variable v is live at a program point p iff some path from p to End contains an r-value occurrence (i.e., a read) of v which is not preceded by an l-value occurrence (i.e., a write).



❖ Usage:

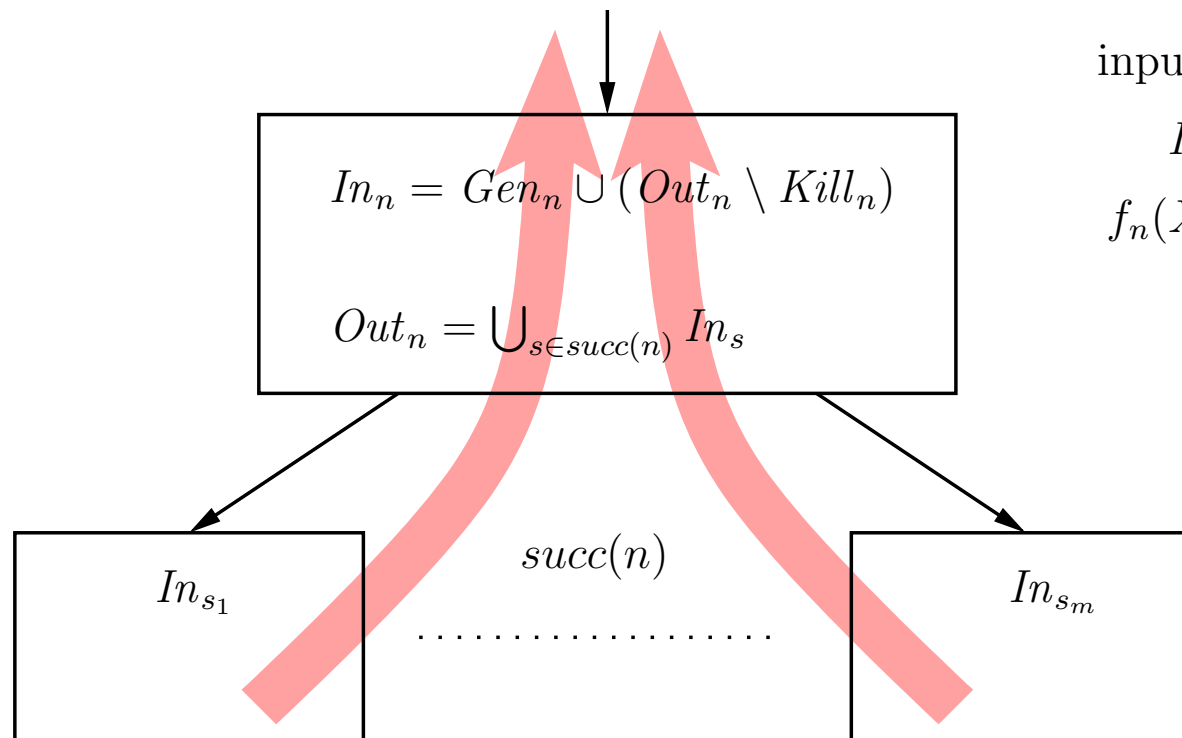
- Live variables are candidates for register allocation.
- An assignment to v at a program point where v is not live can be eliminated.

❖ How to compute live variables?

- Define local constraints that restrict sets of variables live within blocks.
- Define global equalities that relate sets of variables live in different blocks.
- Compute the most general solution that satisfies all the equalities and local constraints.
- The obtained liveness at the block boundaries is locally propagated into them.

Data Flow Analysis for Live Variables

- ❖ **Local constraints** are given by two sets of variables defined for each block:
 - Gen_n : variables used in n such that their use is not preceded by their definition.
 - $Kill_n$: variables (re)defined in n .
- ❖ Let In_n and Out_n represent sets of variables live at the input/output of each basic block n .
- ❖ Global **flow equations** define how information is propagated **backwards through the CFG**:



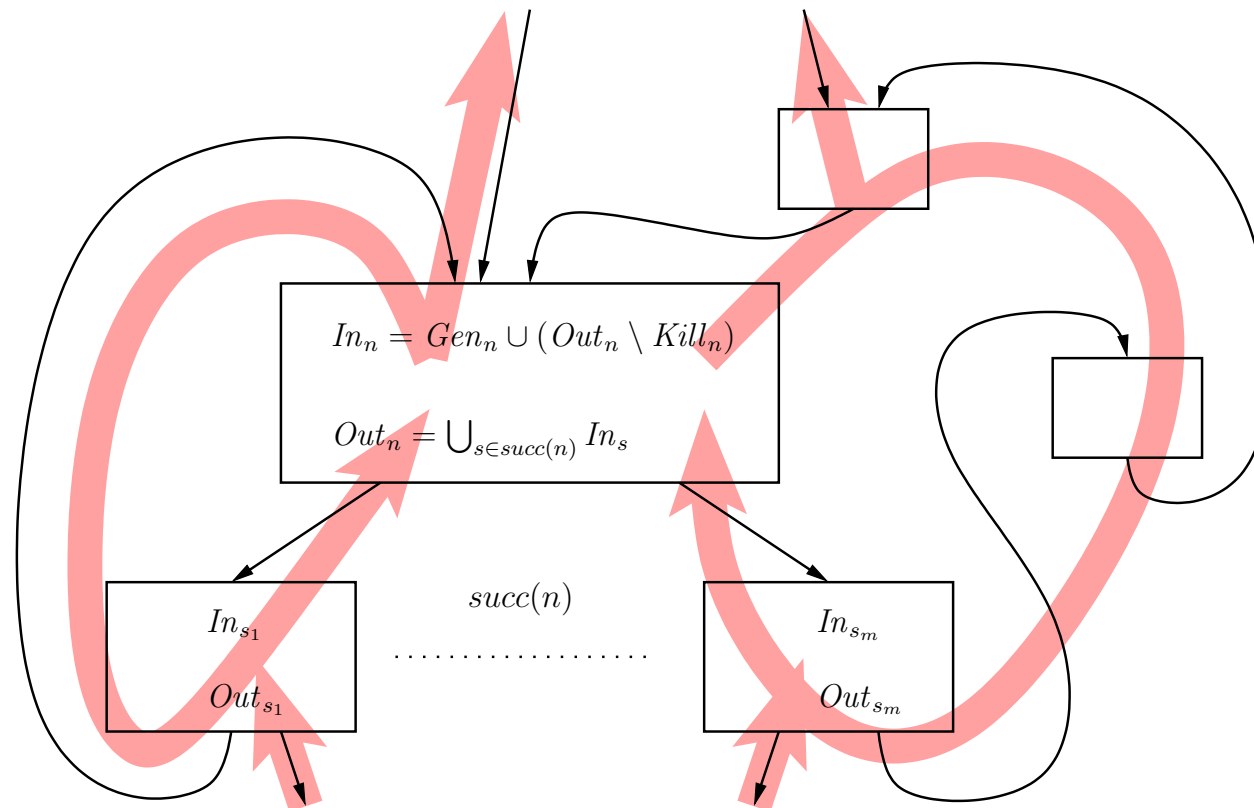
input is a function of output:

$$In_n = f_n(Out_n) \text{ where}$$
$$f_n(X) = Gen_n \cup (X \setminus Kill_n)$$

Computing Live Variables

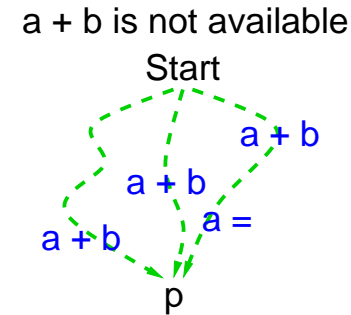
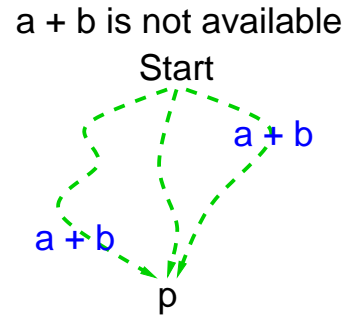
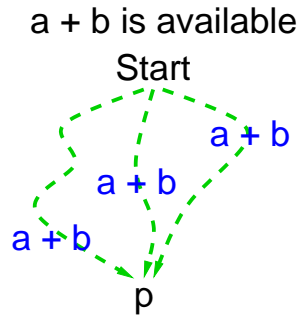
❖ A simple algorithm:

1. Compute *Kill* and *Gen* sets.
2. Start with all *Out* and *In* sets initialised to \emptyset .
3. Update *In* sets according to the current values of *Out* sets.
4. If anything changed, update *Out* sets according to the new values *In* and go to 3.

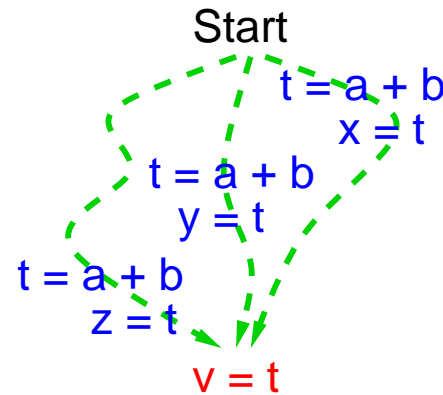
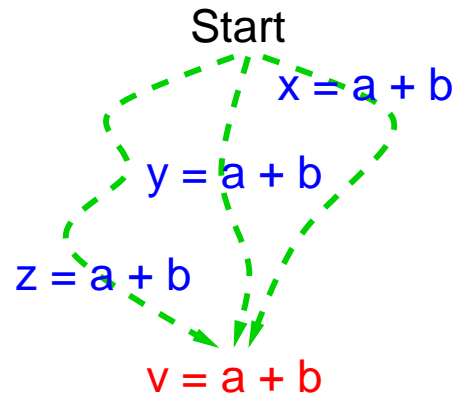


Available expressions

❖ An expression e is available at a program point p if every path from the program entry to p contains an evaluation of e which is not followed by a definition of any operand of e .



❖ Usage: common subexpression elimination.



Data Flow Analysis for Available Expressions

❖ Local constraints:

- $Gen_n = \{e \mid \text{the expression } e \text{ is evaluated in the basic block } n, \text{ and this evaluation is not followed by a definition of any operand of } e\}.$
- $Kill_n = \{e \mid \text{the basic block } n \text{ contains a definition of an operand of } e\}.$

❖ Contrary to live variables, the information propagates **forwards through the CFG**.

❖ Flow equations:

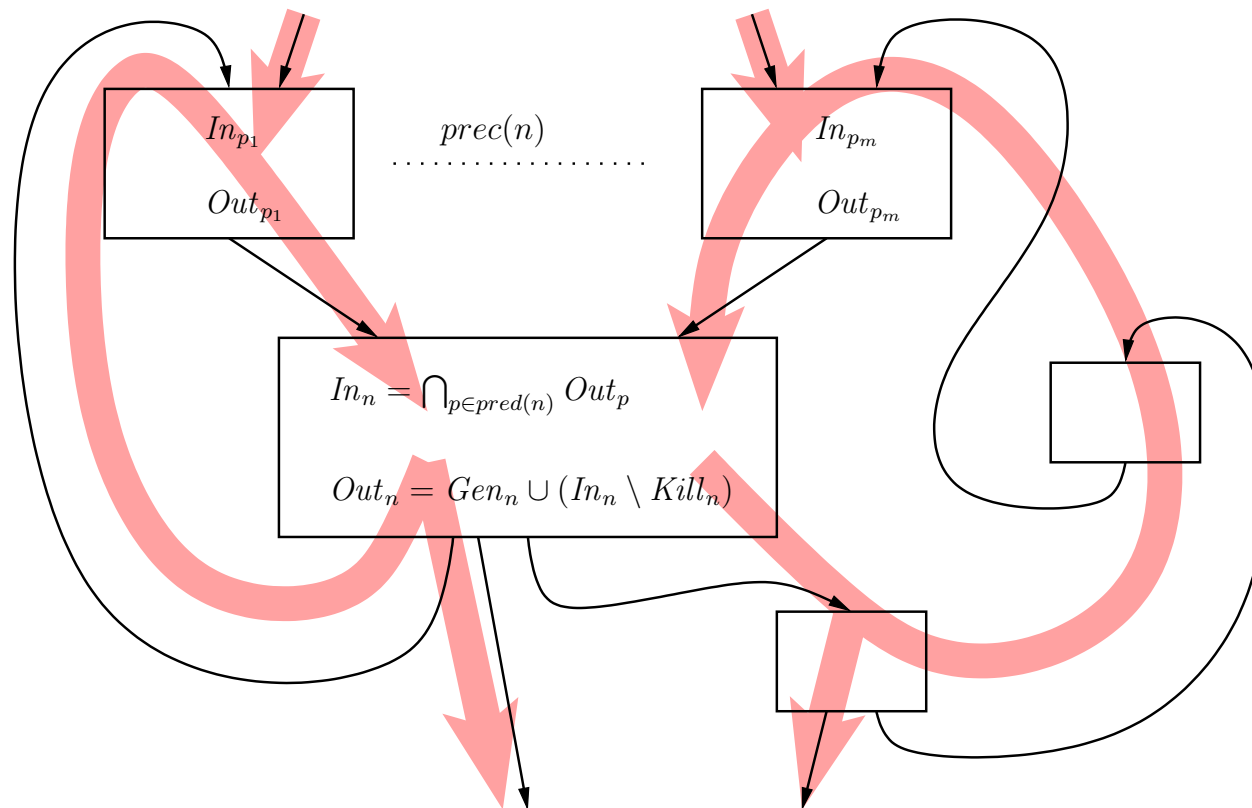
$$In_n = \begin{cases} \emptyset & \text{if } n \text{ is the } Start \text{ block} \\ \bigcap_{p \in pred(n)} Out_p & \text{otherwise} \end{cases}$$

output is a function of input:

$$Out_n = f_n(In_n) \text{ where} \\ f_n(X) = Gen_n \cup (X \setminus Kill_n)$$

Computing Available Expressions

- ❖ A **simple algorithm** (let $Exps$ be the set of all expressions in the program at hand):
 1. Compute $Kill$ and Gen sets.
 2. Start with all Out and In sets initialised to $Exps$ (In of the start block is initialised to \emptyset).
 3. Update Out sets according to the current values of In sets.
 4. If anything changed, update In sets according to the new values Out and go to 3.



