Dataflow Analysis Project

Zhaomo Yang

Overview

For part 2 and part 3 of the project, we are going to implement a generic dataflow analysis framework and three analyses based on it.

Part 2: **the dataflow analysis framework** and **reaching definition analysis**

Part 3: **liveness analysis** and **may-point-to analysis**

Difference between part 1 and part 2 & 3

Part 1:

- Figuring out how to use LLVM APIs

Part 2 & 3:

- Generic C++ programming
- Implementing algorithms and analyses we learned in the class

Part 2

- Dataflow analysis framework
- Reaching definition

Part 2

- **Dataflow analysis framework**
- Reaching definition

Dataflow Analysis Framework

Goal: implementing a ***generic*** bottom-up intra-procedural dataflow analysis framework

Dataflow Analysis Framework

A lattice is a tuple $(S, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ such that:

- $(S, \sqsubseteq)$ is a poset
- $\forall a \in S . \bot \sqsubseteq a$
- $\forall a \in S . a \sqsubseteq \top$
- Every two elements from S have a lub and a glb
- $\sqcup$ is the least upper bound operator, called a join
- $\sqcap$ is the greatest lower bound operator, called a meet

Dataflow Analysis Framework

A lattice is a tuple $(S, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ such that:

- $(S, \sqsubseteq)$ is a poset
- $\forall a \in S . \bot \sqsubseteq a$
- $\forall a \in S . a \sqsubseteq \top$
- Every two elements from S have a lub and a glb
- $\sqcup$ is the least upper bound operator, called a join
- $\sqcap$ is the greatest lower bound operator, called a meet

Dataflow Analysis Framework

$$(S, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$$

To instantiate a specific analysis, a ***generic*** dataflow analysis framework needs the six elements above.

Dataflow Analysis Framework

$$(S, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$$

To instantiate a specific analysis, a ***generic*** dataflow analysis framework needs the six elements above.

Since our framework is a ***bottom-up*** framework, do we need all of them?

Dataflow Analysis Framework

To instantiate a specific analysis, a ***generic*** dataflow analysis framework needs the six elements above.

Since our framework is a ***bottom-up*** framework, do we need all of them?

Dataflow Analysis Framework

We provide an incomplete base template class ***DataFlowAnalysis***. To create a dataflow analysis based on the base class ***DataFlowAnalysis***, you need to provide:

- class Info: the class that represents the information at each program point; $S \sqsubseteq \sqcup$
- bool Direction: the direction of analysis. If it is true, then the analysis is a forward analysis; otherwise it is a backward analysis.
- Info InitialState: the input of the very first instruction of the analysis.
- Info Bottom: the bottom of the lattice. $\bot$

```
/*
 * This is the base template class to represent the generic dataflow
 * analysis framework.
 * For a specific analysis, you need to create a sublcass of it.
 */
template <class Info, bool Direction>
class DataFlowAnalysis {

  private:
        typedef std::pair<unsigned, unsigned> Edge;
        // Index to instruction map
        std::map<unsigned, Instruction *> IndexToInstr;
        // Instruction to index map
        std::map<Instruction *, unsigned> InstrToIndex;
        // Edge to information map
        std::map<Edge, Info *> EdgeToInfo;
        // The bottom of the lattice
        Info Bottom;
        // The initial state of the analysis
        Info InitialState;
        // EntryInstr points to the first instruction
        // to be processed in the analysis
        Instruction * EntryInstr;
```

Dataflow Analysis Framework

We provide an incomplete base template class ***DataFlowAnalysis***. To create a dataflow analysis based on the base class ***DataFlowAnalysis***, you need to provide:

- **class Info:** the class that represents the information at each program point; $S \sqsubseteq \sqcup$
- **bool Direction:** the direction of analysis. If it is true, then the analysis is a forward analysis; otherwise it is a backward analysis.
- Info InitialState: the input of the very first instruction of the analysis.
- Info Bottom: the bottom of the lattice. $\bot$

```
/*
 * This is the base template class to represent the generic dataflow
 * analysis framework.
 * For a specific analysis, you need to create a sublcass of it.
 */
template <class Info, bool Direction>
class DataFlowAnalysis {
  ...
};
```

To instantiate a forward analysis called MyForwardAnalysis

```
class MyForwardAnalysis : public DataFlowAnalysis<MyInfo, true> {
  ...
};
```

Dataflow Analysis Framework

We provide an incomplete base template class ***DataFlowAnalysis***. To create a dataflow analysis based on the base class ***DataFlowAnalysis***, you need to provide:

- class Info: the class that represents the information at each program point;

$$S \sqsubseteq \sqcup$$

- bool Direction: the direction of analysis. If it is true, then the analysis is a forward analysis; otherwise it is a backward analysis.
- **Info InitialState:** the input of the very first instruction of the analysis.
- **Info Bottom:** the bottom of the lattice $\perp$

```
DataFlowAnalysis(Info & bottom, Info & initialState);
```

