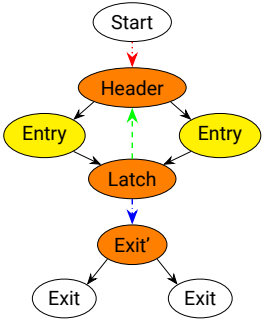
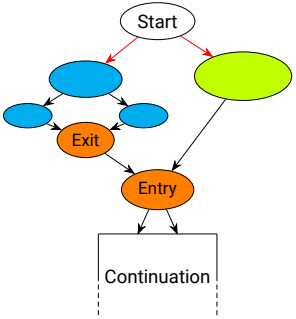
Lifting CFGs to Structured Control Flow in MLIR

Markus Böck
University of Cambridge





So you would like to do loop optimizations…

So you would like to do loop optimizations…

```
func @foo(%skip: i1, %init: f32) -> f32 {
 cf.cond_br %skip, ^bb1(%init : f32),
                   ^bb0(%init : f32)

^bb0(%iter: f32):
 %cond, %value = call @bar(%iter)
 cf.cond_br %cond, ^bb1(%iter : f32),
                   ^bb0(%value : f32)

^bb1(%res: f32):
 return %res : f32
}
```

but your IR looks like this

So you would like to do loop optimizations…

```
func @foo(%skip: i1, %init: f32) -> f32 {
 cf.cond_br %skip, ^bb1(%init : f32),
                   ^bb0(%init : f32)

^bb0(%iter: f32):
 %cond, %value = call @bar(%iter)
 cf.cond_br %cond, ^bb1(%iter : f32),
                   ^bb0(%value : f32)

^bb1(%res: f32):
 return %res : f32
}
```

but your IR looks like this

So you would like to do loop optimizations…

```
func @foo(%skip: i1, %init: f32) -> f32 {
 cf.cond_br %skip, ^bb1(%init : f32),
                   ^bb0(%init : f32)

^bb0(%iter: f32):
 %cond, %value = call @bar(%iter)
 cf.cond_br %cond, ^bb1(%iter : f32),
                   ^bb0(%value : f32)

^bb1(%res: f32):
 return %res : f32
}
```

but your IR looks like this

`llvm::LoopAnalysis`?

So you would like to do loop optimizations…

```
func @foo(%skip: i1, %init: f32) -> f32 {
 cf.cond_br %skip, ^bb1(%init : f32),
                   ^bb0(%init : f32)

^bb0(%iter: f32):
 %cond, %value = call @bar(%iter)
 cf.cond_br %cond, ^bb1(%iter : f32),
                   ^bb0(%value : f32)

^bb1(%res: f32):
 return %res : f32
}
```

but your IR looks like this

`llvm::LoopAnalysis`?

- Potentially invalidated analysis

So you would like to do loop optimizations…

```
func @foo(%skip: i1, %init: f32) -> f32 {
 cf.cond_br %skip, ^bb1(%init : f32),
                   ^bb0(%init : f32)

^bb0(%iter: f32):
 %cond, %value = call @bar(%iter)
 cf.cond_br %cond, ^bb1(%iter : f32),
                   ^bb0(%value : f32)

^bb1(%res: f32):
 return %res : f32
}
```

but your IR looks like this

`llvm::LoopAnalysis`?

- Potentially invalidated analysis
- Does not canonicalize loops

So you would like to do loop optimizations…

```
func @foo(%skip: i1, %init: f32) -> f32 {
 cf.cond_br %skip, ^bb1(%init : f32),
                   ^bb0(%init : f32)

^bb0(%iter: f32):
 %cond, %value = call @bar(%iter)
 cf.cond_br %cond, ^bb1(%iter : f32),
                   ^bb0(%value : f32)

^bb1(%res: f32):
 return %res : f32
}
```

but your IR looks like this

`llvm::LoopAnalysis`?

- Potentially invalidated analysis
- Does not canonicalize loops
- Transformation APIs inconvenient

What is Structured control flow

```
func @foo(%skip: i1, %init: f32)
     -> f32 {
  %true = arith.constant true
  %0 = scf.if %skip -> (f32) {
    scf.yield %init : f32
  } else {
    %1:2 = scf.while (%iter = %init) {
      %2:2 = func.call @bar(%iter)
      %3 = arith.xori %2#0, %true
      scf.condition(%3) %2#1, %iter
    } do {
    ^bb0(%n_iter: f32, %arg3: f32):
      scf.yield %n_iter : f32
    }
    scf.yield %1#1 : f32
  }
  return %0 : f32
}
```

What is Structured control flow

- Dedicated operations

```
func @foo(%skip: i1, %init: f32)
     -> f32 {
  %true = arith.constant true
  %0 = scf.if %skip -> (f32) {
    scf.yield %init : f32
  } else {
    %1:2 = scf.while (%iter = %init) {
      %2:2 = func.call @bar(%iter)
      %3 = arith.xori %2#0, %true
      scf.condition(%3) %2#1, %iter
    } do {
    ^bb0(%n_iter: f32, %arg3: f32):
      scf.yield %n_iter : f32
    }
    scf.yield %1#1 : f32
  }
  return %0 : f32
}
```

What is Structured control flow

- Dedicated operations

```
func @foo(%skip: i1, %init: f32)
     -> f32 {
  %true = arith.constant true
  %0 = scf.if %skip -> (f32) {
    scf.yield %init : f32
  } else {
    %1:2 = scf.while (%iter = %init) {
      %2:2 = func.call @bar(%iter)
      %3 = arith.xori %2#0, %true
      scf.condition(%3) %2#1, %iter
    } do {
    ^bb0(%n_iter: f32, %arg3: f32):
      scf.yield %n_iter : f32
    }
    scf.yield %1#1 : f32
  }
  return %0 : f32
}
```

What is Structured control flow

- Dedicated operations

```
func @foo(%skip: i1, %init: f32)
     -> f32 {
  %true = arith.constant true
  %0 = scf.if %skip -> (f32) {
    scf.yield %init : f32
  } else {
    %1:2 = scf.while (%iter = %init) {
      %2:2 = func.call @bar(%iter)
      %3 = arith.xori %2#0, %true
      scf.condition(%3) %2#1, %iter
    } do {
    ^bb0(%n_iter: f32, %arg3: f32):
      scf.yield %n_iter : f32
    }
    scf.yield %1#1 : f32
  }
  return %0 : f32
}
```

What is Structured control flow

- Dedicated operations
  - ⇒ Trivial traversal

```
func @foo(%skip: i1, %init: f32)
     -> f32 {
  %true = arith.constant true
  %0 = scf.if %skip -> (f32) {
    scf.yield %init : f32
  } else {
    %1:2 = scf.while (%iter = %init) {
      %2:2 = func.call @bar(%iter)
      %3 = arith.xori %2#0, %true
      scf.condition(%3) %2#1, %iter
    } do {
    ^bb0(%n_iter: f32, %arg3: f32):
      scf.yield %n_iter : f32
    }
    scf.yield %1#1 : f32
  }
  return %0 : f32
}
```

