pyMLIR

Release 0.5

Mar 23, 2023

Contents

pyMLIR is a Python Interface for the Multi-Level Intermediate Representation (MLIR).

CHAPTER 1

Creating a Custom Dialect

One of MLIR's most powerful features is being able to define custom dialects. While the opaque syntax is always supported by pyMLIR, parsing "pretty" definitions of custom dialects is done by adding them to the `dialects` field of the MLIR parser, as in the following snippet:

```
import mlir

# Load dialect from Python file
import mydialect

# Add dialect to the parser
m = mlir.parse_path('/path/to/file.mlir', dialects=[mydialect.dialect])
```

A dialect is represented by a `Dialect` class, which is composed of custom types and operations. In this document, we use `toy` as the dialect's name.

The structure of a dialect file is usually as follows:

```
# Imports
# Dialect type AST node classes
# Dialect operation AST node classes
# Dialect class definition
```

1.1 Simple Dialect Syntax API

To make dialect definition as simple as possible, pyMLIR provides a Syntax API based on Python's `str.format` grammar. Defining a dialect type or operation using the Syntax API is then performed as follows:

```
from mlir.dialect import DialectOp, DialectType
from dataclasses import dataclass
import mlir.astnodes as mast

@dataclass
```

(continues on next page)

(continued from previous page)

```
class RaggedTensorType(DialectType):
    """ AST node class for the example "toy" dialect representing a ragged tensor. """
   implementation: mast.StringLiteral
   dims: List[mast.Dimension]
   type: Union[mast.TensorType, mast.MemRefType]
    _syntax_ = ('toy.ragged < {implementation.string_literal} , {dims.dimension_list_
↪ranked} '
                '{type.tensor_memref_element_type} >')
```

The syntax format parses any `{name.type}` token as an AST node field `name` with type `type`. The types that can be used either come from `mlir.lark`, or from the `preamble` argument to the `Dialect` class (see below). Note the spaces between tokens - they represent the fact that whitespace can be inserted between them.

pyMLIR will then detect the three fields (`implementation`, `dims`, and `type`) and inject them into the AST node type. You can specify more than one match for your type or operation, and if fields are not defined they will be set as `None`. Example:

```
@dataclass
class DensifyOp(DialectOp):
    """ AST node for an operation with an optional value. """
    arg: SsaId
    type: TensorType
    pad: Optional[Union[StringLiteral, float, int, bool]] = None
    _syntax_ = ['toy.densify {arg.ssa_id} : {type.tensor_type}',
                'toy.densify {arg.ssa_id} , {pad.constant_literal} : {type.tensor_
↪type}']
```

When dumping the code back to MLIR, pyMLIR remembers which match created the AST node and will create the appropriate code.

Constructing the dialect itself follows creating the object with a unique dialect name, and all the operations and types.

```
from mlir.dialect import Dialect
from mlir import parse_path

# Define dialect
my_dialect = Dialect('toy', ops=[DensifyOp], types=[RaggedTensorType])

# Use dialect to parse file
module = parse_path('/path/to/toy_file.mlir', dialects=[my_dialect])
```

1.2 Advanced Dialect Behavior

In order to extend custom behavior in the dialect (e.g., to change how a node is read or written), you can extend the `DialectOp` or `DialectType` classes. In addition, there are two mechanisms that can be used in the `Dialect` class in order to parse concepts beyond nodes for types and operations: `preamble` and `transformers`.

