# JIT Technical Overview

The JIT can run and optimize PyTorch programs separate from the Python interpreter. This overview is organized into sections that go over different independent components:

1. Core Program Representation - The JIT executes TorchScript, a subset of Python. This section describes how TorchScript programs are represented in the JIT, and serves as the interchange format between components of the JIT.
2. Generating Programs - TorchScript programs can be created either through tracing Python code or through directly writing TorchScript. This section describes how Models are created from these frontends.
3. Executing Programs - Once created, TorchScript models are optimized and run. Since this is a just-in-time compiler, programs are optimized as they are executed, so this section describes both how programs are optimized and how they get run.
4. Saving Programs - TorchScript is often created in Python and then used from C++. This section describes how the save and load process works.
5. Python Bindings - TorchScript code is normally created and used from Python, so this section describes how the Python components interact with the code in this directory.

For concepts that are actual classes in the JIT, we use capitalized words in code font, e.g. `Graph` or `Value`.

Sections start with a reference to the source file where the code related to the section resides.

## Table of Contents

# Core Program Representation

## Modules

api/module.h

At the top level, all TorchScript programs are represented as a Module. Modules contain:

- named Parameters - `Tensors` used in training such as `weight` or `bias`
- named Buffers - `Tensors` that are part of the training state of a module but do not appear in module.parameters() and do not participate in gradient descent.
- named sub-Modules - used for code organization.
- named Attributes - all other attributes that are not in the above three categories. Typically used for configuration and are not saved/restored in the modules `state_dict` .
- named Methods - functions that can be run on the module such as `forward`

This mirrors the `nn.Module` objects used in Python. All TorchScript code is a member of some module. This includes pure functions such as those created by annotating a Python function with `@torch.jit.script` , which

are represented internally as a Module that has a single method `forward` that contains the implementation of the function.

## Parameters

api/module.h

Modules contain Parameter objects, which simply hold a "slot" where a `Tensor` can be placed. These `Tensors` are accessible by the Methods of the Module or the parent Module.

## Method

api/module.h

A Method is a piece of TorchScript code that takes a number of arguments and produces an output value. Methods have several subcomponents. A FunctionSchema describes the types and names of the input arguments and return value. A list of `member_inputs` describes which Parameters are accessed by the method (this is blank for pure functions). A `Graph` object describes the actual code inside the method. The Method also maintains a GraphExecutor which is used to actually execute the `Graph` that defines the method.

The `Graph` inside the Method is a pure function. The Parameters used by the Method are added as additional inputs to this graph before it is run. This allows the GraphExecutor to treat method inputs and method parameters the same for the purposes of optimization and execution, simplifying the process for executing programs.

Methods also contain helper functions for inserting calls to the Method from other Method objects.

## FunctionSchema

aten/src/ATen/core/function_schema.h

Each Method has a FunctionSchema that describes the Types of the arguments and return values of a function. Operators (builtin primitives that are called by the Interpreter) also have FunctionSchema. FunctionSchema are analogous to a function *declaration* in C++. They describe how to call the function but do not provide an implementation.

## Graph

ir.h

Graphs are the root of the intermediate representation (IR) used to define the implementation of TorchScript functions. If you are familiar with LLVM, they are analogous to an `llvm::Function` object. A `Graph` is composed of `Nodes`, `Blocks`, and `Values`. `Nodes` are instructions (e.g. do a matrix multiply). `Nodes` are organized into `Blocks` of sequentially executed `Nodes`. Each `Node` produces a list of output `Values`, and also consumes a list of input `Values`. As an example, a user may write the following TorchScript code:

```
@torch.jit.script
def f(a, b):
  c = a + b
  d = c * c
  e = torch.tanh(d * c)
  return d + (e + e)
```

The frontend, described later in this document will turn into a `Graph`:

```
graph(%0 : Double(2),
      %1 : Double(2)):
  %2 : int = prim::Constant[value=1]()
  %3 : Double(2) = aten::add(%0, %1, %2)
  %4 : Double(2) = aten::mul(%3, %3)
  %5 : Double(2) = aten::mul(%4, %3)
  %6 : Double(2) = aten::tanh(%5)
  %7 : Double(2) = aten::add(%6, %6, %2)
  %8 : Double(2) = aten::add(%5, %7, %2)
  return (%8)
```

This is the canonical textual representation of the IR. You should be able to easily find (almost all) of the elements we discussed above.

- `graph` is the `Graph`
- `%x` are `Value`s
- `%x : Double(2)` is a type annotation of `Value` `%x` (see below for a list of supported types).
- `%x : T1, %y : T2 = namespace::name(%z, %w)` is a `Node` which represents the `namespace::name` operator (this name is usually referred to as the `Node`s *kind*). It takes `%z` and `%w` `Value`s as inputs, and returns two outputs (`%x`, `%y`) of types `T1` and `T2` respectively.

Finally, nodes can have extra pieces of information assigned to them, which are called *attributes*. You can see that it's used in the `prim::Constant` node, which returns the `value` attribute when it's called. There's a fixed list of types you can attach:

- `int64_t`
- `double`
- `Tensor`
- `Graph` (useful for e.g. slicing subgraphs that are meant to be fused)
- `std::string`
- and lists of them (not nested)

Graphs in the JIT are in single-static assignment (SSA) form, meaning that each `Value` has precisely one defining `Node` that can be looked up directly from the `Value` (`Node* n = v.node()`).

**Ownership Model** `Blocks`, `Nodes`, and `Values` are *owned* by the `Graph` they appear in and may only appear in a single `Graph`. This is enforced by assertions. Creation and deletion of `Nodes` is done via `Graph` objects (e.g. `Graph::create`). Creation and Deletion of `Blocks` and `Values` is done via `Node` objects (e.g. `Node::addOutput`, `Node::addBlock`). The code also enforces certain consistency properties. For instance, `Node::destroy` removes a `Node`, but it is only valid to call this function if the `Values` produced by this `Node` are no longer used, which can be accomplished using other functions such as `Value::replaceAllUsesWith`.

Because `Graph` owns all its `Nodes`, `Values`, and `Blocks`, these values are always passed around by raw pointer. Generally developers should not write code that holds `Value`, `Node`, or `Block` objects indefinitely without also holding a `shared_ptr` to their owning `Graph`.

## Node

ir.h

A `Node` represents a single built-in operator such as a matrix multiply or a convolution. `NodeKind Node::kind()` identifies the operator the `Node` represents. Different operators (e.g. conv vs matrix-multiply) are represented by different kinds rather than via subclassing of `Node`, as one would find in LLVM. A `NodeKind` is a `Symbol` object, which is just an interned string inside some namespace. Symbols can be created from strings, e.g. through `Symbol::fromQualString("aten::add")`, so there is not a closed set of `NodeKind` values that a `Node` might have. This design was chosen to allow the open registration of new operators and user-defined operators.

> *Code in the JIT should always assume the universe of valid `Node` kinds is open and subject to be expanded.*

This reflects the reality of the PyTorch operator library where there are already several hundred valid operators.

