How to Build your own MLIR Dialect

February 04, 2023

Marius Brehler <marius.brehler@iml.fraunhofer.de>

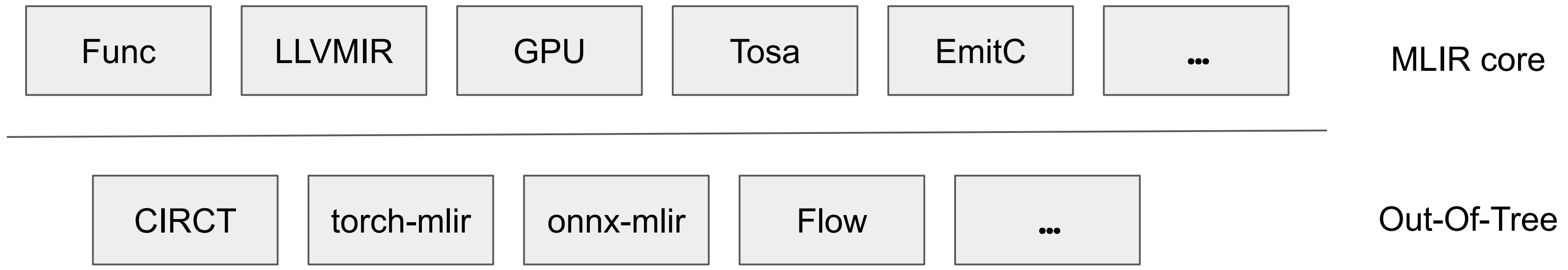
Fraunhofer IML

Outline

- Multi-Level Intermediate Representation
- The Standalone Example
- Extending the Standalone Example for `LLVM_EXTERNAL_PROJECTS`
- Using your Dialect in Other Projects
- Adding Types (and Attributes) to your Dialect

Multi-Level Intermediate Representation

- MLIR is a framework to build a reusable and extensible compiler infrastructure
- Operations, attributes and types, forming an intermediate representation (IR), are grouped in dialects
- Dialects can be part of the MLIR core (upstream) or live in separate repositories



The Standalone Example

- An example of an out-of-tree MLIR dialect: https://github.com/llvm/llvm-project/tree/main/mlir/examples/standalone

- Build the standalone example with:

```
mkdir build && cd build
cmake -G Ninja .. -DMLIR_DIR=$PREFIX/lib/cmake/mlir -DLLVM_EXTERNAL_LIT=$BUILD_DIR/bin/llvm-lit
cmake --build . --target check-standalone
```

- This assumes that you have built LLVM and MLIR in `$BUILD_DIR` and installed them to `$PREFIX`

The Standalone Example

- An example of an out-of-tree MLIR dialect:
  https://github.com/llvm/llvm-project/tree/main/mlir/examples/standalone

- Build the standalone example with:

```
mkdir build && cd build
cmake -G Ninja .. -DMLIR_DIR=$PREFIX/lib/cmake/mlir -DLLVM_EXTERNAL_LIT=$BUILD_DIR/bin/llvm-lit
cmake --build . --target check-standalone
```

- This assumes that you have built LLVM and MLIR in `$BUILD_DIR` and installed them to `$PREFIX`

In-tree and Out-of-tree – Definitions

In the **LLVM** world

- In-tree refers to a monolithic build
- In-tree *can* refer to the source location

- Out-of-tree normally refers to work with a separate repository (source location)

- Projects call building with the `LLVM_EXTERNAL_PROJECTS` mechanism
  - *In-tree* or
  - *Out-of-tree, monolithic build*

In the **CMake** world

- In-tree refers to build in the directory where the source code is located

- Out-of-tree refers to build in a separate build directory (build location)

The `LLVM_EXTERNAL_PROJECTS` mechanism

- Build the standalone example with:

```
mkdir build && cd build

cmake -G Ninja `$LLVM_SRC_DIR/llvm` \
-DLLVM_TARGETS_TO_BUILD=host -DCMAKE_BUILD_TYPE=Release \
-DLLVM_ENABLE_PROJECTS=mlir \
-DLLVM_EXTERNAL_PROJECTS=standalone-dialect -DLLVM_EXTERNAL_STANDALONE_DIALECT_SOURCE_DIR=../

cmake --build . --target check-standalone
```