What is Structured control flow

- Dedicated operations
  - ⇒ Trivial traversal
  - ⇒ Trivial “contains”

```
func @foo(%skip: i1, %init: f32)
     -> f32 {
  %true = arith.constant true
  %0 = scf.if %skip -> (f32) {
    scf.yield %init : f32
  } else {
    %1:2 = scf.while (%iter = %init) {
      %2:2 = func.call @bar(%iter)
      %3 = arith.xori %2#0, %true
      scf.condition(%3) %2#1, %iter
    } do {
    ^bb0(%n_iter: f32, %arg3: f32):
      scf.yield %n_iter : f32
    }
    scf.yield %1#1 : f32
  }
  return %0 : f32
}
```

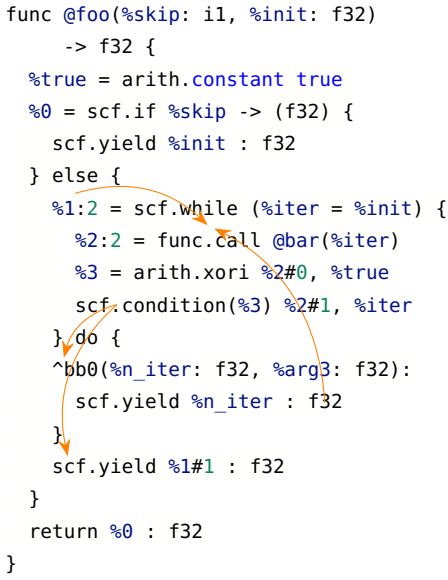
What is Structured control flow

- Dedicated operations
  - ⇒ Trivial traversal
  - ⇒ Trivial “contains”
- Region Successors
  - Between regions
  - From and to parent region

```
func @foo(%skip: i1, %init: f32)
     -> f32 {
  %true = arith.constant true
  %0 = scf.if %skip -> (f32) {
    scf.yield %init : f32
  } else {
    %1:2 = scf.while (%iter = %init) {
      %2:2 = func.call @bar(%iter)
      %3 = arith.xori %2#0, %true
      scf.condition(%3) %2#1, %iter
    } do {
    ^bb0(%n_iter: f32, %arg3: f32):
      scf.yield %n_iter : f32
    }
    scf.yield %1#1 : f32
  }
  return %0 : f32
}
```

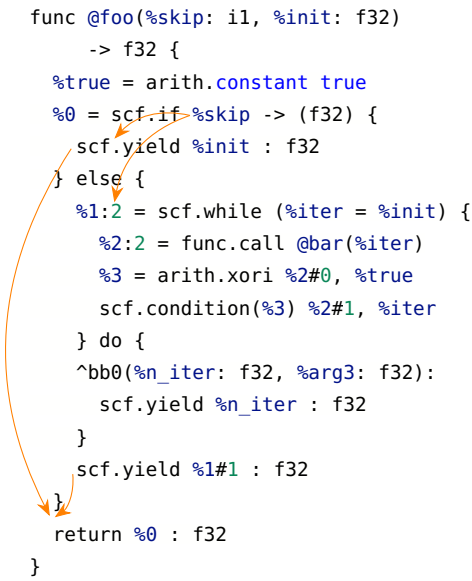
What is Structured control flow

- Dedicated operations
  - ⇒ Trivial traversal
  - ⇒ Trivial “contains”
- Region Successors
  - Between regions
  - From and to parent region



What is Structured control flow

- Dedicated operations
  - ⇒ Trivial traversal
  - ⇒ Trivial “contains”
- Region Successors
  - Between regions
  - From and to parent region



What is Structured control flow

- Dedicated operations
  - ⇒ Trivial traversal
  - ⇒ Trivial “contains”
- Region Successors
  - Between regions
  - From and to parent region
- Single-Entry Single-Exit
  - ⇒ Trivial (post)dominance

```
func @foo(%skip: i1, %init: f32)
     -> f32 {
  %true = arith.constant true
  %0 = scf.if %skip -> (f32) {
    scf.yield %init : f32
  } else {
    %1:2 = scf.while (%iter = %init) {
      %2:2 = func.call @bar(%iter)
      %3 = arith.xori %2#0, %true
      scf.condition(%3) %2#1, %iter
    } do {
    ^bb0(%n_iter: f32, %arg3: f32):
      scf.yield %n_iter : f32
    }
    scf.yield %1#1 : f32
  }
  return %0 : f32
}
```

What is Structured control flow

- Dedicated operations
  - ⇒ Trivial traversal
  - ⇒ Trivial “contains”
- Region Successors
  - Between regions
  - From and to parent region
- Single-Entry Single-Exit
  - ⇒ Trivial (post)dominance
- Single back-edge

```
func @foo(%skip: i1, %init: f32)
     -> f32 {
  %true = arith.constant true
  %0 = scf.if %skip -> (f32) {
    scf.yield %init : f32
  } else {
    %1:2 = scf.while (%iter = %init) {
      %2:2 = func.call @bar(%iter)
      %3 = arith.xori %2#0, %true
      scf.condition(%3) %2#1, %iter
    } do {
    ^bb0(%n_iter: f32, %arg3: f32):
      scf.yield %n_iter : f32
    }
    scf.yield %1#1 : f32
  }
  return %0 : f32
}
```

What is Structured control flow

- Dedicated operations
  - ⇒ Trivial traversal
  - ⇒ Trivial “contains”
- Region Successors
  - Between regions
  - From and to parent region
- Single-Entry Single-Exit
  - ⇒ Trivial (post)dominance
- Single back-edge
- Dataflow through:
  - Region dominance

```
func @foo(%skip: i1, %init: f32)
     -> f32 {
  %true = arith.constant true
  %0 = scf.if %skip -> (f32) {
    scf.yield %init : f32
  } else {
    %1:2 = scf.while (%iter = %init) {
      %2:2 = func.call @bar(%iter)
      %3 = arith.xori %2#0, %true
      scf.condition(%3) %2#1, %iter
    } do {
    ^bb0(%n_iter: f32, %arg3: f32):
      scf.yield %n_iter : f32
    }
    scf.yield %1#1 : f32
  }
  return %0 : f32
}
```

