(Correctly) Extending Dominance to MLIR Regions

Jeff Niu, Siddharth Bhat

US LLVM Developers Meeting, 2023

Agenda

What is dominance & why is it important?

→ Key property of SSA (liveness & reachability).

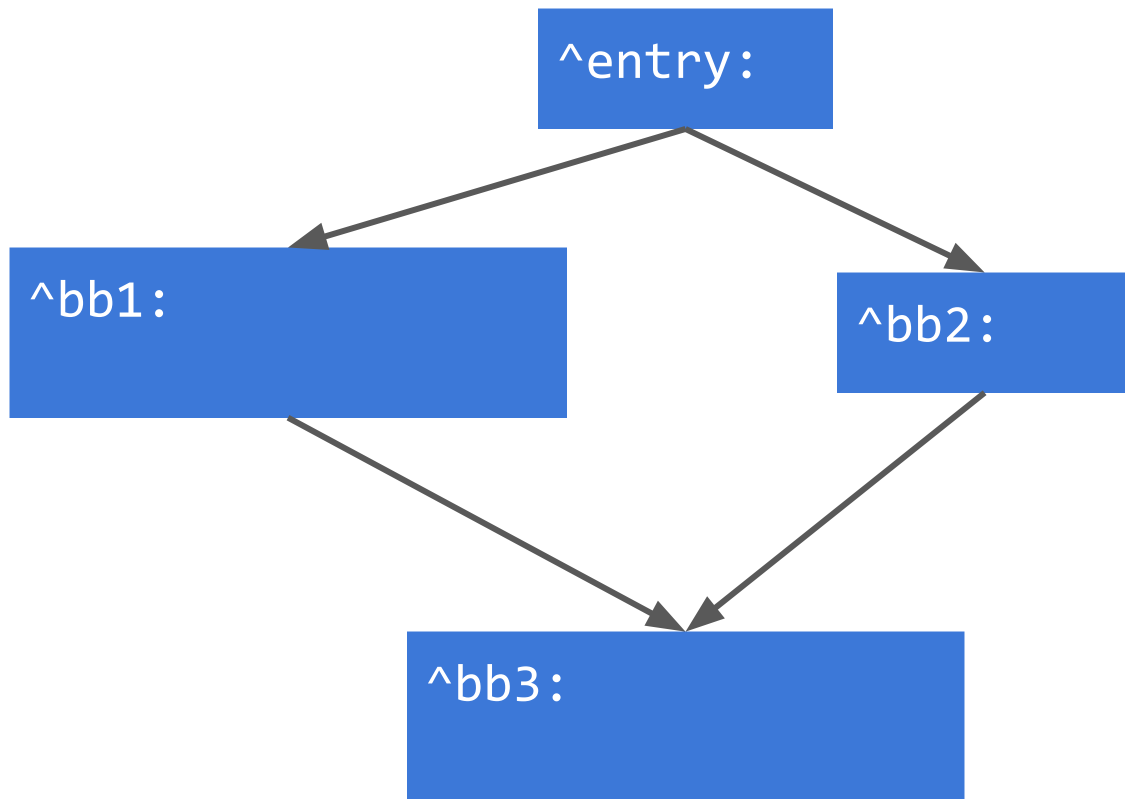
```
Instruction does not dominate all uses!
  %x = add i32 %6, 10
  store i32 %x, i32* %3, align 4
```

→ Dominance, Dominator Tree in LLVM.

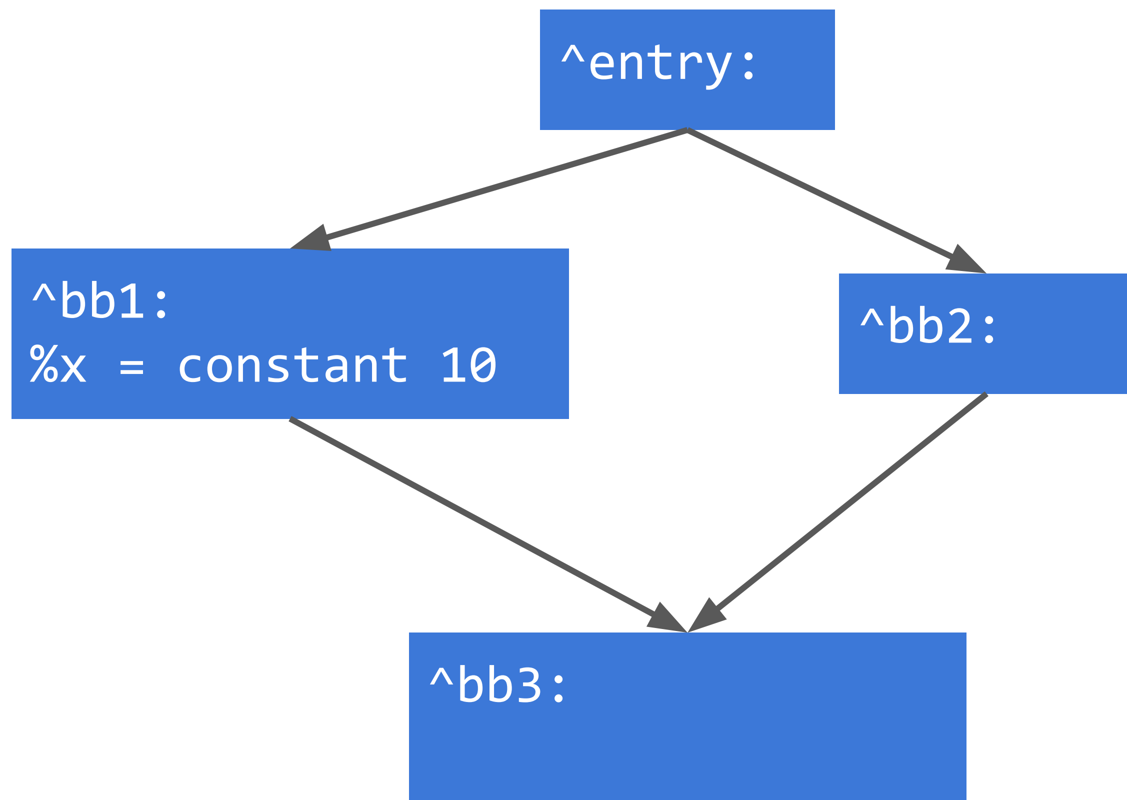
→ Regions, Dominators in MLIR, its problems.

→ Potential Solutions.

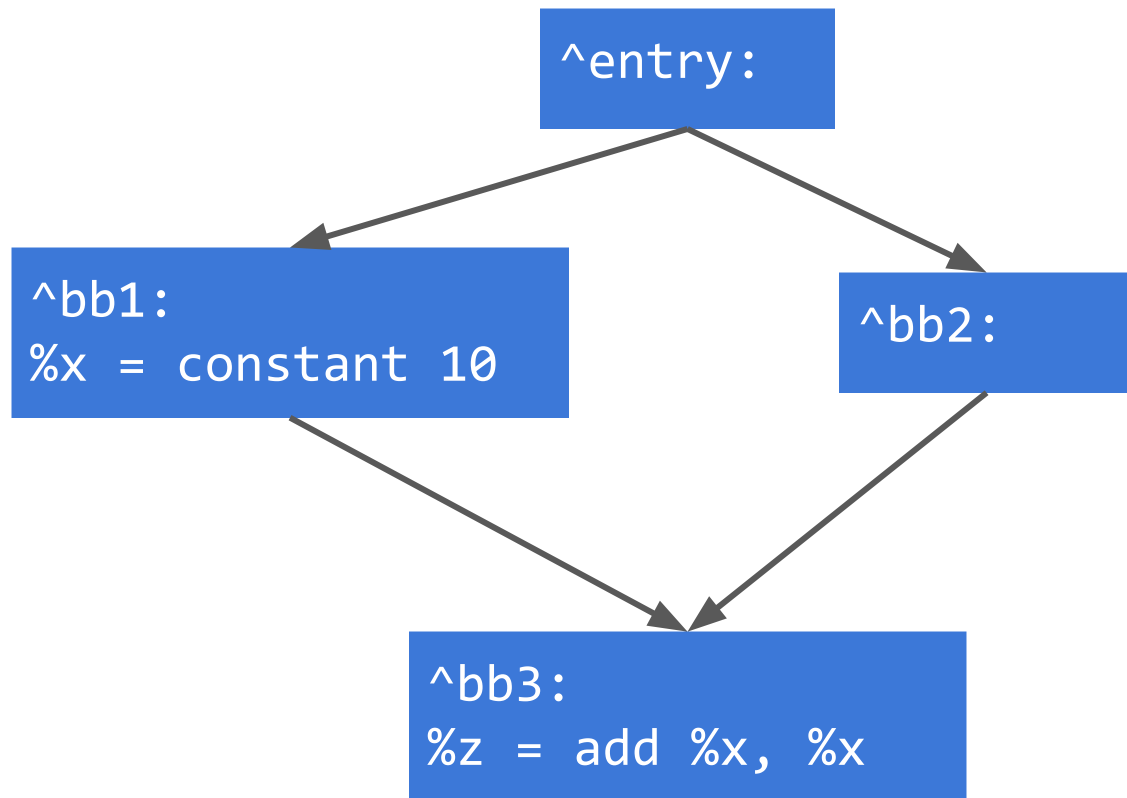
Dominance in LLVM: Legality



Dominance in LLVM: Legality

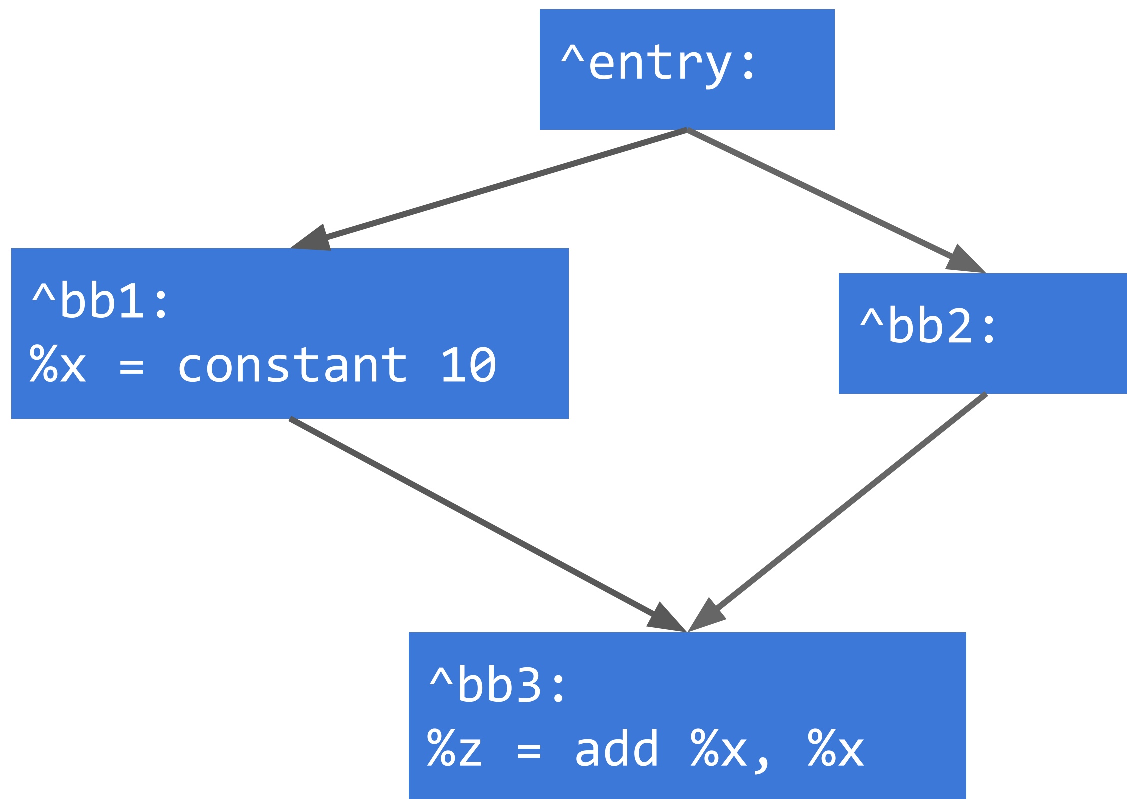


Dominance in LLVM: Legality

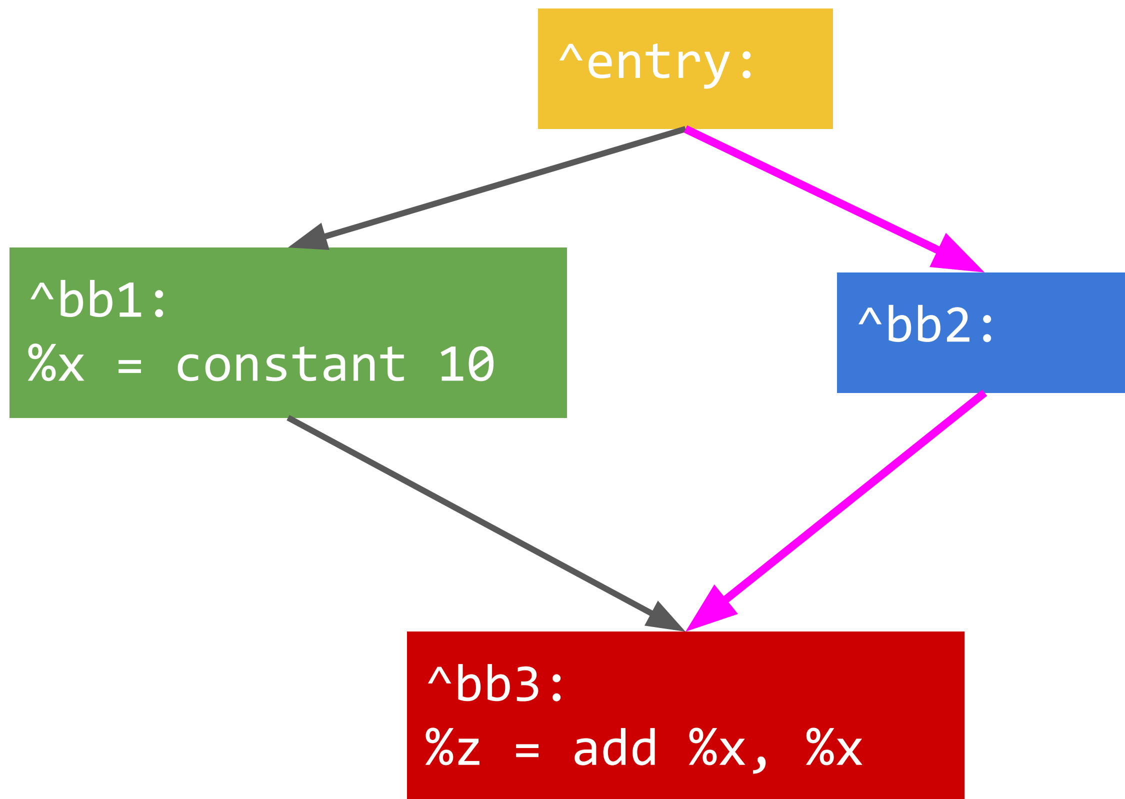


Dominance in LLVM: Legality

Is this IR legal?

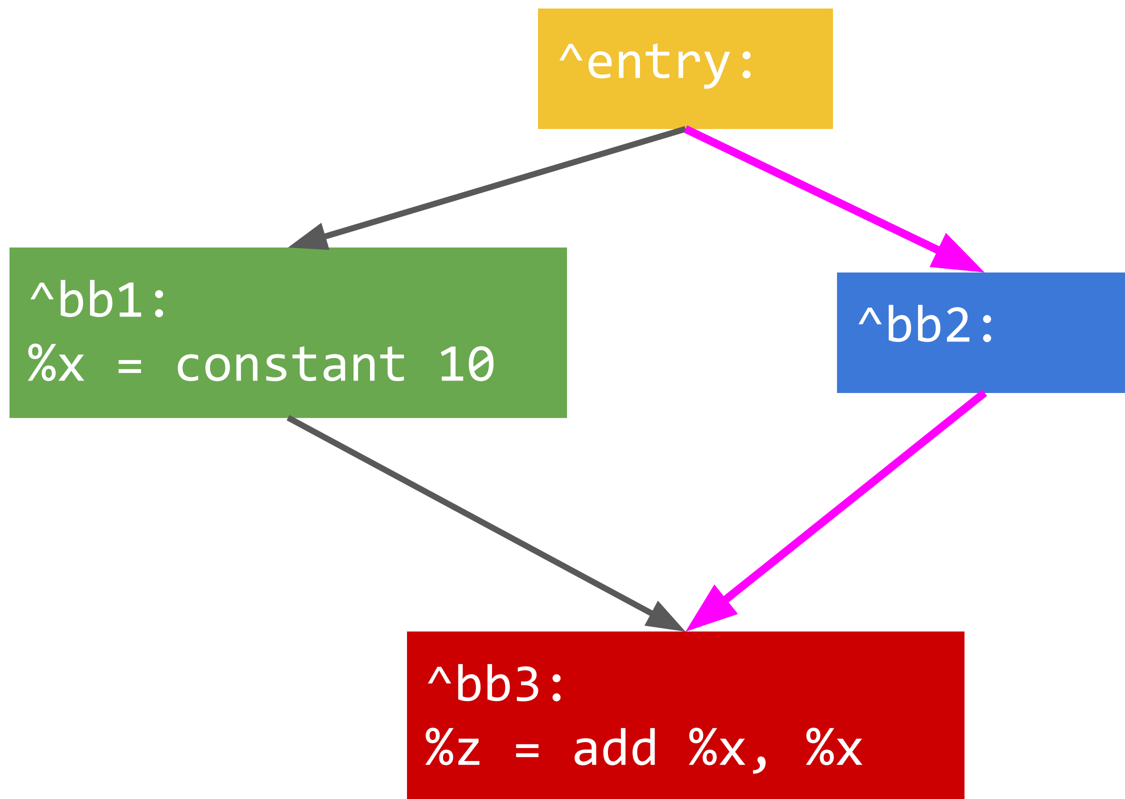


Dominance in LLVM: Legality



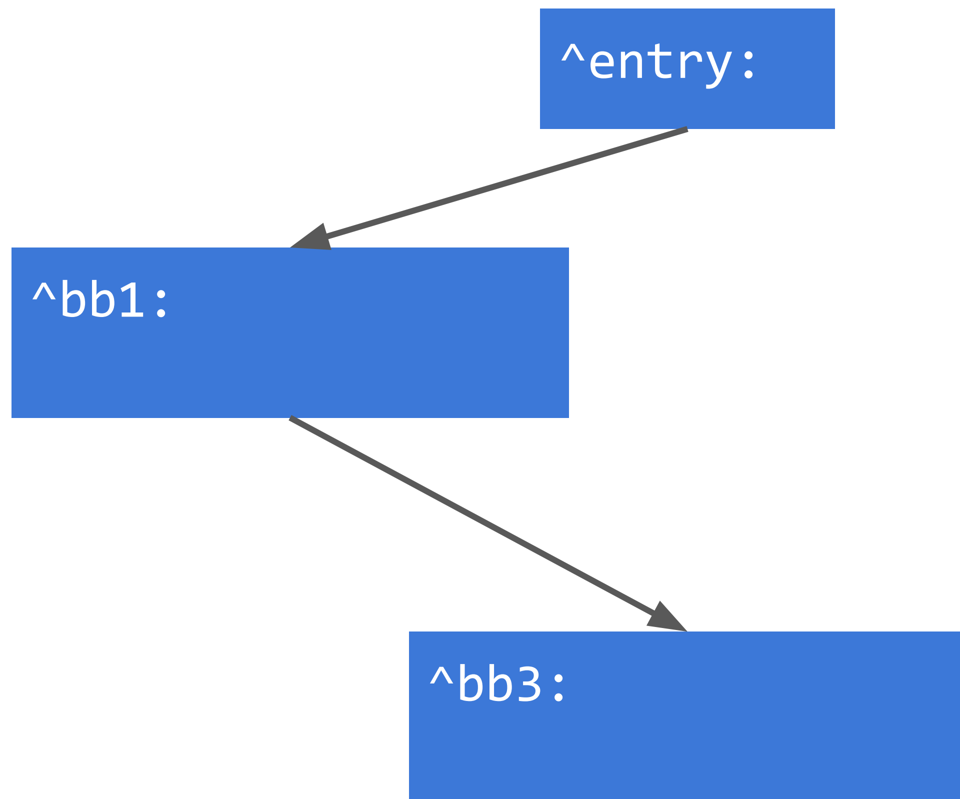
%x **unavailable**
in path
entry-bb2-bb3
=> illegal

