SSA - Final

15-411/15-611 Compiler Design

Seth Copen Goldstein

November 14, 2019

SSA - review

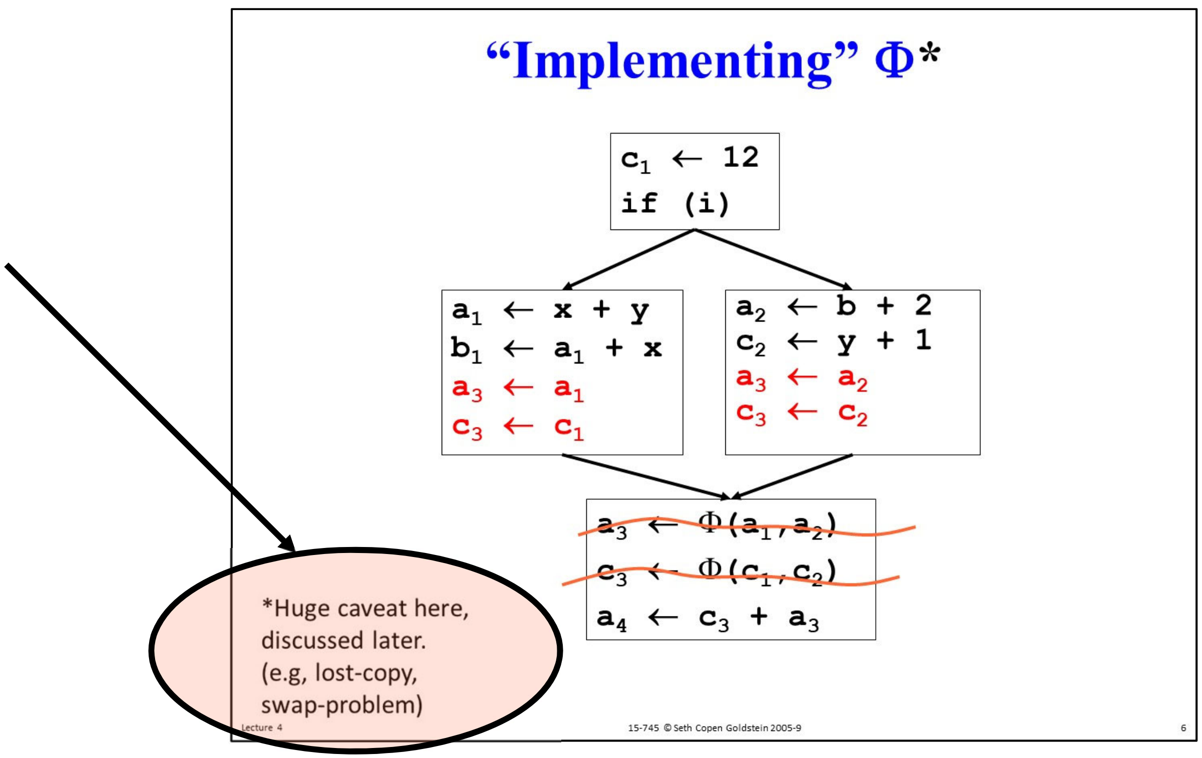
- Static single assignment is an **IR** where every variable has only ONE definition in the program text
  - single **static** definition
  - (Could be in a loop which is executed dynamically many times.)
- Φ-functions used at CFG merge points
- Definitions dominate uses

Advantages of SSA

- Makes du-chains explicit
- Makes dataflow optimizations
  - Easier
  - faster
- Improves register allocation
  - Makes building interference graphs easier
  - Easier register allocation algorithm
  - Decoupling of spill, color, and coalesce
- For most programs reduces space/time requirements

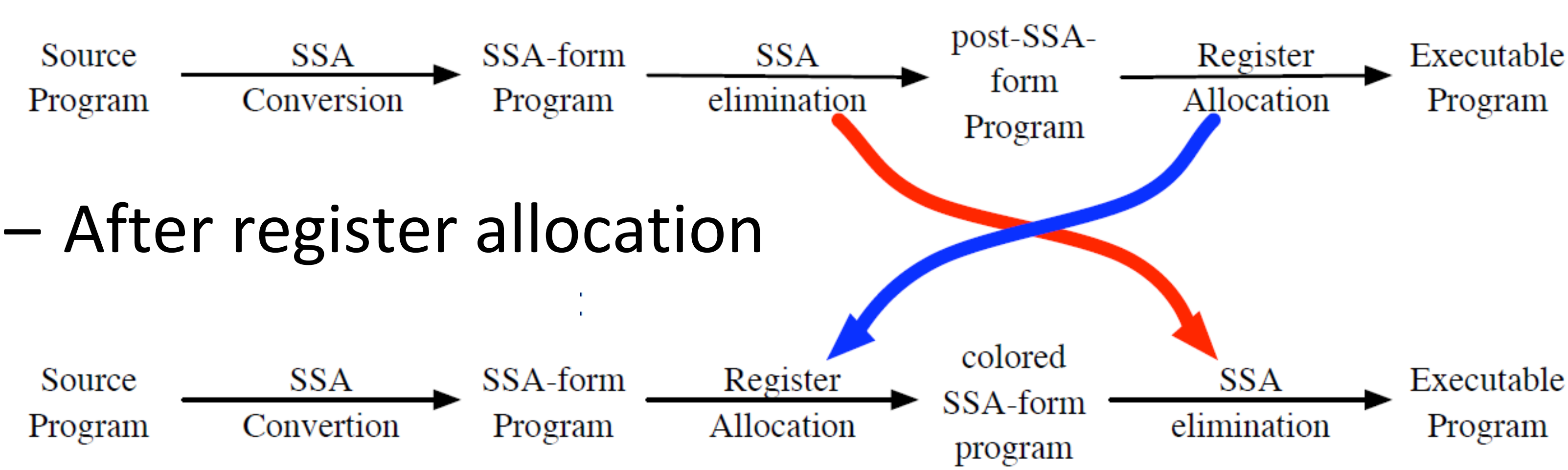
But, …

- Eventually, have to get out of SSA and deconstruct all the Φ-functions
- Recall from Lecture 4



But, …

- Eventually, have to get out of SSA and deconstruct all the Φ-functions
- Two choices:
  - Before register allocation
  - After register allocation



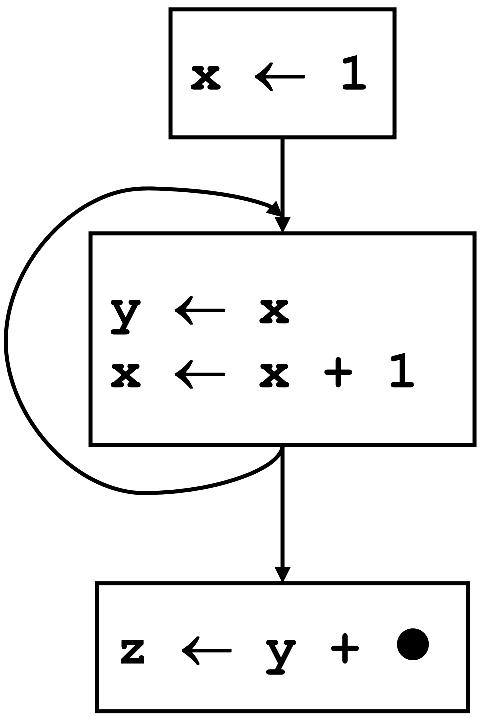
When to deconstruct SSA?

- Before register allocation
  - deconstructing SSA can introduce lots of copies which are easier to eliminate without register constraints
- After register allocation ☺
  - Enables decoupled register allocation
    - spill, color, coalesce
  - Φ-functions may have sources which are registers and memory.
  - Complicated by code-motion optimizations

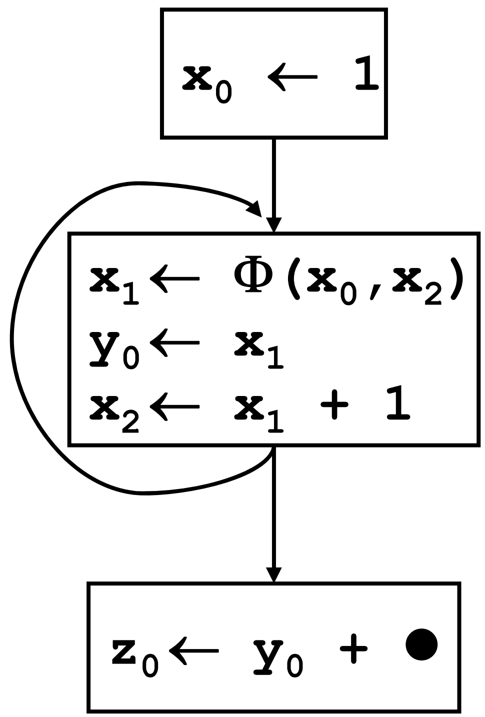
Conventional-SSA

- Conversion to SSA creates “Conventional SSA”
- Main feature:
  - variables involved in a Φ-function never interfere
  - Thus, can allocate to a single **resource**
    - the same register
    - the same frame slot (in case of spill)
- However, code motion can destroy this property

