Defining operations, types, attributes, and dialects at runtime.

Mathieu Fehr and others
Motivation

Currently, Operations and Types can only be defined at compile-time.
Motivation

Currently, Operations and Types can only be defined at compile-time.

```python
def ConstantOp : Toy_Op<"constant"> {  
    let summary = "constant operation";
    let description = [{ ... }];

    let arguments = (ins F64ElementsAttr:$value);

    let results = (outs F64Tensor);

    let verifier = [{ return ::verify(*this); }];

    let printer = [{ return ::print(printer, *this); }];
    let parser = [{ return ::parseConstantOp(parser, result); }];
}
```
Motivation

Defining operations (or types, dialects, ...) at runtime would let users:
Motivation

Defining operations (or types, dialects, …) at runtime would let users:

- define operations with metaprogramming
Motivation

Defining operations (or types, dialects, …) at runtime would let users:

- define operations with metaprogramming
- define operations from a configuration file
Motivation

Defining operations (or types, dialects, …) at runtime would let users:

- define operations with metaprogramming
- define operations from a configuration file
- define operations from another language, like Python
Motivation

Defining operations (or types, dialects, …) at runtime would let users:

- define operations with metaprogramming
- define operations from a configuration file
- define operations from another language, like Python

without the need to generate C++, or recompile MLIR
RFC: Extensible dialects

Creating an ExtensibleDialect class, that allows the definition of operations and types at runtime.
RFC: Extensible dialects

Creating an ExtensibleDialect class, that allows the definition of operations and types at runtime.

We also allow the definition of traits and dialects at runtime with the MLIRContext.
RFC: Extensible dialects

Creating an ExtensibleDialect class, that allows the definition of operations and types at runtime.

We also allow the definition of traits and dialects at runtime with the MLIRContext.

First patch should be ready tomorrow, and will add dynamic operations and types.
Example: Adding complex numbers to "math"
Example: Adding complex numbers to "math"

We want to be able to parse and print:

```
func @foo(%re: f32, %im: f32) -> !math.complex<f32> {
  %res = math.make_complex(%re, %im) : f32
  return %0 : !math.complex<f32>
}
```
Example: Adding complex numbers to "math"

We want to be able to parse and print:

```swift
func @foo(%re: f32, %im: f32) -> !math.complex<f32> {
  %res = math.make_complex(%re, %im) : f32
  return %0 : !math.complex<f32>
}
```
Making a dialect extensible

def Math_Dialect : Dialect {
    let summary = "A math dialect";
    let name = "math";

    ...

}
def Math_Dialect : Dialect {
    let summary = "A math dialect";
    let name = "math";

    ...

    let isExtensible = 1;
}

Making a dialect extensible
Making a dialect extensible

```python
def Math_Dialect : Dialect {
  let summary = "A math dialect";
  let name = "math";

  ...

  let isExtensible = 1;
}
```

Note: We could make all dialects extensible by default.
Making a dialect extensible

```python
def Math_Dialect : Dialect {
    let summary = "A math dialect";
    let name = "math";

    ...

    let isExtensible = 1;
}
```

Note: We could make all dialects extensible by default.

We can also define a new dialect at runtime.

```python
mlirCtx->addDynamicDialect("math");
Dialect* mathDialect = mlirCtx->getOrLoadDialect("math");
```
Defining "!math.complex<T>"

Each dynamic type has a name

```cpp
auto name = "complex";
```
Defining "!math.complex<T>"

Each dynamic type has a name

```cpp
auto name = "complex";
```

The type name is used for parsing and printing, as well as looking up a type in a dialect.
Defining "!math.complex<T>"

Each dynamic type has a name

```cpp
auto name = "complex";
```

The type name is used for parsing and printing, as well as looking up a type in a dialect

```cpp
auto typeDefinition = dialect.getDynamicTypeDefinition("complex");
```
Defining "!math.complex\langle T\rangle"  

Dynamic types are always parameterized by attributes, and can define a verifier.
Defining "!math.complex<T>"

Dynamic types are always parameterized by attributes, and can define a verifier.

```cpp
auto argsVerifier = [](function_ref<InFlightDiagnostic()> emitError,
                        ArrayRef<Attribute> params) {
    // Check that there is exactly 1 parameter
    // Check that the parameter is a numeric type
    ...
};
```
Defining "!math.complex<T>"