- `LLVM_EXTERNAL_PROJECTS` defines the external projects to be built
- `LLVM_EXTERNAL_STANDALONE_DIALECT_SOURCE_DIR`defines the source code location
- `$LLVM_SRC_DIR` points to the root of the monorepo

CMakeLists.txt – Out-of-tree Source, Component Build

```
find_package(MLIR REQUIRED CONFIG)

message(STATUS "Using MLIRConfig.cmake in: ${MLIR_DIR}")
message(STATUS "Using LLVMConfig.cmake in: ${LLVM_DIR}")

set(LLVM_RUNTIME_OUTPUT_INTDIR ${CMAKE_BINARY_DIR}/bin)
set(LLVM_LIBRARY_OUTPUT_INTDIR ${CMAKE_BINARY_DIR}/lib)
set(MLIR_BINARY_DIR ${CMAKE_BINARY_DIR})

list(APPEND CMAKE_MODULE_PATH "${MLIR_CMAKE_DIR}")
list(APPEND CMAKE_MODULE_PATH "${LLVM_CMAKE_DIR}")
include(TableGen)
include(AddLLVM)
include(AddMLIR)
include(HandleLLVMOptions)
```

- `find_package()` imports information which were exported by a project
- `find_package(MLIR ..)` imports information from the installed MLIR and calls `find_package(LLVM ..)`

CMakeLists.txt – External Project Mechanism Build

```
if(CMAKE_SOURCE_DIR STREQUAL CMAKE_CURRENT_SOURCE_DIR)
 find_package(MLIR REQUIRED CONFIG)

 message(STATUS "Using MLIRConfig.cmake in: ${MLIR_DIR}")
 message(STATUS "Using LLVMConfig.cmake in: ${LLVM_DIR}")

 set(LLVM_RUNTIME_OUTPUT_INTDIR ${CMAKE_BINARY_DIR}/bin)
 set(LLVM_LIBRARY_OUTPUT_INTDIR ${CMAKE_BINARY_DIR}/lib)
 set(MLIR_BINARY_DIR ${CMAKE_BINARY_DIR})

 list(APPEND CMAKE_MODULE_PATH "${MLIR_CMAKE_DIR}")
 list(APPEND CMAKE_MODULE_PATH "${LLVM_CMAKE_DIR}")

 include(TableGen)
 include(AddLLVM)
 include(AddMLIR)
 include(HandleLLVMOptions)
else()
 [..]
```

- Only call `find_package(MLIR ..)` if built against an installed MLIR
- `CMAKE_SOURCE_DIR` is *normally* equal to `CMAKE_CURRENT_SOURCE_DIR` this case
- No need to load CMake modules (`include(AddLLVM)` etc.) when using the external project mechanism

CMakeLists.txt – External Project Mechanism Build

```
 [..]
else()
 # Build via external project mechanism
 set(MLIR_MAIN_SRC_DIR ${LLVM_MAIN_SRC_DIR}/../mlir)
 set(MLIR_INCLUDE_DIR ${MLIR_MAIN_SRC_DIR}/include)
 set(MLIR_GENERATED_INCLUDE_DIR ${LLVM_BINARY_DIR}/tools/mlir/include)
 set(MLIR_INCLUDE_DIRS "${MLIR_INCLUDE_DIR};${MLIR_GENERATED_INCLUDE_DIR}")
endif()
```

- The exported project settings are not yet available
- Set the variables manually
- Fix the lit tests

External Project Mechanism Build

```
set(STANDALONE_SOURCE_DIR ${PROJECT_SOURCE_DIR})
set(STANDALONE_BINARY_DIR ${PROJECT_BINARY_DIR})
include_directories(${LLVM_INCLUDE_DIRS})
include_directories(${MLIR_INCLUDE_DIRS})
include_directories(${PROJECT_SOURCE_DIR}/include)
include_directories(${PROJECT_BINARY_DIR}/include)
include_directories(${STANDALONE_SOURCE_DIR}/include)
include_directories(${STANDALONE_BINARY_DIR}/include)
link_directories(${LLVM_BUILD_LIBRARY_DIR})
add_definitions(${LLVM_DEFINITIONS})
```

```
config.standalone_obj_root = "@CMAKE_BINARY_DIR@"
config.standalone_obj_root = "@STANDALONE_BINARY_DIR@"

[..]

lit_config.load_config(config, "@CMAKE_SOURCE_DIR@/test/lit.cfg.py")
lit_config.load_config(config, "@STANDALONE_SOURCE_DIR@/test/lit.cfg.py")
```