Nodes produce output `Values` and take input `Values` as arguments. For instance, a matrix-multiply will take two input `Tensors` and produce one output `Tensor`. `Nodes` can produce multiple outputs. For instance `prim::TupleUnpack` splits a tuple into its components, so it has a number of outputs equal to the number of members of the tuple. Though `Nodes` may have multiple outputs, the number of outputs is *statically known* for each `Node`. Operations which may produce a dynamic amount of results, e.g. splitting a `Tensor` into chunks of size 2, will be represented as an operator that results a list object.

Because `Nodes` are not subclassed per-operator, it is very easy to construct invalid `Nodes`, e.g. by forgetting an input or an output, or by passing `Values` of the wrong Type. To help avoid this, `Graph` provides the method `Graph::insert` for constructing `Nodes` that guarantees `Nodes` have the correct setup. This method uses the database of registered Operators and their FunctionSchema to construct `Nodes` using that schema.

PyTorch IR supports function overloading, which means that a single NodeKind may correspond to multiple operators. For example, the kind `aten::add` has the following overloads (`Scalar` means `float` or `int` in this case):

- `aten::add(Tensor self, Tensor other) -> Tensor`
- `aten::add(Tensor self, Scalar other) -> Tensor`
- `aten::add(int self, int other) -> int`
- `aten::add(float self, float other) -> float`

For `Nodes` representing built-in Operators, the method `Node::schema` can also look up the FunctionSchema registered for that Operator.

Each overload corresponds to a different `FunctionSchema` object. A `Node` can be queried for its schema using the `schema()` method (it will check the argument types, and will try to match one of the options for its `kind()`).

Note that the chosen overload is not shown in any way in the textual output. If you're unsure which function a node resolves to, you might need to check the type annotations of its input values.

Each node also has a set of attributes which are named integers, strings, floats, `Tensors`, subgraphs, or lists of these types. These are used by special primitive operators to encode additional data in the `Node`. For instance `prim::Constant` defines a compile-time constant value. For `Tensor` constants, it will have a single `Tensor` attribute with the name `attr::value` which contains the value of the constant.

Attributes are *rarely used*. Operators like convolution or matrix-multiply have no attributes and take their arguments through the input list. This includes things that might be typically thought of as constants, like the stride of the convolution. In PyTorch, any of this information is potentially a dynamic property of the program so `Nodes` are always encoded in a way that allows these values to be dynamically determined. However, we recognize that many

inputs are almost always constants, so we make it easy to quickly check if an input is constant and get its value with `c10::optional<IValue> Node::get(Symbol name)`, which returns an `IValue` (a concrete value for the input) in the case the node is constant and `nullopt` otherwise.

## Block

ir.h

Nodes are organized into sequentially executed lists inside a `Block`. A `Node` is a member of precisely one `Block`. The `Graph` itself has a top-level `Graph::block()`, and control-flow nodes (`prim::If` and `prim::Loop`) also have sub-blocks. While it is possible to design a `Graph` representation that does not have a sequential order for nodes (i.e. a sea-of-nodes representation), we find it is much easier to debug and understand `Blocks` when there is a specific canonical order for all of the nodes. This does not preclude optimization passes from changing the order when it would improve performance, and the interpreter is allowed to execute `Nodes` out-of-order if the re-ordering preserves the semantics (much like an out-of-order processor does). Having the ordering ensures that graphs can always be easily printed, and that we can easily step through the execution of a graph.

Values are Block-scoped. A `Value` is in scope for the remainder of the `Block` it is defined in, including in the sub-blocks of any `Node` defined after it. `Values` go out of scope at the end of the block in which they are defined.

When `Nodes` are inserted into a `Graph`, they are inserted at a special "insertion point" that is part of the state of the `Graph`. On construction, this will go to the end of the `Graph`.

Each `Block` has two dummy nodes that are not included in the list of nodes in the `Block`. The `prim::Param` node represents the inputs to the `Block` and does not have a `prev()` or `next()` `Node`. The `prim::Return` `Node` represents the outputs of a `Block`. The list of `Nodes` in a `Block` is implemented as a circular linked list with the `prim::Return` `Node` serving as the beginning/end sentinel. Inserting and deleting at arbitrary places is efficient. Developers may also encounter implementations inside of IR objects that use this fact (e.g. appending to a `Block` is equivalent to putting the node before the `prim::Return` Node).

Iterators for the `Block::nodes()` list are invalided when the current `Node` they point to is moved or deleted. Otherwise iterators remain valid.

Blocks also contain a list of input and output values. The meaning of these values depends on where the `Block` is used. For the Graph's top-level `Block`, these are inputs and outputs to the `Graph`, and line up with the FunctionSchema associated with the Method that owns the `Graph`.

**Control-flow** is represented using sub-Blocks rather than a control-flow graph representation. A `prim::If` has one `Block` for the true branch and one `Block` for the else. A `prim:Loop` has a block for the loop body (there is no condition block, instead the end of the loop body computes whether to re-enter the loop body). This representation ensures we have structured control-flow. This limitation makes a lot of optimizations easier and is true for the vast majority of networks. A `Node` can look up what `Block` it is in, and a `Block` and can look up its parent (either the `Node` that has it as a sub-Block, or `nullptr` for the main Block).

### If

If-statement (`prim::If`) `Blocks` have no inputs, and the outputs are the new values of variables in the outer block whose values were altered in the if-statement. Example IR for an if-statement looks like:

```
%y_1, ..., %y_r = prim::If(%condition)
  block0():  # TRUE BRANCH, never takes arguments, has to return r outputs
    %t_1, ..., %t_k = some::node(%a_value_from_outer_block)
    -> (%t_1, ..., %t_r)
  block1():  # FALSE BRANCH, never takes arguments, has to return r outputs
    %f_1, ..., %f_m = some::node(%a_value_from_outer_block)
    -> (%f_1, ..., %f_r)
```

Values corresponding to `%y_1, ..., %y_r` will become either `%t_1, ..., %t_r`, or `%f_1, ..., %f_r` depending on the value of `%condition` at runtime.

Here's an example translation of a Python program and its corresponding IR:

```
def f(a, b, c):
    d = a + b
    if c:
        e = d + d
    else:
        e = b + d
    return e
```

```
graph(%a : Dynamic,
      %b : Dynamic,
      %c : Dynamic):
  %2 : int = prim::Constant[value=1]()
  %3 : Dynamic = aten::add(%a, %b, %2)
  %5 : Dynamic = prim::If(%c)
    block0():
      %6 : int = prim::Constant[value=1]()
      %7 : Dynamic = aten::add(%3, %3, %6)
      -> (%7)
    }
    block1():
      %8 : int = prim::Constant[value=1]()
      %9 : Dynamic = aten::add(%b, %3, %8)
      -> (%9)
  return (%5)
```

The outputs of the if-statement serve a role similar to that of a Φ (Phi) function node in traditional SSA control-flow graphs.