Towards a General Framework

- ❖ We are going to formulate data flow analyses in a more **abstract and generic way** in terms of **lattices and monotone functions**.

- ❖ Sets of **data flow values** that may be associated with blocks as their *Gen/Kill/In/Out* values can be seen as **partially ordered sets** wrt. some ordering \sqsubseteq .

- ❖ The order \sqsubseteq reflects the **information content** of the particular values depending on how the computed information is to be used.
 - **Live variables:**
 - The fewer variables are live, the more assignments we can eliminate.
 - Hence, $V_1 \subseteq V_2 \iff V_1 \supseteq V_2$.
 - Use the **complete lattice** $(2^{Vars}, \supseteq)$.

 - **Available expressions:**
 - The more expressions are available, the more common subexpressions we can eliminate.
 - Hence, $E_1 \subseteq E_2 \iff E_1 \sqsubseteq E_2$.
 - Use the **complete lattice** $(2^{Exps}, \subseteq)$.

Towards a General Framework

❖ Local propagation of information within blocks:

- **Live variables**—bottom up, against the control flow:
 - $In_n = f_n(Out_n)$ where $f_n(X) = Gen_n \cup (X \setminus Kill_n)$,
 - f_n is **monotone** in $(2^{Vars}, \supseteq)$ ($V_1 \supseteq V_2 \implies f_n(V_1) \supseteq f_n(V_2)$).
- **Available expressions**—top down, along the control flow:
 - $Out_n = f_n(In_n)$ where $f_n(X) = Gen_n \cup (X \setminus Kill_n)$,
 - f_n is **monotone** in $(2^{Exps}, \subseteq)$ ($E_1 \subseteq E_2 \implies f_n(E_1) \subseteq f_n(E_2)$).

Towards a General Framework

❖ Local propagation of information within blocks:

- **Live variables**—bottom up, against the control flow:
 - $In_n = f_n(Out_n)$ where $f_n(X) = Gen_n \cup (X \setminus Kill_n)$,
 - f_n is **monotone** in $(2^{Vars}, \supseteq)$ ($V_1 \supseteq V_2 \implies f_n(V_1) \supseteq f_n(V_2)$).
- **Available expressions**—top down, along the control flow:
 - $Out_n = f_n(In_n)$ where $f_n(X) = Gen_n \cup (X \setminus Kill_n)$,
 - f_n is **monotone** in $(2^{Exps}, \subseteq)$ ($E_1 \subseteq E_2 \implies f_n(E_1) \subseteq f_n(E_2)$).

❖ Confluence—merging information of two or more flow paths and computing the maximal information consistent with all the incoming edges:

- **Live variables**—exists some path ... :
 - $Out_n = \bigcup_{s \in succ(n)} In_s$,
 - \cup is the **meet** in $(2^{Vars}, \supseteq)$.
- **Available expressions**—for all paths ... :
 - $In_n = \bigcap_{p \in pred(n)} Out_p$,
 - \cap is the **meet** $(2^{Exps}, \subseteq)$.

Monotone Framework

❖ Ingredients:

- A **complete lattice** (L, \sqsubseteq) of flow values with explicitly stated \top , \perp , and \sqcap . It has to satisfy the **descending chain condition**: every descending chain $x_1 \sqsupseteq x_2 \sqsupseteq \dots$ eventually stabilises (after a finite number of steps).
- A set \mathcal{F} of **flow functions** $L \rightarrow L$. Every basic block n is mapped to its flow function f_n . \mathcal{F} is closed under composition, the functions are **monotone**.
- A **data flow** F : against/along the CFG.
- An **extreme flow variable** E ($In/Out_{Start/End}$) and its **value** val_E .

	Available Expressions	Live Variables
L	2^{Exps}	2^{Vars}
\sqsubseteq	\subseteq	\supseteq
\sqcap	\cap	\cup
\top, \perp	$Exps, \emptyset$	$\emptyset, Vars$
E	In_{Start}	Out_{End}
val_E	\emptyset	\emptyset
F	along control flow	against control flow
\mathcal{F}	$\{f : L \rightarrow L \mid \exists l_k, l_g : f(l) = (l \setminus l_k) \cup l_g\}$	
f_n	$f_n(X) = Gen_n \cup (X \setminus Kill_n)$	

Performing the Analysis

- ❖ The Goal: The **most exhaustive/informative assignment** of flow values to blocks that is safe (consistent with all equations) called as the **maximum fixpoint (MFP) solution**.

- ❖ **Initialise** all values to \top , set the value of the extremal label E to val_E .

- ❖ **Round Robin**—simple, not efficient:
 - Traverse all blocks in a fixed order.
 - Always apply the flow function (f_n) and propagate along the flow F using \sqcap .
 - Terminate when all values stabilise.

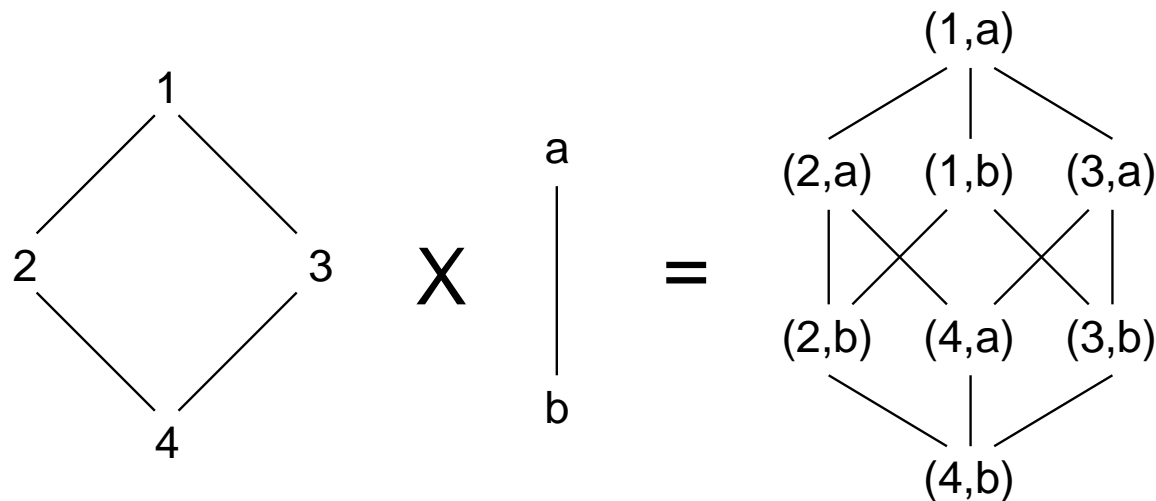
- ❖ Use a **worklist**—faster, but with a memory overhead:
 - Maintain a worklist of blocks with changed values (initially, put all blocks there).
 - Keep processing elements of the worklist until it is empty:
 - Update a block n from the list locally using f_n and erase it from the list.
 - If its value changed, propagate the change to its successors wrt. F using \sqcap .
 - Add the updated successors to the worklist.

Cartesian Product of Lattices

❖ In order to sketch **correctness of the iterative solution of flow equations**, we need to be able to deal with flow values for each control block, i.e., with vectors of flow values. This leads us to a need of Cartesian product lattices.

❖ A **Cartesian product of lattices** $(L_1, \sqsubseteq_1), \dots, (L_n, \sqsubseteq_n)$ is defined as the pair (L, \sqsubseteq) where:

- $L = L_1 \times \dots \times L_n$ and
- $\sqsubseteq \subseteq L \times L$ is defined as $(x_1, \dots, x_n) \sqsubseteq (y_1, \dots, y_n)$ iff $x_1 \sqsubseteq_1 y_1, \dots, x_n \sqsubseteq_n y_n$.



❖ Then:

- (L, \sqsubseteq) is a **lattice** where $(x_1, \dots, x_n) \sqcap (y_1, \dots, y_n) = (x_1 \sqcap_1 y_1, \dots, x_n \sqcap_n y_n)$.
- The product of complete lattices is a **complete lattice**.
- The product of lattices satisfying the descending chain condition satisfies the **descending chain condition**.

Correctness of Round Robin

- ❖ To simplify, assume that Round Robin always first computes all meets and then applies all flow functions.
- ❖ **Statement:** Round Robin always terminates with the maximum fixpoint solution (MFP).
- ❖ **Idea of the proof:**
 - Consider k basic blocks, giving $2k$ *In/Out* flow variables. A **global state of the whole analysis** is a $2k$ -tuple of flow values, an element of the classical **Cartesian product** of $2k$ lattices of flow values.
 - The product lattice inherits the important properties of the flow value lattice. It is also **complete** and satisfies the **descending chain condition**.
 - Consider a function $\mathbf{f} : L^{2k} \rightarrow L^{2k}$ applying all the flow functions. It is **monotone**.
 - Consider a function $\mathbf{f}_{\sqcap} : L^{2k} \rightarrow L^{2k}$ computing all the meets. It is also **monotone**.
 - Now, notice that:
 - $\mathbf{f}_{\circ} = \mathbf{f} \circ \mathbf{f}_{\sqcap}$ is **monotone too**. It corresponds to one iteration of the algorithm.
 - Apparently, **solutions of the analyses are the fixpoints of \mathbf{f}_{\circ}** .
 - Take **Knaster-Tarski with the descending chain condition**.
 - And you shall see the light ...

Very Busy Expressions/Anticipable

- ❖ An expression e is very busy/anticipable at a program point p if every path from p to exit contains an evaluation of e which is not preceded by a redefinition of an operand of e .
- ❖ Usage: safety of code hoisting—moving code that occurs in all branches above the branching.
- ❖ The corresponding data flow analysis:
 - Lattice of data flow values $(2^{Exps}, \subseteq)$ with $(\top, \perp, \sqcap) = (Exps, \emptyset, \cap)$.
 - Flow F : against the control flow.
 - flow functions: $f_n(X) = Gen_n \cup (X \setminus Kill_n)$.
 - Extreme flow variable: Out_{End} .
 - Initial value of the extreme flow variable: \emptyset .
 - $Gen_n = \{e \mid \text{the expression } e \text{ is evaluated in the basic block } n, \text{ and this evaluation is not preceded (within } n) \text{ by a definition of any operand of } e\}$.
 - $Kill_n = \{e \mid \text{the basic block } n \text{ contains a definition of an operand of } e\}$.

Reaching Definitions

- ❖ A definition $dx : x = y$ reaches a program point p if it appears (without a redefinition of x) on some path from the program entry to p .
- ❖ Usage: **copy propagation**—a use of a variable x at a program point p can be replaced by y if $dx : x = y$ is the only definition of x which reaches p , and y is not modified between dx and p .
- ❖ The corresponding **data flow analysis**:
 - Lattice of data flow values: $(2^{Vars \times Labs}, \supseteq)$ with $(\top, \perp, \sqcap) = (\emptyset, Vars \times Labs, \cup)$.
 - Flow F : **along the control flow**.
 - Flow equations: $f_n(X) = Gen_n \cup (X \setminus Kill_n)$.
 - Extreme flow variable: $Start_{In}$.
 - Initial value of the extreme flow variable: $\{d_x : x = undef \mid x \in Vars\}$.
 - $Gen_n = \{d_v \mid \text{the variable } v \text{ is defined in the basic block } n, \text{ and this definition is not followed (within } n) \text{ by a definition of } v\}$.
 - $Kill_n = \{d_v \mid \text{the basic block } n \text{ contains a definition of } v\}$.