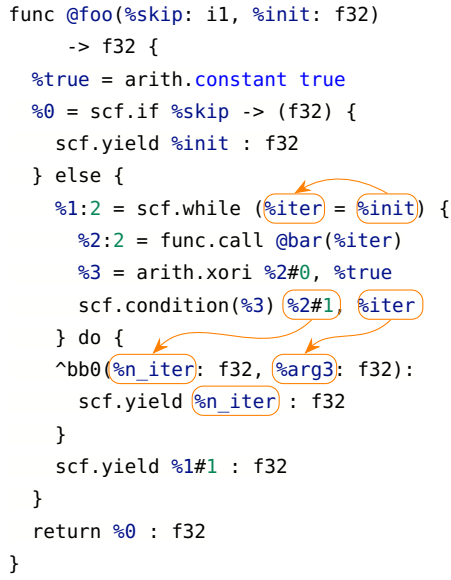
What is Structured control flow

- Dedicated operations
  - ⇒ Trivial traversal
  - ⇒ Trivial “contains”
- Region Successors
  - Between regions
  - From and to parent region
- Single-Entry Single-Exit
  - ⇒ Trivial (post)dominance
- Single back-edge
- Dataflow through:
  - Region dominance

```
func @foo(%skip: i1, %init: f32)
     -> f32 {
  %true = arith.constant true
  %0 = scf.if %skip -> (f32) {
    scf.yield %init : f32
  } else {
    %1:2 = scf.while (%iter = %init) {
      %2:2 = func.call @bar(%iter)
      %3 = arith.xori %2#0, %true
      scf.condition(%3) %2#1, %iter
    } do {
    ^bb0(%n_iter: f32, %arg3: f32):
      scf.yield %n_iter : f32
    }
    scf.yield %1#1 : f32
  }
  return %0 : f32
}
```

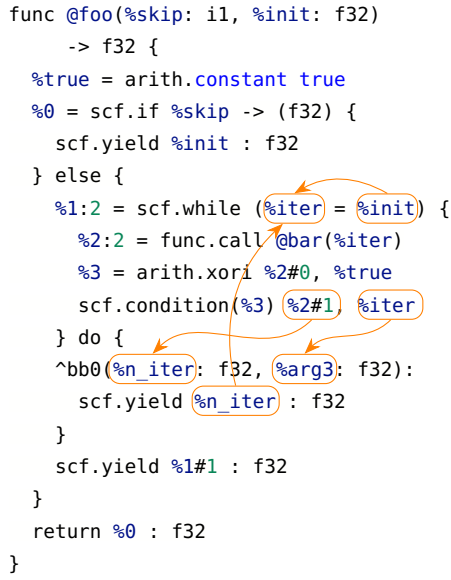
What is Structured control flow

- Dedicated operations
  - ⇒ Trivial traversal
  - ⇒ Trivial “contains”
- Region Successors
  - Between regions
  - From and to parent region
- Single-Entry Single-Exit
  - ⇒ Trivial (post)dominance
- Single back-edge
- Dataflow through:
  - Region dominance
  - Explicit arguments/operands/results



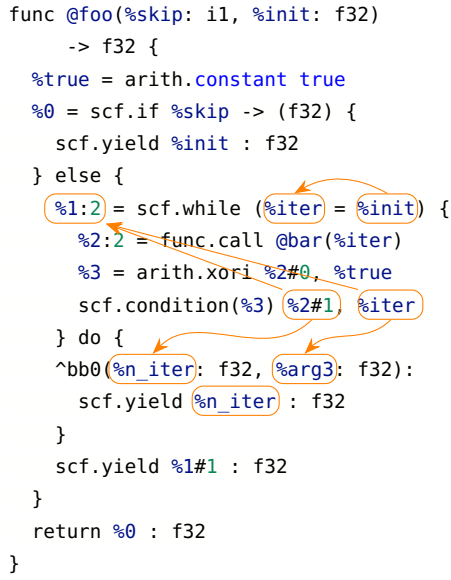
What is Structured control flow

- Dedicated operations
  - ⇒ Trivial traversal
  - ⇒ Trivial “contains”
- Region Successors
  - Between regions
  - From and to parent region
- Single-Entry Single-Exit
  - ⇒ Trivial (post)dominance
- Single back-edge
- Dataflow through:
  - Region dominance
  - Explicit arguments/operands/results



What is Structured control flow

- Dedicated operations
  - ⇒ Trivial traversal
  - ⇒ Trivial “contains”
- Region Successors
  - Between regions
  - From and to parent region
- Single-Entry Single-Exit
  - ⇒ Trivial (post)dominance
- Single back-edge
- Dataflow through:
  - Region dominance
  - Explicit arguments/operands/results



What is Structured control flow

```
class WhileOp : public Op<WhileOp, ...> {
public:
  …
  Operation::operand_range getInits();
  MutableOperandRange getInitsMutable();
  Operation::result_range getResults();

  ConditionOp getConditionOp();
  YieldOp getYieldOp();

 Block::BlockArgListType getBeforeArguments();
 Block::BlockArgListType getAfterArguments();
};
```

- Explicit arguments/operands/results

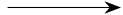
```
func @foo(%skip: i1, %init: f32)
    -> f32 {
  %true = arith.constant true
  %0 = scf.if %skip -> (f32) {
    scf.yield %init : f32
  } else {
   %1:2 = scf.while (%iter = %init) {
     %2:2 = func.call @bar(%iter)
     %3 = arith.xori %2#0, %true
     scf.condition(%3) %2#1, %iter
    } do {
    ^bb0(%n_iter: f32, %arg3: f32):
     scf.yield %n_iter : f32
    }
    scf.yield %1#1 : f32
  }
  return %0 : f32
}
```

What we want

```
func @foo(%skip: i1, %init: f32) -> f32 {
 cf.cond_br %skip, ^bb1(%init : f32),
                   ^bb0(%init : f32)

^bb0(%iter: f32):
 %cond, %value = call @bar(%iter)
 cf.cond_br %cond, ^bb1(%iter : f32),
                   ^bb0(%value : f32)

^bb1(%res: f32):
 return %res : f32
}
```

What we want

```
func @foo(%skip: i1, %init: f32) -> f32 {
 cf.cond_br %skip, ^bb1(%init : f32),
                   ^bb0(%init : f32)

^bb0(%iter: f32):
 %cond, %value = call @bar(%iter)
 cf.cond_br %cond, ^bb1(%iter : f32),
                   ^bb0(%value : f32)

^bb1(%res: f32):
 return %res : f32
}
```



```
func @foo(%skip: i1, %init: f32)
     -> f32 {
  %true = arith.constant true
  %0 = scf.if %skip -> (f32) {
    scf.yield %init : f32
  } else {
    %1:2 = scf.while (%iter = %init) {
      %2:2 = func.call @bar(%iter)
      %3 = arith.xori %2#0, %true
      scf.condition(%3) %2#1, %iter
    } do {
    ^bb0(%n_iter: f32, %arg3: f32):
      scf.yield %n_iter : f32
    }
    scf.yield %1#1 : f32
  }
  return %0 : f32
}
```

Why Control Flow Graphs as Inputs

Why Control Flow Graphs as Inputs

- Input is a CFG:
  - LLVM IR, SPIR-V,
    CPython Bytecode, JVM Bytecode etc.