Dynamic types are always parameterized by attributes, and can define a verifier.

```cpp
auto argsVerifier = [](function_ref<InFlightDiagnostic()> emitError, 
                      ArrayRef<Attribute> params) {

    // Check that there is exactly 1 parameter
    // Check that the parameter is a numeric type

    ...
};
```

Note: argsVerifier is a `unique_function`, and thus can have captures.
Defining "!math.complex<T>"

We can define a custom parser and printer
We can define a custom parser and printer

```cpp
auto parser = [] (DialectAsmParser &parser, vector<Attribute> &parsed) {
    ...
};

auto printer = [] (DialectAsmPrinter &printer, ArrayRef<Attribute> args) {
    ...
};
```
Defining "!math.complex\(<T>\)"

We can define a custom parser and printer

```cpp
auto parser = [](DialectAsmParser &parser, vector<Attribute> &parsed) {
    ...
};

auto printer = [](DialectAsmPrinter &printer, ArrayRef<Attribute> args) {
    ...
};
```

Otherwise, a default parser and printer is defined.

The default format would be here: "!math.complex\(<T>\)"
Defining "!math.complex<T>"

We can create a type definition, then register it in the dialect

```cpp
auto complexType = DynamicTypeDefinition::get(dialect, name, argsVerifier, parser, printer);
mathDialect->addDynamicType(complexType);
```
Defining "!math.complex<T>"

We can create a type definition, then register it in the dialect

```cpp
auto complexType =
    DynamicTypeDefinition::get(dialect, name, argsVerifier, parser, printer);
mathDialect->addDynamicType(complexType);
```

This already allows us to parse this function:

```cpp
func @foo(%c: !math.complex<f32>) -> {
    return %c : !math.complex<f32>
}
```
Defining "math.make_complex"
Defining "math.make_complex"

We need to write the verifier of "make_complex"

```c
auto verifier = [](Operation* op) {

}
```
Defining "math.make_complex"

We need to write the verifier of "make_complex"

```cpp
auto verifier = [](Operation* op) {
    if (op->getNumOperands() != 2 || op->getNumResults() != 1)
        return failure();
}
```
auto verifier = [](Operation* op) {
    if (op->getNumOperands() != 2 || op->getNumResults() != 1)
        return failure();

    auto dynType = op->getResult(0).dyn_cast<DynamicType>();
    if (!dynType) { return failure(); }
}

Defining "math.make_complex"

We need to write the verifier of "make_complex"
auto verifier = [](Operation* op) {
    if (op->getNumOperands() != 2 || op->getNumResults() != 1)
        return failure();
    auto dynType = op->getResult(0).dyn_cast<DynamicType>();
    if (!dynType) { return failure(); }
    if (dynType.getName() != "math.complex")
        return failure();
}

We need to write the verifier of "make_complex"
auto verifier = [](Operation* op) {
    if (op->getNumOperands() != 2 || op->getNumResults() != 1)
        return failure();

    auto dynType = op->getResult(0).dyn_cast<DynamicType>();
    if (!dynType) { return failure(); }

    if (dynType.getDefinition() != complexTypeDef)
        return failure();

    return success();
};

We need to write the verifier of "make_complex"
Defining "math.make_complex"

We need to write the verifier of "make_complex"

```cpp
auto verifier = [](Operation* op) {
    if (op->getNumOperands() != 2 || op->getNumResults() != 1)
        return failure();

    auto dynType = op->getResult(0).dyn_cast<DynamicType>();
    if (!dynType) { return failure(); }

    if (dynType.getDefinition() != complexTypeDef)
        return failure();

    return failure();
};
```
Defining "math.make_complex"

We need to write the verifier of "make_complex"

```cpp
auto verifier = [](Operation* op) {
    if (op->getNumOperands() != 2 || op->getNumResults() != 1)
        return failure();
    auto dynType = op->getResult(0).dyn_cast<DynamicType>();
    if (!dynType) { return failure(); }
    if (dynType.getDefinition() != complexTypeDef)
        return failure();
    if (dynType.getParams()[0].getType() != op->getOperand(0).getType())
        return failure();
    if (dynType.getParams()[0].getType() != op->getOperand(1).getType())
        return failure();
    return success();
};
```
Defining "math.make_complex"