To instantiate a forward analysis called MyForwardAnalysis

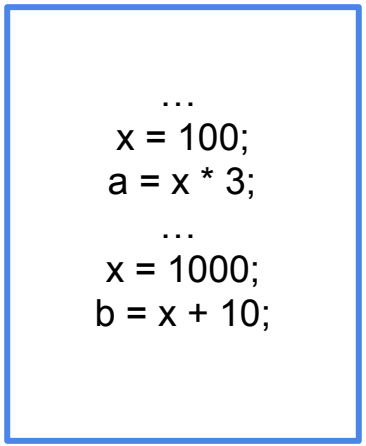
```
MyForwardAnalysis(MyInfo & InitialStates, MyInfo & Bottom);
```

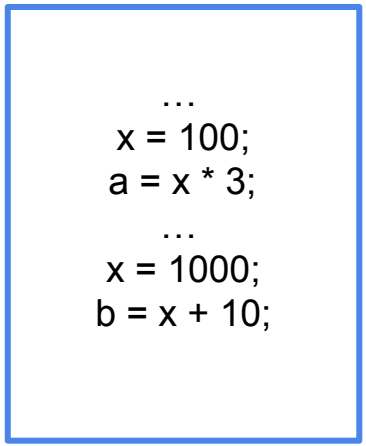
Static Single Assignment (SSA)

Static Single Assignment (SSA) is a form of program in which

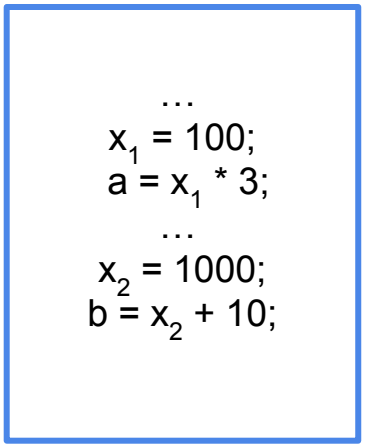
- each variable is assigned exactly once, and
- every variable is defined before it is used

SSA can enable many optimizations so many compiler IRs are in the SSA form.

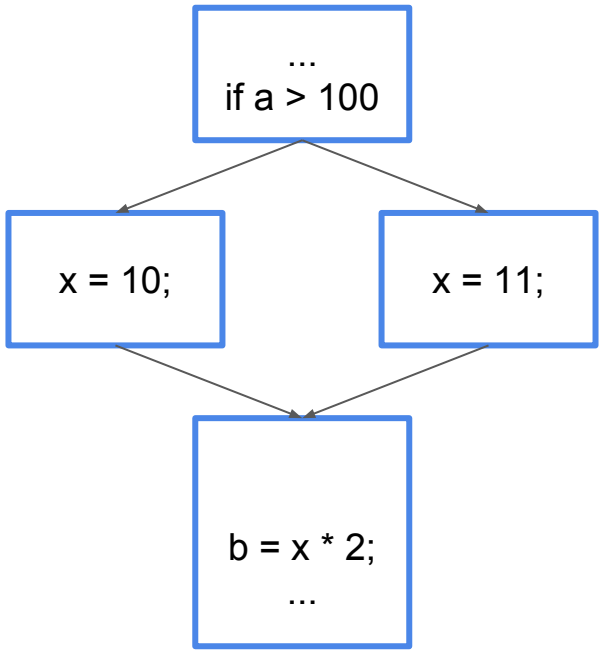

…
x = 100;
a = x * 3;
…
x = 1000;
b = x + 10;

