Boxing/Unboxing in the operator library

When calling operators, there are generally two call paradigms:

Boxed Paradigm

In the boxed world, all operator arguments are wrapped (or "boxed") into an IValue (which is a tagged enum similar to std::variant). Ops are called with a std::vector<IValue> containing their arguments. JIT for example keeps arguments on a call stack (i.e. std::vector<IValue>) similar to how the x86 function call stack works. When an op is called, it gets a pointer to that call stack, pops its arguments from the stack, does some computations, pushes its returns back on the stack, and returns to the interpreter to call the next op in line. The lite interpreter for mobile does the very same thing, just in a more minimal version.

```
// Operator implementations (simplified) look like
void my_operator(vector<IValue>* stack) {...}

// and they (simplified) can be called like
// (assume schema: my_op(int arg1, Tensor arg2, float arg3) -> Tensor)
std::vector<IValue> call_stack {3, my_tensor, 4.2};
Dispatcher::callBoxed(my_op, &stack);
```

Unboxed Paradigm

When calling an op directly (say from Eager Python or the Eager C++ Frontend), you usually don't want to have to care about IValues, boxing and call stacks. If an operator takes three arguments, then they should be passed in as three arguments with their concrete types. Also, when implementing a kernel, it would be neat to not have to know about the call stack but just define the operator to actually take three arguments instead of a stack.

```
// Operator implementations look like
Tensor my_operator(int64_t arg1, Tensor arg2, double arg3) { ... }

// and they (simplified) can be called like
Dispatcher::callUnboxed(my_op, 3, my_tensor, 4.2);
// note: the python eager frontend and the c++ frontend have even nicer
// APIs, which ultimately map to this one.
```

Note that caffe2 was fully boxed, in caffe2 no unboxed paradigm existed.

Bridging Paradigms

We do need both the boxed and unboxed paradigms. Examples for boxed calling are JIT and Mobile, and also the still existing caffe2 frontend. Unboxed calling happens from Python and C++ eager mode. Examples for boxed operator

implementations are caffe2 kernels that are exported to PyTorch, but also backend fallback kernels like Lazy, AMP or Profiling that "hook" into the dispatcher to run some code instead of the actual kernel, but then re-dispatch to the actual kernel. Examples for unboxed operator implementations are almost all other operators, whether they come from native_functions.yaml or from the custom operator API.

Having both paradigms also means we need to build a bridge. This is where "boxing" or "unboxing" comes in.

- **Unboxing**: If you call an unboxed kernel with a boxed calling API (say JIT calls an op from native_functions.yaml), then we need to run unboxing logic so that the boxed arguments from JIT are unpacked and the operator implementation is called with individual and fully typed arguments. This is done using some C++ metaprogramming.
- **Boxing**: If you call a boxed operator implementation with an unboxed calling API (say Eager Mode calls an op with Profiling, AMP or Lazy enabled, or Eager Mode calls a caffe2 op), then we need to run boxing logic so that the arguments are packed up to the call stack expected by those operator implementations.

The following diagram shows a high level overview of how these pieces interconnect.

Boxing/Unboxing Architecture Overview

1. The boxed op calling API takes an operator name and a stack of IValues and calls that operator. It is used from the mobile lite interpreter, when a caffe2 model calls a PyTorch op, and when backends like lazy or AMP re-dispatch an operator. Starting with https://github.com/pytorch/pytorch/pull/36838, it is also used by JIT to call ops.
2. Given a stack of IValues, boxed dispatch iterates over the IValues that are arguments for the current operator call, finds the Tensor arguments, constructs the dispatch key from those arguments, and finds the operator implementation function pointer that should be called for this dispatch key.
3. A boxed op implementation is an operator implementation that is implemented in terms of a stack of IValues. Caffe2 operators that are added to the PyTorch operator library are for example written this way. Also, backend fallback kernels like lazy or AMP, i.e. kernels that the backend wants to be called for all operators need to be implemented in a way that is independent from the actual operator signature and are implemented as boxed kernels.
4. The unboxed calling API takes an operator name and a set of arguments in unboxed form, i.e. the arguments have types like Tensor, int64_t and std::string. This is the API called from C++ Eager Mode. Our Python bindings also call into this API.
5. Based on the actual arguments, unboxed dispatch does some compile-

time metaprogramming to know which arguments are tensor arguments, directly accesses only those to construct the dispatch key (because of the metaprogramming it doesn't need to look at or iterate over other arguments), and finds the kernel function pointer that should be called for this dispatch key.

6. An unboxed operator implementation is an operator implementation that is implemented in terms of unboxed values like Tensor, int64_t and std::string. This is usually a plain C++ function that is registered as an implementation for an operator. Custom Operators are written this way. Also, we have some codegen that registers the kernels for operators in native_functions.yaml as unboxed operator implementations.
7. When the unboxed calling API wants to call an operator implementation that is written as a boxed kernel, it needs to box the arguments into a stack of IValues. We have a piece of metaprogramming for that. This kind of metaprogramming is only possible in the unboxed world where we have access to the parameter types. This means this code is above the "op call time / op registration time" line and is part of "op call time". At op call time, when calling an unboxed kernel, we have the necessary information to generate this code. At registration time, when registering a boxed kernel, we do not.
8. When the boxed calling API wants to call a kernel that is written as an unboxed operator implementation, it needs to unbox the arguments into actual values like Tensor, int64_t, or std::string. This is also solved by a piece of metaprogramming. As in point (7), this metaprogramming needs to be generated in the unboxed world because it needs to know the actual parameter types. That means it has to live below the "op call time / op registration time" line and is part of "op registration time". At op registration time, when registering an unboxed kernel, we have the necessary type information to generate this. At op call time, when calling a boxed kernel, we do not. So what actually happens is that whenever an unboxed kernel is registered, we auto-generate a boxed wrapper kernel (3) for it and also register that boxed kernel to be called when the boxed calling API wants to call this kernel.