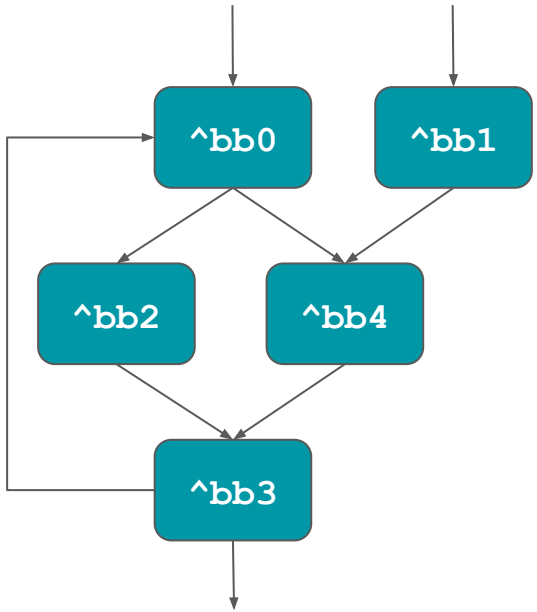
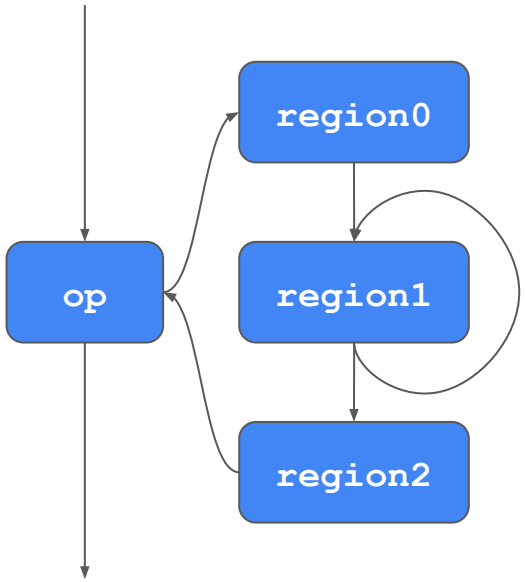
Multiple Exit MLIR Regions

Control Flow in MLIR

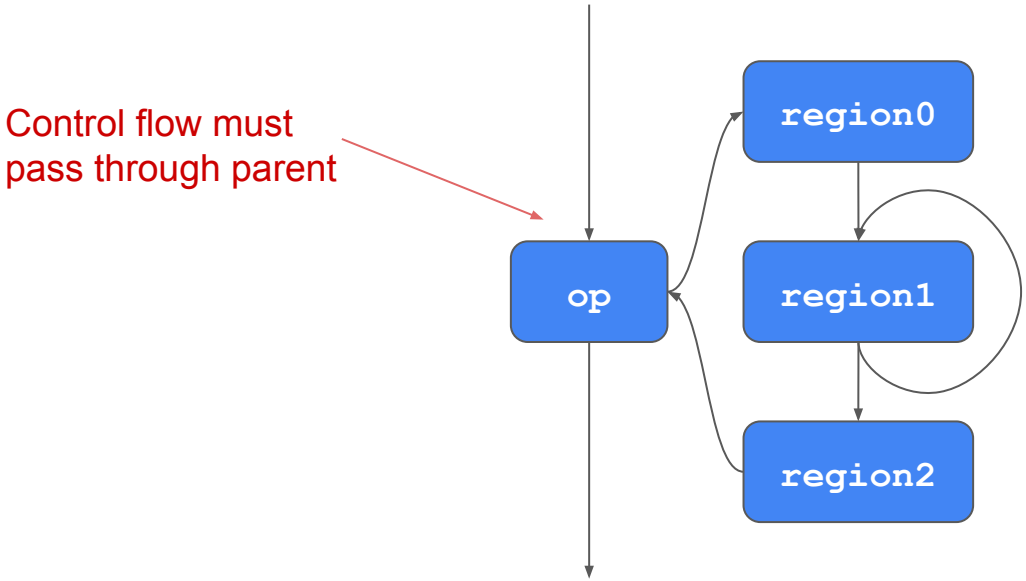


Basic blocks and CFGs



Region control-flow

Problem


Control flow must
pass through parent
op
region0
region1
region2

Problem

```
def all_true(values):
    for v in values:
        if not v:
            return False
    return True
```