Why Control Flow Graphs as Inputs

- Input is a CFG:
  - LLVM IR, SPIR-V,
    CPython Bytecode, JVM Bytecode etc.
- Input easier to lower to CFGs

Why Control Flow Graphs as Inputs

- Input is a CFG:
  - LLVM IR, SPIR-V,
    CPython Bytecode, JVM Bytecode etc.
- Input easier to lower to CFGs
  - Lexical nesting ≠ Control flow nesting

Why Control Flow Graphs as Inputs

- Input is a CFG:
  - LLVM IR, SPIR-V,
    CPython Bytecode, JVM Bytecode etc.
- Input easier to lower to CFGs
  - Lexical nesting ≠ Control flow nesting

```
for (int i = 0; i < n; i++) {
  if (enough(value[i])) {
    outOfLoopAction();
    break;
  }

  if (!value[i]) {
    action();
    continue;
  }

  if (fits(value[i])) {
    outOfLoopAction();
    return value[i];
  }
}
```

Why Control Flow Graphs as Inputs

- Input is a CFG:
  - LLVM IR, SPIR-V,
    CPython Bytecode, JVM Bytecode etc.
- Input easier to lower to CFGs
  - Lexical nesting ≠ Control flow nesting
    - Multiple nested exits

```
for (int i = 0; i < n; i++) {
  if (enough(value[i])) {
    outOfLoopAction();
    break;
  }

  if (!value[i]) {
    action();
    continue;
  }

  if (fits(value[i])) {
    outOfLoopAction();
    return value[i];
  }
}
```

Why Control Flow Graphs as Inputs

- Input is a CFG:
  - LLVM IR, SPIR-V,
    CPython Bytecode, JVM Bytecode etc.
- Input easier to lower to CFGs
  - Lexical nesting ≠ Control flow nesting
    - Multiple nested exits
    - Multiple back edges

```
for (int i = 0; i < n; i++) {
  if (enough(value[i])) {
    outOfLoopAction();
    break;
  }

  if (!value[i]) {
    action();
    continue;
  }

  if (fits(value[i])) {
    outOfLoopAction();
    return value[i];
  }
}
```

Why Control Flow Graphs as Inputs

- Input is a CFG:
  - LLVM IR, SPIR-V,
    CPython Bytecode, JVM Bytecode etc.
- Input easier to lower to CFGs
  - Lexical nesting ≠ Control flow nesting
    - Multiple nested exits
    - Multiple back edges
    - Multiple loop conditions

```
for (int i = 0; i < n; i++) {
  if (enough(value[i])) {
    outOfLoopAction();
    break;
  }

  if (!value[i]) {
    action();
    continue;
  }

  if (fits(value[i])) {
    outOfLoopAction();
    return value[i];
  }
}
```

Why Control Flow Graphs as Inputs

- Input is a CFG:
  - LLVM IR, SPIR-V,
    CPython Bytecode, JVM Bytecode etc.
- Input easier to lower to CFGs
  - Lexical nesting ≠ Control flow nesting
    - Multiple nested exits
    - Multiple back edges
    - Multiple loop conditions
    - `goto` 😢

```
for (int i = 0; i < n; i++) {
  if (enough(value[i])) {
    outOfLoopAction();
    break;
  }

  if (!value[i]) {
    action();
    continue;
  }

  if (fits(value[i])) {
    outOfLoopAction();
    return value[i];
  }
}
```

Implementation

Perfect Reconstructability of Control Flow from Demand Dependence Graphs

HELGE BAHMANN, Google Zürich
NICO REISSMANN, MAGNUS JAHRE, and JAN CHRISTIAN MEYER,
Norwegian University of Science and Technology

Implementation

- No code duplication

Perfect Reconstructability of Control Flow from Demand Dependence Graphs

HELGE BAHMANN, Google Zürich
NICO REISSMANN, MAGNUS JAHRE, and JAN CHRISTIAN MEYER,
Norwegian University of Science and Technology

Implementation

- No code duplication
- Arbitrary control flow

Perfect Reconstructability of Control Flow from Demand Dependence Graphs

HELGE BAHMANN, Google Zürich
NICO REISSMANN, MAGNUS JAHRE, and JAN CHRISTIAN MEYER,
Norwegian University of Science and Technology

Implementation

- No code duplication
- Arbitrary control flow
- Dialect agnostic

Perfect Reconstructability of Control Flow from Demand Dependence Graphs

HELGE BAHMANN, Google Zürich
NICO REISSMANN, MAGNUS JAHRE, and JAN CHRISTIAN MEYER,
Norwegian University of Science and Technology

Implementation

- No code duplication
- Arbitrary control flow
- Dialect agnostic
- Upstream as driver and `--lift-cf-to-scf`

Perfect Reconstructability of Control Flow from Demand Dependence Graphs

HELGE BAHMANN, Google Zürich
NICO REISSMANN, MAGNUS JAHRE, and JAN CHRISTIAN MEYER,
Norwegian University of Science and Technology

Implementation

Perfect Reconstructability of Control Flow from Demand Dependence Graphs

HELGE BAHMANN, Google Zürich
NICO REISSMANN, MAGNUS JAHRE, and JAN CHRISTIAN MEYER,
Norwegian University of Science and Technology

- No code duplication
- Arbitrary control flow
- Dialect agnostic
- Upstream as driver and `--lift-cf-to-scf`
- Paper extended to handle:
  - Block arguments
  - Multiple Return-like operations

Top-level loop

```
def cfg_to_scf(region):
  consolidate_return_likes(region)
  worklist: list[BasicBlock] = [region.entry]
  while len(worklist) != 0:
    start_block = worklist.pop_back()
    assert dominates_all_successors(start_block)

    # Step 1: Cycles → do-while ops.
    worklist += transform_cycles_to_do_while(start_block)
    assert is_dag(start_block)
    # Step 2: Handling branches.
    worklist += transform_branches(start_block)
```

Return-likes

```
func.func @multi_return() -> i32 {
  %cond = "test.test1"() : () -> i1
  cf.cond_br %cond, ^bb1, ^bb3
^bb1:
  %0 = "test.test2"() : () -> i32
  return %0 : i32
^bb3:
  %1 = "test.test4"() : () -> i32
  return %1 : i32
}
```

Return-likes

```
func.func @multi_return() -> i32 {
  %cond = "test.test1"() : () -> i1
  cf.cond_br %cond, ^bb1, ^bb3
^bb1:
  %0 = "test.test2"() : () -> i32
  return %0 : i32
^bb3:
  %1 = "test.test4"() : () -> i32
  return %1 : i32
}
```

```
func.func @multi_return() -> i32 {
  %cond = "test.test1"() : () -> i1
  cf.cond_br %cond, ^bb1, ^bb3
^bb1:
  %0 = "test.test2"() : () -> i32
  cf.br ^bb4(%0 : i32)
^bb3:
  %1 = "test.test4"() : () -> i32
  cf.br ^bb4(%1 : i32)
^bb4(%arg0 : i32):
  return %arg0 : i32
}
```

