



Value Forwarding in MLIR

Control-Flow Interfaces and Canonicalizations You Get for Free

Matthias Springer (NVIDIA)

LLVM Developers' Meeting 2026 – Quick Talk – October 27, 2026



Forwarded Values vs. Produced/Consumed Values

- **Forwarded Operand:** An operand that is passed through by the operation and forwarded to a block argument and/or operation result.
- **Forwarded Value:** A block argument or operation result that is passed through by the operation from a forwarded operand.
- **Consumed Operand:** An operand that is not a forwarded operand.
- **Produced Value:** A value that is not a forwarded value. A value that is “made up” by the operation.
- This talk: Control Flow Modeling + Data Flow Modeling $\Rightarrow$ Canonicalization



Forwarded Values / Operands

forwarded values

```
%r0, %r1 = scf.for %iv = %lb to %ub step %step  
  iter_args(%arg0 = %0, %arg1 = %1) -> (i32, i32) {  
    scf.yield %arg0, %arg1 : i32, i32  
  }
```

forwarded operands



Produced Values / Consumed Operands

produced value consumed operands

```
%r0, %r1 = scf.for [%iv] = [%lb to %ub step %step]  
  iter_args(%arg0 = %0, %arg1 = %1) -> (i32, i32) {  
    scf.yield %arg0, %arg1 : i32, i32  
  }
```

The diagram illustrates the produced values and consumed operands for the `scf.for` operation. The `%iv` operand is highlighted with a blue box and labeled "produced value". The `%lb`, `%ub`, and `%step` operands are highlighted with green boxes and labeled "consumed operands".



!builtin.token: Non-forwardable Values ([RFC](#))

```
%0 = scf.loop token(%token1) -> i32 {  
  %1, %2 = scf.loop token(%token2) -> (f32, f32) {  
    scf.if ... {  
      scf.break [%token1] %a : i32  
    }  
    scf.if ... {  
      scf.break [%token2] %b, %c : f32, f32  
    }  
  }  
}
```

Must never appear
in as a forwarded
operand. Explicit
opt-in via ODS
required!

RegionBranchOpInterface



Some Operations that Implement the Interface

- `scf.execute_region`
- `scf.for`
- `scf.if`
- `scf.index_switch`
- `scf.while`
- `scf.forall` (partly)
- `scf.parallel` (partly)
- `scf.reduce` (partly)
- Your custom op?



Terms and API

- Interface comes as a pair:
RegionBranchOpInterface and RegionBranchTerminatorOpInterface

- Control flow modeling 

```
graph LR; A[Region Branch Point] --> B[Region Successor]
```

 - Region branch point: source of a control edge (can be “parent”)
 - Successor region: destination of a control flow edge (can be “parent”)
 - Query edges with `getSuccessorRegions(RegionBranchPoint)`

- Data flow modeling
 - Forwarded Operand = Successor Operand. Query with `getSuccessorOperands(RegionBranchPoint)`.
 - Forwarded Value = Successor Input. Query with `getSuccessorInputs(RegionSuccessor)`.



Canonicalization: Remove Uses of Forwarded Values

```
%r0, %r1 = scf.for ... iter_args(%arg0 = %0, %arg1 = %1) -> ... {  
  // ...  
  scf.yield %arg1, %arg1 : ...  
}  
  
use(%r0, %r1)
```



Canonicalization: Remove Uses of Forwarded Values

```
%r0, %r1 = scf.for ... iter_args(%arg0 = %0, %arg1 = %1) -> ... {  
  // ...  
  scf.yield %arg1, %arg1 : ...  
}  
  
use(%r0, %r1)
```



Canonicalization: Remove Uses of Forwarded Values

Possible values: {%0, %1}

Possible values: {%0, %1}

Possible values: {%1}

```
%r0, %r1 = scf.for ... iter_args(%arg0 = %0, %arg1 = %1) -> ... {
```

/, Possible values: {%1}

```
    scf.yield %arg1, %arg1 : ...
```

```
}
```

```
use(%r0, %r1)
```

Look for forwarded values with exactly one forwarded operand.



Canonicalization: Remove Uses of Forwarded Values

Possible values: {%0, %1}

Possible values: {%0, %1}

Possible values: {%1}

```
%r0, %r1 = scf.for ... iter_args(%arg0 = %0, %arg1 = %1) -> ... {
```

```
  / Possible values: {%1}
```

```
  scf.yield %1, %1 : ...
```

```
}
```

```
use(%r0, %1)
```

Look for forwarded values with exactly one forwarded operand.