Dominance in LLVM: Legality

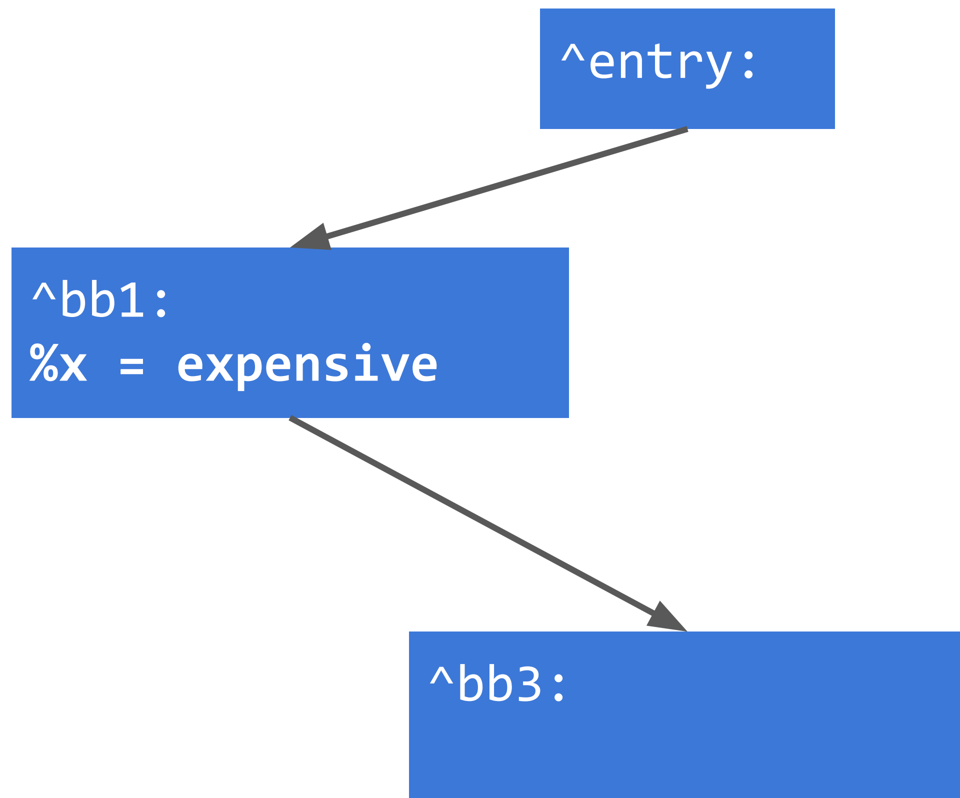


^bb1 doesn't dominate **^bb3**
=> illegal

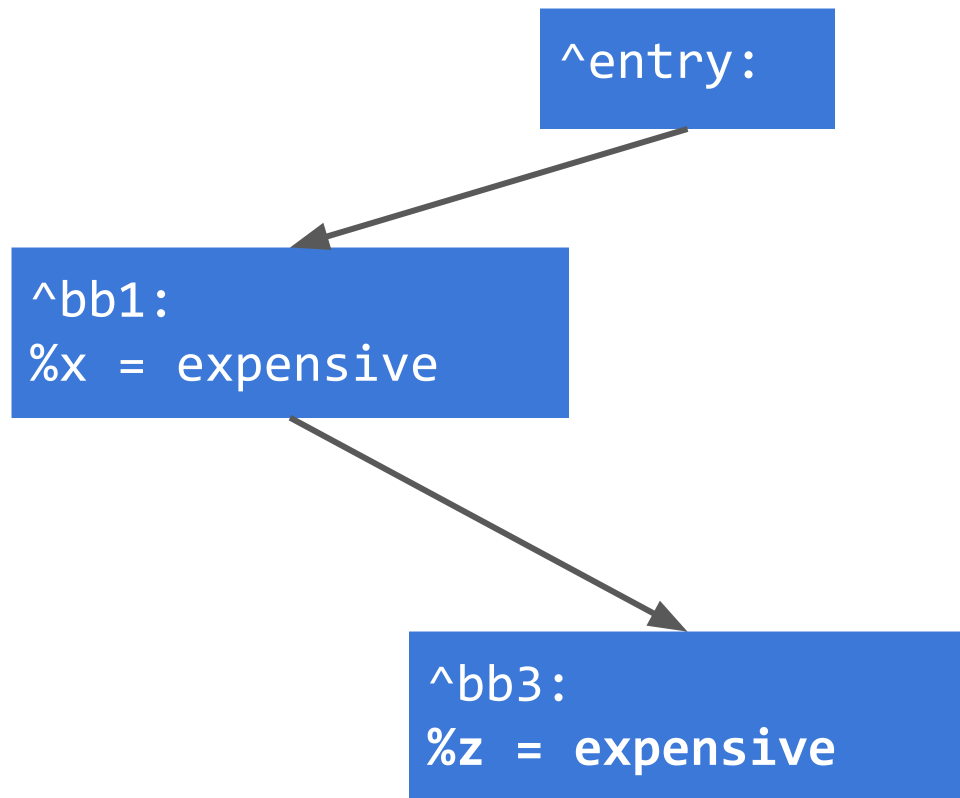
Dominance in LLVM: CSE: Example



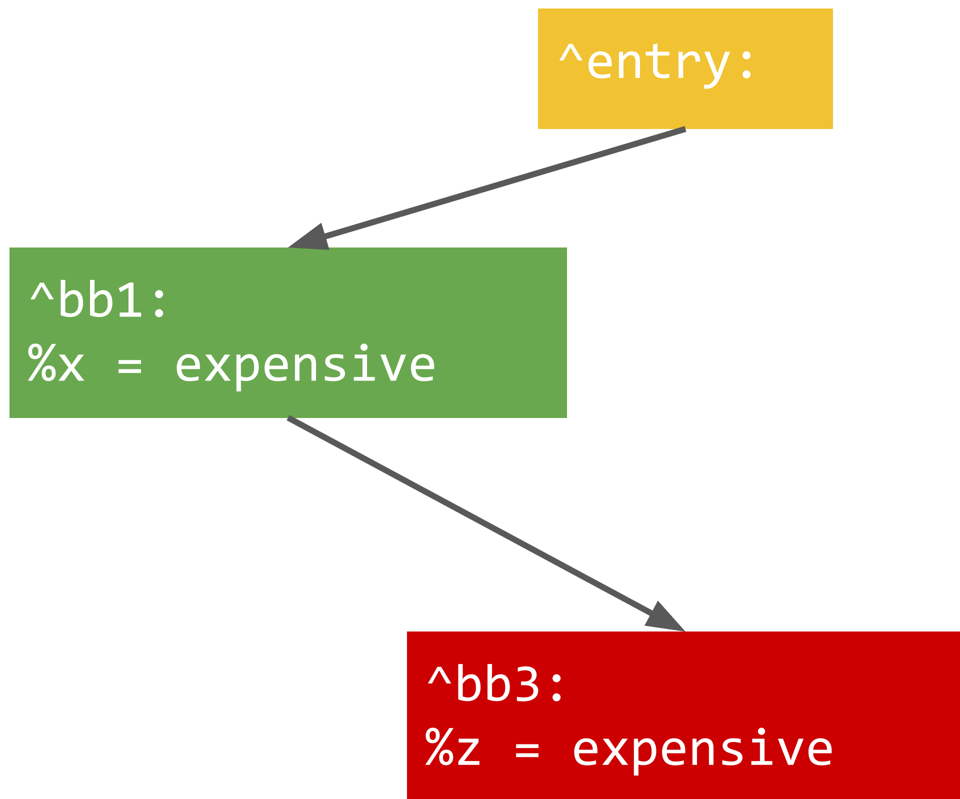
Dominance in LLVM: CSE: Example



Dominance in LLVM: CSE: Example

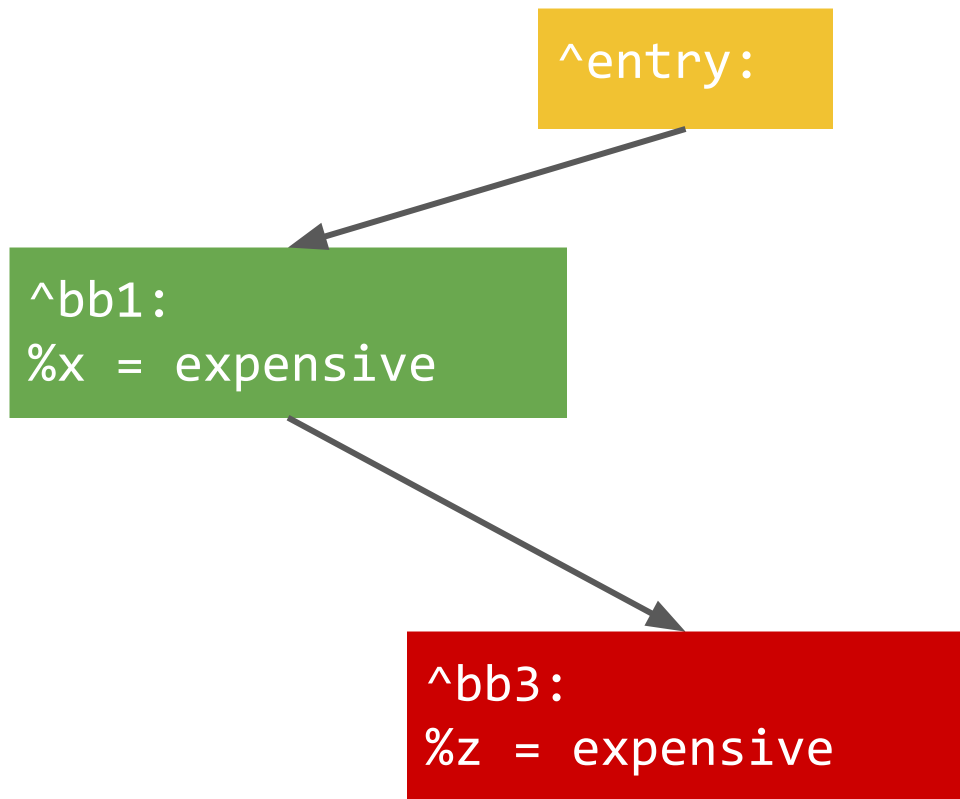


Dominance in LLVM: CSE: Example



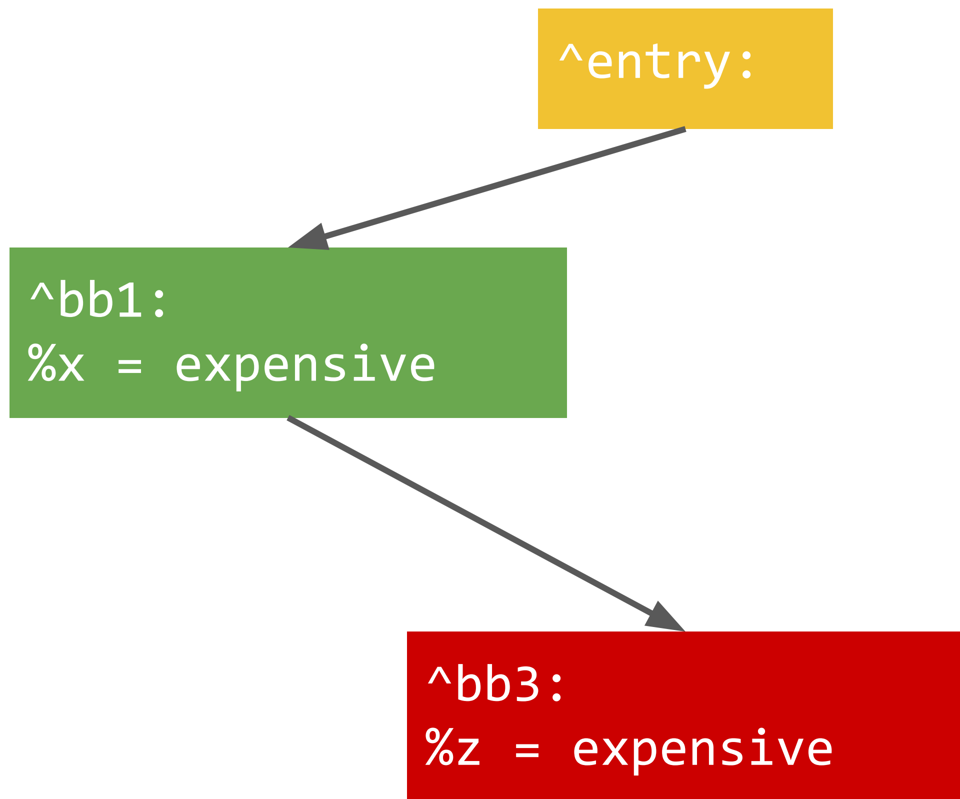
%x **available**
when reaching %z

Dominance in LLVM: CSE: Example



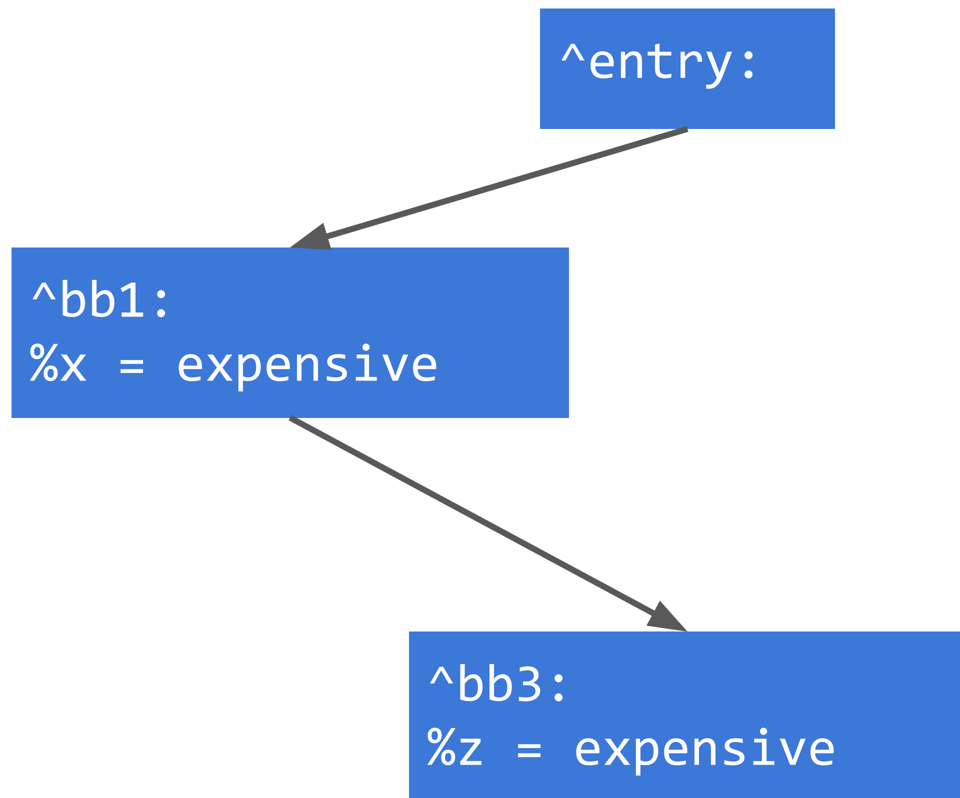
%x **available**
when reaching %z
=> CSE legal

Dominance in LLVM: CSE: Example

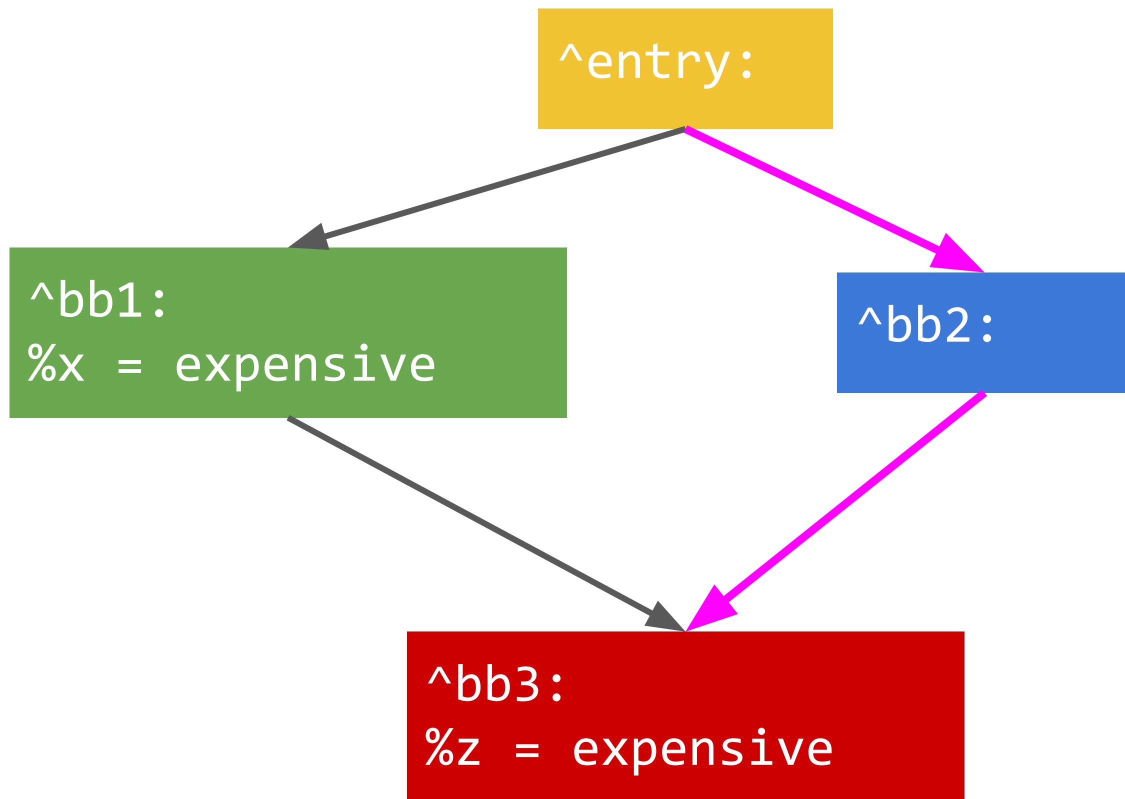


bb1 dominates bb3
=> CSE legal

Dominance in LLVM: CSE: Non Example

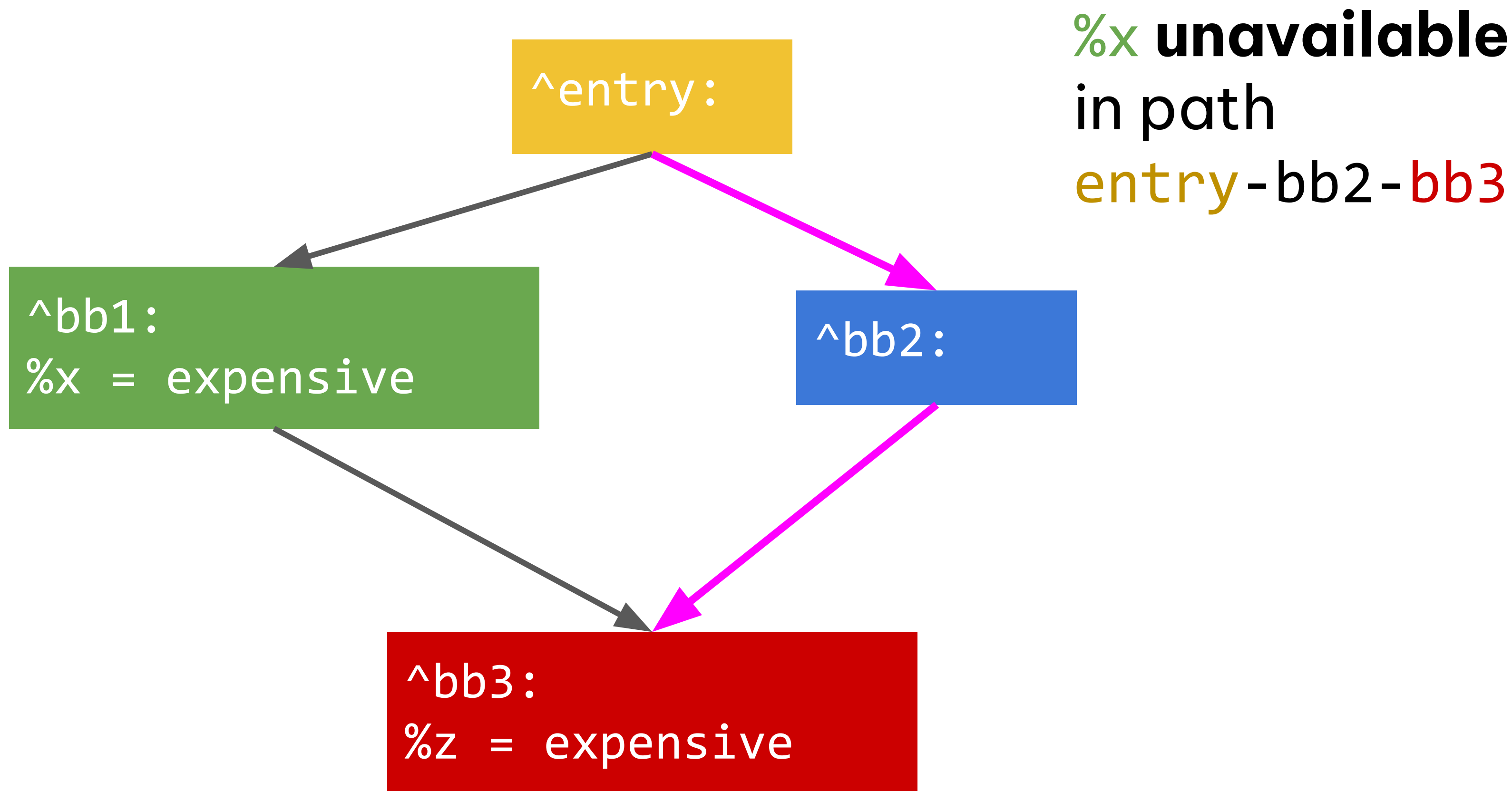


Dominance in LLVM: CSE: Non Example



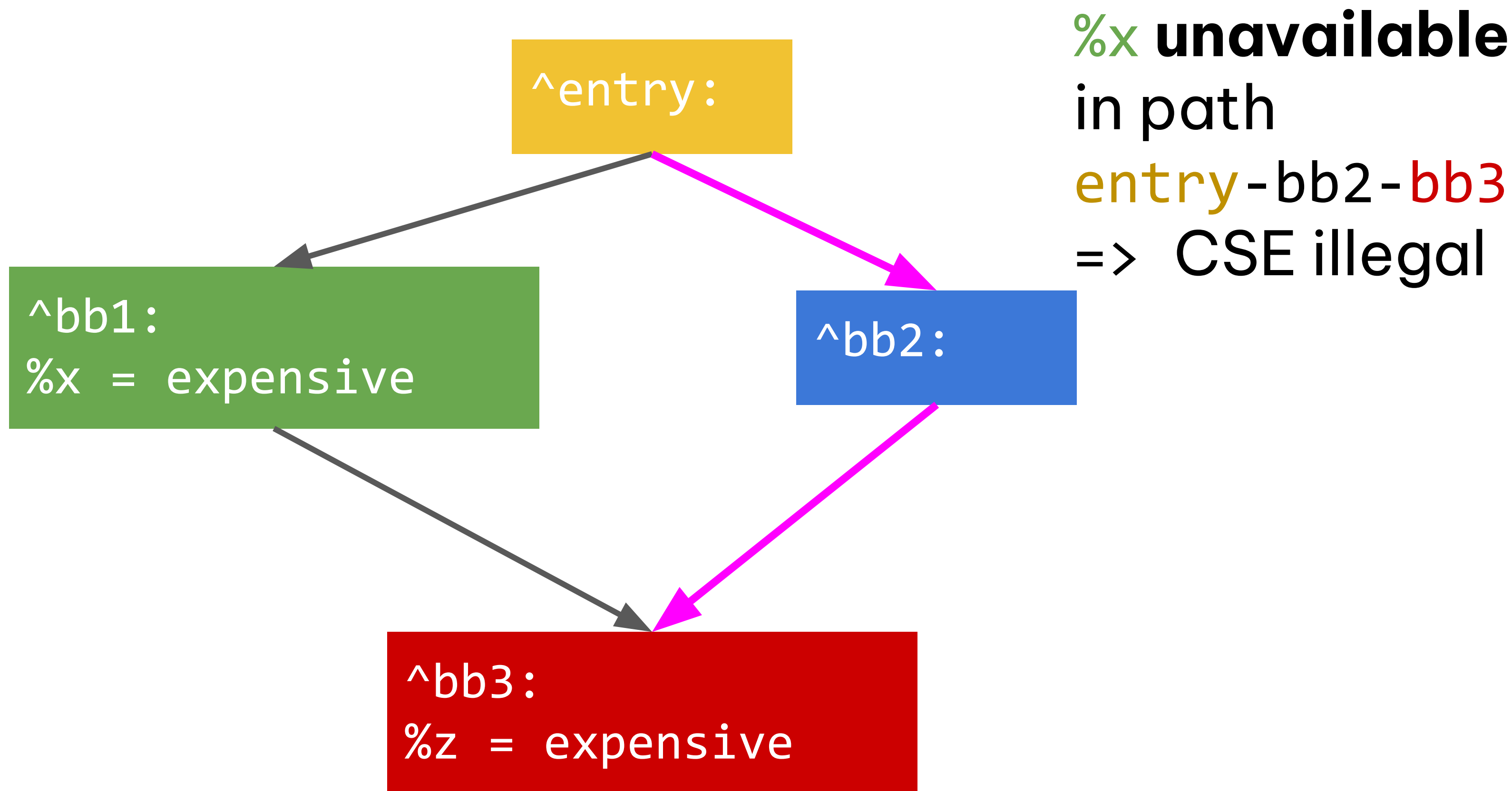
Can we replace %z with %x?