We register an operation the same way we did with the type
Defining "math.make_complex"

We register an operation the same way we did with the type

```cpp
auto name = "make_complex";
auto verifier = [](Operation* op) {
    ...
};
```
Defining "math.make_complex"

We register an operation the same way we did with the type

```cpp
auto name = "make_complex";
auto verifier = [](Operation* op) {
    ...
};;
auto printer = [](OpAsmParser &parser, OperationState &state) {
    ...
};;
```
Defining "math.make_complex"

We register an operation the same way we did with the type

```cpp
auto name = "make_complex";
auto verifier = [] (Operation * op) {
    ...
};
auto printer = [] (OpAsmParser & parser, OperationState & state) {
    ...
};
auto parser = [] (DialectAsmParser & parser, Operation * op) {
    ...
};
```
We register an operation the same way we did with the type

```cpp
auto name = "make_complex";
auto verifier = [](Operation* op) {
    ...
};
auto printer = [](OpAsmParser &parser, OperationState &state) {
    ...
};
auto parser = [](DialectAsmParser &parser, Operation* op) {
    ...
};
auto opDef = DynamicOpDefinition::get(dialect, name, verifier, parser, printer);
mathDialect->addDynamicOp(opDef);
```
Using the math dialect

With this, we have successfully added the complex type and the "make_complex"
operation in the "math" dialect.
Using the math dialect

With this, we have successfully added the complex type and the "make_complex" operation in the "math" dialect.

We can now fully parse this function:

```swift
func @foo(%re: f32, %im: f32) -> !math.complex<f32> {
    %res = math.make_complex(%re, %im) : f32
    return %0 : !math.complex<f32>
}
```
Adding traits to operations
Adding traits to operations

We can add existing traits defined in C++

```cpp
std::vector<DynamicOpTrait> traits;
```
Adding traits to operations

We can add existing traits defined in C++

```cpp
std::vector<DynamicOpTrait> traits;
traits.push_back(mlirContext->getDynamicTrait<ExistingTrait>());
```
Adding traits to operations

We can add existing traits defined in C++

```cpp
std::vector<DynamicOpTrait> traits;
traits.push_back(mlirContext->getDynamicTrait<ExistingTrait>();
dialect.addDynamicOp(dialect, name, verifier, parser, printer, traits);
```
Adding traits to operations

We can also define new traits at runtime

```cpp
auto dynamicTrait = DynamicOpTrait::get([](Operation *op) {
    return success(op->getNumOperands() == 2);
});
```
Adding traits to operations

We can also define new traits at runtime

```cpp
auto dynamicTrait = DynamicOpTrait::get([](Operation *op) {
    return success(op->getNumOperands() == 2);
});

mlirContext->addDynamicTrait("PairOperandsTrait", dynamicTrait);
```
Adding traits to operations

We can also define new traits at runtime

```cpp
auto dynamicTrait = DynamicOpTrait::get([](Operation *op) {
    return success(op->getNumOperands() == 2);
});

mlirContext->addDynamicTrait("PairOperandsTrait", dynamicTrait);

traits.push_back(mlirContext->getDynamicTrait("PairOperandsTrait"));
dialect.addDynamicOp(dialect, name, verifier, parser, printer, traits);
```
Adding interfaces to operations
Adding interfaces to operations

TableGen definition of the MemoryEffects interface:

```cpp
class MemoryEffects<string name> : OpInterface{name} {
    let methods = [
        InterfaceMethod<
            "void", "getEffects",
            (ins "SmallVectorImpl<Effect> &":$effects)
        >,
    ];
}
```
Adding interfaces to operations

TableGen definition of the MemoryEffects interface:

```cpp
class MemoryEffects<string name> : OpInterface{name} {
    let methods = [
        InterfaceMethod<
            "void", "getEffects",
            (ins "SmallVectorImpl<Effect> &":$effects)
        >,
    ];
}
```

We define the implementation of each method, and create the interface implementation

```cpp
auto getEffects = []((SmallVectorImpl<Effect> &effects) {
    // no effects
}
auto memoryEffectsInterface = MemoryEffectsDynImpl::get(getEffects);
```
Adding interfaces to operations

TableGen definition of the MemoryEffects interface:

```cpp
class MemoryEffects<string name> : OpInterface{name} {
    let methods = [
        InterfaceMethod<
            "void", "getEffects",
            (ins "SmallVectorImpl<Effect> &":$effects)
        >,
    ];
}
```

