GVN and Inlining

15-411/15-611 Compiler Design

Ben L. Titzer and Seth Goldstein

April 15, 2025

Today

- Local Value Numbering
- Global Value Numbering
- Inlining
- Static inlining heuristics
- Dynamic inlining heuristics

Value Numbering

- Already covered:
  - common subexpression elimination (CSE)
  - lazy code motion
  - load/store elimination
- Value numbering is a specific approach to CSE
- Unlike lazy code motion, no instructions are moved
- Only previous results of computations are reused
- Works especially well on SSA
- Efficient forward algorithm

Local Value Numbering

- Value numbering is a forward dataflow analysis that assigns *unique numbers* to equivalent computations
- If you like hash tables, it's basically that!
- Usually applied to *pure* computations
- Easiest and best results on SSA form

```
t ← x - y
u ← x + y
v ← x - y
```

Any redundant computations?

Local Value Numbering

- Value numbering is a forward dataflow analysis that assigns *unique numbers* to equivalent computations
- If you like hash tables, it's basically that!
- Usually applied to *pure* computations
- Easiest and best results on SSA form

```
t ← x - y
u ← x + y
v ← x - y
```

Redundant computations

Local Value Numbering

- Value numbering is a forward dataflow analysis that assigns *unique numbers* to equivalent computations
- If you like hash tables, it's basically that!
- Usually applied to *pure* computations
- Easiest and best results on SSA form:
  - Variable renaming guarantees no overwrite

$$
\begin{array}{l}
t_0 \leftarrow \boxed{x_0 - y_0} \\
u_0 \leftarrow x_0 + y_0 \\
v_0 \leftarrow \boxed{x_0 - y_0}
\end{array}
$$

Redundant computations

Local Value Numbering

- Forward-pass over basic block
- Maintain map of (operator, inputs) ↦ variable
  - Represents the GEN part of a typical DF analysis
- Use hash function to speed lookup
- Skip impure instructions

cursor

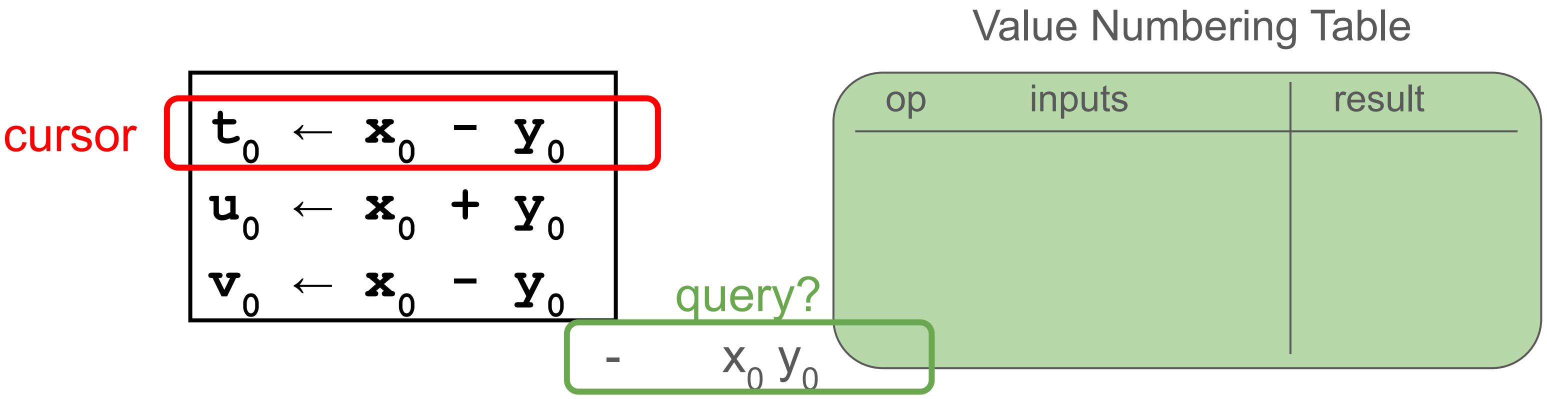
$$\boxed{t_0 \leftarrow x_0 - y_0}$$
$$u_0 \leftarrow x_0 + y_0$$
$$v_0 \leftarrow x_0 - y_0$$

Value Numbering Table

op	inputs	result

Local Value Numbering

- Forward-pass over basic block
- Maintain map of (operator, inputs) ↦ variable
  - Represents the GEN part of a typical DF analysis
- Use hash function to speed lookup
- Skip impure instructions



Local Value Numbering

- Forward-pass over basic block
- Maintain map of (operator, inputs) ↦ variable
  - Represents the GEN part of a typical DF analysis
- Use hash function to speed lookup
- Skip impure instructions

cursor

$$\boxed{t_0 \leftarrow x_0 - y_0}$$
$$u_0 \leftarrow x_0 + y_0$$
$$v_0 \leftarrow x_0 - y_0$$

Value Numbering Table

op	inputs	result
-	x_0 y_0	t_0

new entry

Local Value Numbering

- Forward-pass over basic block
- Maintain map of (operator, inputs) ↦ variable
  - Represents the GEN part of a typical DF analysis
- Use hash function to speed lookup
- Skip impure instructions

$t_0 \leftarrow x_0 - y_0$

cursor $u_0 \leftarrow x_0 + y_0$

$v_0 \leftarrow x_0 - y_0$

Value Numbering Table

op	inputs	result
-	x_0 y_0	t_0

Local Value Numbering

- Forward-pass over basic block
- Maintain map of (operator, inputs) ↦ variable
  - Represents the GEN part of a typical DF analysis
- Use hash function to speed lookup
- Skip impure instructions

$t_0 \leftarrow x_0 - y_0$

cursor $\boxed{u_0 \leftarrow x_0 + y_0}$

$v_0 \leftarrow x_0 - y_0$

query?

$\boxed{+ \quad x_0 \; y_0}$

Value Numbering Table

op	inputs	result
-	x_0 y_0	t_0

Local Value Numbering

- Forward-pass over basic block
- Maintain map of (operator, inputs) ↦ variable
  - Represents the GEN part of a typical DF analysis
- Use hash function to speed lookup
- Skip impure instructions

$t_0 \leftarrow x_0 - y_0$

cursor $u_0 \leftarrow x_0 + y_0$

$v_0 \leftarrow x_0 - y_0$

Value Numbering Table

op	inputs	result
-	x_0 y_0	t_0
+	x_0 y_0	u_0

new entry

Local Value Numbering

- Forward-pass over basic block
- Maintain map of (operator, inputs) ↦ variable
  - Represents the GEN part of a typical DF analysis
- Use hash function to speed lookup
- Skip impure instructions

cursor

$t_0 \leftarrow x_0 - y_0$
$u_0 \leftarrow x_0 + y_0$
$\boxed{v_0 \leftarrow x_0 - y_0}$

Value Numbering Table

op	inputs	result
-	x_0 y_0	t_0
+	x_0 y_0	u_0

Local Value Numbering

- Forward-pass over basic block
- Maintain map of (operator, inputs) ↦ variable
  - Represents the GEN part of a typical DF analysis
- Use hash function to speed lookup
- Skip impure instructions

$t_0 \leftarrow x_0 - y_0$

$u_0 \leftarrow x_0 + y_0$

cursor $v_0 \leftarrow x_0 - y_0$

query? $-$ x_0 y_0

Value Numbering Table

op	inputs	result
-	x_0 y_0	t_0
+	x_0 y_0	u_0

Local Value Numbering

- Forward-pass over basic block
- Maintain map of (operator, inputs) ↦ variable
  - Represents the GEN part of a typical DF analysis
- Use hash function to speed lookup
- Skip impure instructions

$t_0 \leftarrow x_0 - y_0$
$u_0 \leftarrow x_0 + y_0$
cursor $v_0 \leftarrow x_0 - y_0$

hit!!

Value Numbering Table

op	inputs	result
-	x_0 y_0	t_0
+	x_0 y_0	u_0

Local Value Numbering

- Forward-pass over basic block
- Maintain map of (operator, inputs) ↦ variable
  - Represents the GEN part of a typical DF analysis
- Use hash function to speed lookup
- Skip impure instructions

$t_0 \leftarrow x_0 - y_0$

$u_0 \leftarrow x_0 + y_0$

remove $\cancel{v_0 \leftarrow x_0 - y_0}$

rename $v_0 \mapsto t_0$

hit!!

Value Numbering Table

op	inputs	result
-	x_0 y_0	t_0
+	x_0 y_0	u_0

Local Value Numbering

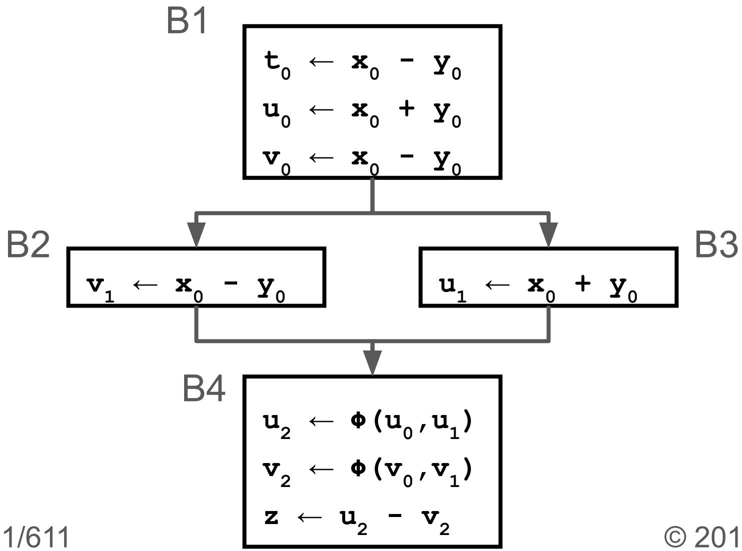
- Simplest form of common subexpression elimination
- No-brainer on SSA form
- Decent hash function needed
- Extremely fast in practice
- Implement this (and GVN) before you try lazy code motion!
- Can extend to impure computations:
  - Requires proper invalidation of numbering table (KILL rules)
  - Good candidate for combining with load elimination
  - Be conservative across side-effecting operations like calls

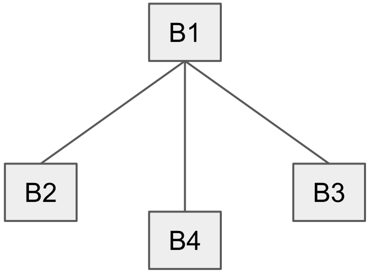
Global Value Numbering

- LVN finds redundant computations in a basic block
- Generalize the algorithm to whole function, that's GVN!
- Requires propagating per-block information from predecessor(s) to successor(s)
- Faster but less powerful dominator-based algorithm
- Forward worklist algorithm does better
- Can do this with a general DF solver
  - Still doesn't move computations like lazy code motion
  - Don't get loop invariant code motion like lazy code motion
- Benefits from independent loop invariant code motion pass

Global Value Numbering using Dominators

- LVN finds redundant computations in a basic block
- Dominator-based algorithm trades accuracy for speed
- Like load elimination from last lecture, propagate local value numbering map from dominator to dominated nodes

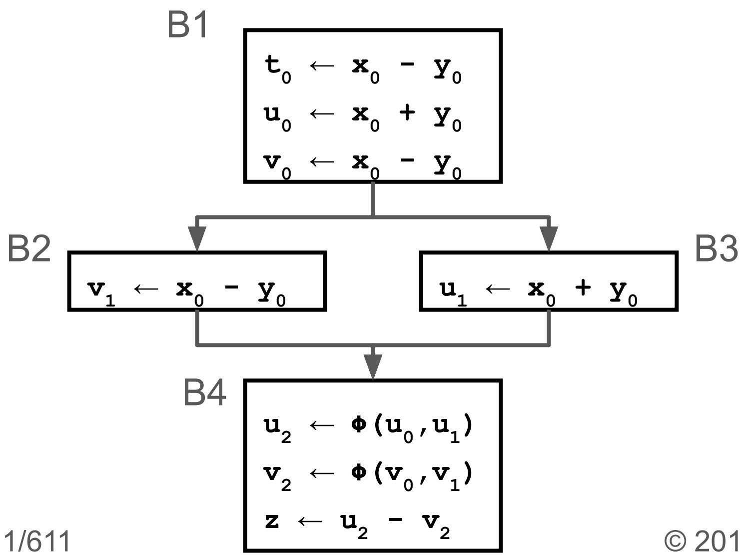


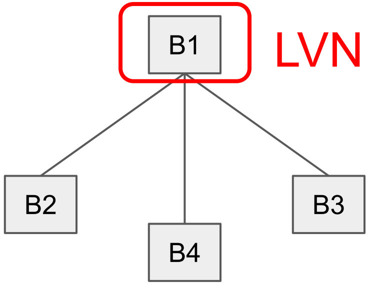


Dominator tree

Global Value Numbering using Dominators

- LVN finds redundant computations in a basic block
- Dominator-based algorithm trades accuracy for speed
- Like load elimination from last lecture, propagate local value numbering map from dominator to dominated nodes

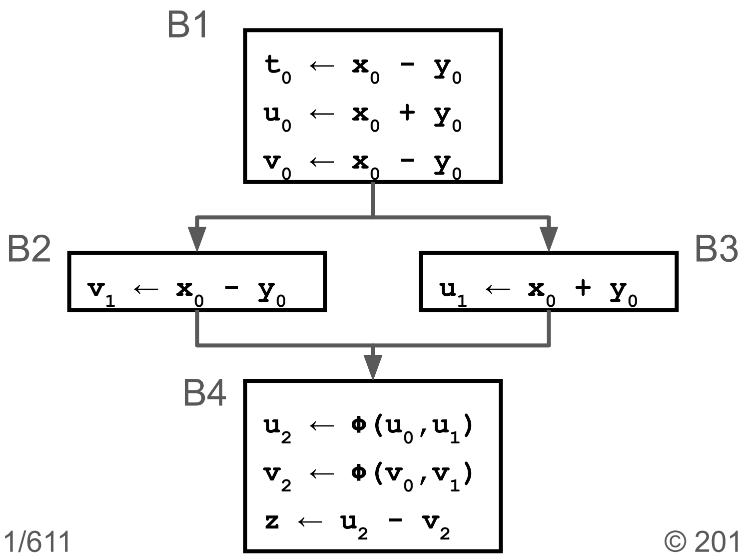


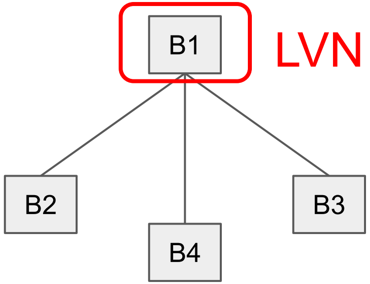


Dominator tree

Global Value Numbering using Dominators

- LVN finds redundant computations in a basic block
- Dominator-based algorithm trades accuracy for speed
- Like load elimination from last lecture, propagate local value numbering map from dominator to dominated nodes



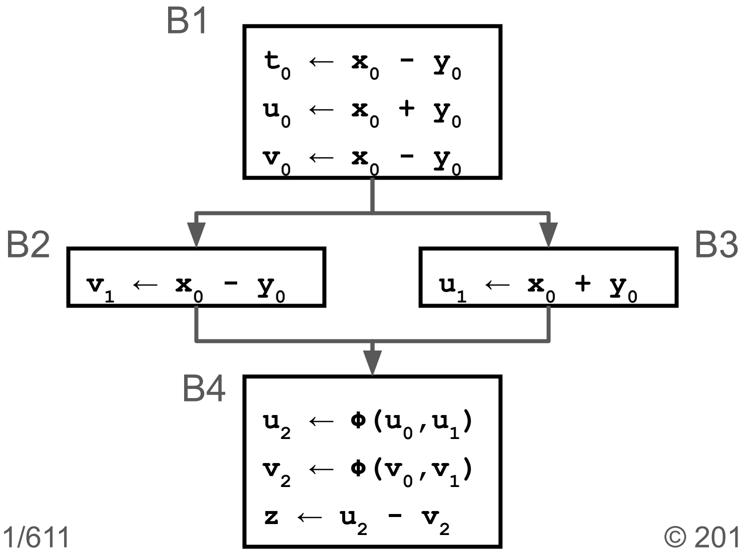


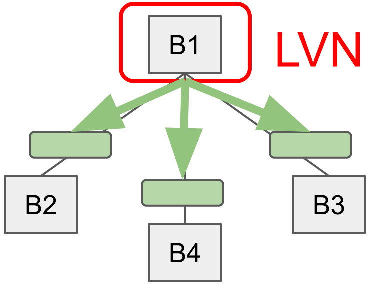
Dominator tree

op	inputs	result
+	x_0 y_0	u_0

Global Value Numbering using Dominators

- LVN finds redundant computations in a basic block
- Dominator-based algorithm trades accuracy for speed
- Like load elimination from last lecture, propagate local value numbering map from dominator to dominated nodes