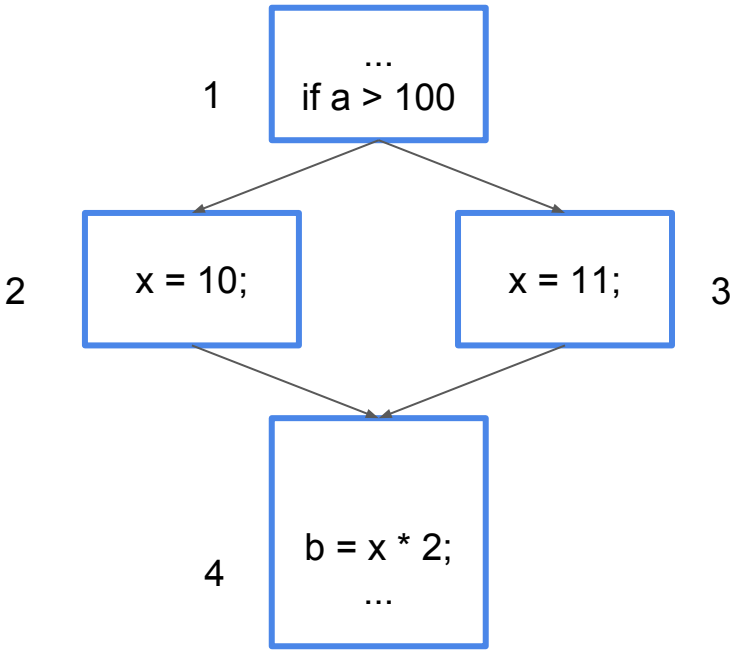


Non-SSA form

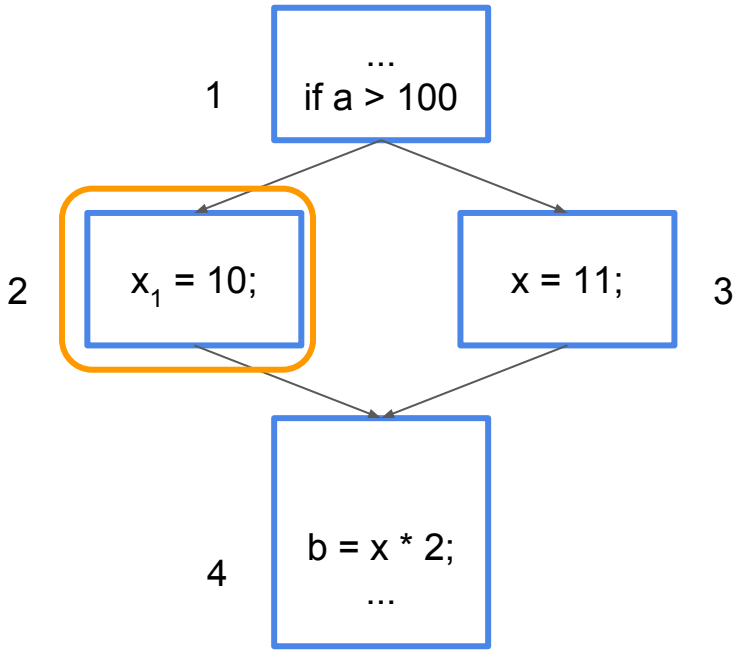


SSA form

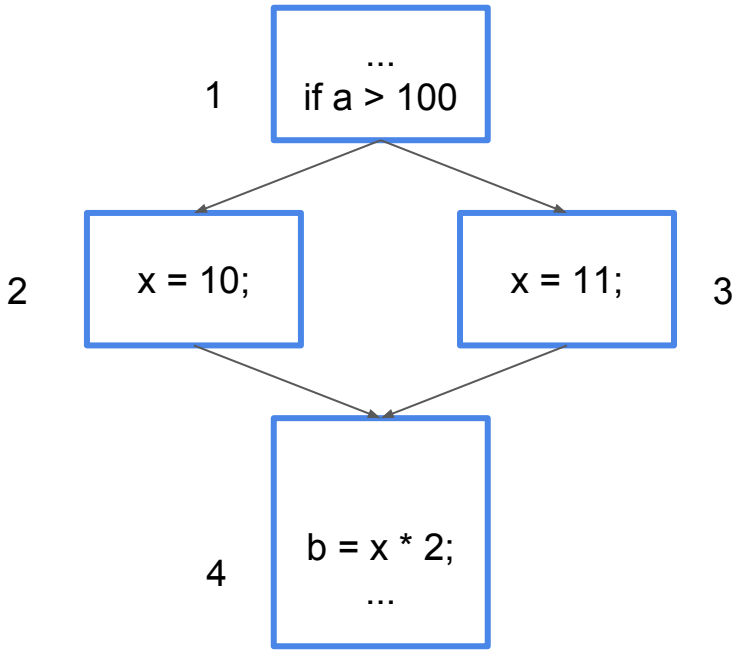

...
if a > 100
x = 10;
x = 11;
b = x * 2;
...