### Loops

Loops are implemented with `prim::Loop` which covers both `while` and `for` loops. A valid instantiation of this node always looks like this:

```
%y_1, ..., %y_r = prim::Loop(%max_trip_count, %initial_condition, %x_1, ..., %x_r)
  block0(%i, %a_1, ..., %a_r):
    %b_1, ..., %b_m = some::node(%a_value_from_outer_block, %a_1)
    %iter_condition = some::other_node(%a_2)
    -> (%iter_condition, %b_1, ..., %b_r)
```

The simplest way to explain the semantics is to consider this Python-like pseudo-code:

```
y_1, ..., y_r = x_1, ..., x_r
condition = initial_condition
i = 0
while condition and i < max_trip_count:
    a_1, ..., a_r = y_1, ..., y_r

    ############################################################
    # Actual body of the loop
    b_1, ..., b_m = some::node(a_value_from_outside_of_the_loop, a_1)
    iter_condition = some::node(a_2)
    ############################################################

    y_1, ..., y_r = b_1, ..., b_r
    condition = iter_condition
    i += 1
```

> *Note that translations of `for` loops simply pass in a constant `true` for both `%initial_condition` and `%iter_condition`. Translations of `while` loops set `%max_trip_count` to the largest value of `int64_t` and do not use `%i`. Those patterns are recognized by our interpreter and optimized accordingly (e.g. `while` loops don't maintain the loop counter).*

For example, this program:

```
def f(x):
    z = x
    for i in range(x.size(0)):
        z = z * z
    return z
```

can be translated as:

```
graph(%z.1 : Dynamic):
  %3 : bool = prim::Constant[value=1]()
  %1 : int = prim::Constant[value=0]()
  %2 : int = aten::size(%z.1, %1)
  %z : Dynamic = prim::Loop(%2, %3, %z.1)
    block0(%i : int, %5 : Dynamic):
      %z.2 : Dynamic = aten::mul(%5, %5)
      -> (%3, %z.2)
  return (%z)
```

### With

With-statements are represented in two different ways. For most of the compilation and optimization process, they are represented as a pair of `prim::Enter` and `prim::Exit` nodes that wrap the nodes corresponding to the body of the with-statement. However, with-statements are temporarily represented for the duration of the `exit_transform` pass using a block-based representation in which a `prim::With` node is inserted after the `prim::Exit` node, all of the nodes between the `prim::Exit` and `prim::Enter` are moved into the first

block of the `prim::With`, and the `prim::Exit` is moved into the second block of the `prim::With`. For example, this program:

```
with c as increment:
  y = x + increment
```

can be translated as:

```
%2 : int = prim::Constant[value=1]()
%increment.1 : int = prim::Enter(%c.1)
%y.1 : Tensor = aten::add(%x.1, %increment.1, %2)
%11 : Tensor = prim::Exit(%c.1)
```

and will temporarily be transformed to:

```
%increment.1 : int = prim::Enter(%c.1)
= prim::With()
  block0():
    %y.1 : Tensor = aten::add(%x.1, %increment.1, %4)
    -> ()
  block1():
    %11 : Tensor = prim::Exit(%c.1)
    -> ()
```

for the duration of the `exit_transform` pass.

## Value

ir.h

A `Value` represents data flowing through the operations in the program, e.g. the output of a matrix-multiply op. `Value` objects are always defined by a single `Node` ( `v.node()` ) due to single-static assignment form. For inputs to a Block/Graph, this node is a special `prim::Param` node that does not appear anywhere in the block's list of nodes. `Value` objects also have a Type (e.g. is it a Tensor? a list? a tuple?) that provides a static guarantee that its value will be of that Type.

A `Value` object has methods that return its definition ( `v.node()` ) and to all of its uses ( `v.uses()` ). Each Use has a pointer to the `Node` whose input list includes the value. Be careful when iterating over `v.uses()` while changing how `v` is used because each change to `v` will invalidate the `v.uses()` iterator.

Values are abstract representations of data in the program. When executing, the actual `Tensors`, list, tuples, etc. are stored in `IValues` (*interpreter* values), which are tagged unions of all possible value types in TorchScript. In retrospect the name `Value` is a bit confusing because it seems like it should be the tagged union, but it originally came from analogy to `llvm::Value`, which serves the same purpose as `jit::Value`.

## Type

aten/src/ATen/core/jit_type.h

TorchScript, unlike Python, is statically typed, so every `Value` has a Type associated with it, and every FunctionSchema has a list of argument types and a return type for a function. Type is the base class of a hierarchy of

C++ objects that represent the built-in types of TorchScript. Types provide methods such as `Type::isSubtypeOf` that describe the typing relationships. Common type are:

- TensorType - a `Tensor` with optionally refined information. It may know its device, type, requires_grad state, and number of dimensions. If it does know the number of dimensions it may know the size of a particular dimension.
- Tuples - e.g. `Tuple[Tensor, Int]`. Each member of the tuple is statically typed and the length of the tuple is statically known.
- List[T] - e.g. `List[Tensor]`. Mutable lists of a particular type.
- Optional[T] - e.g. `Optional[Tensor]`. Either a value of type T or None.
- Dict[K, V] - e.g. `Dict[String, Tensor]`. Dictionaries.

If type S is a subtype of P, then we can substitute an `IValue` that has type S anywhere something of type P is expected. This means that all subtyping relationships also require the representation of the `IValue` for subtypes to be compatible with the representation for the base type.

# Generating Programs

JIT programs are created using either the tracing frontend ( `torch.jit.trace` ) or the scripting frontend ( `torch.jit.script` ). In both cases, the result of these frontends is a complete Module that contains all the code in Methods, and all the model weights in the Parameters of the Module. However, each frontend goes through a different pathway for generating those Modules.

## Tracer

tracer.h tracer_state.h

The tracer produces graphs by recording what actual operations are done on `Tensors`. The entry point from Python into C++ for tracing using `torch.jit.trace` is `_create_method_from_trace`.

A thread local instance of the TracingState object maintains a mapping between actual data being computed during the trace (e.g. Tensors) stored in `IValues`, and the abstract `Value` in the `Graph` that would compute each value. The functions `void setValueTrace(const IValue&, Value*)` and `Value* getValueTrace(const IValue&)` are used by the tracer to maintain this mapping.

An initial `IValue` to `Value` mapping is set up between the inputs to the function being traced and symbolic `Value` inputs to the `Graph` being constructed. If we are tracing a `torch.nn.Module`, the tracer also adds Parameters and sub-Modules to the Module being constructed that correspond to the Python `torch.nn.Module` being traced. Mappings for these values are also added so that uses of the Parameters in the trace will create uses of the Parameters in the `Graph`.

As the trace runs, individual operators create `Nodes` in the `Graph` being traced to record what happens. This code is currently generated per operator in tools/autograd/gen_variable_type.py. It results in code that looks like the following:

```
torch::jit::Node* node = nullptr;
std::shared_ptr<jit::tracer::TracingState> tracer_state;
if (jit::tracer::isTracing()) {
        tracer_state = jit::tracer::getTracingState();
        at::Symbol op_name;
```

```
        op_name = jit::Symbol::fromQualString("aten::__ilshift__");
        node = tracer_state->graph->create(op_name, /*num_outputs=*/0);
        jit::tracer::recordSourceLocation(node);
        jit::tracer::addInputs(node, "self", self);
        jit::tracer::addInputs(node, "other", other);
        tracer_state->graph->insertNode(node);

        jit::tracer::setTracingState(nullptr);
}
TypeDefault::__ilshift__(self, other);
if (tracer_state) {
        jit::tracer::setTracingState(std::move(tracer_state));
        jit::tracer::addOutput(node, self);
}
```

The functions `addInputs` and `addOutput` are overloaded to handle the different data types that operators use.

set/getValueTrace only works on `Tensors` and `Futures`. Other types are not natively traced. Instead aggregates like tuples or lists are often flattened into `Tensors` at the end of a trace and explicitly constructed from individual `Tensors` at the beginning of this trace.