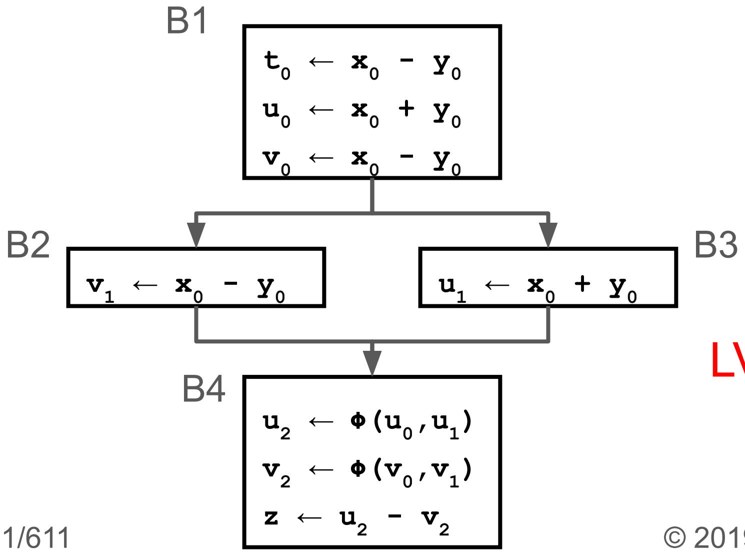


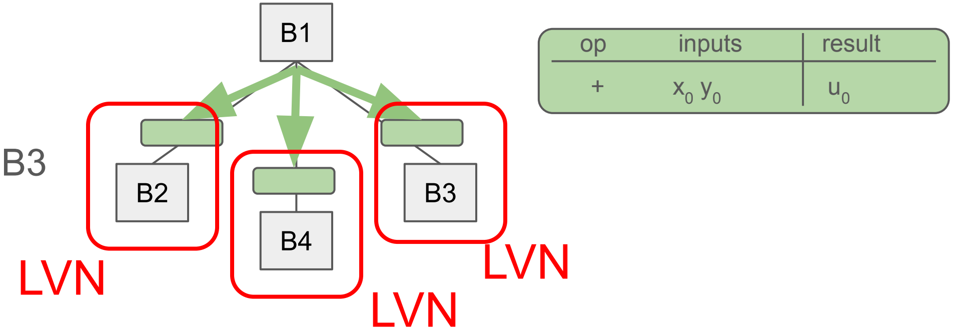
op	inputs	result
+	x_0 y_0	u_0

Dominator tree

Global Value Numbering using Dominators

- LVN finds redundant computations in a basic block
- Dominator-based algorithm trades accuracy for speed
- Like load elimination from last lecture, propagate local value numbering map from dominator to dominated nodes

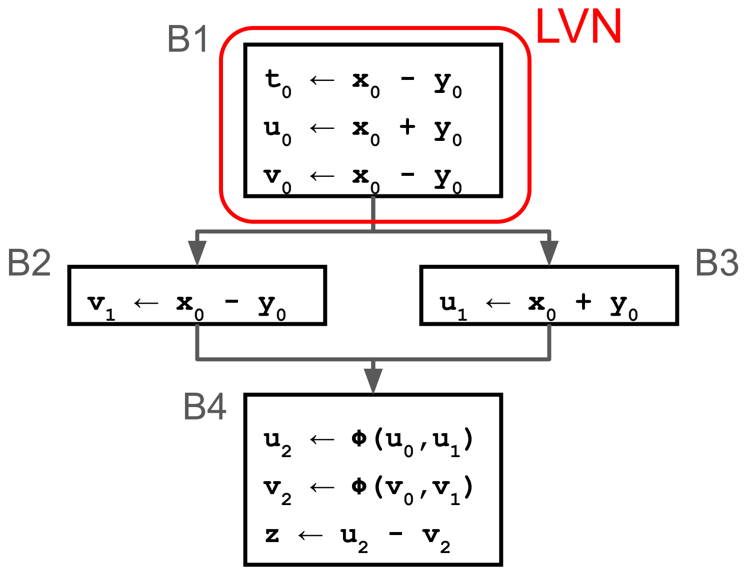




Dominator tree

Global Value Numbering using Dominators

- Perform LVN on dominator, propagate local value numbering map from dominator to dominated nodes



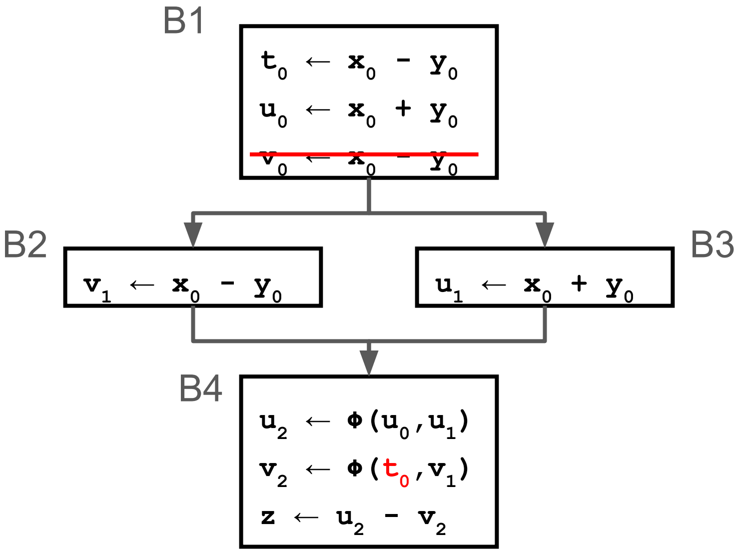
Value Numbering Table

op	inputs	result
-	x_0 y_0	t_0
+	x_0 y_0	u_0

rename $v_0 \mapsto t_0$

Global Value Numbering using Dominators

- Perform LVN on dominator, propagate local value numbering map from dominator to dominated nodes



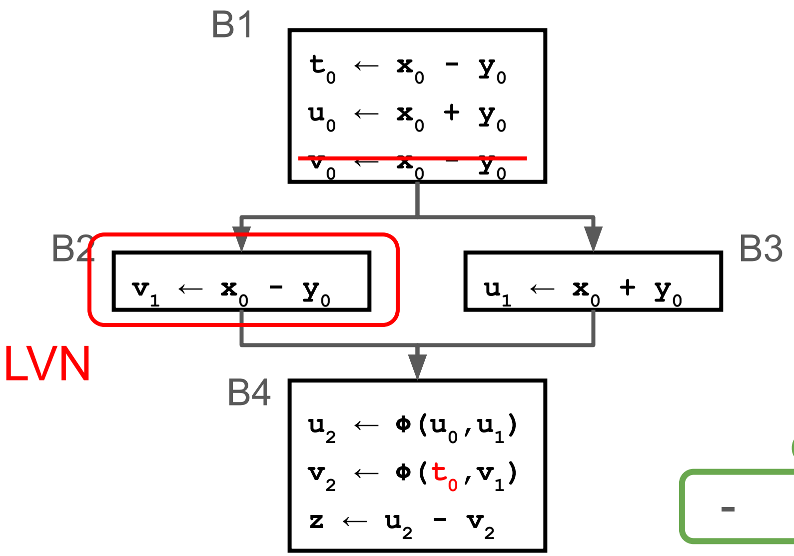
Value Numbering Table

op	inputs	result
-	x_0 y_0	t_0
+	x_0 y_0	u_0

rename $v_0 \mapsto t_0$

Global Value Numbering using Dominators

- Perform LVN on dominator, propagate local value numbering map from dominator to dominated nodes



Value Numbering Table

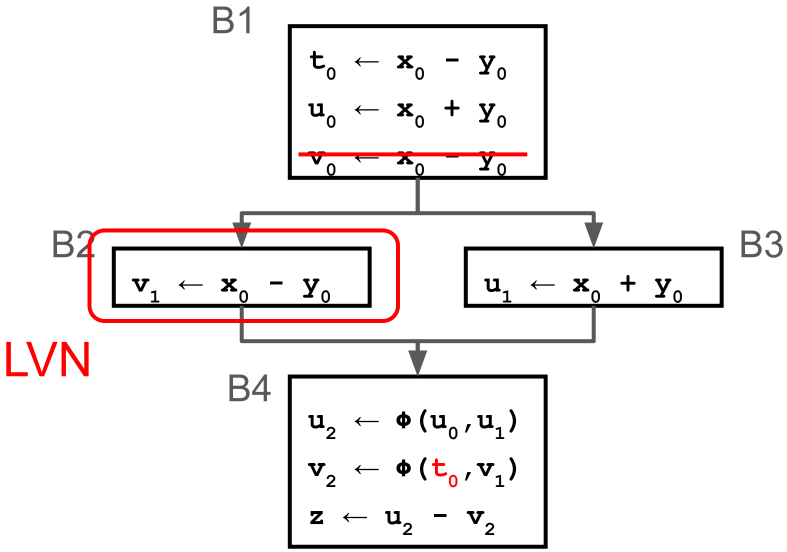
op	inputs	result
-	x_0 y_0	t_0
+	x_0 y_0	u_0

query?

-	x_0 y_0	

Global Value Numbering using Dominators

- Perform LVN on dominator, propagate local value numbering map from dominator to dominated nodes



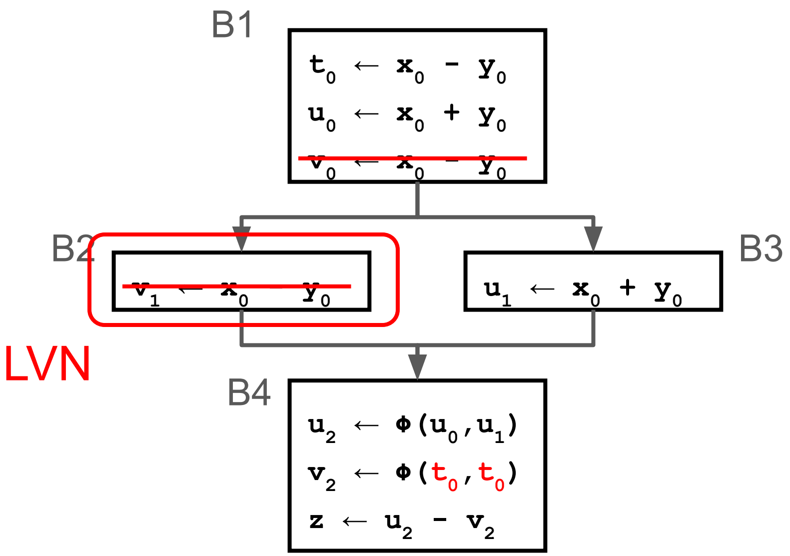
hit!!

Value Numbering Table

op	inputs	result
-	x_0 y_0	t_0
+	x_0 y_0	u_0

Global Value Numbering using Dominators

- Perform LVN on dominator, propagate local value numbering map from dominator to dominated nodes



hit!!

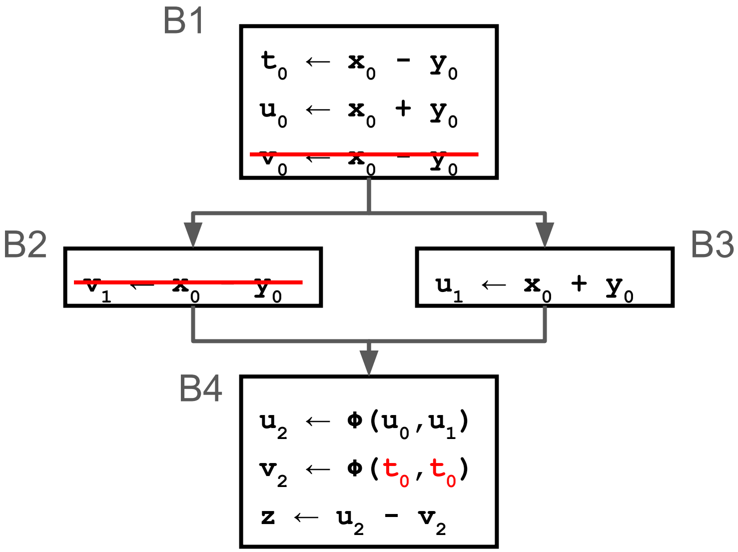
Value Numbering Table

op	inputs	result
-	$x_0\ y_0$	t_0
+	$x_0\ y_0$	u_0

rename $\mathbf{v_1} \mapsto \mathbf{t_0}$

Global Value Numbering using Dominators

- Perform LVN on dominator, propagate local value numbering map from dominator to dominated nodes



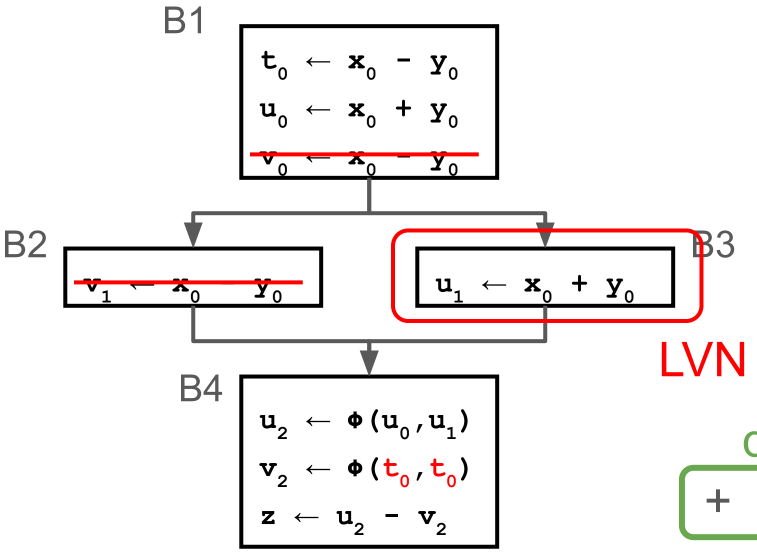
Value Numbering Table

op	inputs	result
-	$x_0\ y_0$	t_0
+	$x_0\ y_0$	u_0

rename $v_1 \mapsto t_0$

Global Value Numbering using Dominators

- Perform LVN on dominator, propagate local value numbering map from dominator to dominated nodes



Value Numbering Table

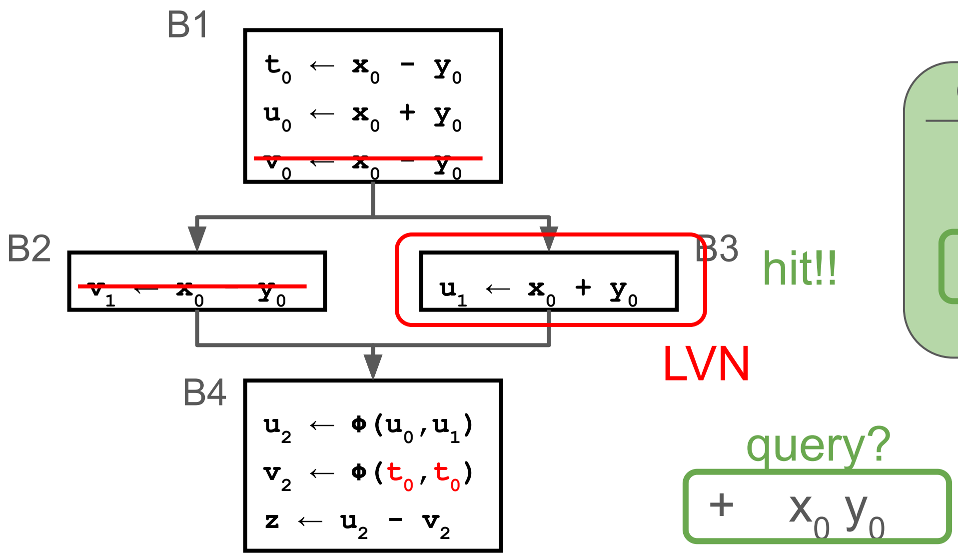
op	inputs	result
-	x_0 y_0	t_0
+	x_0 y_0	u_0

query?