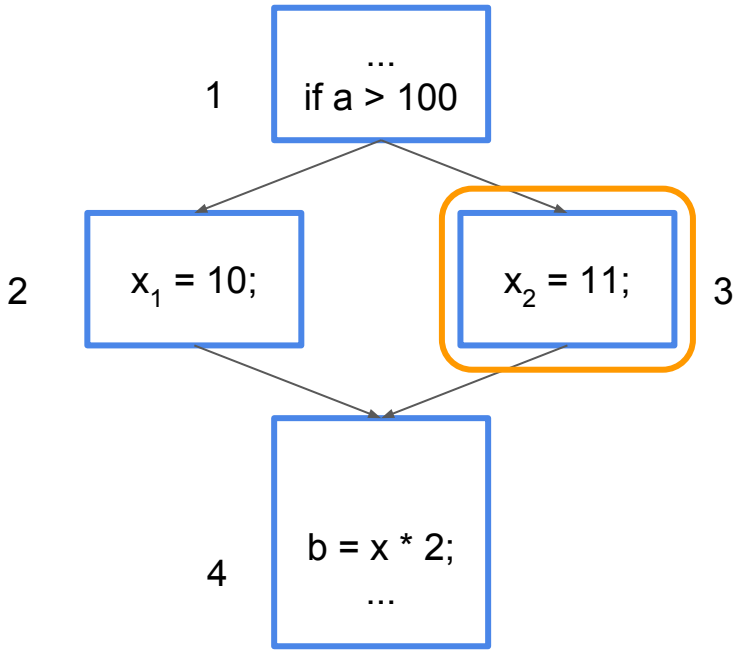
Non-SSA form



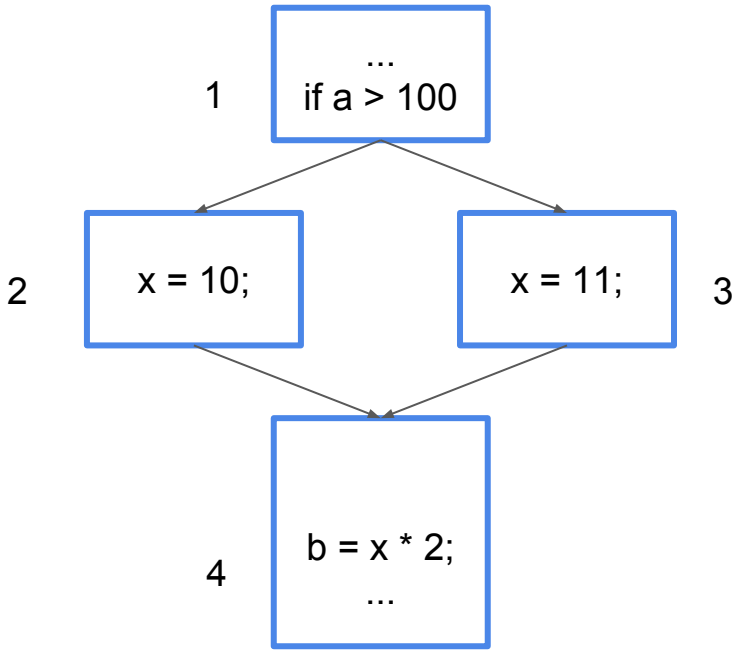
SSA form



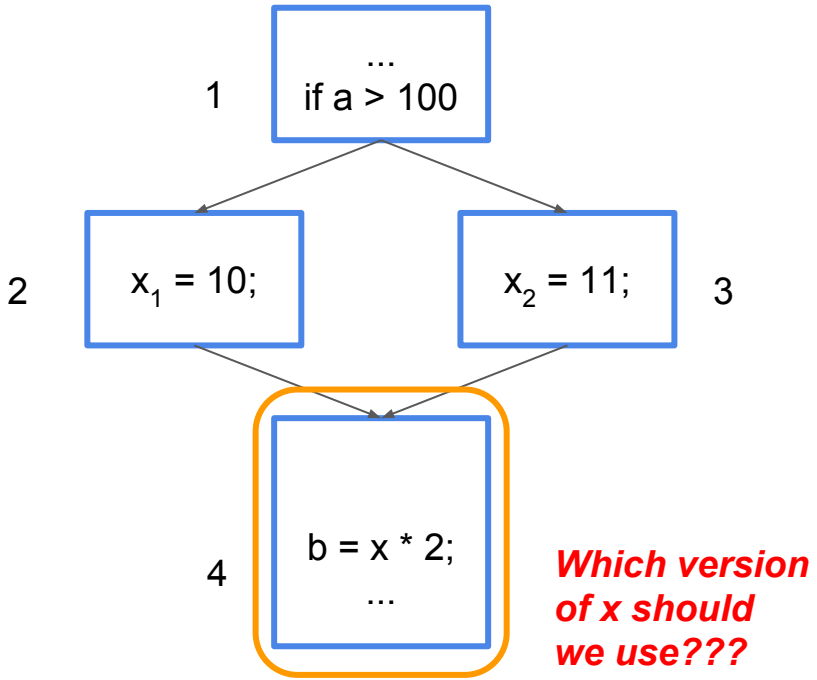
Non-SSA form



SSA form



Non-SSA form



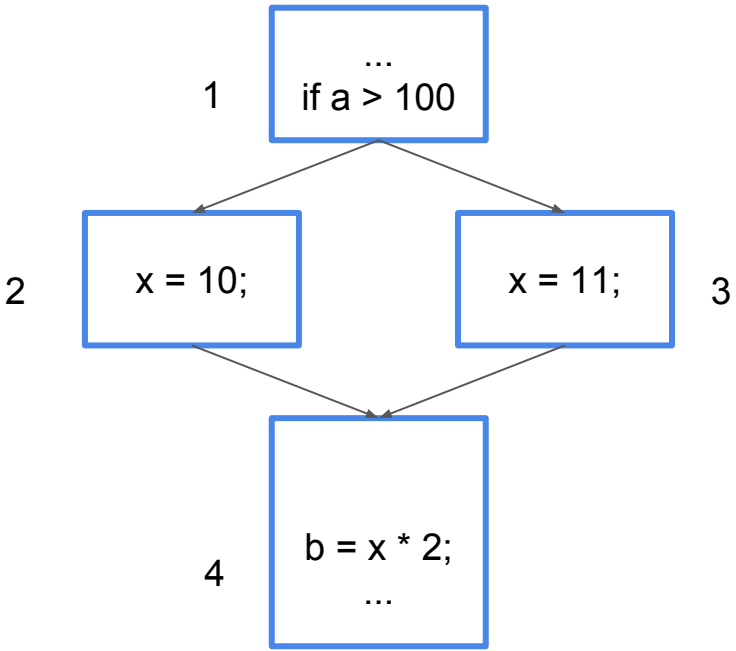
SSA form

Static Single Assignment (SSA)

