Deep Dive on MLIR Internals

OpInterface Implementation

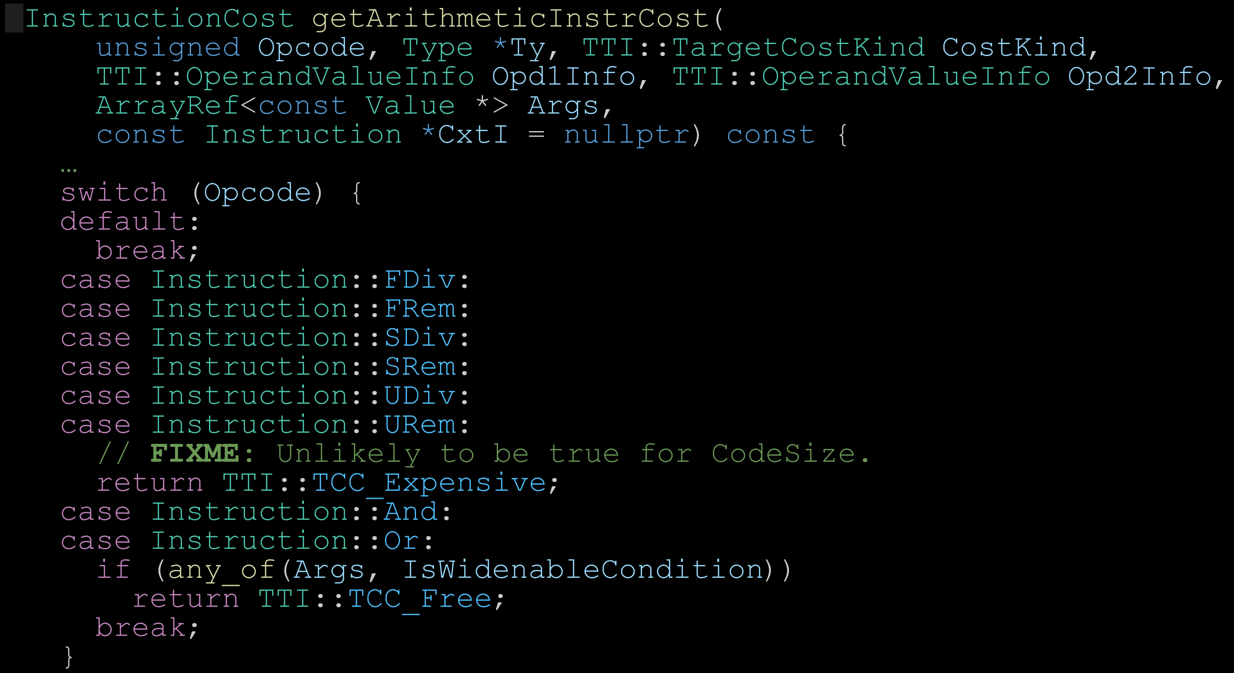
Mehdi Amini - NVIDIA

EuroLLVM 2024

Agenda

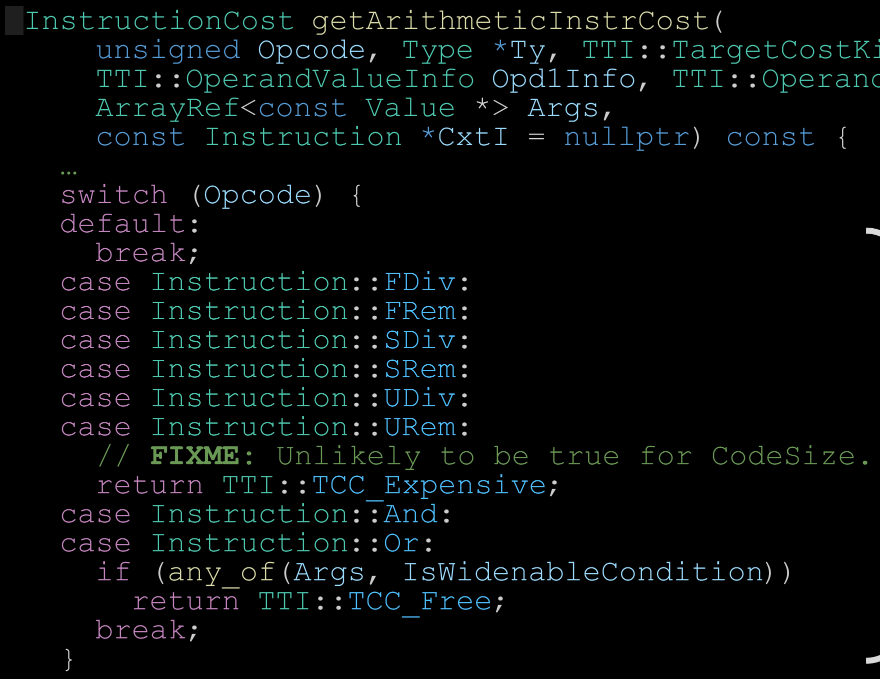
- Some implementation details background (with lot of code)
- Some more details on current implementation
   (more code, but hopefully some high-level intuition!)
- ODS Code Generation (still more code)
- External Interface & Promises

Why Interfaces?



```
InstructionCost getArithmeticInstrCost(
    unsigned Opcode, Type *Ty, TTI::TargetCostKind CostKind,
    TTI::OperandValueInfo Opd1Info, TTI::OperandValueInfo Opd2Info,
    ArrayRef<const Value *> Args,
    const Instruction *CxtI = nullptr) const {
  …
  switch (Opcode) {
  default:
    break;
  case Instruction::FDiv:
  case Instruction::FRem:
  case Instruction::SDiv:
  case Instruction::SRem:
  case Instruction::UDiv:
  case Instruction::URem:
    // FIXME: Unlikely to be true for CodeSize.
    return TTI::TCC_Expensive;
  case Instruction::And:
  case Instruction::Or:
    if (any_of(Args, IsWidenableCondition))
      return TTI::TCC_Free;
    break;
  }
```

Why Interfaces?