The tracer has special behavior when tracing calls to other TorchScript functions. This behavior is implemented in the GraphExecutor right before a `Graph` is about to be run. If tracing is enabled while running the graph, the GraphExecutor will disable tracing, run the graph as normal, and then inline the `Graph` into the trace. It then hooks up the `IValues` computed by running the `Graph` to out `Values` in the inlined graph.

> *When a trace calls a TorchScript function, that function is preserved as is, meaning that control-flow is preserved. This makes it possible to work around tracing issues by generating the subset of the program that cannot be traced using the script frontend and having the trace invoke it.*

The resulting `Graph` created by tracing is installed as the 'forward' method of the Module being created. A Module is produced regardless of whether the thing being traced was a function or a `torch.nn.Module`. In the function case, the Module produced will simply have a single `forward` function, no Parameters, and no sub-Modules.

## Script

The script frontend directly converts Python syntax into Modules. Like many compilers this happens in two phases. First, we generate an abstract syntax tree (AST), which is constructed out of Tree objects. The IR emitter then does semantic analysis on the Tree and lowers it into a Module. We can generate Trees in two ways: (1) using frontend.py, which takes the Python AST and transliterates it into Tree objects, or (2) via the Lexer and Parser which parse Python syntax directly. The Lexer+Parser path may seem redundant but it is crucially important. We need to define builtin functions (frontend/builtin_functions.cpp) when Python is not linked because we allow users to generate TorchScript programs directly from strings containing Python source code (api/include/torch/jit.h) without linking a full Python implementation (e.g. CPython). We also use this Python syntax as the serialization format for TorchScript, since it allows us to make changes to our IR without breaking backward compatibility. Furthermore, the Lexer is reused to implement the FunctionSchema parser, which turns FunctionSchema declarations from strings into FunctionSchema objects.

The following sections look into each the stages in the script frontend in detail.

## Tree

frontend/tree.h

Our frontends produce ASTs in the form of Tree objects. Trees are similar to s-expressions. Leafs (i.e. Atoms) are always strings. Compound trees have a `kind` (e.g `TK_CONST` or `TK_IDENT` defined in lexer.h) and a list of sub-trees. For instance, the Tree for `z.sigmoid() - (x + y)` is:

```
(-
        (+
          (variable (ident x))
          (variable (ident y)))
        (apply
          (.
                (variable (ident z))
                (ident sigmoid))
          (list)
          (list))))
```

This is printed in s-expression style with `(kind ...)` representing compound trees and `string_value` representing strings.

We provide utilities to construct, traverse, and print ASTs without a lot of complicated visitor infrastructure and inheritance.

Each Tree also has a mandatory SourceRange object that describes the range of text that it came from. These will be used for error reporting in the rest of the code.

## Tree Views

frontend/tree_views.h

Trees are easy to construct, visualize and traverse, but extracting information from a large compound tree like that of a function definition is unwieldy since it requires numeric indexing. Tree *Views* are a small layer on top of a tree that make it possible to create and de-structure trees of particular kinds. For example, here is the tree view for the apply node which provides named accessors for its subtrees: the function being called, the inputs, and the attributes (i.e. kwargs):

```
struct Apply : public Expr {
  Expr callee() const {
    return Expr(subtree(0));
  }
  List<Expr> inputs() const {
    return List<Expr>(subtree(1));
  }
  List<Attribute> attributes() const {
    return List<Attribute>(subtree(2));
  ...
};
```

The typical way to traverse a tree is to `switch` on the kind and then construct the appropriate Tree view:

```
switch (tree.kind()) {
  case TK_VAR:
```

```
        auto var = Var(tree); // construct tree view
      return environment_stack->getSugaredVar(var.name());
case '.': {
      auto select = Select(tree); // construct tree view
      auto sv = emitSugaredExpr(select.value(), 1);
      return sv->attr(select.range(), method, select.selector().name());
}
case TK_APPLY: {
      auto apply = Apply(tree); // construct tree view
      return emitApplyExpr(apply, n_binders);
} break;
```

## frontend.py

torch/jit/frontend.py

One way we construct Tree objects is directly from Python ASTs. This logic is contained inside frontend.py and is intentionally very minimal.

> *We endeavor to keep most of the JIT code written in C++, because most of the JIT functionality still needs to work without Python installed.*

This code simply constructs the Tree, filtering out the AST nodes of Python that we do not support.

## Lexer

frontend/lexer.h

When loading TorchScript code directly from a string, we using a standard Lexer+Parser combo. The Lexer takes an initial string and then exposes a stateful interface for walking the Tokens of the string, providing a standard set of functions:

- `next()` advances the lexer, returning the current token
- `cur()` provides the current token
- `lookahead()` provides the token coming after the current token
- `nextIf(int token_kind)` advances the token if it matches token kind.

Similar to Python, the Lexer handles the white-space sensitive nature of Python blocks. The Tokens `TK_INDENT`, `TK_DEDENT`, and `TK_NEWLINE` are injected into the token stream when code first becomes indented, when it dedents, and at the end of a statement. For instance for this stream:

```
if
  .
  .
```

We would get a token stream `TK_IF TK_NEWLINE TK_INDENT . TK_NEWLINE . TK_NEWLINE TK_DEDENT`. Unmatched opening brackets disable the injection of these tokens. The result is that the Parser can simply treat `TK_INDENT`, `TK_DEDENT` and `TK_NEWLINE` like C's `{`, `}`, and `;`.

## Tokens

frontend/lexer.h

Tokens are either keywords ( `def` ), operators ( `+` ), literals ( `3.4` ), or identifiers ( `foo` ). A `token_kind` integer identifies what it is and is the exact same type as the `kind` of a Tree. For single-character Tokens (e.g. `+` ), the kind is the same as the character, enable statements like:

```
if (lexer.nextIf('+')) {
        // handle + ...
}
```

Multi-character token kinds are defined in a list, `TC_FORALL_TOKEN_KINDS` . Tokens also have a `text()` field that records the actual string producing the token and is used by identifiers and literals to construct the actual values (e.g. the numeric value of a floating point literal).

## Parser

frontend/parser.h

The Parser uses the Lexer to build the AST for function definitions. `parseFunction` is the entrypoint for parsing a single `def ...` and will return a `Def` tree view.

The Parser is written as a top-down precedence parser, or "Pratt" parser. They are simpler and easier to understand than typical parser generators, while still being flexible enough to parse programming languages. For the most part parsing is done by recursive decent. To resolve operator precedence issues, the function to parse an expression is augmented with a precedent *p* such that calling the function means *parse an expression whose operators all have precedence higher than p*.

## IR Emitter

frontend/ir_emitter.h

The file ir_emitter.cpp translates Trees into Modules. The main entrypoint is `defineMethodsInModule` which takes a list of Def Tree Views representing function definitions and adds them as Methods to the module. During the lowering processing *semantic checking* occurs. The IR emitter checks that all used variables are defined (sometimes called scope checking), and that all values have compatible types (type-checking). During this process it also emits the graph nodes corresponding to each statement in the Tree and generates a FunctionSchema for the whole definition.