We define the implementation of each method, and create the interface implementation:

```cpp
auto getEffects = []((SmallVectorImpl<Effect> &effects) {
    // no effects
}
auto memoryEffectsInterface = MemoryEffectsDynImpl::get(getEffects);
```
Adding interfaces to operations

We can give a list of interfaces when creating a new operation
Adding interfaces to operations

We can give a list of interfaces when creating a new operation

```cpp
vector<DynamicOpInterface> interfaces;
interfaces.push_back(memoryEffectsInterface);
dialect.addDynamicOp(dialect, name, verifier, parser, printer, traits, interfaces);
```
Interaction with traits and interface

We interact with the traits/interfaces the same way as with normal operations
Interaction with traits and interface

We interact with the traits/interfaces the same way as with normal operations

```cpp
op.hasTrait<ConcreteTrait>();
auto interface = dyn_cast<ConcreteOpInterface>(op);
```
Interaction with traits and interface

We interact with the traits/interfaces the same way as with normal operations

```cpp
op.hasTrait<ConcreteTrait>();
auto interface = dyn_cast<ConcreteOpInterface>(op);
```

In particular, we can reuse generic passes
Using interfaces in existing passes

```plaintext
module {
    func @foo(%re: f32, %im: f32) -> () {
        %res = math.make_complex(%re, %im) : f32
        return
    }
}
```
Using interfaces in existing passes

```mlir
module {
    func @foo(%re: f32, %im: f32) -> () {
        %res = math.make_complex(%re, %im) : f32
        return
    }
}
```

```bash
mlir-opt -canonicalize
```

```mlir
module {
    func @foo(%re: f32, %im: f32) -> () {
        return
    }
}
```
Completeness of operation definition
Completeness of operation definition

It is also possible to define the fold hook, and the canonicalization patterns:
Completeness of operation definition

It is also possible to define the fold hook, and the canonicalization patterns:

dialect.addDynamicOp(dialect, name, verifier, parser,
    printer, traits, interfaces,
    foldHook, getCanonicalizationPatterns);
How this is implemented?
How this is implemented?

Each C++ class defines at compile-time a unique TypeID.

```cpp
TypeID::get<Concrete0p>();
```
How this is implemented?

Each C++ class defines at compile-time a unique TypeID.

```
TypeID::get<ConcreteOp>();
```

In MLIR, TypeIDs uniquely identify ops, types, traits, interfaces, and dialects.
How this is implemented?

Each C++ class defines at compile-time a unique TypeID.

```cpp
TypeID::get<ConcreteOp>();
```

In MLIR, TypeIDs uniquely identify ops, types, traits, interfaces, and dialects.

```cpp
op.isa<ConcreteOp>();
```

```cpp
op.getTypeID() == typeid::get<ConcreteOp>();
```
How this is implemented?

Each C++ class defines at compile-time a unique TypeID.

```
TypeID::get<ConcreteOp>();
```

In MLIR, TypeIDs uniquely identify ops, types, traits, interfaces, and dialects.

```
op.isa<ConcreteOp>()
```

```
op.getTypeID() == TypeID::get<ConcreteOp>()
```

```
addOperation<ConcreteOp>()
```

```
addOperation(TypeID::get<ConcreteOp>(), ConcreteOp::print, ConcreteOp::parser, ...)
```
Defining a TypeID at runtime
Defining aTypeID at runtime

Internally, a TypeID represent the address of an allocated object.

So, the address of a malloc should be a unique TypeID.
Defining a TypeID at runtime

Internally, a TypeID represent the address of an allocated object.

So, the address of a malloc should be a unique TypeID.

```cpp
struct TypeIDAllocator {
    vector<unique_ptr<TypeID::Storage>> ids;
}
```
Defining a TypeID at runtime

Internally, a TypeID represent the address of an allocated object.