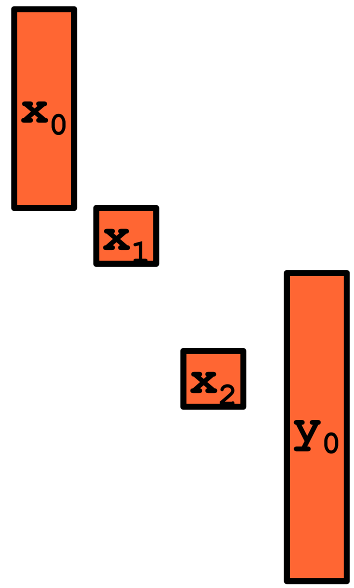
→ SSA



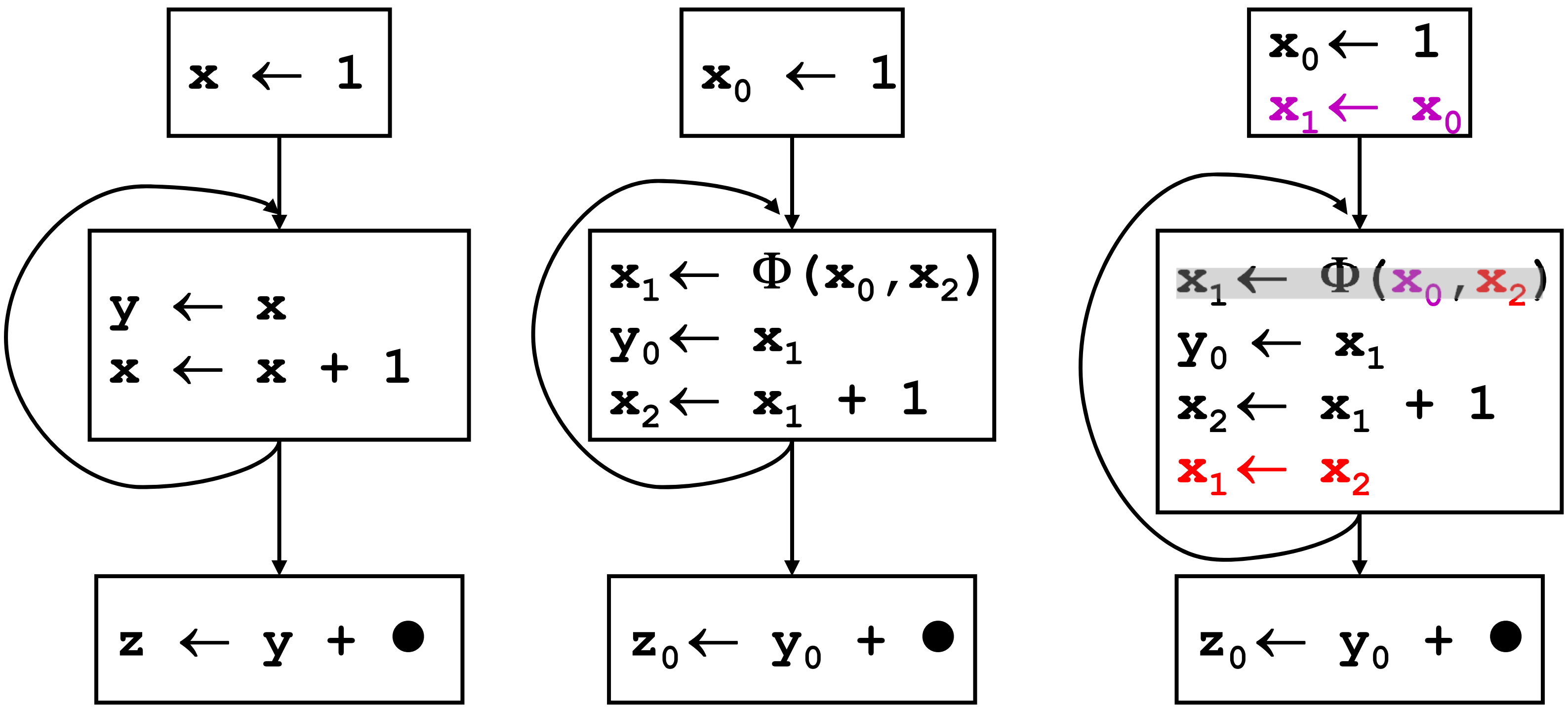
Original



In SSA



Leaving SSA

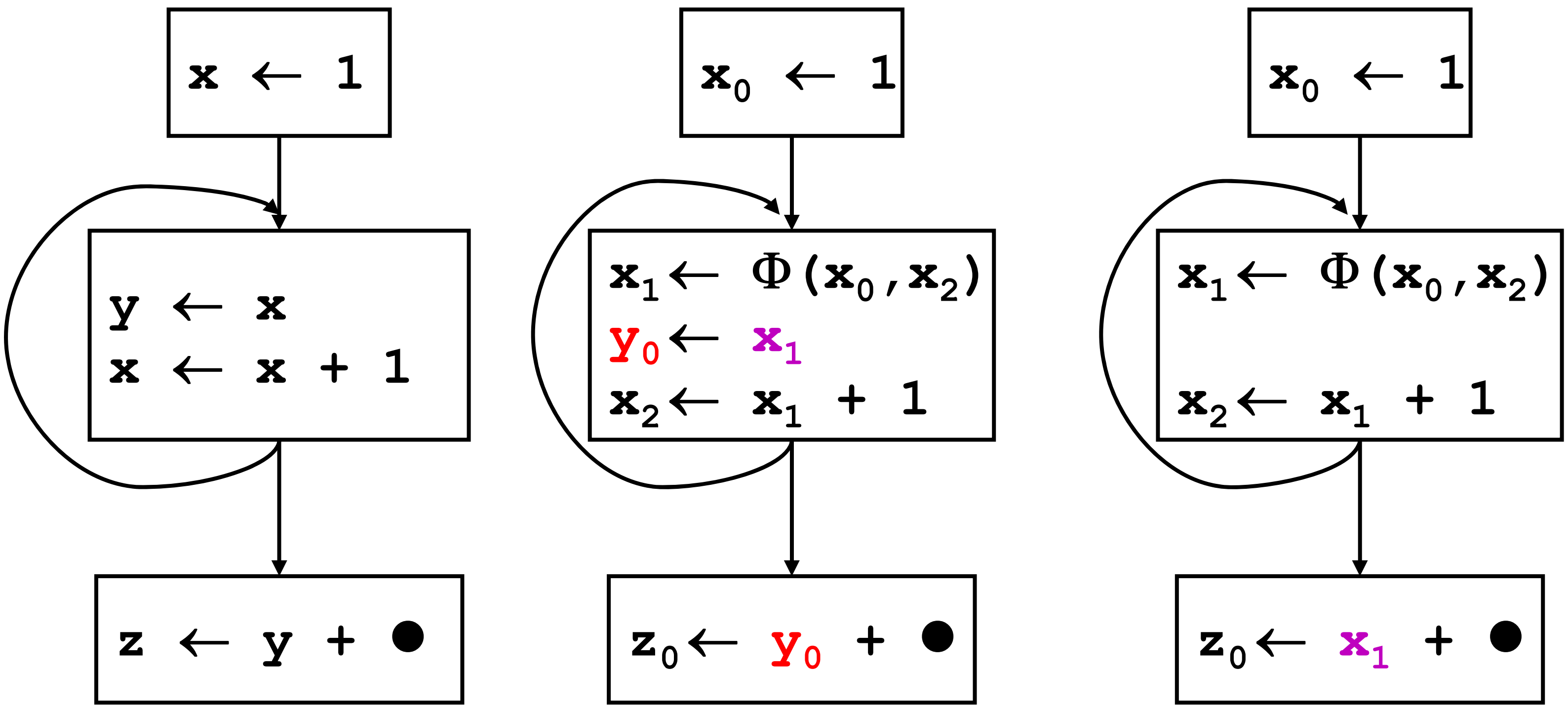


Original

In SSA

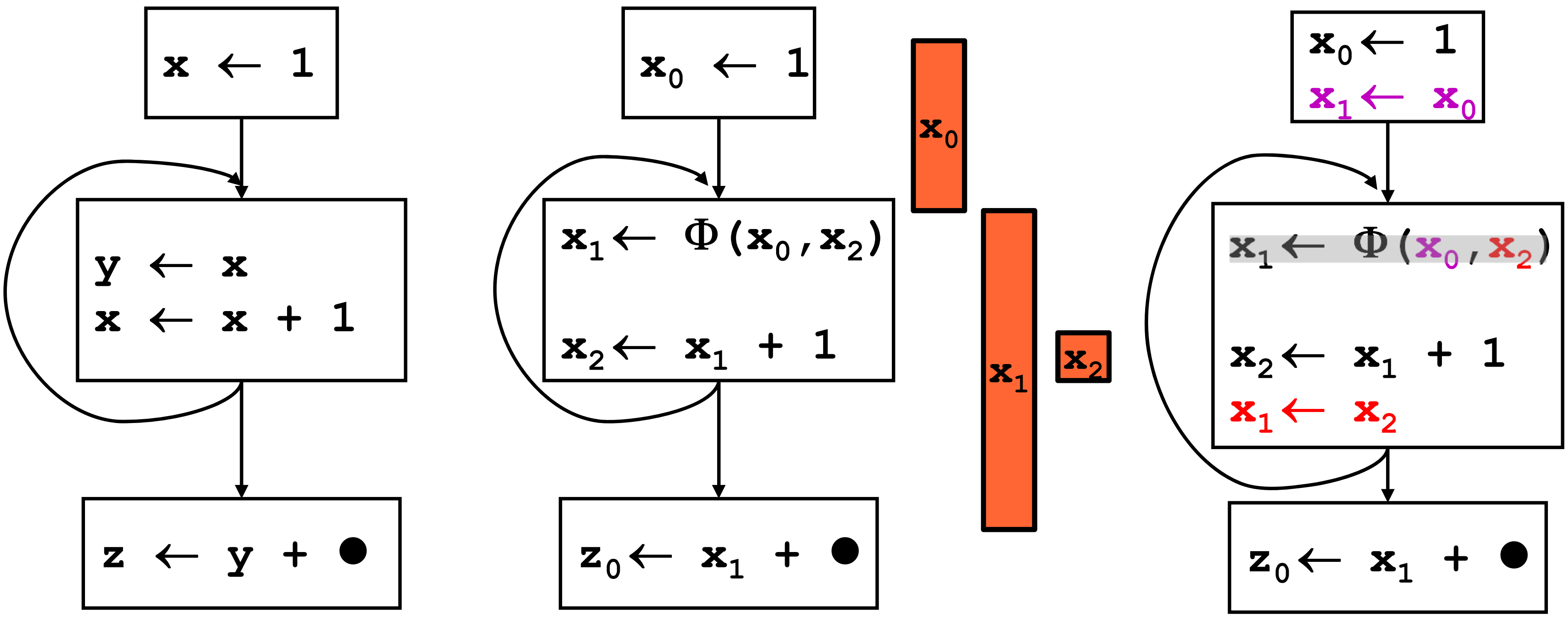
Out of SSA

Copy Folding



Original | In SSA | Copy Folded

Leaving SSA After Copy Folding

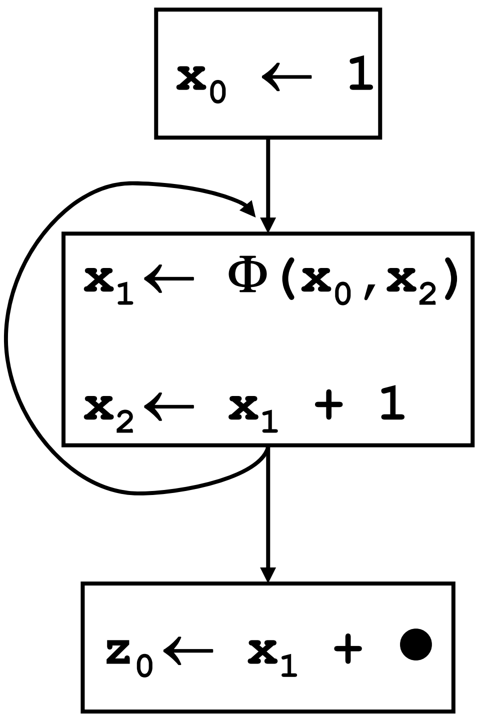