A few helper objects exist in the lowering process. SugaredValues are special values that represent objects that can appear during compilation but that are not first class values. For instance, in TorchScript methods `self` refers to the module, and `self.weight` refers to a Parameter of the module. Neither are first-class Types and have no corresponding `Value` in a graph. Resolver objects are `std::functions` that resolve externally-defined variables to SugaredValues. For instance, the identifier `torch` which contains most of our built-in ops is looked up through Resolver objects which interact with the Python state of the program.

The Environment tracks the mapping between variable names and the SugaredValues they refer to.

## SugaredValue

frontend/sugared_value.h

SugaredValues are how the IR emitter represents non-first class values during `Graph` creation. These values are things like the Module or a Python function call that do not have corresponding `Value` objects in the `Graph` .

The IR emitter *desugars* the SugaredValue objects to instructions in the graph based on how they are used. The SugaredValue class has a number of abstract methods on it such as `attr` or `call`. Consider the expression `self.foo`. For methods, `self` will resolve to a special SugaredValue subclass, ModuleValue. When the emitter sees `self.foo`, it will then call the ModuleValue function `sv.attr("foo")`, asking the ModuleValue how it should desugar itself when the attribute `"foo"` accessed. If `foo` is a parameter, it would then ensure that the parameter was added to the Method being compiled, and return a `SimpleValue` sugared value that contains the `Value` object representing the parameter as an input. If `foo` were a sub-Module then it would return another SugaredModule. The method `call` is invoked when the emitter sees the value used as a function call.

SugaredValues are also how we interact with Python runtime during the compilation process. For instance, `math.pi` is resolved to 3.1415... by first resolving `math` to a SugaredValue representing accesses to Python modules (PythonModuleValue) whose `attr` function turns Python numbers into `prim::Constant` `Nodes` in the graph.

Finally, normal `Values` are also represented by the SimpleValue SugaredValue in places where it is valid that either a SugaredValue or a normal `Value` will appear.

## Resolver

frontend/resolver.h

Any undefined variable during compilation is resolved with a call to an externally-provided Resolver. When called from Python (e.g `torch.jit.script`) this resolver interacts with the Python runtime via pybind11 to resolve symbols like `torch` and `math` to their Python equivalents.

*The combination of SugaredValue and Resolver decouples the implementation of the IR emitter from the pybind11 Python bindings that enable its interaction with the Python state.*

This makes it possible to use most of the IR emitter functionality when Python is not present.

## Environment

frontend/ir_emitter.cpp

The Environment object tracks the assignment of variable names during compilation. It is local to the IR emitter file. A stack of environments exist, with a new environment being created for sub-blocks introduced by control flow. The Environment keeps two tables, one for values which are not first class in the type system (SugaredValues) and a type table for values which are. When first class values are set, we emit a prim::Store, and when they are referenced we emit a prim::Load. SugaredValues are not re-assignable.

## Conversion To SSA

frontend/convert_to_ssa.cpp

As explained in the Block section, the IR is represented in structured control flow composed of ifs & loops. This makes it easier to optimize and lower to other compilers which do not support unstructured control flow. We lower Python control flow (break, continue, return) to this simplified form. We do closing over any variables in the environment, so we are able to convert all writes and reads from the environment directly to SSA form.

Conversion to SSA works in multiple parts.

- First, we add loads and stores to control flow operators (ifs & loops).

- Then we erase break & continue statements from the graph and replace them with `prim::LoopContinuation(%loop_continue_condition, %loop_carried_vars)`. Break statements have the continue condition set to false, and continue statements inline the loop condition. %loop_carried_vars are the loop carried variables of the inner most loop that contains the break or continue statement, are added by inserting prim::Load calls at the location of the statement.
- Then we inline the loop condition into the graph loops.
- Next we erase loads and stores, removing all stores and replacing all loads with whatever the in-scope value of the variable name is.
- Finally, we remove `prim::LoopContinuation` s and `prim::ReturnStmt` s in the exit_transform pass.

## Exit Transform

frontend/exit_transforms.cpp

This pass takes in a graph where LoopContinuation & ReturnStmts exist in the graph and erases them, correctly setting block outputs. `prim::LoopContinuation(*vals)` means that the values are targeting the most recent loop block. `prim::ReturnStmt(*vals)` means that the values are targeting the most recent `Closure` or `Graph` `Block`.

If a block has an exit node, no further instructions will be executed until the exit target has been reached. If we encounter a node that contains nested blocks that may have hit an exit node, such as an if statement that exits in one block and does not exit in the other, we use a boolean value to indicate if the exit has been hit or not. Then, we conditionalize further execution.

Python example:

```
while i < 5:
  if i == 3:
    i += 1
    continue
  i += 2
```

-> transforms to

```
continue_loop = i < 5
while continue_loop:
  if i == 3:
    i = i + 1
    continue_loop = i < 5
    did_exit = True
  if did_exit:
    pass
  else:
    i = i + 2
    continue_loop = i < 5
```

The pass also keeps track of nodes or blocks that will always throw Exceptions so that we do not unnecessarily conditionalize execution. In the following example, we can treat the if statement as always Returning and remove the `print` statement.

```
if i < 0:
  raise Exception("Negative input")
else:
  return math.sqrt(i)
print(i)  # unreachable code
```

In the above example, the if statement will have one output: `math.sqrt(i)` on the false branch, and `prim::Uninitialized` in the true branch. `prim::Uninitialized` is inserted by the compiler when it can prove the value will never be used. It can be introduced by exceptions, breaks, continues, and returns.

We initially considered doing the Transform pass before Loads and Stores were removed from the graph. However, this breaks when a loop carried variable is captured in a break or continue and then is refined in the rest of the loop body. In the below example, at the point of the `continue`, `x` has type `Optional[int]` but is refined to `int` after the continue statement.

```
...
if cond:
  if i < 3:
      x = torch.jit.annotate(Optional[int], None)
      continue
  x = 1
else:
  x = 2
print(x)
```

If we were to rearrange the graph before loads & stores were removed:

```
if cond:
  if i < 3:
    x = torch.jit.annotate(Optional[int], None)
    did_continue = True
    continue
  else:
    did_continue = False
  if not did_continue:
    x = 1
else:
  x = 2
if not did_continue:
  print(x)
```

The type of `x` at the print statement would be `Optional[int]`, which breaks its original type.

## Python-Compiler Interaction

python/script_init.cpp

A set of special SugaredValues are used to translate between objects in the Python environment and `Values` in the `Graph` during the compilation process. The entry-point for this behavior is `toSugaredValue(py::object`

`obj, ...)` which takes a pybind11 Python value and figures out how to turn it into an appropriate SugaredValue. `Values` exist to represent Python functions, Python modules, and `ScriptModule` objects.

# Executing Programs

TorchScript is executed using an interpreter attached to a JIT-optimizer and compiler. The entry-point for execution is the GraphExecutor object that is created on demand inside a Method when the method is first called. This section first goes over the semantics of graphs, i.e. what does it mean to execute a graph? And then details how the implementation works.

## Evaluation Semantics

TorchScript programs implement a very small subset of Python that is necessary to run models.