So, the address of a malloc should be a unique TypeID.

```cpp
struct TypeIDAllocator {
    TypeID allocateID() {
        ids.emplace_back(new TypeID::Storage());
        return TypeID(ids.back().get());
    }

    vector<unique_ptr<TypeID::Storage>> ids;
};
```
How registration of operations works

We define operations or dialects at runtime by opening the MLIR API a bit.
How registration of operations works

We define operations or dialects at runtime by opening the MLIR API a bit.

```c++
void ExtensibleDialect::addDynamicOperation(DynamicOpDefinition dynOp) {
    addOperation(ctx->allocateTypeID(), dynOp.verifier, dynOp.parser, ...);
}
```
How registration of operations works

We define operations or dialects at runtime by opening the MLIR API a bit.

```cpp
void ExtensibleDialect::addDynamicOperation(DynamicOpDefinition dynOp) {
    addOperation(ctx->allocateTypeID(), dynOp.verifier, dynOp.parser, ...);
}

void MLIRContext::addDynamicDialect(StringRef name) {
    registerDialect(ctx->allocateTypeID(), name);
}
```
How registration of types works

Types need in addition a storage for parameters.
How registration of types works

Types need in addition a storage for parameters.

```cpp
struct DynamicTypeStorage : public TypeStorage {

    ...

    /// The type definition.
    /// Contains the parser, printer, verifier.
    DynamicTypeDef *typeDef;

    /// The type parameters. Can contain arbitrary data.
    ArrayRef<Attribute> params;
};
```
How registration of types works

Types need in addition a storage for parameters.

```cpp
struct DynamicTypeStorage : public TypeStorage {
    ...

    /// The type definition.
    /// Contains the parser, printer, verifier.
    DynamicTypeDefinition *typeDef;

    /// The type parameters. Can contain arbitrary data.
    ArrayRef<Attribute> params;
};

class DynamicType
    : public Type::TypeBase<DynamicType, Type, detail::DynamicTypeStorage> {
    ...
}
```
How interface implementation works

Interfaces are handled in operations, dialects, and types by InterfaceMap.

class InterfaceMap {
    ...
    SmallVector<std::pair<TypeID, void *>> interfaces;
};
How interface implementation works

Interfaces are handled in operations, dialects, and types by InterfaceMap.

```cpp
class InterfaceMap {
    ...
    SmallVector<std::pair<TypeID, void *>> interfaces;
};
```

The void* elements are structs of function pointers, representing the method implementations. They are called Model or Concept.
How interface implementation works

class MemoryEffects<string name> : OpInterface{name} {
  let methods = [
    InterfaceMethod<
      "void", "getEffects",
      (ins "SmallVectorImpl<Effect> &":$effects)>
  ];
}

How interface implementation works

class MemoryEffects<

let methods = [ InterfaceMethod< 
  "void", "getEffects", 
  (ins "SmallVectorImpl<Effect> &":$effects)
>, ];
}

struct MemoryEffectsModel {
  void (*getEffects)(Operation* op, smallVectorImpl<Effect> &effects);
};

TableGen generation
How interface implementation works

```cpp
class MemoryEffects<string name> : OpInterface{name} {
    let methods = [
        InterfaceMethod<
            "void", "getEffects",
            (ins "SmallVectorImpl<Effect> &":$effects)
        ],
    ];
}
```

```
struct MemoryEffectsModel {
    void (*getEffects)(Operation* op, smallVectorImpl<Effect> &effects);
};
```

We would like to have unique_function instead
How interface implementation works

We generate a unique_function version that we store in the ExtensibleDialect.

```cpp
// Generated by TableGen
struct MemoryEffectsDynModel {
    getEffects unique_function<void(Operation* op, smallVectorImpl<Effect> &effects)>;
};
```
How interface implementation works

We generate a unique_function version that we store in the ExtensibleDialect.

```cpp
// Generated by TableGen
struct MemoryEffectsDynModel {
    getEffects unique_function<void(Operation* op, smallVectorImpl<Effect> &effects)>
};
```

And we retrieve it from the original Model

```cpp
// Generated by TableGen
void getEffects(Operation* op, smallVectorImpl<Effect> &effects) {
    auto dialect = op->getDialect().cast<ExtensibleDialect>();
    auto model = dialect->getDynamicModelFor(op, TypeID::get<ConcreteInterface>());
    return model.method1(...);
}
```
Summary

We can define at runtime new operations, types, traits, and dialects.

We can reuse existing passes on the dynamic operations.
Summary

We can define at runtime new operations, types, traits, and dialects.

We can reuse existing passes on the dynamic operations.

Our approach, in combination with proper Python bindings, should allow users to define entire dialects in Python without the need to recompile MLIR.