$+ \quad x_0\ y_0$

Global Value Numbering using Dominators

- Perform LVN on dominator, propagate local value numbering map from dominator to dominated nodes

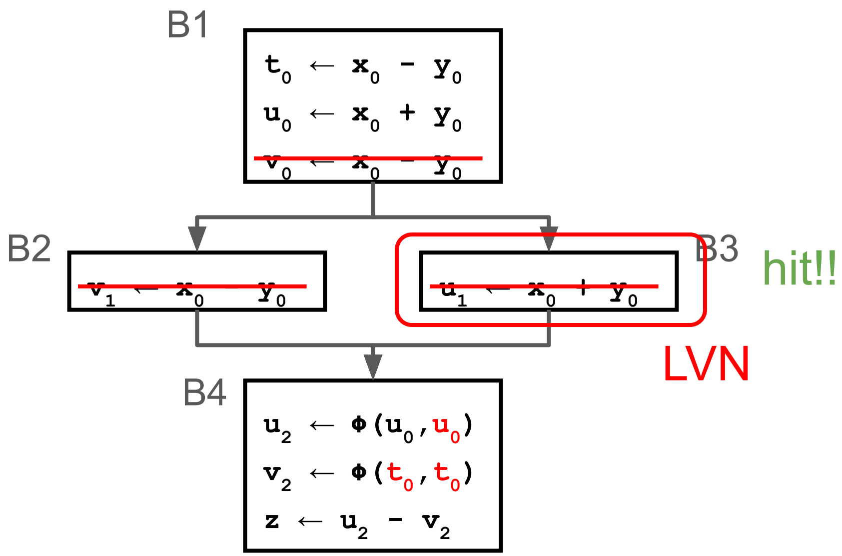


Value Numbering Table

op	inputs	result
-	x_0 y_0	t_0
+	x_0 y_0	u_0

Global Value Numbering using Dominators

- Perform LVN on dominator, propagate local value numbering map from dominator to dominated nodes



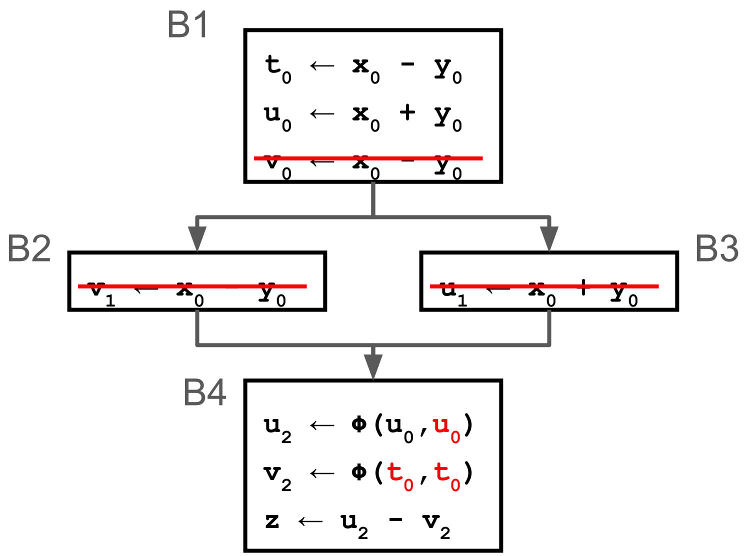
Value Numbering Table

op	inputs	result
-	x_0 y_0	t_0
+	x_0 y_0	u_0

rename $u_1 \mapsto u_0$

Global Value Numbering using Dominators

- Perform LVN on dominator, propagate local value numbering map from dominator to dominated nodes

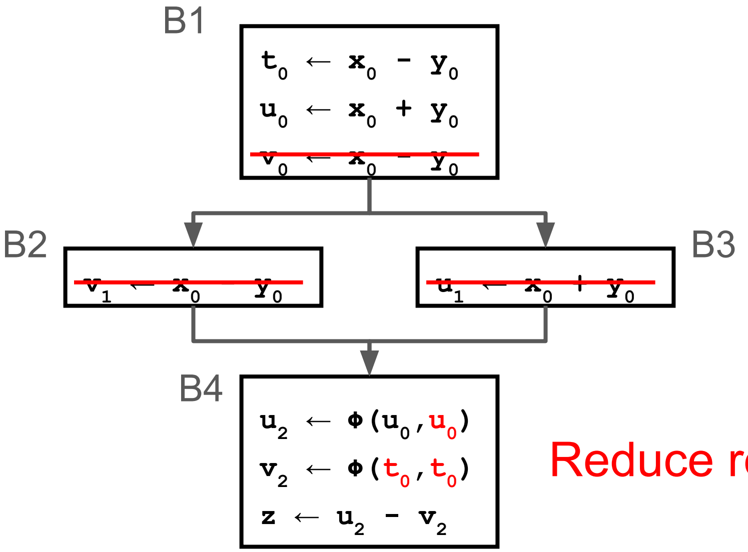


Value Numbering Table

op	inputs	result
-	x_0 y_0	t_0
+	x_0 y_0	u_0

Global Value Numbering using Dominators

- Perform LVN on dominator, propagate local value numbering map from dominator to dominated nodes



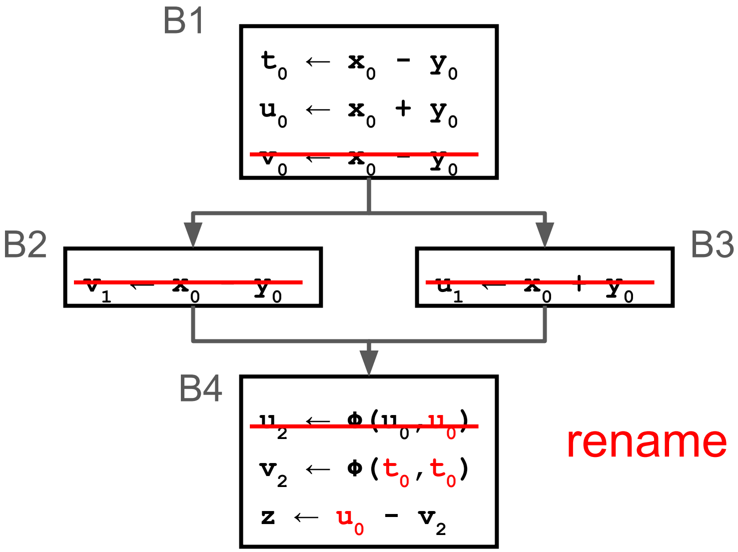
Reduce redundant ϕ

Value Numbering Table

op	inputs	result
-	x_0 y_0	t_0
+	x_0 y_0	u_0

Global Value Numbering using Dominators

- Perform LVN on dominator, propagate local value numbering map from dominator to dominated nodes



rename

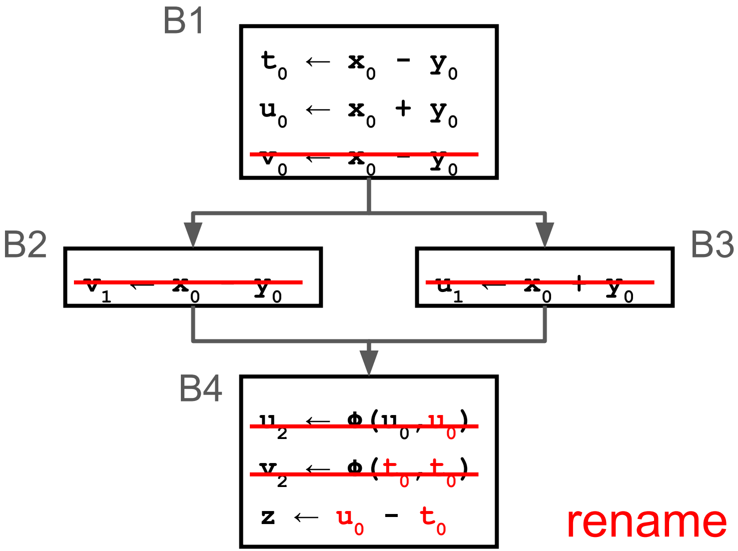
Value Numbering Table

op	inputs	result
-	x_0 y_0	t_0
+	x_0 y_0	u_0

$u_2 \mapsto u_0$

Global Value Numbering using Dominators

- Perform LVN on dominator, propagate local value numbering map from dominator to dominated nodes



rename

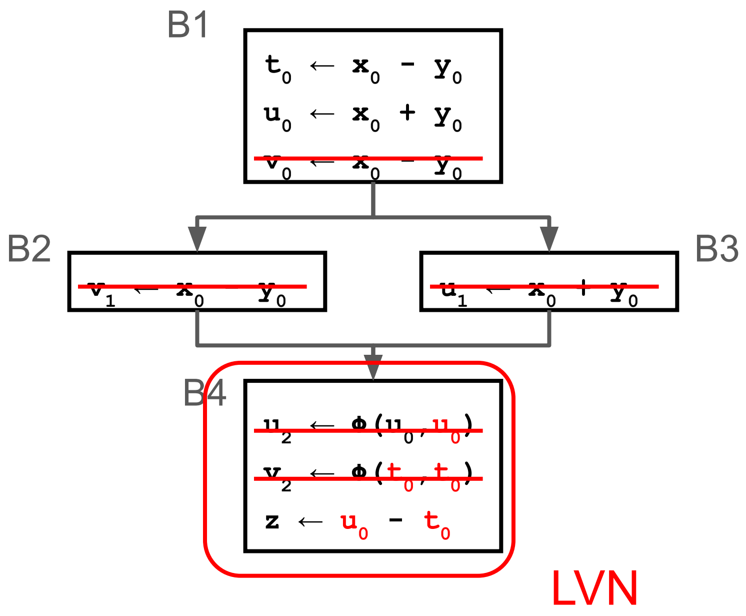
Value Numbering Table

op	inputs	result
-	x_0 y_0	t_0
+	x_0 y_0	u_0

$v_2 \mapsto t_0$

Global Value Numbering using Dominators

- Perform LVN on dominator, propagate local value numbering map from dominator to dominated nodes

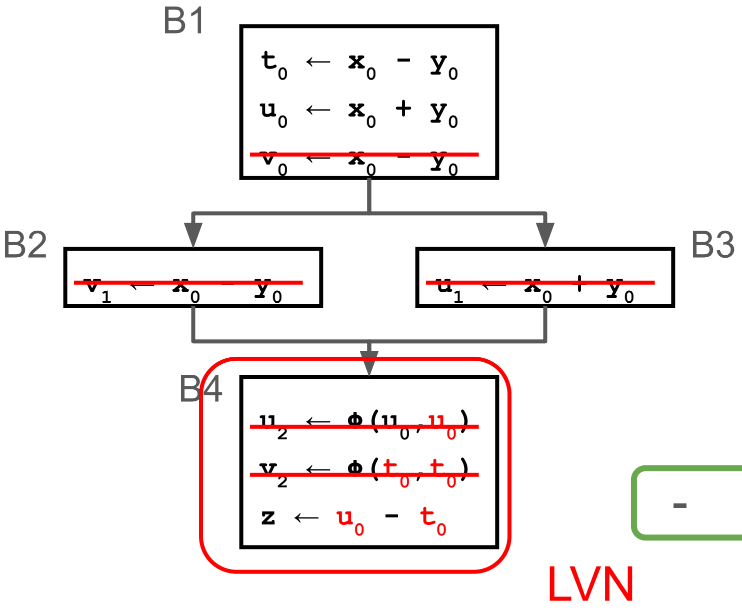


Value Numbering Table

op	inputs	result
-	x_0 y_0	t_0
+	x_0 y_0	u_0

Global Value Numbering using Dominators

- Perform LVN on dominator, propagate local value numbering map from dominator to dominated nodes



Value Numbering Table

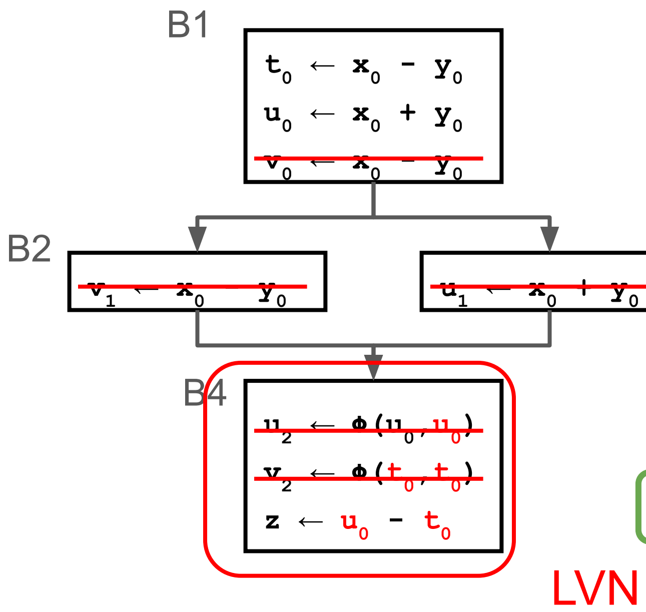
op	inputs	result
-	x_0 y_0	t_0
+	x_0 y_0	u_0

query?

-	u_0 t_0	

Global Value Numbering using Dominators

- Perform LVN on dominator, propagate local value numbering map from dominator to dominated nodes



Value Numbering Table

op	inputs	result
-	x_0 y_0	t_0
+	x_0 y_0	u_0
-	u_0 t_0	z

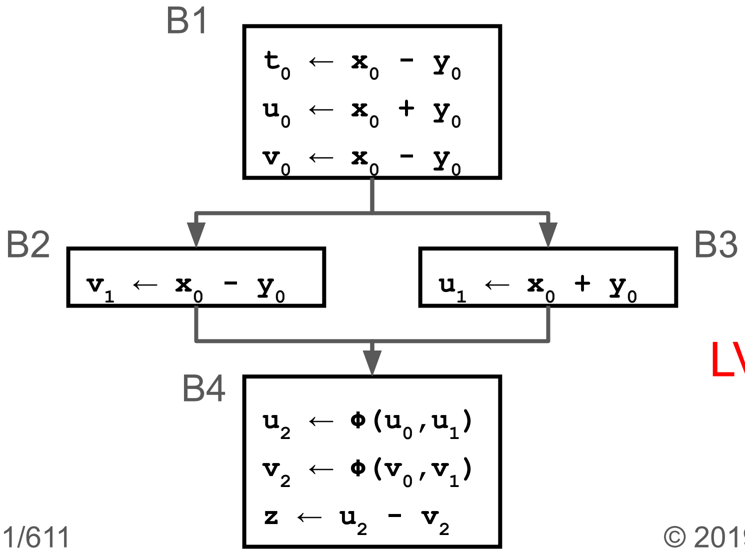
new entry

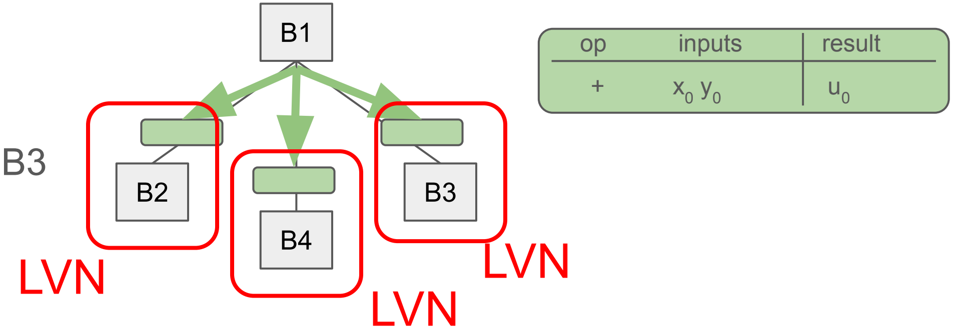
query?

-	u_0 t_0

Global Value Numbering using Dominators

- Propagate local value numbering map from dominator to dominated nodes
- Note: we did *not* propagate along control flow edges!

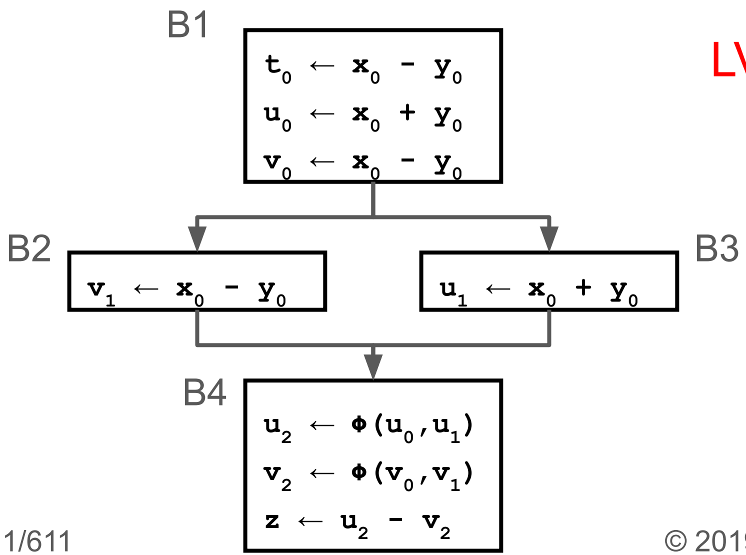


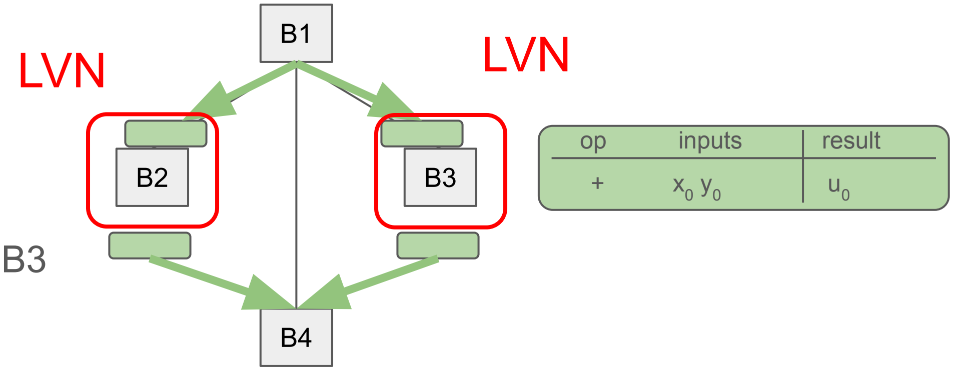


Dominator tree

Global Value Numbering using Dominators

- Propagate local value numbering map from dominator to dominated nodes
- Note: we did *not* propagate along control flow edges!
- What would happen if we did?