Original | Copy Folded | Out of SSA

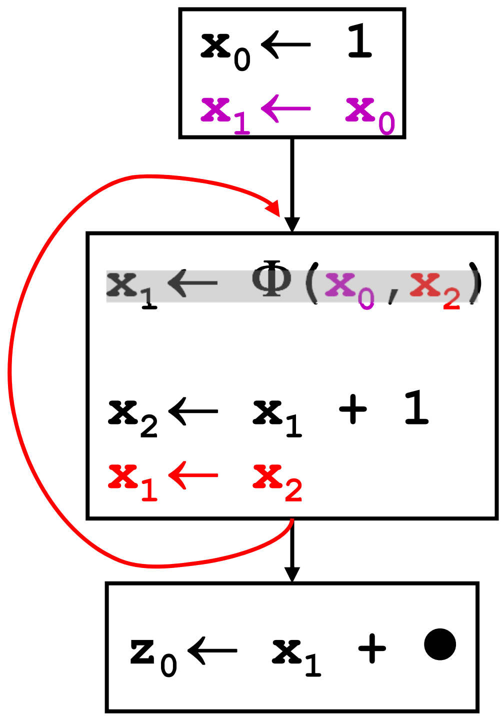
“Lost Copy” Problem

Critical Edges

critical edge



Copy Folded

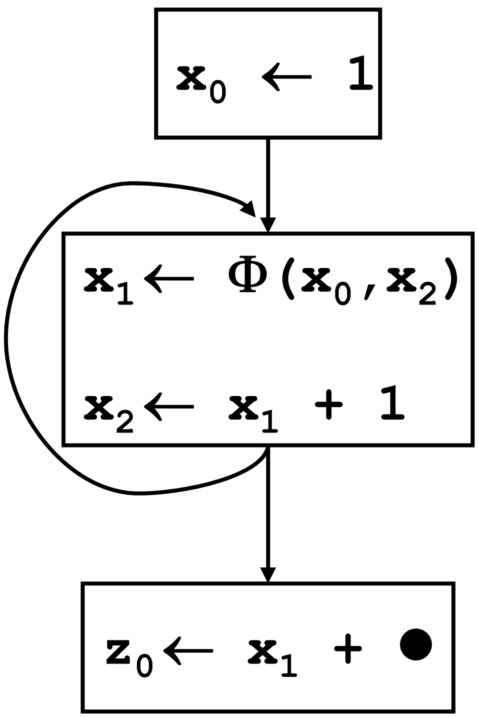


INCORRECT

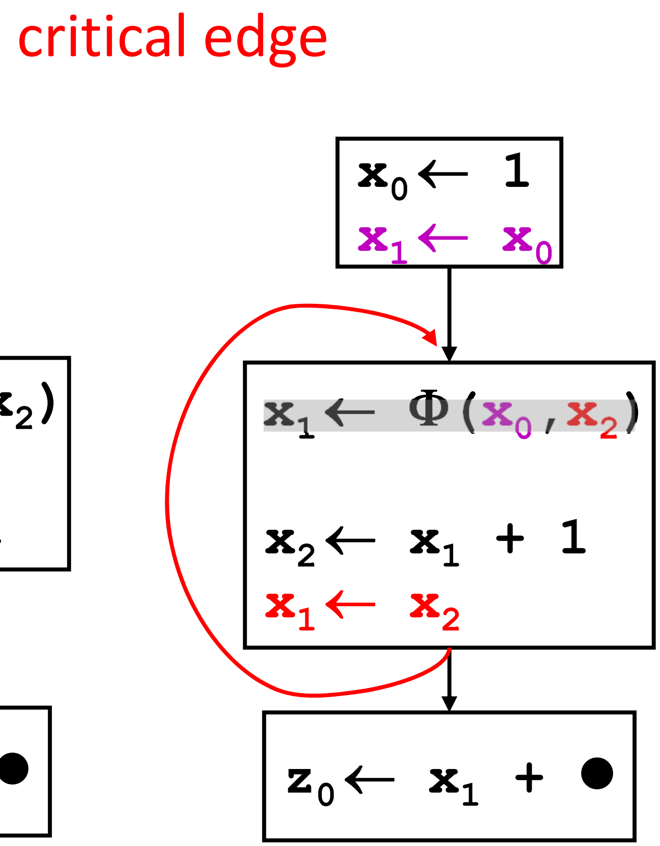
A critical edge is an edge a→b where

- a has > 1 successor and
- b has > 1 predecessor.