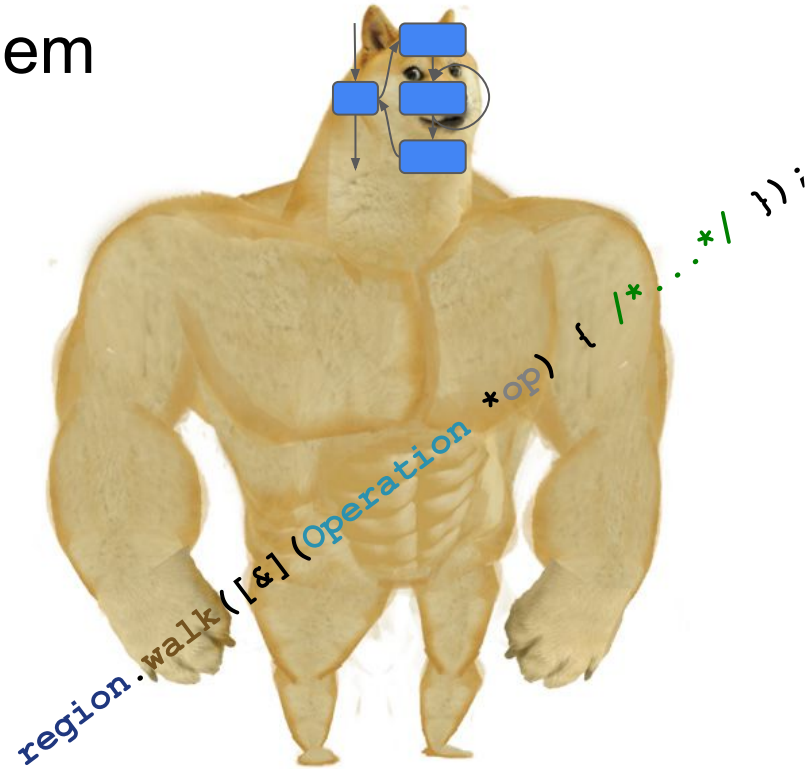
```
def add_ints(values):
    result = 0
    for v in values:
        if not isinstance(v, int):
            continue
        result += v
    return result
```

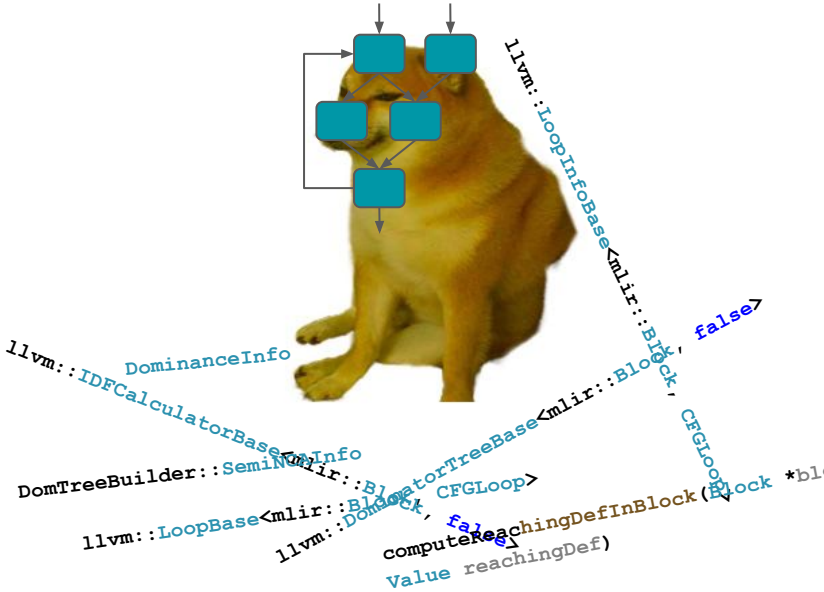
Problem

```
def all_true(values):
    for v in values:
        if not v:
            return False
    return True
```

```
def add_ints(values):
    result = 0
    for v in values:
        if not isinstance(v, int):
            continue
        result += v
    return result
```

Problem


region.walk([&](Operation *op) { /*...*/ });


llvm::IDFCalculatorBase<mlir::Block, false>
DominanceInfo
llvm::LoopInfoBase<mlir::Block, CFGLoop>
DomTreeBuilder::SemiNaveInfo
llvm::LoopBase<mlir::Block, CFGLoop>
llvm::DominatorTreeBase<mlir::Block, false>
computeReachingDefInBlock(Block *bl
Value reachingDef)

Solution?

```
def all_true(values):
    for v in values:
        if not v:
            return False
    return True
```

```
func @all_true(%values: ...) {

  %True = constant true
  return %True
}
```

Solution?

```
def all_true(values):
    for v in values:
        if not v:
            return False
    return True
```

```
func @all_true(%values: ...) {
  for %v in %values {

  %True = constant true
  return %True
}
```

Solution?

```
def all_true(values):
    for v in values:
        if not v:
            return False
    return True
```

```
func @all_true(%values: ...) {
  for %v in %values {
    %cond = ...
    if %cond {

  %True = constant true
  return %True
}
```