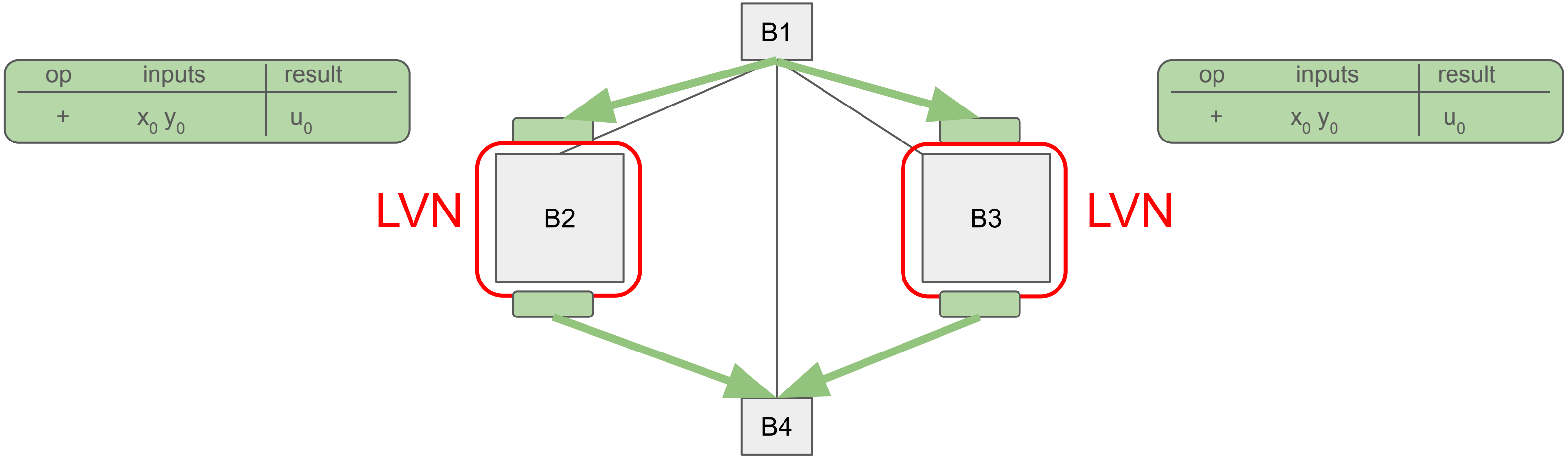




Dominator tree

Global Value Numbering using Dominators

- What would happen if we propagated LVN across CF edges?

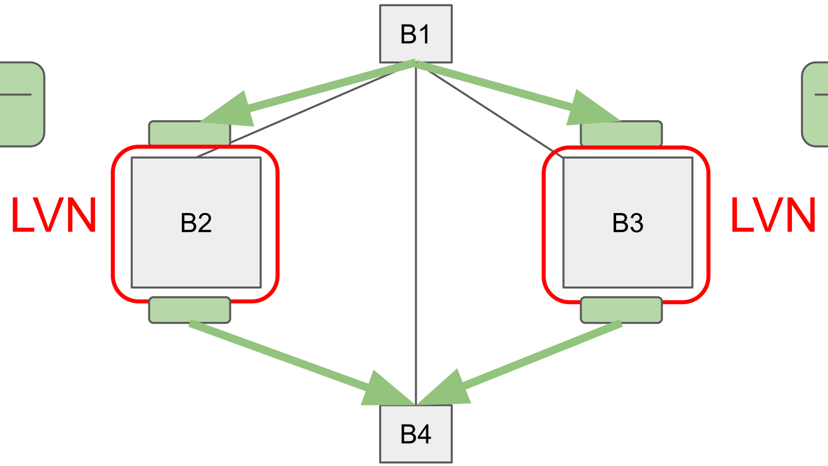


Global Value Numbering using Dominators

- What would happen if we propagated LVN across CF edges?

op	inputs	result
+	x_0 y_0	u_0

+ New entries

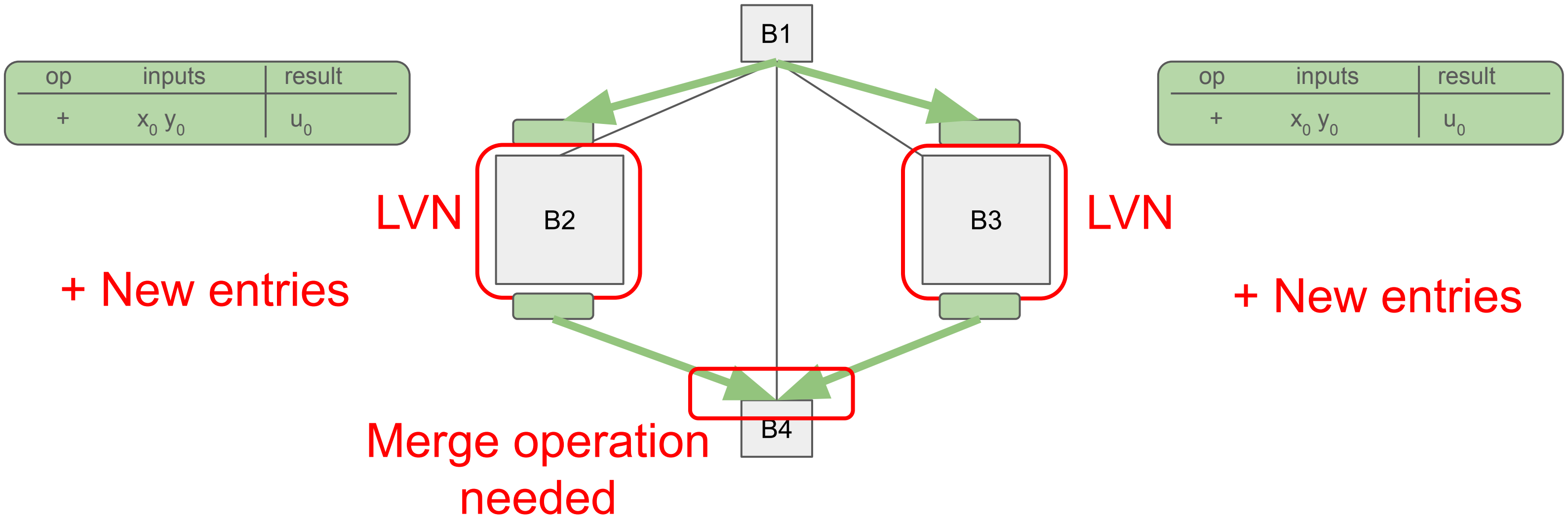


op	inputs	result
+	x_0 y_0	u_0

+ New entries

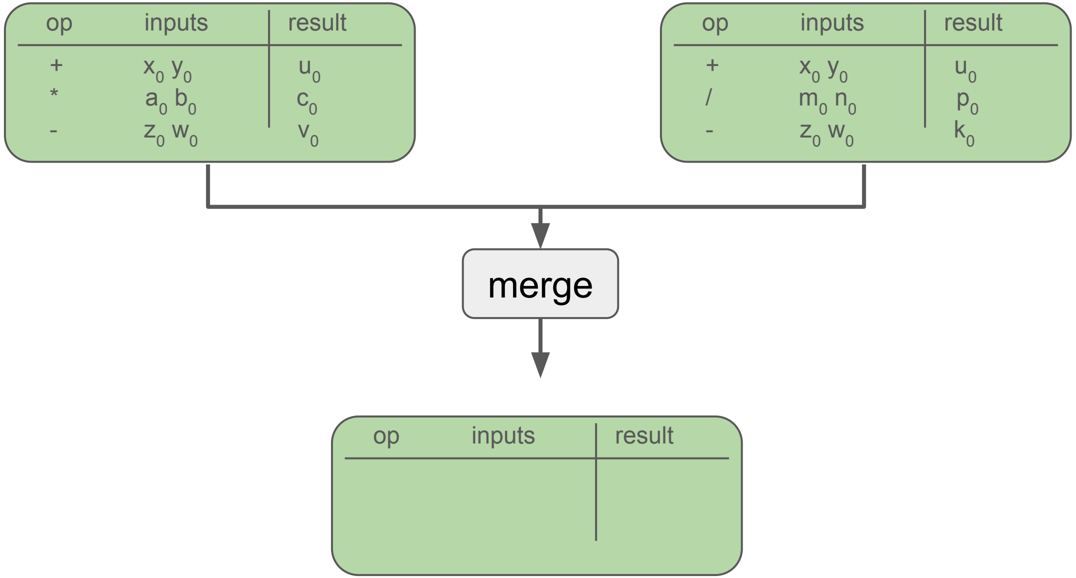
Global Value Numbering using Dominators

- What would happen if we propagated LVN across CF edges?



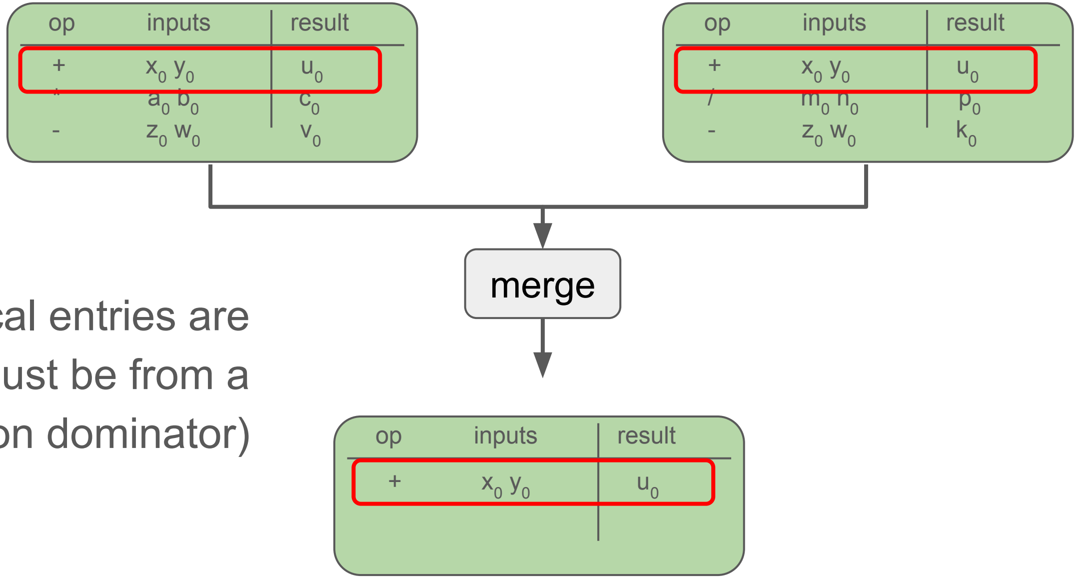
Global Value Numbering using Dominators

- Merging two GVN maps



Global Value Numbering using Dominators

- Merging two GVN maps



1. Identical entries are retained (must be from a common dominator)

Global Value Numbering using Dominators

- Merging two GVN maps

op	inputs	result
+	x_0 y_0	u_0
*	a_0 b_0	c_0
-	z_0 w_0	v_0

op	inputs	result
+	x_0 y_0	u_0
/	m_0 n_0	p_0
-	z_0 w_0	k_0

merge

op	inputs	result
+	x_0 y_0	u_0

2. Mismatched entries are discarded (intersection)

Global Value Numbering using Dominators

- Merging two GVN maps

op	inputs	result
+	x_0 y_0	u_0
*	a_0 b_0	c_0
-	z_0 w_0	v_0

op	inputs	result
+	x_0 y_0	u_0
/	m_0 n_0	p_0
-	z_0 w_0	k_0

merge

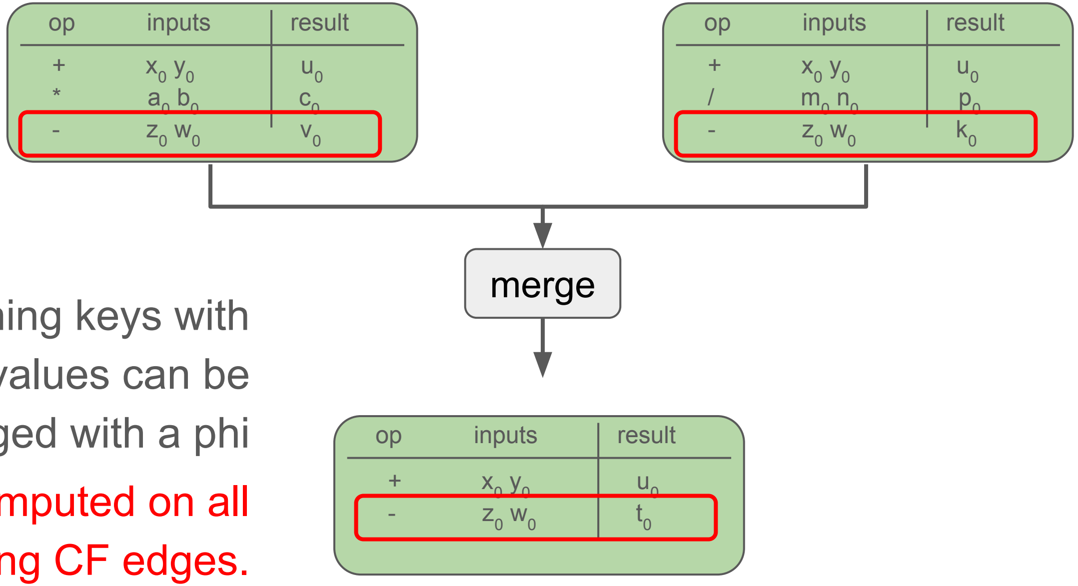
3. Matching keys with mismatched values can be merged with a phi

op	inputs	result
+	x_0 y_0	u_0
-	z_0 w_0	t_0

insert $t_0 \leftarrow \phi(v_0, k_0)$

Global Value Numbering using Dominators

- Merging two GVN maps



3. Matching keys with mismatched values can be merged with a phi

…but must be computed on all incoming CF edges.

insert $\mathbf{t_0 \leftarrow \Phi(v_0,\ k_0)}$

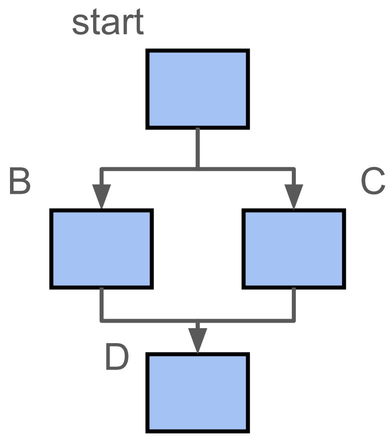
Solving Dataflow Efficiently: Forward Worklist

- GVN is another instance of (forward) global dataflow analysis.
- Can be formulated in the IN, OUT, GEN, KILL framework.
- What's the most efficient solving strategy?
  - Visit in topological order, start to end.
  - Don't process a block until its predecessors have been processed.
  - Approximate or iterate to fixpoint for loops.
- Typical compilers use a forward worklist algorithm.

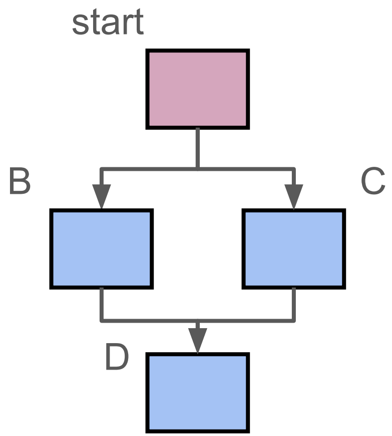
Solving Dataflow Efficiently: Forward Worklist

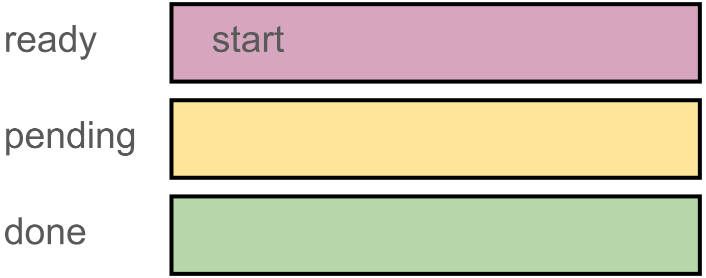
- Use two worklists: `ready` and `pending`
- Put start block into `ready`
- Until all blocks are processed:
    1. Select a block from the `ready` queue and process it.
    2. If no block is ready, select one from the `pending` queue and process it.
    3. Update successor block(s) states.
        - A block becomes ready when all its predecessors are finished.
        - Otherwise, it is pending.
- How do we know this will terminate?