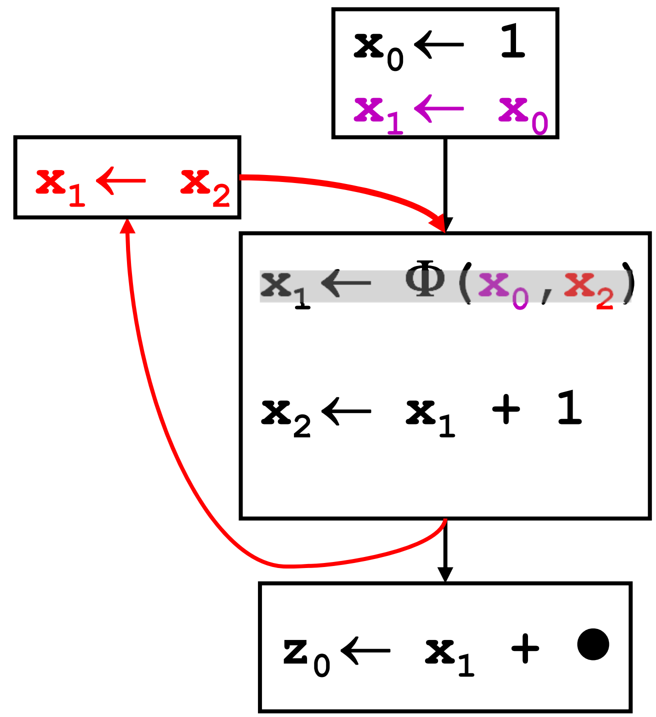
Critical Edges



Copy Folded



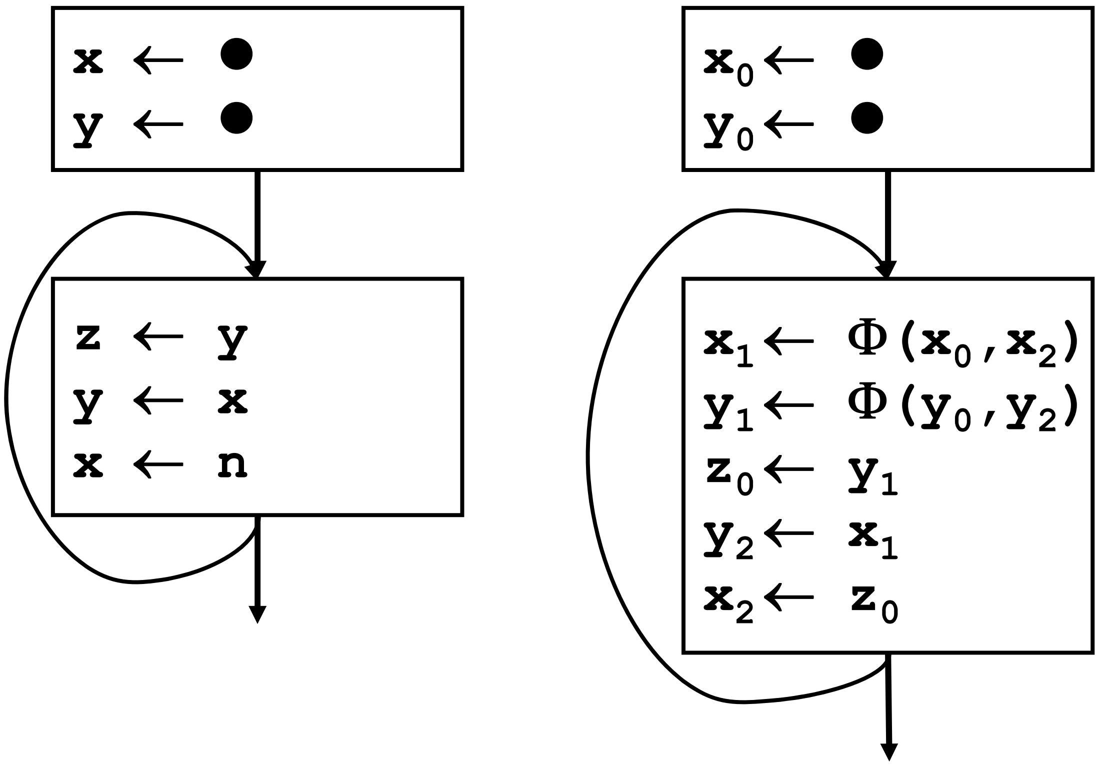
INCORRECT



Inserting block on critical edge

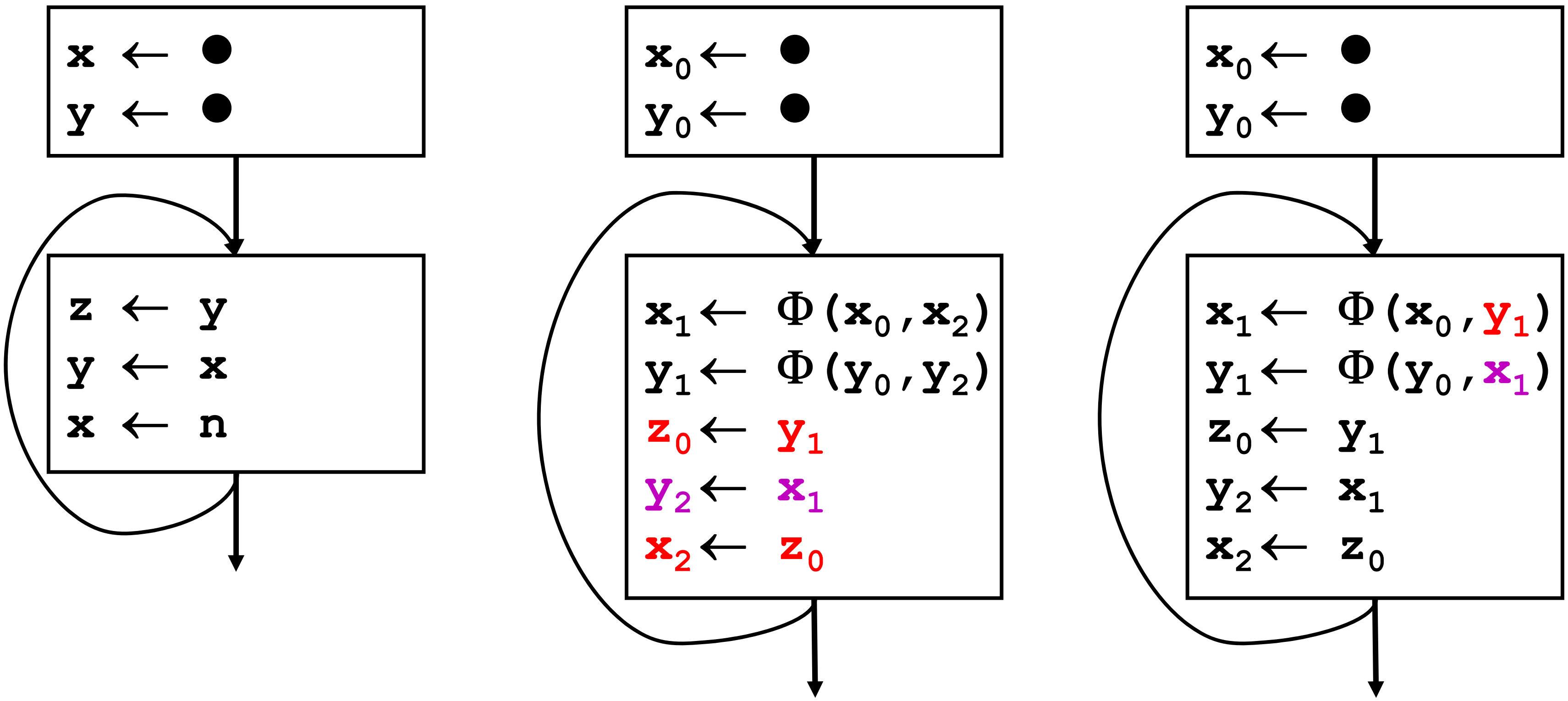
Semantics of Φ-functions

- Semantics of Φ-functions requires copies to be done in parallel.



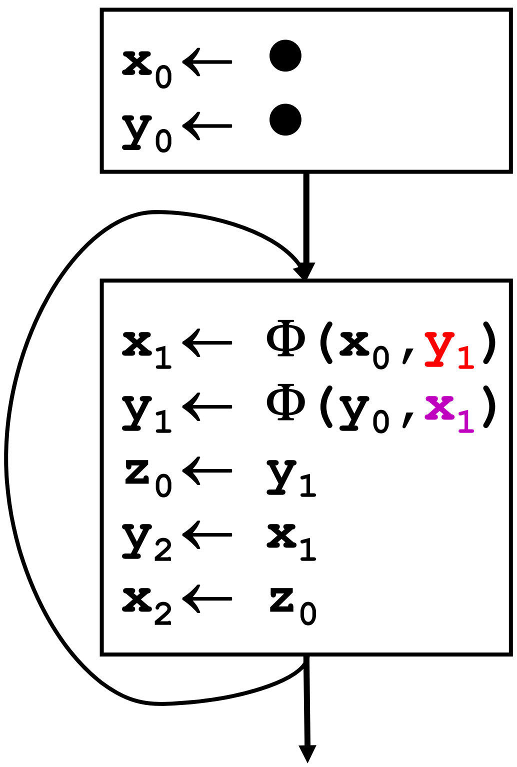
Semantics of Φ-functions

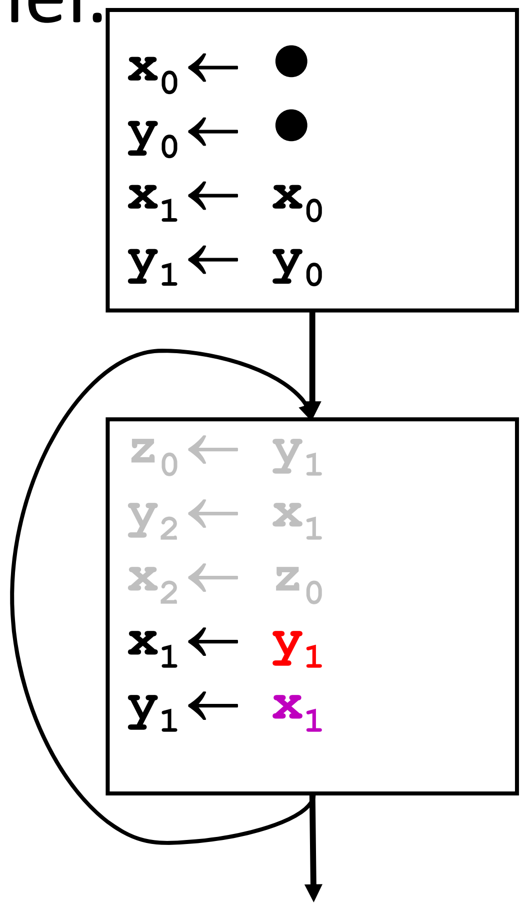
- Semantics of Φ-functions requires copies to be done in parallel.



Semantics of Φ-functions

- Semantics of Φ-functions requires copies to be done in parallel.

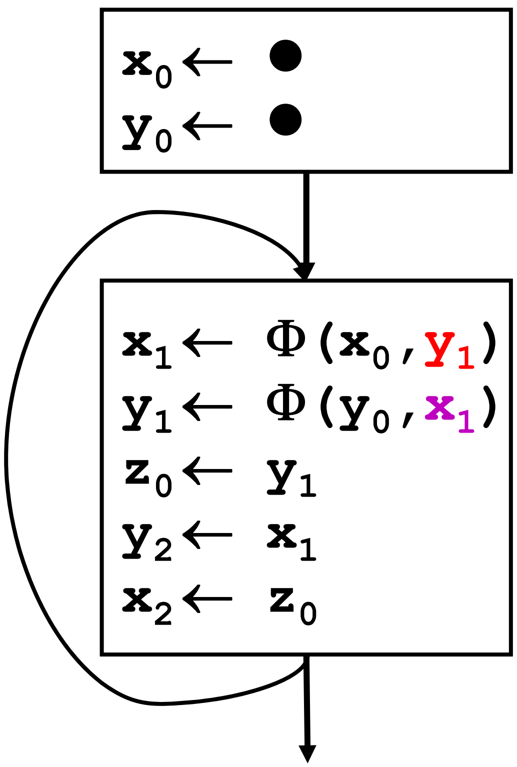


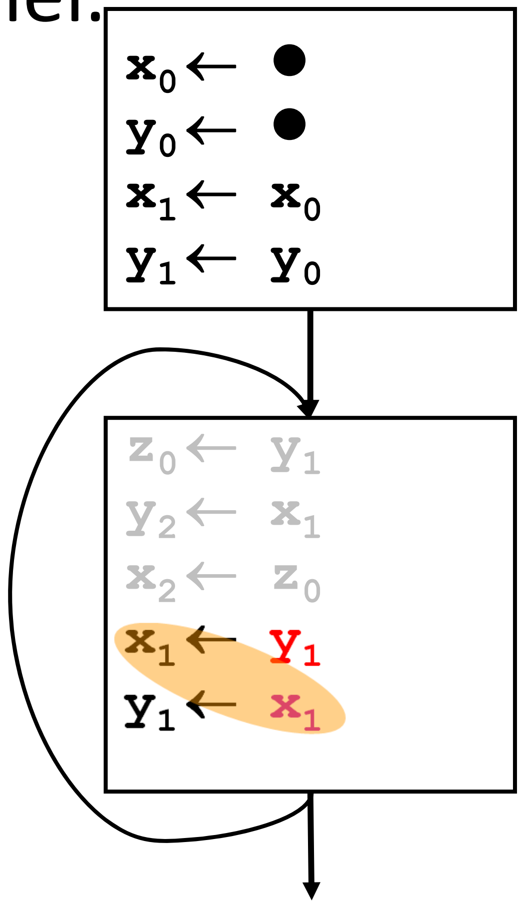


INCORRECT

Semantics of Φ-functions

- Semantics of Φ-functions requires copies to be done in parallel.