Dominance in LLVM: CSE: Non Example



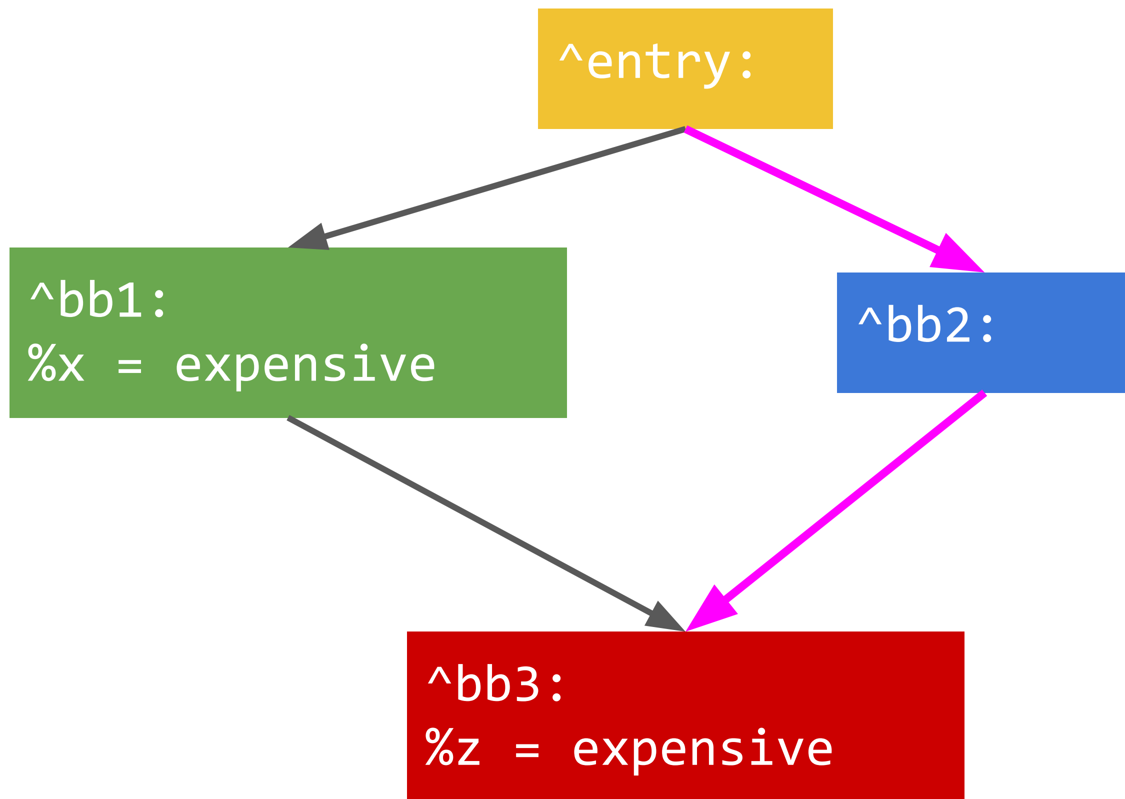
%x **unavailable**
in path
`entry-bb2-bb3`

Dominance in LLVM: CSE: Non Example



`%x` **unavailable** in path `entry-bb2-bb3` => CSE illegal

Dominance in LLVM: CSE: Non Example



^bb1 doesn't dominate ^bb3
=> CSE illegal

Agenda

What is dominance & why is it important?

→ Key property of SSA (liveness & reachability). ✓

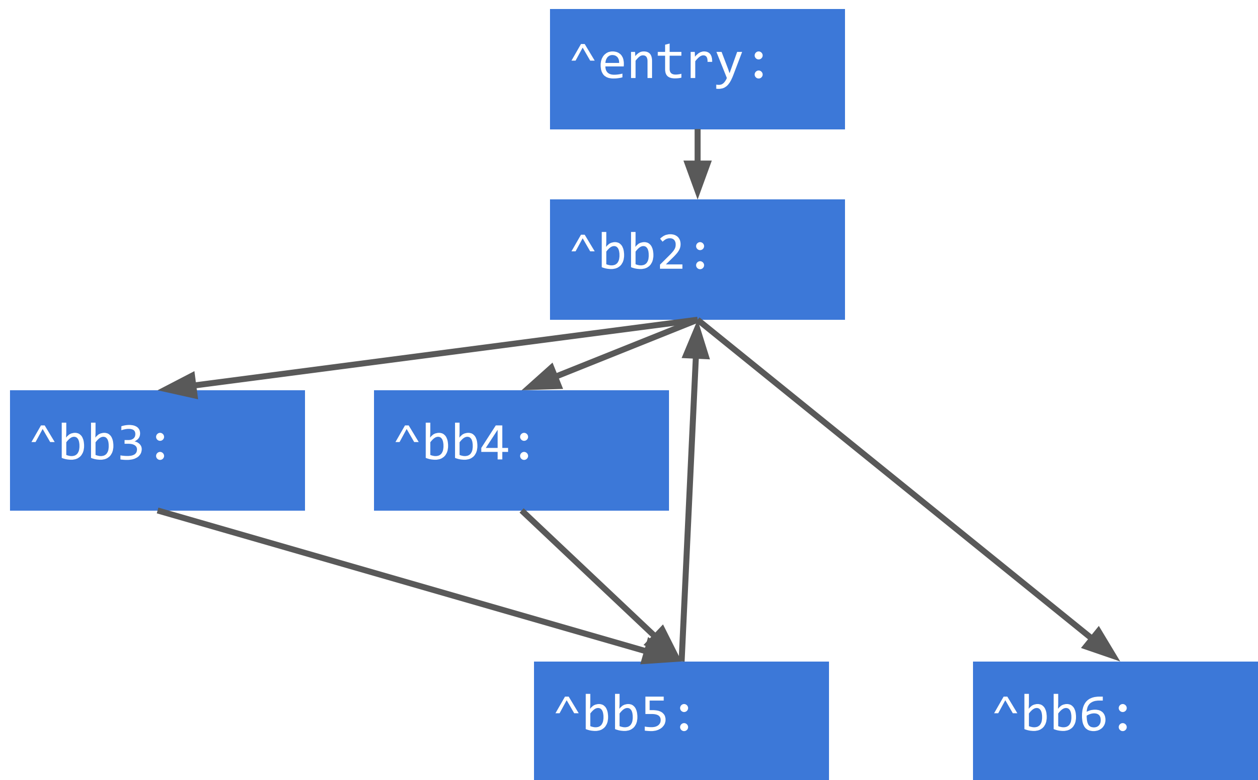
```
Instruction does not dominate all uses!
  %x = add i32 %6, 10
  store i32 %x, i32* %3, align 4
```

→ **Dominance**, Dominator Tree in LLVM. ⬅

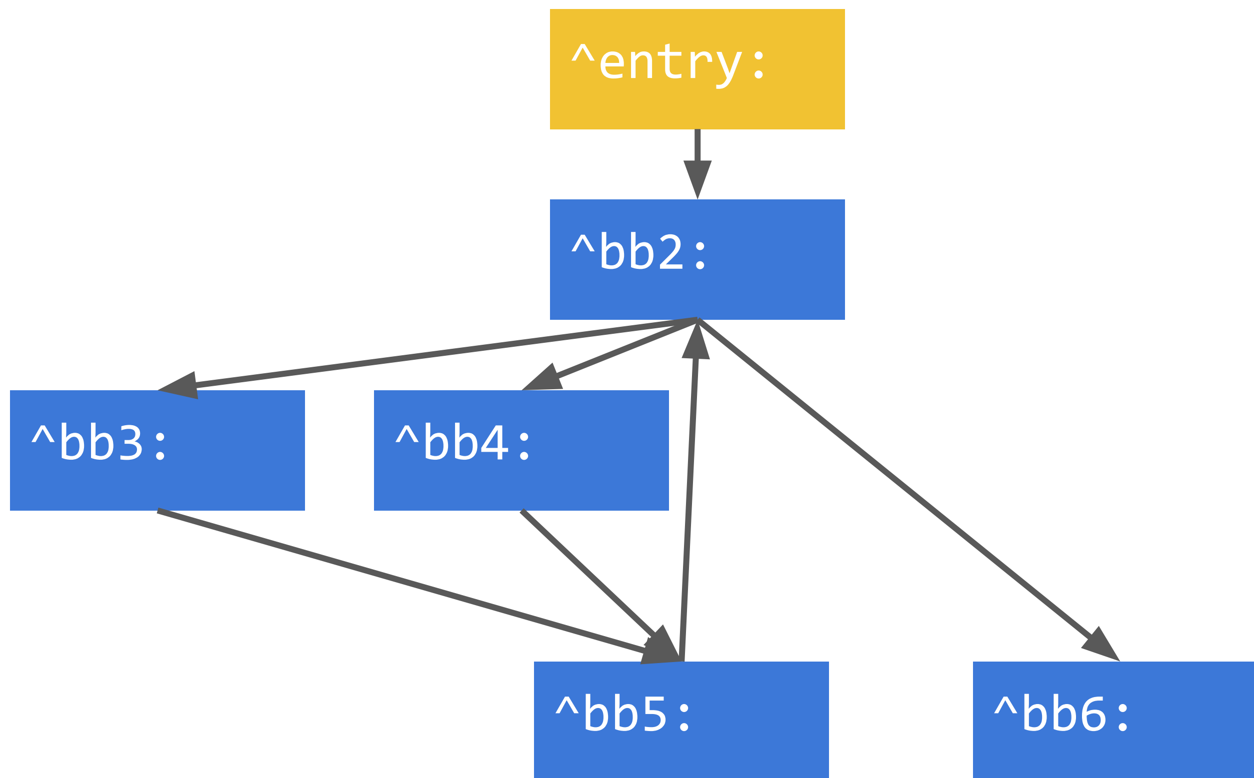
→ Regions, Dominators in MLIR, its problems.

→ Potential Solutions.