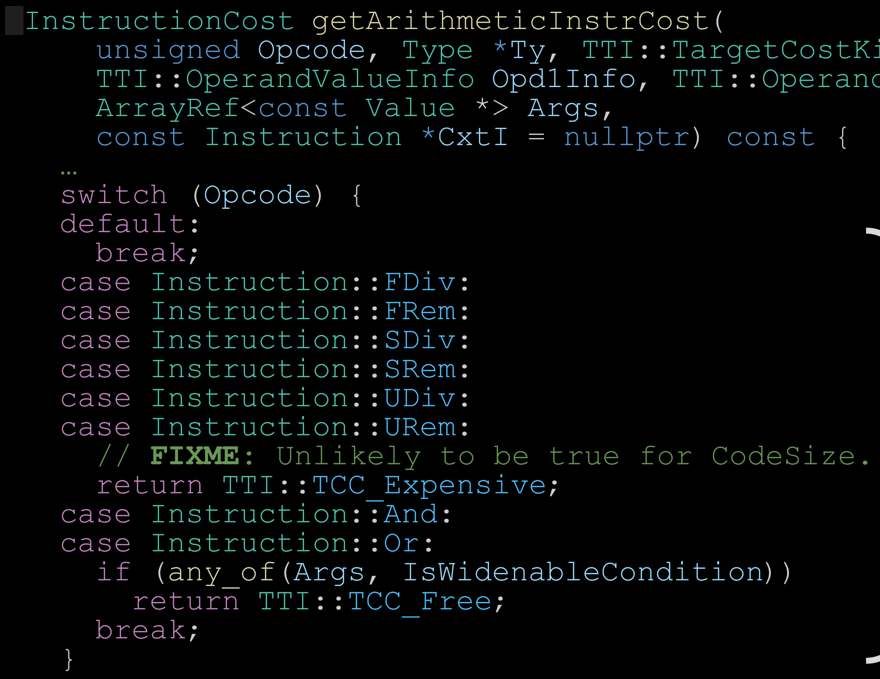


```
InstructionCost getArithmeticInstrCost(
    unsigned Opcode, Type *Ty, TTI::TargetCostKind CostKind,
    TTI::OperandValueInfo Opd1Info, TTI::OperandValueInfo Opd2Info,
    ArrayRef<const Value *> Args,
    const Instruction *CxtI = nullptr) const {
  …
  switch (Opcode) {
  default:
    break;
  case Instruction::FDiv:
  case Instruction::FRem:
  case Instruction::SDiv:
  case Instruction::SRem:
  case Instruction::UDiv:
  case Instruction::URem:
    // FIXME: Unlikely to be true for CodeSize.
    return TTI::TCC_Expensive;
  case Instruction::And:
  case Instruction::Or:
    if (any_of(Args, IsWidenableCondition))
      return TTI::TCC_Free;
    break;
  }
```

LLVM Transformations operate on a closed list of instructions.

MLIR does not have a predefined list
=> how to write generic passes?

Why Interfaces?



```
InstructionCost getArithmeticInstrCost(
    unsigned Opcode, Type *Ty, TTI::TargetCostKind CostKind,
    TTI::OperandValueInfo Opd1Info, TTI::OperandValueInfo Opd2Info,
    ArrayRef<const Value *> Args,
    const Instruction *CxtI = nullptr) const {
  …
  switch (Opcode) {
  default:
    break;
  case Instruction::FDiv:
  case Instruction::FRem:
  case Instruction::SDiv:
  case Instruction::SRem:
  case Instruction::UDiv:
  case Instruction::URem:
    // FIXME: Unlikely to be true for CodeSize.
    return TTI::TCC_Expensive;
  case Instruction::And:
  case Instruction::Or:
    if (any_of(Args, IsWidenableCondition))
      return TTI::TCC_Free;
    break;
  }
```

LLVM Transformations operate on a closed list of instructions.

MLIR does not have a predefined list => how to write generic passes?

```
if (auto iface =
dyn_cast<InstructionCostOpInterface >(op))
  return iface.getArithmeticInstrCost (    );
```

Trait vs OpInterface

Traits provides:

- The ability to check if the trait exists on an op: `op->hasTrait<SomeTrait>();`
- A base class for the concrete Op without virtual methods

Interface provides (**on top of a Trait) polymorphism**:

- A base class for the op, with virtual methods *(conceptually)*

Trait vs OpInterface

Traits provides:

- The ability to check if the trait exists on an op: `op->hasTrait<SomeTrait>();`
- A base class for the concrete Op without virtual methods

Interface provides (**on top of a Trait) polymorphism**:

- A base class for the op, with virtual methods *(conceptually)*

```
template<typename ConcreteOp>
class LinalgOpTrait {
 unsigned getNumParallelLoops() {
   return llvm::count(cast<ConcreteOp>(this->getOperation()).getIteratorTypesArray(),
                       utils::IteratorType::parallel);
 }
}
class LinalgDotOp : public LinalgOpTrait<LinalgDotOp>, ... {
  ...
```

