

MLIR Dataflow Analysis

Jeff Niu jeff@modular.com

Tom Eccles tom.eccles@arm.com

Agenda

- Why a new framework?
- Design of an extensible, composable dataflow analysis framework
- MLIR framework internals
- Upstream example: SCCP
- Flang background
- In-depth Flang example: pointer lifetime analysis

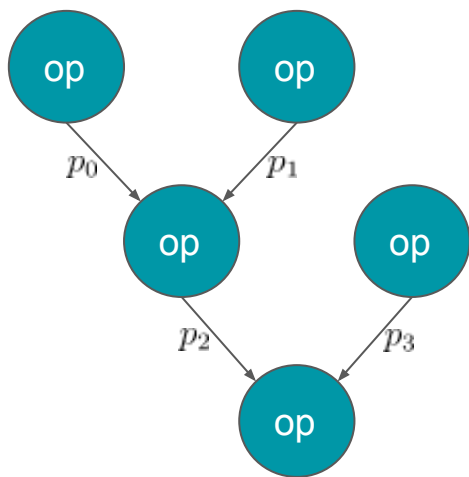
Background

Why a new framework?

- MLIR is an extensible compiler infrastructure
- It needs a dataflow analysis framework that
 - Provides the basic functionality for any dataflow analysis
 - Forms the basis of a library of analyses that compose together
 - And which are extensible out-of-tree
 - Is generally applicable to dataflow problems

Classical dataflow analysis

Sparse



$$S(p_2) = f_{op}(S(p_0), S(p_1))$$

Dense



$$S(p_{i+1}) = f_{op}(S(p_i))$$

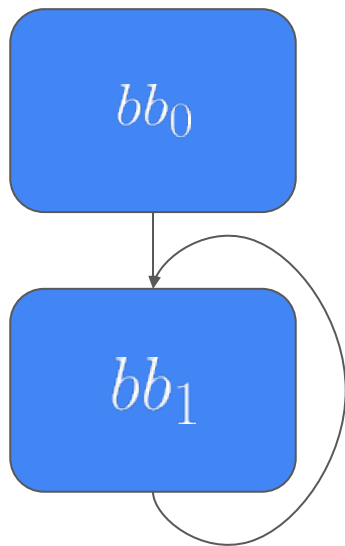
Classical dataflow analysis

Sparse

$$S(\text{arg}(bb_1, i)) = S(\text{out}(bb_0, i)) \cup S(\text{out}(bb_1, i))$$

Dense

$$S(\text{begin}(bb_1)) = S(\text{end}(bb_0)) \cup S(\text{end}(bb_1))$$



$$\top \cup x = x$$

$$\top \cap x = \top$$

$$\perp \cup x = \perp$$

$$\perp \cap x = x$$

Classical dataflow analysis

$$S_{i+1}(p_{n+1}) = S_i(p_{n+1}) \cup f_{op}(S_i(p_n))$$

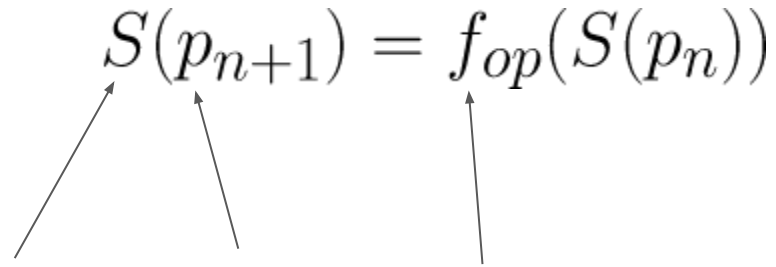
$$S_{i+1}(p_{n+1}) == S_i(p_{n+1})$$

Formulation

Extensibility

$$S(p_{n+1}) = f_{op}(S(p_n))$$

Extensibility

$$S(p_{n+1}) = f_{op}(S(p_n))$$


Composability

$$S = \{S_a, S_b, \dots\}$$

$$f_{op} = f_a \cap f_b$$

Composability

$$S(p_{n+1}) = f_{op}(S(p_n))$$

Composability

$$S(p_{n+1}) = f_{op}(S(p_n))$$

$$S(p_{n+1}) = (f_a \cap f_b)(S(p_n))$$

Composability

$$S(p_{n+1}) = f_{op}(S(p_n))$$

$$S(p_{n+1}) = (f_a \cap f_b)(S(p_n))$$

$$S(p_{n+1}) = f_a(S(p_n)) \cap f_b(S(p_n))$$

Generalized Dataflow


$$S_{i+1}(p_n) = S_i(p_n) \cup \bigcap f_k(\mathbb{S})$$

State of all program points



$$D(S_i(p_n)) = \{\dots\}$$

(Dynamic) dependents of a
specific state



Required framework API

- Define new, unique program points that have state variables
- Define new kinds of state variables
 - Implement `join` and `meet` binary operators (monotonic)
 - Implement `top` and `bottom` hooks
- Define and inject transfer functions that have access to any state variables
- Construct and manipulate the dynamic dependency graph
- Not a nightmare to debug (debug actions framework?)

MLIR Framework Internals

Pieces of the MLIR framework

- `GenericProgramPoint` – `StorageUniquer` user
 - E.g. a block of memory
 - Must have an MLIR `Location`
- `AnalysisState` - state variable base class
 - Meet, join, top, and bottom*
 - Must be printable
 - Contains a list of dependent transfer functions/update callbacks: $D = \{(f_i, p_i), \dots, g_j, \dots\}$
- `DataFlowAnalysis` - transfer function definition(s)
 - `visit` – implements the transfer function
 - `initialize` – seeds the dependency graph + state variables
- `DataFlowSolver`
 - Implements fixed-point iteration, memory management, etc.