Return-likes

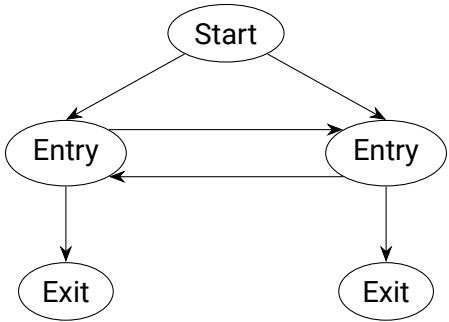
```
func.func @multi_return_likes() -> i32 {
  %cond = "test.test1"() : () -> i1
  cf.cond_br %cond, ^bb1, ^bb3
^bb1:
  %0 = "test.test2"() : () -> i32
  exc.raise
^bb3:
  %1 = "test.test4"() : () -> i32
  return %1 : i32
}
```

Return-likes

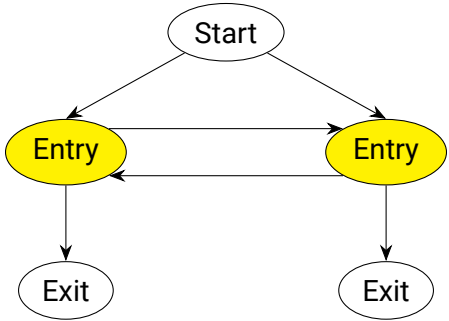
```
func.func @multi_return_likes() -> i32 {
  %cond = "test.test1"() : () -> i1
  cf.cond_br %cond, ^bb1, ^bb3
^bb1:
  %0 = "test.test2"() : () -> i32
  exc.raise
^bb3:
  %1 = "test.test4"() : () -> i32
  return %1 : i32
}
```

