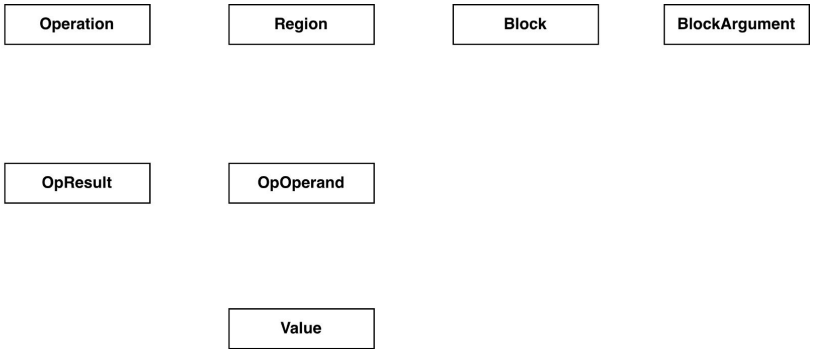
MLIR Fundamentals: Introduction to C++ IR Data Structures and APIs

Matthias Springer (NVIDIA)

ACM Europe School on MLIR 2026 – August 11, 2026

Warmup Quiz

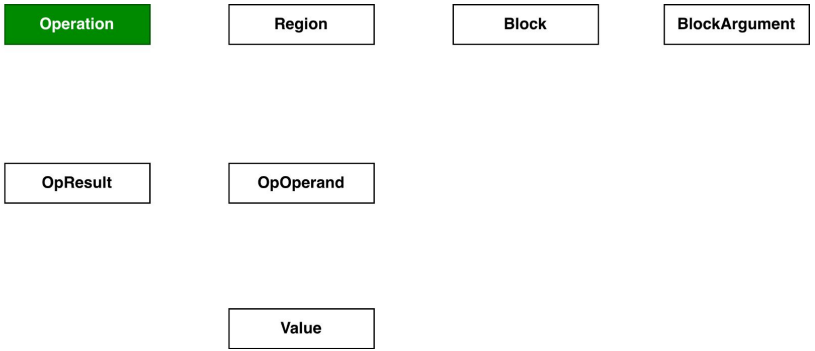
Warmup Quiz: Point at Each




```
func.func @increment(%arg0: !list.list<i32>)-> !list.list<i32> {
 %c1 = arith.constant 1 : i32
 %0 = list.map %arg0 with (%arg1: i32) -> i32 {
   %1 = arith.addi %arg1, %c1 : i32
   list.yield %1 : i32
 }
 return %0 : !list.list<i32>
}
```

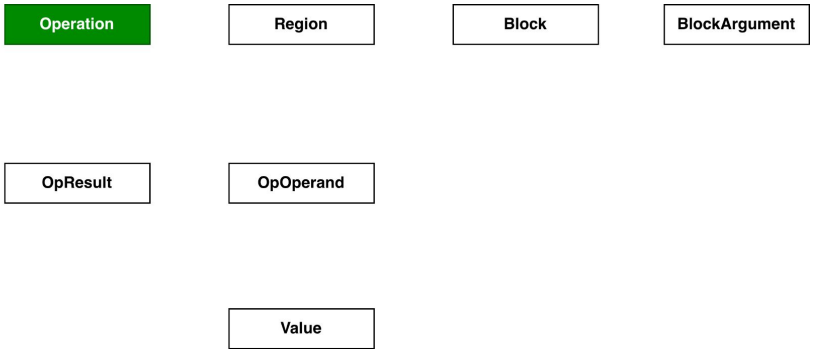
Warmup Quiz: Point at Each

```
func.func @increment(%arg0: !list.list<i32>)-> !list.list<i32> {
 %c1 = arith.constant 1 : i32
 %0 = list.map %arg0 with (%arg1: i32) -> i32 {
   %1 = arith.addi %arg1, %c1 : i32
   list.yield %1 : i32
 }
 return %0 : !list.list<i32>
}
```



Warmup Quiz: Point at Each

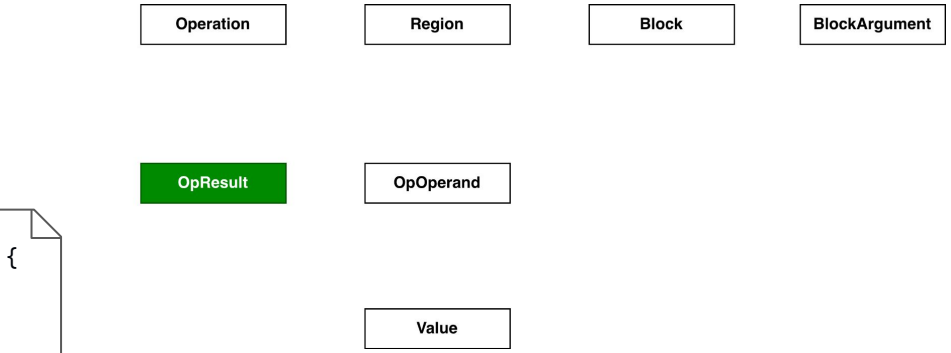
```
func.func @increment(%arg0: !list.list<i32>)-> !list.list<i32> {
 %c1 = arith.constant 1 : i32
 %0 = list.map %arg0 with (%arg1: i32) -> i32 {
   %1 = arith.addi %arg1, %c1 : i32
   list.yield %1 : i32
 }
 return %0 : !list.list<i32>
}
```



6 operations. If you count the implicit `builtin.module`, 7 operations.

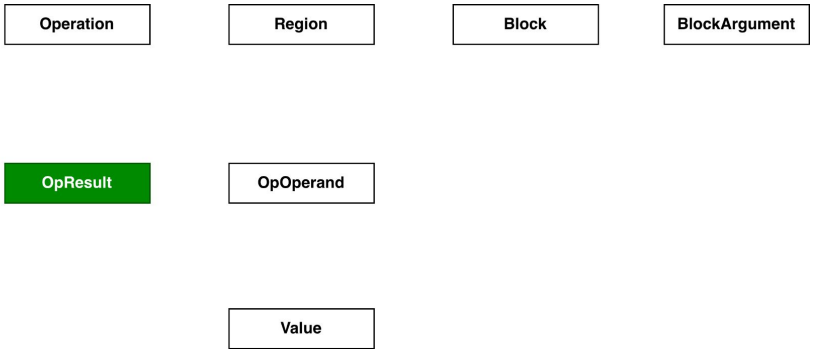
Warmup Quiz: Point at Each

```
func.func @increment(%arg0: !list.list<i32>)-> !list.list<i32> {
 %c1 = arith.constant 1 : i32
 %0 = list.map %arg0 with (%arg1: i32) -> i32 {
   %1 = arith.addi %arg1, %c1 : i32
   list.yield %1 : i32
 }
 return %0 : !list.list<i32>
}
```



Warmup Quiz: Point at Each

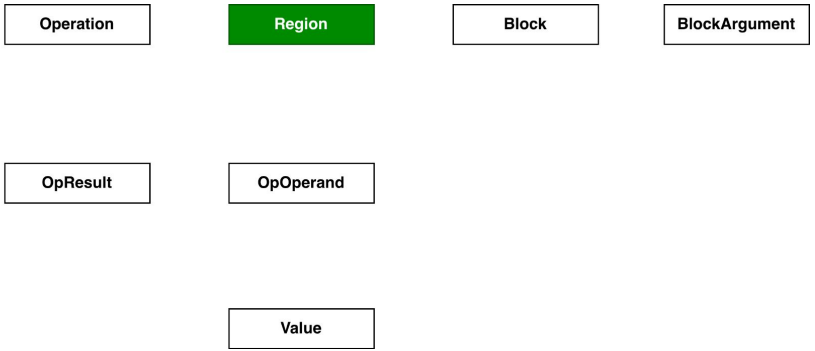
`OpResult`: SSA value defined by an operation.




```
func.func @increment(%arg0: !list.list<i32>)-> !list.list<i32> {
 %c1 = arith.constant 1 : i32
 %0 = list.map %arg0 with (%arg1: i32) -> i32 {
   %1 = arith.addi %arg1, %c1 : i32
   list.yield %1 : i32
 }
 return %0 : !list.list<i32>
}
```

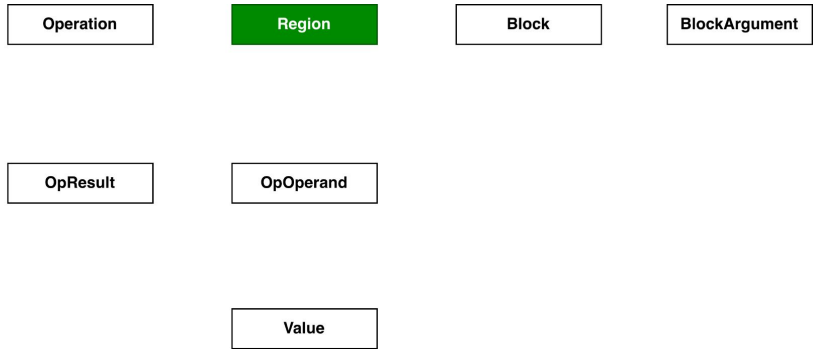
Warmup Quiz: Point at Each

```
func.func @increment(%arg0: !list.list<i32>)-> !list.list<i32> {
 %c1 = arith.constant 1 : i32
 %0 = list.map %arg0 with (%arg1: i32) -> i32 {
   %1 = arith.addi %arg1, %c1 : i32
   list.yield %1 : i32
 }
 return %0 : !list.list<i32>
}
```



Warmup Quiz: Point at Each

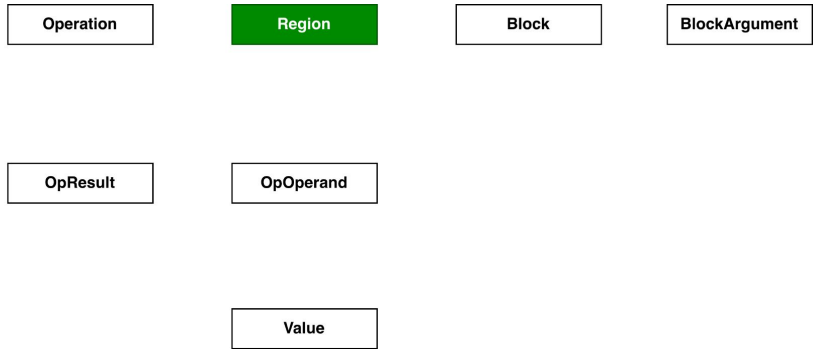
```
func.func @increment(%arg0: !list.list<i32>)-> !list.list<i32> {
  %c1 = arith.constant 1 : i32
  %0 = list.map %arg0 with (%arg1: i32) -> i32 {
    %1 = arith.addi %arg1, %c1 : i32
    list.yield %1 : i32
  }
  return %0 : !list.list<i32>
}
```



Region: list of blocks. `func.func` and `list.map` have one region each.

Warmup Quiz: Point at Each

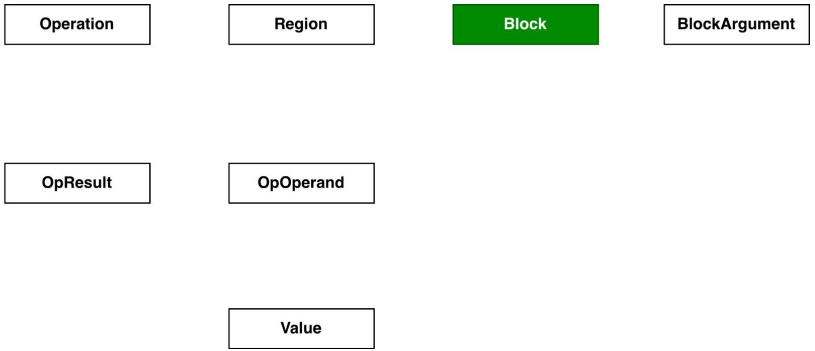
```
"func.func"() <{function_type = (!list.list<i32>) -> !list.list<i32>,
                sym_name = "increment"}> (
{
^bb0(%arg0: !list.list<i32>):
  %0 = "arith.constant"() <{value = 1 : i32}> : () -> i32
  %1 = "list.map"(%arg0) (
  {
  ^bb0(%arg1: i32):
    %2 = "arith.addi"(%arg1, %0) : (i32, i32) -> i32
    "list.yield"(%2) : (i32) -> ()
  }
  ) : (!list.list<i32>) -> !list.list<i32>
  "func.return"(%1) : (!list.list<i32>) -> ()
}
) : () -> ()
```



Region: list of blocks. `func.func` and `list.map` have one region each.