Trait vs OpInterface

Traits provides:

- The ability to check if the trait exists on an op: `op->hasTrait<SomeTrait>();`
- A base class for the concrete Op without virtual methods

Interface provides (**on top of a Trait) polymorphism**:

- A base class for the op, with virtual methods *(conceptually)*

```
template<typename ConcreteOp>
class LinalgOpTrait {
 unsigned getNumParallelLoops() {
   return llvm::count(cast<ConcreteOp>(this->getOperation()).getIteratorTypesArray(),
                      utils::IteratorType::parallel);
 }
}
class LinalgDotOp : public LinalgOpTrait<LinalgDotOp>, ... {
  ...

     if (auto dotOp = dyn_cast<LinalgDotOp>(op0))
       return dotOp.getNumParallelLoops();
```

Traits provide behavior on the concrete op class,
but you need to cast to the concrete type!

OpInterface: it’s just like a virtual base class…

```
class LinalgOpInterface {
public:
  virtual unsigned getNumParallelLoops();
  virtual unsigned getNumReductionLoops();
  virtual unsigned getNumWindowLoops();
  virtual unsigned getNumInputsAndOutputs();
};

class LinalgDotOp :
    public LinalgOpInterface, Op<LinalgDotOp,...> {
public:
  unsigned getNumParallelLoops() override;
  unsigned getNumReductionLoops() override;
  unsigned getNumWindowLoops() override;
  unsigned getNumInputsAndOutputs() override;
  …
};
```

OpInterface: it’s just like a virtual base class…

```
class LinalgOpInterface {
public:
  virtual unsigned getNumParallelLoops();
  virtual unsigned getNumReductionLoops();
  virtual unsigned getNumWindowLoops();
  virtual unsigned getNumInputsAndOutputs();
};

class LinalgDotOp :
    public LinalgOpInterface, Op<LinalgDotOp,...> {
public:
  unsigned getNumParallelLoops() override;
  unsigned getNumReductionLoops() override;
  unsigned getNumWindowLoops() override;
  unsigned getNumInputsAndOutputs() override;
  …
};
```

This cannot work!!

```
LogicalResult tileLinalgOp(
      Operation *op, ArrayRef<int64_t> tileSizes) {
  if (auto linalgOp = dyn_cast<LinalgOpInterface>(op))
    return tileLinalgOp(linalgOp, tileSizes);
  return failure();
}
```

OpInterface: it’s just like a virtual base class…

This cannot work!!

```
class LinalgOpInterface {
public:
  virtual unsigned getNumParallelLoops();
  virtual unsigned getNumReductionLoops();
  virtual unsigned getNumWindowLoops();
  virtual unsigned getNumInputsAndOutputs();
};
```

```
class LinalgDotOp :
    public LinalgOpInterface, Op<LinalgDotOp,...> {
public:
  unsigned getNumParallelLoops() override;
  unsigned getNumReductionLoops() override;
  unsigned getNumWindowLoops() override;
  unsigned getNumInputsAndOutputs() override;
  …
};
```

```
LogicalResult tileLinalgOp(
      Operation *op, ArrayRef<int64_t> tileSizes) {
  if (auto linalgOp = dyn_cast<LinalgOpInterface>(op))
    return tileLinalgOp(linalgOp, tileSizes);
  return failure();
}
```

What you really want here is:

```
LinalgOpInterface iface = TypeSwitch<LinalgOpInterface>(op)
   .Case<LinalgDotOp>() { return cast<LinalgDotOp>(op); }
   .Case<LinalgConvOp>() { return LinalgConvOp(op); }
   ...
```

OpInterface: it’s just like a virtual base class…

```
class LinalgOpInterface {
public:
  virtual unsigned getNumParallelLoops();
  virtual unsigned getNumReductionLoops();
  virtual unsigned getNumWindowLoops();
  virtual unsigned getNumInputsAndOutputs();
};

class LinalgDotOp :
    public LinalgOpInterface, Op<LinalgDotOp,...> {
public:
  unsigned getNumParallelLoops() override;
  unsigned getNumReductionLoops() override;
  unsigned getNumWindowLoops() override;
  unsigned getNumInputsAndOutputs() override;
  …
};
```

This cannot work!!

```
LogicalResult tileLinalgOp(
      Operation *op, ArrayRef<int64_t> tileSizes) {
  if (auto linalgOp = dyn_cast<LinalgOpInterface>(op))
    return tileLinalgOp(linalgOp, tileSizes);
  return failure();
}
```

What you really want here is:

```
LinalgOpInterface iface = TypeSwitch<LinalgOpInterface>(op)
    .Case<LinalgDotOp>() { return cast<LinalgDotOp>(op); }
    .Case<LinalgConvOp>() { return LinalgConvOp(op); }
    ...
```

Return by value: “slicing” of the derived class => this cannot work!!