- Single control flow op remains
- Always top-level
- Only such case

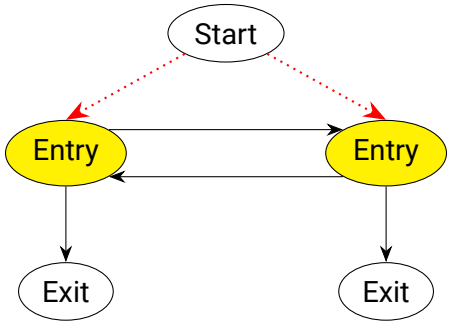
Handling Cycles



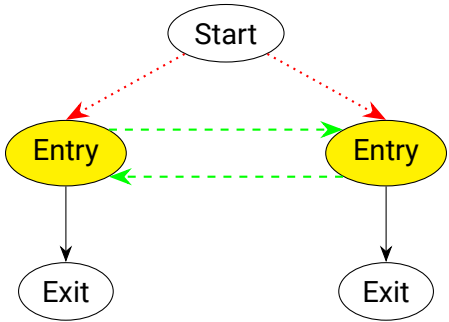
Handling Cycles



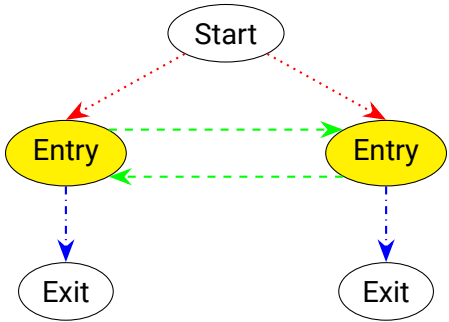
Handling Cycles



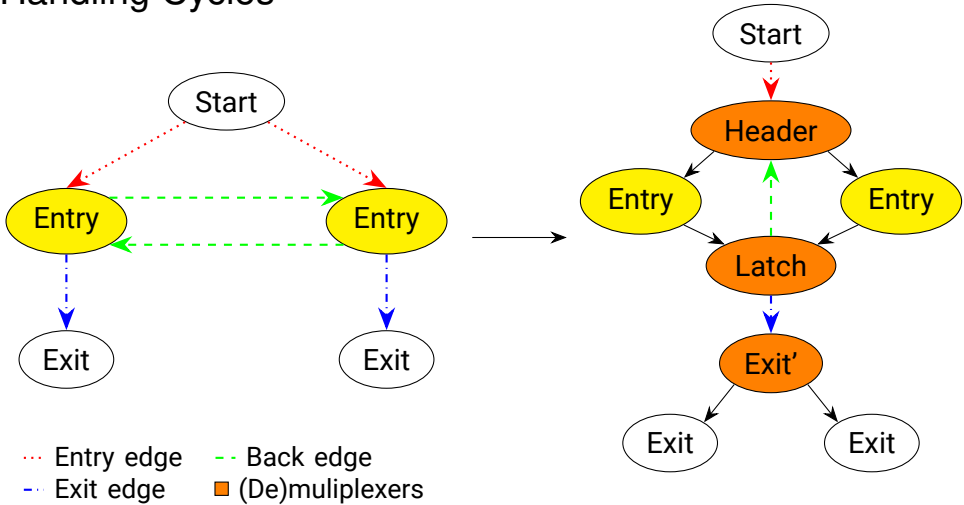
Handling Cycles



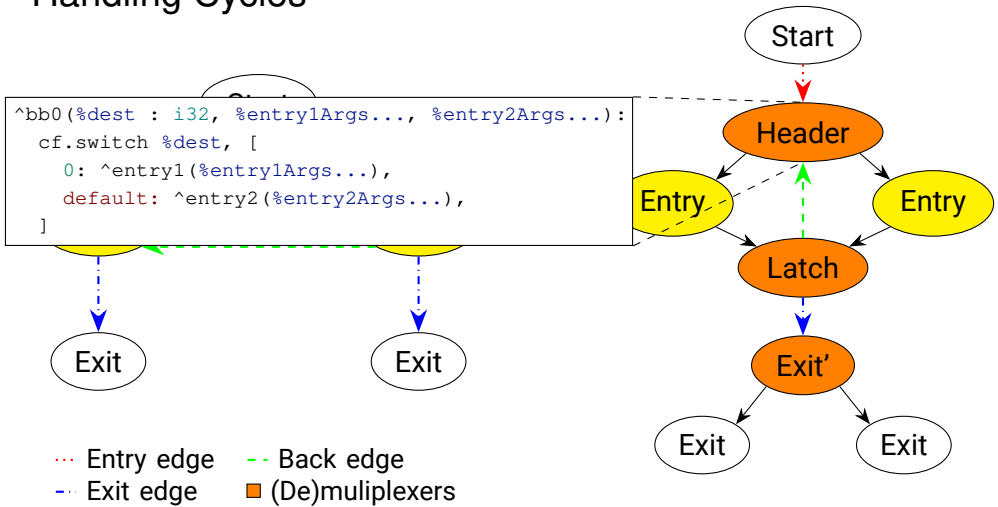
Handling Cycles



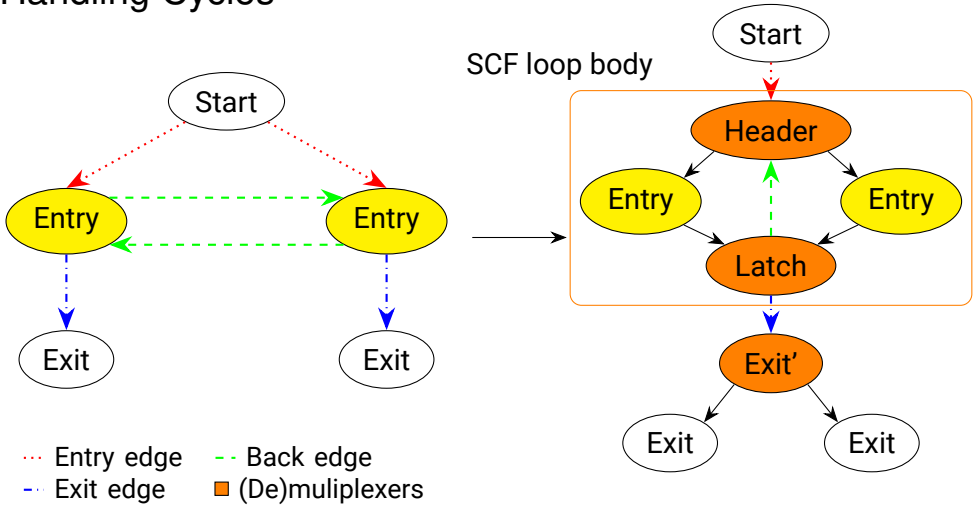
Handling Cycles



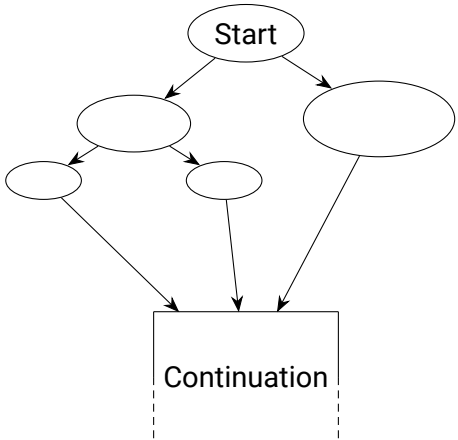
Handling Cycles



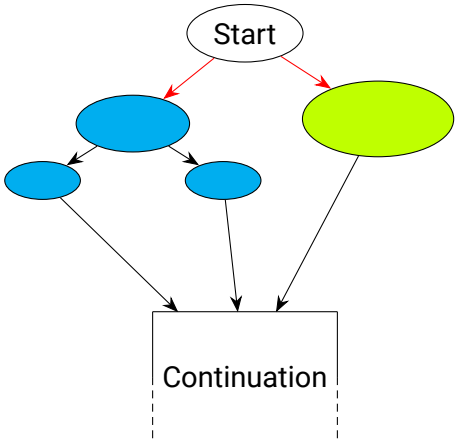
Handling Cycles



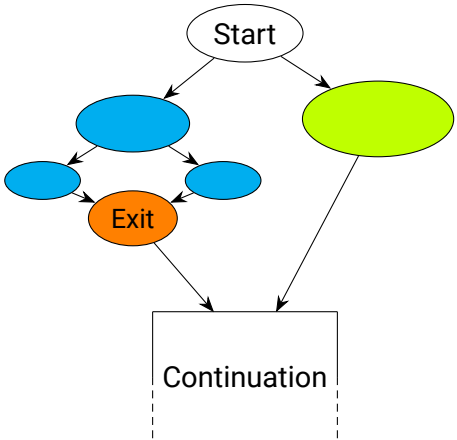
Handling branches



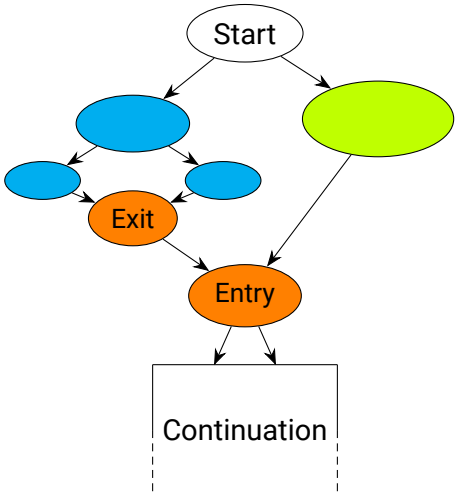
Handling branches



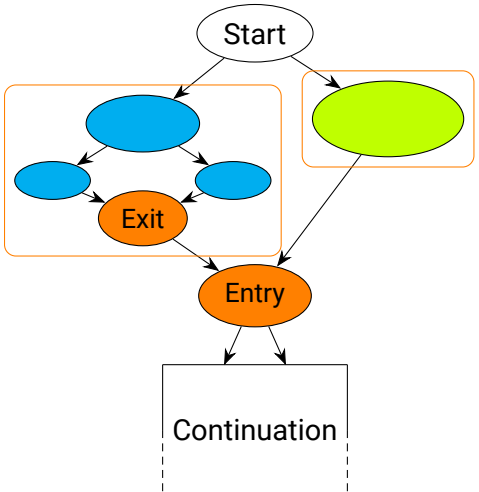
Handling branches



Handling branches



Handling branches



Results

```
func.func @simple_if() {
  %cond = "test.test1"() : () -> i1
  cf.cond_br %cond, ^bb1, ^bb2
^bb1:
  "test.test2"() : () -> ()
  cf.br ^bb3
^bb2:
  "test.test3"() : () -> ()
  cf.br ^bb3
^bb3:
  "test.test4"() : () -> ()
  return
}
```

Results

```
func.func @simple_if() {
  %cond = "test.test1"() : () -> i1
  cf.cond_br %cond, ^bb1, ^bb2
^bb1:
  "test.test2"() : () -> ()
  cf.br ^bb3
^bb2:
  "test.test3"() : () -> ()
  cf.br ^bb3
^bb3:
  "test.test4"() : () -> ()
  return
}
```

```
func.func @simple_if() {
  %0 = "test.test1"() : () -> i1
  scf.if %0 {
    "test.test2"() : () -> ()
  } else {
    "test.test3"() : () -> ()
  }
  "test.test4"() : () -> ()
  return
}
```

Results

```
func.func @if_with_block_args() -> index {
  %cond = "test.test1"() : () -> i1
  cf.cond_br %cond, ^bb1, ^bb2
^bb1:
  %1 = "test.test2"() : () -> (index)
  cf.br ^bb3(%1: index)
^bb2:
  %2 = "test.test3"() : () -> (index)
  cf.br ^bb3(%2: index)
^bb3(%3: index):
  "test.test4"() : () -> ()
  return %3 : index
}
```