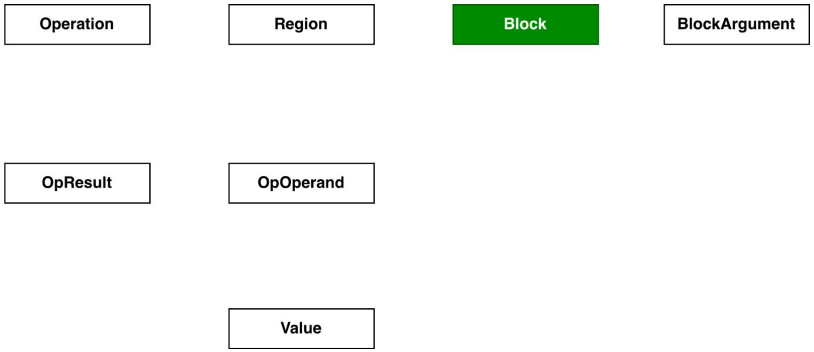
Warmup Quiz: Point at Each

```
func.func @increment(%arg0: !list.list<i32>)-> !list.list<i32> {
 %c1 = arith.constant 1 : i32
 %0 = list.map %arg0 with (%arg1: i32) -> i32 {
   %1 = arith.addi %arg1, %c1 : i32
   list.yield %1 : i32
 }
 return %0 : !list.list<i32>
}
```



Warmup Quiz: Point at Each

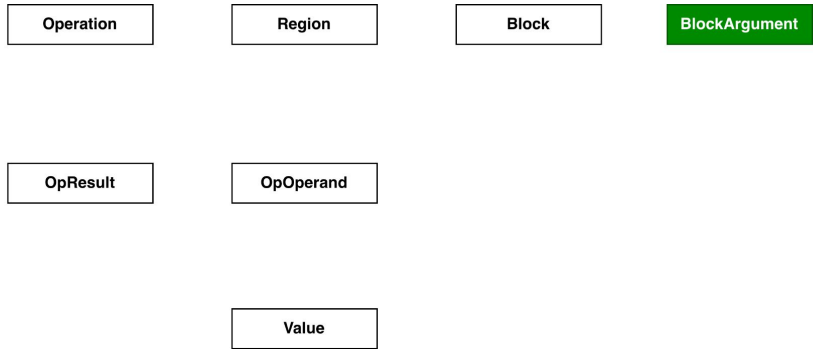
Block: a list of operations. Every region in this example has exactly one block.




```
func.func @increment(%arg0: !list.list<i32>)-> !list.list<i32> {
  %c1 = arith.constant 1 : i32
  %0 = list.map %arg0 with (%arg1: i32) -> i32 {
    %1 = arith.addi %arg1, %c1 : i32
    list.yield %1 : i32
  }
  return %0 : !list.list<i32>
}
```

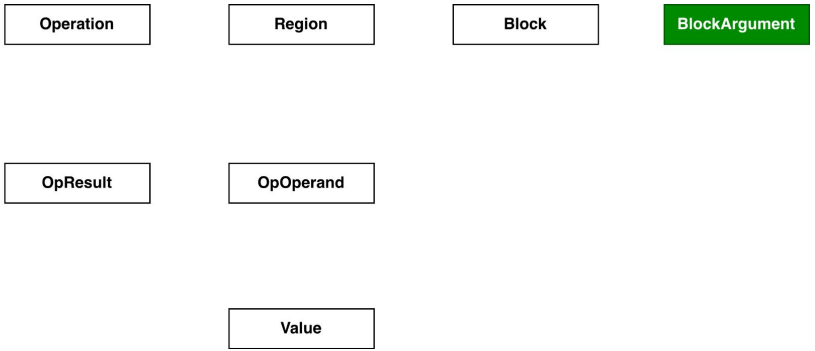
Warmup Quiz: Point at Each

```
func.func @increment(%arg0: !list.list<i32>)-> !list.list<i32> {
 %c1 = arith.constant 1 : i32
 %0 = list.map %arg0 with (%arg1: i32) -> i32 {
   %1 = arith.addi %arg1, %c1 : i32
   list.yield %1 : i32
 }
 return %0 : !list.list<i32>
}
```



Warmup Quiz: Point at Each

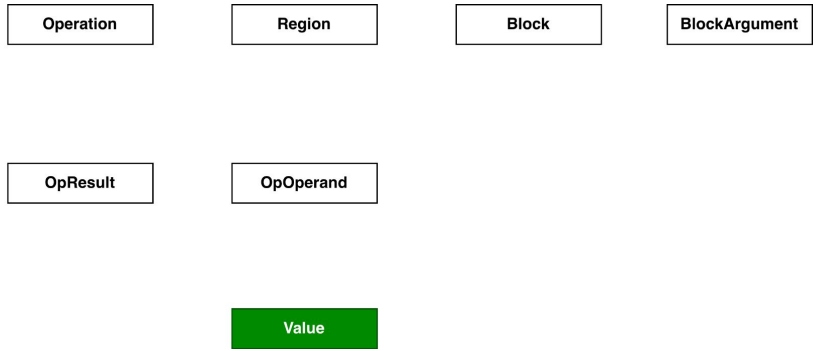
```
func.func @increment(%arg0: !list.list<i32>)-> !list.list<i32> {
 %c1 = arith.constant 1 : i32
 %0 = list.map %arg0 with (%arg1: i32) -> i32 {
   %1 = arith.addi %arg1, %c1 : i32
   list.yield %1 : i32
 }
 return %0 : !list.list<i32>
}
```



BlockArgument = SSA value defined by a block

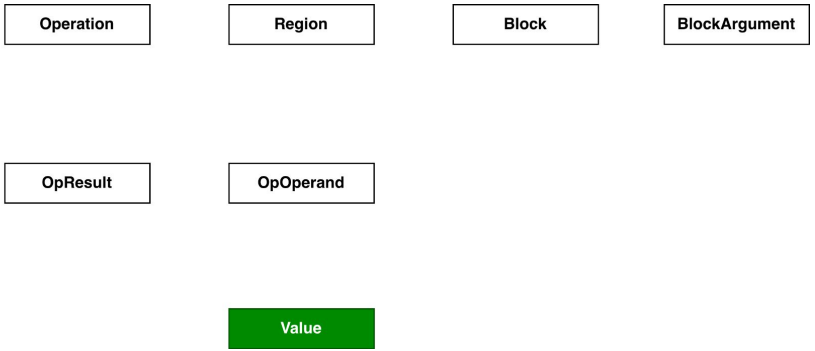
Warmup Quiz: Point at Each

```
func.func @increment(%arg0: !list.list<i32>)-> !list.list<i32> {
 %c1 = arith.constant 1 : i32
 %0 = list.map %arg0 with (%arg1: i32) -> i32 {
   %1 = arith.addi %arg1, %c1 : i32
   list.yield %1 : i32
 }
 return %0 : !list.list<i32>
}
```



Warmup Quiz: Point at Each

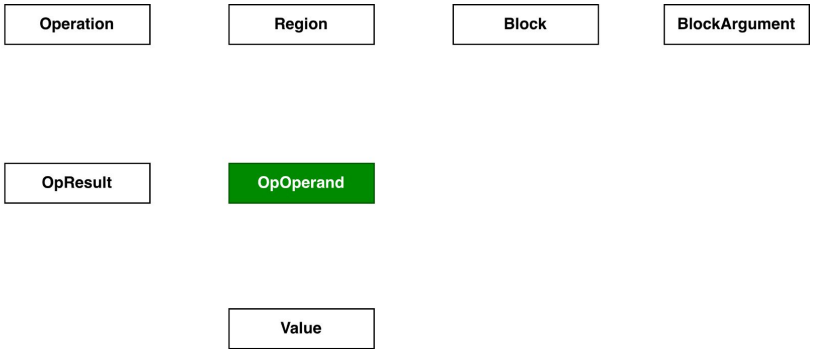
```
func.func @increment(%arg0: !list.list<i32>)-> !list.list<i32> {
 %c1 = arith.constant 1 : i32
 %0 = list.map %arg0 with (%arg1: i32) -> i32 {
   %1 = arith.addi %arg1, %c1 : i32
   list.yield %1 : i32
 }
 return %0 : !list.list<i32>
}
```



Value = OpResult or BlockArgument

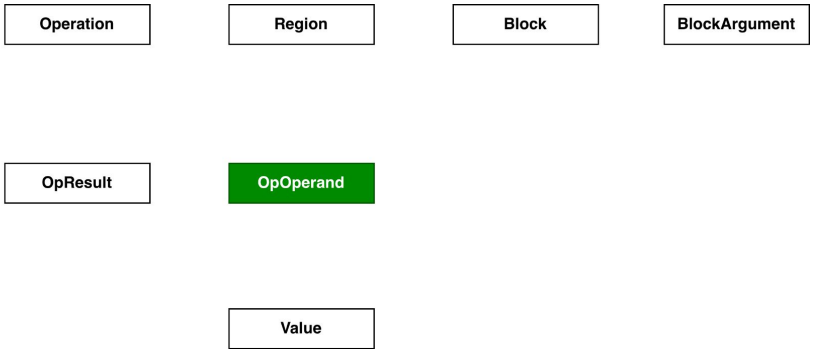
Warmup Quiz: Point at Each

```
func.func @increment(%arg0: !list.list<i32>)-> !list.list<i32> {
 %c1 = arith.constant 1 : i32
 %0 = list.map %arg0 with (%arg1: i32) -> i32 {
   %1 = arith.addi %arg1, %c1 : i32
   list.yield %1 : i32
 }
 return %0 : !list.list<i32>
}
```



Warmup Quiz: Point at Each

```
func.func @increment(%arg0: !list.list<i32>)-> !list.list<i32> {
 %c1 = arith.constant 1 : i32
 %0 = list.map %arg0 with (%arg1: i32) -> i32 {
   %1 = arith.addi %arg1, %c1 : i32
   list.yield %1 : i32
 }
 return %0 : !list.list<i32>
}
```



OpOperand = “use” of an SSA value

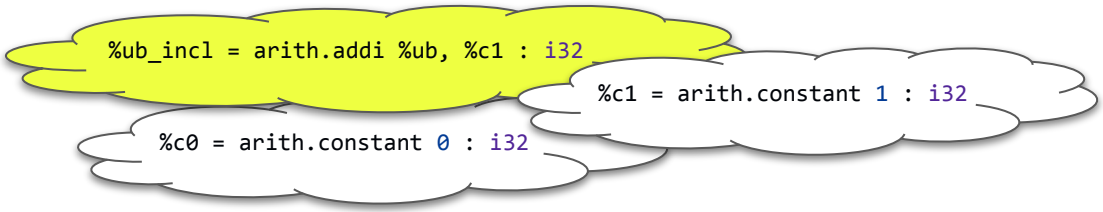
C++ API by Example

Creating an Operation

Operation

- OpResult opResults[numResults]
- Block *block
- Location location
- unsigned orderIndex
- unsigned numResults
- unsigned numRegions
- unsigned numSuccessors
- unsigned numOperands
- OperationName name
- DictionaryAttr attrs
- OpProperties prop
- BlockOperand blockOperands[numSuccessors]
- Region regions[numRegions]
- OpOperand operands[numOperands]

```
Value c0 = ..., c1 = ...;                  // %c0, %c1 already exist as Values
Location loc = ...;

OperationState state(loc, "arith.addi");
state.addOperands({c0, c1});               // exactly 2 operands
state.addTypes(c0.getType());              // exactly 1 result
// no addAttribute(...), no addRegion()
Operation *op = Operation::create(state);  // malloc, create a "detached" operation

op->dump();                                // dump to stderr
op->getOperand().dump()                    // dump operand 0 to stderr
op->destroy();                             // deallocate memory (incl. nested IR)
```



Operations are heap-allocated.

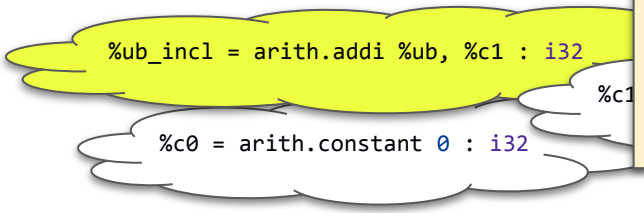
Creating an Operation

Operation

- OpResult opResults[numResults]
- Block *block
- Location location
- unsigned orderIndex
- unsigned numResults
- unsigned numRegions
- unsigned numSuccessors
- unsigned numOperands
- OperationName name
- DictionaryAttr attrs
- OpProperties prop
- BlockOperand blockOperands[numSuccessors]
- Region regions[numRegions]
- OpOperand operands[numOperands]

```
Value c0 = ...                                   0, %c1 already exist as Values
Location loc =

OperationState
state.addOperands({c                             // exactly 2 operands
state.addTypes(c0.getType());                    // exactly 1 result
// no addAttribute(...), no addRegion()
Operation *op = Operation::create(state);        // malloc, create a "detached" operation

op->dump();                                      // dump to stderr
op->getOperand().dump()                          // dump operand 0 to stderr
op->destroy();                                   // deallocate memory (incl. nested IR)
```

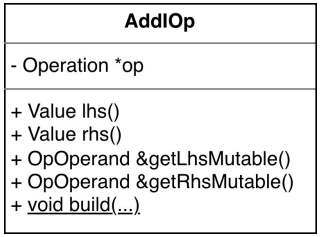
Useful API for Types:
`value.getType()`
`value.setType(t)`
`builder.getI64Type()`



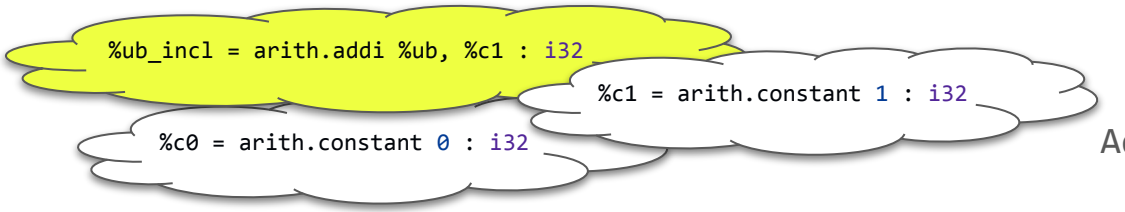
`op->destroy()` just frees the memory. `op->erase()` removes the op from the parent block and destroys it.