TorchScript includes immutable value types:

- `int`
- `float`
- `Tuple[T0, T1, ...]`

As well as mutable reference types:

- `Tensor`
- `List[T]`
- `Dict[K, V]`

A value of a reference type points to an underlying memory location where the data for the reference type is stored, and variable assignment for a reference type can cause multiple values to point to the same underlying data. This is similar to Python's class model.

It is important to remember that TorchScript uses these semantics for `Tensors` so not all computation on `Tensor` is pure. Individual `Tensors` may be *views* of the same underlying data. Views are established by special view creating operations, such as indexing into a Tensor:

```
t = torch.rand(3, 4)
t2 =  t[0] # view of one slice of t
```

Some builtin operators also mutably write to the underlying `Tensor`. In the standard library these operators are always named with a trailing underscore, or take a named `out` `Tensor` where the result is written:

```
t2.relu_() # inplace relu operator, note t is modified as well!
torch.add(t, t, out=t) # update t, without using temporary memory if possible
```

The combination of reference semantics and mutable operators can be more difficult to optimize, but it gives program writers powerful control of the memory usage of their programs. For instance, DenseNets use a concat operation instead of the addition found in a ResNet. Rather than compute a concat of existing `Tensors`, many implementations use `Tensor` indexing and `out` keywords to avoid allocating addition memory for the activations. Ideally a compiler would always be able to do these optimizations, but in practice new ideas are tried all the time that exist outside what compiler writers expect.

In addition to being mutable, `Tensors` also have a set of dynamically determined properties (i.e. properties that can vary from run to run) this includes:

- dtype - their data type int, float, double, etc.
- device - where the `Tensor` lives, e.g. the CPU, or CUDA GPU 0
- rank - the number of dimensions that the `Tensor` has
- size - the precise size of the Tensor
- requires_grad - whether the `Tensor` is recording its gradient with autograd

Changes in these properties change how operators on `Tensors` will evaluate and would make certain optimization invalid. For instance, if we have a fuser capable of generating new CUDA kernels but not CPU kernels, it is only valid to fuse operations where the inputs are known to run only on CUDA devices. The GraphExecutor's job is to still enable optimization even when certain combinations of properties prevent optimizations from occurring.

Nodes in a graph are executed *serially* in the order they appear in a block. `Nodes` may be reordered either during optimization or by the interpreter itself if it can be proven that the new order is not distinguishable from the original execution order. These semantics are necessary since the combination of mutable `Tensors` and potential aliases between `Tensors` makes it unsafe to perform arbitrary reordering otherwise. However, the AliasInfo object can accurately track how alias propagate through builtin operators so optimization passes can query when certain reorders or optimizations are safe.

We also provide user-accessible parallel execution through the `fork` and `wait` primitives. The `fork` primitive begins execution of `fn` in parallel with the current thread of execution, immediately returning a Future object that will hold the result of the forked function. The `wait` method of the future then causes the invoking thread to wait for the value being computed by `fn`.

```
def fn(arg0, arg1, ...):
  ...
  return v

fut = torch.jit.fork(fn, arg0, arg1, ...)
...
v = torch.jit.wait(fut)
```

Currently, the user is responsible for avoiding races between threads. We encourage users to not write to `Tensors` visible from other threads, and may enforce this more strictly in the future.

Optimization passes that wish to exploit multi-threaded execution may automatically convert serial `Blocks` into parallel execution by inserting extra fork and wait events. This design enables our users to manually specify parallelism while also allowing optimization passes to exploit it when safe and profitable.

## IValue

ivalue.h

All evaluation involves computation using `IValues`, 16-byte tagged unions that can hold the concrete representation of any type in TorchScript. TorchScript is statically typed, so it would be possible to operate on unboxed primitive types, but the interface between interpreter, built-in ops and user functions would be significantly more complicated. A single tagged union keeps these interfaces simple and since most objects are `Tensors` anyway, the overhead of storing a tag is small compared to the data stored in the `Tensors`.

IValue contains methods to check the type (e.g. `isTensor()`) and to convert to particular to type (e.g. `toTensor()`). We do not publicly expose the type tag and force clients to use the `isX` methods. This enables us to change the underlying implementation of `IValue` later, e.g. to use an 8-byte value with NaN-boxing. Most operators work on a specific static type, so dynamic dispatch on the tag is not frequently required.

## Operation

All builtin operators are represented using a stack machine concept. An operator pops its arguments off the top of the stack and pushes its result to the stack:

```
using Stack = std::vector<IValue>;
using Operation = std::function<void(Stack*)>;

// schema: example_add(Tensor a, Tensor b) -> Tensor
void example_add(Stack* stack) {
    Tensor a, b;
    // stack before: ? ? ? a b <- back
    pop(stack, a, b); // Templated helper function
                      // that pops a, b and converts them to Tensor
    push(stack, a + b);
    // stack after:
    // ? ? ? c <- back
}
```

Most operations, apart from some vararg primitive operators like prim::Unpack, have an associated FunctionSchema that describes how many inputs will be popped and how many will be pushed.

The stack concept makes it easy to define operators with variable numbers of inputs and outputs without the need to allocate vectors of inputs and outputs for each individual operator.

In practice, the interpreter will allocate one Stack, and it will eventually reach a sufficient size such that no more stack-related memory allocations will occur.

## Operator

runtime/operator.h

The Operator object represents a single registered operator in the system. It combines a FunctionSchema that describes how an Operation executes with a method to look up the corresponding Operation given the Node representing the operator in a Graph. Most Operators are defined by providing a FunctionSchema and an Operation function. However, primitives like prim::Unpack require knowledge of their Node to know how to operate (e.g. how many elements to unpack). These Operators have a function that takes a `Node*` and returns an operation.

## Interpreter

runtime/interpreter.cpp

The interpreter is responsible for the straightforward execution of `Graphs` without any optimization. It is composed of two objects: Code and InterpreterState. Code is a linearized representation of the `Graph` into simple stack-machine Instructions. Code is shared among all the executions of the `Graph` and will include caches for certain operations like the generated CUDA code of FusionGroups.

The InterpreterState is unique to each execution of the `Graph`. It holds a list registers with the intermediate `IValues` used in the execution, the Stack being used by each Operation, and the program counter tracking the position in the instructions. The information represents the complete state of the interpreter. `wait` instructions can cause the interpreter to suspend, and the InterpreterState is used to resume execution where the `wait` occurred, potentially on a different thread.

Instructions in the interpreter have three parts: a list of registers from which to gather `IValues` onto the stack before the instruction, the Operation to run, and a list of registers in which to store the results of the Operation. Alternatively, we could have used individual instructions to load/store values from the stack to registers, but this design was easier to implement, requires fewer instructions since each instruction does more things, and has not yet been a performance bottleneck. Each Operation returns a potential relative jump to compute the next program counter.

Unlike typical interpreters, we do not attempt careful register allocation. Since `Tensors` are reference types, saving registers would only save a few hundred bytes of space in typical applications by cutting down on the number of places a reference could be saved. The data in single a `Tensor` is likely significantly bigger than that, so we forgo register allocation to make debugging easier.