OpInterface: it’s just like a virtual base class…

```
class LinalgOpInterface {
public:
  virtual unsigned getNumParallelLoops();
  virtual unsigned getNumReductionLoops();
  virtual unsigned getNumWindowLoops();
  virtual unsigned getNumInputsAndOutputs();
};

class LinalgDotOp :
    public LinalgOpInterface, Op<LinalgDotOp,...> {
public:
  unsigned getNumParallelLoops() override;
  unsigned getNumReductionLoops() override;
  unsigned getNumWindowLoops() override;
  unsigned getNumInputsAndOutputs() override;
  …
};
```

This cannot work!!

```
LogicalResult tileLinalgOp(
      Operation *op, ArrayRef<int64_t> tileSizes) {
  if (auto linalgOp = dyn_cast<LinalgOpInterface>(op))
    return tileLinalgOp(linalgOp, tileSizes);
  return failure();
}
```

What you really want here is:

```
LinalgOpInterface iface = TypeSwitch<LinalgOpInterface>(op)
  .Case<LinalgDotOp>() { return cast<LinalgDotOp>(op); }
  .Case<LinalgConvOp>() { return LinalgConvOp(op); }
  ...
```

Return by value: “slicing” of the derived class => this cannot work!!

```
std::unique_ptr<LinalgOpInterface> iface =
  TypeSwitch<LinalgOpInterface>(op)
   .Case<LinalgDotOp>() { return std::make_unique<LinalgDotOp>(op); }
   .Case<LinalgConvOp>() { return std::make_unique<LinalgConvOp>(op); }
   ...
```

OpInterface: it’s just like a virtual base class…

This cannot work!!

```
class LinalgOpInterface {
public:
  virtual unsigned getNumParallelLoops();
  virtual unsigned getNumReductionLoops();
  virtual unsigned getNumWindowLoops();
  virtual unsigned getNumInputsAndOutputs();
};

class LinalgDotOp :
    public LinalgOpInterface, Op<LinalgDotOp,...> {
public:
  unsigned getNumParallelLoops() override;
  unsigned getNumReductionLoops() override;
  unsigned getNumWindowLoops() override;
  unsigned getNumInputsAndOutputs() override;
  …
};
```

```
LogicalResult tileLinalgOp(
      Operation *op, ArrayRef<int64_t> tileSizes) {
  if (auto linalgOp = dyn_cast<LinalgOpInterface>(op))
    return tileLinalgOp(linalgOp, tileSizes);
  return failure();
}
```

What you really want here is:

```
LinalgOpInterface iface = TypeSwitch<LinalgOpInterface>(op)
   .Case<LinalgDotOp>() { return cast<LinalgDotOp>(op); }
   .Case<LinalgConvOp>() { return LinalgConvOp(op); }
   ...
```

Return by value: “slicing” of the derived class => this cannot work!!

Heap-alloc for every interface cast?

```
 std::unique_ptr<LinalgOpInterface> iface =
   TypeSwitch<LinalgOpInterface>(op)
    .Case<LinalgDotOp>() { return std::make_unique<LinalgDotOp>(op); }
    .Case<LinalgConvOp>() { return std::make_unique<LinalgConvOp>(op); }
    ...
```

OpInterface: it’s just like a virtual base class…

```
class LinalgOpInterface {
public:
  virtual unsigned getNumParallelLoops();
  virtual unsigned getNumReductionLoops();
  virtual unsigned getNumWindowLoops();
  virtual unsigned getNumInputsAndOutputs();
};

class LinalgDotOp :
    public LinalgOpInterface, Op<LinalgDotOp,...> {
public:
  unsigned getNumParallelLoops() override;
  unsigned getNumReductionLoops() override;
  unsigned getNumWindowLoops() override;
  unsigned getNumInputsAndOutputs() override;
  …
};
```

This cannot work!!

```
LogicalResult tileLinalgOp(
      Operation *op, ArrayRef<int64_t> tileSizes) {
  if (auto linalgOp = dyn_cast<LinalgOpInterface>(op))
    return tileLinalgOp(linalgOp, tileSizes);
  return failure();
}
```

What you really want here is:

```
LinalgOpInterface iface = TypeSwitch<LinalgOpInterface>(op)
   .Case<LinalgDotOp>() { return cast<LinalgDotOp>(op); }
   .Case<LinalgConvOp>() { return LinalgConvOp(op); }
   ...
```

Return by value: “slicing” of the derived class => this cannot work!!

Heap-alloc for every interface cast?

```
std::unique_ptr<LinalgOpInterface> iface =
   TypeSwitch<LinalgOpInterface>(op)
    .Case<LinalgDotOp>() { return std::make_unique<LinalgDotOp>(op); }
    .Case<LinalgConvOp>() { return std::make_unique<LinalgConvOp>(op); }
    ...
```

Back to a predefined list

😢

It’s just a virtual base class…

```
LinalgOpInterface iface = TypeSwitch<LinalgOpInterface>(op)
    .Case<LinalgDotOp>()  { return cast<LinalgDotOp>(op);  }
    .Case<LinalgConvOp>() { return LinalgConvOp(op); }
    ...
```

POP-QUIZZ: what’s the difference here?

```
LinalgDotOp dotOp(op);
```

vs

```
LinalgDotOp linalgOp = cast<LinalgDotOp>(op);
```

Answer: the second one is asserting if the op mismatches

It’s just a virtual base class…

```
class LinalgOpInterface {
public:
  virtual unsigned getNumParallelLoops();
  virtual unsigned getNumReductionLoops();
  virtual unsigned getNumWindowLoops();
  virtual unsigned getNumInputsAndOutputs();
};

class LinalgDotOp :
    public LinalgOpInterface, Op<LinalgDotOp,...> {
public:
  unsigned getNumParallelLoops() override;
  unsigned getNumReductionLoops() override;
  unsigned getNumWindowLoops() override;
  unsigned getNumInputsAndOutputs() override;
  …
};
```

MLIR Fundamentals:

Concrete Op class shouldn’t define any state!