Operations are heap-allocated.

Creating an Operation

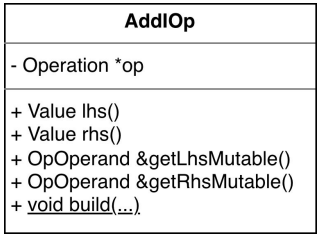



```
Value c0 = ..., c1 = ...;
Location loc = ...;

OperationState state(loc, arith::AddIOp::getOperationName());
arith::AddIOp::build(builder, state, c0, c1);
Operation *op = Operation::create(state);    // malloc

op->dump();
cast<arith::AddIOp>(op).getLhs().dump();
// op->getOperand(0).dump();
op->destroy();                               // free
```



`AddIOp` is a wrapper around `Operation *`.

The More Common Case: Create + Insert




```
Value c0 = ..., c1 = ...;
Location loc = ...;

builder.setInsertionPointAfter(c1.getDefiningOp());
AddIOp op = arith::AddIOp::create(builder, loc, c0, c1);   // malloc + insert


op->dump();
op.getLhs().dump();
// op->getOperand(0).dump();
op->destroy();                                              // free
op->erase();                                                // remove + free
```

```
%c0 = arith.constant 0 : i32
%c1 = arith.constant 1 : i32
// %ub_incl = “arith.addi”(%ub, %c1) : (i32, i32) -> (i32)
%ub_incl = arith.addi %ub, %c1 : i32
```

AddIOp is a wrapper around Operation *.

The More Common Cas

Insertion point API:

```
setInsertionPoint(op)
setInsertionPointAfter(op)
setInsertionPointToStart(block)
setInsertionPointToEnd(block)
OpBuilder::InsertionGuard g(builder);
```

AddIOp
- Operation *op
+ Value lhs() + Value rhs() + OpOperand &getLhsMutable() + OpOperand &getRhsMutable() + void build(...)

```
Value c0 =
Location l

builder.setInsertionPointAfter(...);
AddIOp op = arith::AddIOp::create(builder, loc, c0, c1);   // malloc + insert


op->c
op.ge
// op
op->destroy();                                              // free
op->erase();                                                // remove + free
```

Destroying a non-detached operation leaves a dangling pointer. (More about that later.)

```
%c0 = arith.constant 0 : i32
%c1 = arith.constant 1 : i32
// %ub_incl = "arith.addi"(%ub, %c1) : (i32, i32) -> (i32)
%ub_incl = arith.addi %ub, %c1 : i32
```

AddIOp is a wrapper around Operation *.

Casting an Operation


```
Operation *op = ...;

// dyn_cast is like C++ dynamic_cast.
if (arith::AddIOp addOp = dyn_cast<arith::AddIOp>(op)) {
 addOp.getLhs().dump();
}

// isa<...> --> bool
llvm::errs() << "Is AddIOp: " << isa<arith::AddIOp>(op) << "\n";

// cast<...> asserts that op has the specified type.
arith::AddIOp addOp = cast<arith::AddIOp>(op);
```

Pass `ConcreteOp` by value.
Pass `Operation` by pointer/reference.

Quiz: What if You Pass `ConcreteOp` by Reference?

```
void doSomething(arith::AddIOp *op) { op->getLhs().dump(); }


arith::AddIOp myOp = ...;
doSomething(&myOp);
```

Quiz: What if You Pass `ConcreteOp` by Reference?

```
void doSomething(arith::AddIOp *op) { op->getLhs().dump(); }


arith::AddIOp myOp = ...;
doSomething(&myOp);
```

Nothing breaks, but access may be less efficient (extra pointer indirection).
Equality check may break when comparing pointers.

Quiz: What if You Pass `Operation` by Value?

```
void doSomething(Operation op) { /* … */ }


Operation *myOp = ...;
doSomething(*myOp);
```

Quiz: What if You Pass `Operation` by Value?

```
void doSomething(Operation op) { /* … */ }

Operation *myOp = ...;
doSomething(*myOp);
```

You cannot: all constructors of `Operation` (including the copy constructor) are private. Compiler error. If you could: the operation would be copied, but (probably) not registered in other data structures (inconsistent state). Compare two operations by comparing their pointers.

Query the Parent Block

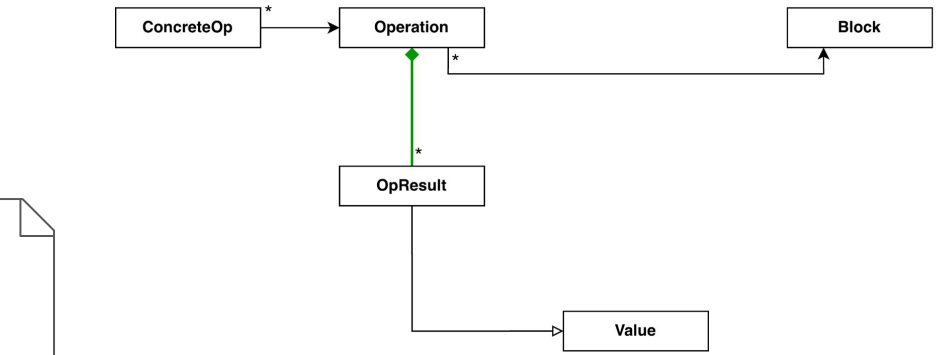

```
func.func @sum(%lb: i32, %ub: i32) -> i32 {
  %c0 = arith.constant 0 : i32
  %c1 = arith.constant 1 : i32
  %ub_incl = arith.addi %ub, %c1 : i32
  ...
}
```

```
// arith::AddIOp addOp = ...;
Operation *addOp = ...;
Block *b = addOp->getBlock();
```

Pass `Block` by reference/pointer.

Query Results

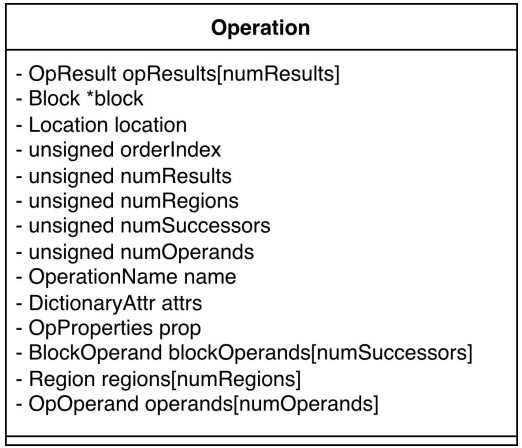
```
func.func @sum(%lb: i32, %ub: i32) -> i32 {
  %c0 = arith.constant 0 : i32
  %c1 = arith.constant 1 : i32
  %ub_incl = arith.addi %ub, %c1 : i32
  ...
}
```




```
// arith::AddIOp addOp = ...;
// OpResult v = addOp.getResult();
Operation *addOp = ...;
OpResult v = addOp->getResult(0);
assert(v.getDefiningOp() == addOp);
```

Pass `Value` / `OpResult` / `BlockArgument` by value.

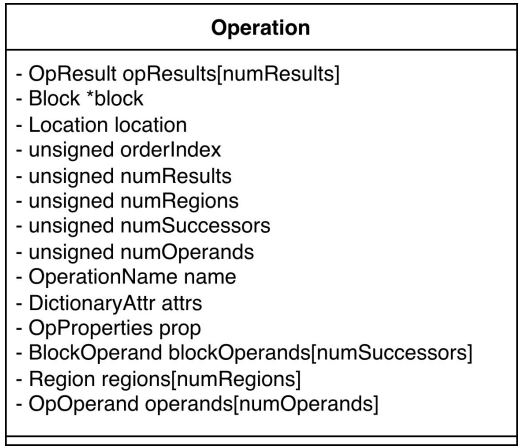
Quiz: Runtime Complexity of Querying an `OpResult`?



How expensive is it to get the i-th result?

```
OpResult v = addOp->getResult(0);
```

Quiz: Runtime Complexity of Querying an `OpResult`?



How expensive is it to get the i-th result?

```
OpResult v = addOp->getResult(0);
```

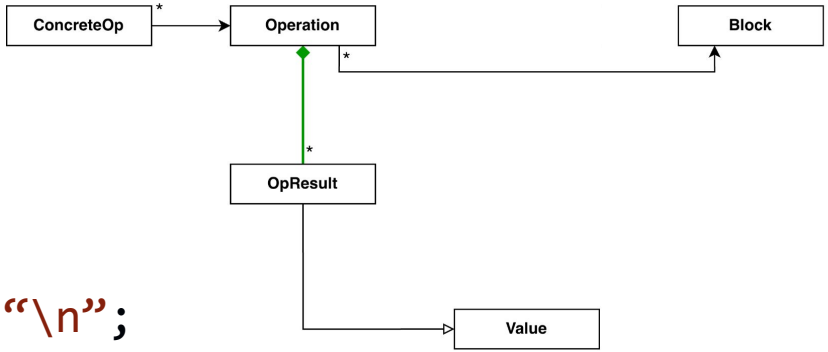
O(1), just take it from the `opResults` array.

Quiz: What's Wrong Here?

```
Operation *op = …;
Value v = op->getResult(0);
op->destroy();
llvm::errs() << “type of v: “
             << v.getType() << “\n”;
```

Quiz: What's Wrong Here?

```
Operation *op = ...;
Value v = op->getResult(0);
op->destroy();
llvm::errs() << "type of v: "
             << v.getType() << "\n";
```

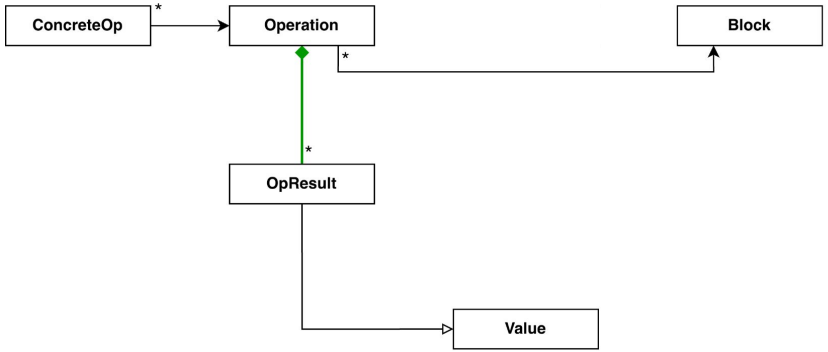


The lifetime of `OpResult` ends with the lifetime of `Operation`. `Value` contains an internal pointer to `op`. After destroying `op`, that pointer is dangling. (Use after free!)

Quiz: How to Add an Additional Result?

Operation

- OpResult opResults[numResults]
- Block *block
- Location location
- unsigned orderIndex
- unsigned numResults
- unsigned numRegions
- unsigned numSuccessors
- unsigned numOperands
- OperationName name
- DictionaryAttr attrs
- OpProperties prop
- BlockOperand blockOperands[numSuccessors]
- Region regions[numRegions]
- OpOperand operands[numOperands]

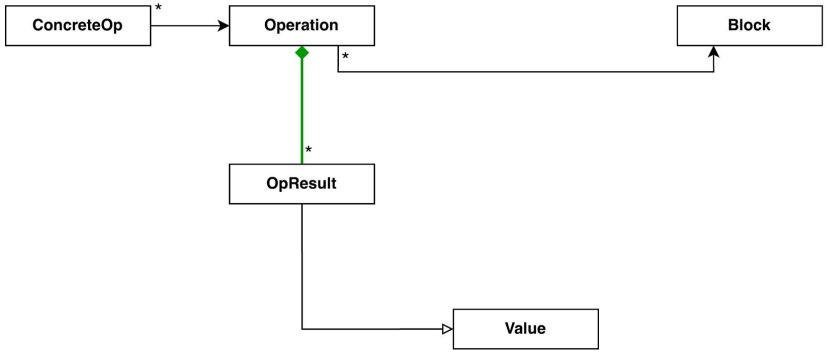



```
// before:
%0 = “test.dummy”() : () -> (i32)

// after:
%0, %1 = “test.dummy” : () -> (i32, i32)
```

Quiz: How to Add an Additional Result?