Results

```
func.func @if_with_block_args() -> index {
  %cond = "test.test1"() : () -> i1
  cf.cond_br %cond, ^bb1, ^bb2
^bb1:
  %1 = "test.test2"() : () -> (index)
  cf.br ^bb3(%1: index)
^bb2:
  %2 = "test.test3"() : () -> (index)
  cf.br ^bb3(%2: index)
^bb3(%3: index):
  "test.test4"() : () -> ()
  return %3 : index
}
```

```
func.func @if_with_block_args() -> index {
  %0 = "test.test1"() : () -> i1
  %1 = scf.if %0 -> (index) {
    %2 = "test.test2"() : () -> index
    scf.yield %2 : index
  } else {
    %2 = "test.test3"() : () -> index
    scf.yield %2 : index
  }
  "test.test4"() : () -> ()
  return %1 : index
}
```

Results

```
func.func @while_loop() {
  "test.test1"() : () -> ()
  cf.br ^bb1
^bb1:
  %cond = "test.test2"() : () -> i1
  cf.cond_br %cond, ^bb2, ^bb3
^bb2:
  "test.test3"() : () -> ()
  cf.br ^bb1
^bb3:
  "test.test4"() : () -> ()
  return
}
```

Results

```
func.func @while_loop() {
  "test.test1"() : () -> ()
  cf.br ^bb1
^bb1:
  %cond = "test.test2"() : () -> i1
  cf.cond_br %cond, ^bb2, ^bb3
^bb2:
  "test.test3"() : () -> ()
  cf.br ^bb1
^bb3:
  "test.test4"() : () -> ()
  return
}
```

```
func.func @while_loop() {
  %c1_i32 = arith.constant 1 : i32
  %c0_i32 = arith.constant 0 : i32
  "test.test1"() : () -> ()
  scf.while : () -> () {
    %0 = "test.test2"() : () -> i1
    %1:2 = scf.if %0 -> (i32, i32) {
      "test.test3"() : () -> ()
      scf.yield %c0_i32, %c1_i32 : i32, i32
    } else {
      scf.yield %c1_i32, %c0_i32 : i32, i32
    }
    %2 = arith.trunci %1#1 : i32 to i1
    scf.condition(%2)
  } do {
    scf.yield
  }
  "test.test4"() : () -> ()
  return
}
```

Results

```
func.func @while_loop_with_block_args() {
  %1 = "test.test1"() : () -> index
  cf.br ^bb1(%1: index)
^bb1(%2: index):
  %cond:2 = "test.test2"()
  cf.cond_br %cond#0, ^bb2(%cond#1: i64),
                      ^bb3(%2: index)
^bb2(%3: i64):
  %4 = "test.test3"(%3) : (i64) -> index
  cf.br ^bb1(%4: index)
^bb3(%5: index):
  "test.test4"() : () -> ()
  return %5 : index
}
```

Results

```
func.func @while_loop_with_block_args() {
  %1 = "test.test1"() : () -> index
  cf.br ^bb1(%1: index)
^bb1(%2: index):
  %cond:2 = "test.test2"()
  cf.cond_br %cond#0, ^bb2(%cond#1: i64),
                      ^bb3(%2: index)
^bb2(%3: i64):
  %4 = "test.test3"(%3) : (i64) -> index
  cf.br ^bb1(%4: index)
^bb3(%5: index):
  "test.test4"() : () -> ()
  return %5 : index
}
```

```
func.func @while_loop_with_block_args() -> index {
  %0 = ub.poison : index
  %1 = "test.test1"() : () -> index
  %2:2 = scf.while (%arg0 = %1) {
    %3:2 = "test.test2"() : () -> (i1, i64)
    %4 = scf.if %3#0 -> (index) {
      %5 = "test.test3"(%3#1) : (i64) -> index
      scf.yield %5 : index
    } else {
      scf.yield %0 : index
    }
    scf.condition(%3#0) %4, %arg0 : index, index
  } do {
  ^bb0(%arg0: index, %arg1: index):
    scf.yield %arg0 : index
  }
  "test.test4"() : () -> ()
  return %2#1 : index
}
```

Results

```
func.func @switch_with_fallthrough(
    %flag: i32, %arg1 : f32, %arg2 : f32) {
  cf.switch %flag : i32, [
    default: ^bb1(%arg1 : f32),
    0: ^bb2(%arg2 : f32),
    1: ^bb3
  ]

^bb1(%arg3 : f32):
  %0 = call @foo(%arg3) : (f32) -> f32
  cf.br ^bb2(%0 : f32)

^bb2(%arg4 : f32):
  call @bar(%arg4) : (f32) -> ()
  cf.br ^bb3

^bb3:
  return
}
```

Results

```
func.func @switch_with_fallthrough(
    %flag: i32, %arg1 : f32, %arg2 : f32) {
  cf.switch %flag : i32, [
    default: ^bb1(%arg1 : f32),
    0: ^bb2(%arg2 : f32),
    1: ^bb3
  ]

^bb1(%arg3 : f32):
  %0 = call @foo(%arg3) : (f32) -> f32
  cf.br ^bb2(%0 : f32)

^bb2(%arg4 : f32):
  call @bar(%arg4) : (f32) -> ()
  cf.br ^bb3

^bb3:
  return
}
```

```
func.func @switch_with_fallthrough(
    %arg0: i32, %arg1: f32, %arg2: f32) {
  %c1_i32 = arith.constant 1 : i32
  %0 = ub.poison : f32
  %c0_i32 = arith.constant 0 : i32
  %1 = arith.index_castui %arg0 : i32 to index
  %2:2 = scf.index_switch %1 -> f32, i32
  case 0 {
    scf.yield %arg2, %c0_i32 : f32, i32
  }
  case 1 {
    scf.yield %0, %c1_i32 : f32, i32
  }
  default {
    %4 = func.call @foo(%arg1) : (f32) -> f32
    scf.yield %4, %c0_i32 : f32, i32
  }
  %3 = arith.index_castui %2#1 : i32 to index
  scf.index_switch %3
  case 0 {
    func.call @bar(%2#0) : (f32) -> ()
    scf.yield
  }
  default {
  }
  return
}
```

Results

```
func.func @multi_entry_loop(%cond: i1) {
  %0 = arith.constant 6 : i32
  %1 = arith.constant 5 : i32
  cf.cond_br %cond, ^bb0, ^bb1

^bb0:
  %exit = call @comp1(%0) : (i32) -> i1
  cf.cond_br %exit, ^bb2(%0 : i32), ^bb1

^bb1:
  %exit2 = call @comp2(%1) : (i32) -> i1
  cf.cond_br %exit2, ^bb2(%1 : i32), ^bb0

^bb2(%arg3 : i32):
  call @foo(%arg3) : (i32) -> ()
  return
}
```