Program Points

- MLIR operations, values, and blocks
- What else?

```
class GenericProgramPoint : public StorageUniquer::BaseStorage {
public:
    virtual ~GenericProgramPoint();

    /// Get the abstract program point's type identifier.
   TypeID getTypeID() const;

    /// Get a derived source location for the program point.
    virtual Location getLoc() const = 0;

    /// Print the program point.
    virtual void print(raw_ostream &os) const = 0;
};
```

Program Points

- MLIR operations, values, and blocks
- What else?

```
class CFGEdge
    : public GenericProgramPointBase<CFGEdge, std::pair<Block *, Block *>> {
public:
    using Base::Base;

    /// Get the block from which the edge originates.
    Block *getFrom() const { return getValue().first; }
    /// Get the target block.
    Block *getTo() const { return getValue().second; }

    /// Print the blocks between the control-flow edge.
    void print(raw_ostream &os) const override;
    /// Get a fused location of both blocks.
    Location getLoc() const override;
};
```

Analysis States

- `ChangeResult` – indicate if the state changed and dependents need to be invoked

```
struct AnalysisState {
    virtual ~AnalysisState();

    /// Create the analysis state at the given program point.
    AnalysisState(ProgramPoint point) : point(point) {}

    /// Returns the program point this state is located at.
    ProgramPoint getPoint() const { return point; }

    /// Join the analysis states.
    virtual ChangeResult join(const AnalysisState &state) = 0;
    /// Meet the analysis states.
    virtual ChangeResult meet(const AnalysisState &state) = 0;

    /// Print the contents of the analysis state.
    virtual void print(raw_ostream &os) const = 0;

    /// Update callback invoked when the state is updated.
    virtual void onUpdate(DataFlowSolver *solver) const {}
};
```

Analysis States

- `ChangeResult` – indicate if the state changed and dependents need to be invoked

```
struct PredecessorState : public AnalysisState {
    using AnalysisState::AnalysisState;

    /// Indicate that there are potentially unknown predecessors.
    ChangeResult setHasUnknownPredecessors() {
        return std::exchange(allKnown, false) ? ChangeResult::Change
            : ChangeResult::NoChange;
    }

    /// Add a known predecessor.
    ChangeResult add(Operation *predecessor);

    /// Union the known predecessor sets. If either has unknown predecessors,
    /// the result has unknown predecessors.
    ChangeResult join(const PredecessorState &state);
    /// Intersect the known predecessor sets. If either has all known
    /// predecessors, the result has all known predecessors.
    ChangeResult meet(const PredecessorState &state);

    /// Whether all predecessors are known. Optimistically assume that we know
    /// all predecessors.
    bool allKnown = true;

    /// The known control-flow predecessors of this program point.
    SmallPtrSet<Operation *, 4> knownPredecessors;
};
```

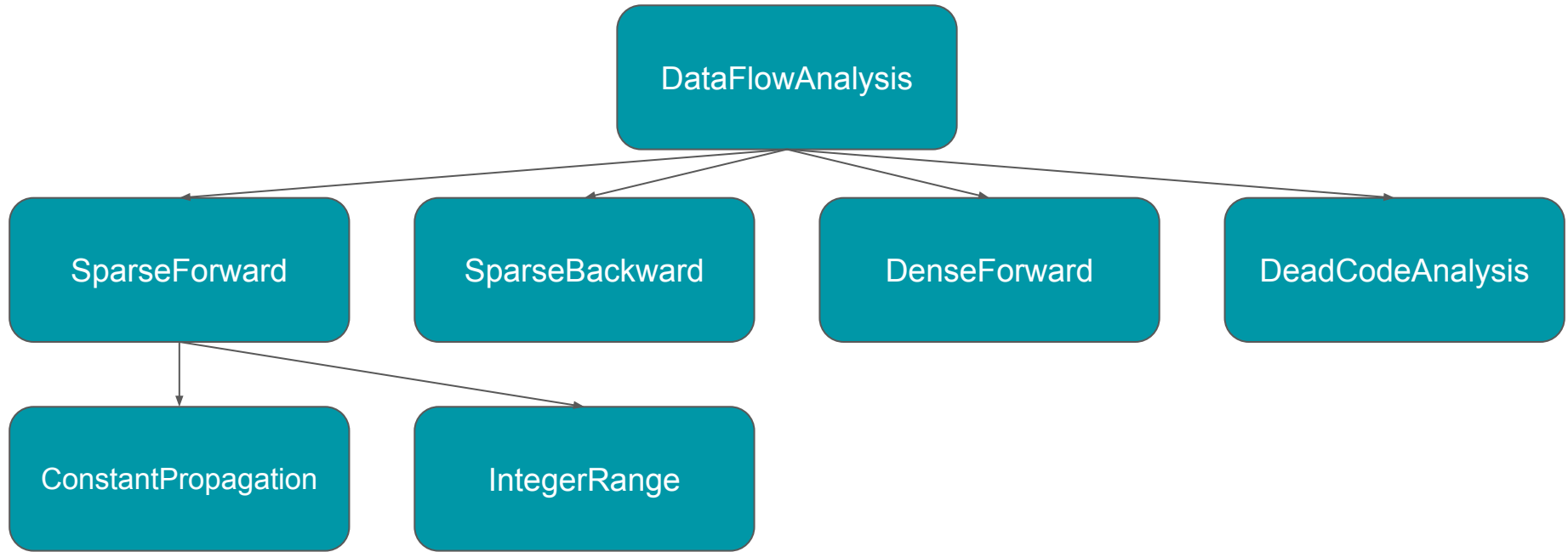
Child Analyses

- Subclasses of `DataFlowAnalysis`
- Implement transfer functions (`visit`) and dynamic dependency graph management (`initialize`)

```
struct DataFlowAnalysis {  
    virtual LogicalResult initialize(Operation *top) = 0;  
    virtual LogicalResult visit(ProgramPoint point) = 0;  
};
```