Solving Dataflow Efficiently: Forward Worklist

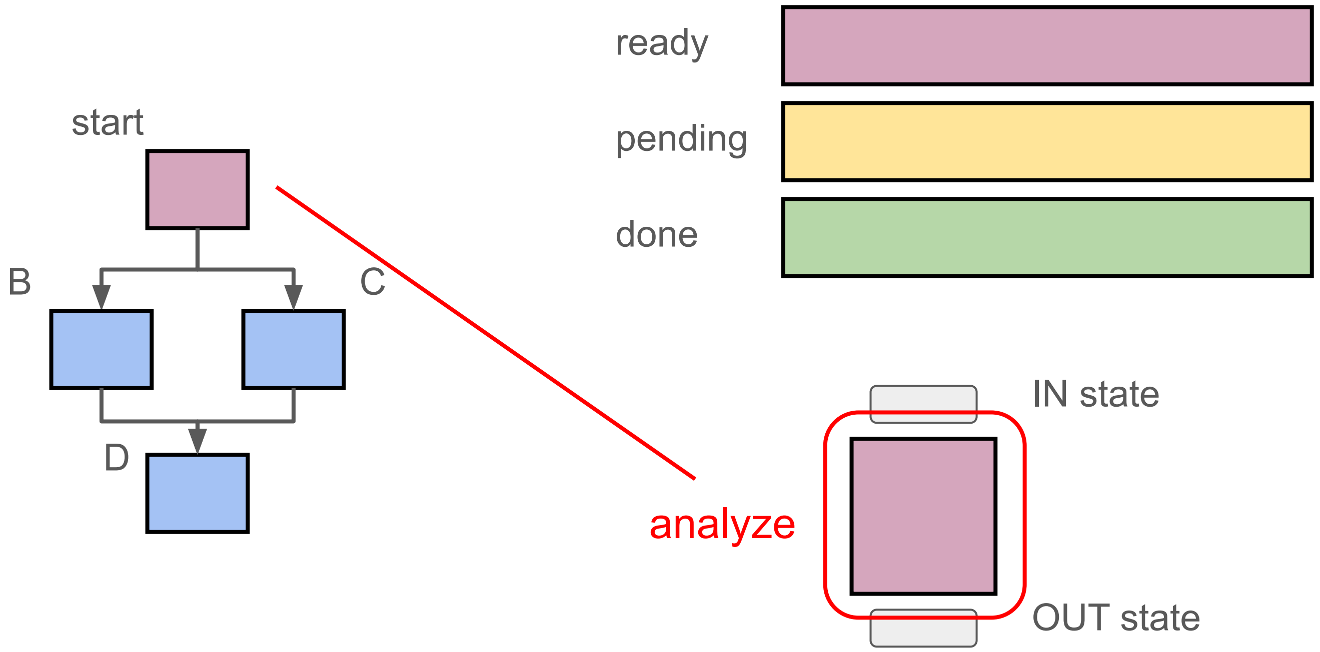


Solving Dataflow Efficiently: Forward Worklist

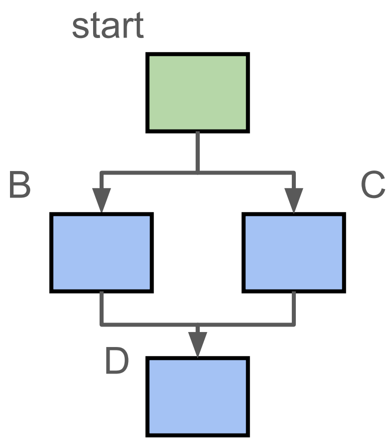


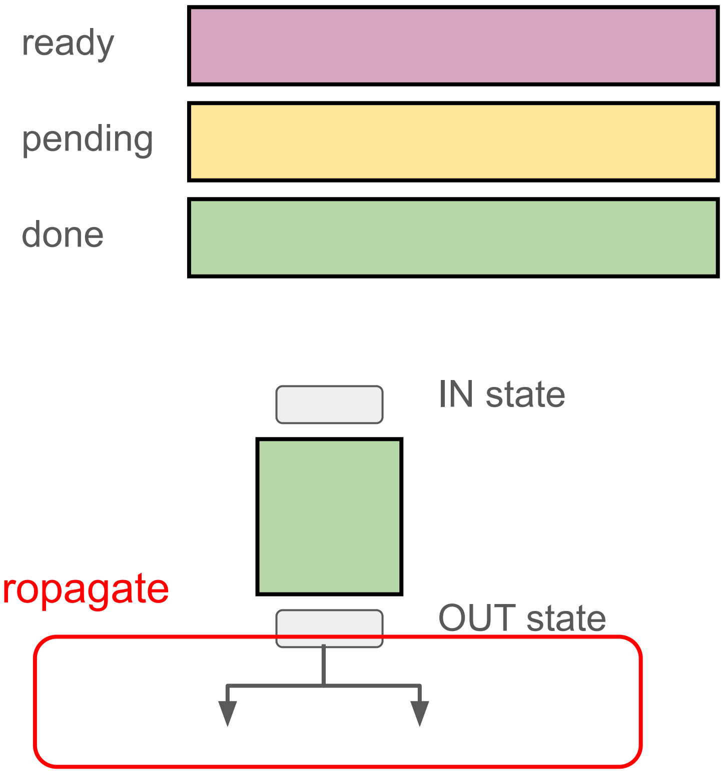


Solving Dataflow Efficiently: Forward Worklist

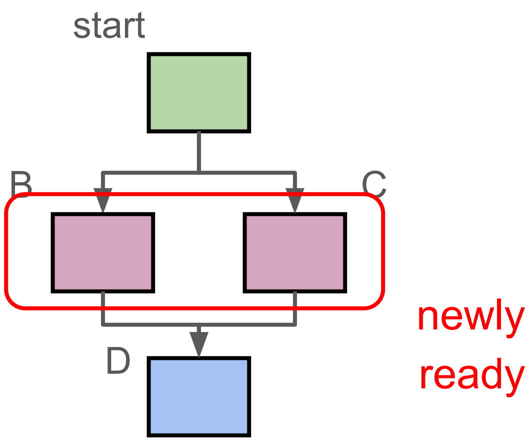


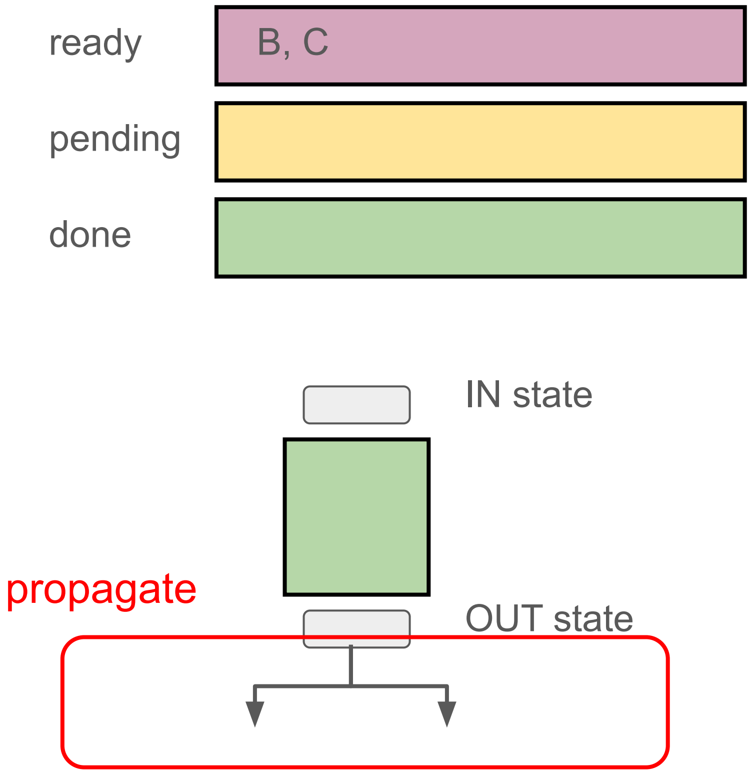
Solving Dataflow Efficiently: Forward Worklist



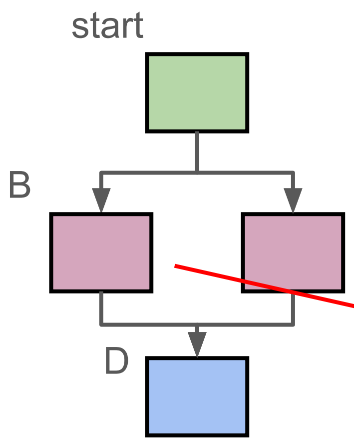


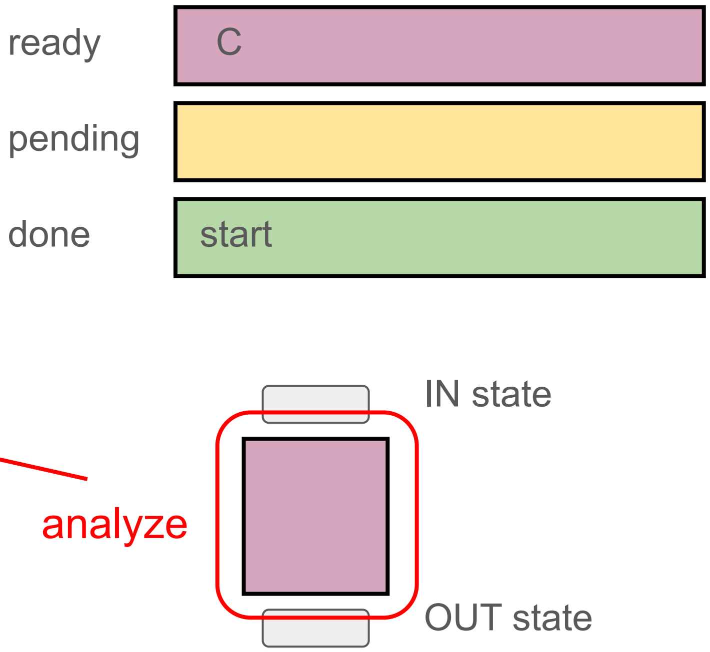
Solving Dataflow Efficiently: Forward Worklist



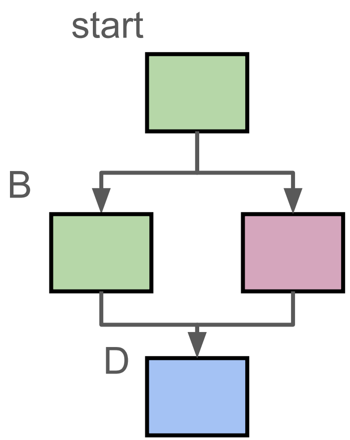


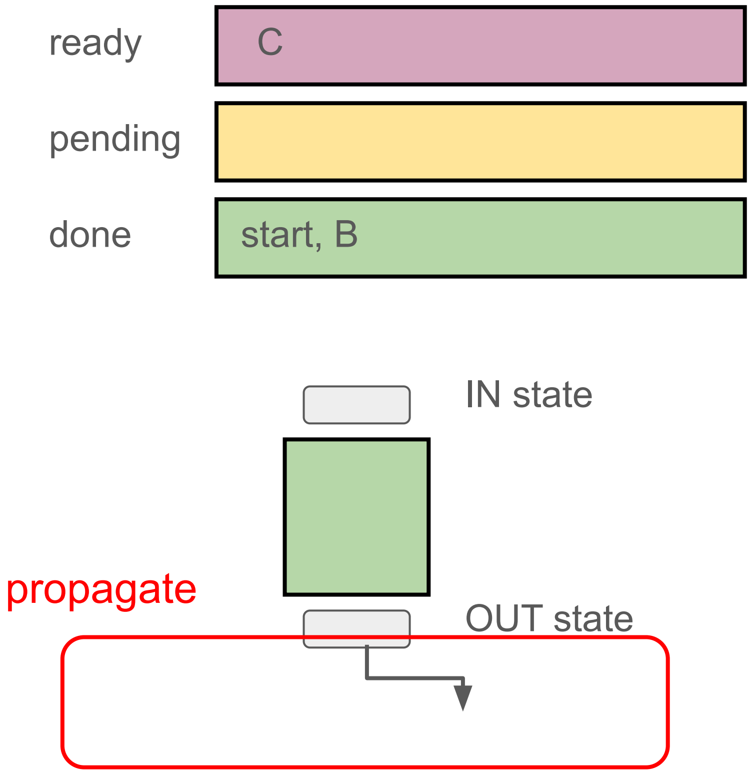
Solving Dataflow Efficiently: Forward Worklist



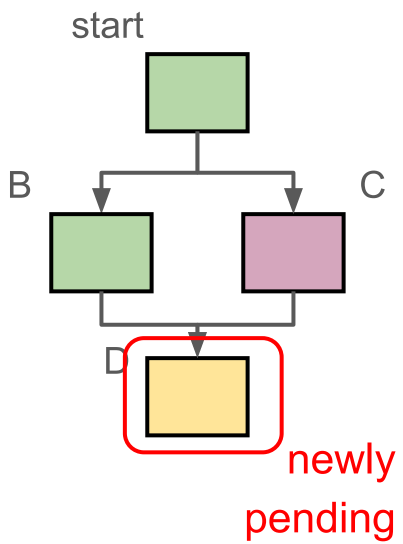


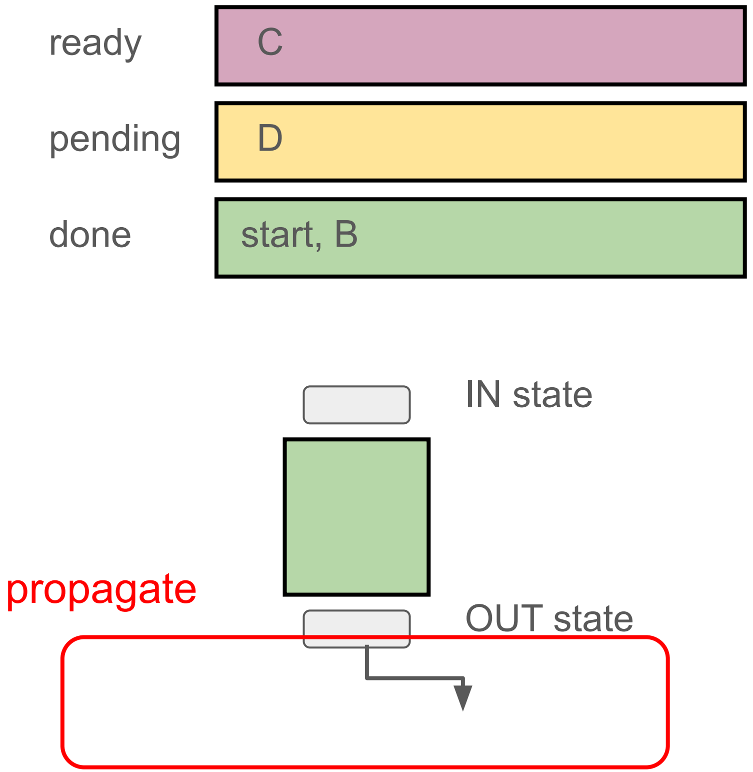
Solving Dataflow Efficiently: Forward Worklist



