# Dyanmic Shapes

NOTE: Effort is being made to make this facility generic so that it can be eventually upstreamed to MLIR in some fashion. However, because MLIR lacks a set of frontend ops and generally does not currently have any frontend oriented infrastructure, it is being prototyped within IREE in order to find a working set of ops and algorithms.

## Levels of dynamicism

In general, there are three levels of shape information that can be present in the input IR (or trivially derived by applying some form of shape inferencing algorithm). Each additional one imposes more work on the compiler and runtime, so generally, the implementation progresses by addressing each once the former is well established:

1. Fully static shapes: No tensors have dynamic dimensions. All tensors are ranked.
2. Ranked Dynamicism: All tensors have ranks, but some dimensions may be unspecified.
3. Unranked Dynamicism: Some tensors have indeterminate ranks.

At this stage, *Dynamic Shapes* in IREE refers to supporting dynamic ranked dynamic tensors, where some dimensions are left unspecified at public function boundaries. It is expected that once this is solid, some support can be considered for unranked dynamicism, and it is further expected that will entail new ops, algorithms and runtime support, apart from what is needed for ranked dynamicism.

Within the category of Ranked Dynamicism, it is well known that some dynamic dimensions are easier to deal with than others: in common DNN use, outer dimensions are much easier and more common with respect to code generation and kernel fanout than dynamic inner dimensions.

While the shape handling machinery is relatively generic, we expect that real backends will be limited with respect to how much they support all combinations of dynamic dimensions. Eventually, IREE intends to solve this by having relatively robust CPU fallback for fully dynamic cases and actionable warnings that pinpoint when more specificity could increase performance.

## Compiler Frontend

In general, the IREE compiler frontend should accept modules containing functions with operands/results that have dynamic dimensions. Such functions may also have runtime dependent shapes in the form of `GetShape`-style ops which get a shape from an arbitrary tensor, perform some arithmetic on it and use the results elsewhere.

### Shape dialect and lowering

IREE is introducing a `shape` dialect with a handful of ops and transformations that are useful for materializing dynamic shape computations in terms of high level ops on tensors.

**Types:**

- `ranked_shape`: This value type represents the dynamic dimensions of a partially known, ranked shape. It is used early in the compiler to represent anywhere that dynamic dimensions need to be

passed (i.e. function args/results, etc). At lower levels of the compiler, it will generally be disaggregated into loose SSA values. This type also carries the datatype used to represent the dimensions. This is currently fixed to i32 but may be leveraged eventually to use smaller integer when such things are known to be legal.

**Ops:**

- `get_ranked_shape`: Takes a tensor SSA value and returns a corresponding `ranked_shape`. Early in the compilation flow, anything that needs a ranked shape should add such ops so that the compiler can later determine which shape computations to materialize. Getting the `ranked_shape` of a static tensor should yield a constant.
- `tie_shape`: Takes tensor and ranked_shape SSA values and returns the tensor. This is used as a junction point by the shape materialization passes to know at various points precisely what the shape is.
- ... TODO: need `get_shape_dim` and conversions to/from 1D tensors and loose SSA values.

## Materialization

### Function signature expansion

Early in the process, all functions should have their arguments and results expanded so that any dynamic tensors in their signature will gain a new argument/result for the corresponding `ranked_shape`. This is done by expanding the signatures and for arguments, inserting placeholder `tie_shape` ops which preserve the association for later materialization. For results, `get_ranked_shape` ops are inserted.

This is carried out by the `iree-shape-expand-function-dynamic-dims` pass, which uses the conversion framework under the hood to perform type expansion.

This pass is typically done early in the compiler flow.

### Shape dependent codegen

A lot of scheduling logic will need to access shapes (i.e. allocation, workgroup size calculation, etc). In general, this should all be done based on a `get_ranked_shape` op and corresponding `get_shape_dim` ops. For fully static cases, these should reduce down to constants. For dynamic dimensions, the `get_ranked_shape` ops serve as anchors where later parts of the compiler know they need to materialize shape values.

### Materializing shape computations

TODO: We have a sketch of this but are still proving it out.

Generally, it should be possible, for any `get_ranked_shape` op, to trace up the use-def chain and materialize shape manipulation arithmetic. Once materialized, a `tie_shape` op should be inserted to memorialize the junction. Eventually, every `get_ranked_shape` op should be follow a `tie_shape` op, and the canonicalization rules will elide the `get_ranked_shape`. There is complexity around blocks, control flow, etc, but this basic algorithm should be workable.

Work is ongoing upstream to provide a facility to register shape functions with ops, which would provide a dynamic, dialect independent way to know what arithmetic to materialize. However, in most cases this is not necessary. The built-in traits around types and sizes will allow most propagation to happen without shape functions. We intend to start with a static set of cases for the rest in order to prove the concept.

**Scalarization**

TODO: We have a sketch of this but are still proving it out.

It is quite common in real-world DNN usage to get the 1D tensor representing a shape and perform arbitrary tensor ops on it (usually basic arithmetic, slicing, concating, tiling, etc). While this is perfectly acceptable from a correctness standpoint, it is usually not performant: shapes are typically very small one dimensional vectors, and computations on them are usually trivial to reduce to small sequences of scalar machine code of a form that CPUs are very good at executing. Further, we often want to run these calculations eagerly when dispatching functions, etc (i.e. to pre-allocate buffers) and having them isolated (versus treating them as arbitrary dense computations) can be quite valuable.

We expect that the type bracketing that happens with `ranked_shape` and the corresponding ops will make it easy to write some simple DRR patterns to identify such shape manipulation sequences and lower them directly to regions of `vm` ops operating on scalars. Such regions can be retained and directly emitted when lowering to the `vm` dialect and/or CPU code generation and would run with low overhead along with any other scheduling code.

While an optimization, we suspect this is an important one.

### Shape inference

TODO: This is mostly placeholder

There is work happening upstream to implement MLIR-integrated shape inference. In the mean-time, IREE expects that the input to the compiler has already had some shape inference performed on it. In practice, for TensorFlow, there is a pass which applies TensorFlow's pre-MLIR shape inference mechanisms to derive such things. This has limitations but is a reasonable starting point.

## Compiler Backends

TODO: This is mostly placeholder.

Much of the newer structured-ops based codegen is capable of working (within bounds) with ranked dynamic shapes without much work. Given the lack of an e2e story, much of this has been done "by way of code review" and there are certainly issues to be resolved.

In addition, there are several ABI issues and negotiations with the backend that still need to be fleshed out.