- Define project specific variables in the `CMakeLists.txt` to be used in `test/lit.site.cfg.py.in`
- The regression test otherwise try to parse config file from `llvm/test/lit.cfg.py`
- Adjust `test/lit.site.cfg.py` accordingly

Using your Dialect in other Projects

If your dialect should be used in another project, you can

- Build all dialects via the external projects mechanism, for example see torch-mlir:
  `LLVM_EXTERNAL_PROJECTS="torch-mlir;torch-mlir-dialects"`

- You can also use CMake's `External_Project_Add()`

- You can add projects via add_sudirectory(), for example like mlir-emitc:
  `add_subdirectory(third_party/mlir-hlo EXCLUDE_FROM_ALL)`

- For `add_subdirectory()` a new CMake option can be introduced:

```
option(EMITC_BUILD_EMBEDDED "Build EmitC as part of another project" OFF)
```

Using your Dialect in other Projects

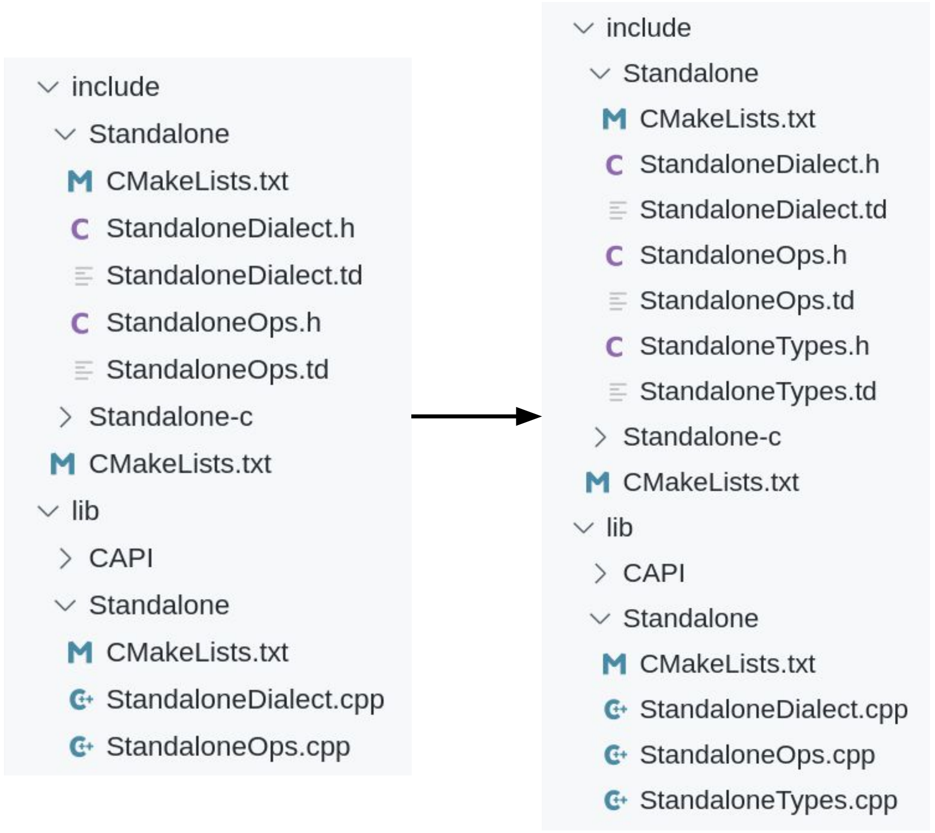
```
if(CMAKE_SOURCE_DIR STREQUAL CMAKE_CURRENT_SOURCE_DIR AND NOT EMITC_BUILD_EMBEDDED)
 find_package(MLIR REQUIRED CONFIG)

 [..]

 list(APPEND CMAKE_MODULE_PATH "${MLIR_CMAKE_DIR}")
 list(APPEND CMAKE_MODULE_PATH "${LLVM_CMAKE_DIR}")
endif()

if(CMAKE_SOURCE_DIR STREQUAL CMAKE_CURRENT_SOURCE_DIR OR EMITC_BUILD_EMBEDDED)
 # Out-of-tree build or embedded into another project
 include(TableGen)
 include(AddLLVM)
 include(AddMLIR)
 include(HandleLLVMOptions)
else()
 # Build via external projects mechanism
 [..]
```