Summary of the Presented Analyses

	Available Expressions	Live Variables	Very Busy Expressions	Reaching Definitions
L	2^{Exps}	2^{Vars}	2^{Exps}	$2^{Vars \times Labs}$
\sqsubseteq	\subseteq	\supseteq	\subseteq	\supseteq
\sqcap	\cap	\cup	\cap	\cup
\top, \perp	$Exps, \emptyset$	$\emptyset, Vars$	$Exps, \emptyset$	$\emptyset, Vars \times Labs$
E	In_{Start}	Out_{End}	Out_{End}	In_{Start}
val_E	\emptyset	\emptyset	\emptyset	$\{d_x : x = undef \mid x \in Vars\}$
F	control flow	reversed control flow	reversed control flow	control flow
\mathcal{F}	$\{f : L \rightarrow L \mid \exists l_k, l_g : f(l) = (l \setminus l_k) \cup l_g\}$			
f_n	$f_n(X) = Gen_n \cup (X \setminus Kill_n)$			

❖ These analyses are often referred to as **bit-vector analyses** since the flow values, which are subsets of a finite set, can be represented as bit-vectors (for each control point).

- Do not confuse with the vectors that one obtains when speaking about flow values for all control points—then we get vectors of vectors.

General Data Flow Analysis

Data Flow Equations For General Flows

- ❖ So far, flow functions based on constant *Gen* and *Kill* sets, independent of the input of the flow functions (and hence of the current values of the flow variables), were considered.
- ❖ However, in general, *Gen* and *Kill* can be functions that have a flow independent part but that can also have a flow dependent part:

$$Gen_n(X) = ConstGen_n \cup DepGen_n(X)$$

$$Kill_n(X) = ConstKill_n \cup DepKill_n(X)$$

$$f_n(X) = Gen_n(X) \cup (X \setminus Kill_n(X))$$

Data Flow Equations For General Flows

- ❖ So far, flow functions based on constant *Gen* and *Kill* sets, independent of the input of the flow functions (and hence of the current values of the flow variables), were considered.
- ❖ However, in general, *Gen* and *Kill* can be functions that have a flow independent part but that can also have a flow dependent part:

$$Gen_n(X) = ConstGen_n \cup DepGen_n(X)$$

$$Kill_n(X) = ConstKill_n \cup DepKill_n(X)$$

$$f_n(X) = Gen_n(X) \cup (X \setminus Kill_n(X))$$

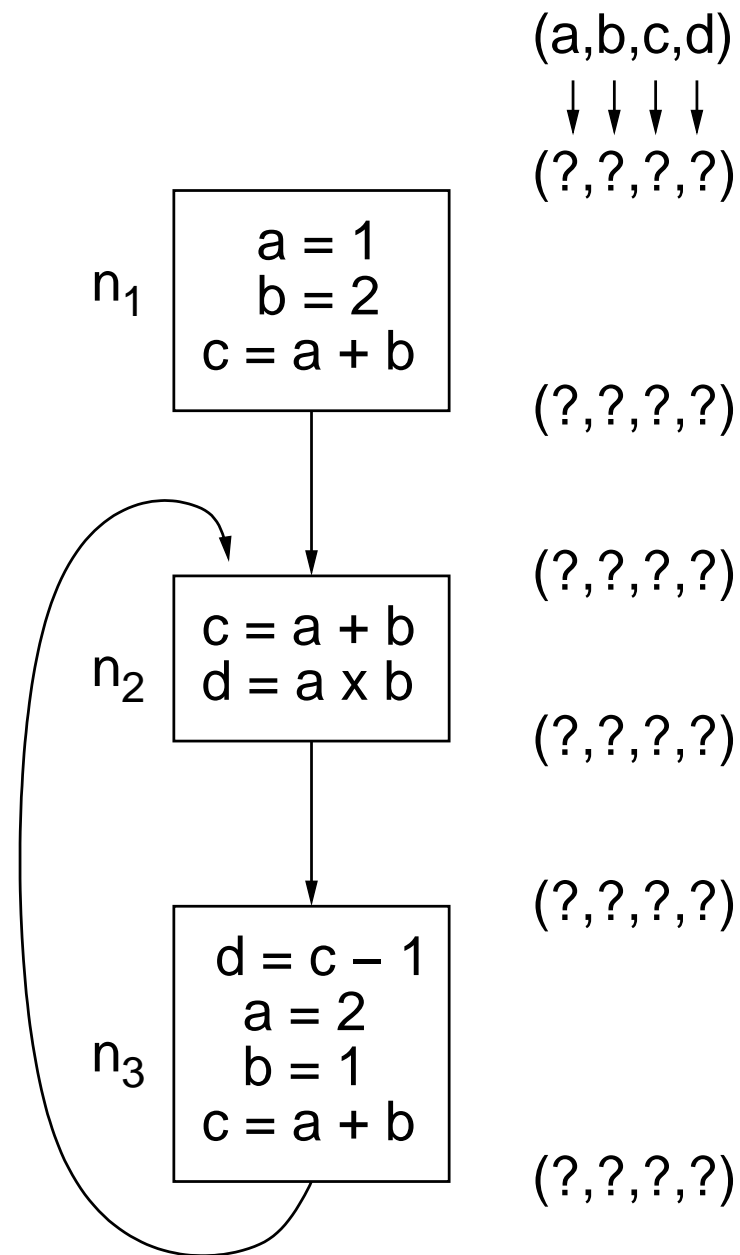
- ❖ Note that the data flow values associated with the program entities whose properties are tracked by an analysis (variables, expressions, ...) may depend on the old flow values associated with them only—the so-called **separable case**, but also on any other entities—the **non-separable case**.

- All the so far considered analyses were separable.
- Examples of non-separable analyses include **faint variable analysis** (transitive liveness of variables), **possibly uninitialised variable analysis** (again with transitive dependencies), or **constant propagation analysis**.
- For non-separable analyses, flow equations are often defined for particular statements (instead of basic blocks).

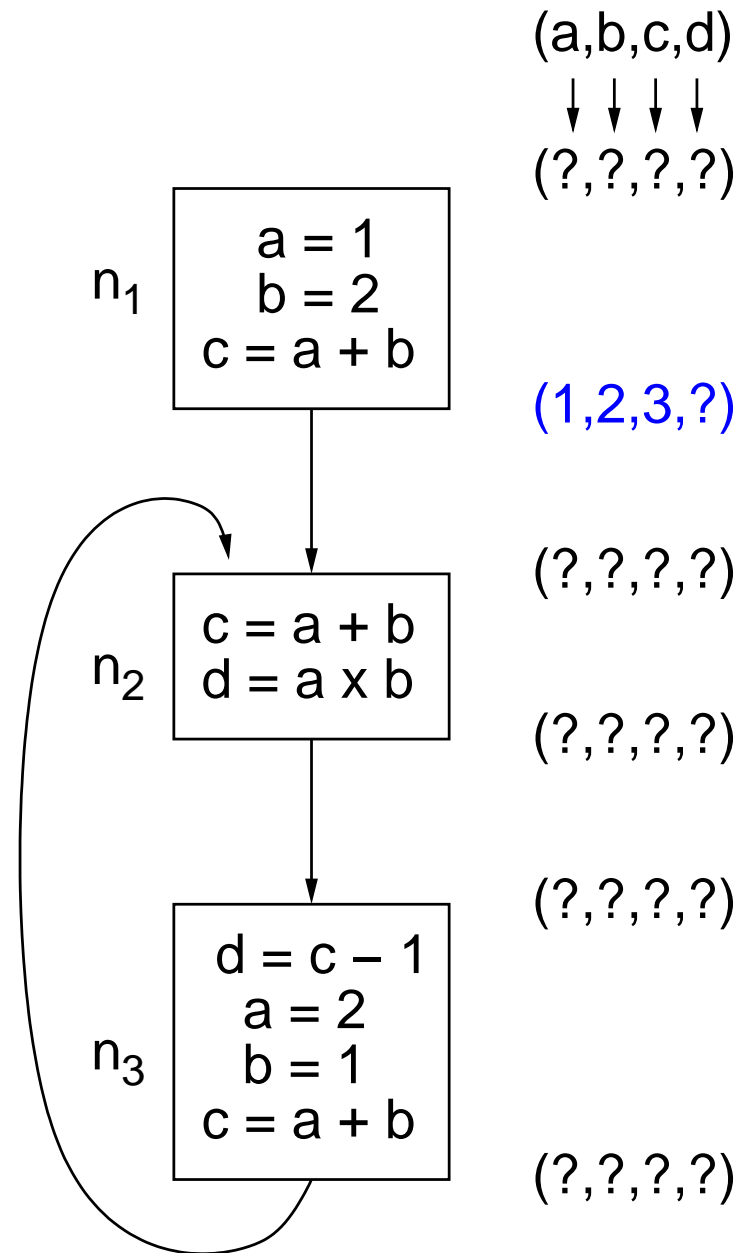
Constant Propagation

- ❖ A value of a **variable v is constant** at a program point p if the value of the variable v will be the same at this point regardless of the program path taken to p .
- ❖ Usage: **Replacing constant variables with constants.**
- ❖ The analysis is interesting from several points of view:
 - **non-separable**: clearly, to compute the value of an expression assigned to some variable, we need values of the variables that appear in the expression,
 - **not a bit-vector analysis**: at least when considering a set of infinitely many different constant values, we have infinitely many possible flow values,
 - **non-distributive analysis**: the MFP solution needs not be the most precise one (to be discussed later on).