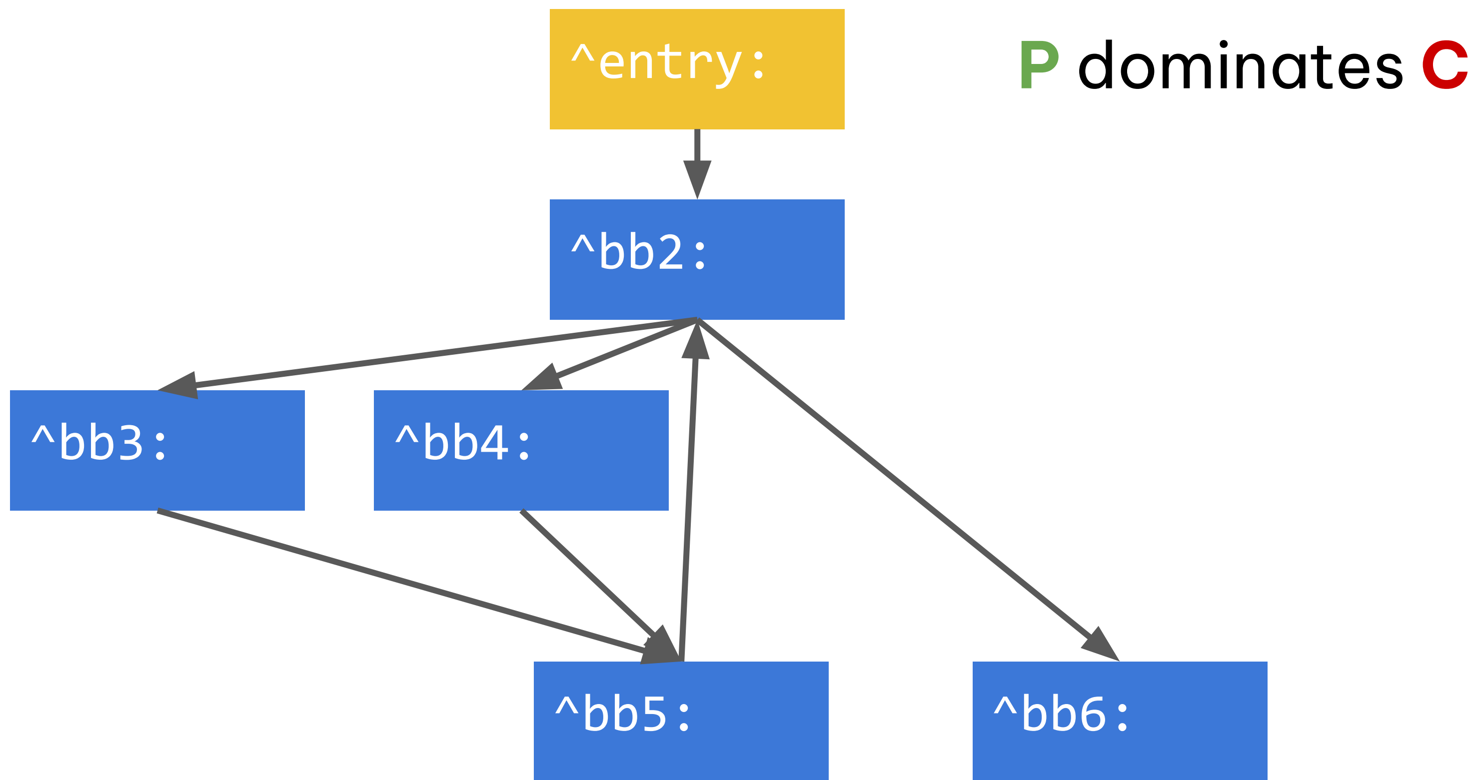
Dominance in LLVM



Dominance in LLVM

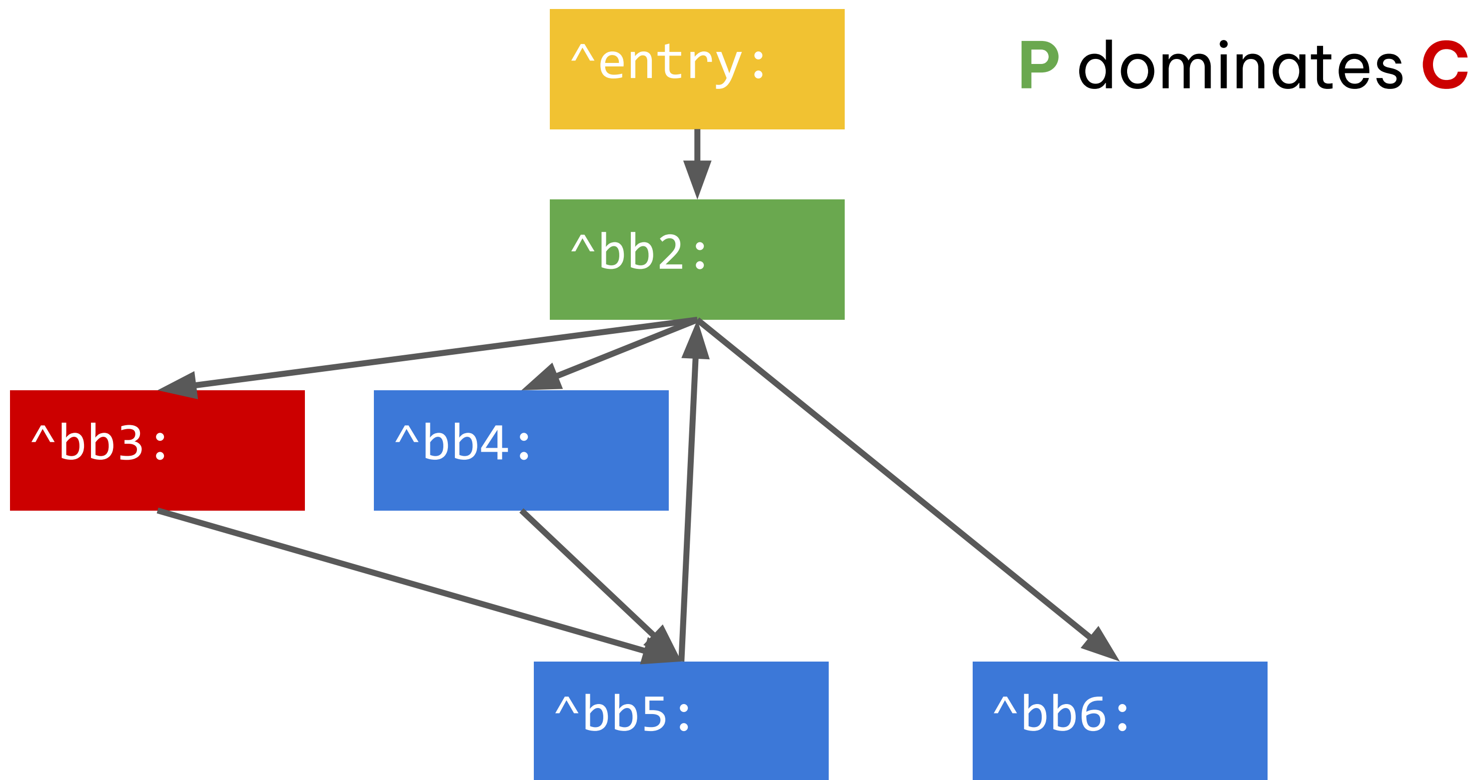


Dominance in LLVM



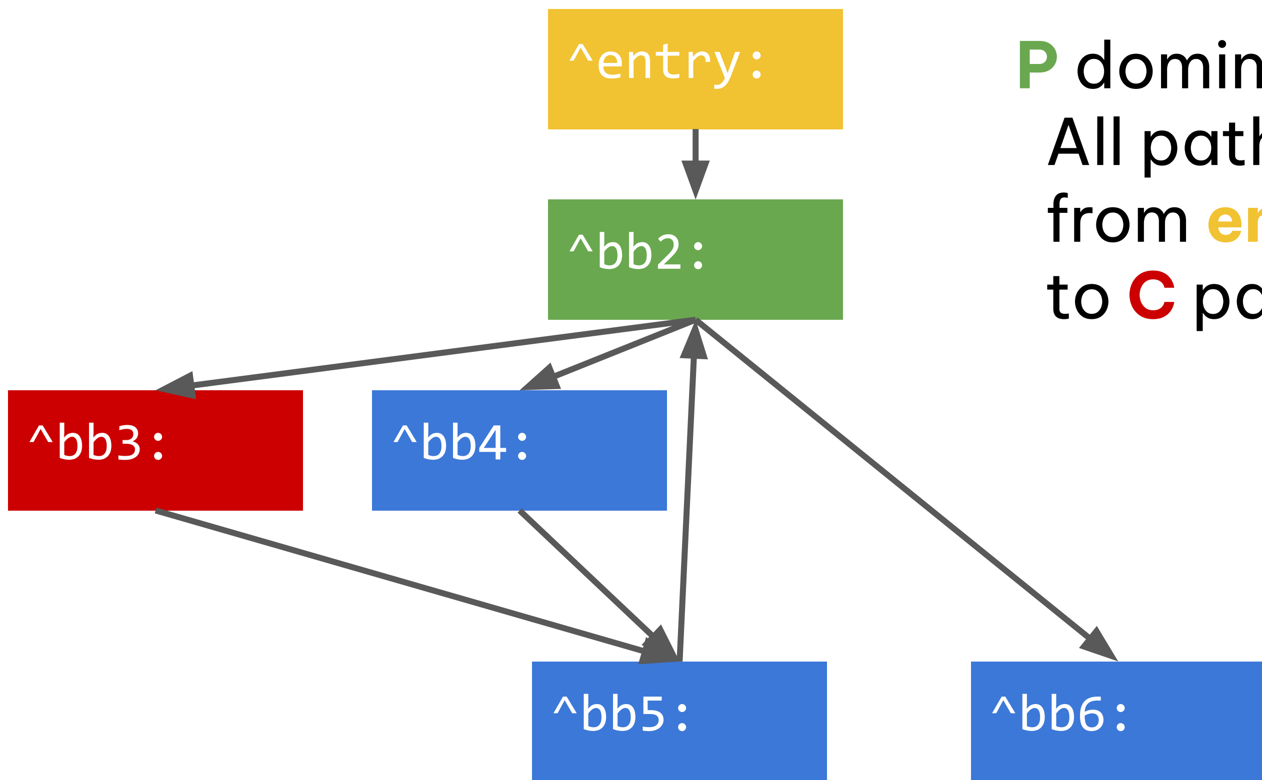
P dominates C

Dominance in LLVM



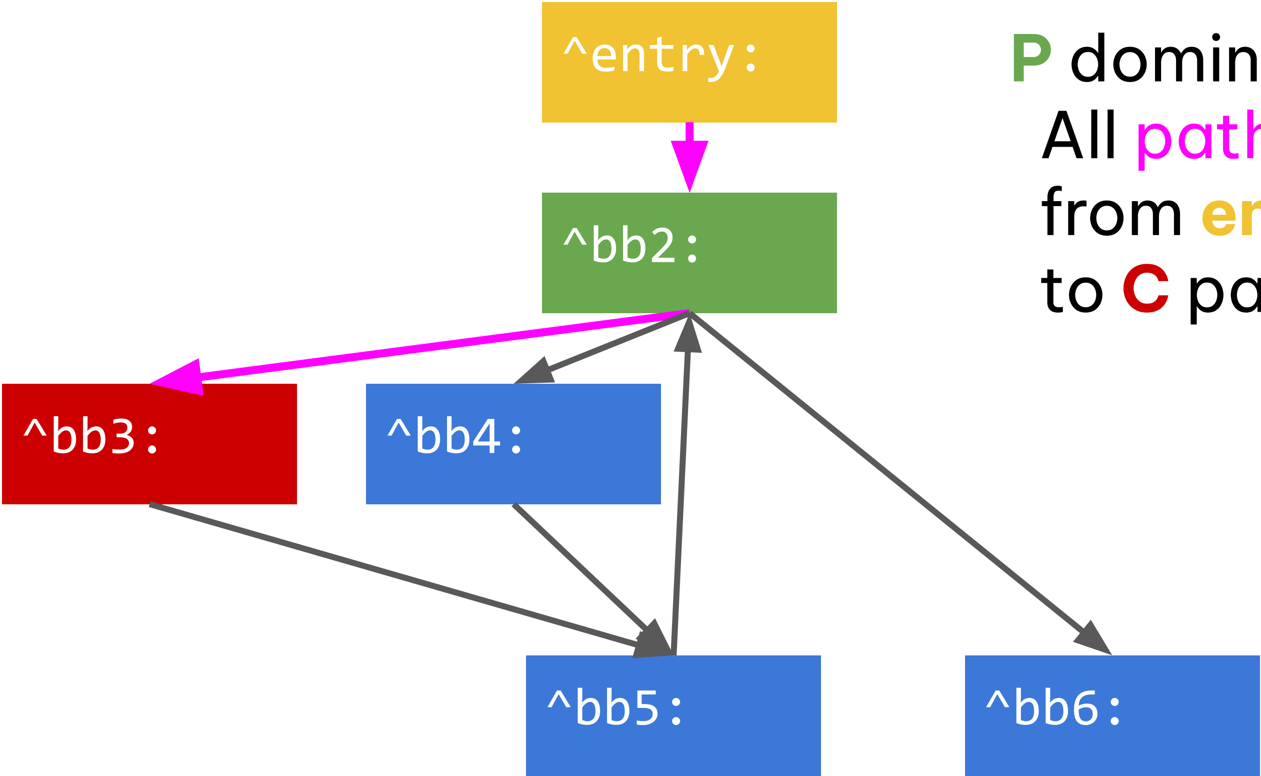
P dominates **C**

Dominance in LLVM



P dominates **C**
All paths
from **entry**
to **C** pass through **P**

Dominance in LLVM



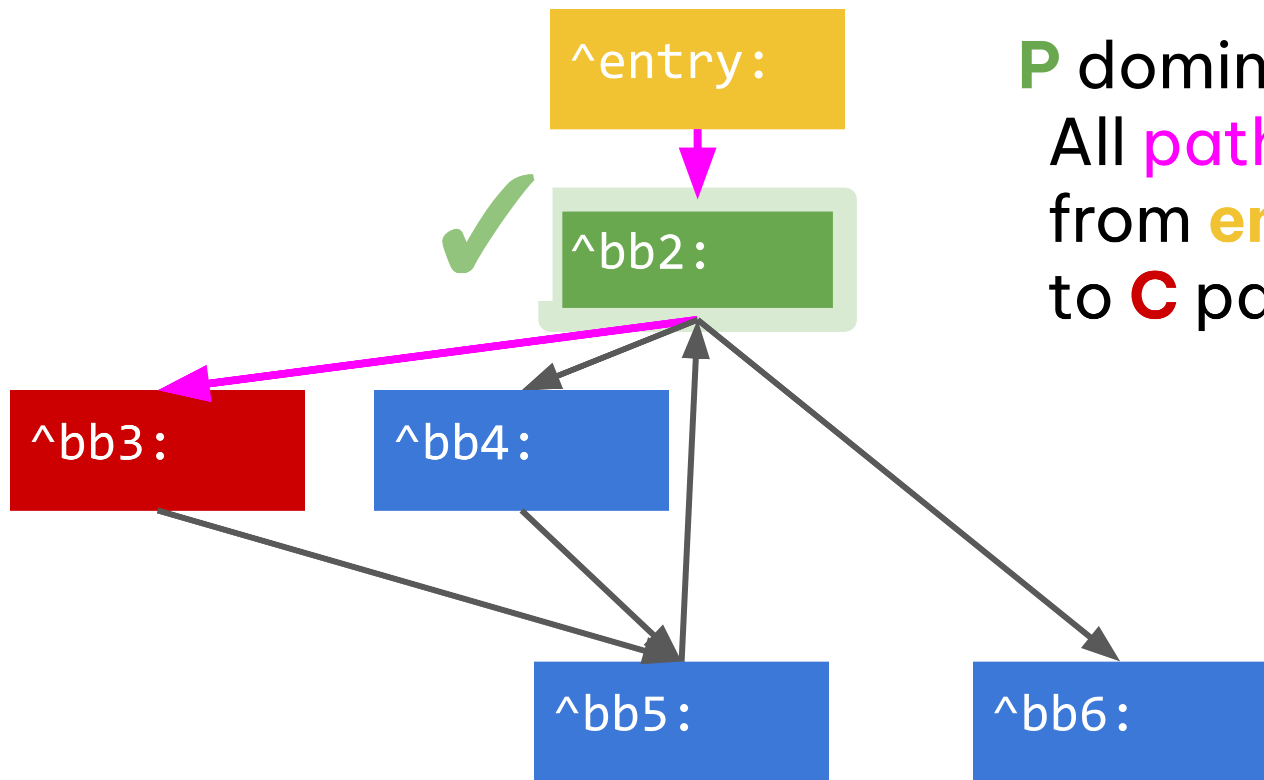
P dominates **C**

All paths

from **entry**

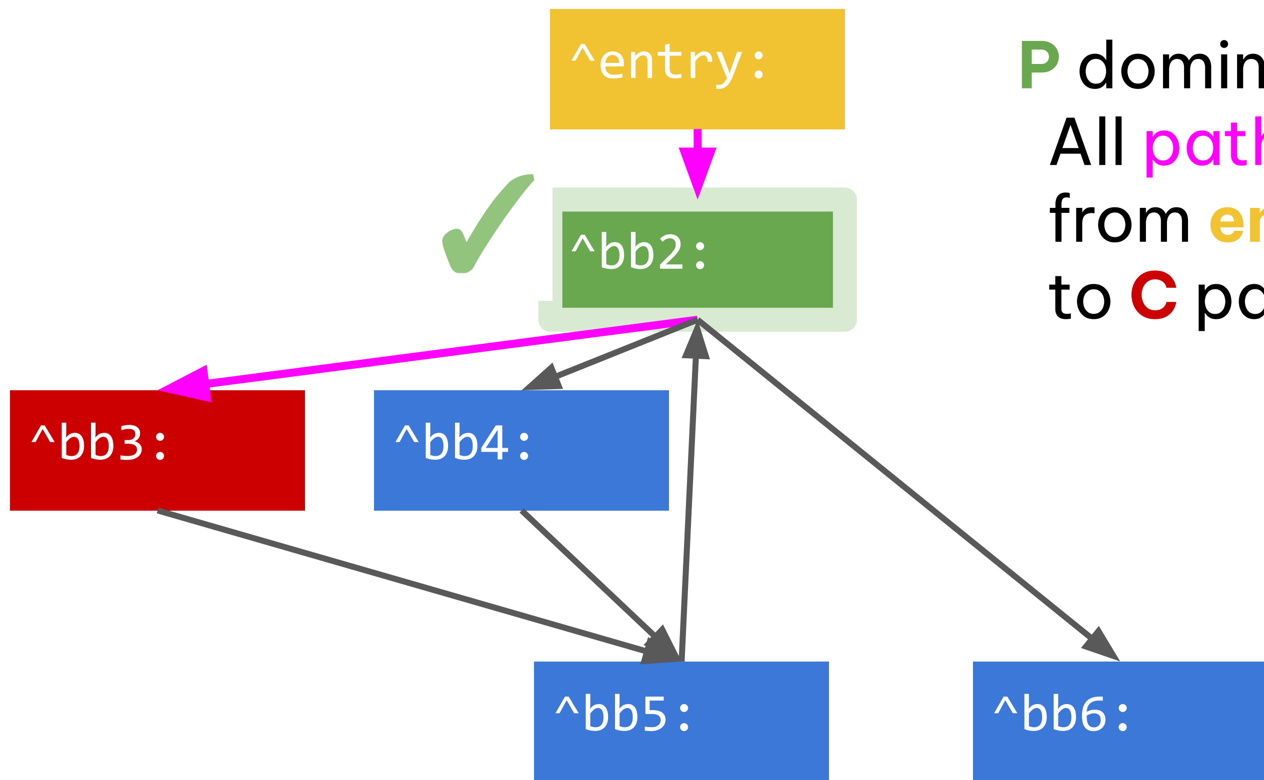
to **C** pass through **P**

Dominance in LLVM



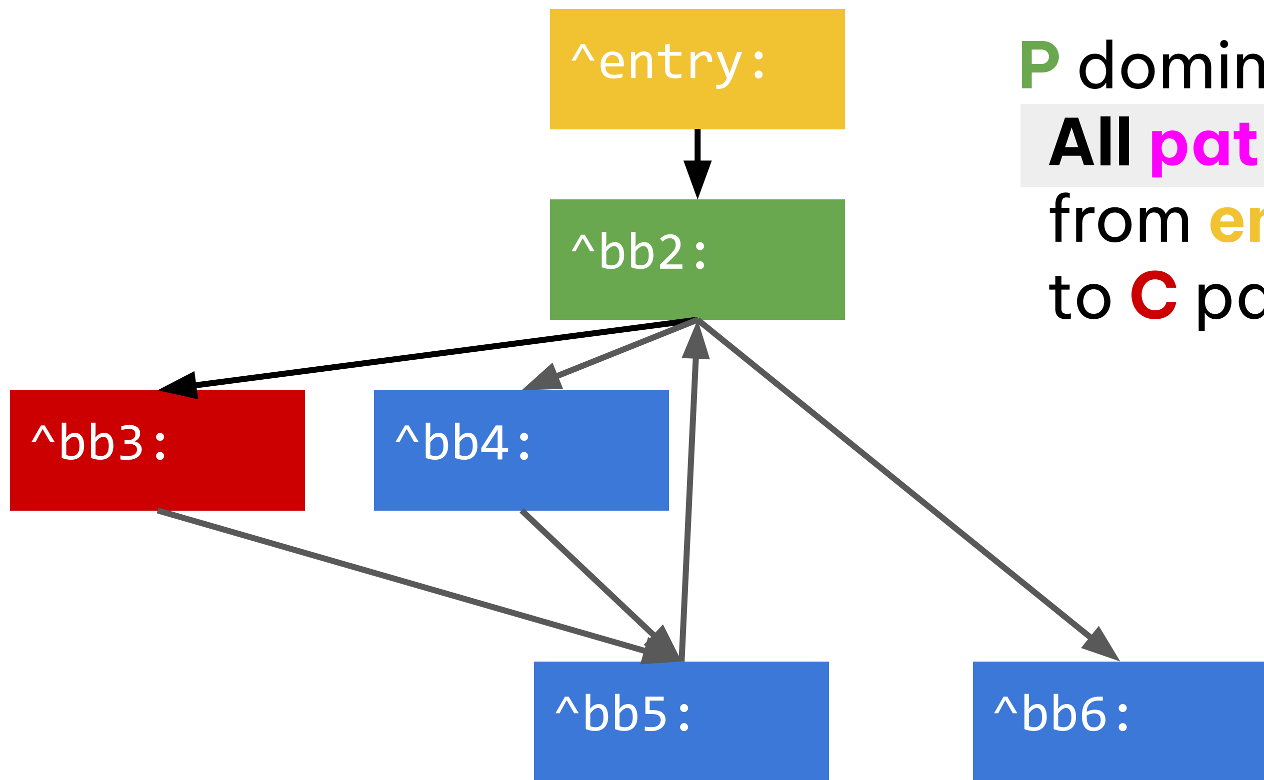
P dominates C
All paths
from entry
to C pass through P

Dominance in LLVM: Are we done?



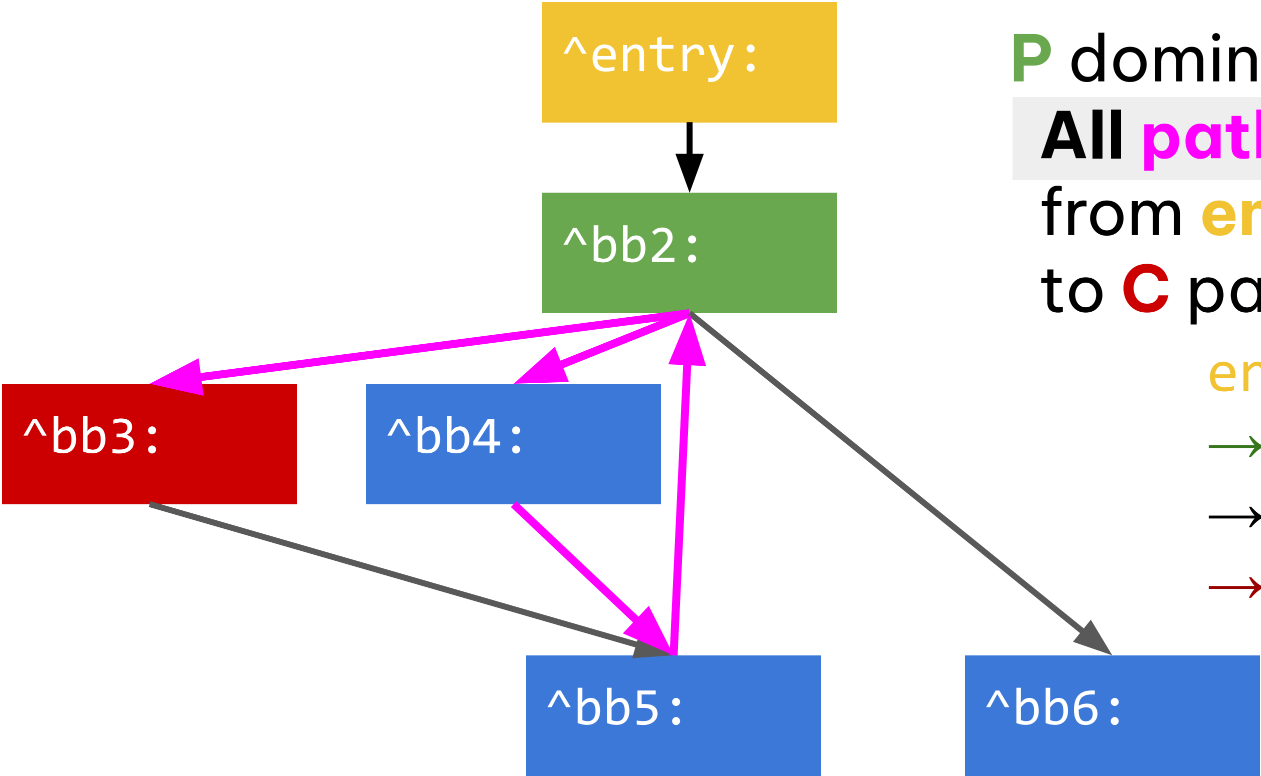
P dominates C
All paths
from entry
to C pass through P

Dominance in LLVM: No we're not!



P dominates C
All paths
from **entry**
to C pass through P

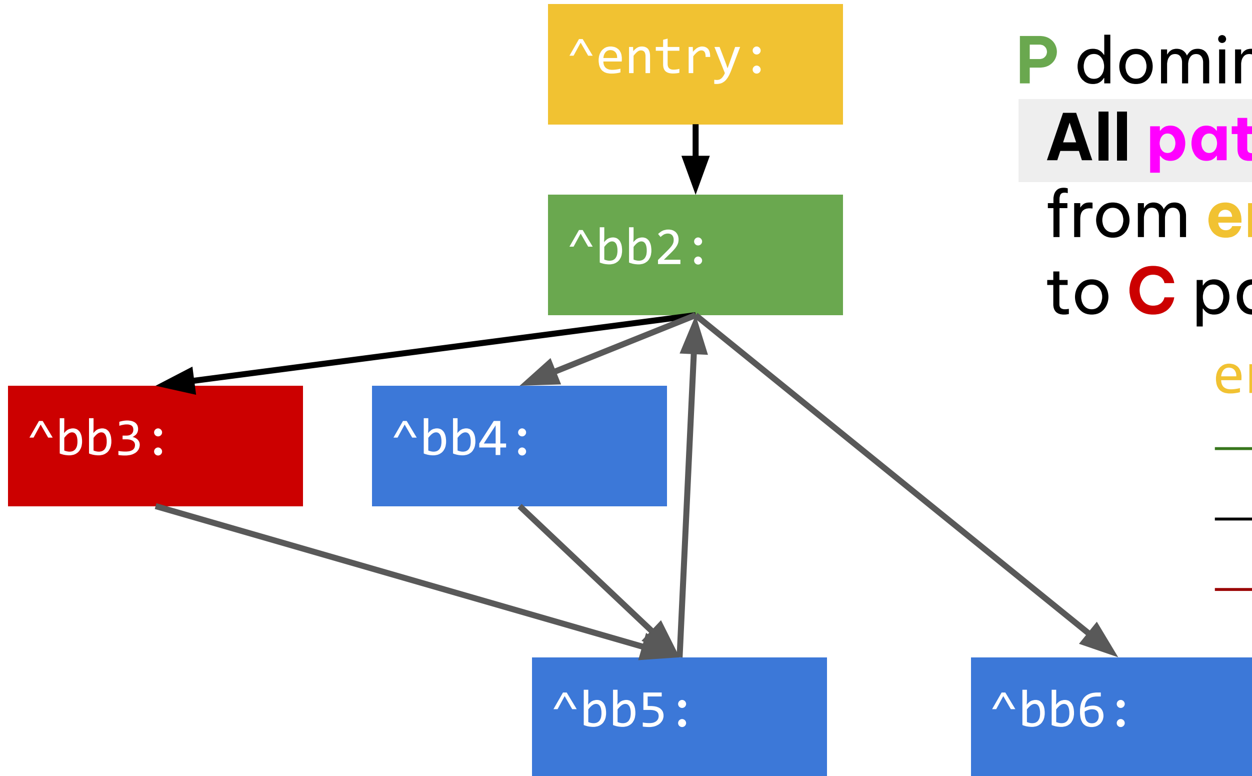
Dominance in LLVM: No we're not!



P dominates C
All paths
from **entry**
to **C** pass through P

entry
→bb2
→(bb4-bb5-bb2)*
→bb3

Dominance in LLVM: Now we are!