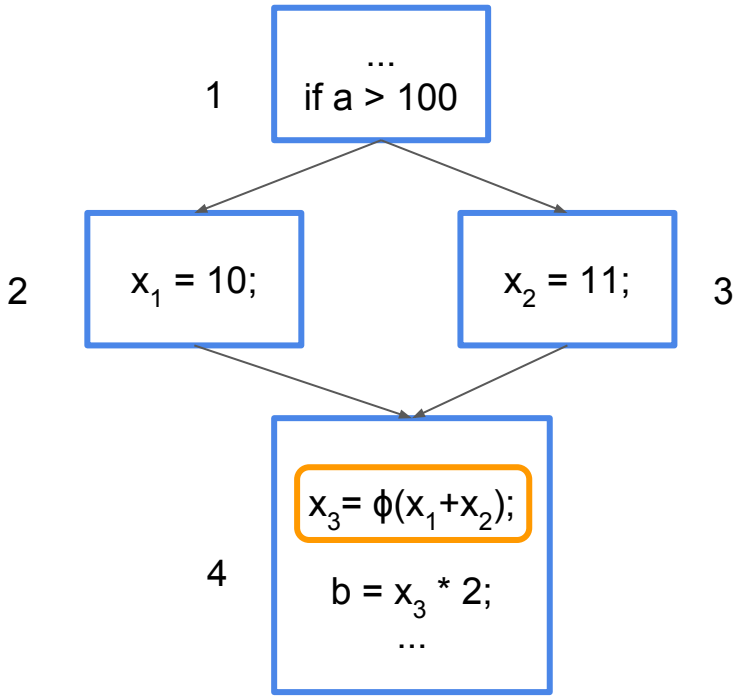
A form of program that can enable many optimizations

- each variable is assigned exactly once
- every variable is defined before it is used
- Insert ϕ (phi) nodes at merge points

Phi nodes can magically synthesize values from different sources.



Non-SSA form



SSA form

LLVM IR and SSA

LLVM IR is in the SSA form

- Each variable is only defined by one instruction
- Each instruction can define at most one variable
- We assign an index to each instruction

We can use instruction indices to refer to variables. That is, variable $\mathbf{x}$ is the value defined by the instruction whose index is $\mathbf{x}$.

LLVM IR and SSA

LLVM IR is in the SSA form

- Each variable is only defined by one instruction
- Each instruction can define at most one variable
- We assign an index to each instruction

The `phi` instruction from LLVM IR represents phi nodes in LLVM IR.

Control Flow Graph

There are two kinds of control flow graphs:

- LLVM CFGs
- DFA CFGs

Control Flow Graph

There are two kinds of control flow graphs:

- LLVM CFGs: built by LLVM.

An LLVM CFG of a function is available when your function pass is running on that function.

Control Flow Graph

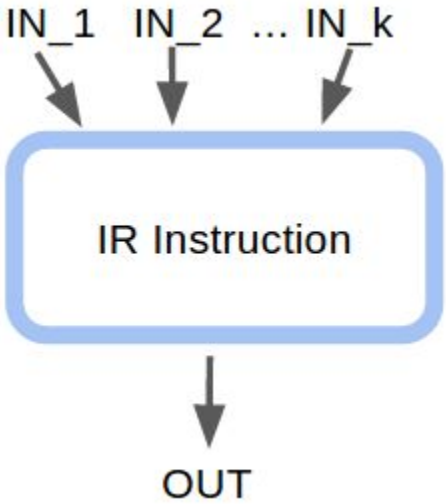
There are two kinds of control flow graphs:

- DFA CFGs: CFGs built and used by the dataflow analysis framework.

initializeForwardMap(Function * func) in 231DFA.h builds a DFA CFG of the given function func for **forward analyses**.