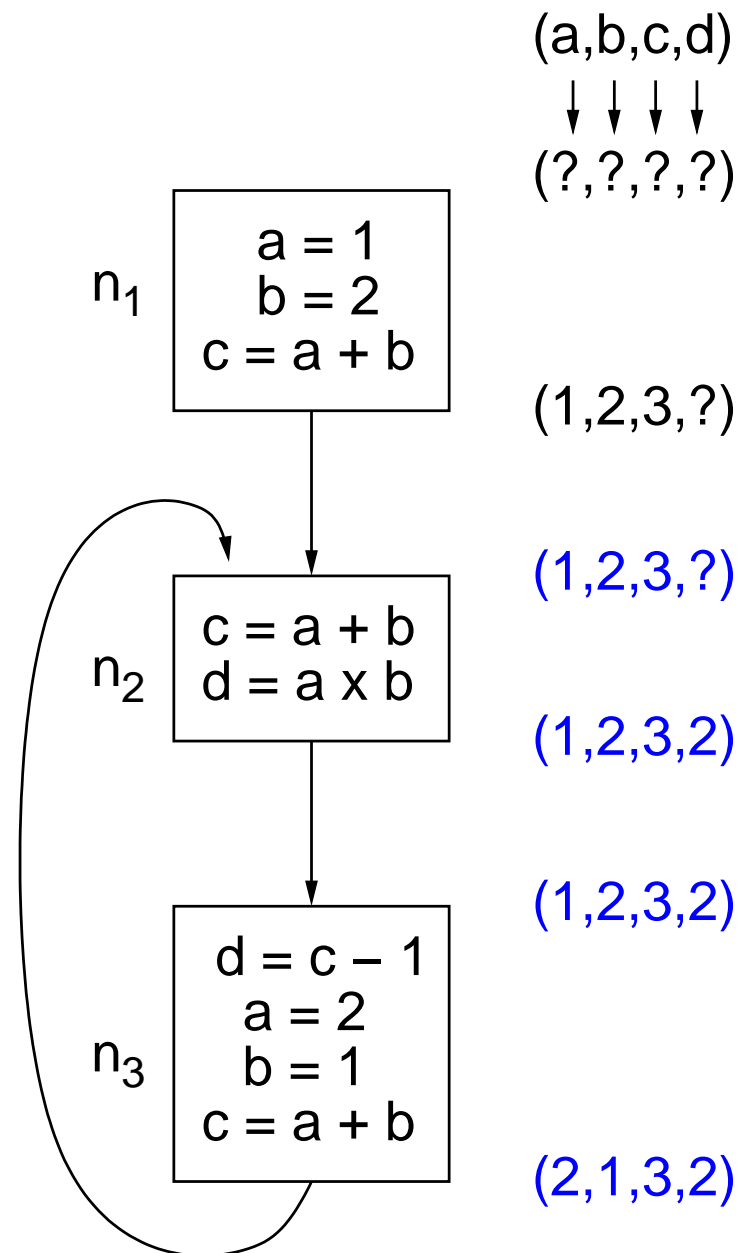
Constant Propagation Movie



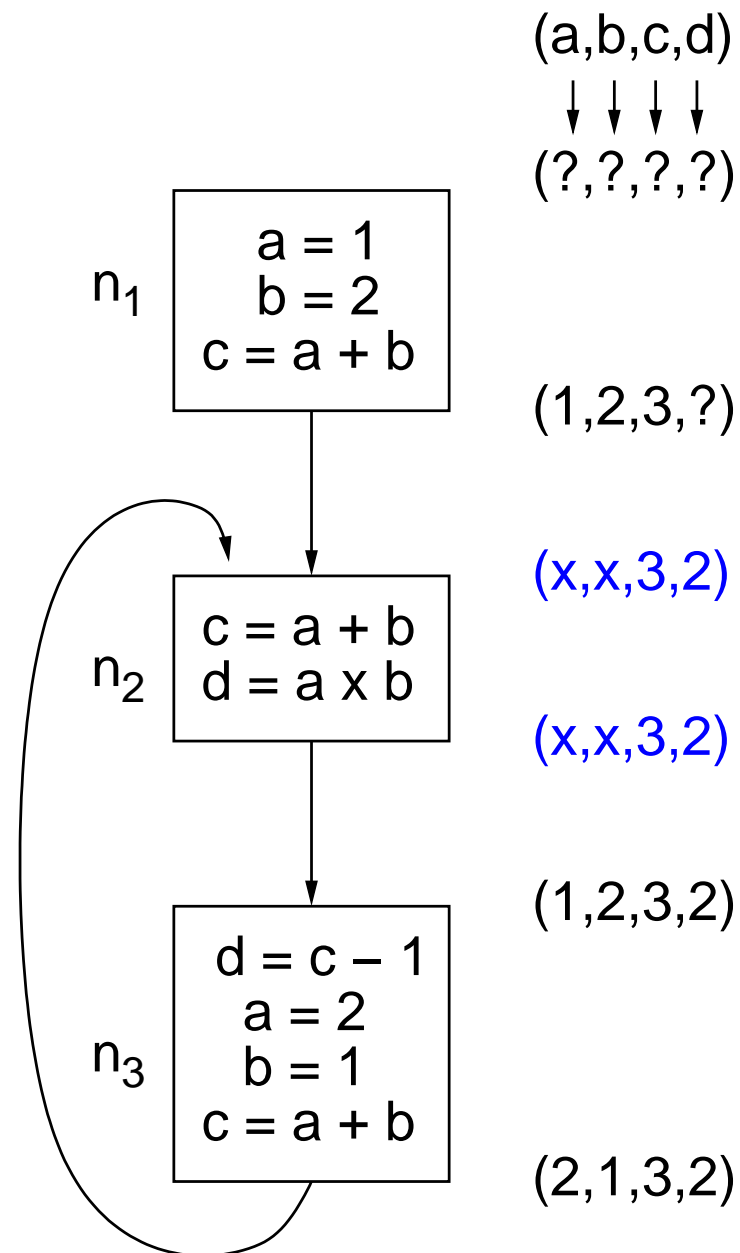
Constant Propagation Movie



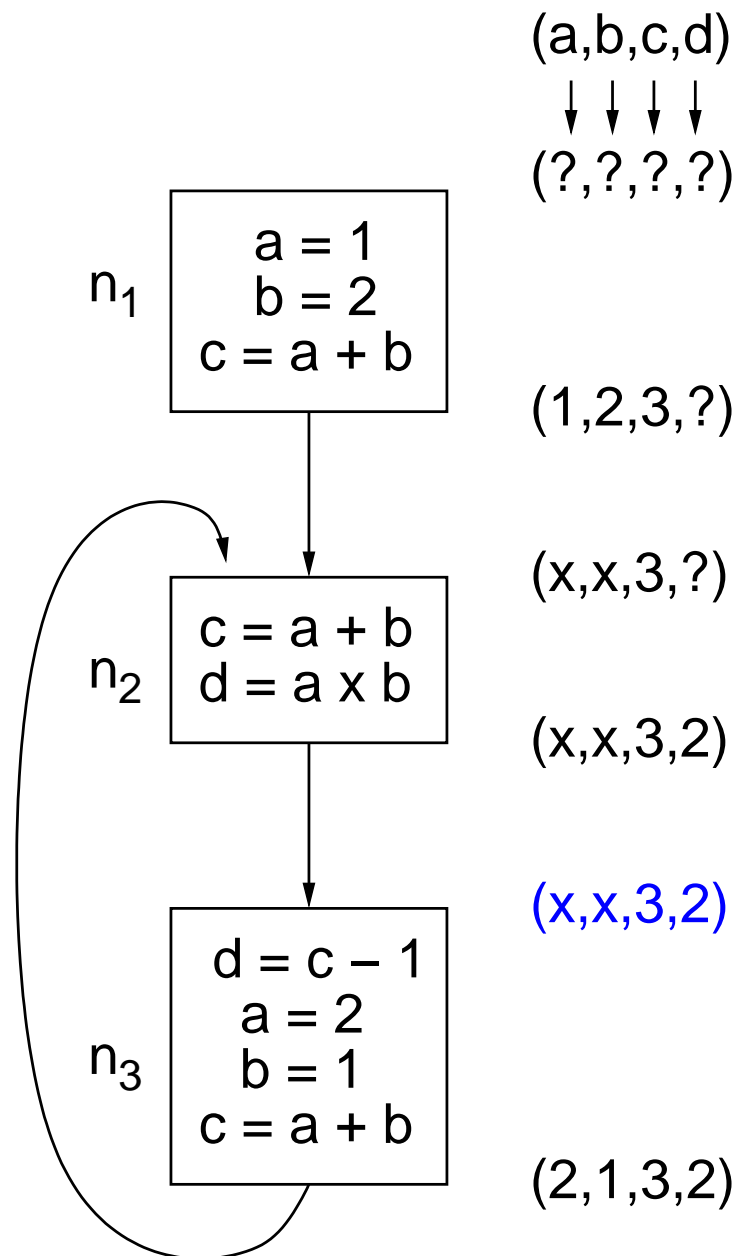
Constant Propagation Movie



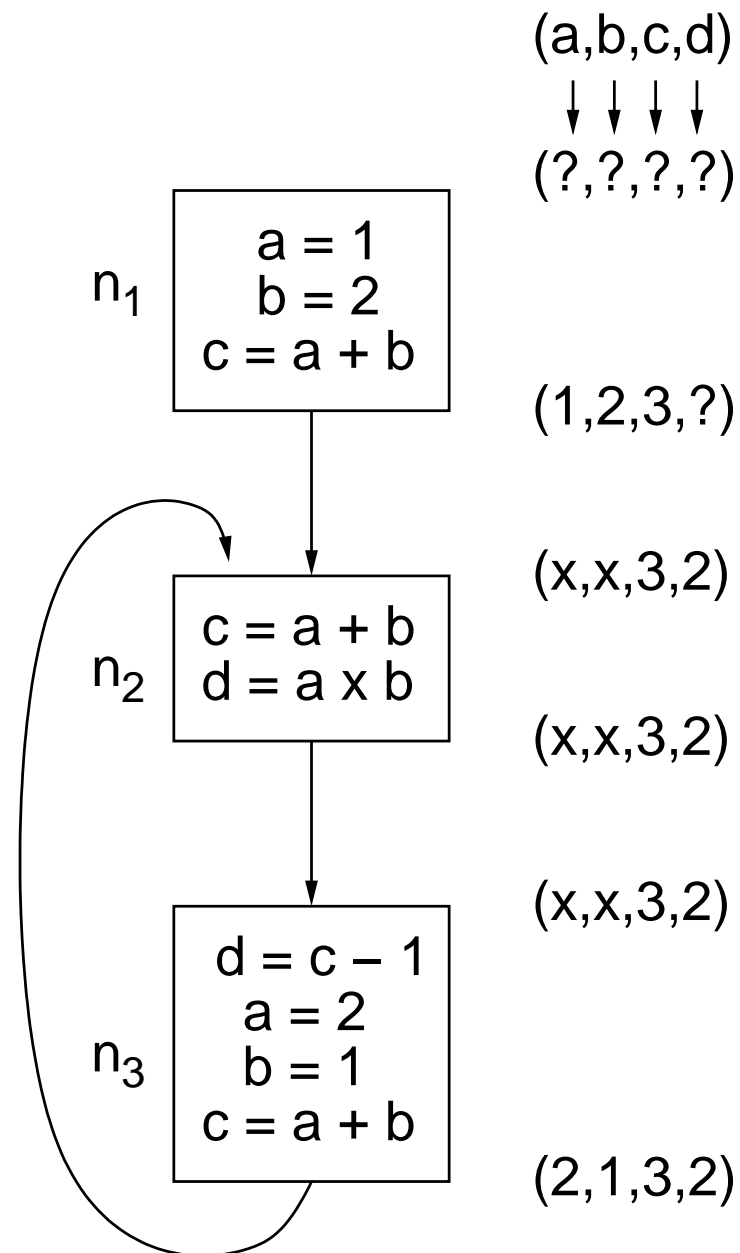
Constant Propagation Movie



Constant Propagation Movie



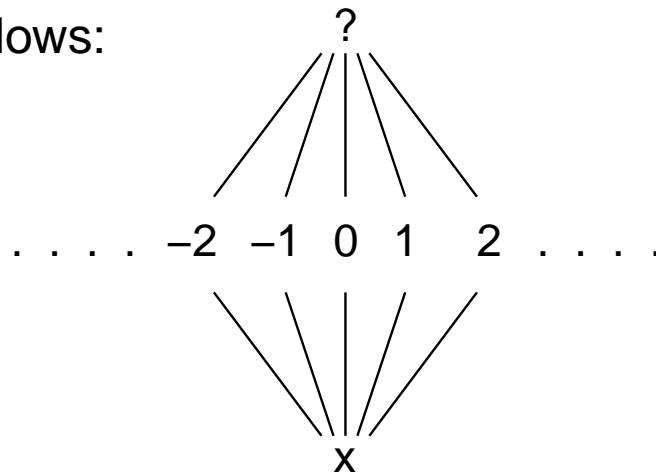
Constant Propagation Movie



Flow Lattice for (Integer) Constant Propagation

❖ In order to track flow values associated with particular variables, the **component lattice** ($\hat{L} = \mathbb{Z} \cup \{?, x\}, \hat{\subseteq}$) can be used where

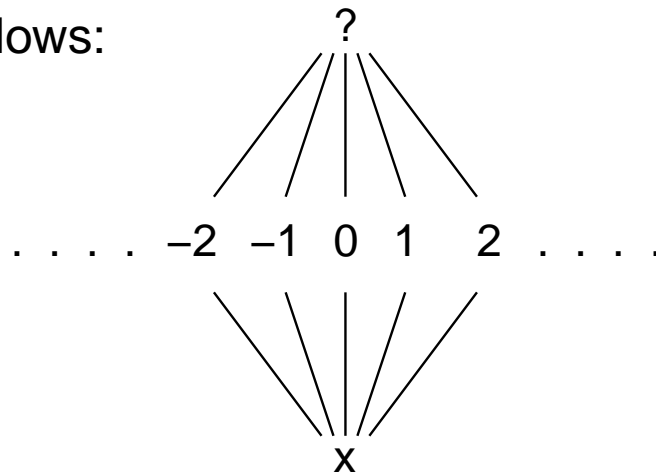
- the maximum $\hat{\top} = ?$ is the **undefined value**, i.e., “anything”, and
- the minimum $\hat{\perp} = x$ is a an indication that the given variable is **non-constant**, and
- the **ordering** $\hat{\subseteq}$ is as follows:



Flow Lattice for (Integer) Constant Propagation

❖ In order to track flow values associated with particular variables, the **component lattice** $(\hat{L} = \mathbb{Z} \cup \{?, x\}, \hat{\sqsubseteq})$ can be used where

- the maximum $\hat{\top} = ?$ is the **undefined value**, i.e., “anything”, and
- the minimum $\hat{\perp} = x$ is a an indication that the given variable is **non-constant**, and
- the **ordering** $\hat{\sqsubseteq}$ is as follows:



❖ The data flow value lattice (L, \sqsubseteq) can then be obtained as the **product of k component lattices** where k is the number of variables (\sqsubseteq, \sqcap are defined by $\hat{\sqsubseteq}, \hat{\sqcap}$).