P dominates **C**

All paths ✓

from **entry**

to **C** pass through **P**

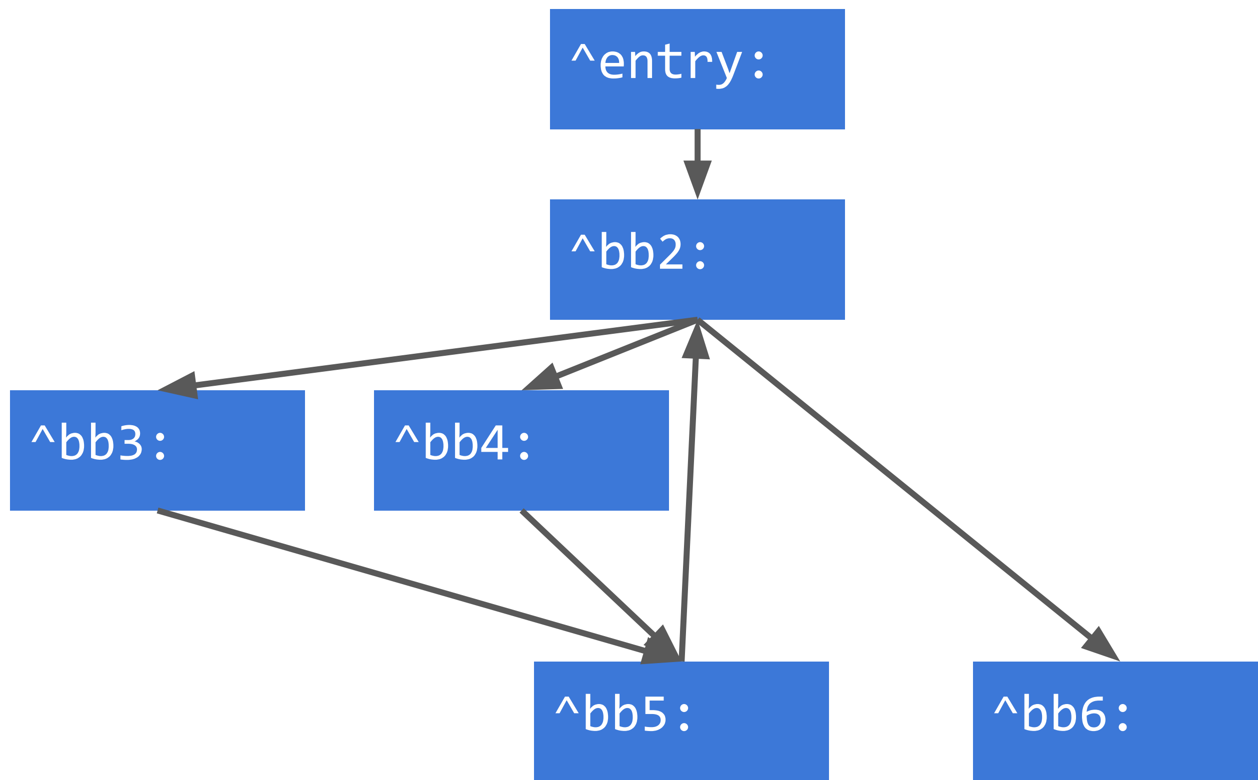
entry

→**bb2**

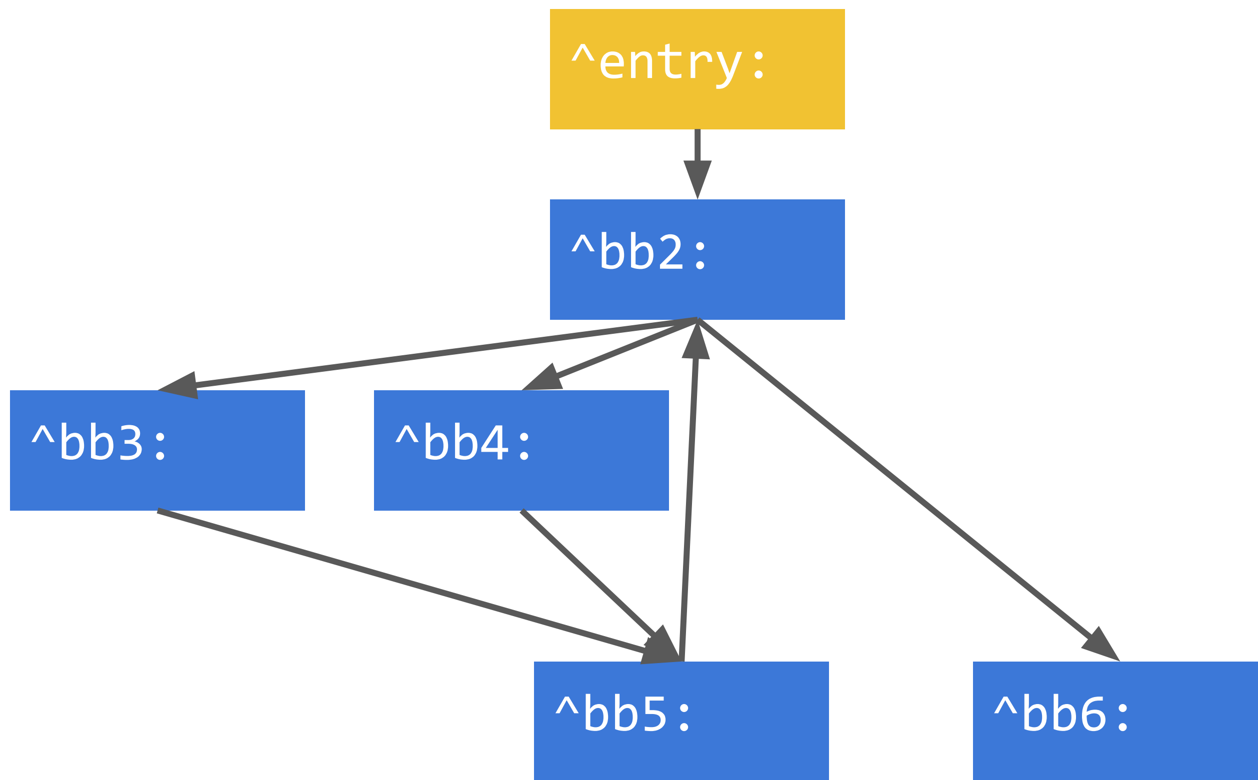
→(bb4-bb5-**bb2**)*

→bb3

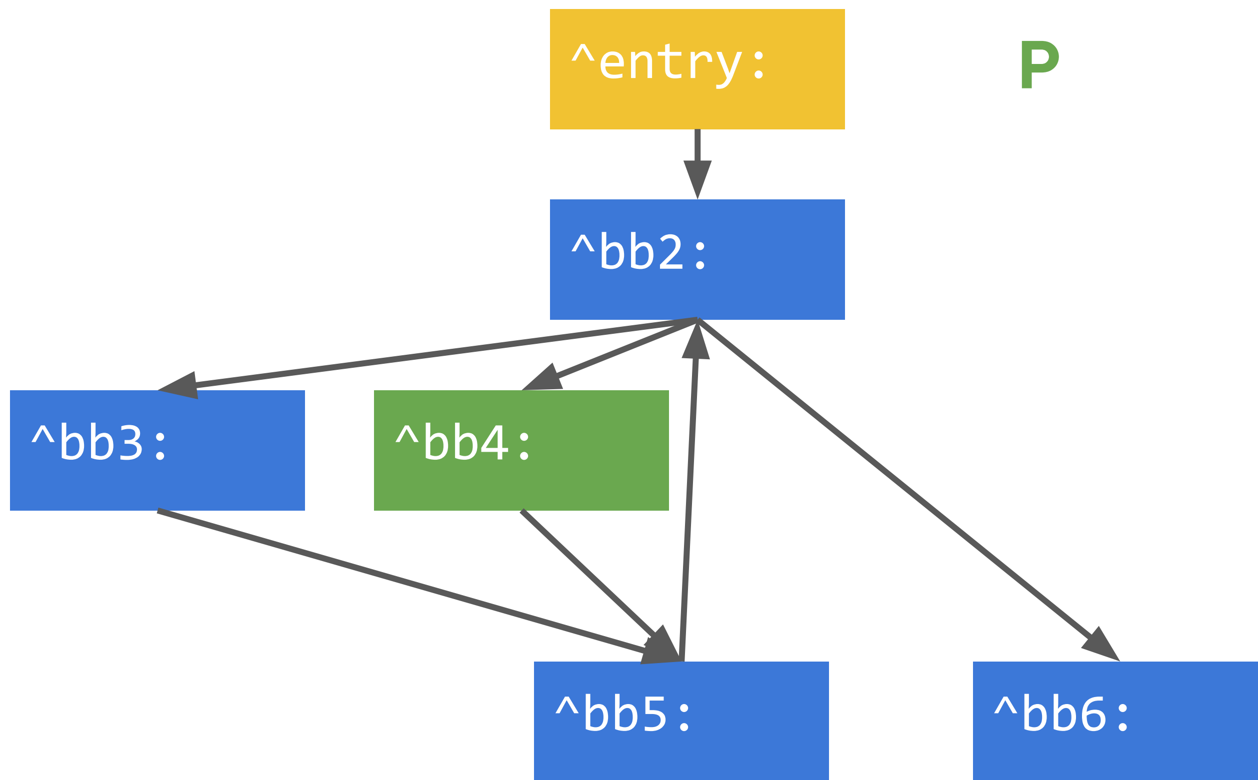
Dominance in LLVM: A Non Example



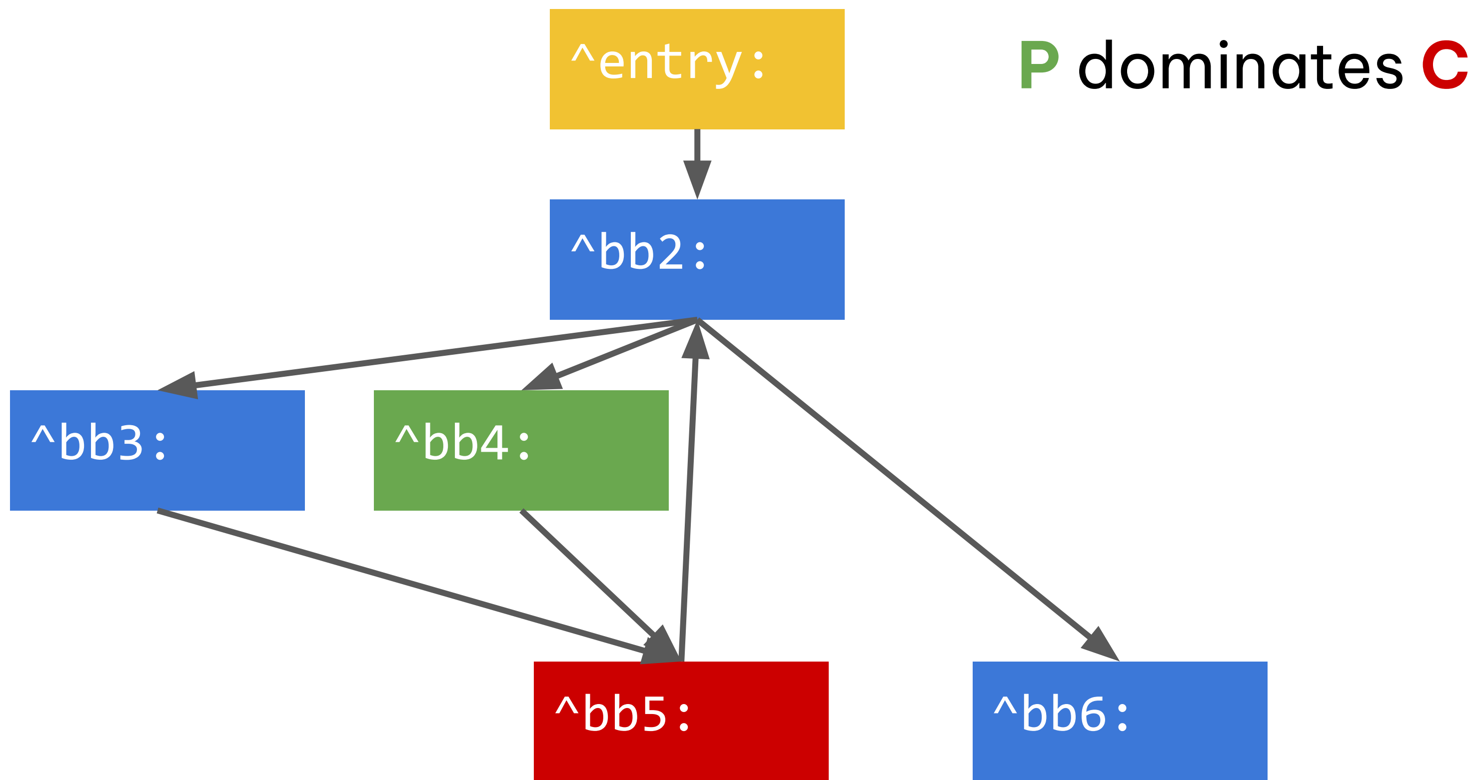
Dominance in LLVM: A Non Example



Dominance in LLVM: A Non Example

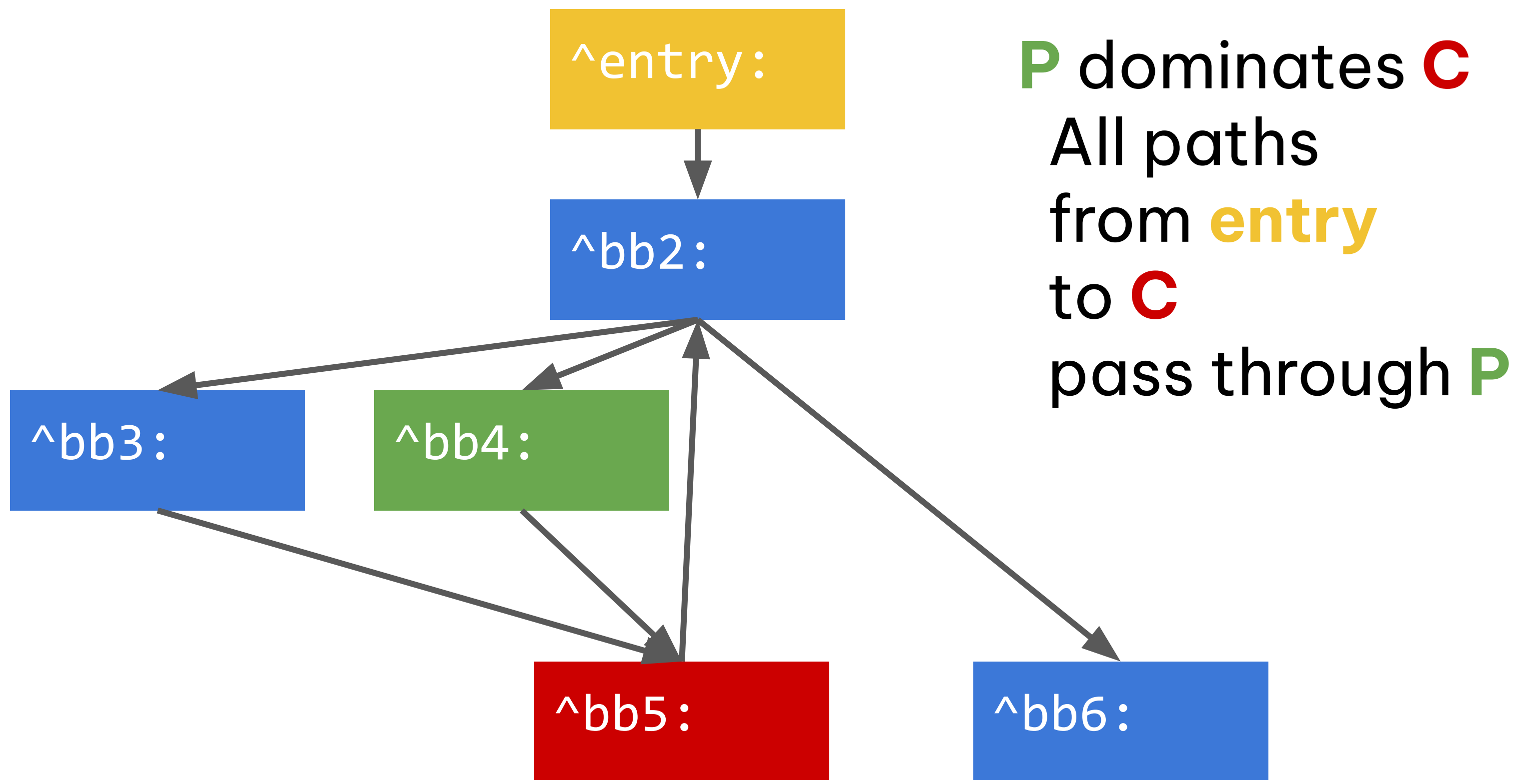


Dominance in LLVM: A Non Example



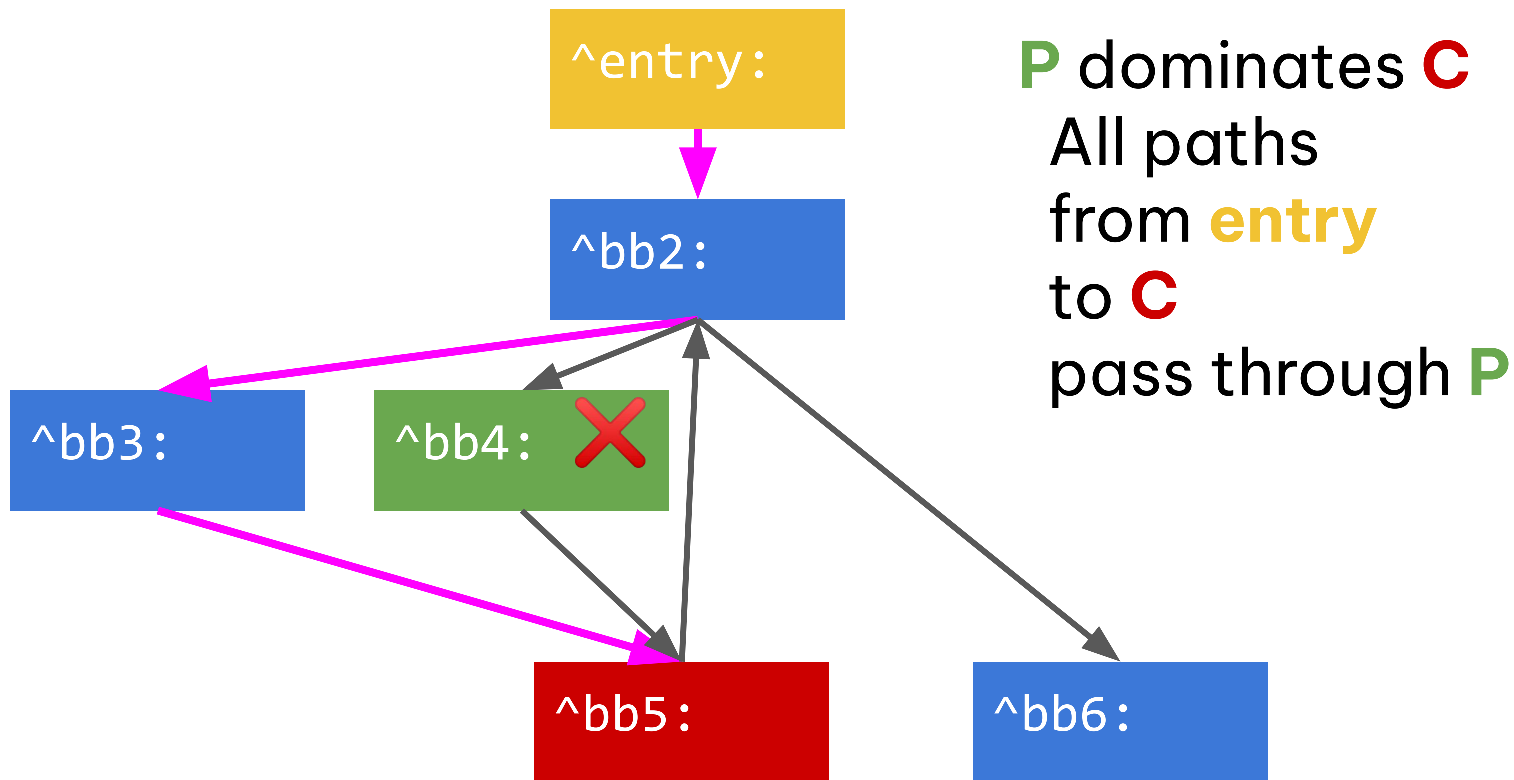
P dominates C

Dominance in LLVM: A Non Example



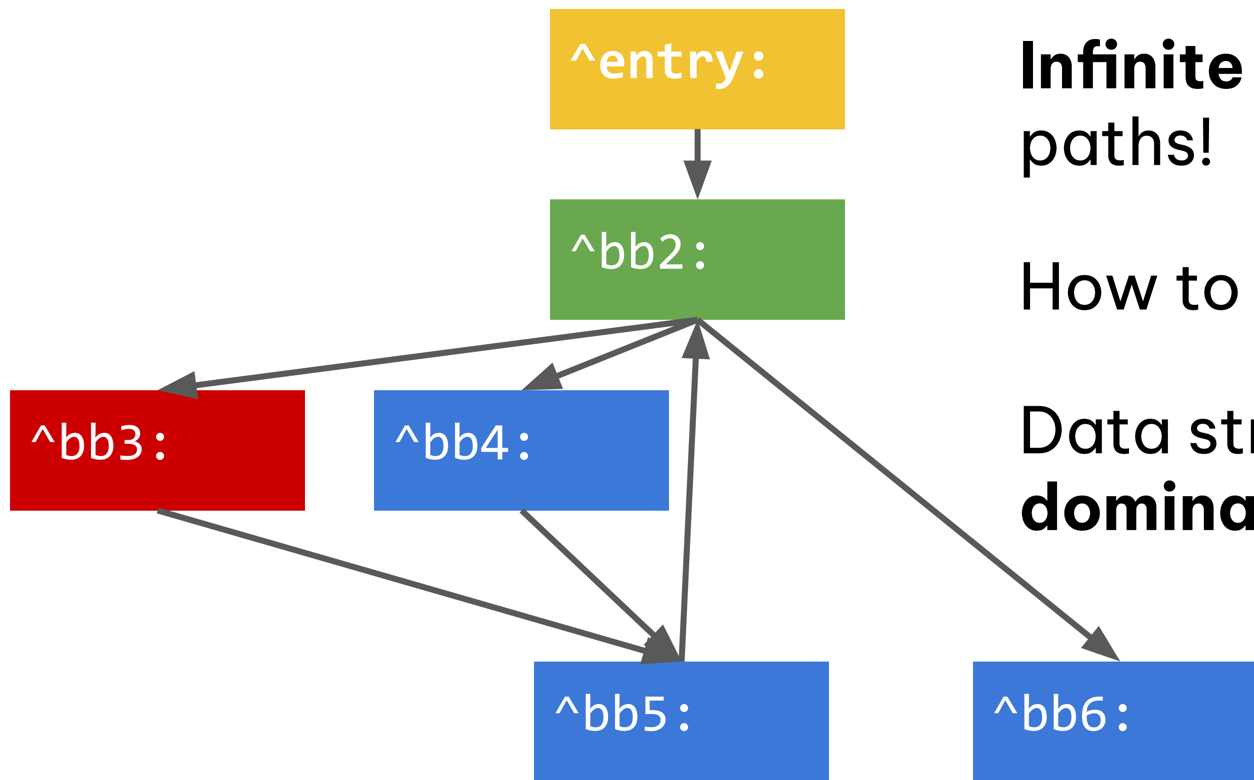
P dominates C
All paths
from entry
to C
pass through P

Dominance in LLVM: A Non Example



P dominates C
All paths
from entry
to C
pass through P

Dominance: How to compute?