Writing a new AST node has four implementation requirements:

1. Populating the `_fields_` static class member
2. Implementing an `__init__` function to parse Lark syntax trees
3. Implementing a `dump` function to output a string with the MLIR syntax
4. Either implementing a Lark rule in the `Dialect` preamble with and mapping the rule name to the class using the `_rule_` static class member, or defining the Lark rules directly in the `_lark_` static class member

For example, if we wanted to be strict with how we dump the `RaggedTensorType`, and use our custom rule for parsing, we would implement the class in the following way:

```
from mlir.dialect import DialectType
from mlir.astnodes import Node, dump_or_value
from lark import Tree
from typing import Union, List

class RaggedTensorType(DialectType):
    _fields_ = ['implementation', 'dims', 'type']
    # Notice that the first argument is optional
    _lark_ = ['"toy.ragged" "<" (string_literal ",")? dimension_list_ranked '
              'tensor_memref_element_type ">"']

    def __init__(self, match: int, dims, type, implementation = None):
        # Note that since _lark_ has only one element, "match" should always be 0
        self.match = match
        self.type = type
        self.dims = dims
        self.implementation = implementation

    def dump(self, indent: int = 0) -> str:
        # Note the exclamation mark denoting a dialect type
        result = '!toy.ragged<'
        if self.implementation:
            result += dump_or_value(self.implementation, indent)
        result += '%sx%s>' % ('x'.join(dump_or_value(d, indent) for d in self.dims),
                              dump_or_value(self.type, indent))
        return result
```

`dump_or_value` is a helper function in `mlir.astnodes` to either write out the value, a list/dict/tuple of values, or literals into MLIR format. For most cases, though, the `_syntax_` format will suffice (and creates shorter code than above).

As for extensions to the dialect itself, `preamble` and `transformers` are keyword arguments that can be given to the `Dialect` class. The former allows arbitrary Lark syntax to be parsed as part of the dialect, and the latter is a dictionary that maps rule names to node-constructing callable functions/classes. This gives a custom dialect full control over the syntax parsing and tree construction.

For example, we can create rules for a new kind of list structure in our toy dialect:

```
my_dialect = Dialect('toy', ops=[DensifyOp], types=[RaggedTensorType],
                     preamble='''
// Exclamation mark in Lark means that string tokens will be preserved upon parsing
!toy_impl_type : "coo" | "csr" | "csc" | "ell"
toy_impl_list   : toy_impl_type ("+" toy_impl_type)*
                     ''',
                     transformers=dict(
                        toy_impl_list=list  # Will construct a list from parsed values
                     ))
```

Now we can parse lists of specific implementation types for our ragged tensor, e.g., `toy.ragged<coo+csr, 32x14xf64>` rather than one string literal. Note that the type `_lark_` or `_syntax_` has to change accordingly.

CHAPTER 2

Builder API

class mlir.builder.builder.**IRBuilder**

MLIR AST builder. Provides convenience methods for adding core dialect operations to a `Block`.

block

The block that the builder is operating on.

position

An instance of `int`, indicating the position where the next operation is to be added in the *`block`*.

Note:

- The concepts here at not true to the implementation in llvm-project/mlir. It should be seen more of a convenience to emit MLIR modules.
- This class shared design elements from `llvmlite.ir.IRBuilder`, querying mechanism from `loopy`.

Registering a custom dialect builder

register_dialect(*name: str*, *dialect_builder: mlir.builder.builder.DialectBuilder*, *overwrite: bool = False*) → None

Position/block manipulation

position_at_entry(*block: mlir.astnodes.Block*)

Starts building at *block*'s entry.

position_at_exit(*block: mlir.astnodes.Block*)

Starts building at *block*'s exit.

goto_block(*block: mlir.astnodes.Block*)

Context to start building at *block*'s exit.

Example usage:

```
with builder.goto_block(block):
    # starts building at *block*'s exit.
    z = builder.addf(x, y, F64)

# goes back to building at the builder's earlier position
```

goto_entry_block(*block: mlir.astnodes.Block*)
Context to start building at *block*'s entry.

Example usage:

```
with builder.goto_block(block):
    # starts building at *block*'s entry.
    z = builder.addf(x, y, F64)

# goes back to building at the builder's earlier position
```

goto_before(*query: mlir.builder.match.MatchExpressionBase*, *block: Optional[mlir.astnodes.Block] = None*)
Enters a context to build at the point just before *query* gets matched in *block*.