Operation

- OpResult opResults[numResults]
- Block *block
- Location location
- unsigned orderIndex
- unsigned numResults
- unsigned numRegions
- unsigned numSuccessors
- unsigned numOperands
- OperationName name
- DictionaryAttr attrs
- OpProperties prop
- BlockOperand blockOperands[numSuccessors]
- Region regions[numRegions]
- OpOperand operands[numOperands]

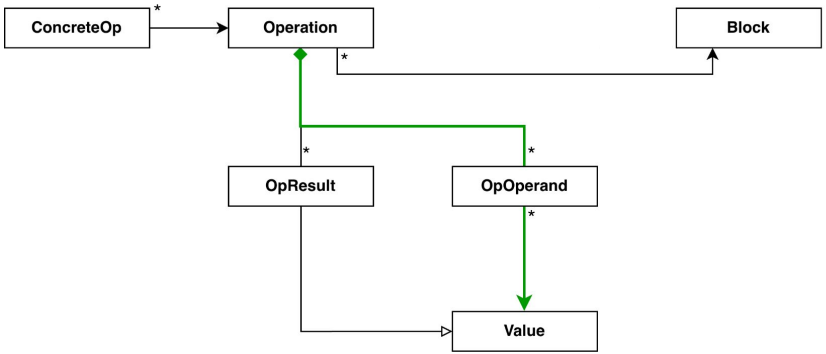


Additional results require more bytes in `Operation`. Create a new `Operation`, RAUW (“replace-all-uses-with”), erase.

```
// before:
%0 = “test.dummy”() : () -> (i32)

// after:
%0, %1 = “test.dummy” : () -> (i32, i32)
```

Query (Op)Operands

```
func.func @sum(%lb: i32, %ub: i32) -> i32 {
  %c0 = arith.constant 0 : i32
  %c1 = arith.constant 1 : i32
  %ub_incl = arith.addi %ub, %c1 : i32
  ...
}
```

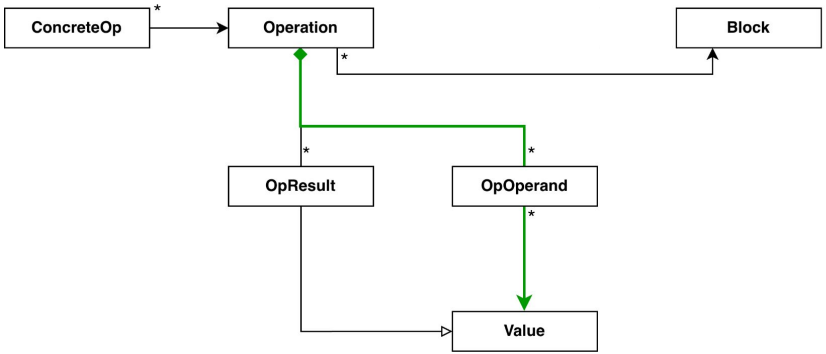



```
// arith::AddIOp addOp = ...;
// OpOperand &use = addOp.getLhsMutable();
Operation *addOp = ...;
OpOperand &use = addOp->getOpOperand(0);
Value lhs = use.get();
```

OpOperand = “use” of an SSA value.
Pass OpOperands as reference/pointer.

Query (Op)Operands

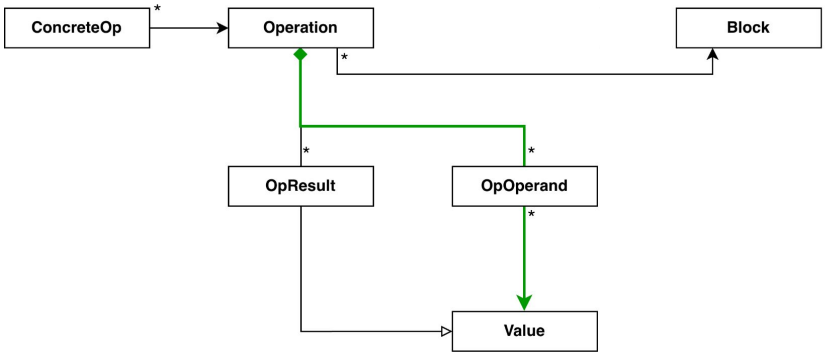
```
func.func @sum(%lb: i32, %ub: i32) -> i32 {
  %c0 = arith.constant 0 : i32
  %c1 = arith.constant 1 : i32
  %ub_incl = arith.addi %ub, %c1 : i32
  ...
}
```




```
// arith::AddIOp addOp = ...;
// Value lhs = addOp.getLhs();
Operation *addOp = ...;
Value lhs = addOp->getOperand(0);
```

Changing (Op)Operands

```
func.func @sum(%lb: i32, %ub: i32) -> i32 {
  %c0 = arith.constant 0 : i32
  %c1 = arith.constant 1 : i32
  %ub_incl = arith.addi %c0, %c1 : i32
  ...
}
```

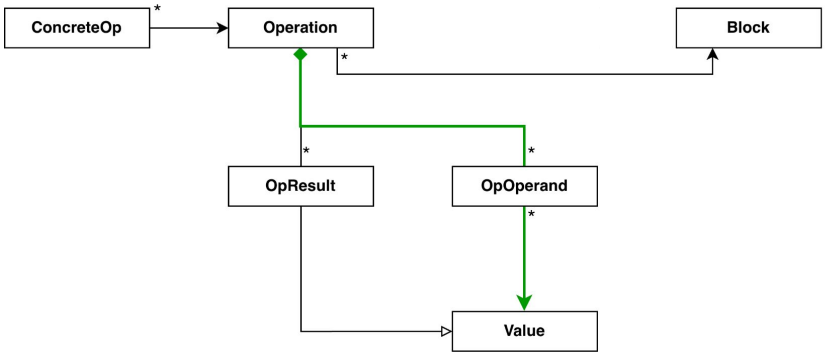



```
// arith::AddIOp addOp = ...;
// OpOperand &use = addOp.getLhsMutable();
Operation *addOp = ...;
OpOperand &use = addOp->getOpOperand(0);
use.assign(c0);   // or: use.set(c0)
```

OpOperand = “use” of an SSA value.
Pass OpOperands as reference/pointer.

Changing (Op)Operands

```
func.func @sum(%lb: i32, %ub: i32) -> i32 {
  %c0 = arith.constant 0 : i32
  %c1 = arith.constant 1 : i32
  %ub_incl = arith.addi %c0, %c1 : i32
  …
}
```

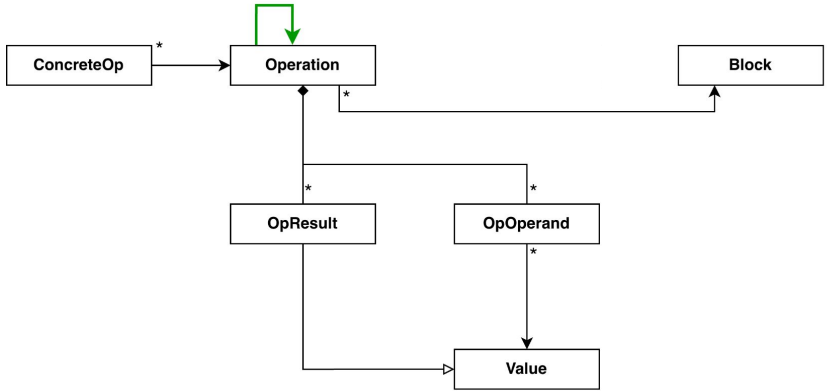



```
Operation *addOp = …;
addOp->setOperand(0, c0);
```

OpOperand = “use” of an SSA value.
Pass OpOperands as reference/pointer.

Query Previous / Next Op in Block

```
func.func @sum(%lb: i32, %ub: i32) -> i32 {
  %c0 = arith.constant 0 : i32
  %c1 = arith.constant 1 : i32
  %ub_incl = arith.addi %c0, %c1 : i32
  …
}
```

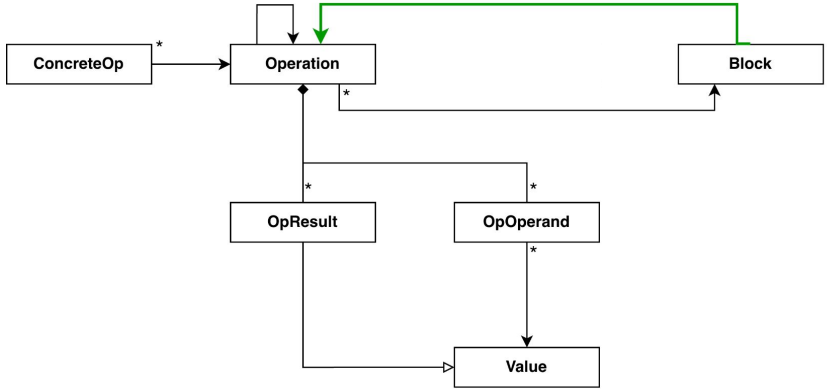



```
Operation *c1 = …;
Operation *c0 = c1->getPrevNode();
Operation *ubIncl = c1->getNextNode();
```

Operations in a block form a doubly-linked list.

Query the First / Last Operation of a Block

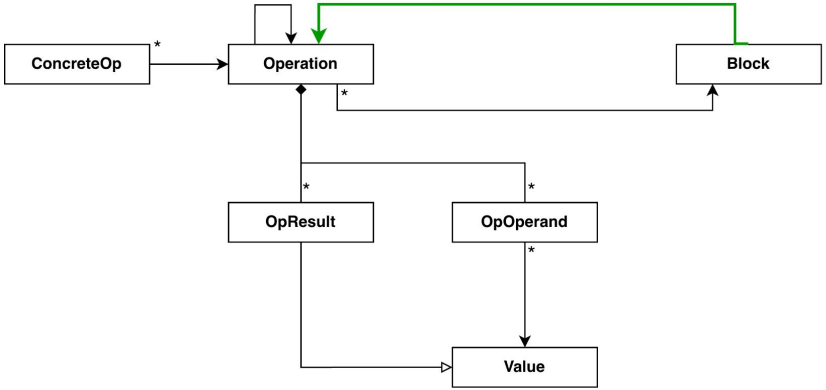
```
func.func @sum(%lb: i32, %ub: i32) -> i32 {
  %c0 = arith.constant 0 : i32
  ...
  func.return %sum : i32
}
```



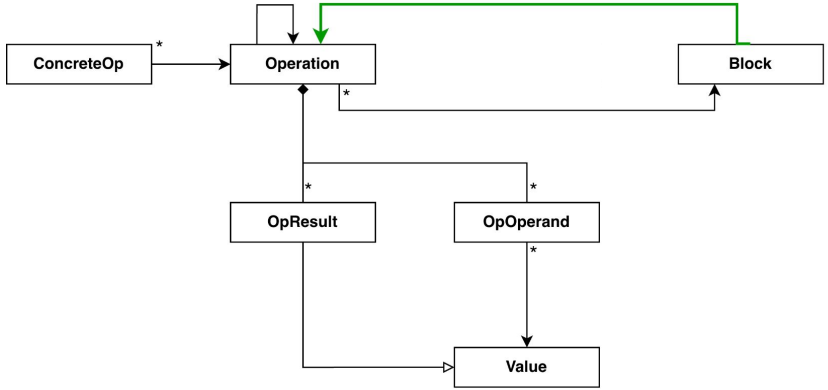

```
Block *funcBlock = ...;
Operation *first = &funcBlock->front();
Operation *last = &funcBlock->back();
```

Blocks store a pointer to the first and last operation. There are also iterator APIs such as `Block::begin()`, `Block::end()`, `Block::rbegin()`, `Block::rend()`, `Block::getOps<OpType>`.

Quiz: Runtime Complexity of Getting the i-th Operation?



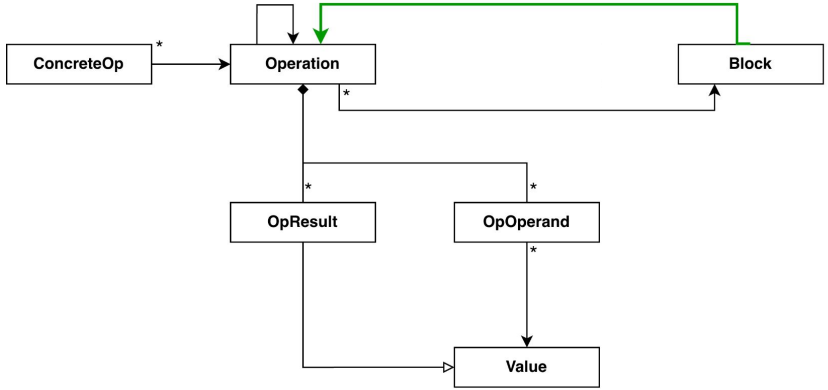
Quiz: Runtime Complexity of Getting the i-th Operation?



Get the first operation in the block, then walk the linked list i times. Runtime cost is linear in the number of operations per block. “Getting the i-th element” is uncommon.

Quiz: Runtime Complexity of Inserting an Operation?

(Ignore malloc and the steps required to build the `AddIOp`.)

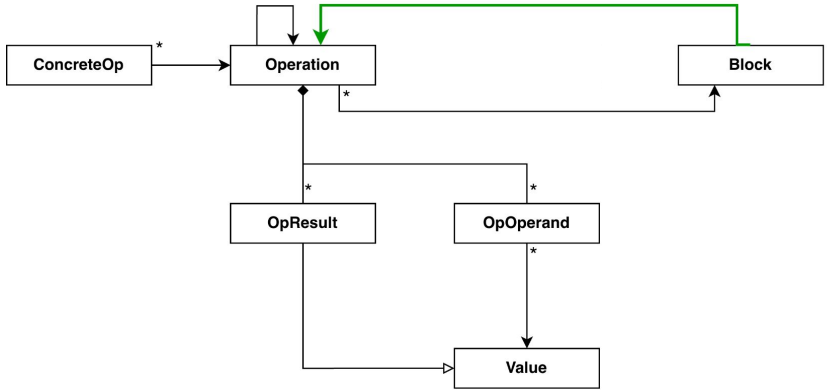



```
func.func @sum(%lb: i32, %ub: i32) -> i32 {
  %c0 = arith.constant 0 : i32
  %c1 = arith.constant 1 : i32
  %ub_incl = arith.addi %c0, %c1 : i32
  ...
}
```

```
Operation *c1 = ...;
OpBuilder b(getContext());
b.setInsertionPoint(c1);   // set insertion point before c1
AddIOp newOp = AddIOp::create(b, loc, lhsVal, rhsVal);
```

Quiz: Runtime Complexity of Inserting an Operation?