Infinite number of paths!

How to compute?

Data structure: **dominator tree**

Agenda

What is dominance & why is it important?

→ Key property of SSA (liveness & reachability). ✓

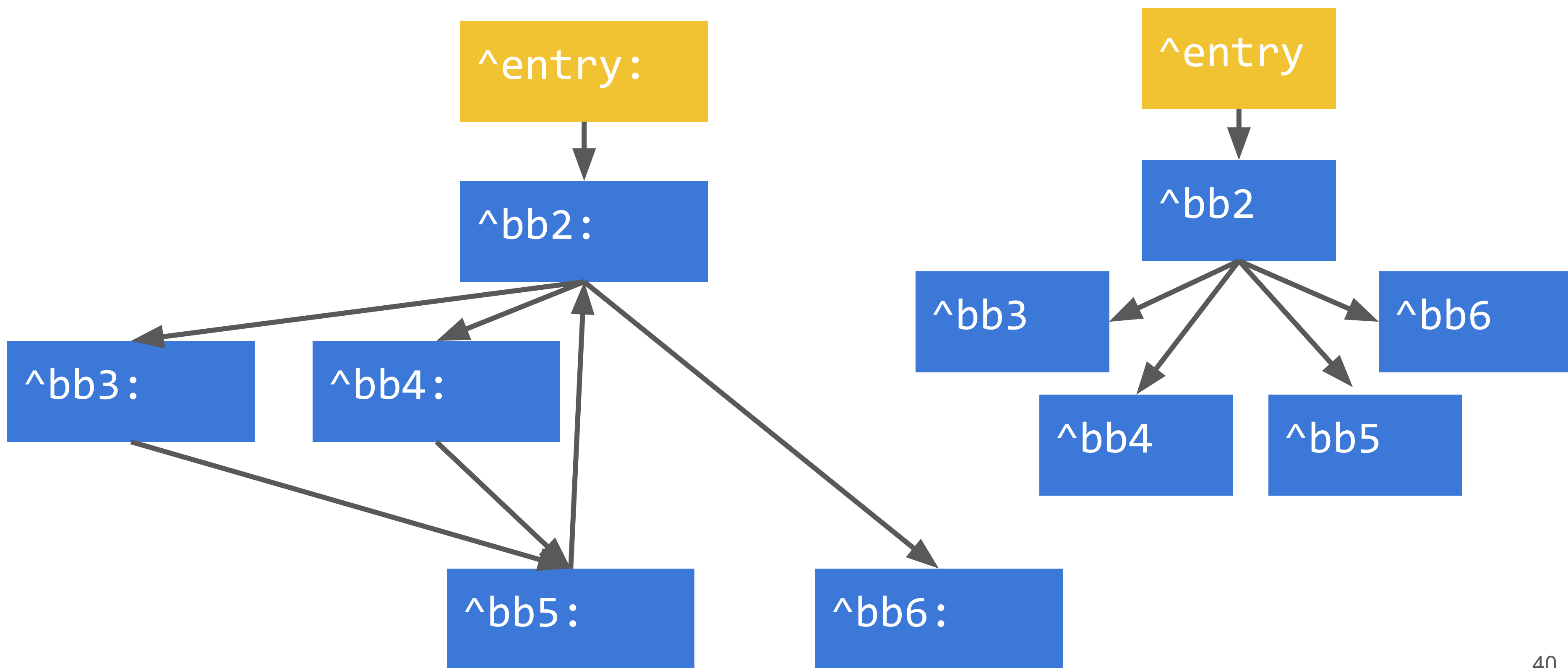
```
Instruction does not dominate all uses!
  %x = add i32 %6, 10
  store i32 %x, i32* %3, align 4
```

→ Dominance, **Dominator Tree** in LLVM. ⬅

→ Regions, Dominators in MLIR, its problems.

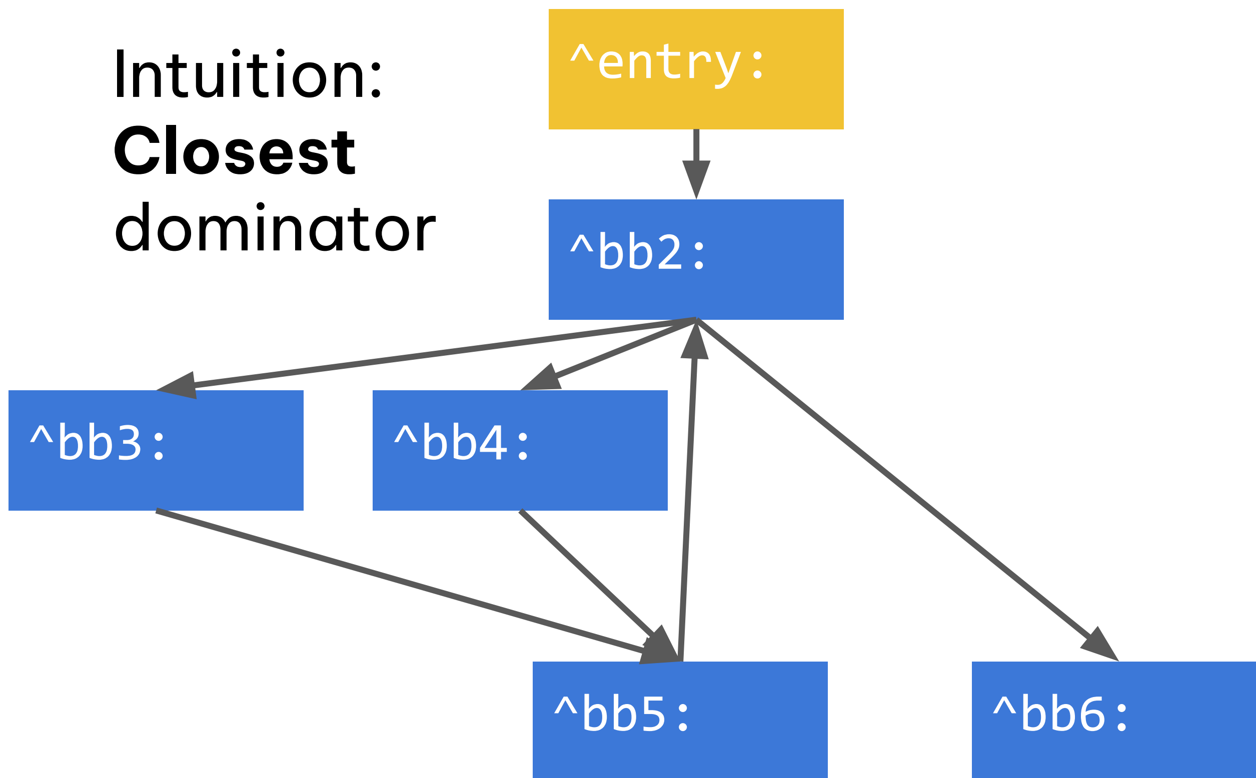
→ Potential Solutions.

Dominance: Dominator **Tree**



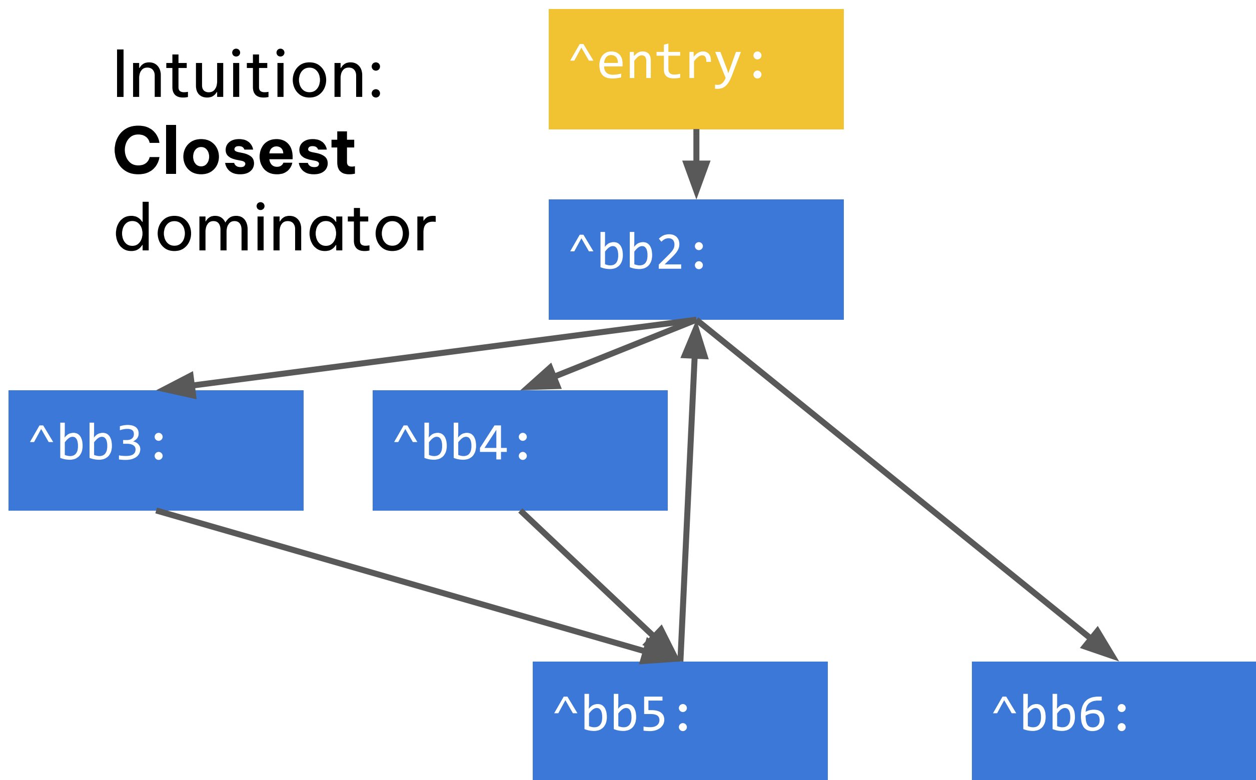
Dominance: Dominator **Tree**

Intuition:
Closest
dominator



Dominance: Dominator **Tree**

Intuition:
Closest
dominator



^entry

Dominance: Dominator **Tree**

Intuition:
Closest
dominator

^entry:

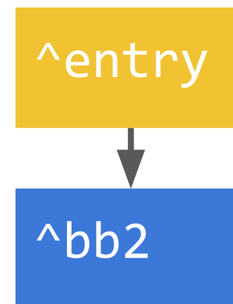
^bb2:

^bb3:

^bb4:

^bb5:

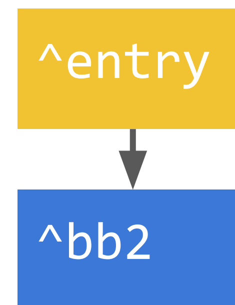
^bb6:



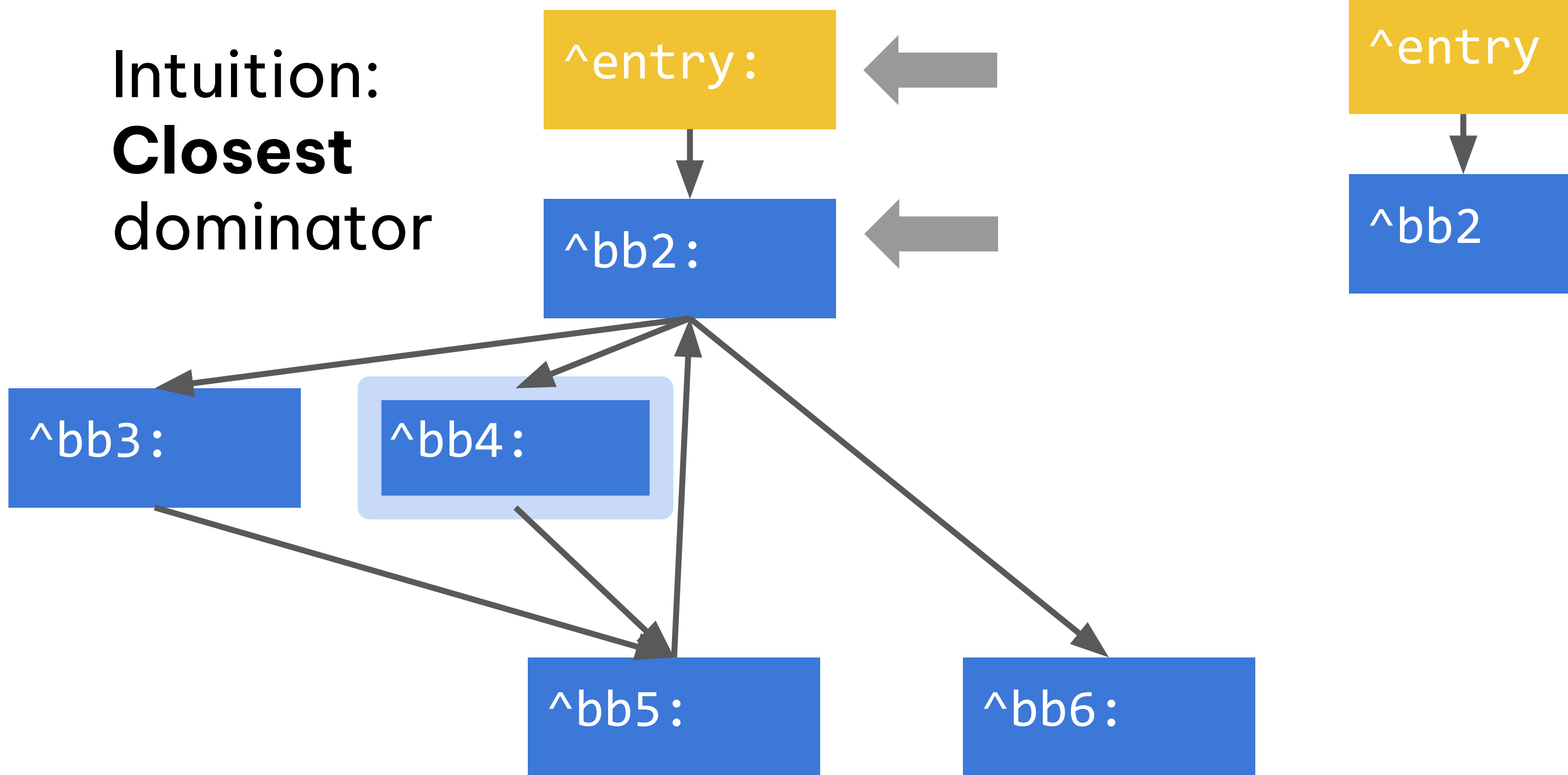
Dominance: Dominator **Tree**

Intuition:
Closest
dominator

```
^entry:
^bb2:
^bb3:
^bb4:
^bb5:
^bb6:
```



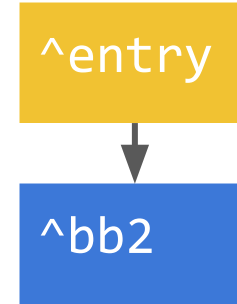
Dominance: Dominator **Tree**



Dominance: Dominator **Tree**

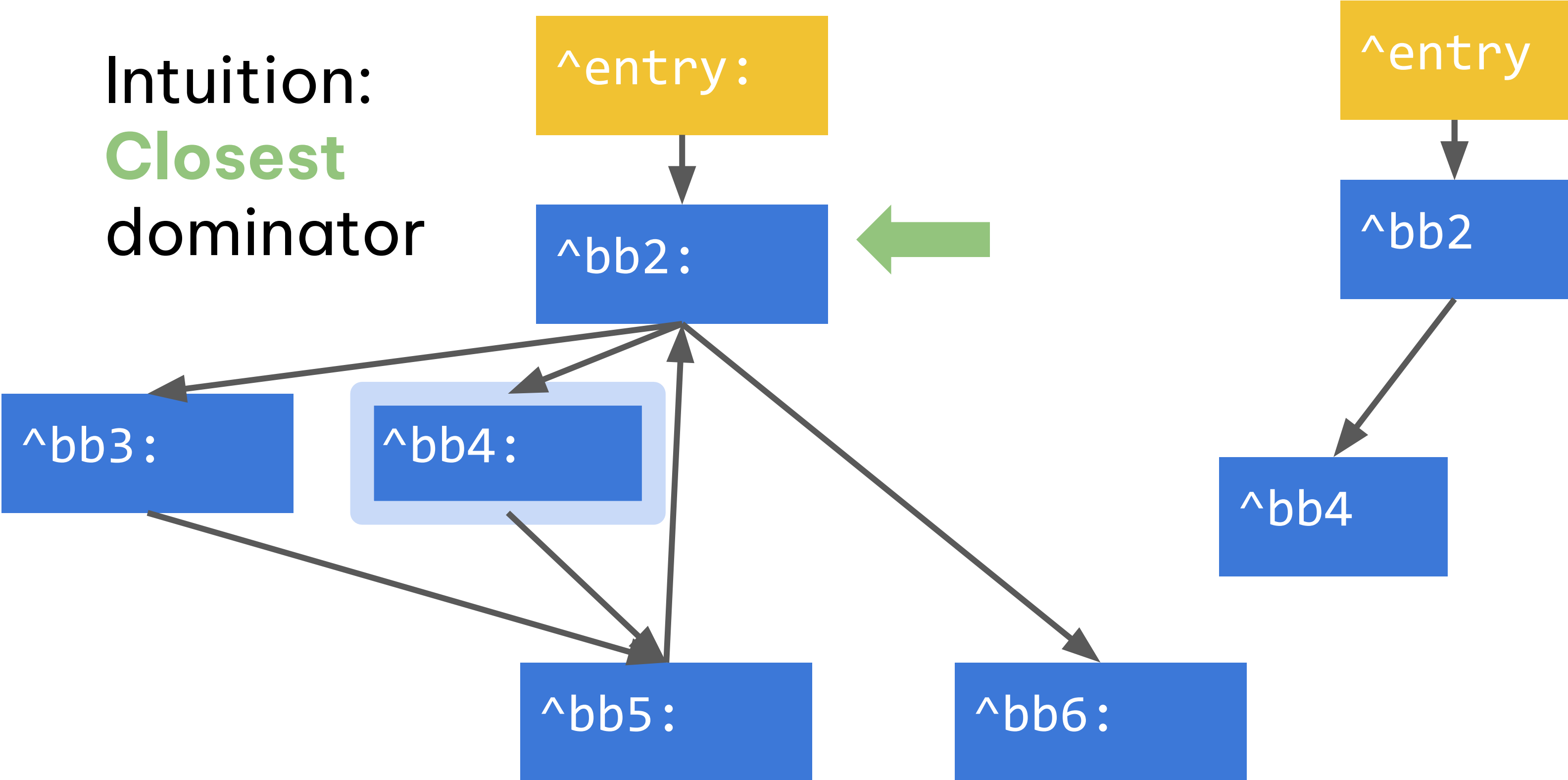
Intuition:
Closest
dominator

```
^entry:
^bb2:
^bb3:
^bb4:
^bb5:
^bb6:
```



Dominance: Dominator **Tree**

Intuition:
Closest
dominator



Dominance: Dominator **Tree**

Intuition:
Closest
dominator

^entry:

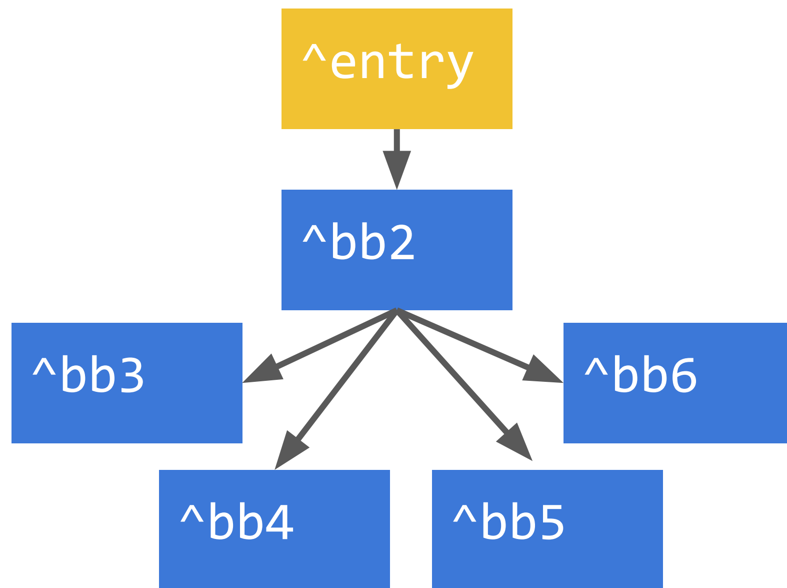
^bb2:

^bb3:

^bb4:

^bb5:

^bb6:



Dominance: Dominator **Tree**

Intuition: **Closest** dominator

^entry:

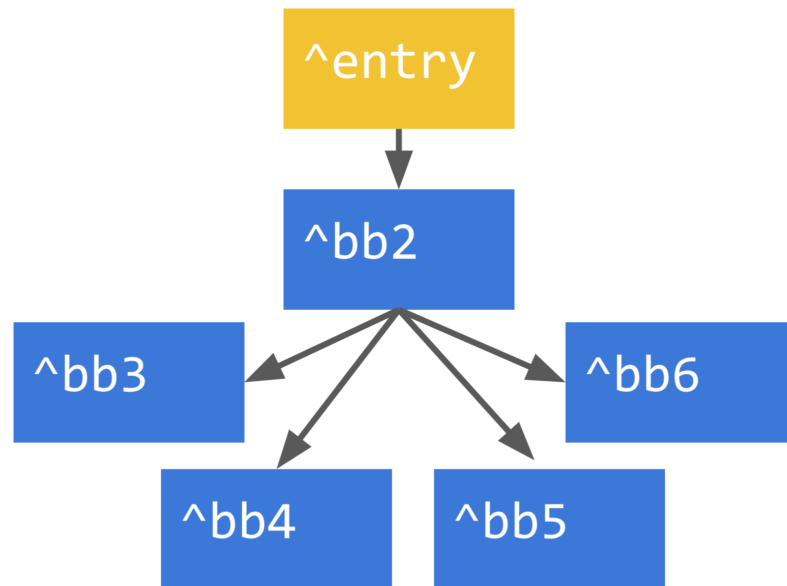
^bb2:

^bb3:

^bb4:

^bb5:

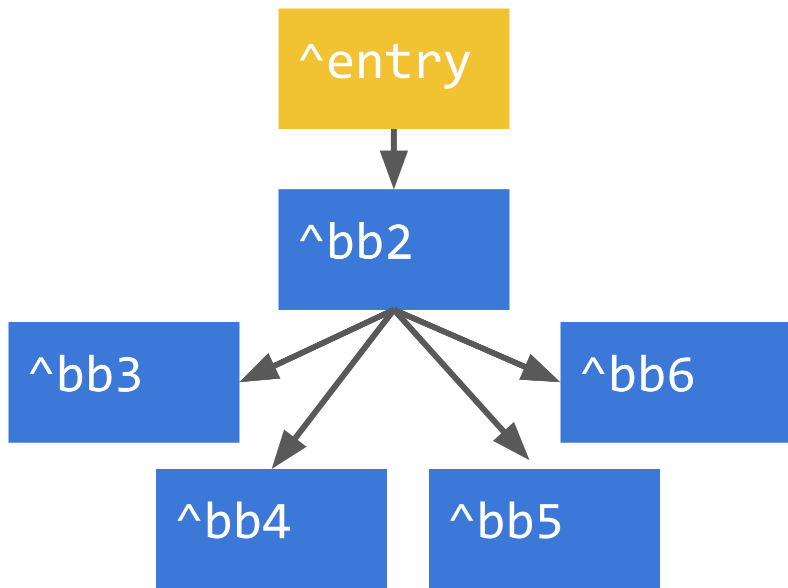
^bb6:



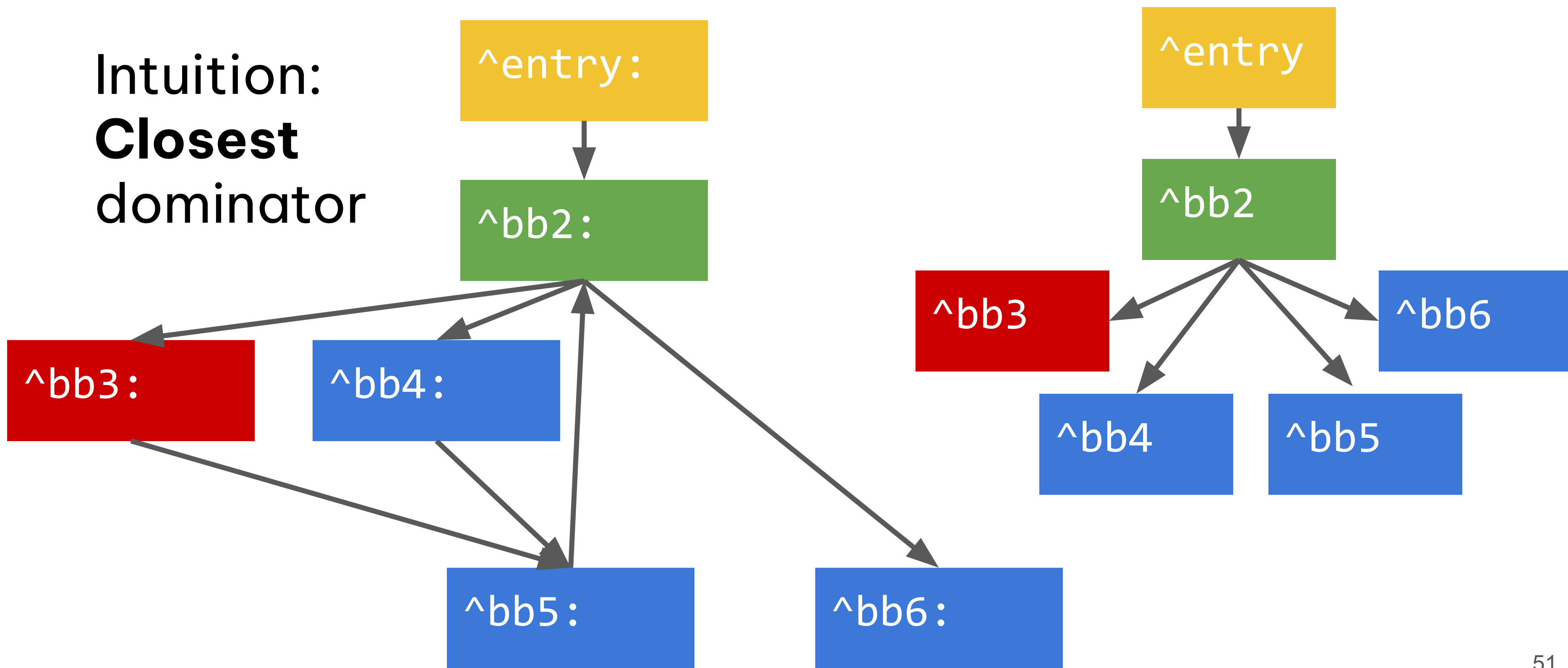
Dominance: Dominator **Tree**: Example

Intuition:
Closest
dominator

```
^entry:
^bb2:
^bb3:
^bb4:
^bb5:
^bb6:
```



Dominance: Dominator **Tree**: Example



Dominance: Dominator **Tree**: Non Example

Intuition: **Closest** dominator

^entry:

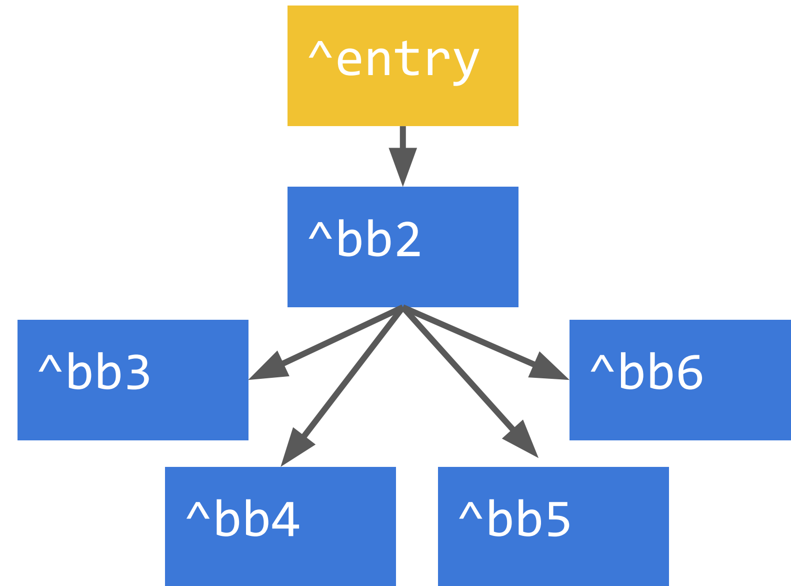
^bb2:

^bb3:

^bb4:

^bb5:

^bb6:



Dominance: Dominator **Tree**: Non Example

Intuition: **Closest** dominator

^entry:

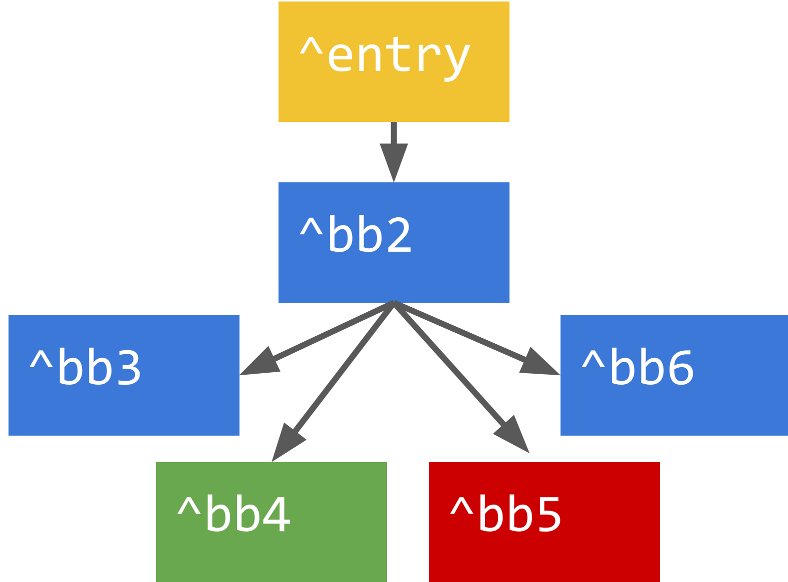
^bb2:

^bb3:

^bb4:

^bb5:

^bb6:



Agenda

What is dominance & why is it important?

→ Key property of SSA (liveness & reachability). ✓

```
Instruction does not dominate all uses!
  %x = add i32 %6, 10
  store i32 %x, i32* %3, align 4
```

→ Dominance, Dominator Tree in LLVM. ✓

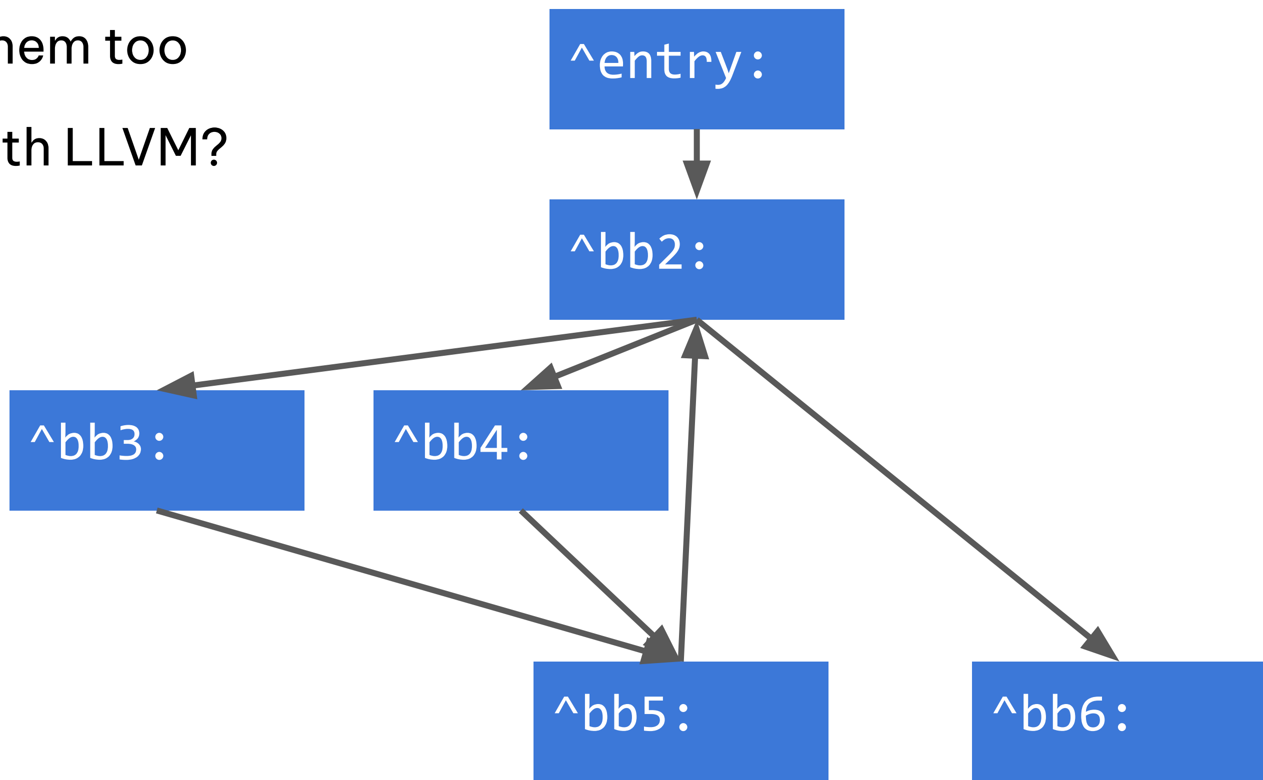
→ **Regions**, Dominators in MLIR, its problems.

→ Potential Solutions.

Dominance in MLIR

CFGs… MLIR has them too

Any differences with LLVM?



Dominance in MLIR

Not really, besides indirect gotos

All the LLVM infra can be re-used!

```
template class llvm::DominatorTreeBase<mlir::Block, /*IsPostDom=*/false>;
template class llvm::DominatorTreeBase<mlir::Block, /*IsPostDom=*/true>;
template class llvm::DomTreeNodeBase<mlir::Block>;
```

Dominance in MLIR

Special mention: **block arguments**

Semantically equivalent to phi nodes, “just” different representation (with respect to dominance)

```
56:
  %57 = phi i64 [ 0, %47 ], [ %62, %61 ]
  %58 = icmp slt i64 %57, %55
  br i1 %58, label %59, label %60
```

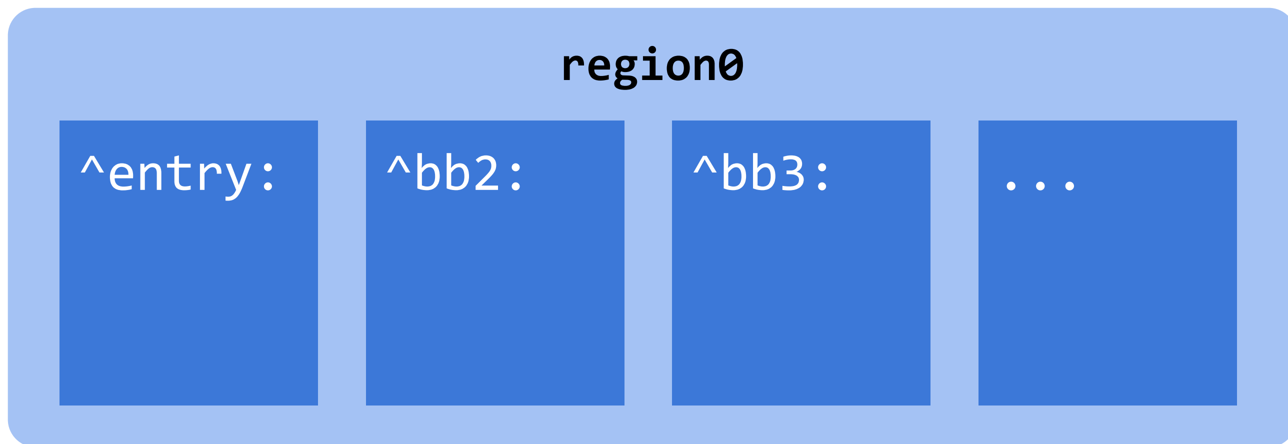
```
^bb56(%57: i64):

  %58 = llvm.icmp "slt", %57, %55 : i64
  llvm.br %58, ^bb59, ^bb60(%54)
```

New kid on the “block”: Regions!

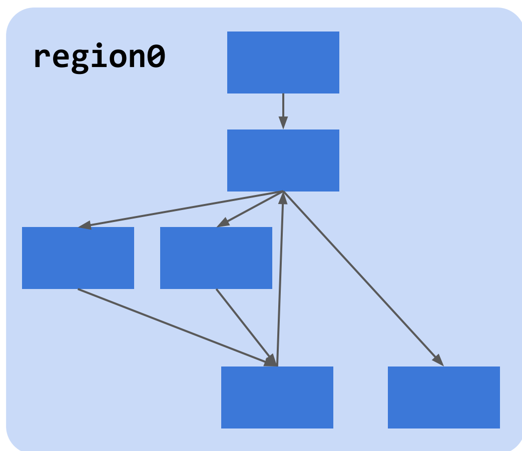
“An ordered list of MLIR blocks” (from the LangRef)

So regions have no semantics, right?

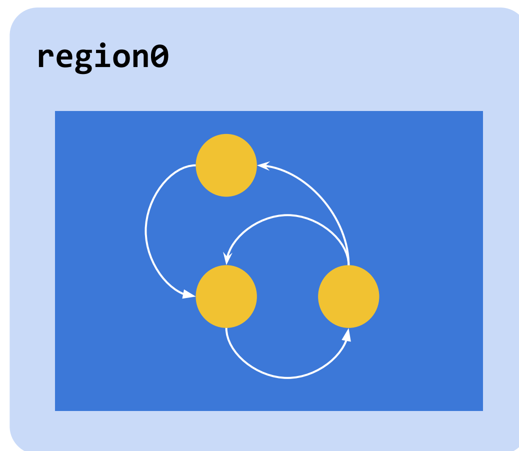


There are two region “kinds”

“SSACFG” regions



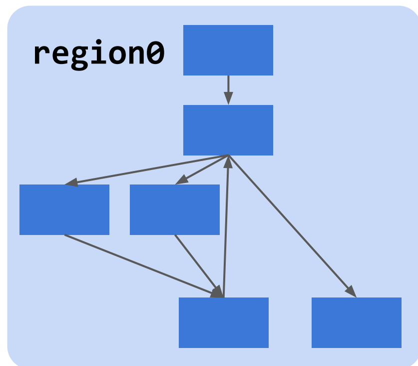
“Graph” regions



SSACFG Regions

“What-you-expect” regions

- Contain a CFG
- Operations obey SSA
- First block is **always** the entry block



Graph Regions

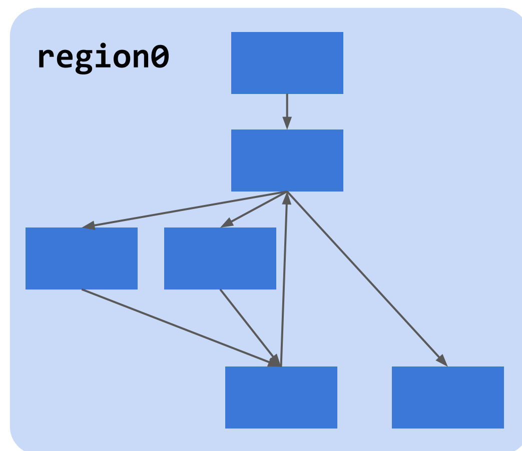
Using MLIR to represent graphs

- Single block*
- Operations can use results before they are defined

```
graph(%a: index) {
  %b = index.add %a, %c
  %c = index.add %b, %c
}
```

Semantics of regions

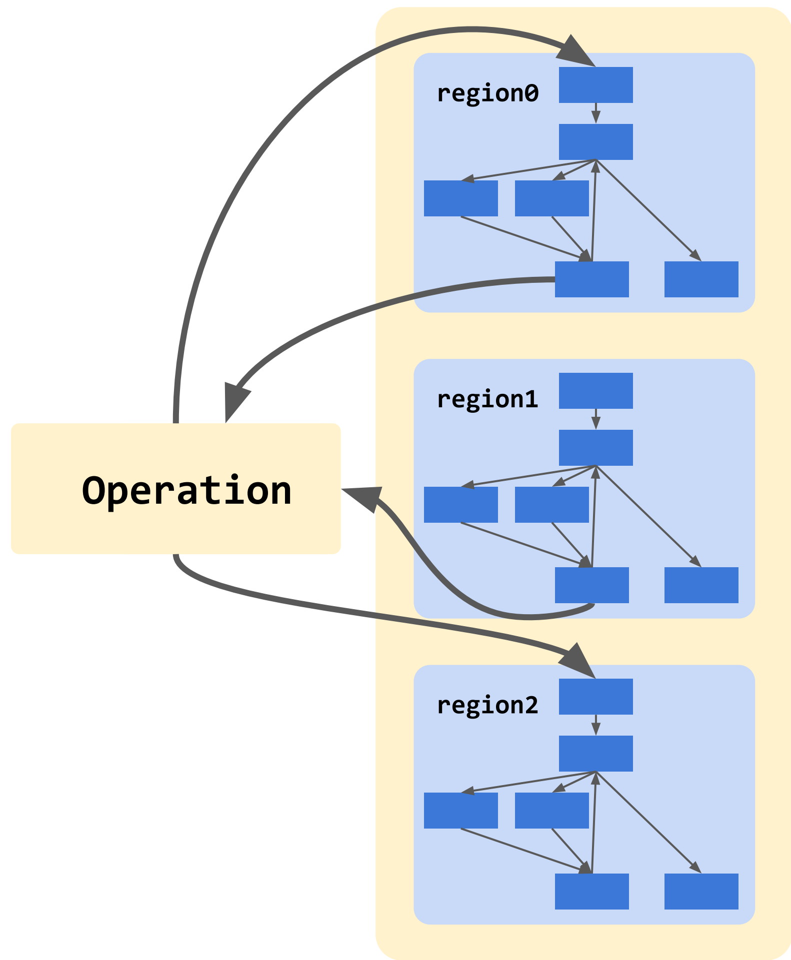
Semantics *within* SSACFG and graph regions are partially defined by MLIR



Semantics of regions

Semantics across regions completely defined by the *parent operation* 🤠

The semantics within a region is not imposed by the IR. Instead, the containing operation defines the semantics of the regions it contains.