However, we do need to ensure that values are destructed immediately after their last use. Because Torch reference counts `Tensors`, they will be deallocated immediately when their last reference is gone. To ensure we use a minimum amount of memory we want to ensure that the interpreter releases the reference as soon as it is no longer used. To do this, each Instruction also has a set of flags which indicate the inputs to the operation which will no longer be used after the operation. For these inputs, the `IValue` is moved rather than copied from the register file, ensuring the reference will go dead as soon as the Operation no longer needs it. extra instructions may be inserted into the program to explicitly drop references for values whose last use depends on the control flow of the program.

The following is an example program in `Graph` form and its equivalent in interpreter Instructions:

```
graph(%x : Tensor,
      %hx : Tensor,
      %cx : Tensor,
      %w_ih : Tensor,
      %w_hh : Tensor,
      %b_ih : Tensor,
      %b_hh : Tensor):
  %7 : int = prim::Constant[value=4]()
  %8 : int = prim::Constant[value=1]()
  %9 : Tensor = aten::t(%w_ih)
  %10 : Tensor = aten::mm(%x, %9)
  %11 : Tensor = aten::t(%w_hh)
  %12 : Tensor = aten::mm(%hx, %11)
  %13 : Tensor = aten::add(%10, %12, %8)
  %14 : Tensor = aten::add(%13, %b_ih, %8)
  %gates : Tensor = aten::add(%14, %b_hh, %8)
  %16 : Tensor[] = aten::chunk(%gates, %7, %8)
  %ingate.1 : Tensor, %forgetgate.1 : Tensor, %cellgate.1 : Tensor, %outgate.1 :
Tensor = prim::ListUnpack(%16)
  %ingate : Tensor = aten::sigmoid(%ingate.1)
  %forgetgate : Tensor = aten::sigmoid(%forgetgate.1)
  %cellgate : Tensor = aten::tanh(%cellgate.1)
  %outgate : Tensor = aten::sigmoid(%outgate.1)
  %25 : Tensor = aten::mul(%forgetgate, %cx)
```

```
  %26 : Tensor = aten::mul(%ingate, %cellgate)
  %cy : Tensor = aten::add(%25, %26, %8)
  %28 : Tensor = aten::tanh(%cy)
  %hy : Tensor = aten::mul(%outgate, %28)
  %30 : (Tensor, Tensor) = prim::TupleConstruct(%hy, %cy)
  return (%30)
```

```
0, 1, 2, 3, 4, 5, 6 = Load
7 = Constant
8 = t move(3)
9 = mm move(0), move(8)
10 = t move(4)
11 = mm move(1), move(10)
12 = add move(9), move(11), 7
13 = add move(12), move(5), 7
14 = add move(13), move(6), 7
15, 16, 17, 18 = ConstantChunk move(14)
19 = sigmoid move(15)
20 = sigmoid move(16)
21 = tanh move(17)
22 = sigmoid move(18)
23 = mul move(20), move(2)
24 = mul move(19), move(21)
25 = add move(23), move(24), move(7)
26 = tanh 25
27 = mul move(22), move(26)
28 = TupleConstruct move(27), move(25)
 = Store move(28)
```

## Graph Executor

[runtime/graph_executor.cpp](runtime/graph_executor.cpp)

All program execution starts with a graph executor. Its responsible for running optimizations (potentially involving the JIT-compilation of fused kernel code), and then handing the `Graph` or subcomponents of it off to an interpreter to actually run.

In this section, we use a running example program that computes one step of an LSTM to show how the graph is transformed:

This section will use an example this LSTM program:

```
@torch.jit.script
def LSTMCellS(x, hx, cx, w_ih, w_hh, b_ih, b_hh):
    gates = x.mm(w_ih.t()) + hx.mm(w_hh.t()) + b_ih + b_hh
    ingate, forgetgate, cellgate, outgate = gates.chunk(4, 1)
    ingate = torch.sigmoid(ingate)
    forgetgate = torch.sigmoid(forgetgate)
    cellgate = torch.tanh(cellgate)
    outgate = torch.sigmoid(outgate)
    cy = (forgetgate * cx) + (ingate * cellgate)
```

```
    hy = outgate * torch.tanh(cy)
    return hy, cy
```

After going through the the frontend, we start with this unoptimized graph:

```
graph(%x : Tensor,
      %hx : Tensor,
      %cx : Tensor,
      %w_ih : Tensor,
      %w_hh : Tensor,
      %b_ih : Tensor,
      %b_hh : Tensor):
  %7 : int = prim::Constant[value=4]()
  %8 : int = prim::Constant[value=1]()
  %9 : Tensor = aten::t(%w_ih)
  %10 : Tensor = aten::mm(%x, %9)
  %11 : Tensor = aten::t(%w_hh)
  %12 : Tensor = aten::mm(%hx, %11)
  %13 : Tensor = aten::add(%10, %12, %8)
  %14 : Tensor = aten::add(%13, %b_ih, %8)
  %gates : Tensor = aten::add(%14, %b_hh, %8)
  %16 : Tensor[] = aten::chunk(%gates, %7, %8)
  %ingate.1 : Tensor, %forgetgate.1 : Tensor, %cellgate.1 : Tensor, %outgate.1 :
Tensor = prim::ListUnpack(%16)
  %ingate : Tensor = aten::sigmoid(%ingate.1)
  %forgetgate : Tensor = aten::sigmoid(%forgetgate.1)
  %cellgate : Tensor = aten::tanh(%cellgate.1)
  %outgate : Tensor = aten::sigmoid(%outgate.1)
  %25 : Tensor = aten::mul(%forgetgate, %cx)
  %26 : Tensor = aten::mul(%ingate, %cellgate)
  %cy : Tensor = aten::add(%25, %26, %8)
  %28 : Tensor = aten::tanh(%cy)
  %hy : Tensor = aten::mul(%outgate, %28)
  %30 : (Tensor, Tensor) = prim::TupleConstruct(%hy, %cy)
  return (%30)
```

Execution starts in `GraphExecutor::run`, which takes a Stack of inputs.

### Specialization

The executor *specializes* the `Graph` for the particular set of inputs. Specialization is handled by the `ArgumentSpec` object which extracts a "signature" composed of all the properties being specialized. We only specialize to the properties of `Tensors`. The `ArgumentSpec` only records properties for `Tensors` that either appear directly in the inputs to the graph or inside Tuples that are inputs to the `Graph`. The properties recorded are currently:

- dtype
- rank, but not size
- requires_grad
- device type (CPU, CUDA)
- defined - whether the `Tensor` exists or is a placeholder

The ArgumentSpec object is used as a key into a cache that holds pre-optimized Code objects (held in an ExecutionPlan object). On a cache hit, an InterpreterState is created and the Code in the cache is run.