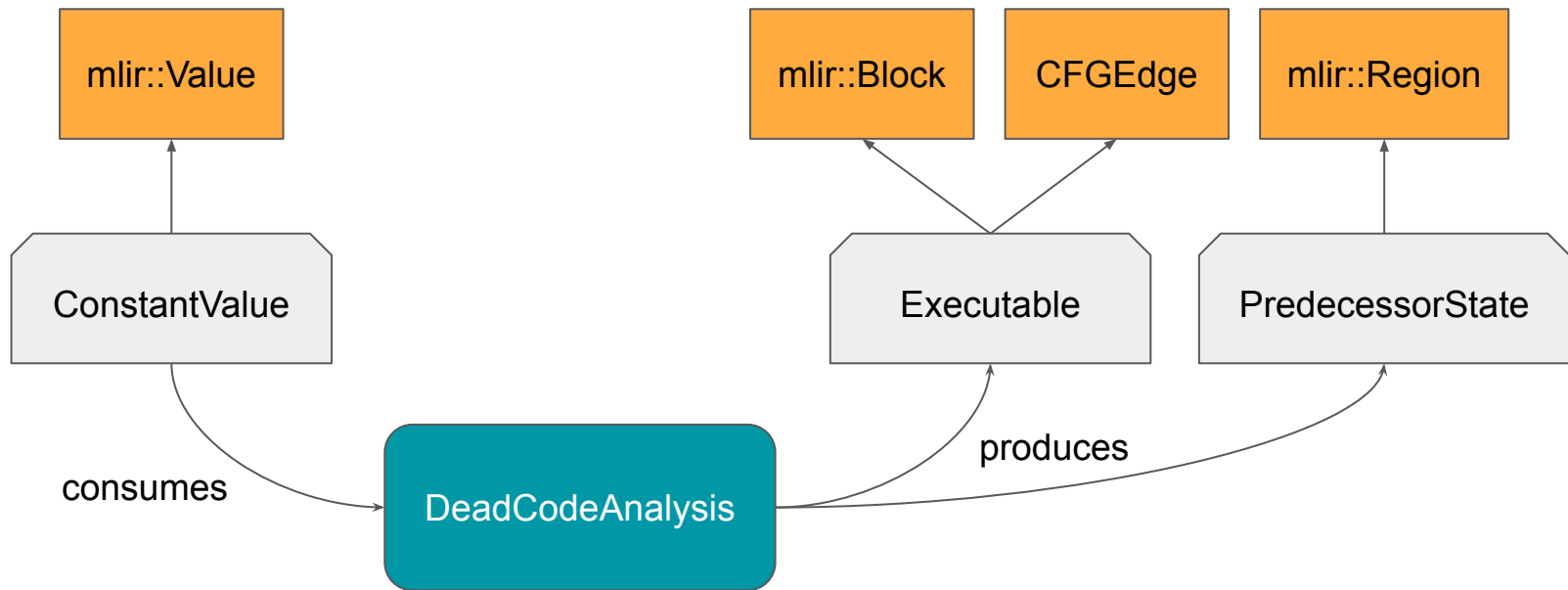
Child Analyses

- Subclasses of `DataFlowAnalysis`

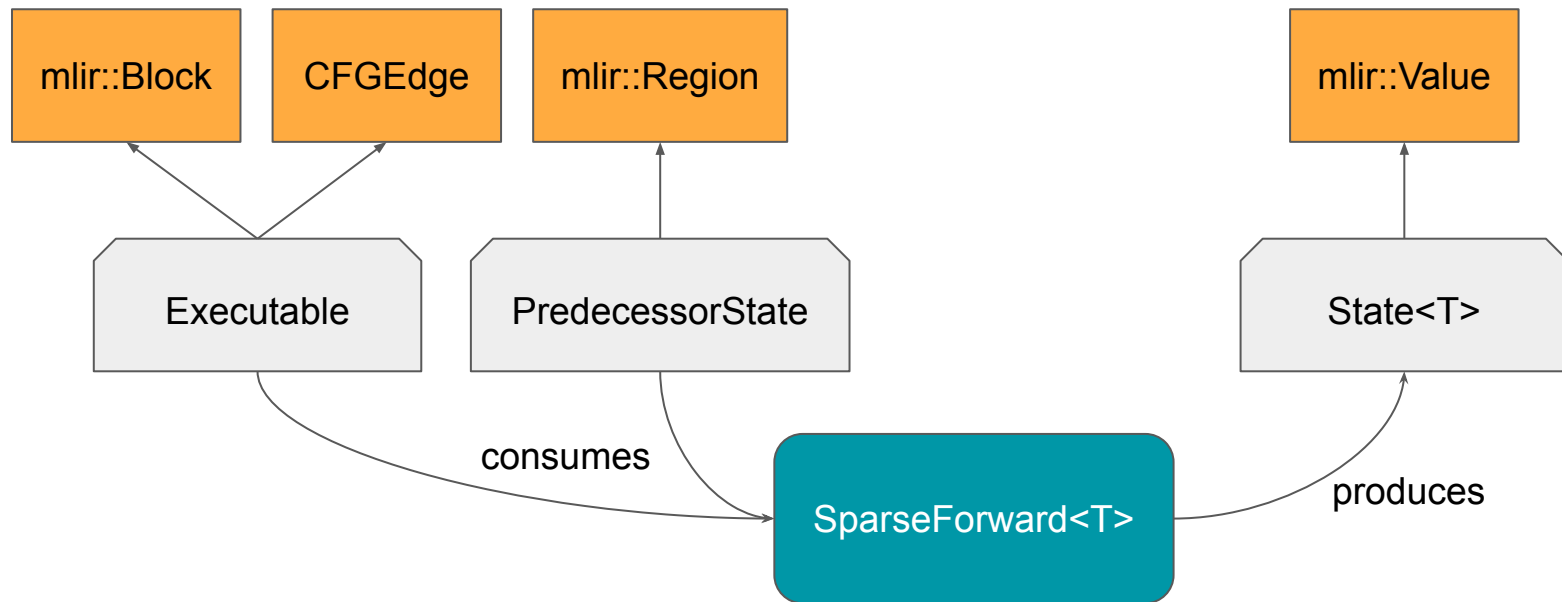


SCCP Example

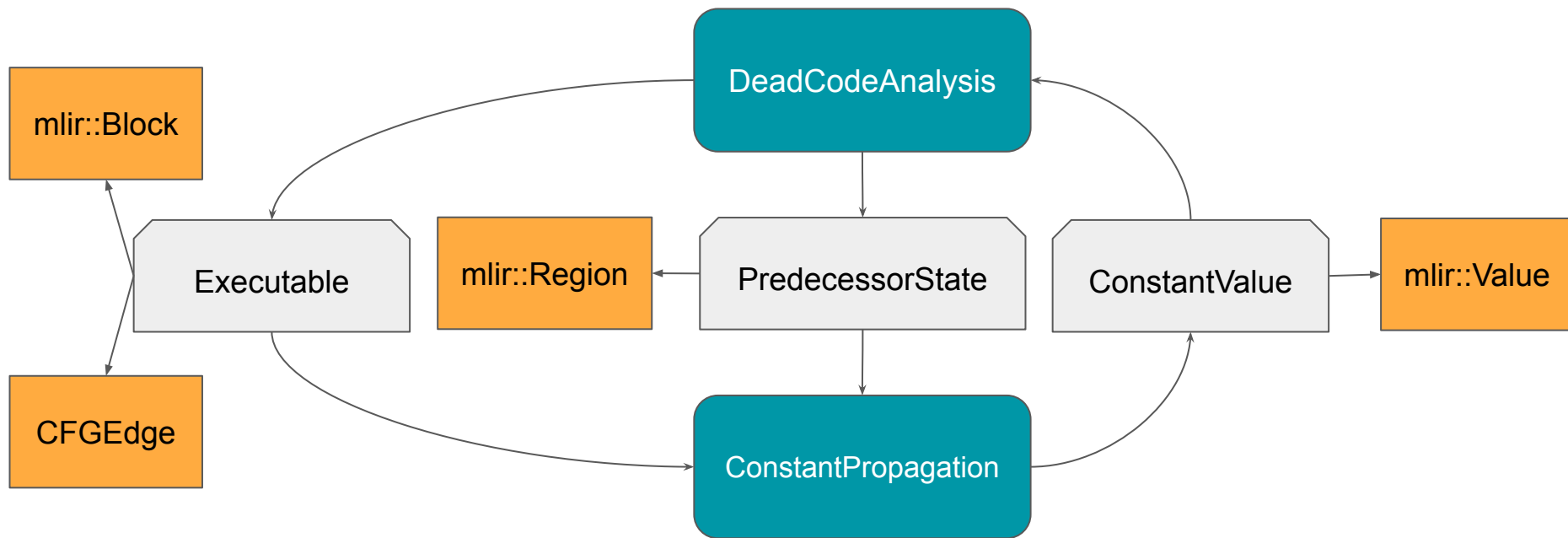
Components of DeadCodeAnalysis



Components of SparseForwardAnalysis



SCCP as a composition



Interfaces

- MLIR operation polymorphism
- Different operations from all kinds of dialects can implement a common API via interfaces
- Interfaces require operations to implement specific methods
- Allows writing generic passes or analyses