Results

```
func.func @multi_entry_loop(%cond: i1) {
  %0 = arith.constant 6 : i32
  %1 = arith.constant 5 : i32
  cf.cond_br %cond, ^bb0, ^bb1

^bb0:
  %exit = call @comp1(%0) : (i32) -> i1
  cf.cond_br %exit, ^bb2(%0 : i32), ^bb1

^bb1:
  %exit2 = call @comp2(%1) : (i32) -> i1
  cf.cond_br %exit2, ^bb2(%1 : i32), ^bb0

^bb2(%arg3 : i32):
  call @foo(%arg3) : (i32) -> ()
  return
}
```

```
func.func @multi_entry_loop(%arg0: i1) {
  %true = arith.constant true
  %c1_i32 = arith.constant 1 : i32
  %c0_i32 = arith.constant 0 : i32
  %c6_i32 = arith.constant 6 : i32
  %c5_i32 = arith.constant 5 : i32
  %0 = arith.extui %arg0 : i1 to i32
  %1:2 = scf.while (%arg1 = %0) : (i32) -> (i32, i32) {
    %2 = arith.index_castui %arg1 : i32 to index
    %3:4 = scf.index_switch %2 -> i32, i32, i32, i32
    case 0 {
      %5 = func.call @comp2(%c5_i32) : (i32) -> i1
      %6 = arith.extui %5 : i1 to i32
      %7 = arith.xori %5, %true : i1
      %8 = arith.extui %7 : i1 to i32
      scf.yield %c1_i32, %c5_i32, %6, %8 : i32, i32, i32, i32
    }
    default {
      %5 = func.call @comp1(%c6_i32) : (i32) -> i1
      %6 = arith.extui %5 : i1 to i32
      %7 = arith.xori %5, %true : i1
      %8 = arith.extui %7 : i1 to i32
      scf.yield %c0_i32, %c6_i32, %6, %8 : i32, i32, i32, i32
    }
    %4 = arith.trunci %3#3 : i32 to i1
    scf.condition(%4) %3#0, %3#1 : i32, i32
  } do {
  ^bb0(%arg1: i32, %arg2: i32):
    scf.yield %arg1 : i32
  }
  call @foo(%1#1) : (i32) -> ()
  return
}
```

Custom dialects

```
/// Transformation lifting any dialect implementing control flow graph
/// operations to a dialect implementing structured control flow operations.
/// `region` is the region that should be transformed.
/// The implementation of `interface` is responsible for the conversion of the
/// control flow operations to the structured control flow operations.
FailureOr<bool> transformCFGToSCF(Region &region,
  CFGToSCFInterface &interface, DominanceInfo &dominanceInfo);
```

Custom dialects - Branches

```
class CFGToSCFInterface {
  /// 'controlFlowCondOp' → SCF op.
  virtual FailureOr<Operation *>
  createStructuredBranchRegionOp(
    OpBuilder &builder, Operation *controlFlowCondOp,
    TypeRange resultTypes, MutableArrayRef<Region> regions) = 0;
```

Custom dialects - Branches

```
class CFGToSCFInterface {
  /// 'controlFlowCondOp' → SCF op.
  virtual FailureOr<Operation *>
  createStructuredBranchRegionOp(
    OpBuilder &builder, Operation *controlFlowCondOp,
    TypeRange resultTypes, MutableArrayRef<Region> regions) = 0;
```

Custom dialects - Branches

```
class CFGToSCFInterface {
  /// 'controlFlowCondOp' → SCF op.
  virtual FailureOr<Operation *>
  createStructuredBranchRegionOp(
    OpBuilder &builder, Operation *controlFlowCondOp,
    TypeRange resultTypes, MutableArrayRef<Region> regions) = 0;
```

Custom dialects - Branches

```
class CFGToSCFInterface {
  /// 'controlFlowCondOp' -> SCF op.
  virtual FailureOr<Operation *>
  createStructuredBranchRegionOp(
    OpBuilder &builder, Operation *controlFlowCondOp,
    TypeRange resultTypes, MutableArrayRef<Region> regions) = 0;

  /// Create 'yield' op.
  virtual LogicalResult createStructuredBranchRegionTerminatorOp(
    Location loc, OpBuilder &builder, Operation *branchRegionOp,
    Operation *replacedControlFlowOp, ValueRange results) = 0;
```

Custom dialects - Branches

```
class CFGToSCFInterface {
  /// 'controlFlowCondOp' -> SCF op.
  virtual FailureOr<Operation *>
  createStructuredBranchRegionOp(
    OpBuilder &builder, Operation *controlFlowCondOp,
    TypeRange resultTypes, MutableArrayRef<Region> regions) = 0;

  /// Create 'yield' op.
  virtual LogicalResult createStructuredBranchRegionTerminatorOp(
    Location loc, OpBuilder &builder, Operation *branchRegionOp,
    Operation *replacedControlFlowOp, ValueRange results) = 0;
```

Custom dialects - Loops

```
class CFGToSCFInterface {
  …
  virtual FailureOr<Operation *> createStructuredDoWhileLoopOp(
    OpBuilder &builder, Operation *replacedOp, ValueRange loopValuesInit,
    Value condition, ValueRange loopValuesNextIter, Region &&loopBody) = 0;
```

Custom dialects - Loops

```
class CFGToSCFInterface {
  …
  virtual FailureOr<Operation *> createStructuredDoWhileLoopOp(
    OpBuilder &builder, Operation *replacedOp, ValueRange loopValuesInit,
    Value condition, ValueRange loopValuesNextIter, Region &&loopBody) = 0;
```

Custom dialects - Loops

```
class CFGToSCFInterface {
  …
  virtual FailureOr<Operation *> createStructuredDoWhileLoopOp(
    OpBuilder &builder, Operation *replacedOp, ValueRange loopValuesInit,
    Value condition, ValueRange loopValuesNextIter, Region &&loopBody) = 0;
```

Custom dialects - Loops

```
class CFGToSCFInterface {
  …
  virtual FailureOr<Operation *> createStructuredDoWhileLoopOp(
    OpBuilder &builder, Operation *replacedOp, ValueRange loopValuesInit,
    Value condition, ValueRange loopValuesNextIter, Region &&loopBody) = 0;
```

Future and enabled work

Future and enabled work

- Actual loop optimisations 🙂

Future and enabled work

- Actual loop optimisations 🙂
- Further lifting
  - Induction variables (`scf.for`)
  - Affine, SCEV

Future and enabled work

- Actual loop optimisations 🙂
- Further lifting
  - Induction variables (`scf.for`)
  - Affine, SCEV
- Structured exception handling?

Future and enabled work

- Actual loop optimisations 🙂
- Further lifting
  - Induction variables (`scf.for`)
  - Affine, SCEV
- Structured exception handling?
- Improving runtime complexity