Solution?

```python
def all_true(values):
    for v in values:
        if not v:
            return False
    return True
```

```
func @all_true(%values: ...) {
  for %v in %values {
    %cond = ...
    if %cond {
      %False = constant false
      return %False
    }
    continue
  }
  %True = constant true
  return %True
}
```

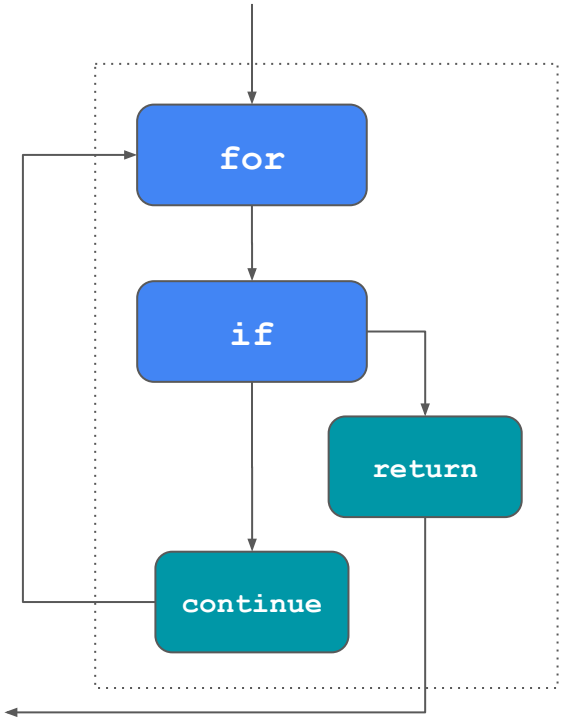
Modeling

- Two interfaces:
  - Control flow “nodes” (ops with regions)
  - Control flow terminators
- Nodes form a “control flow tree”
- Terminators may branch to the beginning of any ancestor region or after any ancestor node

Multiple exit regions

```
for %v in %values {
  %cond = ...
  if %cond {
    %False = constant false
    return %False
  }
  do_stuff()
  continue
}
```

Two exits from
the loop region



mem-2-reg

```python
def loop_stuff(k):
    s = 0
    for i in range(10, k):
        if i < s:
            break
        s += i
    return s
```

```
func @loop_stuff(%k: index) -> index {
  %s = alloca : index
  %i = alloca : index
  store 0, %s
  store 10, %i
  while {
    %0 = load %i
    %1 = lt %0, %k
    %2 = add 1, %0
    store %2, %i
    cond %1
  } do {
    %0 = load %i
    %1 = load %s
    %2 = lt %0, %1
    if %2 {
      break
    }
    %3 = add %0, %1
    store %3, %s
    yield
  }
  %0 = load %s
  return %0
}
```

```
func @loop_stuff(%k: index) -> index {
  %r:2 = while (%i = 10, %s = 0) {
    %0 = lt %i, %k
    %1 = add 1, %i
    cond %0 (%1, %s)
  } do (%i, %s) {
    %0 = lt %i, %s
    if %0 {
      break %i, %s
    }
    %1 = add %i, %s
    yield %i, %1
  }
  return %r#1
}
```

Function Splitting

```
async def coro_fn(t):
    s = 10
    if s < t
        s += await bar(s)
    return s
```

Walk

```
async.func @coro_fn(%t: index) -> index {
 %s = alloca : index
 store 10, %s
 %0 = load %s
 %1 = lt %0, %t
 if %1 {
   %2 = await @bar(%0)
   %3 = add %2, %0
   store %3, %s
 }
 %2 = load %s
 return %2
}
```

`s` lives

Check live variables

`s` dies

MLIR Language Reference

each block represents a compiler basic block where instructions inside the block are executed in order and **terminator operations implement control flow branches between basic blocks**

Consequences?

- Not much
  - Not enough generic MLIR transformation passes
- Control flow nodes are marked as having opaque memory effects*
  - This is a hack
- New region type?

How to model side effects?

```
store 1 -> v
conditional_exit
store 2 -> v
```

How to model side effects?

```
store 1 -> v
conditional_exit
store 2 -> v
```

Compute live-out variables

How to model side effects?

```
store 1 -> v
conditional_exit
store 2 -> v
```

Compute live-out variables

```
for ... {
  store 1, %s
  if %cond {
    break
  }
  %1 = load %s
}
%0 = load %s
```

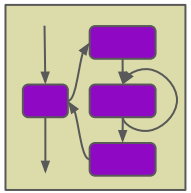
Can’t move after

{read %s}

Can move before

Questions?

THIS PRESENTATION MADE BY



REGION GANG