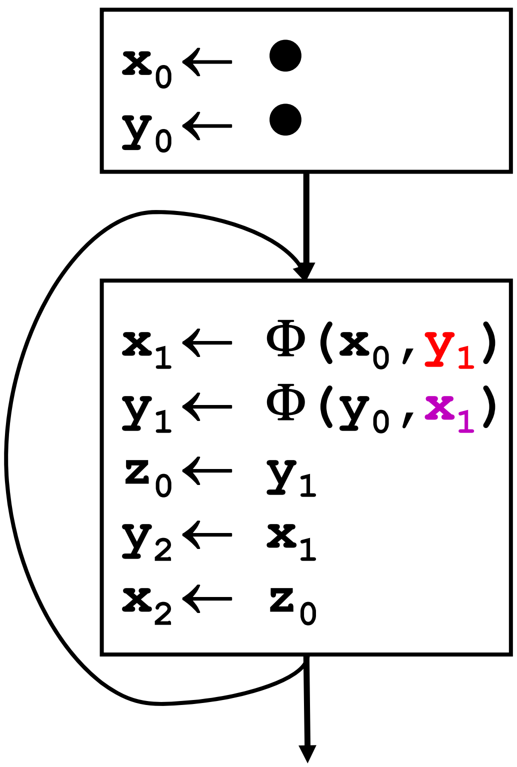


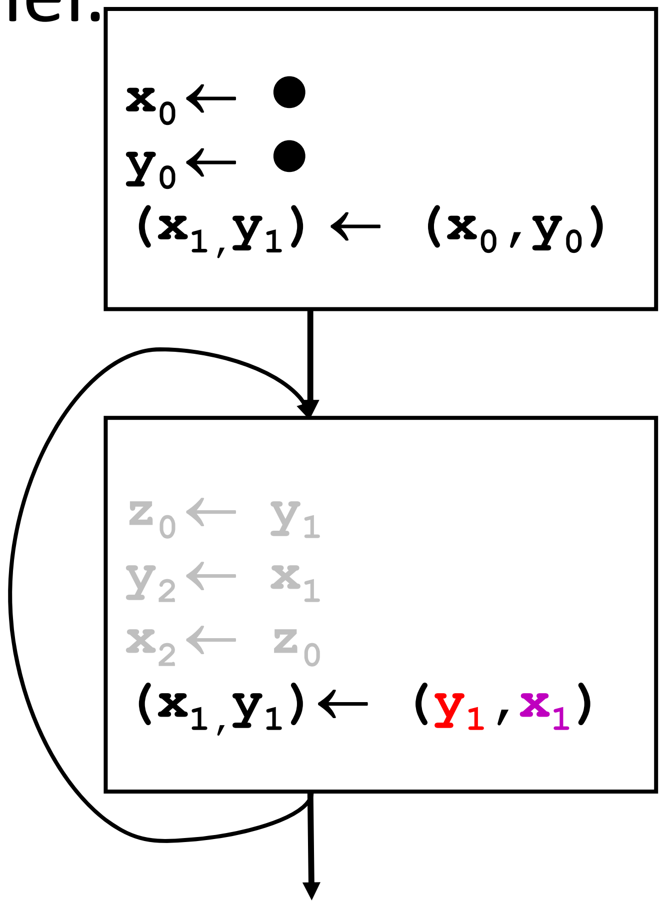
Lost value of $\mathbf{x}_1$! because did Φ assignments sequentially

INCORRECT

Semantics of Φ-functions

- Semantics of Φ-functions requires copies to be done in parallel.





Using Parallel copies

Impact of Spilling

- What happens when we spill a Φ related variable?
- For example:
  - $\mathbf{r} \leftarrow \Phi(\mathbf{r}, \mathbf{m}_0)$
  - $\mathbf{m}_1 \leftarrow \Phi(\mathbf{r}, \mathbf{m}_0)$
- Could require memory-memory move after deconstructing SSA

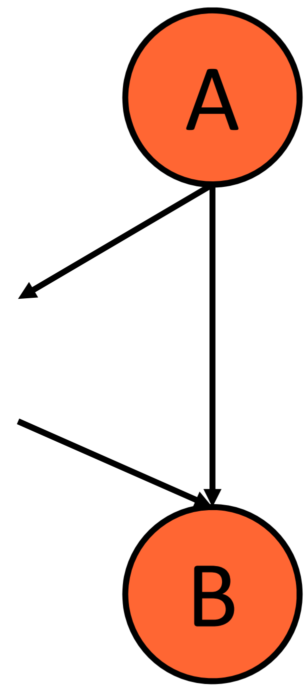
Solution

- Critical Edge Splitting
- Convert back to Conventional-SSA (CSSA)
- Register Allocation
  - Build interference graph
  - pre-spilling
  - coloring
- Deconstruct SSA
  - put parallel-copies in predecessors
  - Eliminate parallel copies
- Coalescing

Note: we changed traditional register allocation sequence

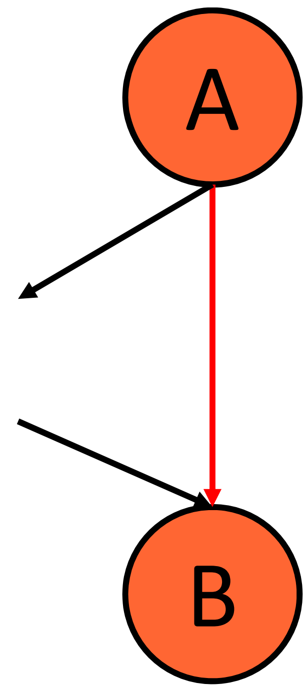
Removing a Critical Edge

- A critical edge is an edge a→b where
  - a has > 1 successor and
  - b has > 1 predecessor.
- For each edge (a,b) in CFG where a > 1 succ and b > 1 pred
  - Insert new block Z
  - replace (a,b) with
    - (a,z) and (z,b)



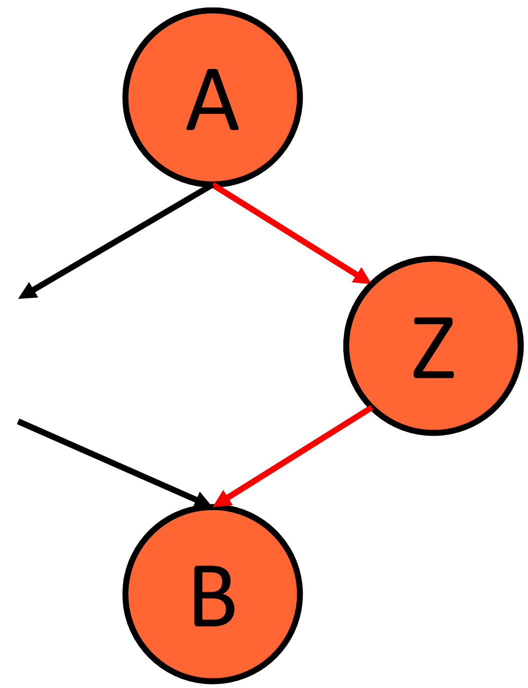
Removing a Critical Edge

- A critical edge is an edge a→b where
  - a has > 1 successor and
  - b has > 1 predecessor.
- For each edge (a,b) in CFG
  where a > 1 succ and
  b > 1 pred
  - Insert new block Z
  - replace (a,b) with
    - (a,z) and (z,b)



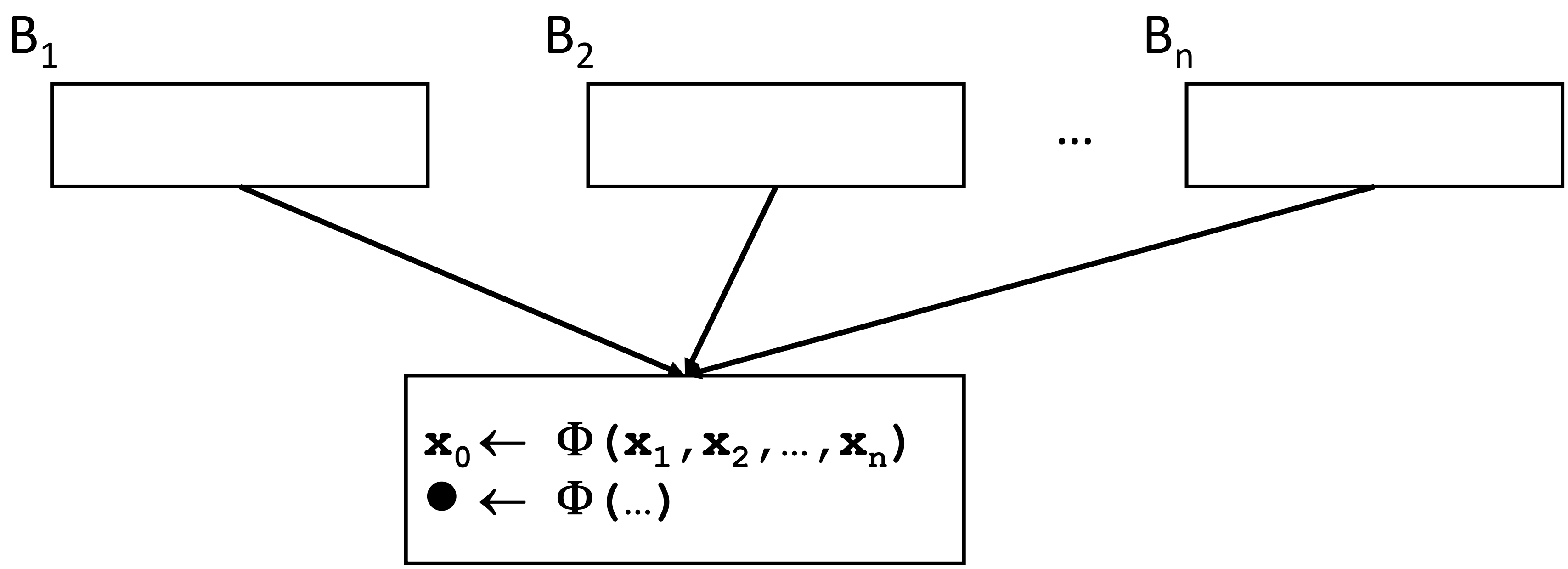
Removing a Critical Edge

- A critical edge is an edge a→b where
  - a has > 1 successor and
  - b has > 1 predecessor.
- For each edge (a,b) in CFG where a > 1 succ and b > 1 pred
  - Insert new block Z
  - replace (a,b) with
    - (a,z) and (z,b)