(a virtual table pointer counts as “state”)

The only state is a single member inherited here:

`Operation *state;`

Inheritance is the root of all evil

Sean Parent @ Going Native 2013 (slides and sources)

=> Polymorphism & Virtual dispatch … without inheritance!

```
class LinalgOpInterface :
    public mlir::Op<LinalgOpInterface, ...> {

 public:
  int getNumParallelLoops() const {
    self->getNumParallelLoops(); }
  };
```

Inheritance is the root of all evil

Sean Parent @ Going Native 2013 (slides and sources)

=> Polymorphism & Virtual dispatch … without inheritance!

```
class LinalgOpInterface :
    public mlir::Op<LinalgOpInterface, ...> {

 public:
  int getNumParallelLoops() const {
    self->getNumParallelLoops(); }
  };
```

```
private:
 struct Concept {
   virtual ~Concept() = default;
   virtual int getNumParallelLoops() const = 0;
 };
 shared_ptr<const Concept> self;
```

Inheritance is the root of all evil

Sean Parent @ Going Native 2013 (slides and sources)

=> Polymorphism & Virtual dispatch … without inheritance!

```
class LinalgOpInterface :
    public mlir::Op<LinalgOpInterface,...> {

 public:
  int getNumParallelLoops() const {
    self->getNumParallelLoops(); }
  };
```

```
private:
 struct Concept {
   virtual ~Concept() = default;
   virtual int getNumParallelLoops() const = 0;
 };
 shared_ptr<const Concept> self;

 template <typename ConcreteOp>
 struct Model : Concept {
   Model(ConcreteOp x) : impl(move(x)) {}
   int getNumParallelLoops() const override {
       return impl.getNumParallelLoops();
   }
   ConcreteOp impl;
 };
```

Inheritance is the root of all evil

Sean Parent @ Going Native 2013 (slides and sources)

=> Polymorphism & Virtual dispatch … without inheritance!

```
class LinalgOpInterface :
    public mlir::Op<LinalgOpInterface,...> {

 public:
  int getNumParallelLoops() const {
    self->getNumParallelLoops(); }
  };

  template <typename T>
  LinalgOpInterface(T x) :
    self(make_shared<Model<T>>(move(x))) {}
```

```
private:
 struct Concept {
   virtual ~Concept() = default;
   virtual int getNumParallelLoops() const = 0;
 };
 shared_ptr<const Concept> self;

 template <typename ConcreteOp>
 struct Model : Concept {
   Model(ConcreteOp x) : impl(move(x)) {}
   int getNumParallelLoops() const override {
       return impl.getNumParallelLoops();
   }
   ConcreteOp impl;
 };
```

Inheritance is the root of all evil

Sean Parent @ Going Native 2013 (slides and sources)

=> Polymorphism & Virtual dispatch … without inheritance!

```
class LinalgOpInterface :
    public mlir::Op<LinalgOpInterface, ...> {

 public:
  int getNumParallelLoops() const {
    self->getNumParallelLoops(); }
  };

  template <typename T>
  LinalgOpInterface(T x) :
    self(make_shared<Model<T>>(move(x))) {}
```

Still cannot be constructed from an *Operation **

Heap alloc on every interface cast!

```
private:
 struct Concept {
   virtual ~Concept() = default;
   virtual int getNumParallelLoops() const = 0;
 };
 shared_ptr<const Concept> self;

 template <typename ConcreteOp>
 struct Model : Concept {
   Model(ConcreteOp x) : impl(move(x)) {}
   int getNumParallelLoops() const override {
       return impl.getNumParallelLoops();
   }
   ConcreteOp impl;
 };
```

The method must exist on the concrete class, does not need to be virtual. It can be provided by a trait!!

Inheritance is the root of all evil

Sean Parent @ Going Native 2013 (slides and sources)

=> Polymorphism & Virtual dispatch … without inheritance!

```
class LinalgOpInterface :
    public mlir::Op<LinalgOpInterface, ...> {

 public:
  int getNumParallelLoops() const {
    self->getNumParallelLoops(); }
  };

  template <typename T>
  LinalgOpInterface(T x) :
    self(make_shared<Model<T>>(move(x))) {}
```

Still cannot be constructed from an *Operation **

Heap alloc on every interface cast!

```
private:
 struct Concept {
   virtual ~Concept() = default;
   virtual int getNumParallelLoops() const = 0;
 };
 shared_ptr<const Concept> self;

 template <typename ConcreteOp>
 struct Model : Concept {
   Model(ConcreteOp x) : impl(move(x)) {}
   int getNumParallelLoops() const override {
       return impl.getNumParallelLoops();
   }
   ConcreteOp impl;
 };
```

Inheritance is the root of all evil: stateless!

Initial Version (pre-ODS)

```
class LinalgOpInterface :
    public mlir::Op<LinalgOpInterface, ...> {

 public:
  int getNumParallelLoops() const {
    self->getNumParallelLoops(); }
  };

  template <typename T>
  LinalgOpInterface(T x) :
    self(make_shared<Model<T>>(move(x))) {}
```

Still cannot be constructed from an *Operation* *

Heap alloc on every interface cast!

```
private:
 struct Concept {
   virtual ~Concept() = default;
   virtual int
    getNumParallelLoops(Operation *) const = 0;
 };
 const Concept *self;

 template <typename ConcreteOp>
 struct Model : Concept {
   int getNumParallelLoops(Operation *op)
                             const override {
     return cast<ConcreteOp>(op).getNumParallelLoops();
   }
 };
```

Take the state explicitly!

Stateless!

Can be allocated once.