Flow Equations for Constant Propagation

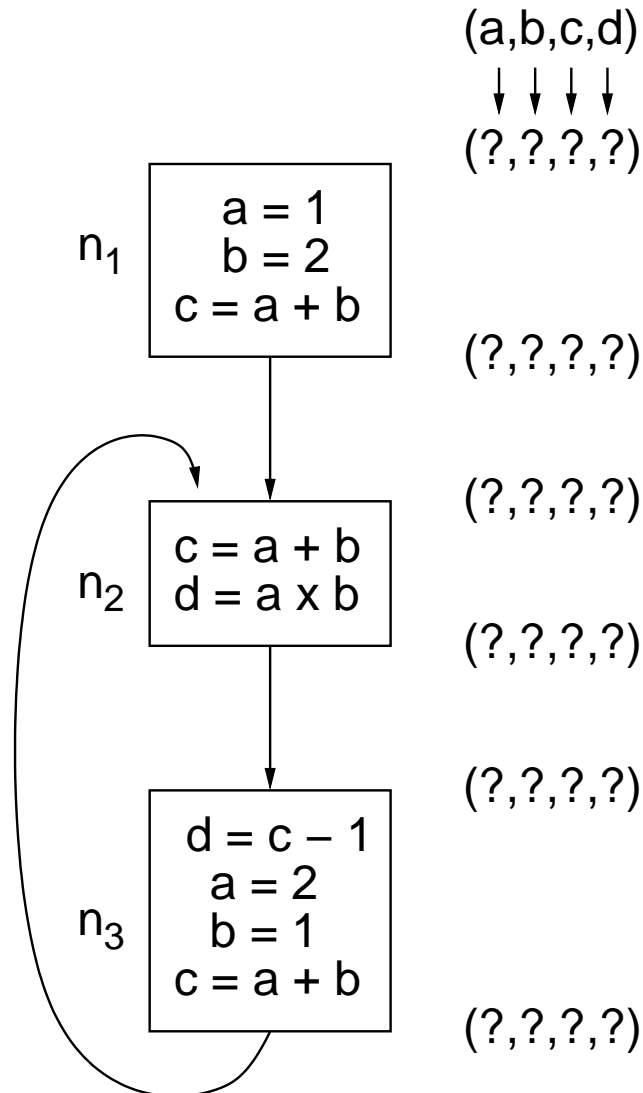
	$ConstGen_n$	$DepGen_n(X)$	$ConstKill_n$	$DepKill_n(X)$
$v = c,$ $c \in Const$	$\{(v, c)\}$	\emptyset	\emptyset	$\{(v, d) \mid (v, d) \in X\}$
$v = e,$ $e \in Exps$	\emptyset	$\{(v, eval(e, X))\}$	\emptyset	$\{(v, d) \mid (v, d) \in X\}$
$read(v)$	$\{(v, x)\}$	\emptyset	\emptyset	$\{(v, d) \mid (v, d) \in X\}$
other	\emptyset	\emptyset	\emptyset	\emptyset

$eval(a_1 \text{ op } a_2, X)$	$(a_1, ?) \in X$	$(a_1, x) \in X$	$(a_1, c_1) \in X$
$(a_2, ?) \in X$?	x	?
$(a_2, x) \in X$	x	x	x
$(a_2, c_2) \in X$?	x	$c_1 \text{ op } c_2$

Meet over Paths Solution and Distributivity

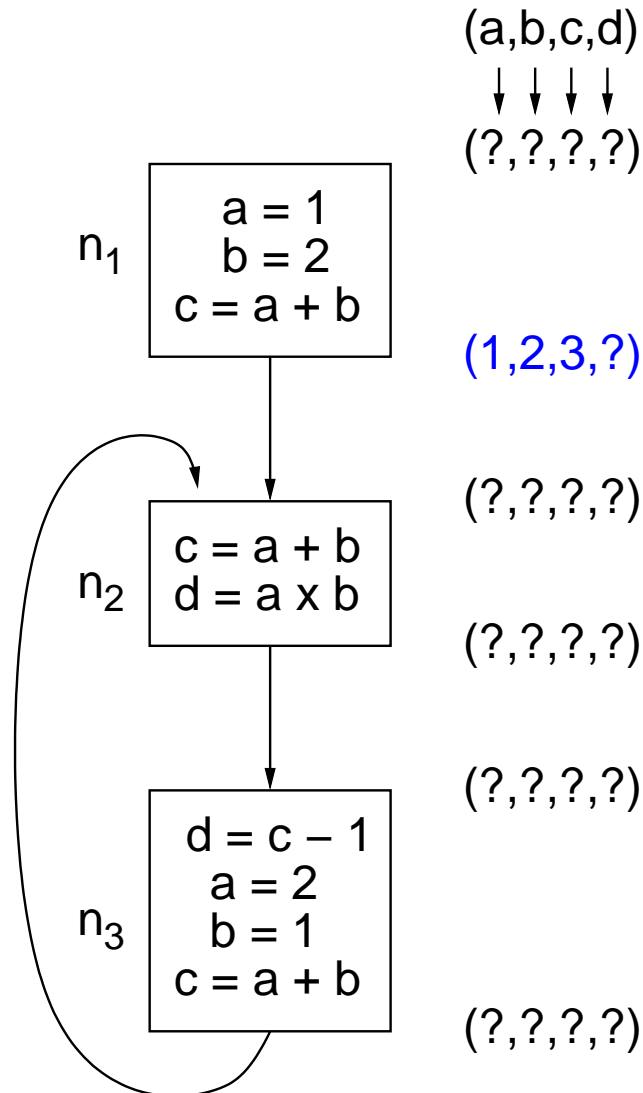
Constant Propagation Movie 2

We proceed carefully according to the defined framework.



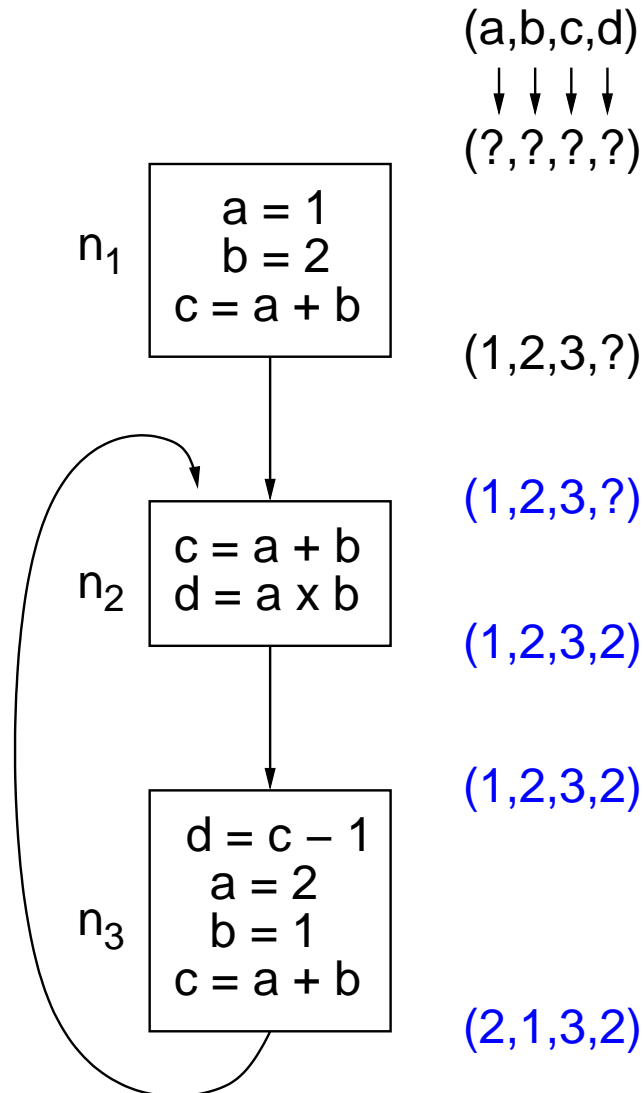
Constant Propagation Movie 2

We proceed carefully according to the defined framework.



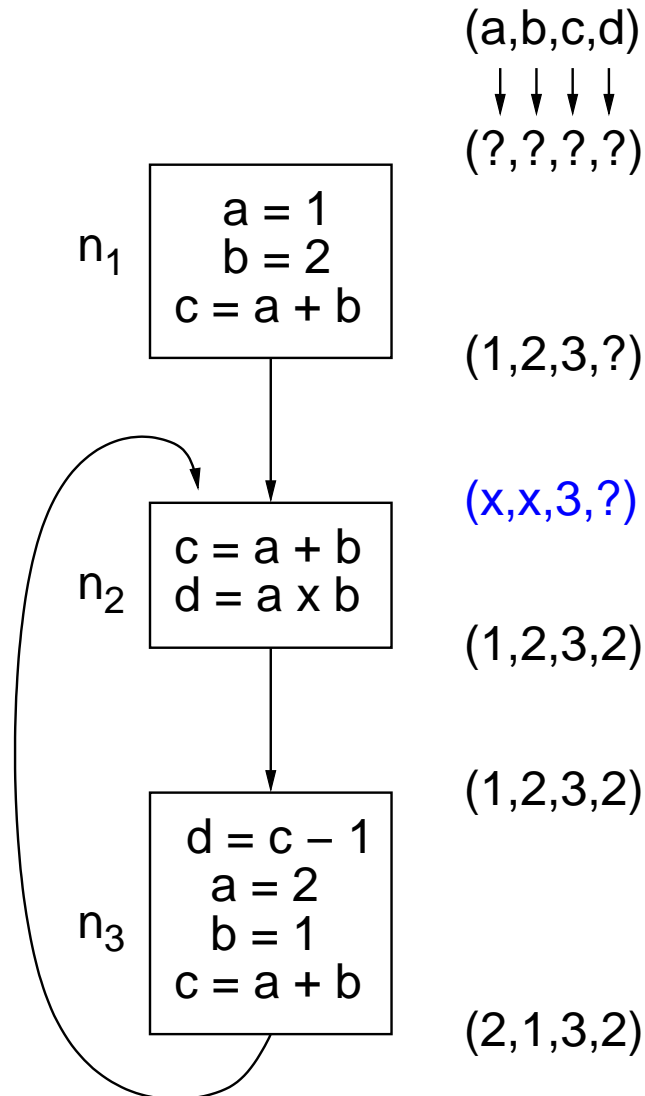
Constant Propagation Movie 2

We proceed carefully according to the defined framework.



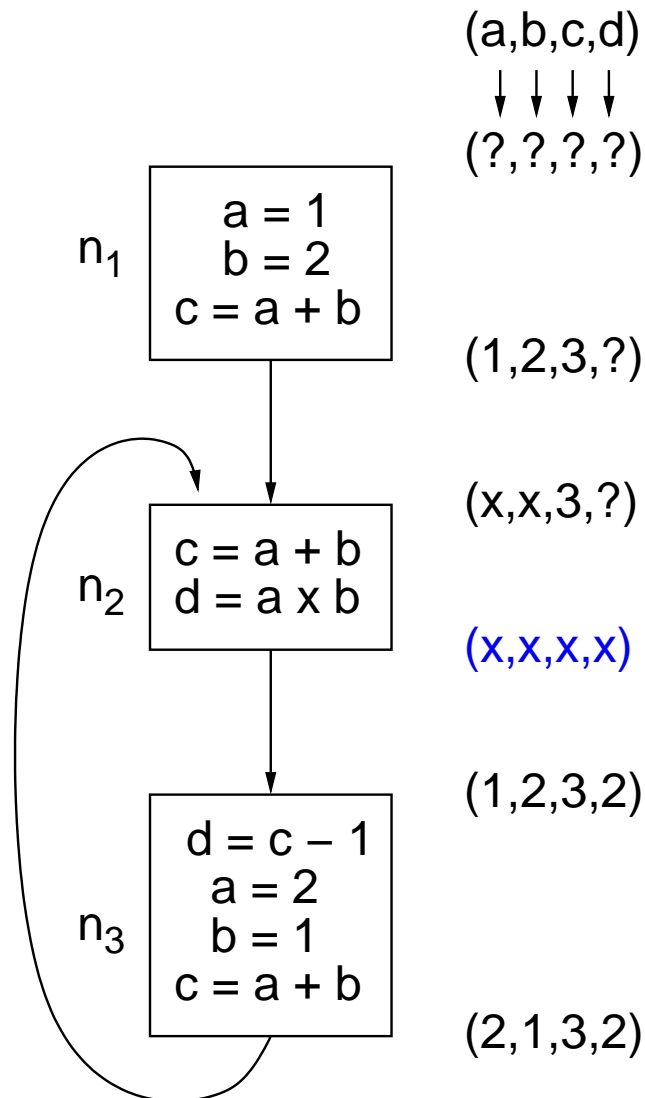
Constant Propagation Movie 2

We proceed carefully according to the defined framework.



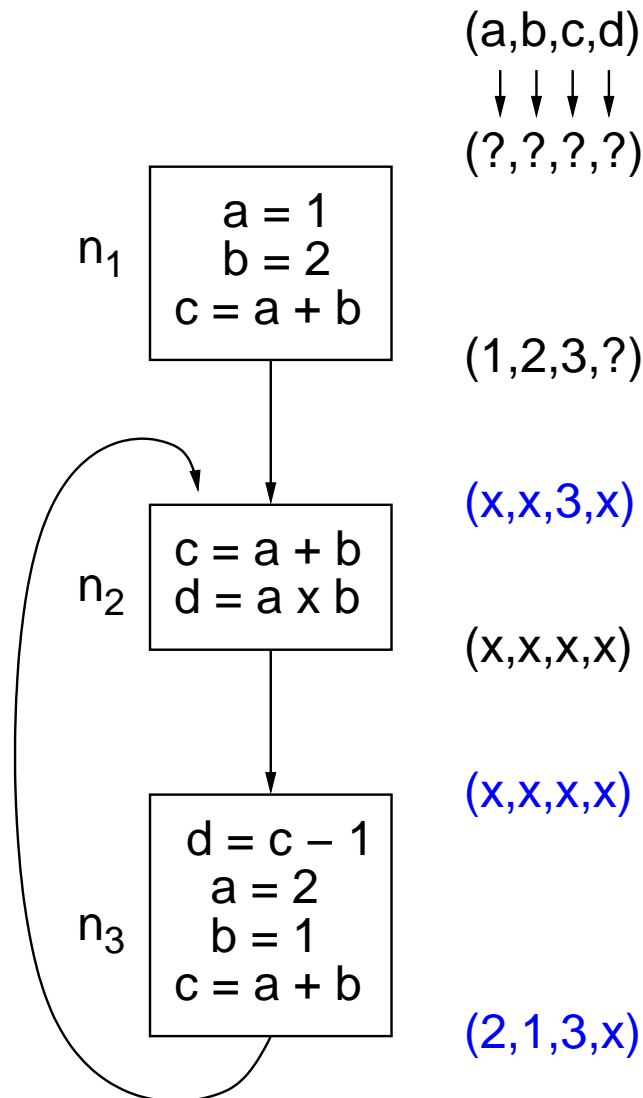
Constant Propagation Movie 2

We proceed carefully according to the defined framework.