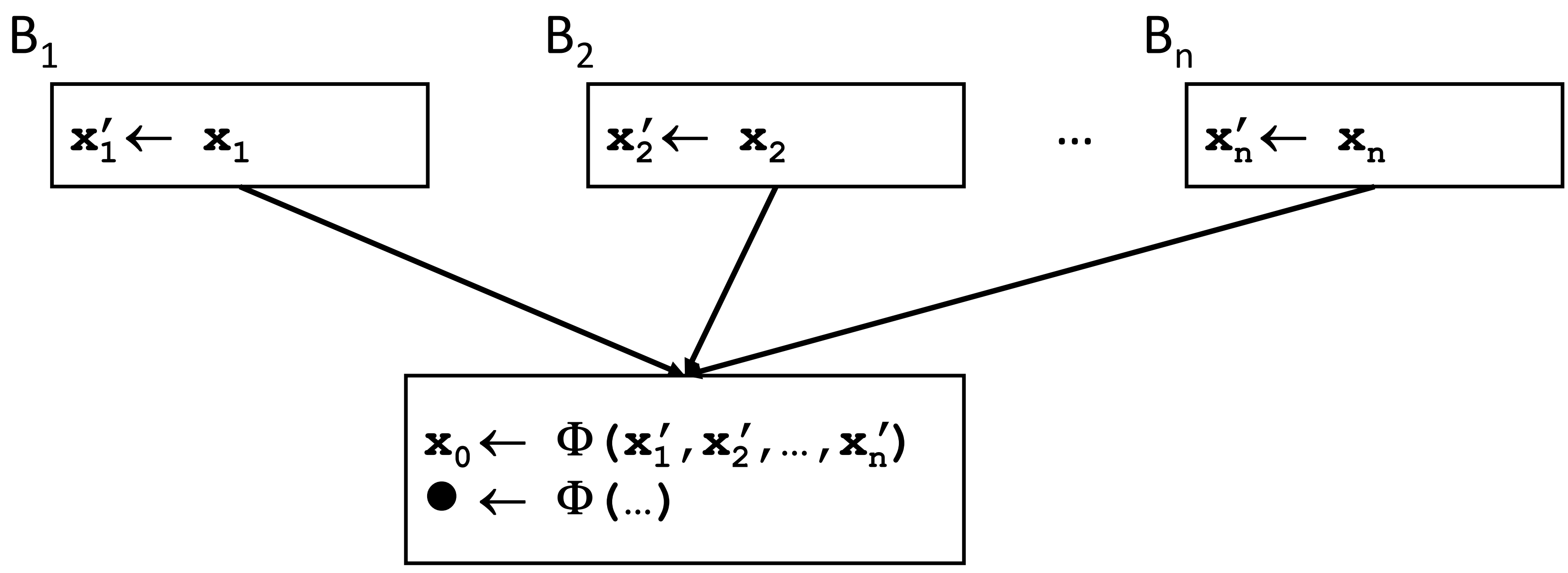
Converting to CSSA

- Goal is to ensure that all Φ related variables do NOT interfere
  - insert copies to (possibly) split live ranges



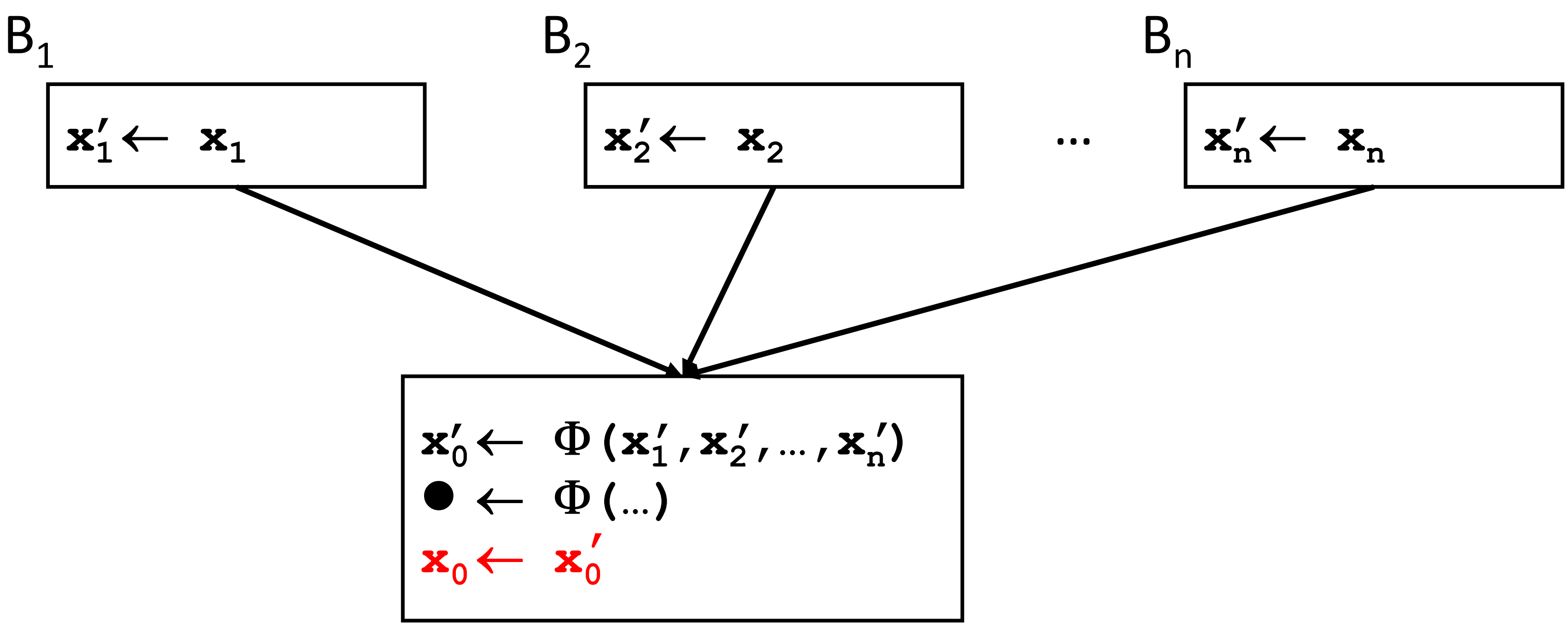
Converting to CSSA

- Goal is to ensure that all Φ related variables do NOT interfere
  - For each argument, insert copy at end of predecessor block and use copy in Φ-function



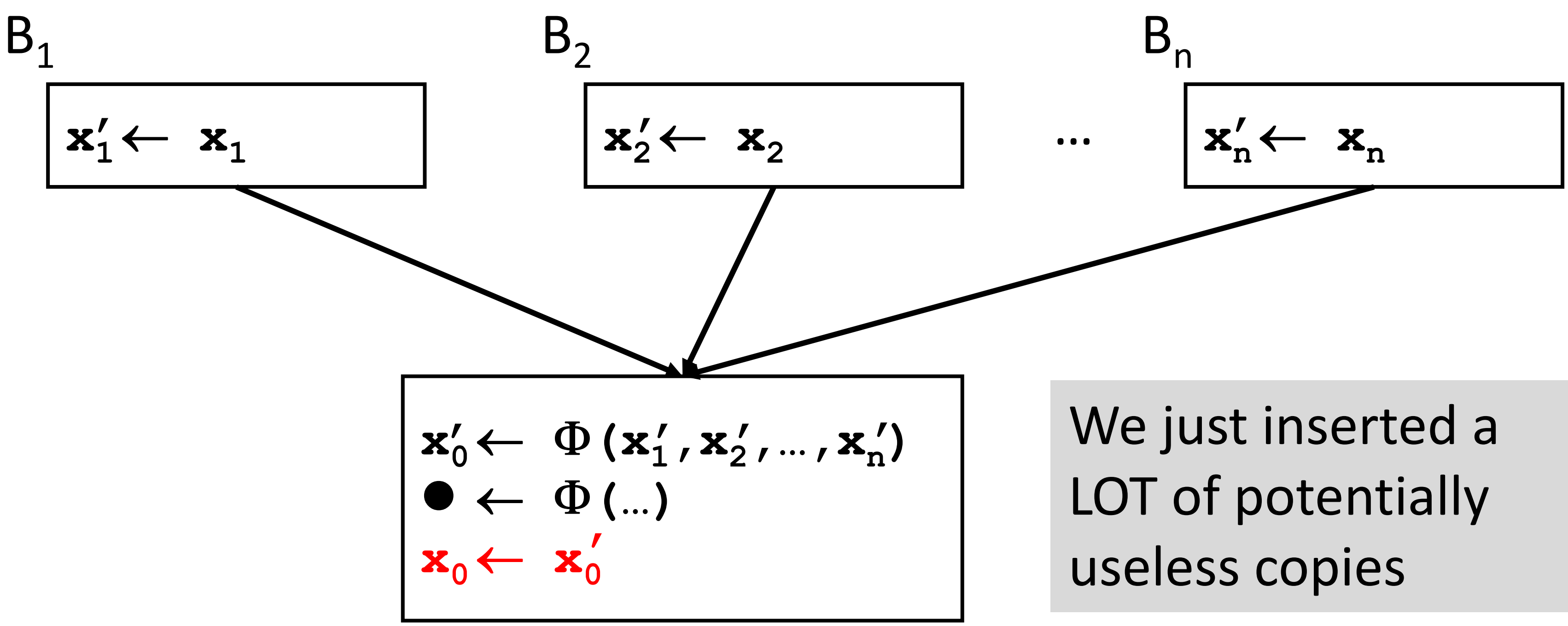
Converting to CSSA

- Goal is to ensure that all Φ related variables do NOT interfere
  - For each argument, insert copy at end of predecessor block and use copy in Φ-function
  - Rename destination
  - Insert copy AFTER all Φ-functions in block



Converting to CSSA

- Goal is to ensure that all Φ related variables do NOT interfere
  - For each argument, insert copy at end of predecessor block and use copy in Φ-function
  - Rename destination
  - Insert copy AFTER all Φ-functions in block



We just inserted a LOT of potentially useless copies

Register Allocation on CSSA

- Build interference graph
- Pre-spill to make it colorable
  - If spill a Φ-related variable, make sure all from same Φ-function use same memory slot!
  - Why do we know this is ok?
  - [Cheat 1: if you spill one, spill them all]
- Color using SEO