Agenda

What is dominance & why is it important?

→ Key property of SSA (liveness & reachability). ✓

```
Instruction does not dominate all uses!
  %x = add i32 %6, 10
  store i32 %x, i32* %3, align 4
```

→ Dominance, Dominator Tree in LLVM. ✓

→ Regions, **Dominators** in MLIR, its problems.

→ Potential Solutions.

Dominance of MLIR regions

parent_region

some.op

region0

some.op

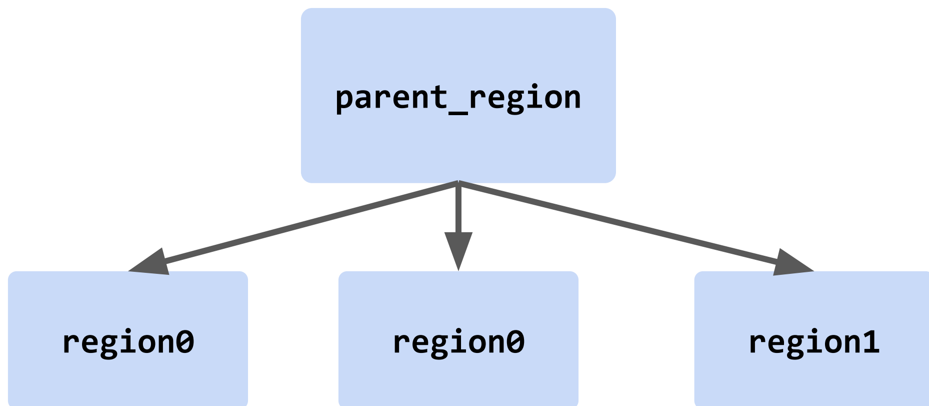
region0

region1

```
func @foo(%arg0: index, %arg1: index) {
  // parent_region
  "some.op"() ({
    // region0
    use(%arg0)
  }) : () -> ()

  "some.op"() ({
    // region0
    use(%arg0)
  }, {
    // region1
    use(%arg1)
  }) : () -> ()
}
```

Dominance of MLIR regions



Dominance across graph regions is the same

```
func @foo(%arg0: index, %arg1: index) {
  // parent_region
  "some.op"() ({
    // region0
    use(%arg0)
  }) : () -> ()

  "some.op"() ({
    // region0
    use(%arg0)
  }, {
    // region1
    use(%arg1)
  }) : () -> ()
}
```

Introduction

What is dominance & why is it important?

→ Key property of SSA. ✓

→ Used to reason about liveness & reachability. ✓

```
Instruction does not dominate all uses!
  %x = add i32 %6, 10
  store i32 %x, i32* %3, align 4
```

Agenda:

→ Dominance & Dominator Tree in LLVM ✓

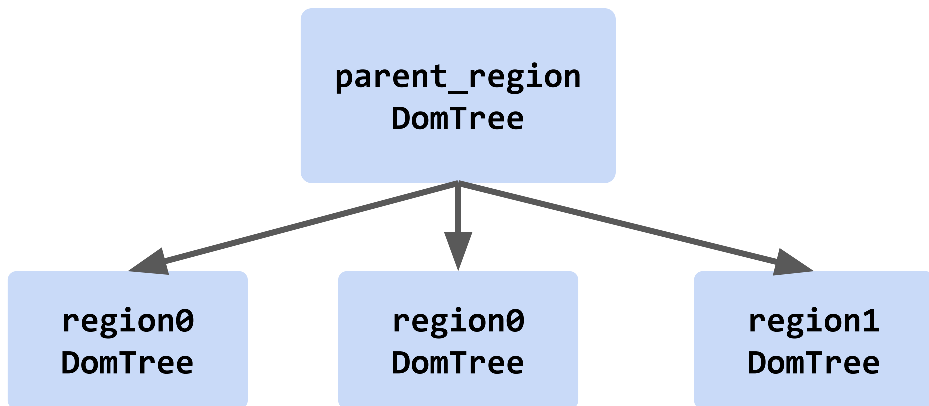
→ Dominance & **Dominator Tree** in MLIR; its problems ⬅

Dominator Tree in MLIR

```
/// Return true if the specified block A properly dominates block B.
bool DominanceInfoBase::properlyDominates(Block *a, Block *b) const {
  // A block dominates itself but does not properly dominate itself.
  if (a == b) return false;
  // If both blocks are not in the same region, `a` properly dominates `b` if
  // `b` is defined in an operation region that (recursively) ends up being
  // dominated by `a`. Walk up the list of containers enclosing B.
  Region *regionA = a->getParent();
  if (regionA != b->getParent()) {
    b = regionA ? regionA->findAncestorBlockInRegion(*b) : nullptr;
    // If we could not find a valid block b then it is a not a dominator.
    if (b == nullptr)
      return false;

    // Check to see if the ancestor of `b` is the same block as `a`.  A properly
    // dominates B if it contains an op that contains the B block.
    if (a == b)
      return true;
  }
  // Otherwise, they are two different blocks in the same region, use DomTree.
  return getDomTree(regionA).properlyDominates(a, b);
}
```

Dominator Tree in MLIR



"Hierarchical Dominance"

Syntactic, **not** Semantic!

```
func @foo(%arg0: index, %arg1: index) {
  // parent_region
  "some.op"() ({
    // region0
    use(%arg0)
  }) : () -> ()

  "some.op"() ({
    // region0
    use(%arg0)
  }, {
    // region1
    use(%arg1)
  }) : () -> ()
}
```

Agenda

What is dominance & why is it important?

→ Key property of SSA (liveness & reachability). ✓

```
Instruction does not dominate all uses!
  %x = add i32 %6, 10
  store i32 %x, i32* %3, align 4
```

→ Dominance, Dominator Tree in LLVM. ✓

→ Regions, Dominators in MLIR, **its problems**.

→ Potential Solutions.

The sane world

I.e. functions and control-flow

I.e. semantically equivalent to CFGs

But *stronger* guarantees (e.g. reducibility)

```
func @foo(%c: i1, %v: index) -> index {
  %0 = scf.if %c -> index {
    %idx0 = index.constant 0
    hlcf.yield %idx0 : index
  } else {
    hlcf.yield %v : index
  }
  return %0 : index
}
```

Implicit capture

RegionBranchOpInterface

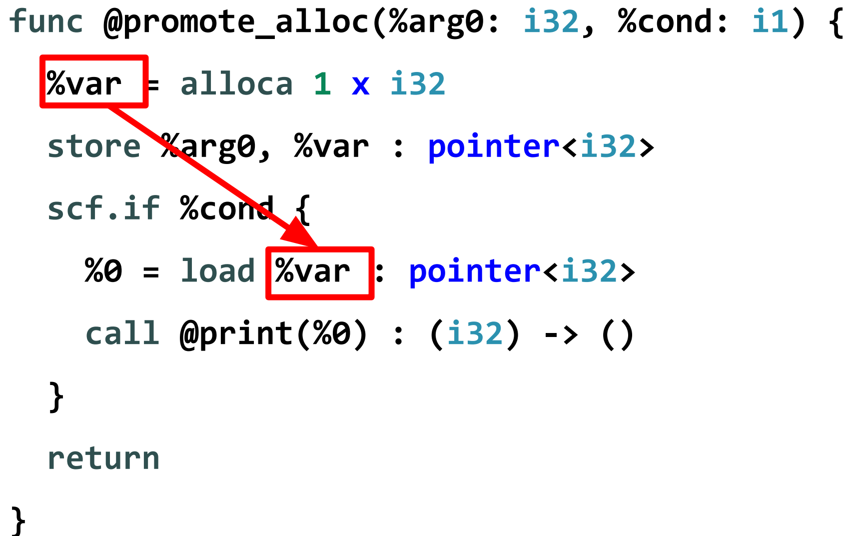
Mem2Reg and dominance

Mem2Reg uses dominance to compute liveness of local variables

```
func @promote_alloc(%arg0: i32, %cond: i1) {
  %var = alloca 1 x i32
  store %arg0, %var : pointer<i32>
  scf.if %cond {
    %0 = load %var : pointer<i32>
    call @print(%0) : (i32) -> ()
  }
  return
}
```

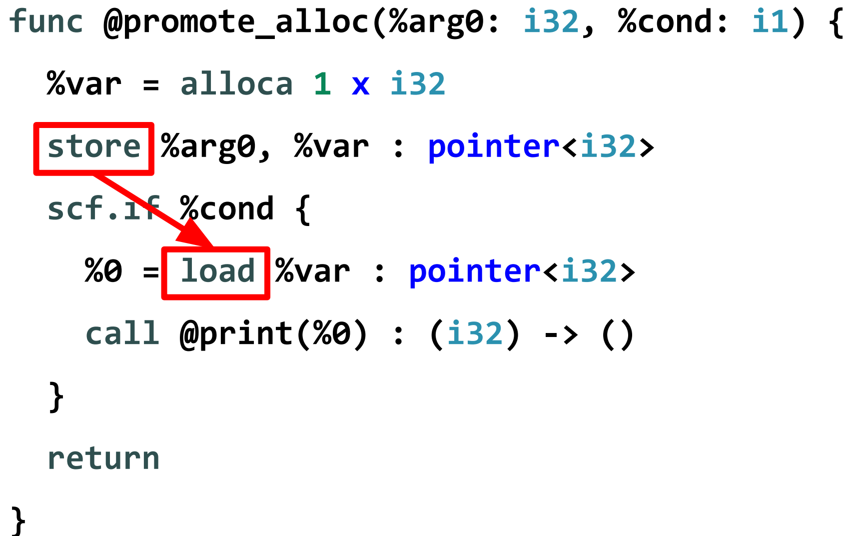
Mem2Reg and dominance

The alloca dominates the load, therefore the variable is live



Mem2Reg and dominance

Nearest dominating store of the load is store of %arg0



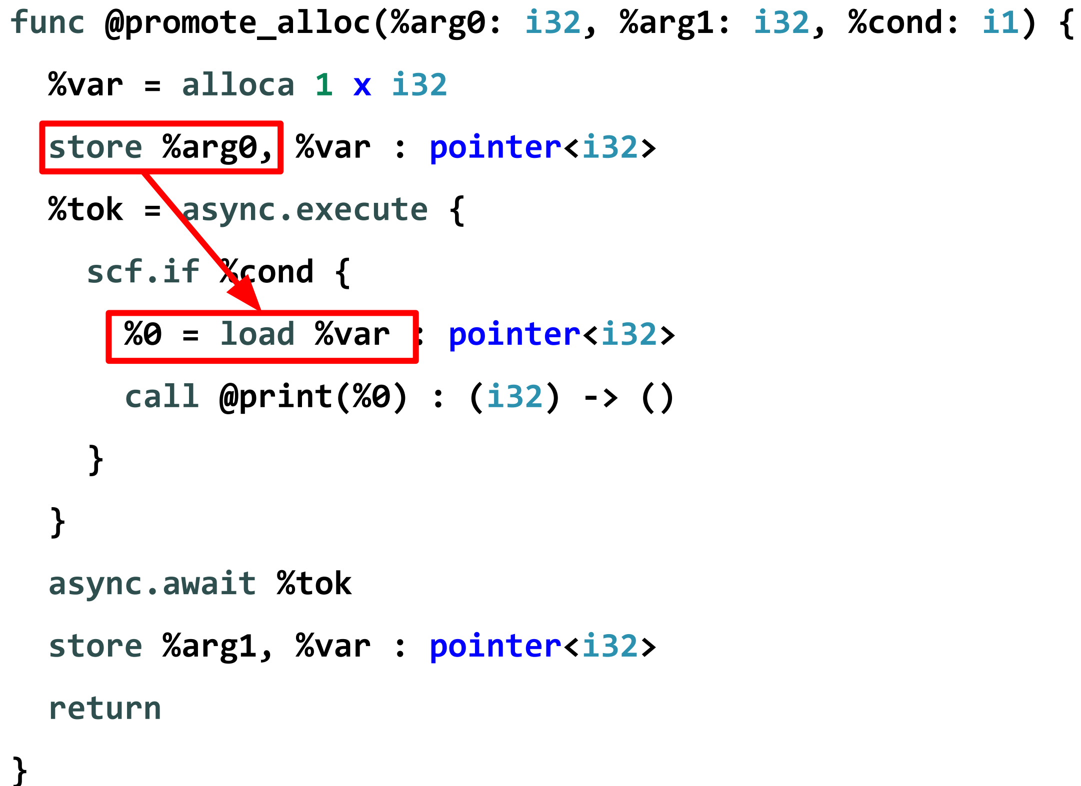
Mem2Reg and dominance

Therefore load result can be replaced with %arg0

```
func @promote_alloc(%arg0: i32, %cond: i1) {
  %var = alloca 1 x i32
  store %arg0, %var : pointer<i32>
  scf.if %cond {
    %0 = load %var : pointer<i32>
    call @print(%arg0) : (i32) -> ()
  }
  return
}
```

RegionBranchOpInterface

Staring into the abyss



Staring into the abyss

```
func @promote_alloc(%arg0: i32, %arg1: i32, %cond: i1) {
  %var = alloca 1 x i32
  store %arg0, %var : pointer<i32>
  %tok = async.execute {
    scf.if %cond {
      %0 = load %var : pointer<i32>
      call @print(%arg0) : (i32) -> ()
    }
  }
  async.await %tok
  store %arg1, %var : pointer<i32>
  return
}
```

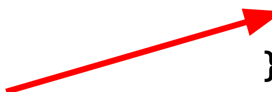
Waits for region to execute

Dead store

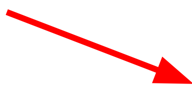
Staring into the abyss

```
func @promote_alloc(%arg0: i32, %arg1: i32, %cond: i1) {
  %var = alloca 1 x i32
  store %arg0, %var : pointer<i32>
  %tok = async.execute {
    scf.if %cond {
      %0 = load %var : pointer<i32>
      call @print(%arg0) : (i32) -> ()
    }
  }

  store %arg1, %var : pointer<i32>
  return
}
```



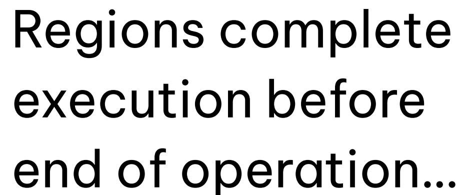
May execute after second store!



Staring into the abyss

```
%tensor = arith.constant dense<0> : tensor<4xi32>
scf.parallel (%i) = (%zero) to (%four)
    step (%step) init(%zero) -> i32 {
  %tensor_0 = arith.constant dense<0> : tensor<4xi32>
  %e = tensor.extract %tensor_0[%i] : tensor<4xi32>
  scf.reduce(%e) : i32 {
    ^bb0(%lhs: i32, %rhs: i32):
    %res = arith.addi %lhs, %rhs : i32
    scf.reduce.return %res : i32
  }
}
```

Staring into the abyss




```
%tensor = arith.constant dense<0> : tensor<4xi32>
scf.parallel (%i) = (%zero) to (%four)
    step (%step) init(%zero) -> i32 {
  %tensor_0 = arith.constant dense<0> : tensor<4xi32>
  %e = tensor.extract %tensor_0[%i] : tensor<4xi32>
  scf.reduce(%e) : i32 {
    ^bb0(%lhs: i32, %rhs: i32):
    %res = arith.addi %lhs, %rhs : i32
    scf.reduce.return %res : i32
  }
}
```

Staring into the abyss

%tensor dominates
%tensor_0

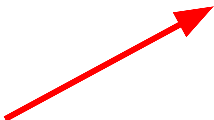
```
%tensor = arith.constant dense<0> : tensor<4xi32>
scf.parallel (%i) = (%zero) to (%four)
    step (%step) init(%zero) -> i32 {
  %tensor_0 = arith.constant dense<0> : tensor<4xi32>
  %e = tensor.extract %tensor_0[%i] : tensor<4xi32>
  scf.reduce(%e) : i32 {
    ^bb0(%lhs: i32, %rhs: i32):
    %res = arith.addi %lhs, %rhs : i32
    scf.reduce.return %res : i32
  }
}
```

Staring into the abyss

CSE

```
%tensor = arith.constant dense<0> : tensor<4xi32>
scf.parallel (%i) = (%zero) to (%four)
    step (%step) init(%zero) -> i32 {
  %tensor_0 = arith.constant dense<0> : tensor<4xi32>
  %e = tensor.extract %tensor_0[%i] : tensor<4xi32>
  scf.reduce(%e) : i32 {
    ^bb0(%lhs: i32, %rhs: i32):
    %res = arith.addi %lhs, %rhs : i32
    scf.reduce.return %res : i32
  }
}
```

Staring into the abyss



Can lower to async.execute... or gpu.launch!!

Where does the memory live? 🤔

```
%tensor = arith.constant dense<0> : tensor<4xi32>
scf.parallel (%i) = (%zero) to (%four)
    step (%step) init(%zero) -> i32 {
  %tensor_0 = arith.constant dense<0> : tensor<4xi32>
  %e = tensor.extract %tensor[%i] : tensor<4xi32>
  scf.reduce(%e) : i32 {
    ^bb0(%lhs: i32, %rhs: i32):
    %res = arith.addi %lhs, %rhs : i32
    scf.reduce.return %res : i32
  }
}
```

Staring into the abyss

Host memory

Device memory

```
%tensor = arith.constant dense<0> : tensor<4xi32>
scf.parallel (%i) = (%zero) to (%four)
    step (%step) init(%zero) -> i32 {
  %tensor_0 = arith.constant dense<0> : tensor<4xi32>
  %e = tensor.extract %tensor[%i] : tensor<4xi32>
  scf.reduce(%e) : i32 {
    ^bb0(%lhs: i32, %rhs: i32):
    %res = arith.addi %lhs, %rhs : i32
    scf.reduce.return %res : i32
  }
}
```

Read of host memory

Dominance in MLIR: Static v/s Dynamic

LLVM
Dynamic property (reachability)

MLIR
Static property (nesting of regions)

What does this preclude?

`mem2reg` in MLIR:

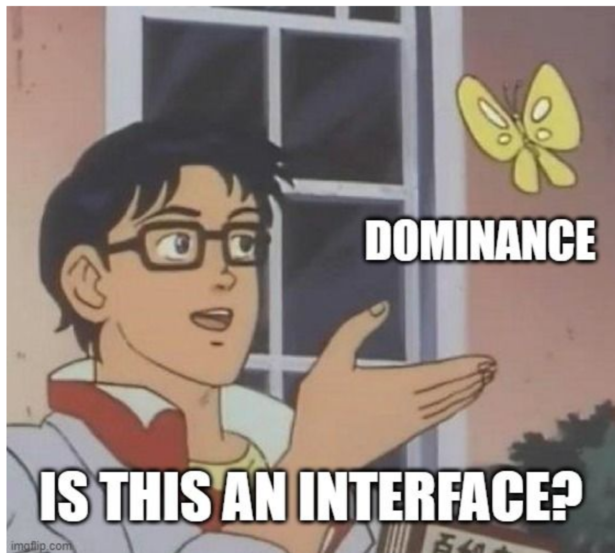
- Already implemented on CFGs! See Théo's talk
- Cannot be written in general for regions - RegionBranchOpInterface is not enough

Anything that needs dominance à la LLVM

- CSE, DCE, LICM,
- Even canonicalizer patterns that match across regions…

Evolving Dominance in MLIR

Add Interfaces!



Op defines **how**
parent region connects to child

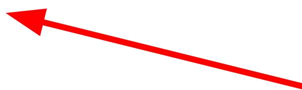
Describe liveness relationships?

Ask the region if it changes semantics of ops inside of it and define how? 🤔

How this helps

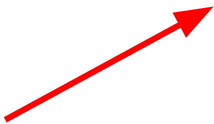
```
func @promote_alloc(%arg0: i32, %arg1: i32, %cond: i1) {
  %var = alloca 1 x i32
  store %arg0, %var : pointer<i32>
  %tok = async.execute {
    scf.if %cond {
      %0 = load %var : pointer<i32>
      call @print(%0) : (i32) -> ()
    }
  }

  store %arg1, %var : pointer<i32>
  return
}
```



“I don’t understand this region” - `mem2reg`

How this helps



Op may change the semantics of the region

(CPU vs GPU)

```
%tensor = arith.constant dense<0> : tensor<4xi32>
scf.parallel (%i) = (%zero) to (%four)
    step (%step) init(%zero) -> i32 {
  %tensor_0 = arith.constant dense<0> : tensor<4xi32>
  %e = tensor.extract %tensor_0[%i] : tensor<4xi32>
  scf.reduce(%e) : i32 {
    ^bb0(%lhs: i32, %rhs: i32):
    %res = arith.addi %lhs, %rhs : i32
    scf.reduce.return %res : i32
  }
}
```

Agenda

What is dominance & why is it important?

→ Key property of SSA (liveness & reachability).

```
Instruction does not dominate all uses!
  %x = add i32 %6, 10
  store i32 %x, i32* %3, align 4
```

→ Dominance, Dominator Tree in LLVM.

→ Regions, Dominators in MLIR, its problems.

→ Potential Solutions (New interface).

→ Design **not final! Please** talk to us with your ideas!