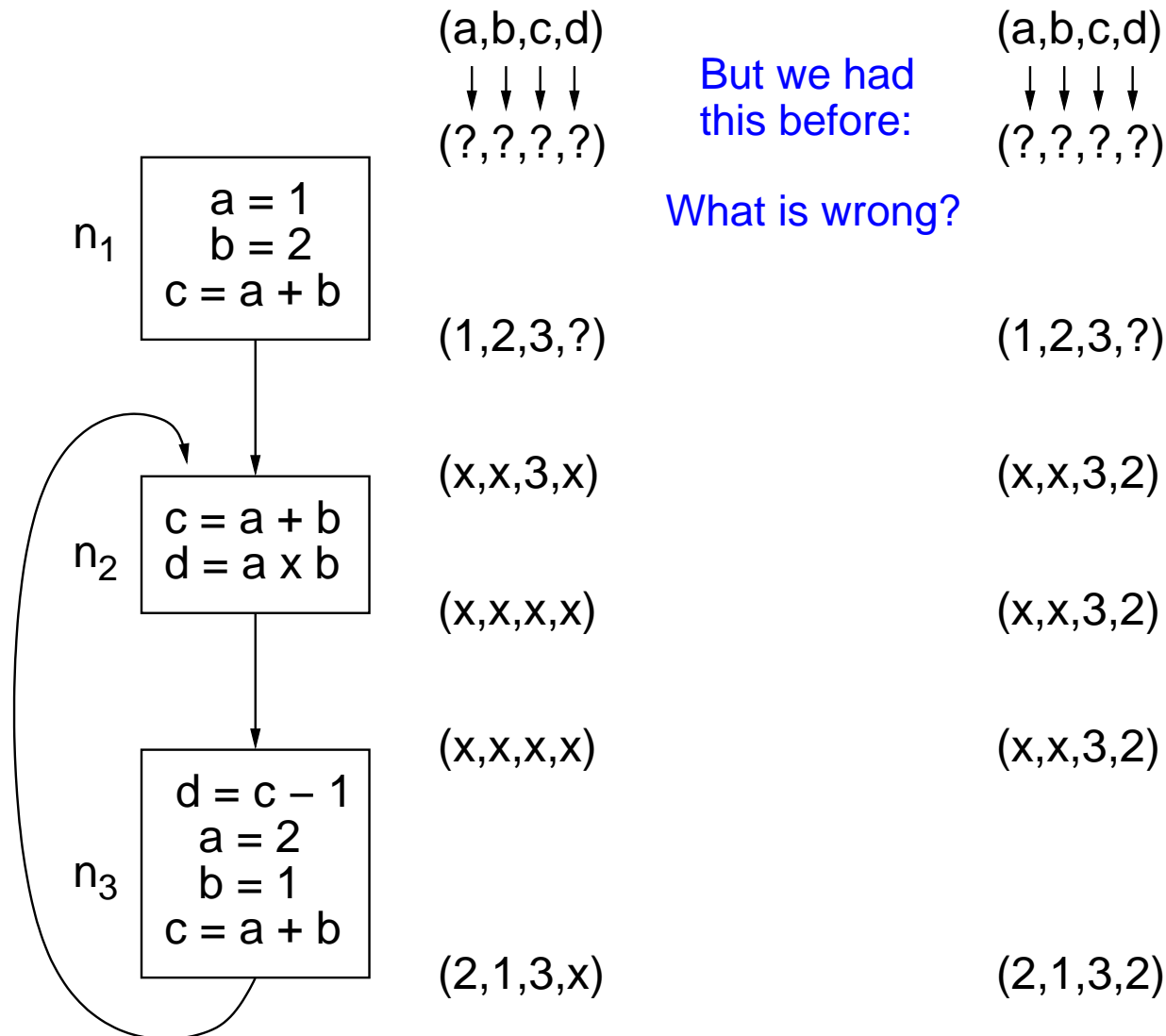
Constant Propagation Movie 2

We proceed carefully according to the defined framework.



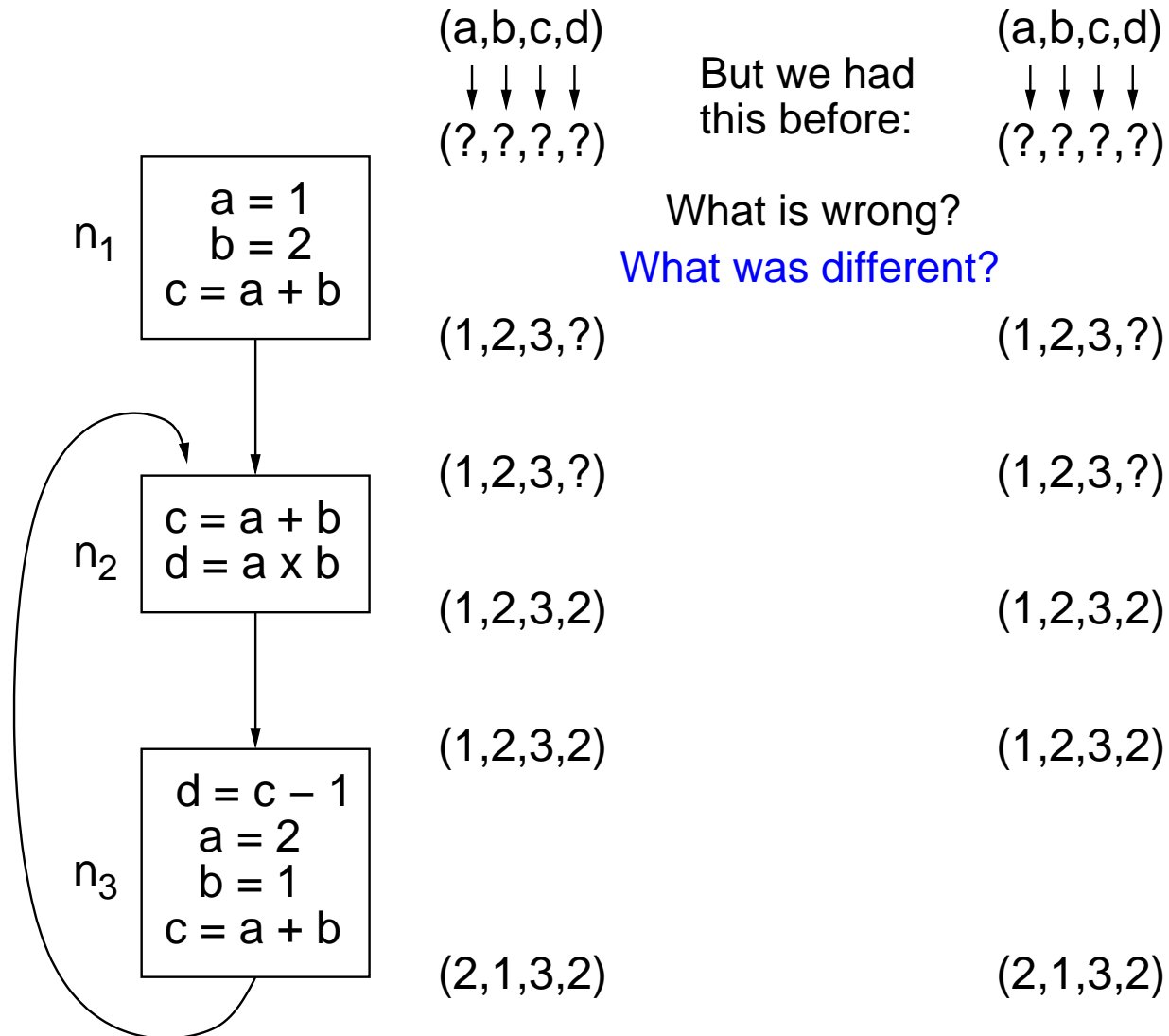
Constant Propagation Movie 2

We proceed carefully according to the defined framework.



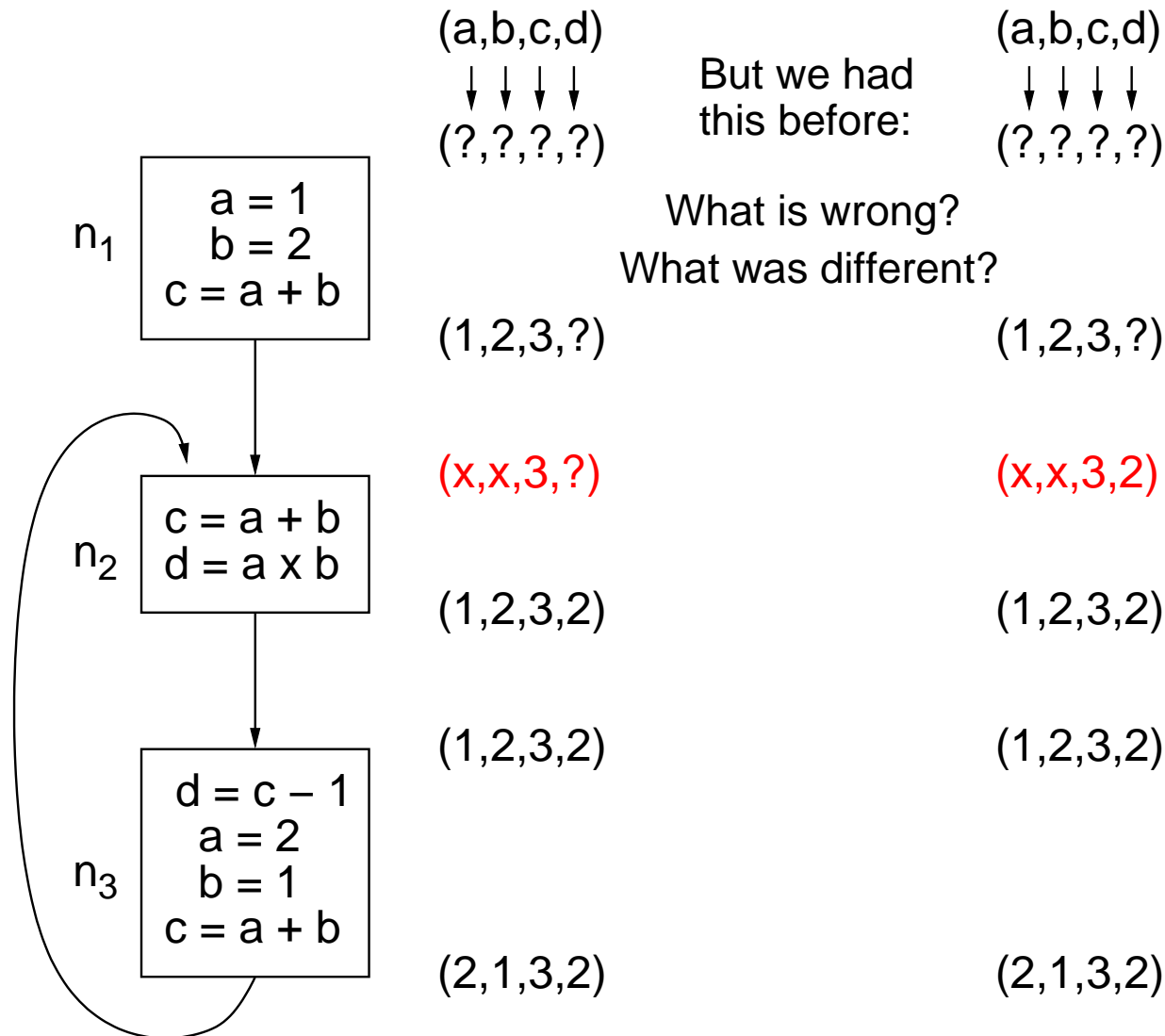
Constant Propagation Movie 2

Let us compare the two computations from the last point they were the same.