Extending your Dialect with Types



```
include
  Standalone
    CMakeLists.txt
    StandaloneDialect.h
    StandaloneDialect.td
    StandaloneOps.h
    StandaloneOps.td
  Standalone-c
  CMakeLists.txt
lib
  CAPI
  Standalone
    CMakeLists.txt
    StandaloneDialect.cpp
    StandaloneOps.cpp
```

```
include
  Standalone
    CMakeLists.txt
    StandaloneDialect.h
    StandaloneDialect.td
    StandaloneOps.h
    StandaloneOps.td
    StandaloneTypes.h
    StandaloneTypes.td
  Standalone-c
  CMakeLists.txt
lib
  CAPI
  Standalone
    CMakeLists.txt
    StandaloneDialect.cpp
    StandaloneOps.cpp
    StandaloneTypes.cpp
```

- Types could be specified in `StandaloneOps.td`
- However, separate files already exist for the dialect and its operations
- We add new files for the our types
  - `StandaloneTypes.td`
  - `StandaloneTypes.h`
  - `StandaloneTypes.cpp`

Types – `include/Standalone/StandaloneTypes.td`

```
#ifndef STANDALONE_TYPES
#define STANDALONE_TYPES

include "mlir/IR/AttrTypeBase.td"
include "Standalone/StandaloneDialect.td"

class Standalone_Type<string name, string typeMnemonic, list<Trait> traits = []>
   : TypeDef<Standalone_Dialect, name, traits> {
 let mnemonic = typeMnemonic;
}

def Standalone_CustomType : Standalone_Type<"Custom", "custom"> {
   let summary = "Standalone custom type";
   let description = "Custom type in standalone dialect";
   let parameters = (ins StringRefParameter<"the custom value">:$value);
   let assemblyFormat = "`<` $value `>`";
}

#endif // STANDALONE_TYPES
```

Types – `include/Standalone/StandaloneOps.td`

```
#ifndef STANDALONE_OPS
#define STANDALONE_OPS

include "Standalone/StandaloneDialect.td"
include "Standalone/StandaloneTypes.td"
include "mlir/Interfaces/InferTypeOpInterface.td"
include "mlir/Interfaces/SideEffectInterfaces.td"

[..]

#endif // STANDALONE_OPS
```

- `StandaloneTypes.td` already includes `StandaloneDialect.td`

Types – `include/Standalone/CMakeLists.txt`

```
add_mlir_dialect(StandaloneOps standalone)
add_mlir_doc(StandaloneDialect StandaloneDialect Standalone/ -gen-dialect-doc)
add_mlir_doc(StandaloneOps StandaloneOps Standalone/ -gen-op-doc)
```

- No changes to the `CMakeLists.txt` file required

```
function(add_mlir_dialect dialect dialect_namespace)
 set(LLVM_TARGET_DEFINITIONS ${dialect}.td)
 mlir_tablegen(${dialect}.h.inc -gen-op-decls)
 mlir_tablegen(${dialect}.cpp.inc -gen-op-defs)
 mlir_tablegen(${dialect}Types.h.inc -gen-typedef-decls -typedefs-dialect=${dialect_namespace})
 mlir_tablegen(${dialect}Types.cpp.inc -gen-typedef-defs -typedefs-dialect=${dialect_namespace})
 mlir_tablegen(${dialect}Dialect.h.inc -gen-dialect-decls -dialect=${dialect_namespace})
 mlir_tablegen(${dialect}Dialect.cpp.inc -gen-dialect-defs -dialect=${dialect_namespace})
 add_public_tablegen_target(MLIR${dialect}IncGen)
 add_dependencies(mlir-headers MLIR${dialect}IncGen)
endfunction()
```

Attributes – `include/Standalone/CMakeLists.txt`

```
add_mlir_dialect(StandaloneOps standalone)
add_mlir_doc(StandaloneDialect StandaloneDialect Standalone/ -gen-dialect-doc)
add_mlir_doc(StandaloneOps StandaloneOps Standalone/ -gen-op-doc)
```