Interfaces

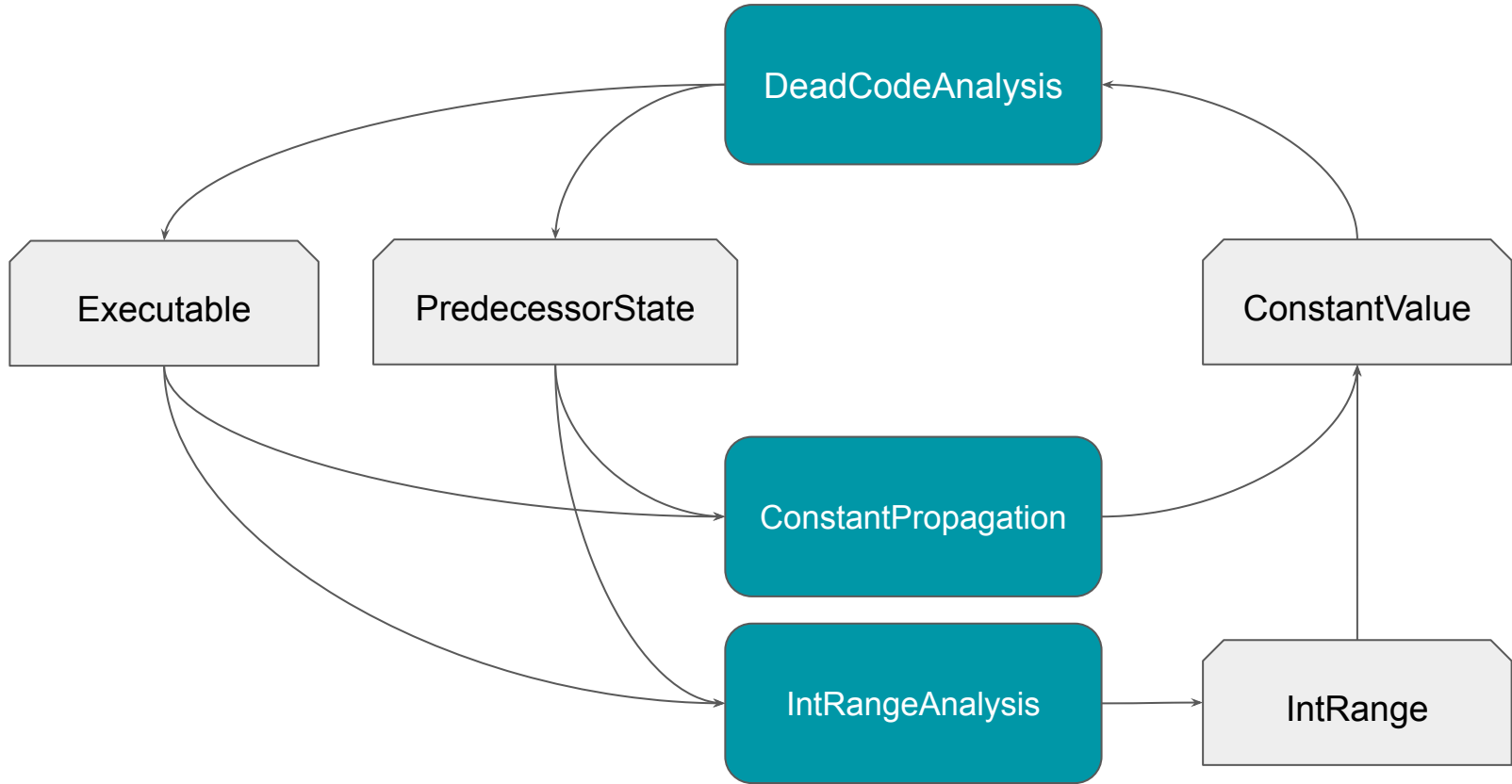
- DeadCodeAnalysis:
 - BranchOpInterface
 - RegionBranchOpInterface + RegionBranchTerminatorOpInterface
- ConstantPropagation
 - Operation::fold hooks

```
void DeadCodeAnalysis::visit(ProgramPoint p) {
    // ...
    bool live = false;
    for (Block *pred : b->getPredecessors()) {
        if (auto itf = dyn_cast<BranchOpInterface>(pred->getTerminator())) {
            SmallVector<Attribute> operands;
            for (Value operand : itf->getOperands())
                operands.push_back(getOrCreate<ConstantValue>(operand).getValue());
            Block *s = itf.getSuccessorsForOperands(operands);
            live |= !s || s != b;
        } else {
            live = true;
        }
    }
}
```

Interfaces

```
void SparseConstantPropagation::visit(ProgramPoint p) {
    // ...
    if (!getOrCreate<Executable>(op->getBlock()).isLive()) return;
    SmallVector<Attribute> operands = ... SmallVector<OpFoldResult> results;
    if (succeeded(op->fold(operands, results))) {
        for (auto [value, result] : llvm::zip(results, op->getResults())) {
            if (auto attr = result.dyn_cast<Attribute>()) {
                getOrCreate<ConstantValue>(result).update(attr);
            } else {
                getOrCreate<ConstantValue>(result).update(
                    getOrCreate<ConstantValue>(result.get<Value>()));
            }
        }
    } else {
        // Mark all result values as overdefined.
    }
}
```

Augment SCCP with integer range analysis



Augment SCCP with integer range analysis

```
func.func @sccp(%cond: i1) {  
    %c0 = constant 0  
    %c3 = constant 3  
    %c7 = constant 7  
    %0 = select %cond ? %c0 : %c3 // Constant propagation gives up  
    %1 = max %0, %c7 // Range analysis concludes ConstantValue(%1) = 7  
    // ...  
}
```

Known Issues

- Directly subclassing `DataFlowAnalysis` is not trivial
- Hard to control iteration order / convergence
 - Getting good performance requires fiddling with the dependency graph generation order
- Cost of abstraction