Elimination of Φ-functions

- Put parallel copies in predecessor blocks
- Sequentialize the parallel copies

Elimination of Φ-functions

- Put parallel copies in predecessor blocks
- Sequentialize the parallel copies

B_1: $(x_0, x_1, \dots, x_n) \leftarrow (x_{01}, x_{11}, \dots, x_{n1})$

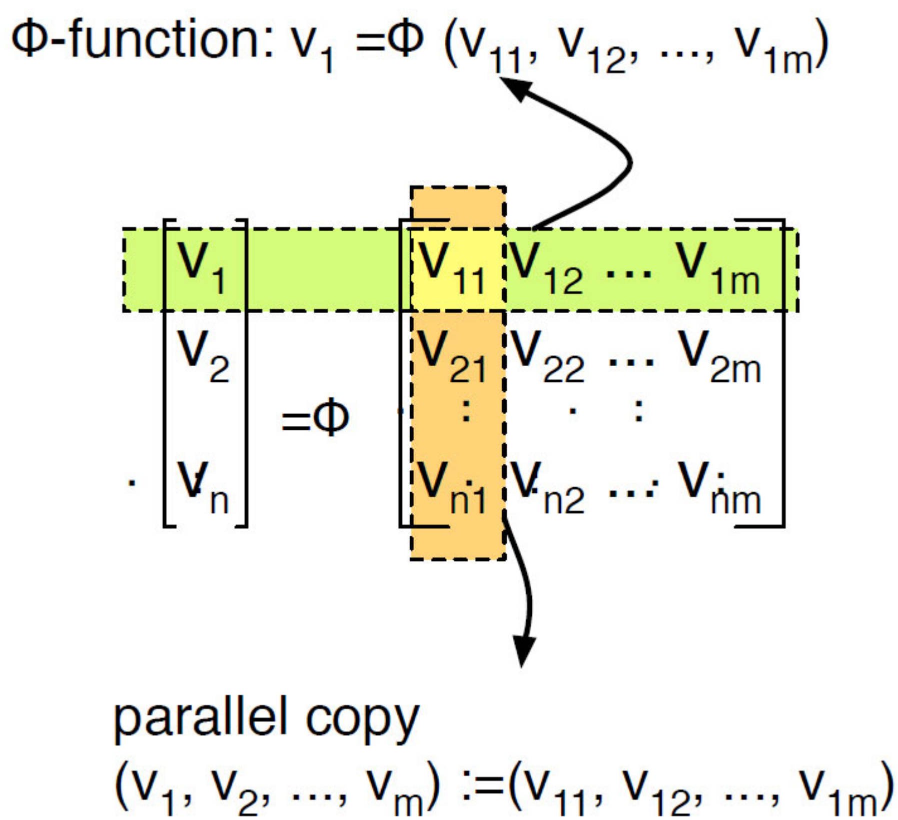
…

B_m: $(x_0, x_1, \dots, x_n) \leftarrow (x_{0m}, x_{1m}, \dots, x_{nm})$

$x_0 \leftarrow \Phi(x_{01}, x_{02}, \dots, x_{0m})$
$x_1 \leftarrow \Phi(x_{11}, x_{12}, \dots, x_{1m})$
…
$x_n \leftarrow \Phi(x_{n1}, x_{n2}, \dots, x_{nm})$

Elimination of Φ-functions

- Put parallel copies in predecessor blocks
- Sequentialize the parallel copies



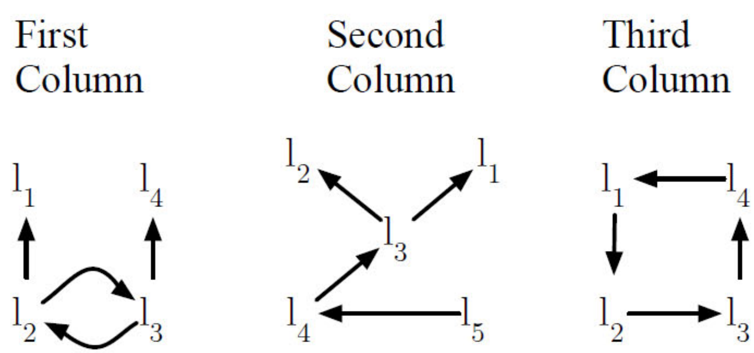
[Pereira&Palsberg 2010]

Parallel Copies

- $(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_n) \leftarrow (\mathbf{x}_{01}, \mathbf{x}_{11}, \ldots, \mathbf{x}_{n1})$
- Each parallel copy forms a "location transfer graph" [Pereira&Palsberg 2010]
  - edges in graph are the pairwise copies that need to be performed
- In LTG, in-degree is at most 1

Φ-matrix

$$\begin{bmatrix} l_1 \\ l_2 \\ l_3 \\ l_4 \end{bmatrix} = \Phi \begin{bmatrix} l_2 & l_3 & l_4 \\ l_3 & l_3 & l_1 \\ l_2 & l_4 & l_2 \\ l_3 & l_5 & l_3 \end{bmatrix}$$



Parallel Copies

- $(x_0, x_1, \ldots, x_n) \leftarrow (x_{01}, x_{11}, \ldots, x_{n1})$
- Each parallel copy forms a "location transfer graph" [Pereira&Palsberg 2010]
  - edges in graph are the pairwise copies that need to be performed
- In LTG, in-degree is at most 1
- If we spilled correctly (e.g., all Φ-related variables are spilled to same slot), then also:
  - out-degree of any node is at most 1
  - if node in graph is memory location, then

Spartan Transfer Graphs

- If we spilled correctly (e.g., all Φ-related variables are spilled to same slot), then also:
  - in-degree of any node is at most 1
  - out-degree of any node is at most 1
  - if node in graph is memory location, then
    - in-degree + out-degree is at most 1, or
    - edge on node is self-loop
- These graphs are "Spartan Transfer Graphs" [PP10]

Sequentializing Parallel Copies

- Each connected component forms
  - A Cycle
    (Then, all nodes are registers)
  - A Path
    (Then 1st may be memory store and/or last node may be memory load)
- Can implement as sequential code:
  - cycles use register swap
  - Paths use moves (mov, ld, st as appropriate)

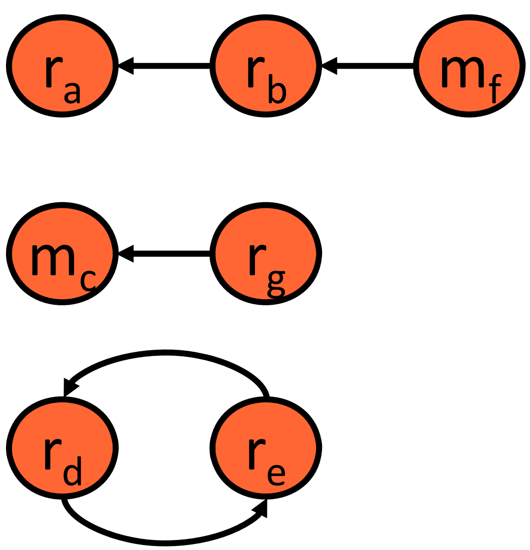
Parallel Copies

- $(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_n) \leftarrow (\mathbf{x}_{01}, \mathbf{x}_{11}, \ldots, \mathbf{x}_{n1})$
- Each parallel copy forms a “location transfer graph” [Pereira&Palsberg 2010]
  - edges in graph are the pairwise copies that need to be performed
- If we spilled correctly (e.g., all Φ-related variables are spilled to same slot), then LTG is either cycle or path
  - If cycle, only registers involved
  - If path and memory involved, then
    - 1^{st} copy may be store and last copy may be load

Example LTG to code

$$\begin{bmatrix} r_a \\ r_b \\ m_c \\ r_d \\ r_e \end{bmatrix} = \Phi \begin{bmatrix} \cdots & r_b & \cdots \\ \cdots & m_f & \cdots \\ \cdots & r_g & \cdots \\ \cdots & r_e & \cdots \\ \cdots & r_d & \cdots \end{bmatrix}$$

Creates LTG with 3 connected components



mov $r_a \leftarrow r_b$
ld $r_b \leftarrow m_f$

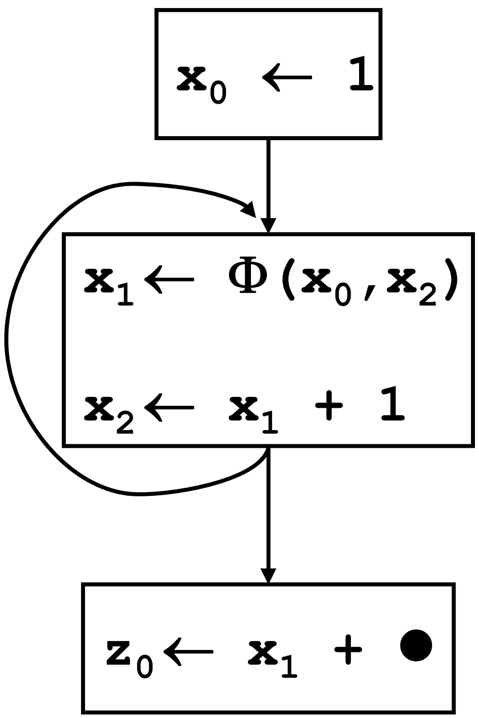
st $m_c \leftarrow r_g$

xchg $r_d \leftrightarrow r_e$

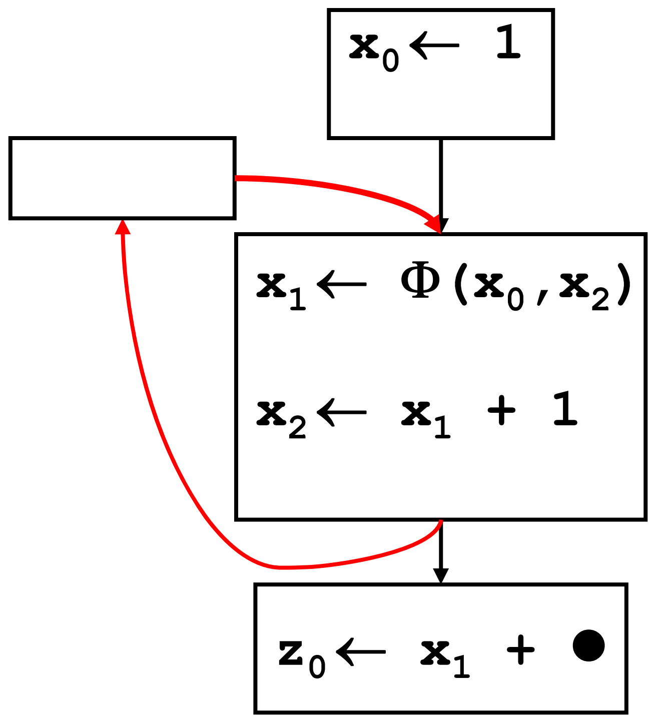
Putting it all together

- Critical Edge Splitting
- Convert back to Conventional-SSA (CSSA)
- Register Allocation
  - Build interference graph
  - pre-spilling
  - coloring
  - coalescing
- Deconstruct SSA
  - put parallel-copies in predecessors
  - Eliminate parallel copies