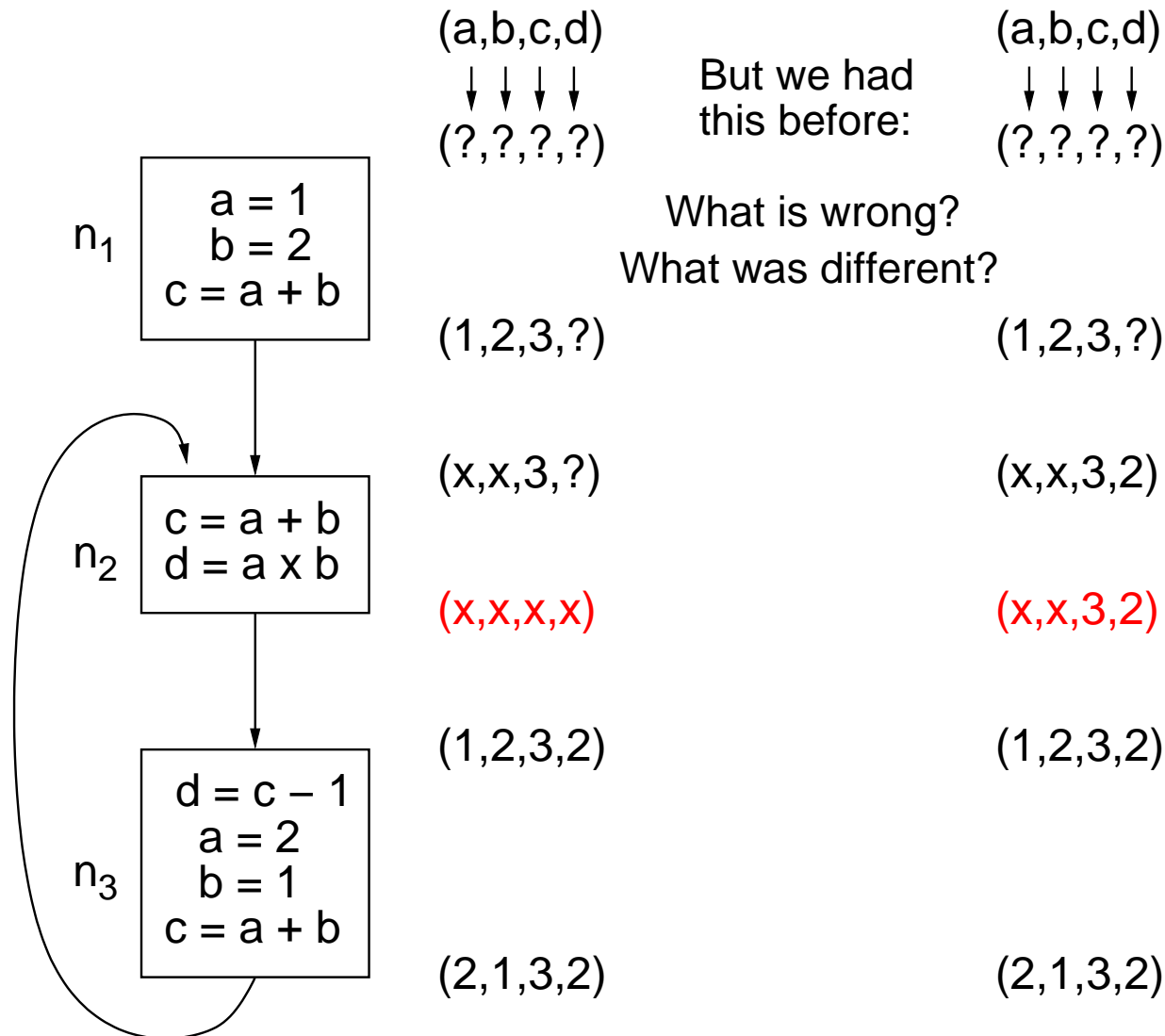
Constant Propagation Movie 2

Still the same.



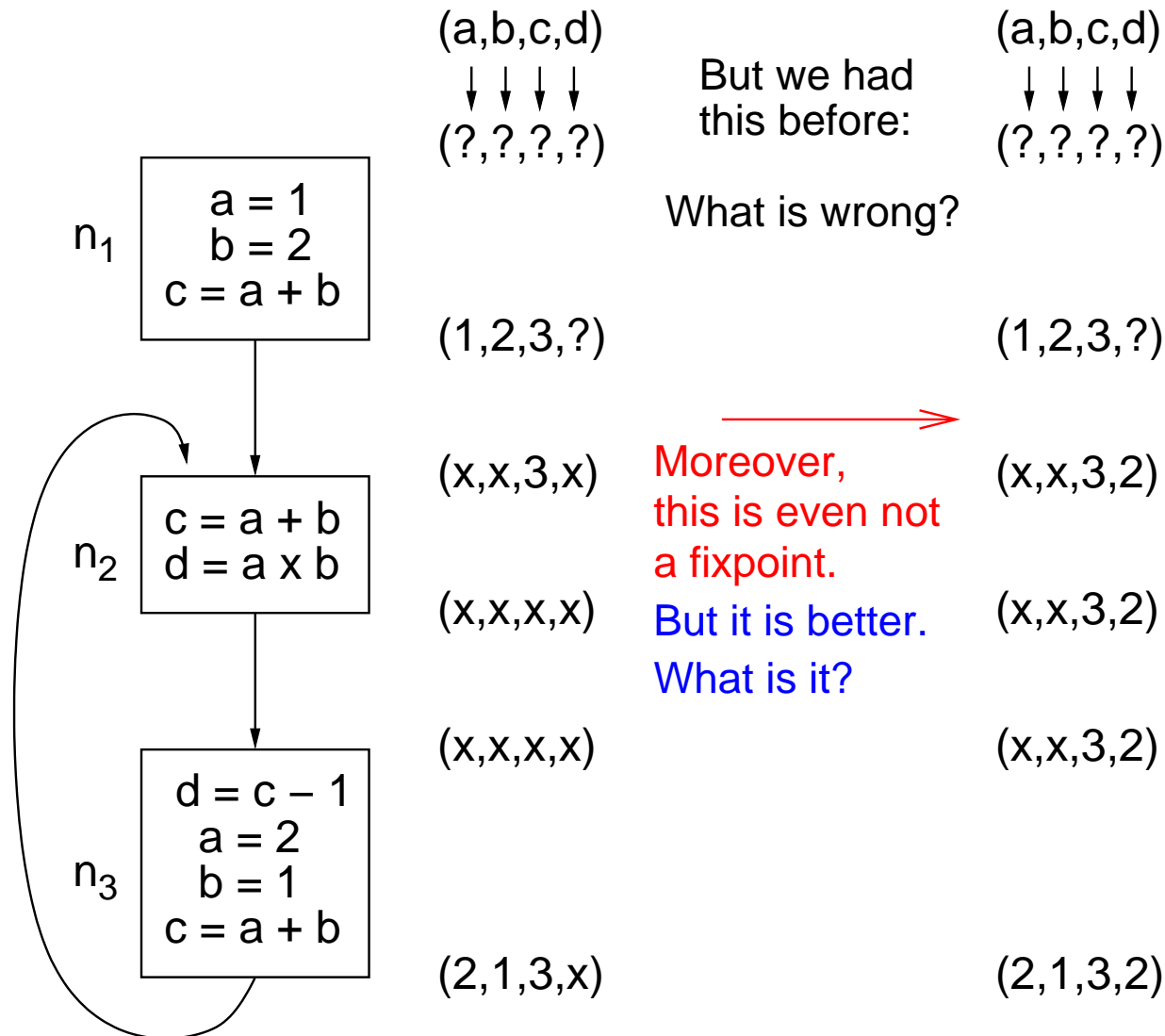
Constant Propagation Movie 2

The difference is here: applying the flow function to merged input values **X** merging the results of the flow function applied to input values.



Constant Propagation Movie 2

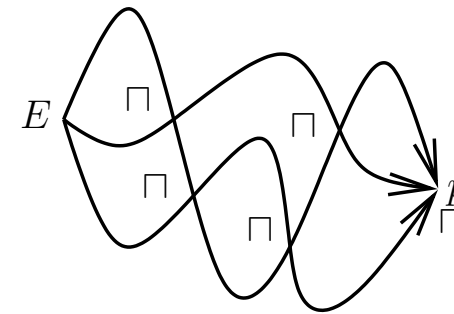
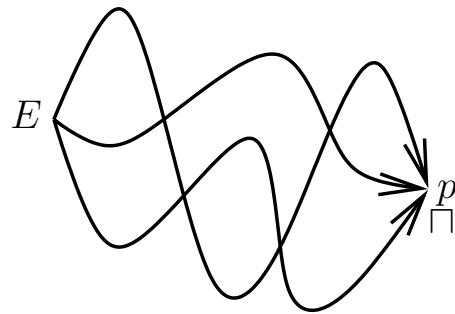
The previous computation does not follow the algorithm and yields a better solution that is not a solution of the fixpoint framework. It is the so called **Meet over Paths (MOP)** solution.



Meet over Paths (MoP) Solution

❖ Let us make more precise what we want:

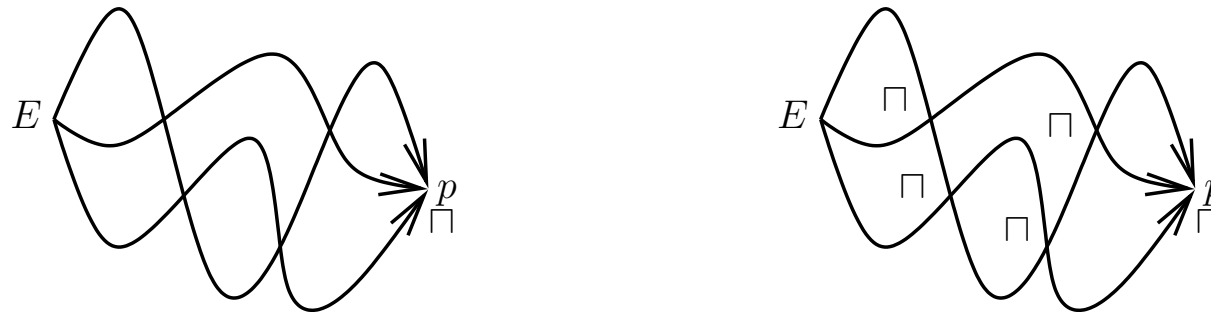
- For each program point p , we want the **maximum flow information** that is **consistent** with all flow values that we could get when evaluating **all flow paths** leading to p .
- This is the so called **Meet over Paths** solution (MoP).
- More formally: For each program path $\rho = p_1 p_2 \dots p_k$, $p_1 = Start$, $p_k = p$ leading through the CFG to p , $val_\rho = f_k \circ \dots \circ f_2 \circ f_1(val_{In_{Start}})$. We want to compute $val_p = \sqcap \{val_\rho \mid \rho \text{ leads to } p\}$. A similar definition applies for the backward case.
- This is a **path based specification**, contrary to the specification of MFP, which is **edge based**.



Meet over Paths (MoP) Solution

❖ Let us make more precise what we want:

- For each program point p , we want the **maximum flow information** that is **consistent** with all flow values that we could get when evaluating **all flow paths** leading to p .
- This is the so called **Meet over Paths** solution (MoP).
- More formally: For each program path $\rho = p_1 p_2 \dots p_k$, $p_1 = Start$, $p_k = p$ leading through the CFG to p , $val_\rho = f_k \circ \dots \circ f_2 \circ f_1(val_{In_{Start}})$. We want to compute $val_p = \sqcap \{val_\rho \mid \rho \text{ leads to } p\}$. A similar definition applies for the backward case.
- This is a **path based specification**, contrary to the specification of MFP, which is **edge based**.



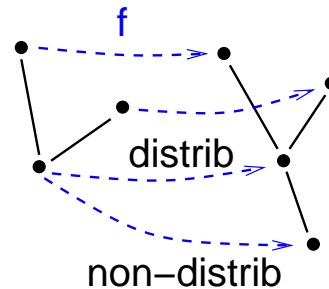
❖ However, the MoP solution is **not algorithmically computable** already for constant propagation. Notice that in presence of loops, there is an **infinite number of paths**.

❖ Fortunately, the **MFP solution can be still used**. It can be less precise than the MoP solution, but it is still safe.

Distributivity of Monotone Frameworks

❖ A monotone framework is **distributive** iff

$$\forall f \in \mathcal{F}, \forall x, y \in L : f(x \sqcap y) = f(x) \sqcap f(y)$$

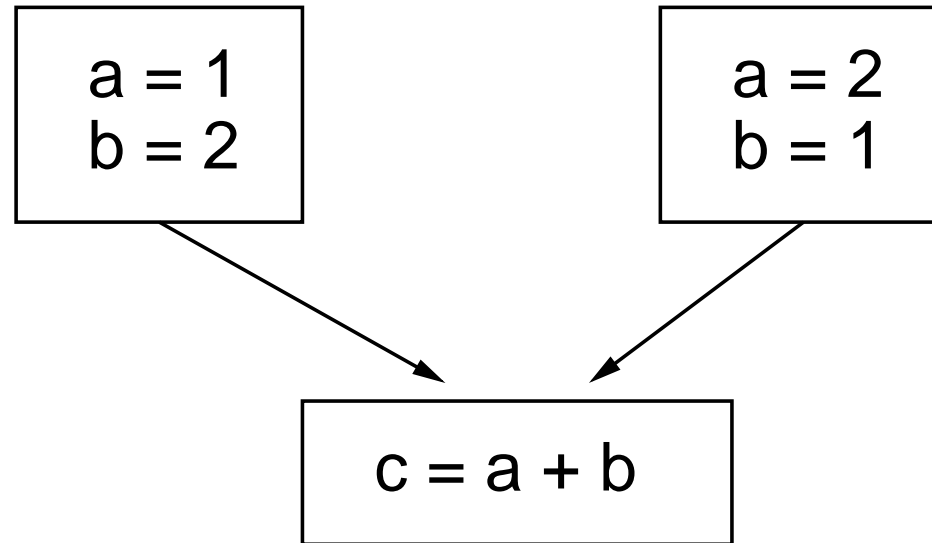


❖ Why is it important?