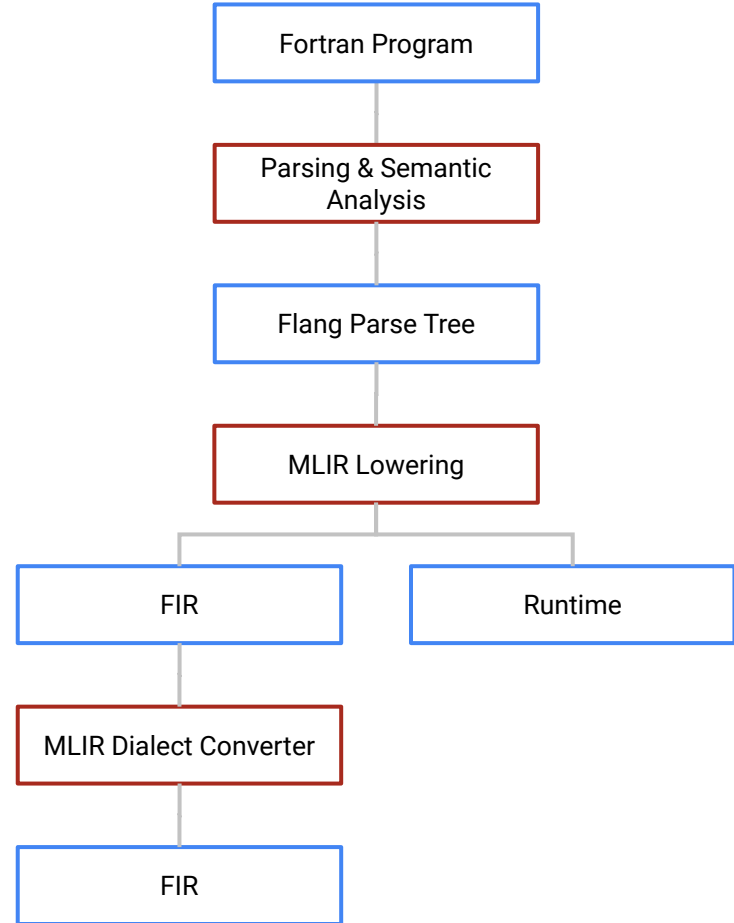
Flang Background

What is Flang?

- Flang is a new Fortran frontend written from scratch
- Flang lives in the llvm-project tree. Nowadays, all development happens upstream
- (LLVM) Flang is not Classic Flang (which lives out of tree)
- Ready to try, not yet ready for production use
 - Flang should compile valid Fortran programs and produce working executables (but there are open bugs on github)
 - Performance is not yet on par with Classic Flang and GFortran
 - `-flang-experimental-exec`

What is Flang?

- Traditional compilation flow
- Lowers from parse tree to FIR (“Fortran Intermediate Representation” - an MLIR dialect)
- MLIR interfaces with LLVM IR through the LLVM Dialect
- Lots of Flang is implemented in MLIR passes



FIR

```
func.func @QPsub(%arg0: !fir.ref<!fir.array<100xi32>>) {
  %c99 = arith.constant 99 : index
  %c0 = arith.constant 0 : index
  %c1 = arith.constant 1 : index
  %c100 = arith.constant 100 : index

  // prepare argument x
  %0 = fir.shape %c100 : (index) -> !fir.shape<1>
  %1 = fir.array_load %arg0(%0) : (!fir.ref<!fir.array<100xi32>>, !fir.shape<1>) -> !fir.array<100xi32>

  // allocate a temporary to copy x into
  %2 = fir.allocmem !fir.array<100xi32>
  %3 = fir.shape %c100 : (index) -> !fir.shape<1>
  %4 = fir.array_load %2(%3) : (!fir.heap<!fir.array<100xi32>>, !fir.shape<1>) -> !fir.array<100xi32>

  // array copy from %1 (x) to the temporary array %4
  %5 = fir.do_loop %arg1 = %c0 to %c99 step %c1 unordered iter_args(%arg2 = %4) -> (!fir.array<100xi32>) {
    %7 = fir.array_fetch %1, %arg1 : (!fir.array<100xi32>, index) -> i32
    %8 = fir.no_reassoc %7 : i32
    %9 = fir.array_update %arg2, %8, %arg1 : (!fir.array<100xi32>, i32, index) -> !fir.array<100xi32>
    fir.result %9 : !fir.array<100xi32>
  }
  fir.array_merge_store %4, %5 to %2 : !fir.array<100xi32>, !fir.array<100xi32>, !fir.heap<!fir.array<100xi32>>

  // call bar() on temporary array
  %6 = fir.convert %2 : (!fir.heap<!fir.array<100xi32>>) -> !fir.ref<!fir.array<100xi32>>
  fir.call @QPbar(%6) fastmath<contract> : (!fir.ref<!fir.array<100xi32>>) -> ()

  // free temporary array, return
  fir.freemem %2 : !fir.heap<!fir.array<100xi32>>
  return
}
```

```
! call bar on a copy of x
subroutine sub(x)
  integer :: x(100)
  call bar((x))
end subroutine
```

Flang Example: Pointer Lifetime Analysis

How to use the dataflow analysis framework in practice

-Ofast and -fstack-arrays

- Flang already puts array *variables* on the stack, no matter their size
 - There is a pass to move these to the heap, but it is not enabled by default
- However, Flang usually allocates *temporary* arrays on the heap
- At -Ofast it is common to move these allocations to the stack to save allocation overheads

-fstack-arrays in Flang

- The easy way to implement this would be to modify all of the locations in Flang which create temporary arrays so that they conditionally allocate on the stack
- But the community felt it would be hard to remember to check the `stack-arrays` flag (or call the right helper function) every time some new code allocates an array
- So we need a pass which looks over all heap array allocations in FIR and rewrites them as stack array allocations **when it is safe to do so**.