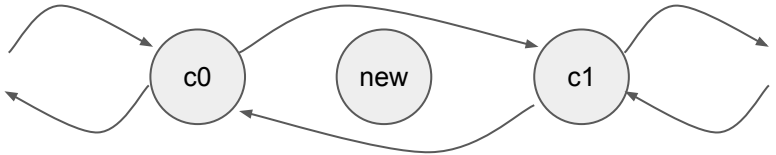
(Ignore malloc and the steps required to build the `AddIOp`.)



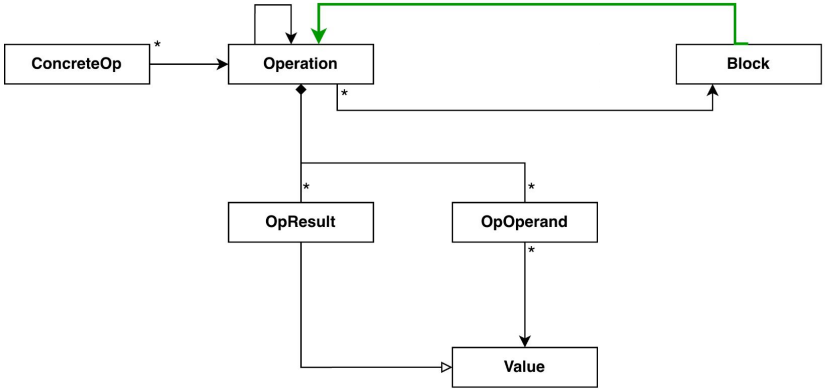

```
func.func @sum(%lb: i32, %ub: i32) -> i32 {
  %c0 = arith.constant 0 : i32
  %c1 = arith.constant 1 : i32
  %ub_incl = arith.addi %c0, %c1 : i32
  …
}
```

Insertion into a doubly-linked list: 4 pointer updates.
One more pointer update to set the “parent” block.

```
Operation *c1 = …;
OpBuilder b(getContext());
b.setInsertionPoint(c1);   // set insertion point before c1
AddIOp newOp = AddIOp::create(b, loc, lhsVal, rhsVal);
```



Quiz: Runtime Complexity of Moving an Operation?

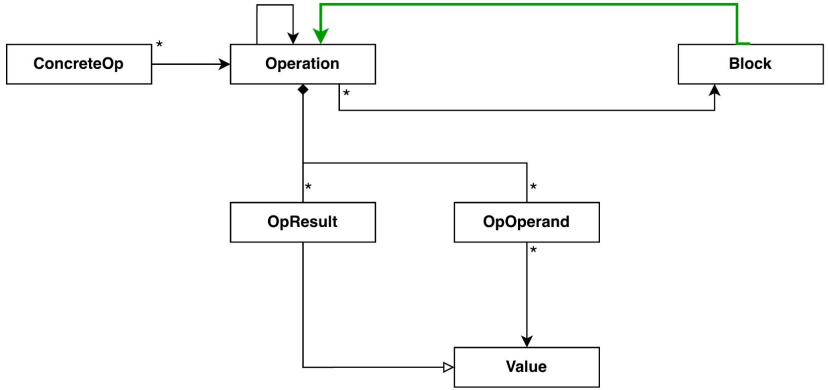



```
func.func @sum(%lb: i32, %ub: i32) -> i32 {
  %c0 = arith.constant 0 : i32
  %c1 = arith.constant 1 : i32
  %ub_incl = arith.addi %c0, %c1 : i32
  …
}
```

```
Operation *c0 = …;
Operation *c1 = …;
c1->moveBefore(c0);
```

Quiz: Runtime Complexity of Moving an Operation?

```
func.func @sum(%lb: i32, %ub: i32) -> i32 {
  %c0 = arith.constant 0 : i32
  %c1 = arith.constant 1 : i32
  %ub_incl = arith.addi %c0, %c1 : i32
  …
}
```

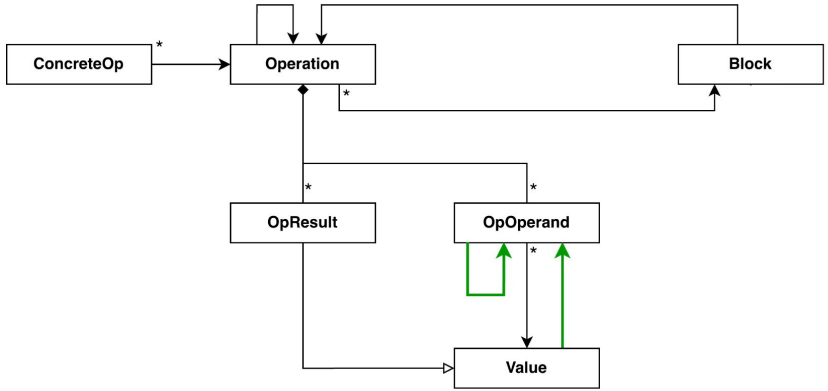



```
Operation *c0 = …;
Operation *c1 = …;
c1->moveBefore(c0);
```

1. Remove operation from doubly-linked list: 2 pointer updates.
2. Insert operation into doubly-linked list: 4 pointer updates.
3. Set block (owner) pointer on `Operation`.

Query the Uses of an SSA Value

```
func.func @sum(%lb: i32, %ub: i32) -> i32 {
  %c0 = arith.constant 0 : i32
  %c1 = arith.constant 1 : i32
  %c2 = arith.addi %c1, %c1 : i32
  %c3 = arith.addi %c2, %c1 : i32
}
```

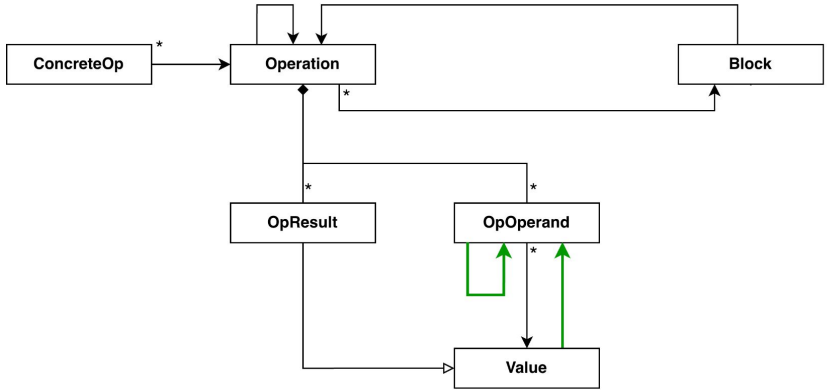



```
Value c1 = …;
OpOperand &firstUse = *ub.getUses().begin();
OpOperand &secondUse = *static_cast<OpOperand *>(
    firstUse.getNextOperandUsingThisValue());
```

OpOperand = “use” of an SSA value
Uses of an SSA value form a doubly-linked list.
Uses have no deterministic order!

Query the Uses of an SSA Value

```
func.func @sum(%lb: i32, %ub: i32) -> i32 {
  %c0 = arith.constant 0 : i32
  %c1 = arith.constant 1 : i32
  %c2 = arith.addi %c1, %c1 : i32
  %c3 = arith.addi %c2, %c1 : i32
}
```




```
Value c1 = ...;
for (OpOperand &use : c1.getUses()) { ... }
for (Operation *user : c1.getUsers()) { ... }
```

OpOperand = “use” of an SSA value
Uses of an SSA value form a doubly-linked list.
Uses have no deterministic order!

Summary: OpOperand API

	OpOperand
Get from Operation	`op->getOpOperand(i)` `addOp.getLhsMutable()`
Get Owning Operation	`oo.getOwner()`
Get Operand Number	`oo.getOperandNumber()`
Get Referenced Thing	`oo.get(value)`
Set Referenced Thing	`oo.set(value)`
Get Referenced Thing from Op	`op->getOperand(i)`
Get all Uses	`value.getUses()`
Get all Users	`value.getUsers()`

Quiz: Replace-All-Uses-With: What's the Resulting IR?

```
func.func @sum(%lb: i32, %ub: i32) -> i32 {
  %c0 = arith.constant 0 : i32
  %c1 = arith.constant 1 : i32
  %c2 = arith.addi %c1, %c1 : i32
  %c3 = arith.addi %c2, %c1 : i32
}
```

```
Value c0, c1;
c1.replaceAllUsesWith(c0);
```

Quiz: Replace-All-Uses-With: What's the Resulting IR?

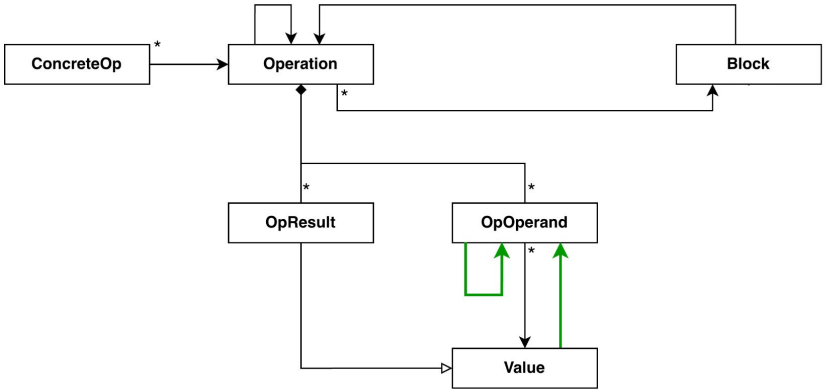
```
func.func @sum(%lb: i32, %ub: i32) -> i32 {
  %c0 = arith.constant 0 : i32
  %c1 = arith.constant 1 : i32
  %c2 = arith.addi %c1, %c1 : i32
  %c3 = arith.addi %c2, %c1 : i32
}
```

```
func.func @sum(%lb: i32, %ub: i32) -> i32 {
  %c0 = arith.constant 0 : i32
  %c1 = arith.constant 1 : i32
  %c2 = arith.addi %c0, %c0 : i32
  %c3 = arith.addi %c2, %c0 : i32
}
```

```
Value c0, c1;
c1.replaceAllUsesWith(c0);
```

Quiz: What's a Possible Implementation of RAUW?

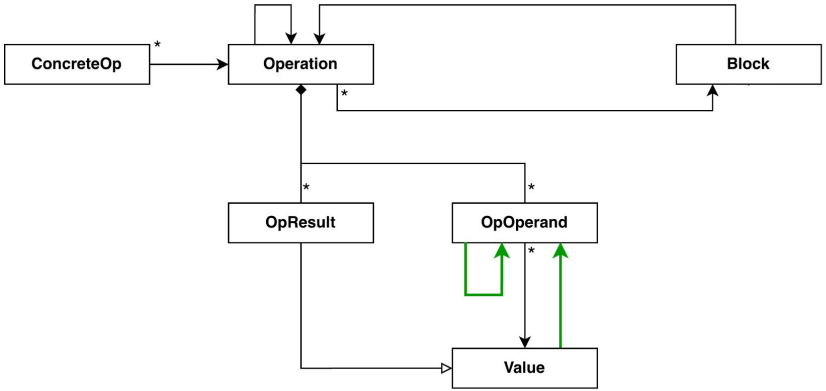
```
func.func @sum(%lb: i32, %ub: i32) -> i32 {
  %c0 = arith.constant 0 : i32
  %c1 = arith.constant 1 : i32
  %c2 = arith.addi %c1, %c1 : i32
  %c3 = arith.addi %c2, %c1 : i32
}
```




```
Value c0, c1;
c1.replaceAllUsesWith(c0);
```

Quiz: What's a Possible Implementation of RAUW?

```
func.func @sum(%lb: i32, %ub: i32) -> i32 {
  %c0 = arith.constant 0 : i32
  %c1 = arith.constant 1 : i32
  %c2 = arith.addi %c1, %c1 : i32
  %c3 = arith.addi %c2, %c1 : i32
}
```



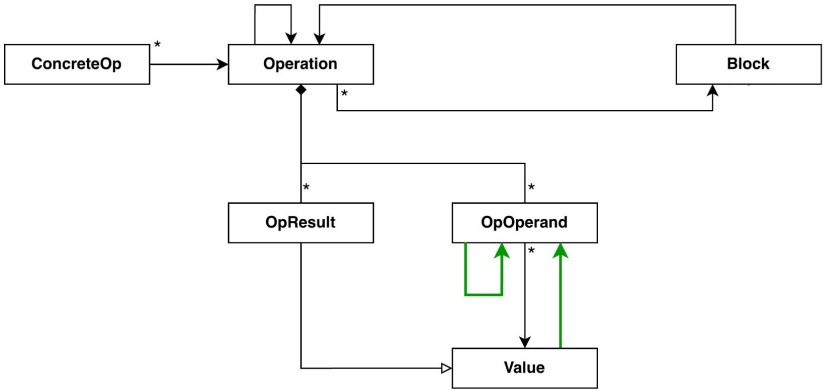

```
Value c0, c1;
c1.replaceAllUsesWith(c0);
```

```
for (OpOperand &use : llvm::make_early_inc_range(c1.getUses()))
  use.set(c0);
```

`make_early_inc_range` increments the iterator before returning the element.