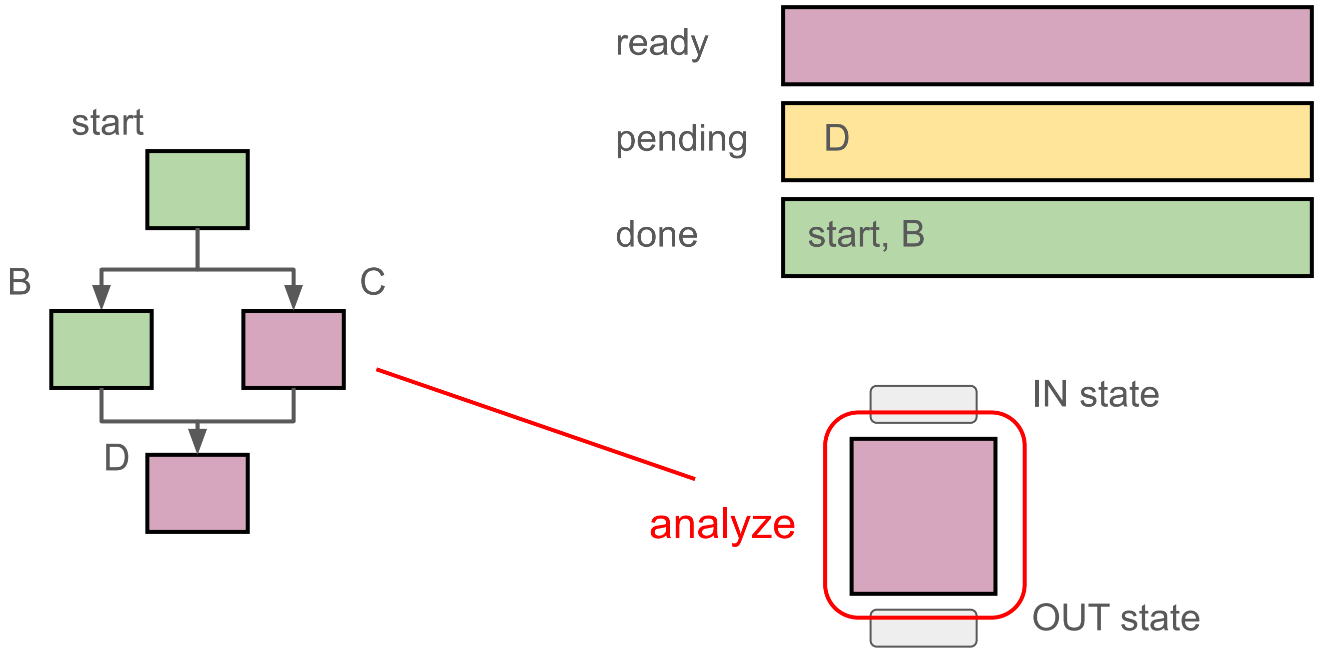


Solving Dataflow Efficiently: Forward Worklist

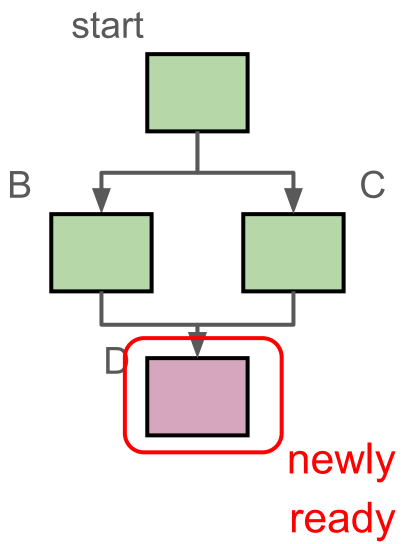


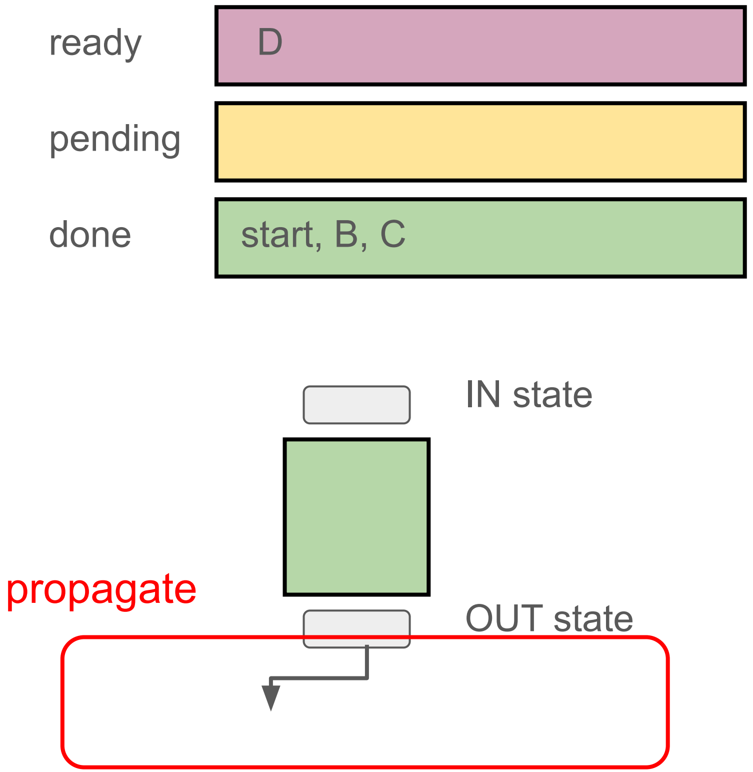


Solving Dataflow Efficiently: Forward Worklist

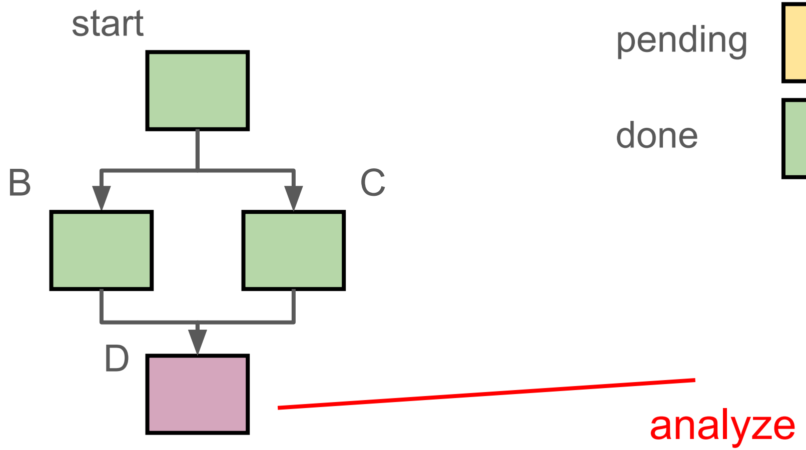


Solving Dataflow Efficiently: Forward Worklist

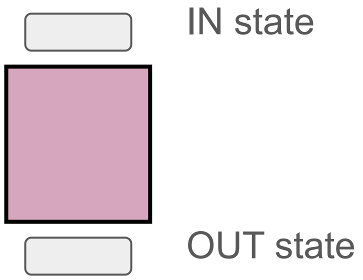




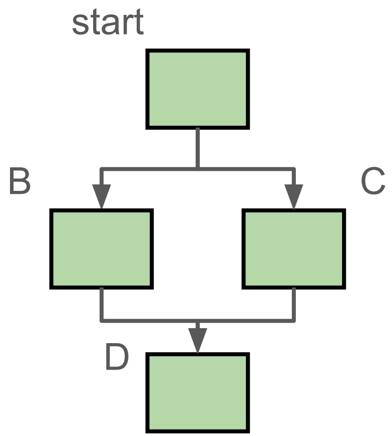
Solving Dataflow Efficiently: Forward Worklist

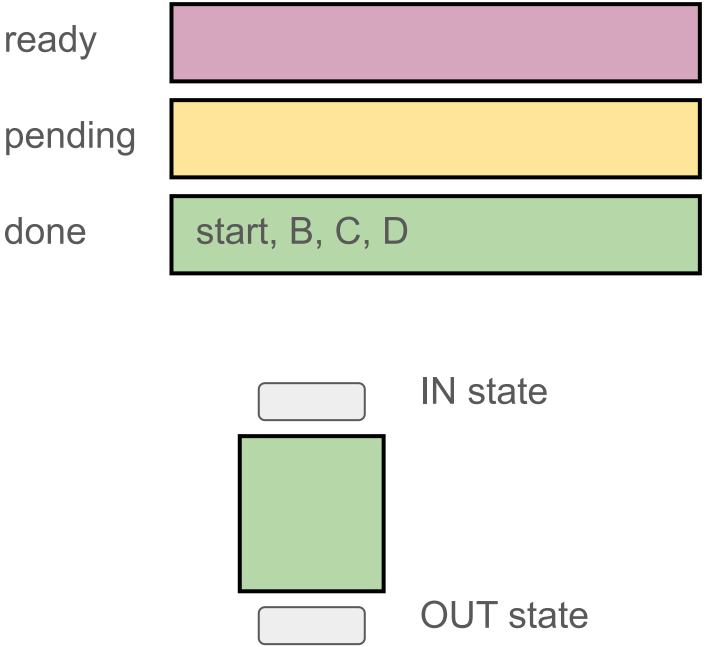


ready	D
pending	
done	start, B, C

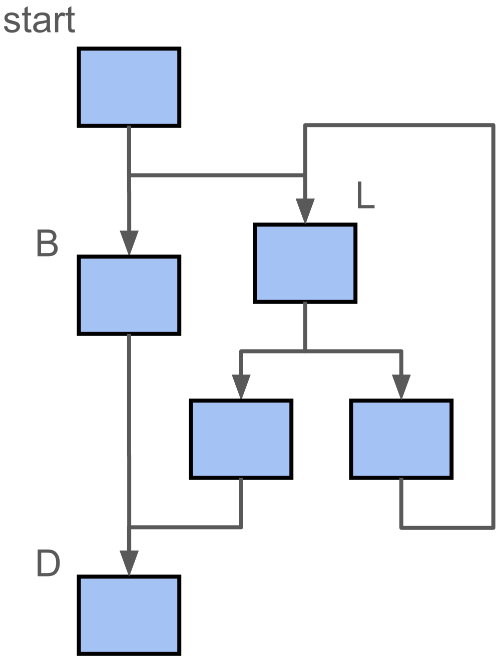


Solving Dataflow Efficiently: Forward Worklist

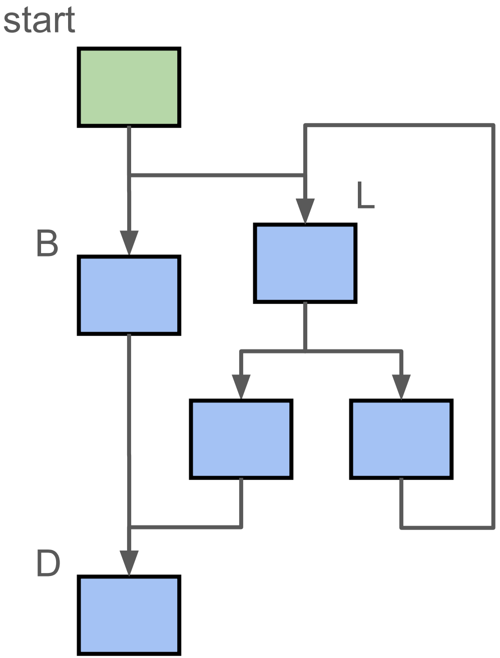




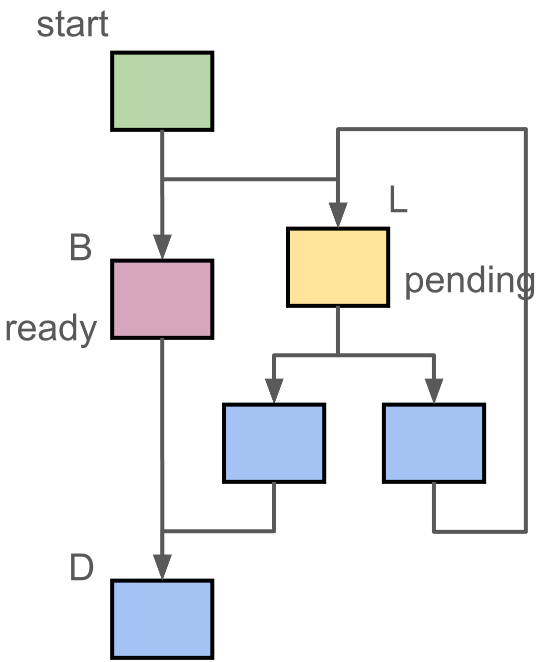
Solving Dataflow Efficiently: What about loops?



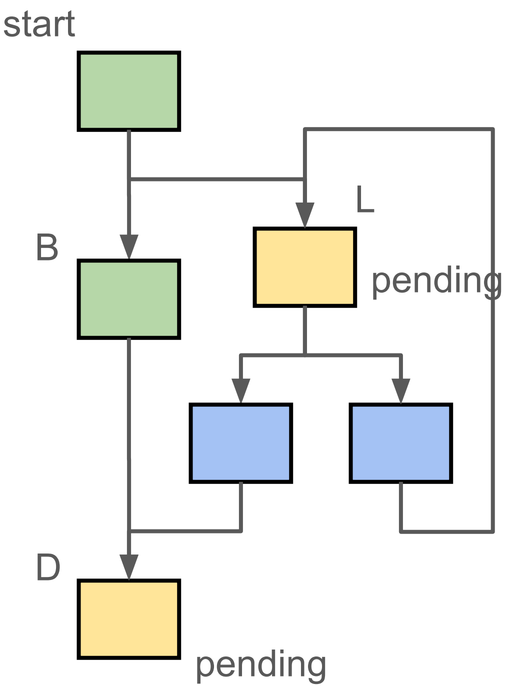
Solving Dataflow Efficiently: What about loops?



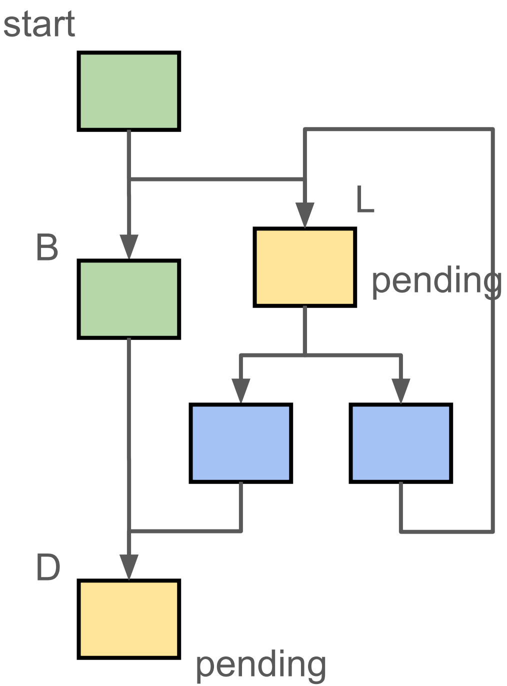
Solving Dataflow Efficiently: What about loops?



Solving Dataflow Efficiently: What about loops?

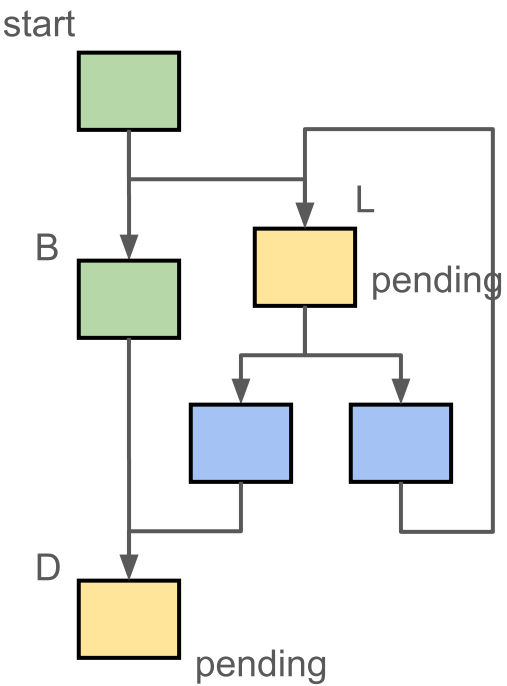


Solving Dataflow Efficiently: What about loops?



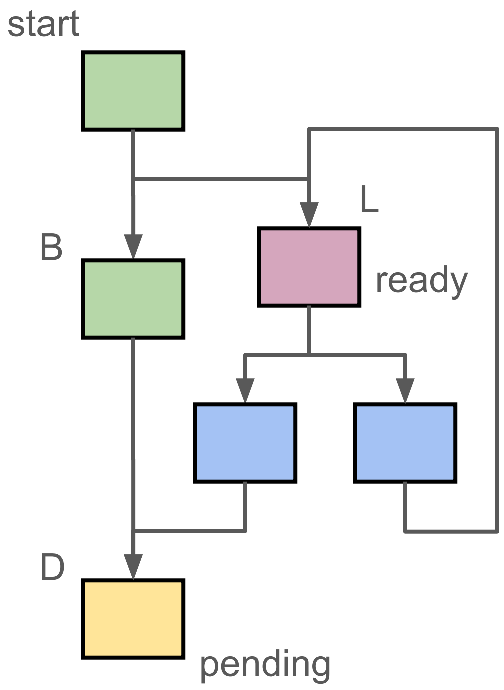
We're stuck!?

Solving Dataflow Efficiently: What about loops?



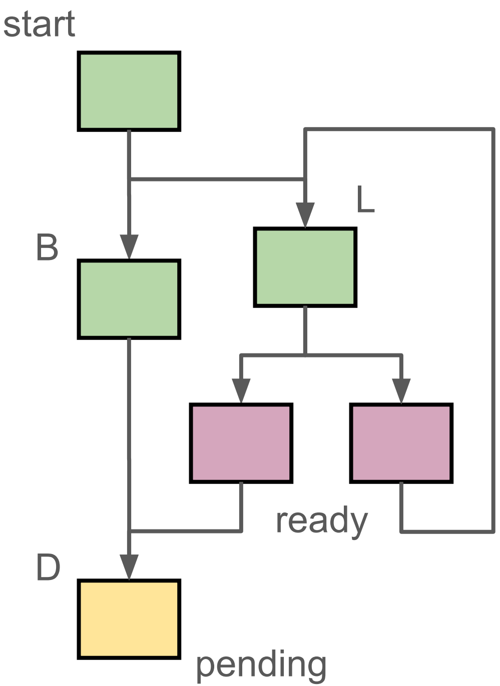
Break cycle by processing loop header with incomplete information.

Solving Dataflow Efficiently: What about loops?



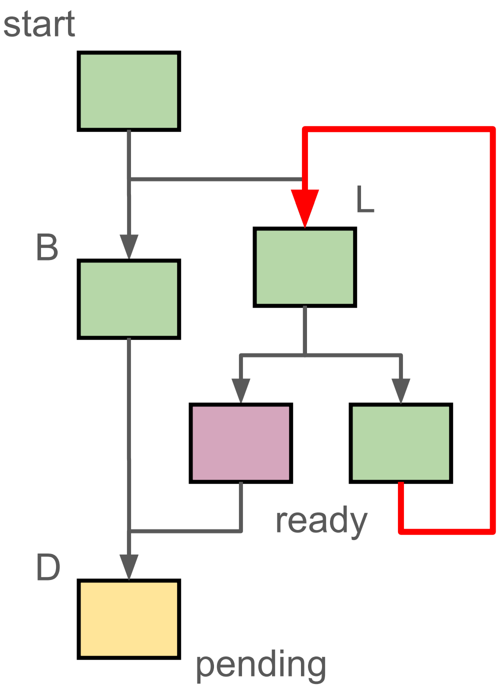
Break cycle by processing loop header with incomplete information.