- This is different if defining attributes!
- `add_mlir_dialect()` doesn't call `mlir_tablegen()` for attributes
- Enhance the `CMakeLists.txt` file by

```
set(LLVM_TARGET_DEFINITIONS StandaloneAttributes.td)
mlir_tablegen(StandaloneAttributes.h.inc -gen-attrdef-decls)
mlir_tablegen(StandaloneAttributes.cpp.inc -gen-attrdef-defs)
add_public_tablegen_target(MLIRStandaloneAttributesIncGen)
```

Types – `include/Standalone/StandaloneTypes.h`

```
#ifndef STANDALONE_STANDALONETYPES_H
#define STANDALONE_STANDALONETYPES_H

#include "mlir/IR/BuiltinTypes.h"

#define GET_TYPEDEF_CLASSES
#include "Standalone/StandaloneOpsTypes.h.inc"

#endif // STANDALONE_STANDALONETYPES_H
```

- Include the auto-generated typedef classes header in the new header file

Types – `lib/Standalone/StandaloneTypes.cpp`

```cpp
#include "Standalone/StandaloneTypes.h"

#include "Standalone/StandaloneDialect.h"
#include "mlir/IR/Builders.h"
#include "mlir/IR/DialectImplementation.h"
#include "llvm/ADT/TypeSwitch.h"

using namespace mlir::standalone;

#define GET_TYPEDEF_CLASSES
#include "Standalone/StandaloneOpsTypes.cpp.inc"

void StandaloneDialect::registerTypes() {
 addTypes<
#define GET_TYPEDEF_LIST
#include "Standalone/StandaloneOpsTypes.cpp.inc"
     >();
}
```

- Include the auto-generated typedef classes as the full definition of the storage classes must be visible
- Provide the implementation of a function that calls the method `addTypes()` which adds the types

Types – `lib/Standalone/StandaloneDialect.cpp`

```
#include "Standalone/StandaloneTypes.h"

[..]

void StandaloneDialect::initialize() {
 addOperations<
#define GET_OP_LIST
#include "Standalone/StandaloneOps.cpp.inc"
     >();
 registerTypes();
}
```

- Register the types in the initialization of the parent dialect

Types – `lib/Standalone/CMakeLists.txt`

```
add_mlir_dialect_library(MLIRStandalone
        StandaloneTypes.cpp
        StandaloneDialect.cpp
        StandaloneOps.cpp



        ADDITIONAL_HEADER_DIRS
        ${PROJECT_SOURCE_DIR}/include/Standalone

        DEPENDS
        MLIRStandaloneOpsIncGen



[..]
```

- Include the new source file in your `CMakeLists.txt`.

Attributes – `lib/Standalone/CMakeLists.txt`

```
add_mlir_dialect_library(MLIRStandalone
        StandaloneTypes.cpp
        StandaloneDialect.cpp
        StandaloneOps.cpp
        StandaloneAttributes.cpp

        ADDITIONAL_HEADER_DIRS
        ${PROJECT_SOURCE_DIR}/include/Standalone

        DEPENDS
        MLIRStandaloneOpsIncGen
        MLIRStandaloneAttributesIncGen

[..]
```

- Include the new source file in your `CMakeLists.txt`.
- Add a dependency to `MLIRStandaloneOpsIncGen` to make sure TableGen generates the required code before

Types – `include/Standalone/StandaloneDialect.td`

```
def Standalone_Dialect : Dialect {

  [..]

  let useDefaultTypePrinterParser = 1;
  let useFoldAPI = kEmitFoldAdaptorFolder;

  let extraClassDeclaration = [{
      void registerTypes();
  }];
}
```

- Let TableGen generate the default printer and parser
- Let TableGen generate a class declaration for
  `void registerTypes();`

How to Build your own MLIR Dialect

- The modifications to the standalone example are available at: https://github.com/marbre/llvm-project/tree/standalone-example-fosdem-2023

- The modifications to the standalone example will be sent via Pharicator to be reviewed for upstream inclusion

Further Resources

- M. Amini & R. Riddle "MLIR Tutorial", 2020 LLVM Developers' Meeting, https://youtu.be/Y4SvqTtOIDk
- https://mlir.llvm.org/docs/Tutorials/CreatingADialect/
- https://mlir.llvm.org/docs/Tutorials/Toy/
- https://mlir.llvm.org/docs/DefiningDialects/AttributesAndTypes/
- S. Neuendorffer, "Architecting out-of-tree LLVM projects using cmake", 2021 LLVM Developers' Meeting, https://youtu.be/7wOU7csj1ME