Quiz: Runtime Complexity of RAUW?

```
func.func @sum(%lb: i32, %ub: i32) -> i32 {
  %c0 = arith.constant 0 : i32
  %c1 = arith.constant 1 : i32
  %c2 = arith.addi %c1, %c1 : i32
  %c3 = arith.addi %c2, %c1 : i32
}
```



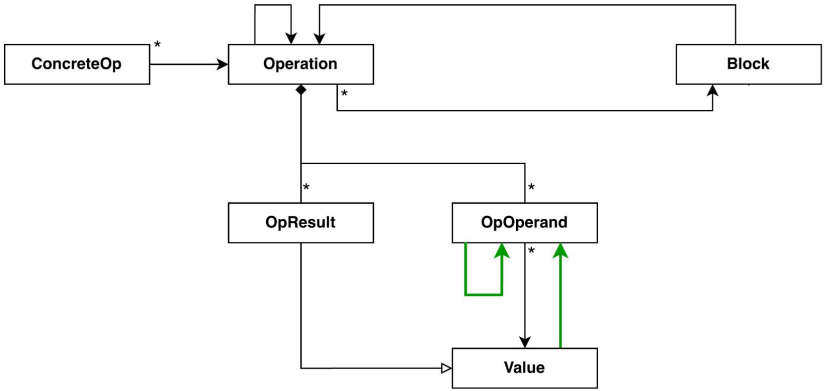

```
Value c0, c1;
c1.replaceAllUsesWith(c0);
```

```
for (OpOperand &use : llvm::make_early_inc_range(c1.getUses()))
  use.set(c0);
```

`make_early_inc_range` increments the iterator before returning the element.

Quiz: Runtime Complexity of RAUW?

```
func.func @sum(%lb: i32, %ub: i32) -> i32 {
  %c0 = arith.constant 0 : i32
  %c1 = arith.constant 1 : i32
  %c2 = arith.addi %c1, %c1 : i32
  %c3 = arith.addi %c2, %c1 : i32
}
```




```
Value c0, c1;
c1.replaceAllUsesWith(c0);
```

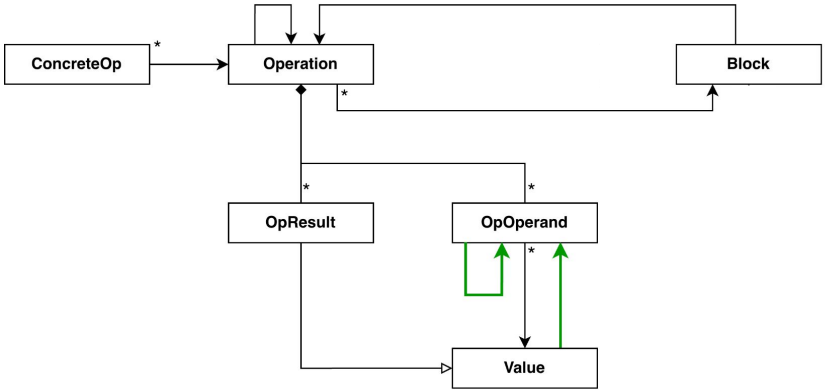
- Iterate over all uses of the original SSA value.
- Remove each use from the doubly-linked list (2 pointer updates).
- Set new SSA value for each use. (OpOperand → Value field.)
- Insert use each use into a doubly-linked list (4 pointer updates).
- **Total cost: 7*num_uses field updates.**

Quiz: What's Wrong?

```
OpResult v = ...;
if (v.getNumUses() == 0) {
  v.getDefiningOp()->erase();
}
```

Quiz: What's Wrong?

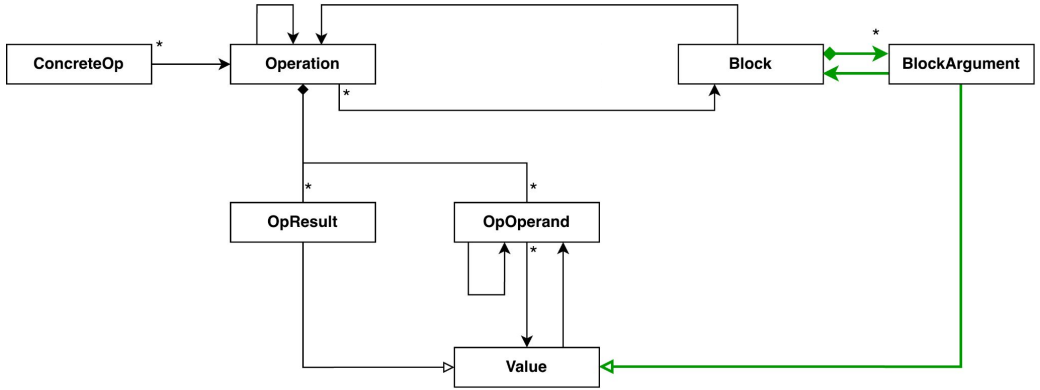
```
OpResult v = ...;
if (v.getNumUses() == 0) {
  v.getDefiningOp()->erase();
}
```



- The defining op of v could have more than one result. (The check is incomplete.)
- Counting the number of uses is requires walking the entire linked list. (The check is inefficient.)
  Better: `v.use_empty()` or `v.use_begin() == v.use_end()`.

Query the Block Arguments of a Block

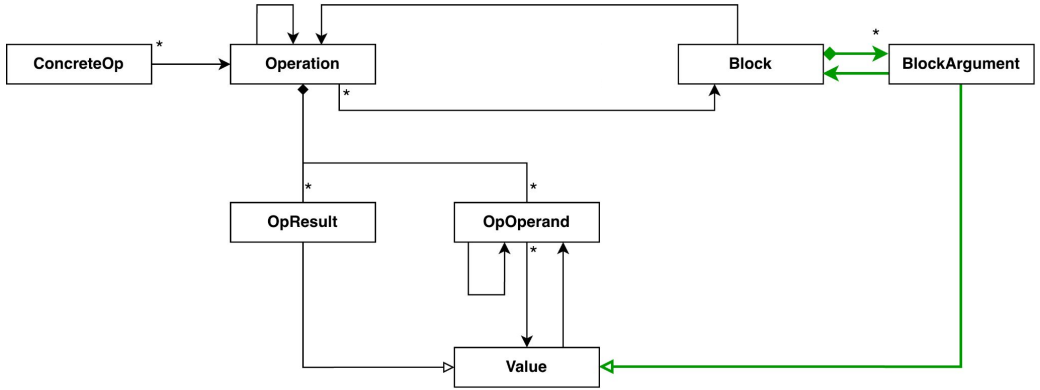
```
func.func @sum(%lb: i32, %ub: i32) -> i32 {
  ...
}
```




```
Block *funcBlock = ...;
BlockArgument lb = b->getArgument(0);
assert(lb.getOwner() == funcBlock);
```

Query the Block Arguments of a Block

```
"func.func"() <{function_type = (i32, i32) -> i32,
                sym_name = "sum"}> ({
^bb0(%lb: i32, %ub: i32):
  ...
}) : () -> ()
```



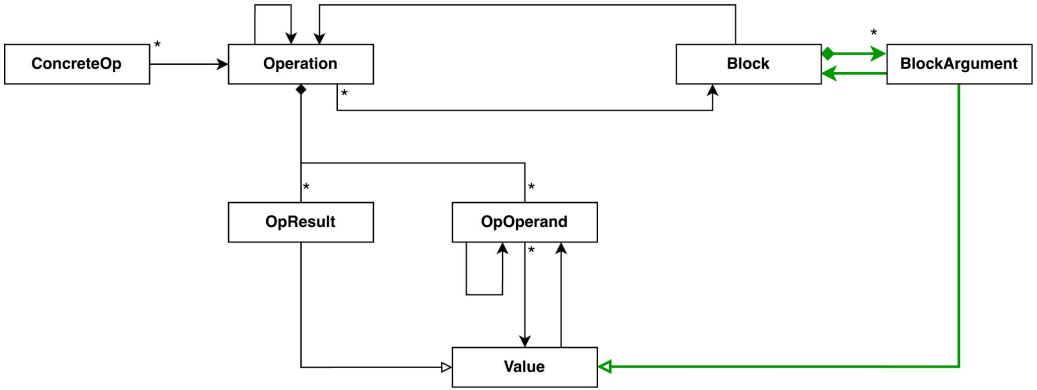

```
Block *funcBlock = ...;
BlockArgument lb = b->getArgument(0);
assert(lb.getOwner() == funcBlock);
```

Quiz: What's Wrong Here?

```
/// Returns true iff the extension op is fed by a vector.transfer_read.
static bool isFedByTransferRead(arith::ExtUIOp extOp) {
  return isa<vector::TransferReadOp>(extOp.getOperand().getDefiningOp());
}
```

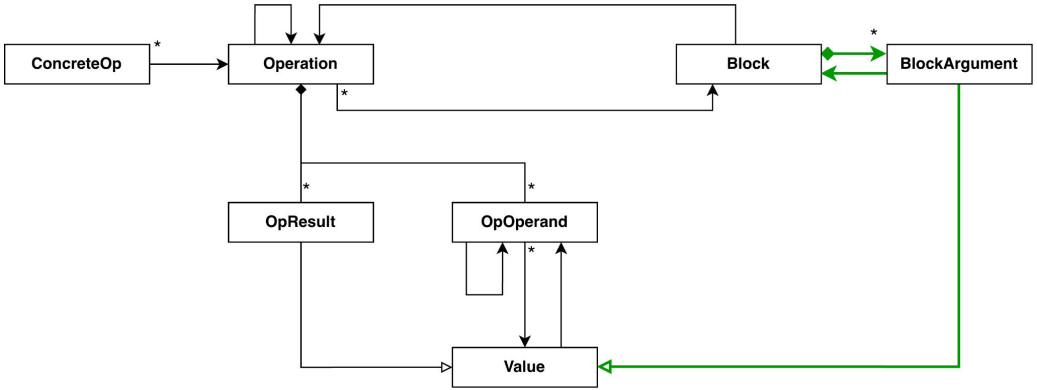
Quiz: What's Wrong Here?

```
/// Returns true iff the extension op is fed by a vector.transfer_read.
static bool isFedByTransferRead(arith::ExtUIOp extOp) {
  return isa_and_present<vector::TransferReadOp>(extOp.getOperand().getDefiningOp());
}
```



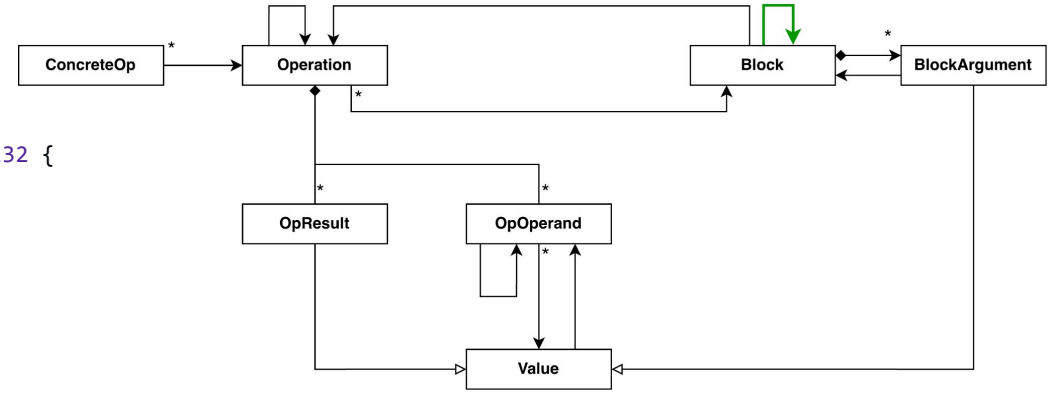
Quiz: What's Wrong Here?

```
/// Returns true iff the extension op is fed by a vector.transfer_read.
static bool isFedByTransferRead(arith::ExtUIOp extOp) {
  return static_cast<bool>(extOp.getOperand().getDefiningOp<vector::TransferReadOp>());
}
```



Query the Previous / Next Block

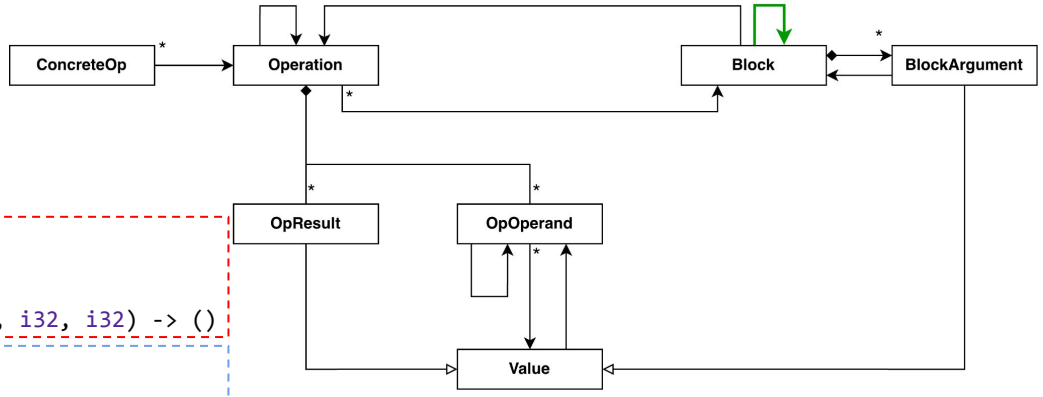
```
func.func @unstructed_cf(%c: i1, %a: i32, %b: i32) -> i32 {
  cf.cond_br %c, ^bb1(%a : i32), ^bb2(%b : i32)
^bb1(%arg0: i32):
  func.return %arg0 : i32
^bb2(%arg1: i32):
  %m = arith.addi %arg1, %arg1 : i32
  func.return %m : i32
}
```




```
Block *entryBlock = ...;
Block *nextBlock = entryBlock->getNextNode();
```

Blocks within a region form a doubly-linked list.

