Pre-dispatch functionalization

Pre-dispatch (doc) tracing exists today, but does **not** give you back an IR that promises to be functionalized. What work is required if we want to functionalize pre-dispatch IR, hook it up to export(), and ensure that all edge cases are handled?

High level brain dump:

- Get functionalization running above pre_dispatch
- Integrate with `aot_autograd.aot_export_module()`
- Handle CompositeImplicitAutograd decomps
- Functionalize autograd/autocast API's

Concrete steps for MVP:

- (1) and (2) are definitely needed
    - I can do most of (1). This is blocked on https://github.com/pytorch/pytorch/pull/106405 (and everything below it in the stack) landing
- (3) is not strictly needed, to get something working, but we'll need it for export to abide by its guarantee that "pre_dispatch IR does not decompose COmpositeImplicit ops"
- (4) is not needed for an MVP

Step 1: Get functionalization to run as part of PreDispatch tracing

Today we have a concept of "per-dispatch-key-modes" (doc), that allow us to push any `TorchDispatchMode`'s onto a given dispatch key. You can "trace at a higher level in the dispatcher" by taking our existing ProxyTorchDispatchMode, and pushing it onto a mode stack attached to any dispatch key.

That design ended up being more than we needed: we now have a dedicated PreDispatch dispatch key. Instead of pushing modes onto arbitrary dispatch keys, we just push our ProxyTorchDispatchMode onto the PreDispatch key's mode stack (code snippets here and here).

We'll need to use the new `FunctionalTensorMode` (a torch_dispatch mode that hasn't landed yet).

Open question about cleanup: should we re-design the PreDispatch key to be more like the Python key?

Step 2: Integrate with aot_export_module

Aot_export_module is the entry-point today for re-using AOTAutograd for export. It does everything from lifting params/buffers, functionalizing, annotating information about input mutations, and (maybe) generating a backwards graph.

We can re-use the existing functionalization/param_lifting code in AOTAutograd by adding a pre_dispatch flag, and carefully instructing AOTAutograd to attach its infra tracing modes to the PreDispatch key instead of the “normal” flow for torch_dispatch modes (the python key)

Step 3: Handling CompositeImplicitAutograd decomps

If export were to decompose all CompositeImplicitAutograd ops, then no work would be required. But if we want CompositeImplicitAutograd ops to stick around in the functionalized graph, some extra work is required.

Problem 1: Functionalization codegen will decompose all CompositeImplicitAutograd ops by default (code).

Problem 2: Today, CompositeIMplicitAutograd ops make no guarantees around schema correctness. This has historically been ok, because all of our subsystems that care about operator schema (functionalization and autograd) have always ran the decompositions.

If we want functionalization not to decompose CompositeIMplicitAutograd ops, we’ll need to audit aten’s list of composites for “ops that lie about their schema”, and make sure to always decompose that subset. We might want to do this via operator tagging. Some known examples
- aten::to
- aten::reshape
- aten::native_batch_norm
- aten::rrelu_with_noise

Also, from Richard Zou : there are a lot of custom ops out there that are CompositeImplicitAutograd, and also make zero schema guarantees. Maybe we should always decompose any non-aten ops that are CompositeImplicitAutograd?

Step 4: Handling autograd/autocast ops

Pre_dispatch IR also includes autograd/autocast ops, like `torch._C._set_grad_enabled(True)`, which are not functional.

If we want every op in a pre_dispatch graph, we'll need to make these ops functional. One way is by turning these into higher order ops. E.g. this code:
```
X = torch.ones(2)
With torch.no_grad():
   x.mul_(2)
   Y = x * x
```

Would translate into a higher order op `no_grad_higher_order(args, inner_graph)`, with `inner_graph` representing the no_grad block.

We'd need the same treatment for `torch._C.set_autocast_enabled`.

Another complication is input mutations: Today, (I think) input mutations are banned on higher order ops. However, it's probably not too uncommon for user code to involve mutating a tensor under no_grad() mode, where that tensor was created outside of the no_grad context.

In order to support this, we'll need:

(1) a way for higher order ops to handle input mutations
(2) When we functionalize a higher order op, we'll want to detect if our input mutation was a "true graph input", or just an intermediate in the outer graph. When functionalizing, if our input was not a true graph input, then we'll need to make sure to update any later uses of the input in the graph with the updated version.