Inheritance is the root of all evil: stateless!

Initial Version (pre-ODS)

```
class LinalgOpInterface :
    public mlir::Op<LinalgOpInterface, ...> {

 public:
  int getNumParallelLoops() const {
    self->getNumParallelLoops(getOperation()); }
  };

  LinalgOpInterface(Operation *op) :
    mlir::Op<LinalgOpInterface, ...>(op) {
    OperationName name = op->getName();
    if (std::optional<RegisteredOperationName> rInfo =
            name.getRegisteredInfo()) {
      self = rInfo->getInterface<ConcreteType>())
```

Cast from *Operation**

through map lookup on the

RegisteredOperationName

```
private:
 struct Concept {
   virtual ~Concept() = default;
   virtual int
    getNumParallelLoops(Operation *) const = 0;
 };
 const Concept *self;

 template <typename ConcreteOp>
 struct Model : Concept {
   int getNumParallelLoops(Operation *op)
                              const override {
     return cast<ConcreteOp>(op).getNumParallelLoops();
   }
 };
```

Inheritance is the root of all evil: stateless!

```
class LinalgOpInterface :
    public mlir::Op<LinalgOpInterface, ...> {

 public:
  int getNumParallelLoops() const {
    self->getNumParallelLoops(getOperation()); }
  };

  LinalgOpInterface(Operation *op) :
    mlir::Op<LinalgOpInterface, ...>(op) {
    OperationName name = op->getName();
    if (std::optional<RegisteredOperationName> rInfo =
            name.getRegisteredInfo()) {
      self = rInfo->getInterface<ConcreteType>())
```

Cast from *Operation**

through map lookup on the

RegisteredOperationName

```
private:
 struct Concept {
   virtual ~Concept() = default;
   virtual int
    getNumParallelLoops(Operation *) const = 0;
 };
 const Concept *self;


 template <typename ConcreteOp>
 struct Model : Concept {
   int getNumParallelLoops(Operation *op)
                              const override {
```

Pointer-to-Pointer to the vtable.

***RegisteredOperationName*: this is the struct created in the MLIRContext when you register an Op: it contains all the metadata for the Op, like Traits, Interfaces, canonicalization patterns, folding hook, …**

A vtable is just a struct defining function pointers…

```
private:
 struct Concept {
   virtual ~Concept() = default;
   virtual int
    getNumParallelLoops(Operation *) const = 0;
 };
 const Concept *self;

 template <typename ConcreteOp>
 struct Model : Concept {
   int getNumParallelLoops(Operation *op)
                              const override {
     return cast<ConcreteOp>(op).getNumParallelLoops();
   }
 };
```

Pointer-to-Pointer to the vtable.

A vtable is just a struct defining function pointers…

C++ doesn’t allow you to get a direct pointer to a vtable… But we can implement one ourselves!

```
private:
 struct Concept {
   virtual ~Concept() = default;
   virtual int getNumParallelLoops(Operation *) const = 0;
 };
 const Concept *self;

 template <typename T>
 struct Model : Concept {
   int getNumParallelLoops(Operation *op)
                               const override {
     return cast<T>(op).getNumParallelLoops();
   }
 };
```

A vtable is just a struct defining function pointers…

C++ doesn't allow you to get a direct pointer to a vtable… But we can implement one ourselves!

```
private:
 struct Concept {
   virtual ~Concept() = default;
   virtual int getNumParallelLoops(Operation *) const = 0;
 };
 const Concept *self;

 template <typename T>
 struct Model : Concept {
   int getNumParallelLoops(Operation *op)
                                            const override {
     return cast<T>(op).getNumParallelLoops();
   }
 };
```

Virtual method becomes a function pointer!

Can we do better? Take by value? Tradeoff…

"Override" implementation becomes static methods

The *Concept* class is just the "vtable" now

Direct pointer to the "vtable", save one load when calling a method on the interface!

```
private:
 struct Concept {
   int (*getNumParallelLoops)(Operation *);
 };

 Concept *self;

 template <typename ConcreteOp>
 struct Model : public Concept {
   Model() : Concept{getNumParallelLoops} {}
   static int getNumParallelLoops(Operation *op) {
     return cast<ConcreteOp>(op).
                           getNumParallelLoops();
   }
 };
```

MLIR Interfaces: cast<>/dyn_cast<>

```
/// Returns the impl interface instance for the given operation.
static typename InterfaceBase::Concept *getInterfaceFor(Operation *op) {
  OperationName name = op->getName();

  // Access the raw interface from the operation info.
  if (std::optional<RegisteredOperationName> rInfo =
          name.getRegisteredInfo()) {
    if (auto *opIface = rInfo->getInterface<ConcreteType>())
      return opIface;
  }
  // Fallback to the dialect to provide it with a chance to implement this
  // interface for this operation.
  if (Dialect *dialect = name.getDialect())
    return dialect->getRegisteredInterfaceForOp<ConcreteType>(name);
  return nullptr;
}
```

OpInterface registered on the Operation (map lookup)

If an operation does not provide an interface, the dialect can still provide it!

https://github.com/llvm/llvm-project/blob/4cb110a84f587d3c65b85d79ab6fc8aa5489fb86/mlir/include/mlir/IR/OpDefinition.h#L2057-L2085