Query the Previous / Next Block



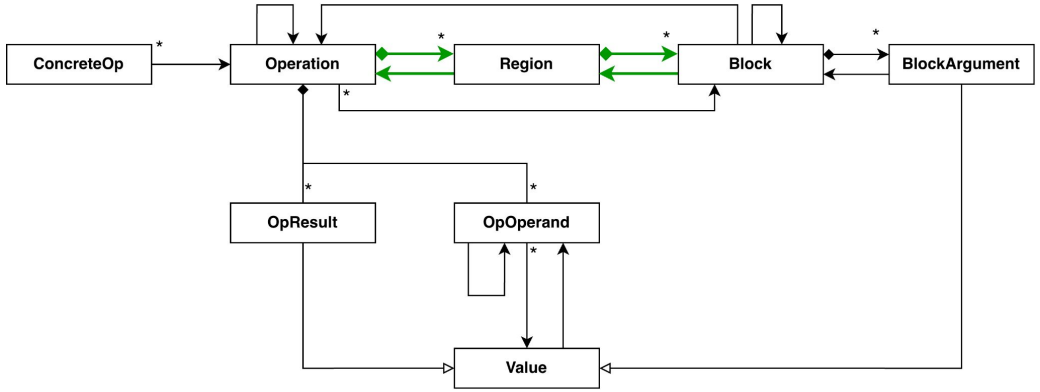

```
"func.func"() <{function_type = (i1, i32, i32) -> i32,
                sym_name = "unstructed_cf"}> ({
^bb0(%arg0: i1, %arg1: i32, %arg2: i32):
 "cf.cond_br"(%arg0, %arg1, %arg2)[^bb1, ^bb2]
    <{operandSegmentSizes = array<i32: 1, 1, 1>}> : (i1, i32, i32) -> ()
^bb1(%0: i32):  // pred: ^bb0
 "func.return"(%0) : (i32) -> ()
^bb2(%1: i32):  // pred: ^bb0
 %2 = "arith.addi"(%1, %1) <{overflowFlags = #arith.overflow<none>}> : (i32, i32) -> i32
 "func.return"(%2) : (i32) -> ()
}) : () -> ()
```

```
Block *entryBlock = ...;
Block *nextBlock = entryBlock->getNextNode();
```

Blocks within a region form a doubly-linked list.

Block are Inside of Regions

```
%r = scf.while (%arg1 = %arg0) : (f32) -> (f32) {
  %0 = "test.make_condition"() : () -> i1
  scf.condition(%0) %arg1 : f32
} do {
^bb0(%arg2: f32):
  scf.yield %arg2 : f32
}
```

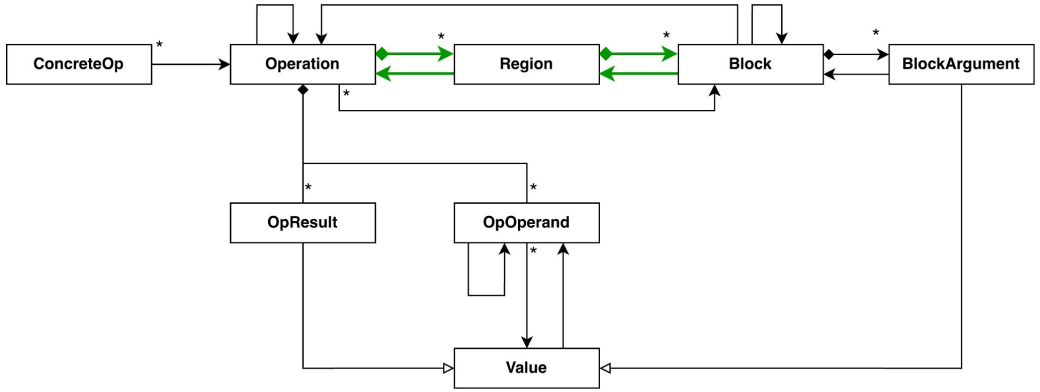



```
Operation *whileOp = …;
Region &beforeRegion = whileOp->getRegion(0);
assert(beforeRegion.getParentOp() == whileOp);
Block *beforeEntryBlock = &beforeRegion.front();
```

Blocks are for unstructured control flow (“goto” / jumps).
Regions are semantic grouping of operations.
A region maintains pointer to the first and last block.

Block are Inside of Regions

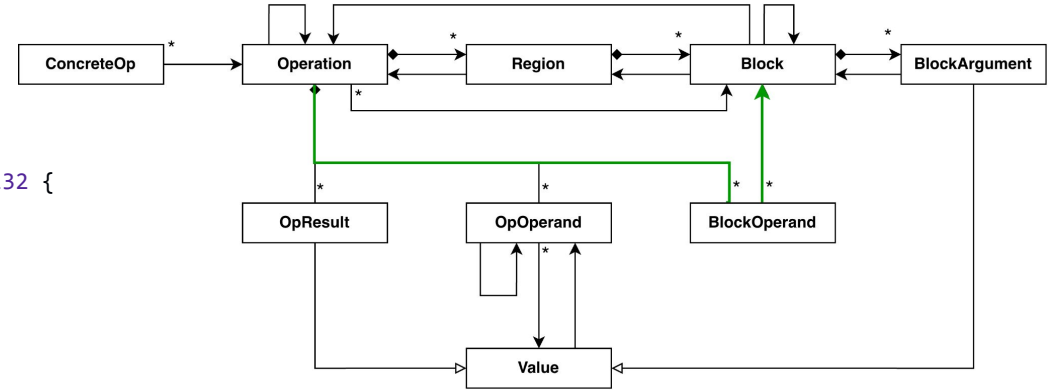
```
%r = "scf.while"(%arg0) (
  {
  ^bb0(%arg1: f32):
    %0 = "test.make_condition"() : () -> i1
    "scf.condition"(%0, %arg1) : (i1, f32) -> ()
  },
  {
  ^bb0(%arg2: f32):
    "scf.yield"(%arg2) : (f32) -> ()
  }
) : (f32) -> f32
```




```
Operation *whileOp = …;
Region &beforeRegion = whileOp->getRegion(0);
assert(beforeRegion.getParentOp() == whileOp);
Block *beforeEntryBlock = &beforeRegion.front();
```

Blocks are for unstructured control flow (“goto” / jumps).
Regions are semantic grouping of operations.
A region maintains pointer to the first and last block.

Querying Block Operands (Successors)



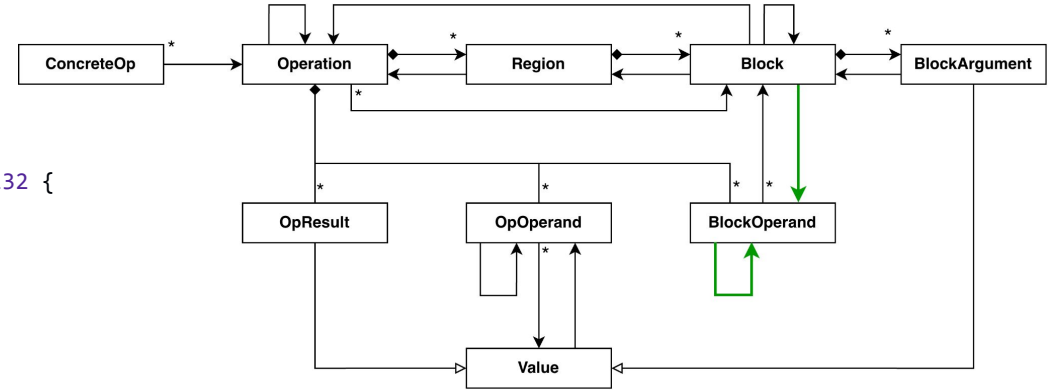

```
func.func @unstructed_cf(%c: i1, %a: i32, %b: i32) -> i32 {
  cf.cond_br %c, ^bb1(%a : i32), ^bb2(%b : i32)
^bb1(%arg0: i32):
  …
^bb2(%arg1: i32):
  …
}
```

```
Operation *condBrOp = …;
BlockOperand &blockOperand = condBrOp->getBlockOperands()[0];
Block *b = blockOperand.get();
assert(condBrOp.getSuccessor(0) == b);
```

BlockOperand = “use” of a Block.

Query the Uses of a Block

```
func.func @unstructed_cf(%c: i1, %a: i32, %b: i32) -> i32 {
  cf.cond_br %c, ^bb1(%a : i32), ^bb1(%b : i32)
^bb1(%arg0: i32):
  …
}
```

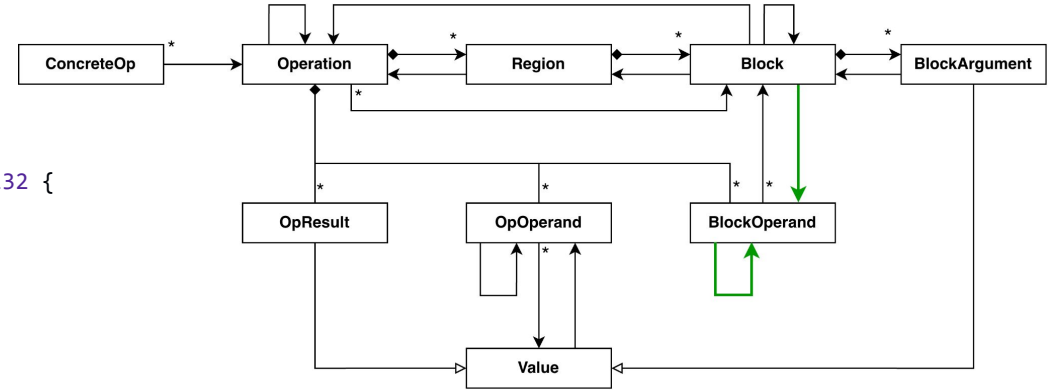


BlockOperand = “use” of a block
The uses of an block form a doubly-linked list.
Uses have no deterministic order!

```
Block *bb1 = …;
BlockOperand &firstUse = *bb1.getUses().begin();
BlockOperand &secondUse = *static_cast<BlockOperand *>(firstUse.getNextOperandUsingThisValue());
```

Query the Uses of a Block

```
func.func @unstructed_cf(%c: i1, %a: i32, %b: i32) -> i32 {
  cf.cond_br %c, ^bb1(%a : i32), ^bb1(%b : i32)
^bb1(%arg0: i32):
  …
}
```

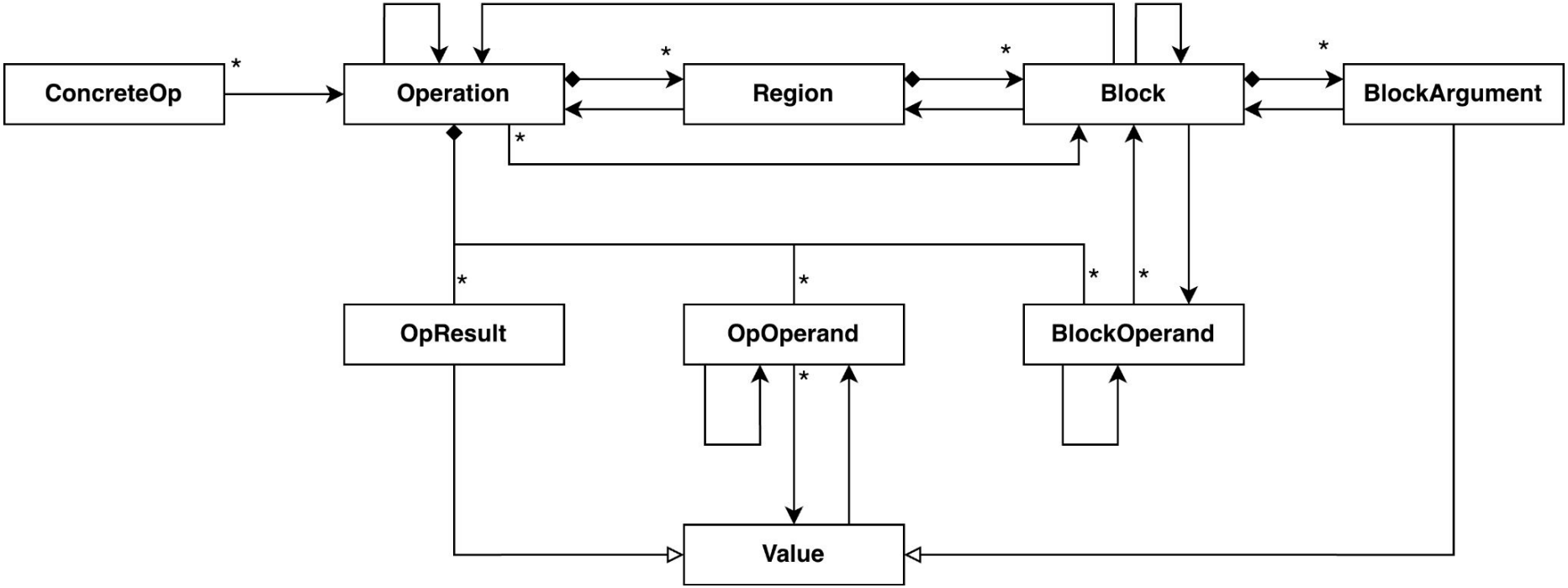



```
Block *bb1 = …;
for (BlockOperand &use : bb1.getUses()) { … }
for (Operation *user : bb1.getUsers()) { … }
```

BlockOperand = “use” of a block
The uses of an block form a doubly-linked list.
Uses have no deterministic order!
Be careful when modifying uses while iterating over them!