Canonicalization: Remove Dead Forwarded Values

```
%r0, %r1 = scf.for ... iter_args(%arg0 = %0, %arg1 = %1) -> ... {  
  // ...  
  scf.yield %1, %1 : ...  
}  
  
use(%r0, %1)
```

Look for forwarded values with exactly one forwarded operand.



Canonicalization: Remove Dead Forwarded Values

```
%r0, %r1 = scf.for ... iter_args(%arg0 = %0, %arg1 = %1) -> ... {  
  // ...  
  scf.yield %1, %1 : ...  
}  
  
use(%r0, %1)
```

Dead

Dead



Canonicalization: Remove Dead Forwarded Values

```
%r0 = scf.for ... iter_args(%arg0 = %0) -> ... {  
  // ...  
  scf.yield %1 : ...  
}  
  
use(%r0, %1)
```



Canonicalization: Remove Uses of Equivalent Forwarded Values

```
%r0, %r1 = scf.for ... iter_args(%arg0 = %0, %arg1 = %0) -> ... {  
    // ...  
    use(%arg0, %arg1)  
    scf.yield %x, %x : ...  
}
```

```
use(%r0, %r1)
```

Look for forwarded values with same forwarded operands from each region branch point.



Canonicalization: Remove Uses of Equivalent Forwarded Values

Possible values: [%0, %x]

Possible values: [%0, %x]

Possible values: [%0, %x]

```
%r0, %r1 = scf.for ... iter_args(%arg0 = %0, %arg1 = %0) -> ... {  
  // Possible values: [%0, %x]  
  use(%arg0, %arg1)  
  scf.yield %x, %x : ...  
}
```

use(%r0, %r1)

Look for forwarded values with same forwarded operands from each region branch point.



Canonicalization: Remove Uses of Equivalent Forwarded Values

Possible values: [%0, %x]

Possible values: [%0, %x]

Possible values: [%0, %x]

```
%r0, %r1 = scf.for ... iter_args(%arg0 = %0, %arg1 = %0) -> ... {  
  // Possible values: [%0, %x]  
  use(%arg0, %arg0)  
  scf.yield %x, %x : ...  
}
```

```
use(%r0, %r0)
```

Look for forwarded values with same forwarded operands from each region branch point.



Apply Canonicalizations to Your Custom Ops

- Implement the `RegionBranchOpInterface`.
- `populateRegionBranchOpInterfaceCanonicalizationPatterns`

BranchOpInterface



Forwarded Values / Operands

forwarded operands

```
func.func @unstructured_control_flow(%c: i1, %x: i32, %y: i32) {  
  cf.cond_br %c, ^bb1(%x : i32), %bb2(%y : i32)  
  ^bb1(%arg0: i32):  
    ...  
  ^bb2(%arg1: i32):  
    ...  
}
```

forwarded values

CallOpInterface



Forwarded Values / Operands

forwarded value

forwarded operands

```
%r = func.call @callee(%a, %b) : (i32, i32) -> f32
```

```
func.func @callee(%x: i32, %y: i32) -> f32 {
```

```
...
```

```
return %v : f32
```

```
}
```

forwarded values

forwarded operand

RemoveDeadValues



RemoveDeadValues Pass

- Canonicalizations are local rewrites.
- RemoveDeadValues runs a program-wide constant / liveness analysis and can find simplifications that a canonicalization pattern cannot find easily.
- Implementation strategy: RemoveDeadValues sometimes materializes liveness information in the IR (e.g. by replacing uses of dead values with `ub.poison`), then runs canonicalization patterns for final cleanup.

scf.condition %c, %val : i32

Data flow analysis may be able to prove that %c is zero.

Summary and Future Work



Summary and Future Work

- Forwarded / Produced / Consumed Values / Operands is a **common concept throughout MLIR interfaces** and taken into account by many analyses.
- Cleanup desirable: Terms and notation **are not aligned across interfaces**.
 - Example: “forwarded value” vs. “successor input”.
 - Introduce a new inherited op interface? (ValueForwardingOpInterface)
- There is room for **compilation time optimization**.
- Open problem: Operation semantics / verifier **may require the presence of certain forwarded values** (e.g. `scf.for` op results must match `iter_args`).
- Open problem: How to model ops with **parallel semantics**?