When is it safe to move allocations?

```
subroutine example(n)
  integer :: n
  real, dimension(:), pointer :: array

  allocate(array(n))
  deallocate(array)
end subroutine
```


When is it safe to move allocations?

```
subroutine example(n)
  integer :: n
  real, dimension(:), pointer :: array

  allocate(array(n))
  deallocate(array)

  call do_anything(array)
end subroutine
```

Using array is already undefined behavior so we don't care what this code does



When is it safe to move allocations?


```
subroutine example(n)
  integer :: n
  real, dimension(:), pointer :: array

  allocate(array(n))

  call do_anything(array)

  deallocate(array)
end subroutine
```

This is valid code - could it do anything which makes moving the allocation invalid?



When is it safe to move allocations?

```
real, dimension(:), pointer :: glbl

subroutine example(n)
  integer :: n
  real, dimension(:), pointer :: array

  allocate(array(n))

  glbl = array ←
deallocate(array)

  call do_something_with_glbl() ←
end subroutine
```

It would be bad if the pointer was stored somewhere that outlived this stack frame

But so long as the pointer was already freed, accesses to the global are undefined behavior are already undefined behavior

Lifetime analysis

```
subroutine example3(n, bool)
  integer :: n
  real, dimension(:), pointer :: array
  logical :: bool

  allocate(array(n))

  call do_anything(array)

  if (bool) then
    deallocate(array)
  end if
end subroutine
```

Could array outlive this stack frame? Maybe.
We need dataflow analysis to spot cases like this

Outline

- It is safe to move an allocation from the heap to the stack if and only if the allocation does not outlive the current stack frame
- We can determine the lifetime of a heap allocation by looking to see when it is freed
- We need to use data-flow analysis to determine the lifetime of a heap allocation

MLIR dense dataflow analysis

- Flang's stack arrays pass uses dataflow analysis to track the allocation state of array variables within functions
- For dense dataflow analysis, there is a lattice value
 - At each operation to represent the state after the operation executes
 - At each block to represent the state at the start of the block

Lattice Point

The state stored for each program point (operation or block)

```
llvm::SmallDenseMap<mlir::Value, AllocationState> stateMap;
```

E.g.

```
{ %0: AllocationState::Allocated,  
  %1: AllocationState::Freed }
```

SSA Variable allocation state

1. $\{\}$ - *unknown* unknown: data-flow analysis has not yet considered the state of this variable at this lattice point
2. Unknown - *known* unknown: data-flow analysis could not determine the allocation state at this lattice point (e.g. because of differences either side of a conditional statement)
3. Allocated - this SSA value has been allocated on the heap but not yet freed by this lattice point
4. Freed - this SSA value has been freed by this lattice point

Example 1

```
func.func @_QPsub(%arg0: !fir.ref<!fir.array<100xi32>>) {  
  %c99 = arith.constant 99 : index  
  %c0 = arith.constant 0 : index  
  %c1 = arith.constant 1 : index  
  %c100 = arith.constant 100 : index  
  
  // prepare argument x  
  %0 = fir.shape %c100 : (index) -> !fir.shape<1>  
  %1 = fir.array_load %arg0(%0) : (!fir.ref<!fir.array<100xi32>>, !fir.shape<1>) -> !fir.array<100xi32>  
  
  // allocate a temporary to copy x into  
  %2 = fir.allocmem !fir.array<100xi32> { %2: AllocationState::Allocated }  
  %3 = fir.shape %c100 : (index) -> !fir.shape<1>  
  %4 = fir.array_load %2(%3) : (!fir.heap<!fir.array<100xi32>>, !fir.shape<1>) -> !fir.array<100xi32>  
  
  // array copy from %1 (x) to the temporary array %4  
  %5 = fir.do_loop %arg1 = %c0 to %c99 step %c1 unordered iter_args(%arg2 = %4) -> (!fir.array<100xi32>) {  
    %7 = fir.array_fetch %1, %arg1 : (!fir.array<100xi32>, index) -> i32  
    %8 = fir.no_reassoc %7 : i32  
    %9 = fir.array_update %arg2, %8, %arg1 : (!fir.array<100xi32>, i32, index) -> !fir.array<100xi32>  
    fir.result %9 : !fir.array<100xi32>  
  }  
  fir.array_merge_store %4, %5 to %2 : !fir.array<100xi32>, !fir.array<100xi32>, !fir.heap<!fir.array<100xi32>>  
  
  // call bar() on temporary array  
  %6 = fir.convert %2 : (!fir.heap<!fir.array<100xi32>>) -> !fir.ref<!fir.array<100xi32>>  
  fir.call @_QPbar(%6) fastmath<contract> : (!fir.ref<!fir.array<100xi32>>) -> ()  
  
  // free temporary array, return  
  fir.freemem %2 : !fir.heap<!fir.array<100xi32>>  
  return  
} { %2: AllocationState::Freed }
```

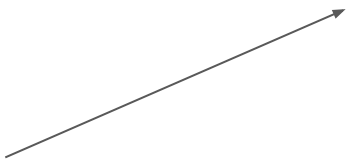
Empty lattice points

← State read from function terminator operation

Example 2

```
func.func @example2(%arg0: i1) {                                     {}
  %a = fir.allocmem !fir.array<1xi8>                             { %a: Allocated }
  scf.if %arg0                                                    { %a: Allocated }
    fir.freemem %a : !fir.heap<!fir.array<1xi8>>                 { %a: Freed }
  } else {
    call doSomething(%a)                                          { %a: Allocated }
  }
  return
}
```