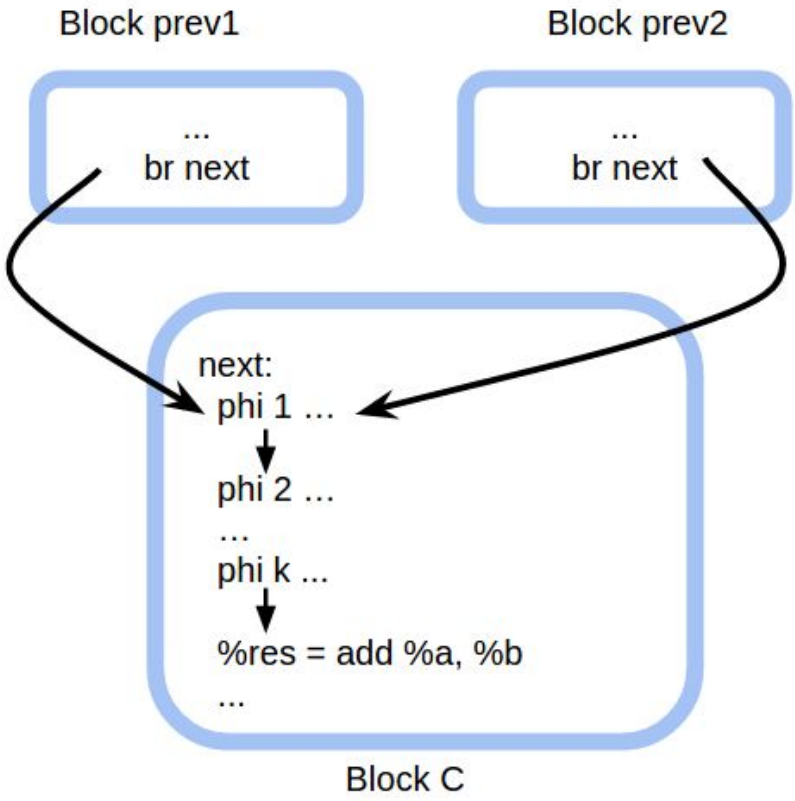
LLVM CFG VS. DFA CFG

First, in a DFA CFG any instruction may have more than one incoming data flows, as is shown below

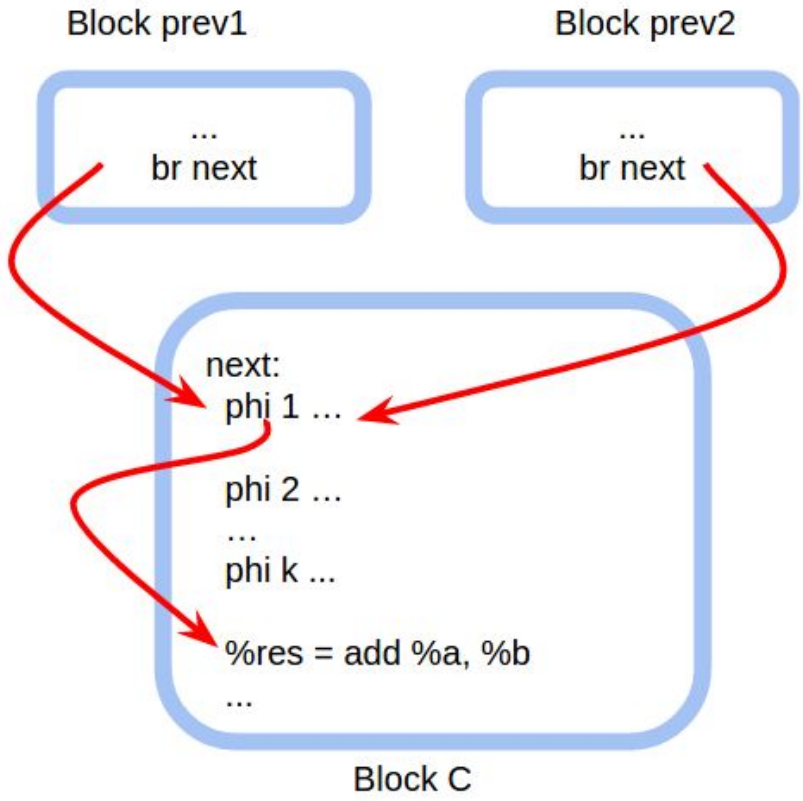


LLVM CFG VS. DFA CFG

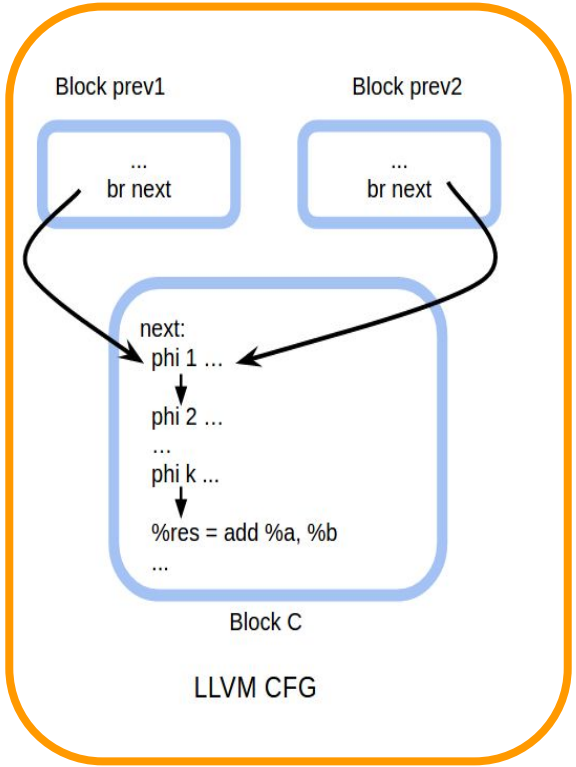
Second, in a DFA CFG, the phi instructions at the beginning of a merging basic block are treated as a unit.