Summary: OpOperand / BlockOperand API

	OpOperand	BlockOperand
Get from Operation	`op->getOpOperand(i)` `addOp.getLhsMutable()`	`op->getBlockOperands()[i]`
Get Owning Operation	`oo.getOwner()`	`bo.getOwner()`
Get Operand Number	`oo.getOperandNumber()`	`bo.getOperandNumber()`
Get Referenced Thing	`oo.get(value)`	`bo.get(block)`
Set Referenced Thing	`oo.set(value)`	`bo.set(block)`
Get Referenced Thing from Op	`op->getOperand(i)`	`op->getSuccessor(i)`
Get all Uses	`value.getUses()`	`block.getUses()`
Get all Users	`value.getUsers()`	`block.getUsers()`



Linked lists for operations, block, uses (OpOperand / BlockOperand). The IR data structures are optimized for insertion / removal / movement of operations and blocks / traversal and modification of def-use chains. They are not optimized for random access by index or counting ops or blocks.

Types and Attributes

Types

- Compile-time constant metadata of SSA values.
- Singletons, uniqued in `MLIRContext`.

```
// Change the type of the arith.addi result.
Builder b(*ctx);
AddIOp addOp = …;
Value v = addOp.getResult();
v.setType(b.getI16Type());
```

```
func.func @test(%a: i32, %b: i32) -> i32 {
  // %0 = "arith.addi"(%a, %b) : (i32, i32) -> (i32)
  %0 = arith.addi %a, %b : i32
  // After: %0 = "arith.addi"(%a, %b) : (i32, i32) -> (i16)
  return %0 : i32
}
```

Quiz: How to Change the Entire IR to `i16`?

```
func.func @test(%a: i32, %b: i32) -> i32 {
  // %0 = "arith.addi"(%a, %b) : (i32, i32) -> (i32)
  %0 = arith.addi %a, %b : i32
  // After: %0 = "arith.addi"(%a, %b) : (i32, i32) -> (i16)
  return %0 : i32
}
```

Quiz: How to Change the Entire IR to `i16`?

```
func.func @test(%a: i32, %b: i32) -> i32 {
  // %0 = "arith.addi"(%a, %b) : (i32, i32) -> (i32)
  %0 = arith.addi %a, %b : i32
  // After: %0 = "arith.addi"(%a, %b) : (i32, i32) -> (i16)
  return %0 : i32
}
```

1. Change type of %a, %b (block arguments) to `i16`.
2. Change type of `%0` (op result) to `i16`.
3. Change function type (actually an attribute!) of `@test` to `(i16,i16)->i16`.

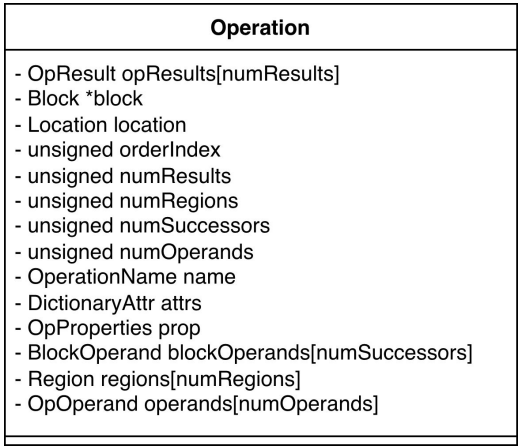
Quiz: How to Change the Entire IR to `i16`?

```
"func.func"() <{function_type = (i32, i32) -> i32, sym_name = "test"}> ({
^bb0(%arg0: i32, %arg1: i32):
  %0 = "arith.addi"(%arg0, %arg1) <{overflowFlags = #arith.overflow<none>}> : (i32, i32) -> i32
  "func.return"(%0) : (i32) -> ()
}) : () -> ()
```

1. Change type of %a, %b (block arguments) to `i16`.
2. Change type of `%0` (op result) to `i16`.
3. Change function type (actually an attribute!) of `@test` to `(i16,i16)->i16`.

Attributes

- Compile-time constant metadata of operations.
- Singletons, uniqued in `MLIRContext`.
- Every operation holds one `DictionaryAttr`, which consists of an array of `NamedAttribute` (non-uniqued). `NamedAttribute` consists of a key `StringAttr` and a value `Attribute`.




```
// Change the type of the arith.addi result.
Builder b(*ctx);
AddIOp addOp = …;
addOp->setAttr(“foo”, b.getIntegerAttr(b.getI64Type(), 42));
```

```
%0 = arith.addi %a, %b : i32
// After: %0 = arith.addi %a, %b {foo = 42} : i32
```

Quiz: How Expensive is Adding an Attribute?

```
addOp->setAttr(“foo”, b.getIntegerAttr(b.getI64Type(), 42));
```

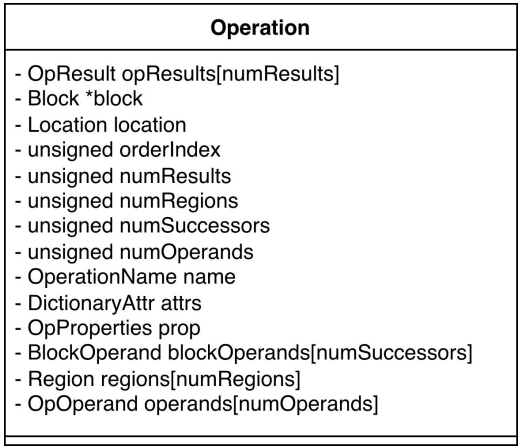
Quiz: How Expensive is Adding an Attribute?

```
addOp->setAttr(“foo”, b.getIntegerAttr(b.getI64Type(), 42));
```

1. Try to lookup existing `IntegerAttr(42 : i64)` in `MLIRContext` (hashing). If not found, allocate new memory and store it in the context.
2. Try to lookup existing `StringAttr(“foo”)` in the `MLIRContext` (hashing). If not found, allocate new memory and store it in the context.
3. Build array of new named attributes and try to lookup a corresponding `DictionaryAttr` in the `MLIRContext` (hashing every attribute!). If not found, allocate new memory and store it in the context. ⇒ O(num_attributes_of_op)
4. Set new attribute dictionary on the operation.
5. Accessing `MLIRContext` needs synchronization when running multi-threaded.

Properties

- Compile-time constant metadata of operations.
- Stored directly on the operation. No roundtrip to the `MLIRContext`.



Builder API

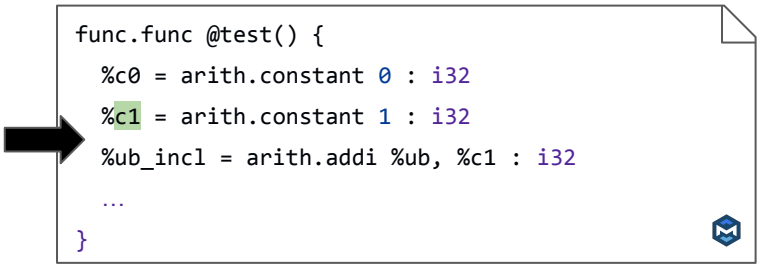
What's a (Op)Builder?

- A wrapper around `MLIRContext` with convenience API for creating attributes and types.
- Has an "insertion point": A cursor that point right before an operation.
- Operations that are created with an `OpBuilder` are inserted at the insertion point. (Block iterator / "before" operation pointer)
- A listener can be attached to a builder.

Creating + Inserting an Operation

AddIOp
- Operation *op
+ Value lhs() + Value rhs() + OpOperand &getLhsMutable() + OpOperand &getRhsMutable() + void build(...)

```
Value c0 = ..., c1 = ...;
Location loc = ...;

OpBuilder b(ctx);
b.setInsertionPointAfter(c1);
arith::AddIOp::create(builder, loc, c0, c1);
```



Quiz: What Would Happen If…

```
Value c0 = ..., c1 = ...;
Location loc = ...;

OpBuilder b(ctx);
b.setInsertionPoint(c1.getDefiningOp());
arith::AddIOp::create(builder, loc, c0, c1);
```

```
func.func @test() {
  %c0 = arith.constant 0 : i32
  %c1 = arith.constant 1 : i32
  ...
}
```

Quiz: What Would Happen If…

```
Value c0 = ..., c1 = ...;
Location loc = ...;

OpBuilder b(ctx);
b.setInsertionPoint(c1.getDefiningOp());
arith::AddIOp::create(builder, loc, c0, c1);
```

```
func.func @test() {
  %c0 = arith.constant 0 : i32
  %ub_incl = arith.addi %ub, %c1 : i32
  %c1 = arith.constant 1 : i32
}
```

Dominance violation!

Quiz: What Would Happen If…

```
Value c0 = ..., c1 = ...;
Location loc = ...;

OpBuilder b(ctx);
b.setInsertionPoint(c1.getDefiningOp());
arith::AddIOp::create(builder, loc, c0, c1);
```

```
func.func @test() {
  %c0 = arith.constant 0 : i32
  %c1 = arith.constant 1 : i32
  ...
}
```

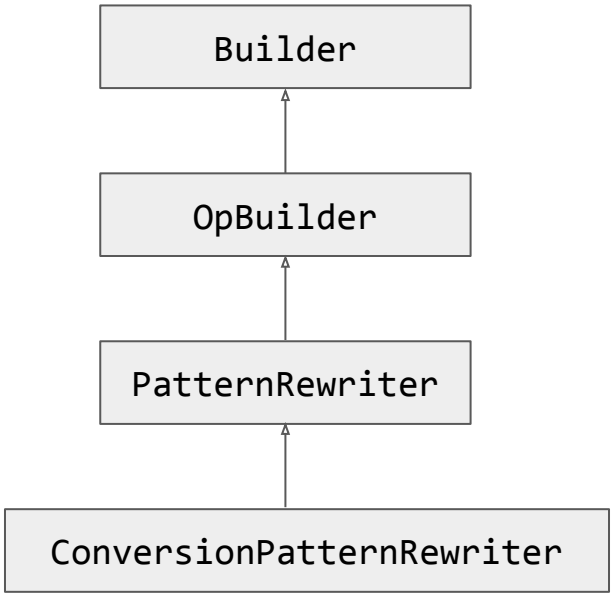
Quiz: What Would Happen If…

```
Value c0 = ..., c1 = ...;
Location loc = ...;

OpBuilder b(ctx);
b.setInsertionPoint(c1.getDefiningOp());
arith::AddIOp::create(builder, loc, c0, c1);
```

```
func.func @test() {
  %c0 = arith.constant 0 : i32
  %c1 = arith.constant 1 : i32
  …
}
```

Op is created, but not inserted.
(Same as `AddIOp::build`.)

Builders and Rewriters



Create Attributes / Types (has no insertion point)

Create Operations / Blocks

Modify IR

Dialect Conversion (not covered today)

There is More: Builders / Rewriter Convenience API

- `Operation *newOp = builder.clone(*op, mapping);`
- `rewriter.eraseOp(op);`
- `rewriter.replaceAllUsesWith(value, anotherValue);`
- `rewriter.replaceOp(op, anotherOp);`
- `rewriter.replaceOpWithNewOp<...>(...);`
- `rewriter.mergeBlock(srcBlock, destBlock, replValues);`
- `rewriter.inlineBlockBefore(...);`
- and many more

Pass Infrastructure

Pass Pipeline

- Traditional compiler = List of passes, applied back-to-back
- MLIR Pass: Piece of C++ code, scheduled on a certain kind of op.
- When there are multiple ops of that kind, the pass can run in parallel.

MLIR Pass: An Example

```
void MyPass::runOnOperation() {
  Operation *op = getOperation();
  op->emitRemark("pass scheduled on this op");

  // …

  if (failed) {
    op->emitError("pass failed");
    signalPassFailure();
  }
}
```

Debugging Infrastructure / Tips

`-mlir-print-ir-after-all`: Print IR after each pass application

`-mlir-print-ir-before-all`: Print IR before each pass application

There are many more convenience helpers for navigating / creating / modifying IR. See `Operation.h`, `Block.h`, `Region.h`, `Builders.h`, `PatternMatch.h`.

Dump IR entities: `op->dump()`, `value.dump()`, `type.dump()`, ...

Take a look at the auto-generated .h.inc files. E.g.:
`build/tutorial/list/ListTypes.h.inc`
`build/tutorial/list/ListOps.h.inc`.

Run with `--verify-each=0` to see unverified IR.

Resources

- Pass Infrastructure
- How Slow is MLIR?
- Understanding the IR Structure