Parameters **block** – Block to query the operations in. Defaults to the builder's block.

Example usage:

```
with builder.goto_before(Reads("%c0") & Isa(AddfOperation)):
    # starts building before operation of form "... = addf %c0, ..."
    z = builder.mulf(x, y, F64)
# goes back to building at the builder's earlier position
```

goto_after(*query: mlir.builder.match.MatchExpressionBase*, *block: Optional[mlir.astnodes.Block] = None*)
Enters a context to build at the point just after *query* gets matched in *block*.

Parameters **block** – Block to query the operations in. Defaults to the builder's block.

Example usage:

```
with builder.goto_after(Writes("%c0") & Isa(ConstantOperation)):
    # starts building after operation of form "%c0 = constant ...: ..."
    z = builder.dim(x, c0, builder.INDEX)

# goes back to building at the builder's earlier position
```

Types

Attr F16 f16 type

Attr F32 f32 type

Attr F64 f64 type

Attr INT32 i32 type

Attr INT64 i64 type

Attr INDEX index type

MemRefType(*dtype: mlir.astnodes.Type, shape: Optional[Tuple[Optional[int], ...]], offset: Optional[int] = None, strides: Optional[Tuple[Optional[int], ...]] = None*) → mlir.astnodes.MemRefType
Returns an instance of `mlir.astnodes.UnrankedMemRefType` if shape is *None*, else returns a

mlir.astnodes.RankedMemRefType.

Standard dialect ops

dim(*memref_or_tensor: mlir.astnodes.SsaId, index: mlir.astnodes.SsaId, memref_type: Union[mlir.astnodes.MemRefType, mlir.astnodes.TensorType], name: Optional[str] = None*)

addf(*op_a: mlir.astnodes.SsaId, op_b: mlir.astnodes.SsaId, type: mlir.astnodes.Type, name: Optional[str] = None*)

mulf(*op_a: mlir.astnodes.SsaId, op_b: mlir.astnodes.SsaId, type: mlir.astnodes.Type, name: Optional[str] = None*)

index_constant(*value: int, name: Optional[str] = None*)

float_constant(*value: float, type: mlir.astnodes.FloatType, name: Optional[str] = None*)

class mlir.builder.builder.**AffineBuilder**(*core_builder: mlir.builder.builder.IRBuilder*)
Affine dialect ops builder.

for_(*lower_bound: Union[int, mlir.astnodes.SsaId], upper_bound: Union[int, mlir.astnodes.SsaId], step: Optional[int] = None, indexname: Optional[str] = None*)

load(*memref: mlir.astnodes.SsaId, indices: Union[mlir.astnodes.AffineExpr, List[mlir.astnodes.AffineExpr]], memref_type: mlir.astnodes.MemRefType, name: Optional[str] = None*)

store(*address: mlir.astnodes.SsaId, memref: mlir.astnodes.SsaId, indices: Union[mlir.astnodes.AffineExpr, List[mlir.astnodes.AffineExpr]], memref_type: mlir.astnodes.MemRefType*)

2.1 Querying expressions

class mlir.builder.match.**All**
Matches with all nodes.

class mlir.builder.match.**And**(*children: List[mlir.builder.match.MatchExpressionBase]*)
Matches if all its children match.

class mlir.builder.match.**Or**(*children: List[mlir.builder.match.MatchExpressionBase]*)
Matches if any of its children match.

class mlir.builder.match.**Not**(*child: mlir.builder.match.MatchExpressionBase*)
Matches if the child does not match.

class mlir.builder.match.**Reads**(*name: Union[str, mlir.astnodes.SsaId]*)
Matches the variables read by the operation.

class mlir.builder.match.**Writes**(*name: Union[str, mlir.astnodes.SsaId]*)
Matches the variable names written by the operation.

class mlir.builder.match.**Isa**(*type: type*)
Matches the operation's type.

CHAPTER 3

Reference

- genindex
- modindex

Python Module Index

m

Index