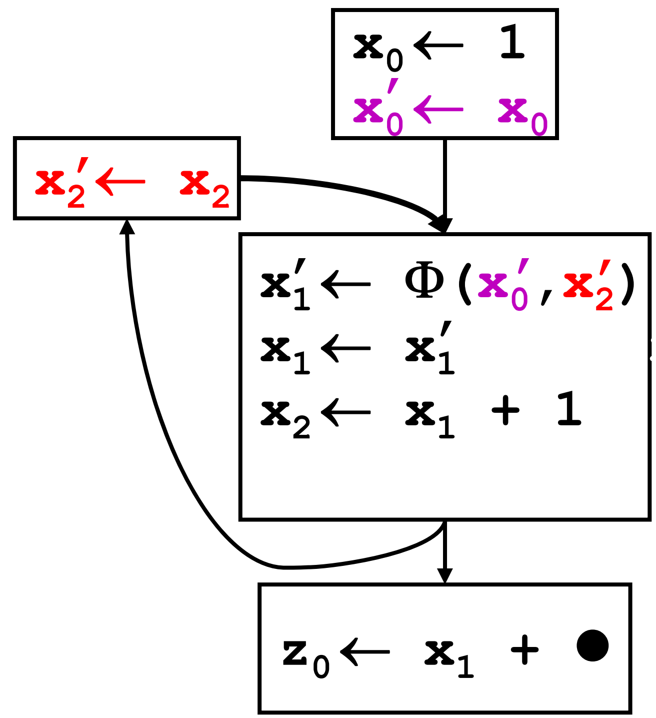
Example 1



Copy Folded

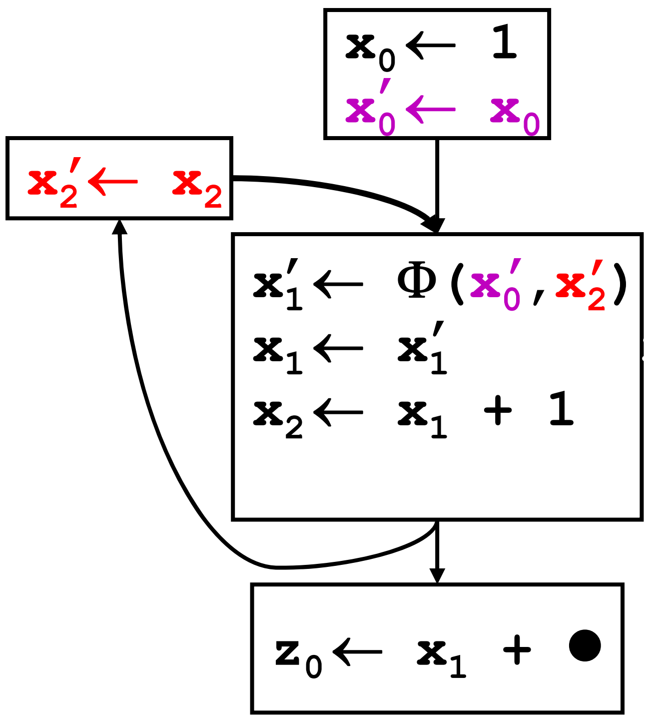


split critical edge

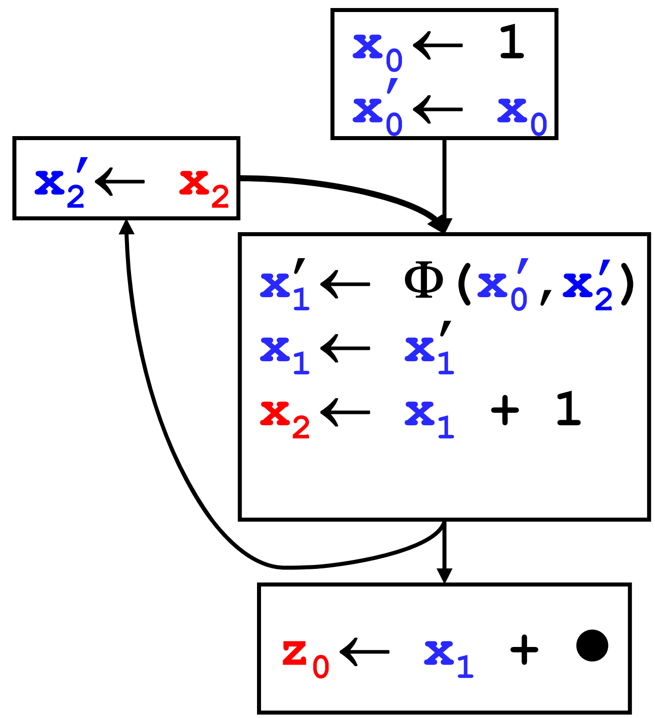


Convert to CSSA

Example 1



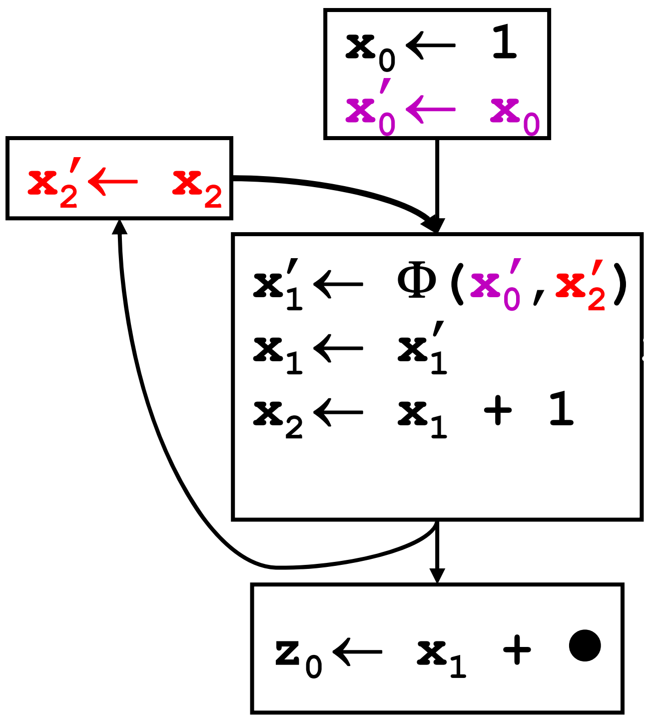
Convert to CSSA



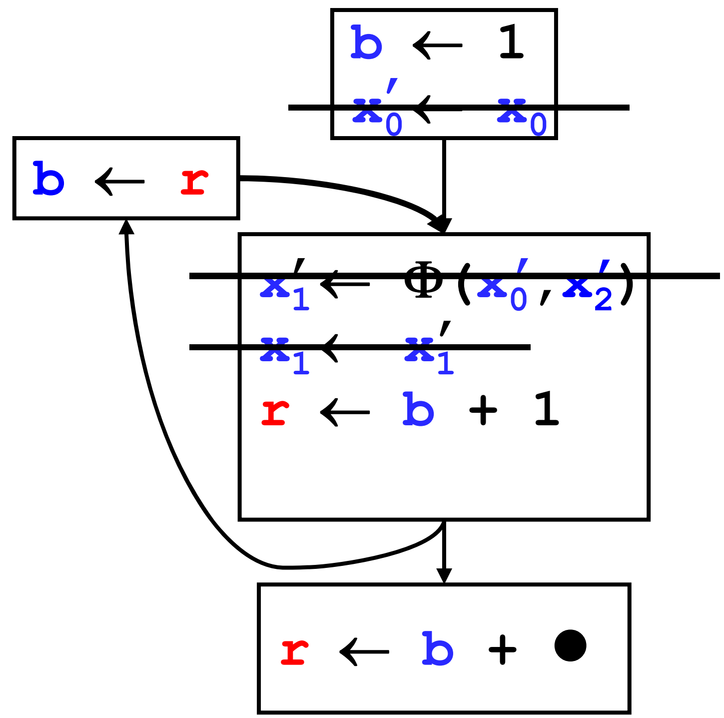
register allocation

Done, since $x \leftarrow \Phi(x,x)$ can simply be eliminated.

Well, we can clean up



Convert to CSSA



register allocation

Example 2

$x_0 \leftarrow \bullet$
$y_0 \leftarrow \bullet$

$x_1 \leftarrow \Phi(x_0, y_1)$
$y_1 \leftarrow \Phi(y_0, x_1)$
$z_0 \leftarrow y_1$
$y_2 \leftarrow x_1$
$x_2 \leftarrow z_0$

Some Fine Tuning

- We added LOTS and LOTS of copies.
- Can reduce added copies when
  - creating CSSA
  - Introducing parallel copies
- Can rely on coalescing, but also …

Reducing Copies Going to CSSA

- Only need to introduce copies if there is interference!
- As building interference graph mark nodes which are Φ-related. If edge between them, introduce copies and update interference graph.
- Can do even better if also do liveness checking, see [Sreedhar et al, 1999]

Reducing Stores for Spilling

- Every path from LTG that ends in a memory slot will produce a store.
  E.g., $a \rightarrow r_1 \rightarrow r_2 \ldots r_x \rightarrow m$ will create
  `st` $\mathbf{r_x \rightarrow m}$ at the end.
- But, only needs to be done once, e.g., at point of definition.
- So, eliminate store and change register allocator to insert store at definition point
- Similar elimination of loads possible. See [Pereira&Palsberg 2010]

Coalescing

- Coalescing becomes even more important.
- Perform before SSA deconstruction (focus on Φ-related variables)
- (See [Boissinot et.al. 2009])

Use SSA

- If you:
  - Using SSA throughout passes
  - Including register allocation
  - And, supporting code motion
- Counts as an optimization

NEW COURSE: LANGUAGE DESIGN & PROTOTYPING

17-396/17-696 – SPRING 2020

Little languages are everywhere! Would you like to – or do you need to – design your own?

In this course, you will:

- Learn how to **critique a language design**
- Practice several **language prototyping** approaches (interpreters, transpilers, fluent APIs)
- Apply techniques for **evaluating language designs with users**
- **Design and prototype your own language** in the final project

Prof. Jonathan Aldrich – T/Th 3-4:20

http://www.cs.cmu.edu/~aldrich/courses/17-396/

Tuesday, Guest Lecture