Solving Dataflow Efficiently: What about loops?



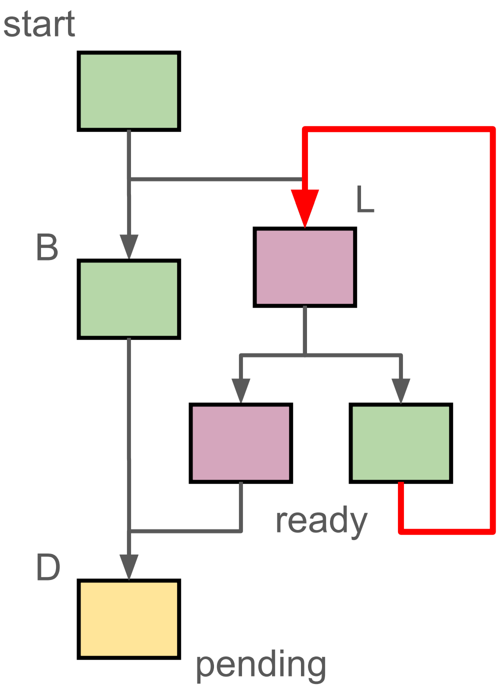
Dominated nodes can proceed.

Solving Dataflow Efficiently: What about loops?



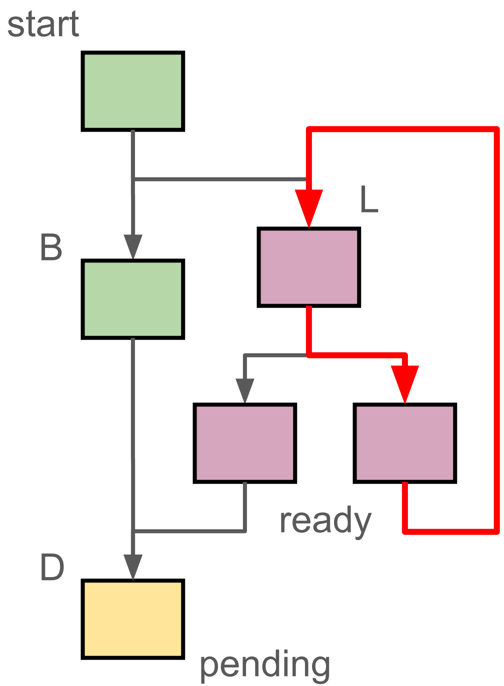
New information on backedge may lead to reanalyzing loop header.

Solving Dataflow Efficiently: What about loops?



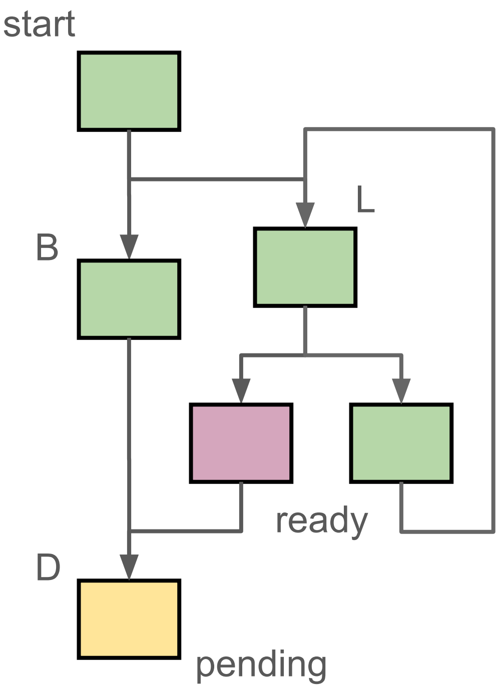
New information on backedge may lead to reanalyzing loop header.

Solving Dataflow Efficiently: What about loops?



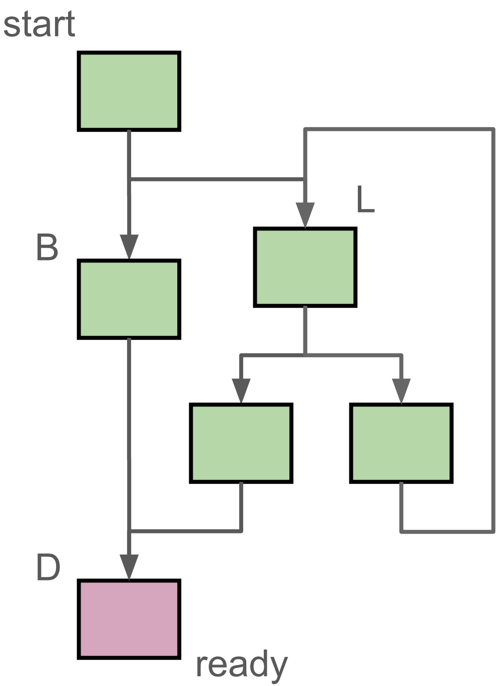
New information can cause dataflow analysis to iterate repeatedly on the loop body.

Solving Dataflow Efficiently: What about loops?



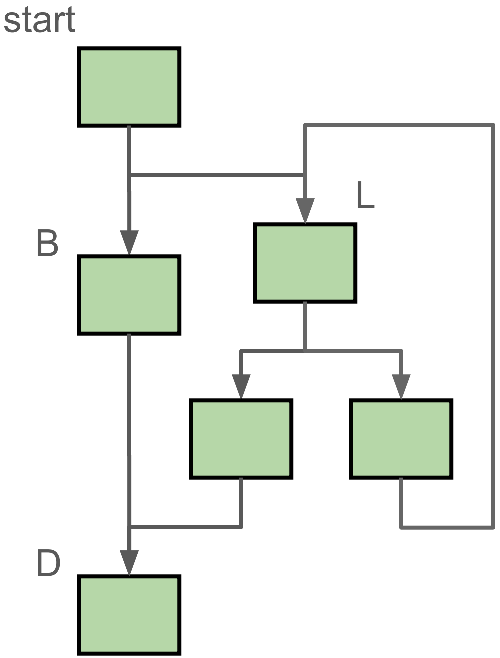
When a fixpoint is reached, dataflow information stabilizes.

Solving Dataflow Efficiently: What about loops?



Eventually all dataflow information stabilizes if equations are properly *monotonic*.

Solving Dataflow Efficiently: What about loops?



Eventually all dataflow information stabilizes if equations are properly *monotonic*.

Inlining

- Why call a function when you can *inline* it?
- Inlining substitutes the body of a function in place of a call
  - Also called “inline substitution” or “procedure integration”

Inlining

- Why call a function when you can *inline* it?
- Inlining substitutes the body of a function in place of a call
  - Also called “inline substitution” or “procedure integration”
- Why?
  - Eliminate the direct cost of a call
    - Spills/reloads across call
    - Argument shuffling
    - Jump + push return address
    - Stack frame setup/teardown
  - Optimize the procedure’s code in the caller’s context
  - Optimizations can be less conservative across a call

Inlining Example

```
double exec(double z0) {
   x0 ← 0.0d;
   y0 ← 1.0d;
   r0 ← fma(x0, y0, z0);
   return r0;
}
```

```
double fma(double a, double b, double c) {
   t0 ← b * c;
   t1 ← a + t0;
   return t1;
}
```

Inlining Example

Argument setup

Body

Return value(s)

```
double exec(double z₀) {
  x₀ ← 0.0d;
  y₀ ← 1.0d;
  a ← x₀;
  b ← y₀;
  c ← z₀;
  t₀ ← b * c;
  t₁ ← a + t₀;
  r₀ ← t₁;
  return r₀;
}
```

```
double fma(double a, double b, double c) {
  t₀ ← b * c;
  t₁ ← a + t₀;
  return t₁;
}
```

Inlining Example

Argument setup

Body

Return value(s)

```
double exec(double z₀) {
  x₀ ← 0.0d;
  y₀ ← 1.0d;
  a ← x₀;
  b ← y₀;
  c ← z₀;
  t₀ ← b * c;
  t₁ ← a + t₀;
  r₀ ← t₁;
  return r₀;
}
```

```
double fma(double a, double b, double c) {
  t₀ ← b * c;
  t₁ ← a + t₀;
  return t₁;
}
```

Inlining Example

Argument setup

Body

Return value(s)

```
double exec(double z₀) {
  x₀ ← 0.0d;
  y₀ ← 1.0d;
  a ← x₀;
  b ← y₀;
  c ← z₀;
  t₀ ← b * c;
  t₁ ← a + t₀;
  r₀ ← t₁;
  return r₀;
}
```

```
double fma(double a, double b, double c) {
  t₀ ← b * c;
  t₁ ← a + t₀;
  return t₁;
}
```

Inlining Example - Copy Propagation

```
double exec(double z₀) {
  x₀ ← 0.0d;
  y₀ ← 1.0d;
  a ← x₀;
  b ← y₀;
  c ← z₀;
  t₀ ← y₀ * z₀;
  t₁ ← x₀ + t₀;
  r₀ ← t₁;
  return t₁;
}
```

```
double fma(double a, double b, double c) {
  t₀ ← b * c;
  t₁ ← a + t₀;
  return t₁;
}
```

Inlining Example - Copy/Constant Propagation

```
double exec(double z0) {
  ~~x0 ← 0.0d;~~
  ~~y0 ← 1.0d;~~
  ~~a ← x0;~~
  ~~b ← y0;~~
  ~~c ← z0;~~
  t0 ← 1.0d * z0;
  t1 ← 0.0d + t0;
  ~~r0 ← t1;~~
  return t1;
}
```

```
double fma(double a, double b, double c) {
  t0 ← b * c;
  t1 ← a + t0;
  return t1;
}
```

Inlining Example - Strength Reduction

```
double exec(double z0) {
  x0 ← 0.0d;
  y0 ← 1.0d;
  a ← x0;
  b ← y0;
  c ← z0;
  t0 ← 1.0d * z0;
  t1 ← 0.0d + t0;
  r0 ← t1;
  return t1;
}
```

Multiplicative and additive identities

```
double fma(double a, double b, double c) {
  t0 ← b * c;
  t1 ← a + t0;
  return t1;
}
```

Inlining Example - Copy Propagation

```
double exec(double z0) {
  x0 ← 0.0d;
  y0 ← 1.0d;
  a ← x0;
  b ← y0;
  c ← z0;
  t0 ← 1.0d * z0;
  t1 ← 0.0d + t0;
  r0 ← t1;
  return z0;
}
```

```
double fma(double a, double b, double c) {
  t0 ← b * c;
  t1 ← a + t0;
  return t1;
}
```

Inlining Example - Final Result

```
double exec(double z₀) {
    return z₀;
}
```

```
double fma(double a, double b, double c) {
   t₀ ← b * c;
   t₁ ← a + t₀;
   return t₁;
}
```

Inlining IR Transforms

- After the decision to inline, it's just graph surgery

1. Split the basic block in which the call occurs
2. Wire in a copy of the inlined function's CFG
3. Replace references to parameter nodes with arguments
4. Turn returns into gotos to the return block

   (Introduce phis if multiple returns)

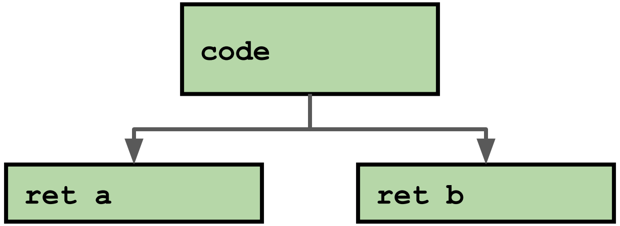
5. Replace temps defined by the call by phis or return values
6. Optimize new blocks and clean up

Inlining IR Transforms

B1

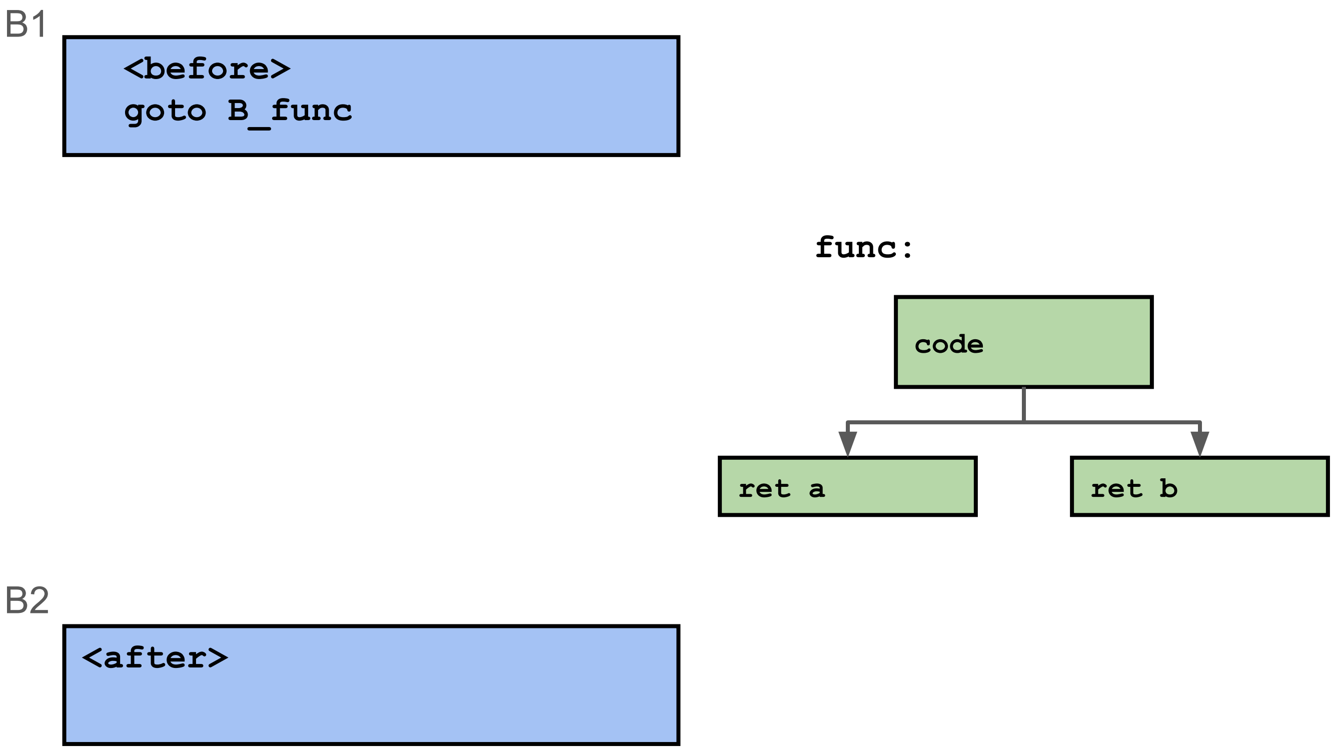
```
<before>
r ← func(x, y, z);
<after>
```

func:



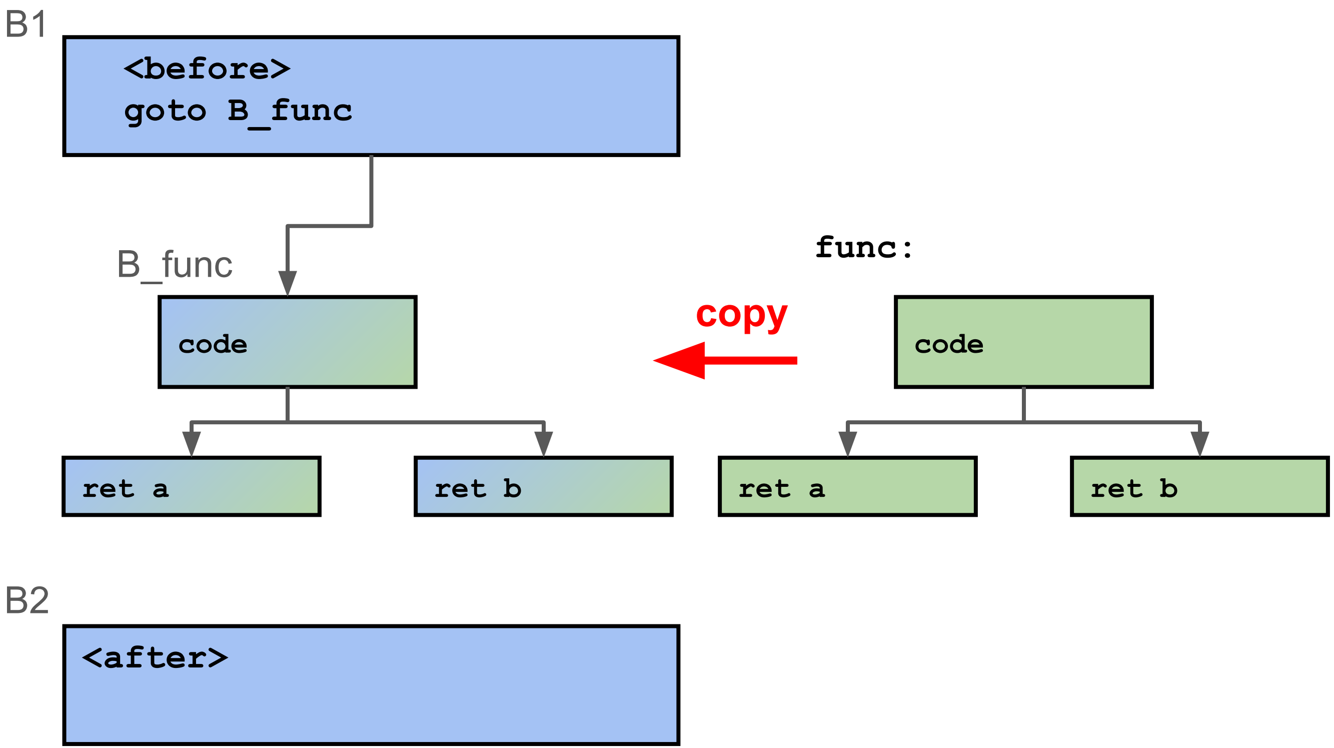
Inlining IR Transforms

1. Split the basic block in which the call occurs.



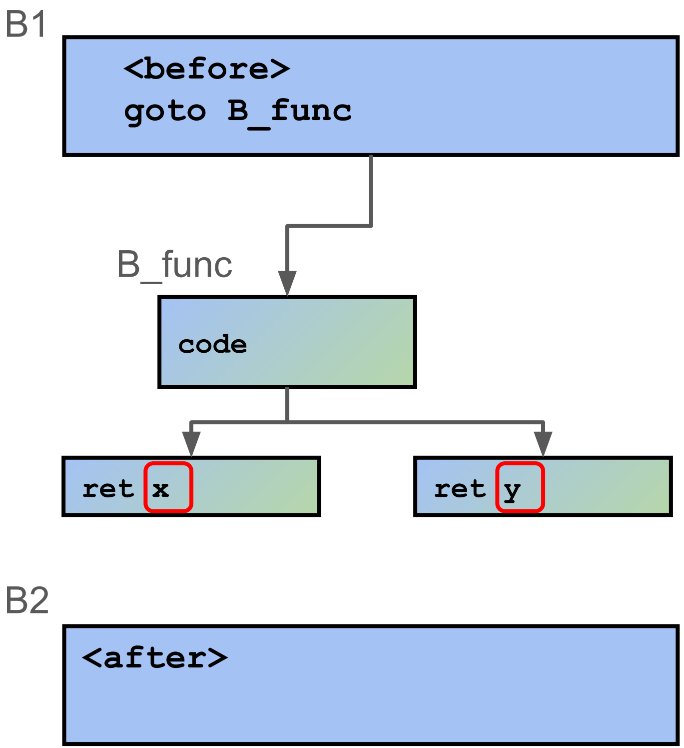
Inlining IR Transforms

2. Wire in a **copy** of the functions' body.



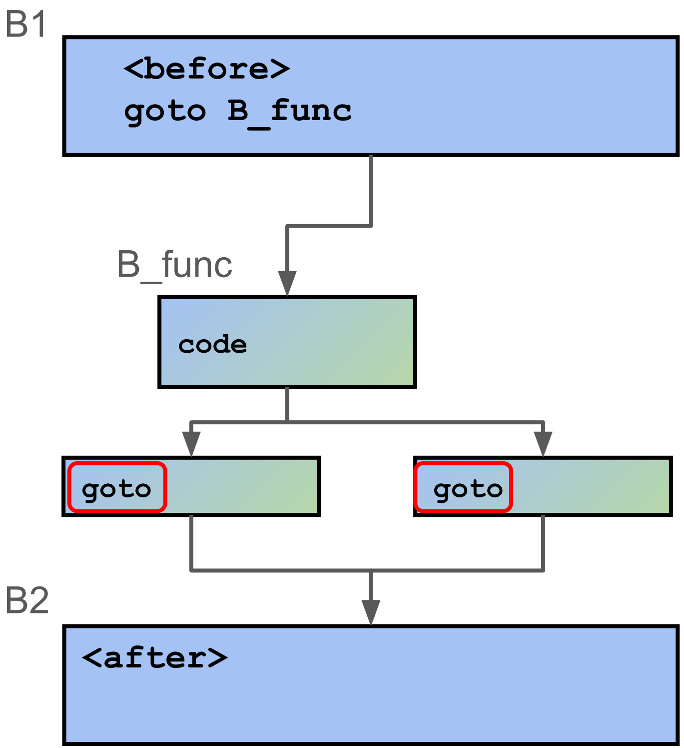
Inlining IR Transforms

3. Replace references to parameters with references to args.



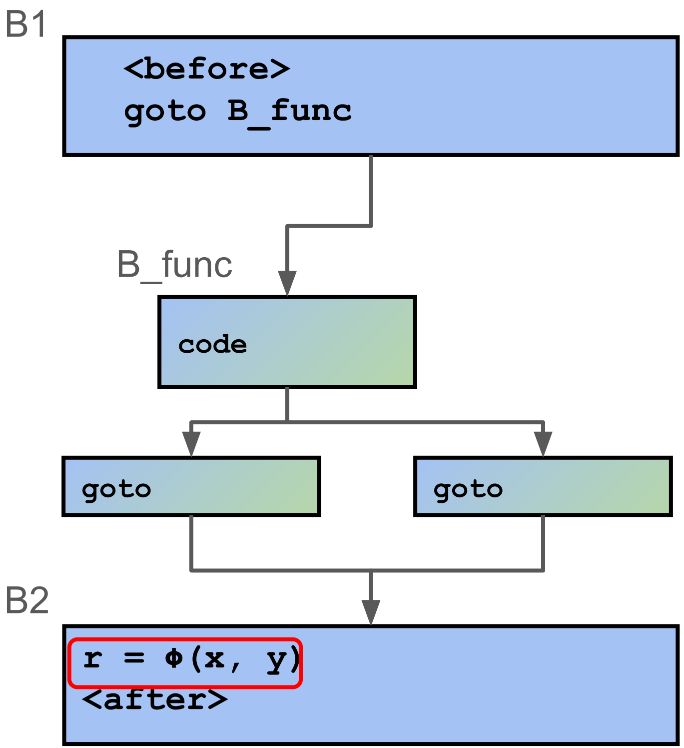
Inlining IR Transforms

4. Turn returns into goto to the return block.



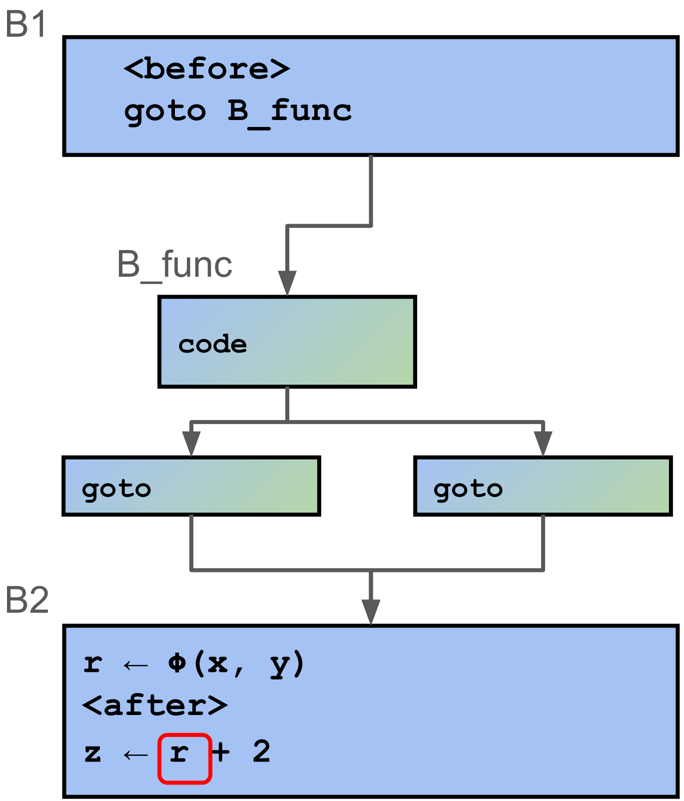
Inlining IR Transforms

4. Turn returns into goto to the return block (introducing phis if necessary).



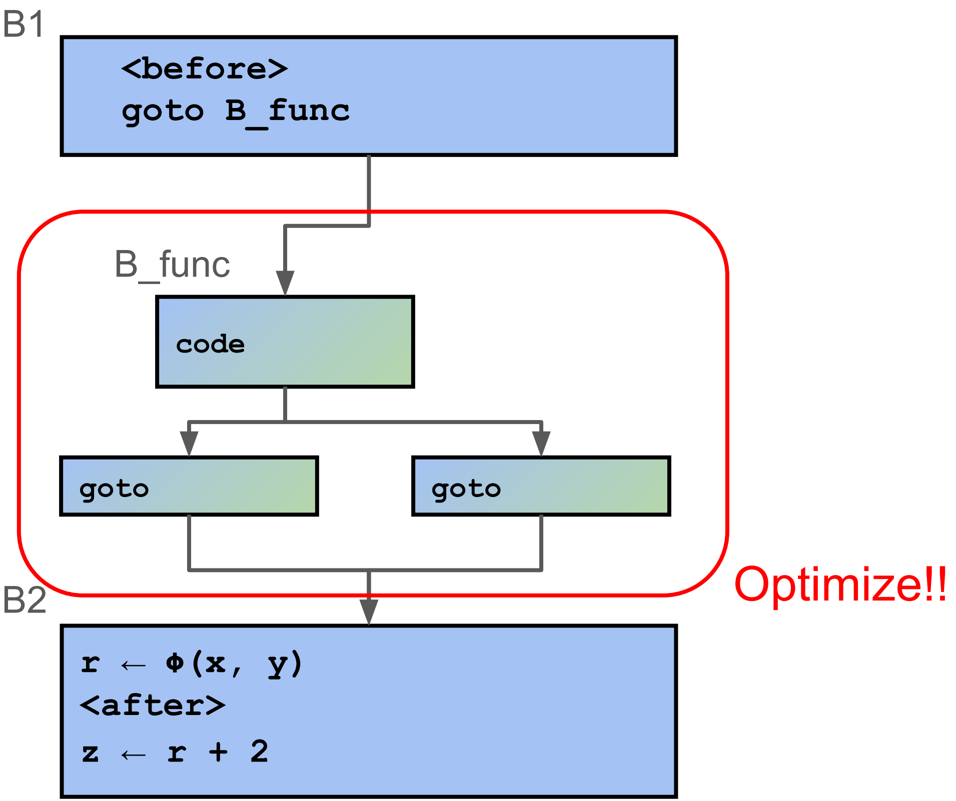
Inlining IR Transforms

5. Replace temps defined by the call with the new return temps.



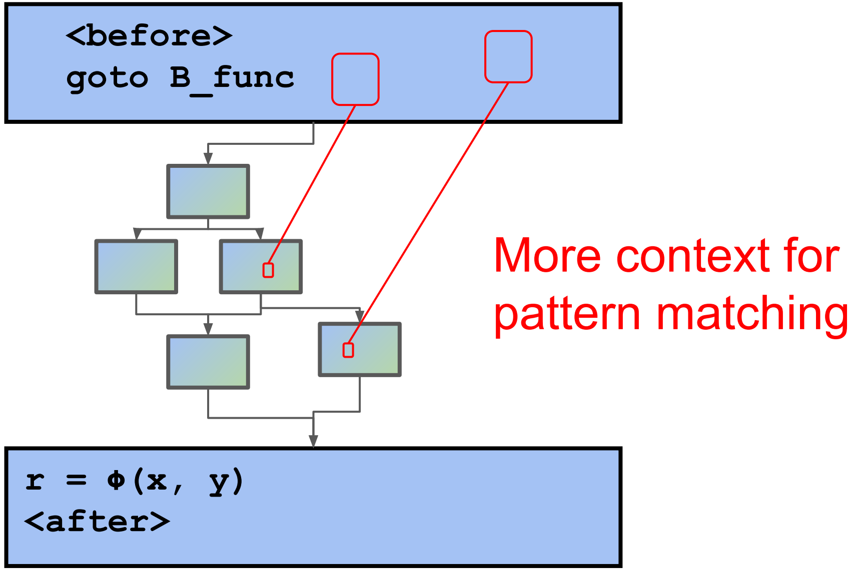
Inlining IR Transforms

6. Optimize the new blocks and clean up.



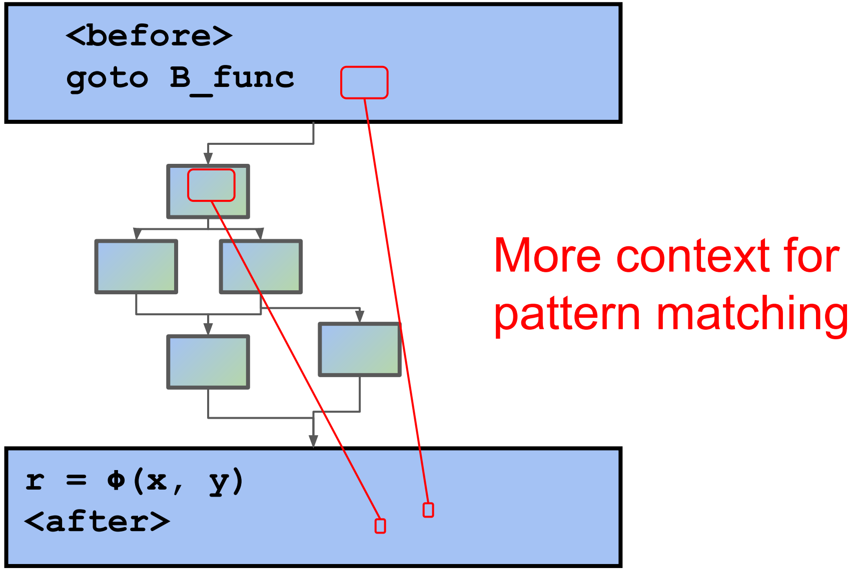
Cascading Optimizations: Strength Reduction

- We've already seen an example of strength reduction being enabled to by inlining.
- Before inlining, the compiler could not see beyond params.



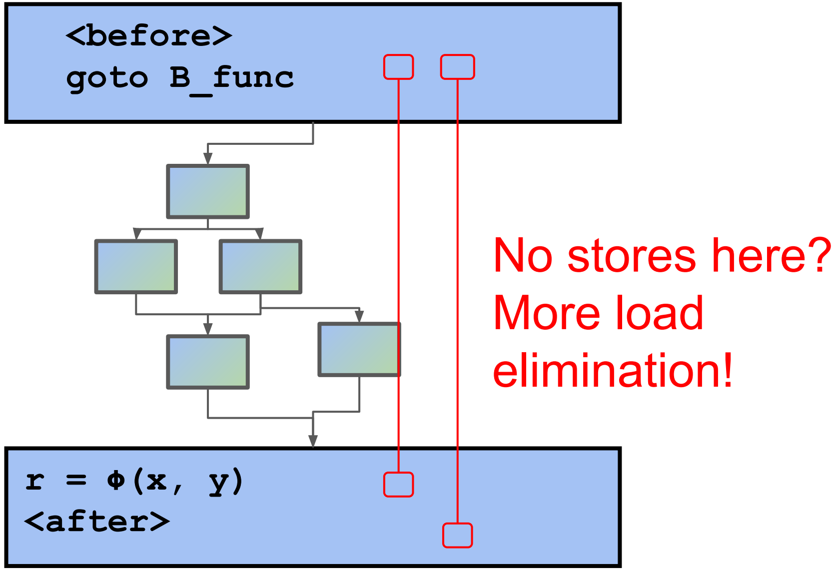
Cascading Optimizations: Strength Reduction

- We’ve already seen an example of strength reduction being enabled to by inlining.
- Before inlining, the compiler could not see beyond params.



Cascading Optimizations: Load Elimination

- Recall from load elimination, stores invalidate entries
- Before inlining, the compiler could not see into calls



Static Inlining Heuristics

- Predict hot callsite
- One target function
- Small function
- Leaf function
- Anticipated optimizations
  - Constant folding, particularly branch folding
  - Strength reduction
  - Load/store elimination
  - Alias analysis
  - Improved types

Dynamic Inlining Heuristics

- Hot call site, discovered through profiling
- Recorded one target function
  - Typically done in JavaScript VMs
  - Can optimize method calls
- Recorded one or more receiver types
  - Typically done in JavaScript and Java VMs
  - Can optimize property accesses, field accesses, and method calls

Downsides to Inlining?

Downsides to Inlining?

- Increased code size (exponential?)
  - I-cache pressure
  - Binary size
- Increased register pressure
- Slower compile time with more memory consumption
- Diminishing returns