The lattice values at each predecessor point are different so this is a known unknown



Example 3

```
func.func @example2(%arg0: i1) {
  %a = fir.allocmem !fir.array<1xi8>
  scf.if %arg0
    fir.freemem %a : !fir.heap<!fir.array<1xi8>>
  } else {
    call doSomething(%a)
    fir.freemem %a : !fir.heap<!fir.array<1xi8>>
  }
  return
}
```

{ }
{ %a: Allocated }
{ %a: Allocated }
 { %a: Freed }
 { %a: Allocated }
 { %a: Freed }
 { %a: Freed }



The lattice values at each predecessor point agree so they can be carried forward

Example 4

```
func.func @returns1(%arg0: i1) { {}  
  %0 = fir.allocmem !fir.array<42xi32> { %0: Allocated }  
  cf.cond_br %arg0, ^bb1, ^bb2 { %0: Allocated }  
^bb1: { %0: Allocated }  
  fir.freemem %0 : !fir.heap<!fir.array<42xi32>> { %0: Freed }  
  return { %0: Freed }  
^bb2: { %0: Allocated }  
  fir.unreachable { %0: Allocated }  
}
```

join(Freed, Allocated) = Unknown -> we can't be sure if this pointer outlives this stack frame

Example 5

```
func.func @returns1(%arg0: i1) {                                     {}  
  %0 = fir.allocmem !fir.array<42xi32>                          { %0: Allocated }  
  cf.cond_br %arg0, ^bb1, ^bb2                                  { %0: Allocated }  
^bb1:                                                            { %0: Allocated }  
  fir.freemem %0 : !fir.heap<!fir.array<42xi32>>              { %0: Freed }  
  return                                                         { %0: Freed }  
^bb2:                                                            { %0: Allocated }  
  fir.freemem %0 : !fir.heap<!fir.array<42xi32>>              { %0: Freed }  
  return                                                         { %0: Freed }  
}
```

join(Freed, Freed) = Freed -> This pointer is definitely freed by function return

Using the framework

Allocation State & Lattice Point

```
/// The state of an SSA value at each program point
```

```
enum class AllocationState {  
    Unknown,  
    Freed,  
    Allocated,  
};
```

```
/// Maps SSA values to their AllocationState at a particular program point.
```

```
class LatticePoint : public mlir::dataflow::AbstractDenseLattice {  
    Ilvm::SmallDenseMap<mlir::Value, AllocationState, 1> stateMap;  
public:  
    MLIR_DEFINE_EXPLICIT_INTERNAL_INLINE_TYPE_ID(LatticePoint)  
    using AbstractDenseLattice::AbstractDenseLattice;
```

```
/// Join the lattice across control-flow edges
```

```
mlir::ChangeResult join(const AbstractDenseLattice &lattice) override;
```

```
// [...]  
}
```

The join method is how predecessor lattice points are combined together to produce this point's value

Analysis class

Using LatticePoint from the previous slide

```
class AllocationAnalysis
  : public mlir::dataflow::DenseDataFlowAnalysis<LatticePoint> {
public:
  using DenseDataFlowAnalysis::DenseDataFlowAnalysis;

  void visitOperation(mlir::Operation *op, const LatticePoint &before,
                     LatticePoint *after) override;
  void setToEntryState(LatticePoint *lattice) override;

protected:
  /// Visit control flow ops and decide whether to call visitOperation
  /// to apply the transfer function
  void processOperation(mlir::Operation *op) override;
};
```

↓

Determine the lattice value after the execution of an MLIR operation

For most use cases, this one doesn't need to be overridden. For stack arrays I needed to allow carrying forward lattices after function calls

Running the analysis

```
mlir::DataFlowSolver solver;  
// these two are always required  
solver.load<mlir::dataflow::SparseConstantPropagation>();  
solver.load<mlir::dataflow::DeadCodeAnalysis>();  
solver.load<mlir::dataflow::AllocationAnalysis>();  
  
if (failed(solver.initializeAndRun(func)))  
    [...]  
  
// afterwards we can fetch lattice points for each (live) op  
mlir::Operation *op = [...];  
const LatticePoint *lattice =  
    solver.lookupState<LatticePoint>(op); // can be null
```

Other details about the stack arrays pass

Where should the new allocation be placed?

```
subroutine placement(n)
  integer :: n
  real, dimension(:), pointer :: array

  do i = 0, 9999999999999999999
    allocate(array(n))
    call do_something(array)
    deallocate(array)
  end do
end subroutine
```

If we put the alloca in the loop body, there will be an allocation for every iteration but none will be freed until the subroutine returns!

Where should the new allocation be placed?

```
subroutine placement2()  
  real, dimension(:), pointer :: array  
  
  do i = 0, 999999999999999999  
    allocate(array(i))  
    call do_something(array)  
    deallocate(array)  
  end do  
end subroutine
```

Impossible to move the allocation
outside of the loop if it *depends upon a
value not available until the loop body!*

We can't move the allocation so the best we can do is a stack save/restore

Where should the new allocation be placed?

- Not inside of a loop
 - In MLIR, we can have loop like operations (`scf.for`) or control flow graph loops between blocks (`cf.cond_br`)
- After all value arguments are available
- Within the same OpenMP region

Does the pass actually help?

- One real Fortran program (SNAP) showed a 23% improvement in run time
- ~3% improvement in exchange2 (spec2017)
- But other benchmarks showed little or no improvement

Any questions?