ODS Generation for OpInterface

```
def ExampleOpInterface :
        OpInterface<"ExampleOpInterface"> {
 let methods = [
   InterfaceMethod<
     "Example of a non-static method.",
     "unsigned", "exampleInterfaceHook",
     /*args=*/(ins)



   >,
   StaticInterfaceMethod<
     "Example of a static method.",
     "unsigned", "exampleStaticInterfaceHook",
     /*args=*/(ins)



   >,
 ];
}
```

```
struct ExampleOpInterfaceInterfaceTraits {
 struct Concept {
   unsigned (*exampleInterfaceHook)(const Concept *impl,
                                    ::mlir::Operation *);
   unsigned (*exampleStaticInterfaceHook)();
 };
```

ODS Generation for OpInterface

```
def ExampleOpInterface :
        OpInterface<"ExampleOpInterface"> {
 let methods = [
   InterfaceMethod<
     "Example of a non-static method.",
     "unsigned", "exampleInterfaceHook",
     /*args=*/(ins)


   >,
   StaticInterfaceMethod<
     "Example of a static method.",
     "unsigned", "exampleStaticInterfaceHook",
     /*args=*/(ins)


   >,
 ];
}
```

```
struct ExampleOpInterfaceInterfaceTraits  {
 struct Concept {
   unsigned (*exampleInterfaceHook)(const Concept *impl,
                                       ::mlir::Operation *);
   unsigned (*exampleStaticInterfaceHook)();
 };
 template<typename ConcreteOp> class Model : public Concept
 public:
   Model() : Concept{exampleInterfaceHook,
                     exampleStaticInterfaceHook} {}
   static inline unsigned exampleInterfaceHook(
        const Concept *impl, ::mlir::Operation *op) {
     return cast<ConcreteOp>(op).exampleInterfaceHook();
   }
   static inline unsigned exampleStaticInterfaceHook() {
     return ConcreteOp::exampleStaticInterfaceHook()
   }
```

The "static" variant is still a "virtual" dispatch!

The "static" variant calls a static method on the op

The "static" variant does not take "state" arguments.

ODS Generation for OpInterface

```
def ExampleOpInterface :
        OpInterface<"ExampleOpInterface"> {
 let methods = [
   InterfaceMethod<
     "Example of a non-static method.",
     "unsigned", "exampleInterfaceHook",
     /*args=*/(ins),
     /*methodBody=*/[{ /* methodBody */}]

   >,
   StaticInterfaceMethod<
     "Example of a static method.",
     "unsigned", "exampleStaticInterfaceHook",
     /*args=*/(ins),
     /*methodBody=*/[{ /* staticMethodBody */}]

   >,
 ];
}
```

methodBody overrides the default behavior of the interface for all operations!

```
struct ExampleOpInterfaceInterfaceTraits {
 struct Concept {
   unsigned (*exampleInterfaceHook)(const Concept *impl,
                                    ::mlir::Operation *);
   unsigned (*exampleStaticInterfaceHook)();
 };
 template<typename ConcreteOp> class Model : public Concept
 public:
   Model() : Concept{exampleInterfaceHook,
                     exampleStaticInterfaceHook} {}
   static inline unsigned exampleInterfaceHook(
        const Concept *impl, ::mlir::Operation *op) {
     /* methodBody */
   }
   static inline unsigned exampleStaticInterfaceHook() {
     /* staticMethodBody */
   }
```

ODS Generation for OpInterface

```
def ExampleOpInterface :
        OpInterface<"ExampleOpInterface"> {
 let methods = [
   InterfaceMethod<
     "Example of a non-static method.",
     "unsigned", "exampleInterfaceHook",
     /*args=*/(ins),
     /*methodBody=*/[{ /* methodBody */}]

   >,
   StaticInterfaceMethod<
     "Example of a static method.",
     "unsigned", "exampleStaticInterfaceHook",
     /*args=*/(ins),
     /*methodBody=*/[{ /* staticMethodBody */}]

   >,
 ];
}
```

```
struct ExampleOpInterfaceInterfaceTraits  {
 struct Concept {
   unsigned (*exampleInterfaceHook)(const Concept *impl,
                                    ::mlir::Operation *);
   unsigned (*exampleStaticInterfaceHook)();
 };
 template<typename ConcreteOp> class Model : public Concept
 public:
   Model() : Concept{exampleInterfaceHook,
                     exampleStaticInterfaceHook} {}
   static inline unsigned exampleInterfaceHook(
        const Concept *impl, ::mlir::Operation *op) {
     /* methodBody */
   }
   static inline unsigned exampleStaticInterfaceHook() {
     /* staticMethodBody */
   }
```

methodBody overrides the default behavior of the interface for all operations!

Example: define the interface in terms of a combination of operation properties

```
InterfaceMethod<"",
  "unsigned", "getNumInputsAndOutputs", (ins), /*methodBody=*/[{
    return $_op.getNumInputs() + $_op.getNumOutputs();
}]>,
```

=> Mental Model: it's like defining non-virtual method on the base class.