- In a distributive framework, **the MFP solution equals the MoP solution**.
- Otherwise, the MoP solution can be better than the MFP solution (however, the MFP solution is still safe).

❖ The **bit-vector frameworks are distributive**, **constant propagation is not**.

Nondistributivity of Constant Propagation



❖ We have that:

- $f(\{(a, 1), (b, 2)\} \sqcap \{(a, 2), (b, 1)\}) = \emptyset$, but
- $f(\{(a, 1), (b, 2)\}) \sqcap f(\{(a, 2), (b, 1)\}) = \{(c, 3)\}$.

A Note on Interprocedural Data Flow Analysis

Interprocedural Data Flow Analysis

- ❖ One possible approach to interprocedural data flow analysis is to use **flow values in the form of functions** (mapping an input to an output).
- ❖ Another approach is to use **procedure/function summaries** having basically a form of a **cache** saying whether a certain procedure/function has already been analysed for some input and, if so, giving the result for that input.
- ❖ Interprocedural analyses may differ in whether and how much **context-sensitive** they are, i.e., in how much they track the context of the procedure/function calls (giving the values of the global variables at the call time, the contents of the stack, etc.).
 - A **trade-off between precision and cost**.
- ❖ For more information, see literature.
 - E.g., the books that are mentioned at the end of the slides.

Pointer Analyses

May and Must Aliasing

❖ Let us consider **static variables only**, no heap, stack, unbounded data structures.

❖ `int* x;` If `x == y`, then `x` and `y` **alias** (point to the same address).

`int* y;`

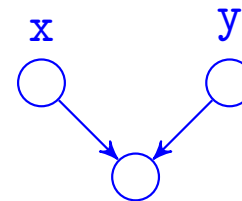
`⋮`

p_1 : `*x = 1;`

`*y = 2;`

p_2 : `i = *x + 3;`

`⋮`



● If `x` and `y` certainly alias at p_1 , then the statement at p_2 may be replaced by `i = 5`.

● If `x` and `y` certainly do not alias at p_1 , then the statement at p_2 may be replaced by `i = 4`.

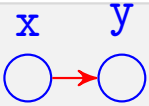
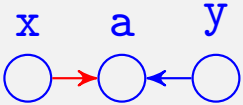
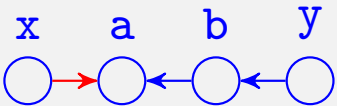
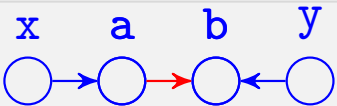
❖ We are interested in the **may and must point-to relations** on program variables, which may be viewed as directed graphs where nodes are program variables.

❖ These points-to relations contain information about aliasing.

- The must point-to relation gives a **must alias equivalence relation** on variables.
- The may point-to relation gives a **may alias similarity relation** on variables.

Flow Sensitive May Point-to Analysis

- Lattice of data flow values $(2^{Vars \times Vars'}, \supseteq)$ with $(\top, \perp, \sqcap) = (\emptyset, Vars \times Vars', \cup)$ where $Vars' = Vars \cup \{\text{null}\}$.
- Flow functions: $f_n(X) = Gen_n(X) \cup (X \setminus Kill_n(X))$.
- Flow F : along the control flow.
- Extreme flow variable E : $Start_{In}$.
- Initial value of E : \emptyset .

	operation	<i>Kill</i>	<i>Gen</i>
Strong upds.	$x = \&y$ 	$\{x \rightarrow a \mid a \in Vars'\}$	$\{x \rightarrow y\}$
	$x = y$ 	$\{x \rightarrow a \mid a \in Vars'\}$	$\{x \rightarrow a \mid y \rightarrow a \wedge a \in Vars'\};$
	$x = *y$ 	$\{x \rightarrow a \mid a \in Vars'\}$	$\{x \rightarrow a \mid y \rightarrow b \rightarrow a \wedge a, b \in Vars'\}$
Weak update	$*x = y$ 	\emptyset	$\{a \rightarrow b \mid x \rightarrow a \wedge y \rightarrow b \wedge a, b \in Vars'\}$

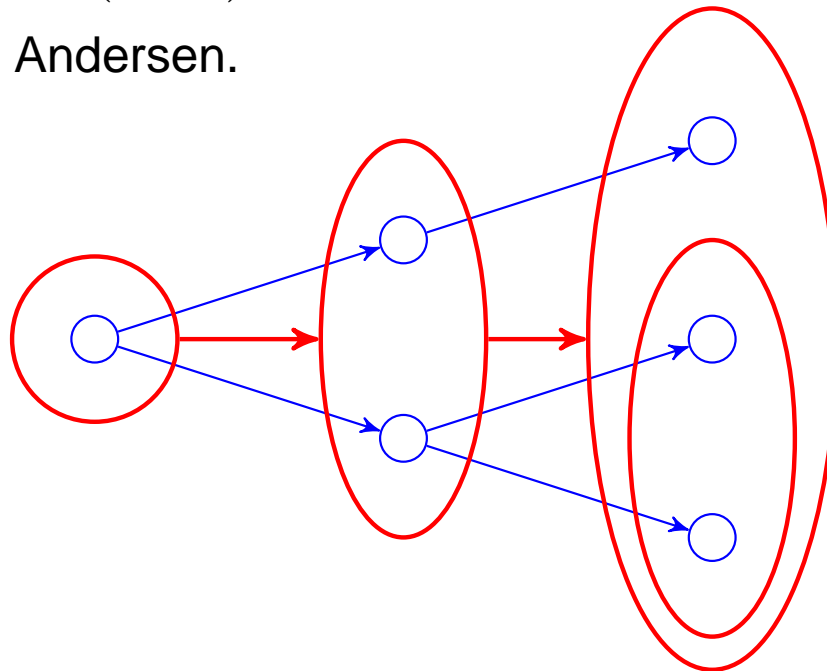
Flow Insensitive Analysis, Andersen's Algorithm

- ❖ Computing a different points-to graph at each program point is expensive.
- ❖ An alternative is a **flow insensitive analysis**.
- ❖ The **Andersen's algorithm**:
 - **Ignore control-flow**, i.e., assume that every block is connected with every block.
 - Replace all strong updates by **weak updates** ($Kill = \emptyset$).
 - Compute a **single points-to relation** that holds regardless of the order in which assignment statements are actually executed.
 - The computed points-to relation is **the least upper bound of all the points-to relations** computed by the flow sensitive analysis.
 - Less, but still comparably accurate than the flow sensitive analysis.
 - Much cheaper.
 - Requires $\mathcal{O}(n^3)$ **time** where n is the number of pointer statements.

Flow Insensitive Analysis, Steensgard's Algorithm

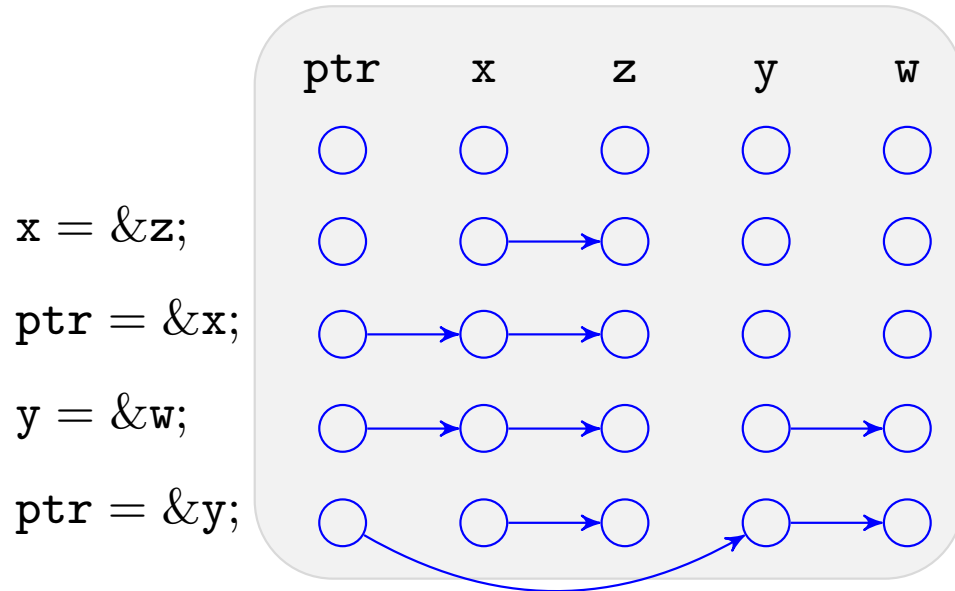
❖ The **Steensgard's algorithm** is essentially the Andersen's algorithm simplified by **merging nodes** that may be pointed-to by the same node.

- Start with singleton equivalence classes.
- Merge two classes if a class may point-to both of them.
- Relate a new class with all the classes that its components were related with.
- May be implemented as a single pass through a program, which gives $\mathcal{O}(n\alpha(n))$ **time** (i.e., almost linear time) algorithm (n is the number of pointer statements and $\alpha(n) = f^{-1}(n)$ where $f(m) = A(m, m)$ for the Ackermann function A).
- **Faster, but less precise** than Andersen.

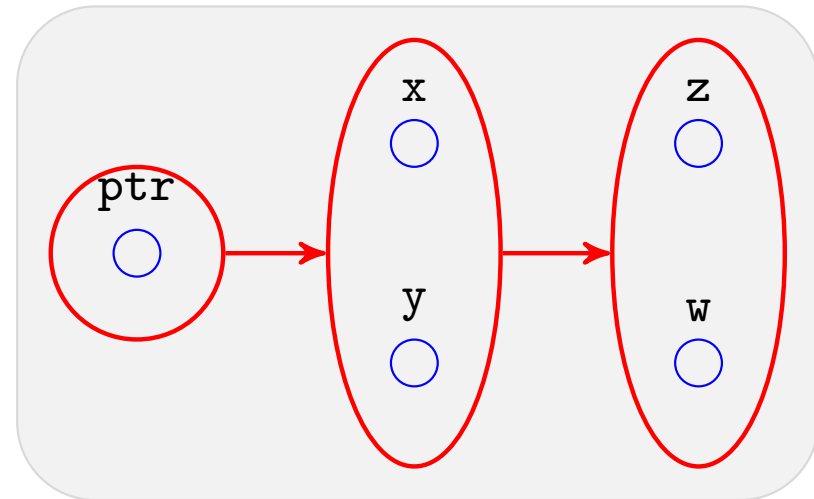
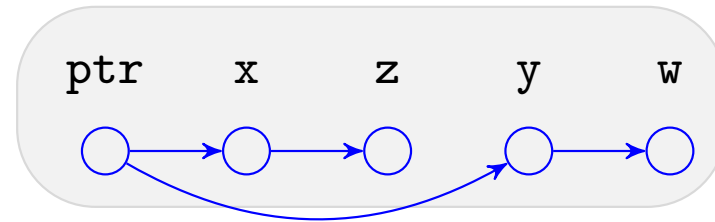


An Example

Flow Sensitive



Andersen



Steensgard

❖ Experience says that **flow/context sensitivity** does not add that much accuracy so that it would be worth the additional computational cost.

Horowitz and Shapiro

❖ The alias analysis algorithm of [Horowitz and Shapiro](#) combines the (usually) faster Steensgard's and (usually) more accurate Andersen's algorithm:

- Divide variables into k different classes.
- Merge only variables contained in the same class.
- Roughly, an $\mathcal{O}(k^2n)$ algorithm.
- $k = 1$ gives Steensgard, $k = |Vars|$ gives Andersen.
- One can iterate over different numbers of classes and different assignments of variables to classes and then take the intersection of the results.
- A good strategy is to consider assignments of variables to classes such that for every pair of variables, it is guaranteed that they belong to different classes for at least one run of the basic algorithm.

Shape Analysis

- ❖ Unlike alias analysis, **shape analysis** tracks the **shapes of the points-to links**, including deep chains of next pointers.
- ❖ Can be used to check for **memory safety** (absence of invalid dereferences, double free operations, memory leaks, etc.) as well as for **shape invariance** (e.g., does one really maintain an unshared, acyclic list somewhere?).
- ❖ Much more demanding.
- ❖ Many different approaches based on
 - **separation logic** used in tools like Space Invader, Slayer, Predator (FIT BUT), ...,
 - **tree automata** used, e.g., in the Forester tool (FIT BUT),
 - **three valued predicate logic with transitive closure** (TVLA),
 - **WSkS** (PALE),
 - **graph grammars**, ...

Literature

Literature

❖ The slides are mostly based on the book **Data Flow Analysis: Theory and Practice** by U.P. Khedker, A. Sanyal, and B. Karkare:

- www.cse.iitb.ac.in/~uday/dfaBook-web

❖ Some parts were also taken from the book **Principles of Program Analysis** by F. Nielson, H. R. Nielson, and C. Hankin.