Infrastructure for Shape Inference and Lowering

Alexandre Eichenberger

IBM Watson: Tong Chen, Alexandre Eichenberger, Kevin O'Brien, Gong Su.

Tokyo Research: Haruki Imai, Kiyo Kawachiya, Tung Le Duc, Yasushi Negishi.

Goals

- **Shape Inference is used to determine the shape of tensors/memrefs:**
  - each ONNX operation defines its output as function of its input tensors,
  - shapes are known at compile time, runtime, or combination of both.
- **Shapes are needed at 2 key moments.**
  1. ONNX Shape Inference (ONNX transforms), where we don't write low level code.
     - Runtime shapes are defined at "-1"; symbolic analysis classifies relations between "-1".
  2. ONNX Lowering (e.g. ONNX to Krnl, Linalg, Tosa*, MHLO*,…).
     - Runtime shapes are explicitly computed by code (e.g. Arith, Math, Shape).
- **Observation:**
  - "Arithmetic on constant shape is the same as code generating shapes at runtime."
  - We want to write this code once, not once per ONNX->ONNX/Krnl/Linalg/…

We introduce 3 key classes

- **Index expressions (IndexExpr and subclasses)**
  - Polymorphic class that represent computations over shapes (e.g. add/ceil/select…).
  - ONNX Shape Inference: generate literal values or question marks (aka “-1”).
  - Shape Lowering: generate literals or create operations that compute shapes at runtime.

- **Index expression builder (IndexExprBuilder and subclasses)**
  - Data structure that extract shape info from graph: attribute, arrays (constant or not), shapes.
  - Each dialect constructs its own subclass (e.g. for ONNX/Analysis, KRNL, MHLO).

- **ShapeHelper (ONNXOpShapeHelper and subclasses)**
  - Encapsulate how to compute the shape for a give ONNX operation.
  - Each ONNX operation defines its own/reuse a subclass.

I. IndexExpr

- **IndexExpr (IE): subclasses represent shape values.**
  - IndexExpr are one of 7 subclasses.
  - Result of constant/affine/nonaffine computations (Literal/Affine/NonAffine IndexExpr).
  - Result of compares (Predicate IndexExpr).
  - Runtime inputs are classified as dims or symbols (Dim/Symbol IE, see affine dialect for details).
  - Unknown-at-compile-time/undefined (Questionmark/Undefined IndexExpr).
- **Queries/Getters**
  - hasAffineExpr(), hasValue(); isDefined(), isLiteral(), isAffine(), isDim(), isSymbol()…
  - getLiteral(), getAffineExpr(), getValue().
- **Operations:**
  - +/-, */floor/ceil, min/max, select, comparators (==, <=, >=,…)

I. IndexExpr (cont.)

- **Index expressions are polymorphic**
  - associated with a builder -> can generate code
  - not associated with a builder -> runtime are represented with question marks (”-1”).

- **All index expressions are part of a scope (IndexExprScope)**
  - repository of all IndexExpr of any kinds
  - scope can be nested
    - e.g. an index variable in one loop (Dim) can become an invariant (Symbol) in a nested loop.
  - IndexExpr can only be used in the scope they are defined (exception: literals).
  - IndexExpr from an outer scope can be imported into current scope.
  - In a given scope, an input cannot be both a Dim and a Symbol.

II. IndexExprBuilder

One per dialect

- **Class describes how to extract values needed for shape inferences.**
- **Extracting values from attributes:**
  - getIntFromArrayAsLiteral (attr, i) returns a LiteralIE from the attribute at index i.
  - getIntFromArrayAsLiterals (attr, list) fills the list with LiteralIE from the attribute.
- **Extracting values from constants/computations:**
  - getIntFromArrayAsSymbol (value, i) retrieves the defining operation of value:
    - If it is a constant operation, returns that constant as LiteralIE (literals are parts of affine symbol).
    - If is not a constant, return that value as a QuestionmarkIE or SymbolIE depending on the phase.
- **Extracting shapes from variables:**
  - getShapeAsDim(value, i) retrieves the shape defined by the type of value.
    - returns LiteralIE, QuestionmarkIE, or DimIE depending on constant and/or phase.
- Many more calls are available.

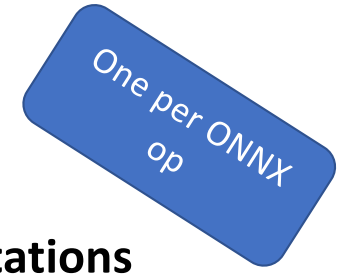
II. IndexExprBuilder (cont)

- **There are 3 subclasses currently defined:**
  - IndexExprBuilderForAnalysis / IndexExprBuilderForKrnl / IndexExprBuilderForMhlo
- **Each subclass must define 3 pure virtual functions:**
  - getConst: returns a constant's DenseElementAttribute.
  - getValue: returns a value from an array at a given location.
  - getShape: returns a shape from a value's type.

- **These virtual functions are dialect / phase dependent:**
  - e.g. runtime variables are returned as QuestionmarkIE during Shape Inference Analysis.
  - e.g. constants are found in ONNXConstantOp in ONNX, KrnlGlobalOp in Krnl.

- **All the IndexExprBuilder functionality is built on these 3 functions.**

III. ONNXOpShapeHelper



- **Each operation defines/reuse a subclass to encapsulate its shape computations**
  - They all share a unique set of parameter for their constructor.
  - They all define a virtual computeShape function to perform the computations.
- **The shapes are retrieved using getOutputDims(i), returning a list of index expressions.**
- **Some operations define additional useful values.**
  - e.g. Normalized pad, kernel sizes for ONNXConvOp.
  - e.g. Reduction dimensions for ONNXReduceSumOp.

III. ONNXOpShapeHelper (cont.)

- **Subclasses need a constructor with (op, operands, index builder, scope) parameters:**
  - op: Operation being analyzed.
  - operand: list of explicit operands (e.g. during lowering). If none given, use the ones currently associated with op.
  - index builder: index builder that is used to create the index expressions being investigated.
  - scope: index expression to use; create one if none provided.
- **Subclasses need a computeShape() method,**
  - Compute the shape for the given op/operands. Depending on the index builder passed, code may be generated to materialize the shapes.

- **ONNXOpShapeHelper provides a lot of functionality to help streamline shape inference**