ODS Generation for OpInterface

```
def ExampleOpInterface :
        OpInterface<"ExampleOpInterface"> {
 let methods = [
   InterfaceMethod<
     "Example of a non-static method.",
     "unsigned", "exampleInterfaceHook",
     /*args=*/(ins),
     /*methodBody=*/[{ /* methodBody */}]

   >,
   StaticInterfaceMethod<
     "Example of a static method.",
     "unsigned", "exampleStaticInterfaceHook",
     /*args=*/(ins),
     /*methodBody=*/[{ /* staticMethodBody */}]

   >,
 ];
}
```

```
template <typename ConcreteOp>
struct ExampleOpInterfaceTrait :
  public ::mlir::OpInterface<ExampleOpInterface,
                         ...>::Trait<ConcreteOp> {
```

```
struct ExampleOpInterfaceInterfaceTraits  {
 struct Concept {
   unsigned (*exampleInterfaceHook)(const Concept *impl,
                                  ::mlir::Operation *);
   unsigned (*exampleStaticInterfaceHook)();
 };
 template<typename ConcreteOp> class Model : public Concept
 public:
   Model() : Concept{exampleInterfaceHook,
                     exampleStaticInterfaceHook} {}
   static inline unsigned exampleInterfaceHook(
          const Concept *impl, ::mlir::Operation *op) {
      /* methodBody */
   }
   static inline unsigned exampleStaticInterfaceHook() {
     /* staticMethodBody */
   }
```

Trait is automatically added as base class of Ops implementing the interface

ODS Generation for OpInterface

```
def ExampleOpInterface :
        OpInterface<"ExampleOpInterface"> {
 let methods = [
   InterfaceMethod<
     "Example of a non-static method.",
     "unsigned", "exampleInterfaceHook",
     /*args=*/(ins),
     /*methodBody=*/[{}],
     /*defaultImplementation=*/[{ /* Impl */}]
   >,
   StaticInterfaceMethod<
     "Example of a static method.",
     "unsigned", "exampleStaticInterfaceHook",
     /*args=*/(ins),
     /*methodBody=*/[{}],
     /*defaultImplementation=*/[{ /* StaticImpl */}]
   >,
 ];
}
```

```
struct ExampleOpInterfaceInterfaceTraits {
 struct Concept {
   unsigned (*exampleInterfaceHook)(const Concept *impl,
                                  ::mlir::Operation *);
   unsigned (*exampleStaticInterfaceHook)();
 };
 template<typename ConcreteOp> class Model : public Concept
 public:
   Model() : Concept{exampleInterfaceHook,
                     exampleStaticInterfaceHook} {}
   static inline unsigned exampleInterfaceHook(
        const Concept *impl, ::mlir::Operation *op) {
     return cast<ConcreteOp>(op).exampleInterfaceHook();
   }
   static inline unsigned exampleStaticInterfaceHook() {
     return ConcreteOp::exampleStaticInterfaceHook();
   }
```

```
template <typename ConcreteOp>
struct ExampleOpInterfaceTrait :
  public ::mlir::OpInterface<ExampleOpInterface,
                               ...>::Trait<ConcreteOp> {
  unsigned exampleInterfaceHook() {
    /* Impl */
  }
  static unsigned exampleStaticInterfaceHook() {
    /* StaticImpl */
  }
```

All operations inherit these methods, but can override!

```
DeclareOpInterfaceMethods<
    ExampleOpInterface,
    ["exampleInterfaceHook"]>
```

=> Mental Model: it's like adding default impl. to virtual methods in the base class

External Interfaces Model

Most of the time the OpInterface is attached to the operation in ODS

Problem: attaching OpInterface implementation to dialect comes with a lot of dependencies, possibly bloating effect for users.

=> Solution: “external interfaces”

```
void mlir::scf::registerBufferizableOpInterfaceExternalModels(
   DialectRegistry &registry) {
 registry.addExtension(+[](MLIRContext *ctx, scf::SCFDialect *dialect) {
   ConditionOp::attachInterface<ConditionOpInterface>(*ctx);
   ExecuteRegionOp::attachInterface<ExecuteRegionOpInterface>(*ctx);
   ForOp::attachInterface<ForOpInterface>(*ctx);
   ...
```

- Using SCF dialect does not imply linking in the bufferization patterns and all the bufferization dialect (and transitive dependencies…)
- Users must explicitly call `registerBufferizableOpInterfaceExternalModels` to be able to bufferize SCF dialect

External Interfaces Model & Promises

External Interfaces Model are a footgun!

What happened if you use SCF, try to call the bufferization, but never called `registerBufferizableOpInterfaceExternalModels`?

=> Long hours of debugging…

Other example, a downstream compiler can be setup as:

- Load Tosa dialect
- Emit Tosa Ops
- Build a pass pipeline: `compileTosaToLLVM()`
- Run the pipeline

`Upstream can introduce new dialects implicitly loaded here and new external interface`

=> Miscompile (or missing optimization)

=> Long hours of debugging…

External Interfaces Model & Promises

Solution: “promises”

```
void ControlFlowDialect::initialize() {
  declarePromisedInterfaces<bufferization::BufferizableOpInterface, BranchOp, CondBranchOp>();
```

No build or link-time dependency, header-only

dependency on the `TypeID<BufferizableOpInterface>`

```
auto bufferizableOp = dyn_cast<BufferizableOpInterface >(op);
```

LLVM ERROR: checking for an interface (`mlir::bufferization::BufferizableOpInterface`) that was promised by dialect 'cf' but never implemented. This is generally an indication that the dialect extension implementing the interface was never registered.

=> Missing:

```
cf::registerBufferizableOpInterfaceExternalModels (registry);
```

Takeaways

- An interface is all just a “virtual table”, manually implemented as a struct of function pointers (the “Model”)
- Each op has a map of *TypeID<Interface>* => *<Model*>*
- Op registration automatically instantiate all the static *<Model*>*
- “External Model” registration means adding an entry in the map for an operation post-op-registration.
- Dialects can provide a fallback Model (for all ops in the dialect)
- Promise are a necessary safety feature

Didn’t cover today: Interface Inheritance, Attr/Type & Dialect Interfaces, details of Dialect Fallback…