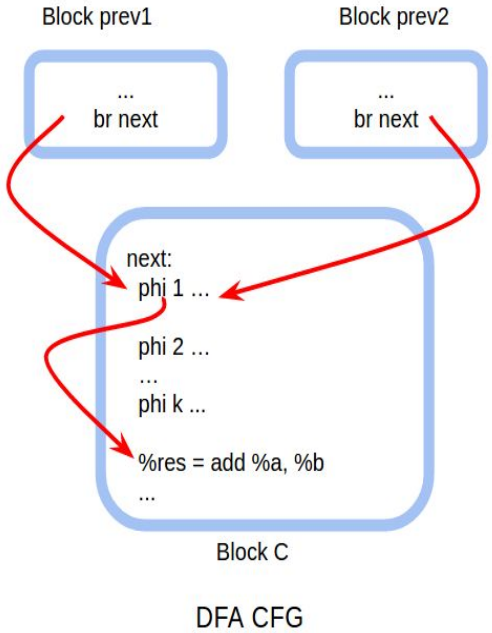


LLVM CFG



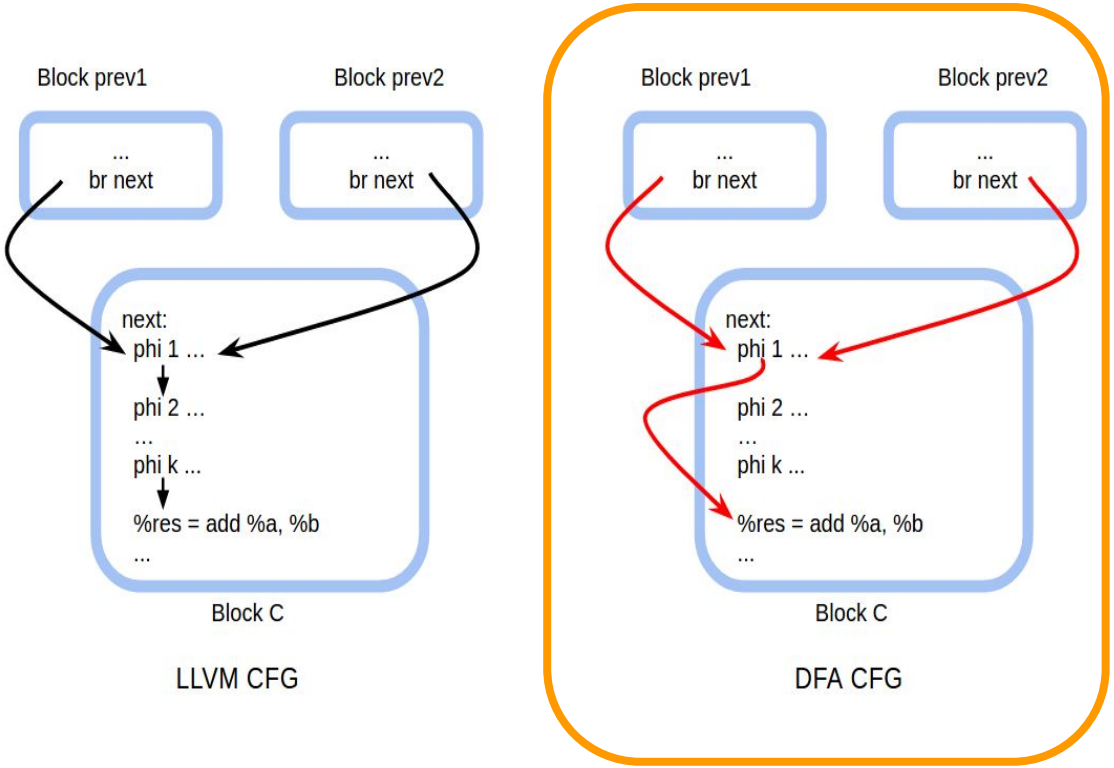
DFA CFG



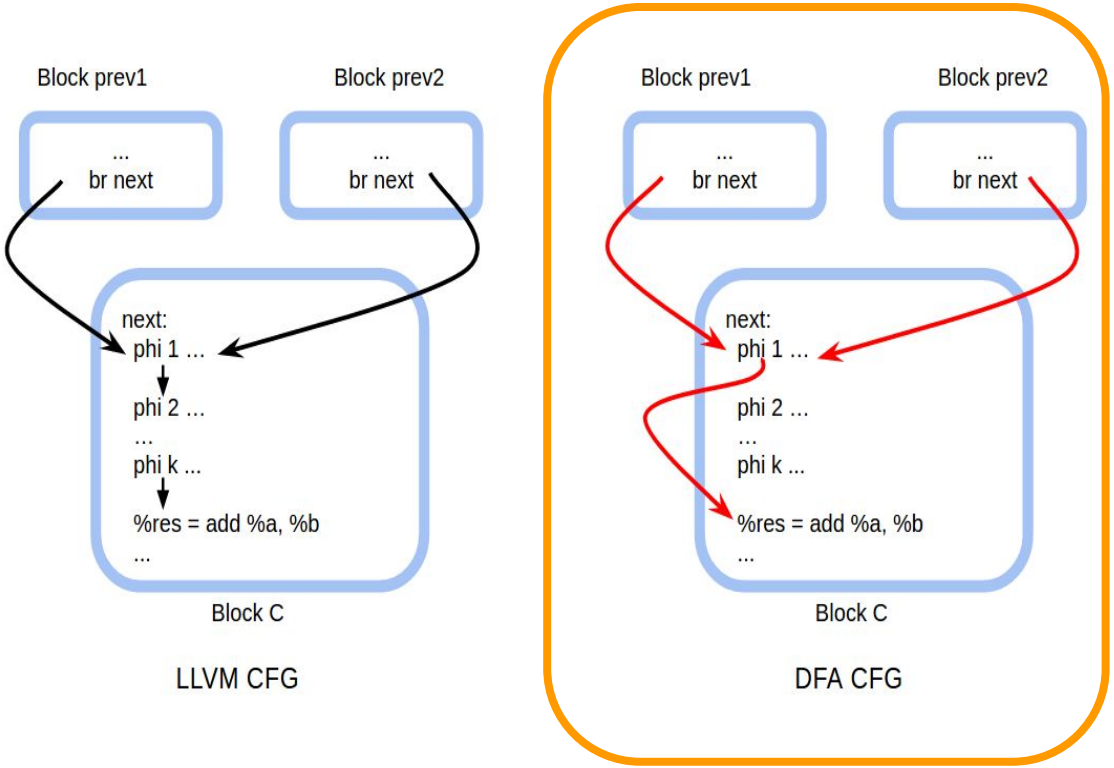


There are k phi instructions at the start of the basic block C.

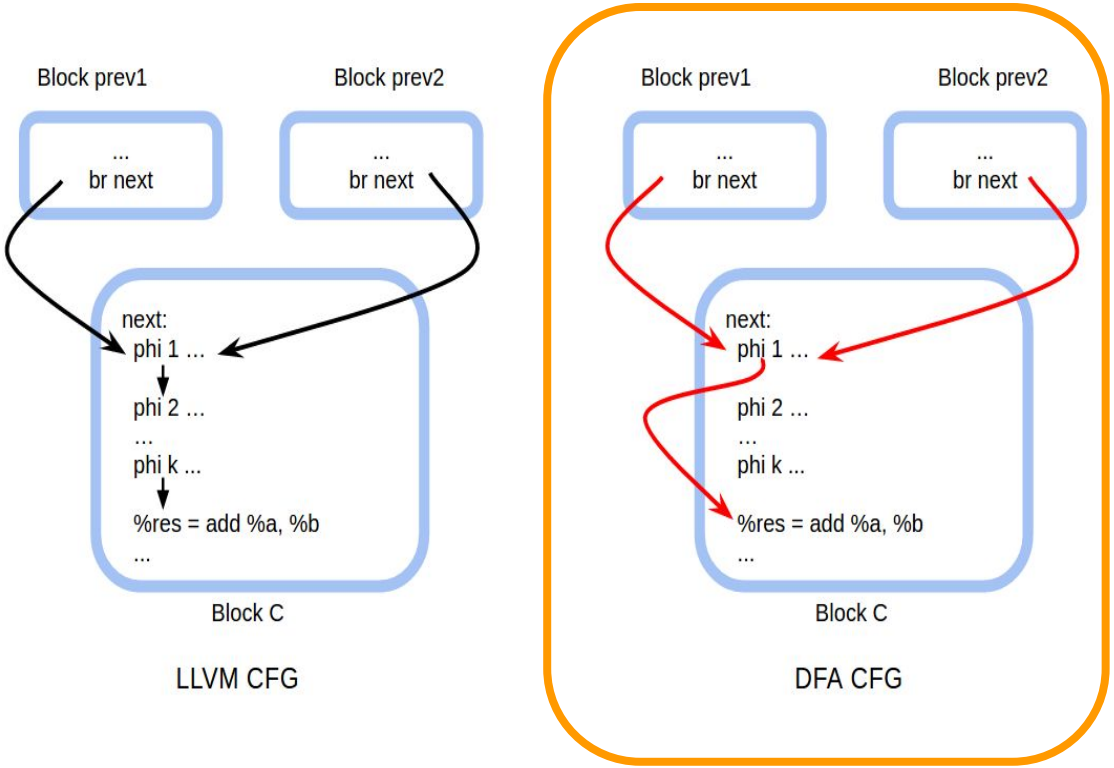
In the LLVM CFG, these phi instructions are connected sequentially.



In the DFA CFG, **phi 1** has an outgoing edge connecting to the first non-phi instruction **%res = add %a, %b** directly.



When encountering a phi instruction, the flow function should process the series of phi instructions together (effectively a PHI node from the lecture) rather than process each phi instruction individually.



This means that the flow function needs to look at the LLVM CFG to iterate through all the later phi instructions at the beginning of the same basic block until the first non-phi instruction.

Only one correct answer

You have to use the ***exact*** flow functions we specified in the webpage.

Given these flow functions, you have to get the ***most precise*** result for each edge.