### Pre-derivative Optimization

On a code cache miss, we generate a new optimized `Graph` on the fly ( `compileSpec` ). It starts by creating a copy of the initial `Graph` and setting the input types to the specialized `Tensor` types observed in this specialization. TensorType inputs to the `Graph` will get refined with types that know the device, number of dimensions, and requires grad state.

```
# post specialization, inputs are now specialized types
graph(%x : Float(*, *),
      %hx : Float(*, *),
      %cx : Float(*, *),
      %w_ih : Float(*, *),
      %w_hh : Float(*, *),
      %b_ih : Float(*),
      %b_hh : Float(*)):
  %7 : int = prim::Constant[value=4]()
  %8 : int = prim::Constant[value=1]()
  %9 : Tensor = aten::t(%w_ih)
  %10 : Tensor = aten::mm(%x, %9)
  %11 : Tensor = aten::t(%w_hh)
  %12 : Tensor = aten::mm(%hx, %11)
  %13 : Tensor = aten::add(%10, %12, %8)
  %14 : Tensor = aten::add(%13, %b_ih, %8)
  %gates : Tensor = aten::add(%14, %b_hh, %8)
  %16 : Tensor[] = aten::chunk(%gates, %7, %8)
  %ingate.1 : Tensor, %forgetgate.1 : Tensor, %cellgate.1 : Tensor, %outgate.1 :
Tensor = prim::ListUnpack(%16)
  %ingate : Tensor = aten::sigmoid(%ingate.1)
  %forgetgate : Tensor = aten::sigmoid(%forgetgate.1)
  %cellgate : Tensor = aten::tanh(%cellgate.1)
  %outgate : Tensor = aten::sigmoid(%outgate.1)
  %25 : Tensor = aten::mul(%forgetgate, %cx)
  %26 : Tensor = aten::mul(%ingate, %cellgate)
  %cy : Tensor = aten::add(%25, %26, %8)
  %28 : Tensor = aten::tanh(%cy)
  %hy : Tensor = aten::mul(%outgate, %28)
  %30 : (Tensor, Tensor) = prim::TupleConstruct(%hy, %cy)
  return (%30)
```

### Required Passes

It then runs "required passes", which are graph transformations necessary to generate legal graphs for the interpreter. (Some passes such as differentiation will introduce `Nodes` that are not defined by operators and require passes to clean up. The combination of `specializeUndef` and `LowerGradOf` clean up these operations.) These passes also remove broadcasting "expand" nodes that get implicitly inserted by the tracer but are not valid for all sizes.

It then runs inference passes to calculate properties of the graph given this particular specialization:

- It propagates constants, pre-computing as much as possible
- It propagates the input ranks, dtypes, devices, and requires_grad information to the rest of the graph where possible.

```
graph(%x : Float(*, *),
      %hx : Float(*, *),
      %cx : Float(*, *),
      %w_ih : Float(*, *),
      %w_hh : Float(*, *),
      %b_ih : Float(*),
      %b_hh : Float(*)):
  %8 : int = prim::Constant[value=1]()
  %9 : Float(*, *) = aten::t(%w_ih)
  %10 : Float(*, *) = aten::mm(%x, %9)
  %11 : Float(*, *) = aten::t(%w_hh)
  %12 : Float(*, *) = aten::mm(%hx, %11)
  %13 : Float(*, *) = aten::add(%10, %12, %8)
  %14 : Float(*, *) = aten::add(%13, %b_ih, %8)
  %gates : Float(*, *) = aten::add(%14, %b_hh, %8)
  %31 : Float(*, *), %32 : Float(*, *), %33 : Float(*, *), %34 : Float(*, *) =
prim::ConstantChunk[chunks=4, dim=1](%gates)
  %ingate : Float(*, *) = aten::sigmoid(%31)
  %forgetgate : Float(*, *) = aten::sigmoid(%32)
  %cellgate : Float(*, *) = aten::tanh(%33)
  %outgate : Float(*, *) = aten::sigmoid(%34)
  %25 : Float(*, *) = aten::mul(%forgetgate, %cx)
  %26 : Float(*, *) = aten::mul(%ingate, %cellgate)
  %cy : Float(*, *) = aten::add(%25, %26, %8)
  %28 : Float(*, *) = aten::tanh(%cy)
  %hy : Float(*, *) = aten::mul(%outgate, %28)
  %30 : (Float(*, *), Float(*, *)) = prim::TupleConstruct(%hy, %cy)
  return (%30)
```

### Derivative Preserving Optimization

It then runs a number of *derivative preserving* optimization passes. If a computation involves `Tensors` that have `requires_grad` and it is valid to compute its derivative, then these passes are only allowed to replace that computation with another computation that is also differentiable. In other words, these passes cannot break autograd. Algebraic rewrites and peephole optimizations are generally derivative preserving but something that generates code, like pointwise fusion, is not.

Current derivative preserving passes:

- Eliminating dead code
- Eliminating common subexpressions
- Pooling redundant constants into single values
- Peephole optimizations, including some algebraic rewrites into simpler operations
- Unrolling small loops
- Batching matrix multiplications that result from unrolling loops

```
graph(%x : Float(*, *),
      %hx : Float(*, *),
```

```
      %cx : Float(*, *),
      %w_ih : Float(*, *),
      %w_hh : Float(*, *),
      %b_ih : Float(*),
      %b_hh : Float(*)):
  %8 : int = prim::Constant[value=1]()
  %9 : Float(*, *) = aten::t(%w_ih)
  %10 : Float(*, *) = aten::mm(%x, %9)
  %11 : Float(*, *) = aten::t(%w_hh)
  %12 : Float(*, *) = aten::mm(%hx, %11)
  %13 : Float(*, *) = aten::add(%10, %12, %8)
  %14 : Float(*, *) = aten::add(%13, %b_ih, %8)
  %gates : Float(*, *) = aten::add(%14, %b_hh, %8)
  %31 : Float(*, *), %32 : Float(*, *), %33 : Float(*, *), %34 : Float(*, *) =
prim::ConstantChunk[chunks=4, dim=1](%gates)
  %ingate : Float(*, *) = aten::sigmoid(%31)
  %forgetgate : Float(*, *) = aten::sigmoid(%32)
  %cellgate : Float(*, *) = aten::tanh(%33)
  %outgate : Float(*, *) = aten::sigmoid(%34)
  %25 : Float(*, *) = aten::mul(%forgetgate, %cx)
  %26 : Float(*, *) = aten::mul(%ingate, %cellgate)
  %cy : Float(*, *) = aten::add(%25, %26, %8)
  %28 : Float(*, *) = aten::tanh(%cy)
  %hy : Float(*, *) = aten::mul(%outgate, %28)
  %30 : (Float(*, *), Float(*, *)) = prim::TupleConstruct(%hy, %cy)
  return (%30)
```

### Post-derivative optimization

The next optimization depends on whether any part of the graph actual requires a gradient to be calculated, which is determined by `needsGradient`. In the case where no gradients are required (i.e. for inference graphs), then we can directly apply optimizations that generate graphs that may not have valid gradients defined. For now this is the `FuseGraph` pass, which looks for adjacent point-wise operations along with reviewing operations such as `split` and `concat`, and creates `prim::FusionGroup` `Nodes` in the graph to replace these operations. The Operator registered to execute `prim::FusionGroup` nodes will generate a new CUDA kernel for each unique `Node`, which replaces the original separate execution.

Note the two phases for compilation of fusion groups: First, the `FuseGraph` pass splits the `Graph` into fusible sub-Graphs and returns the resulting `Graph` to the graph executor. Second, when the `Graph` is turned into Code, the Operation for the FusionGroup node will be looked up and a new CUDA kernel generated for the body. Other compilers should work in a similar way by first introducing a new operator into the `Graph` where the compiled code should run, and then registering an Operator that implements that `Node` which performs the actual compilation.

In the case where no gradients are required, the optimization process is finished, a Code object is constructed from the `Graph`, it is added to the code cache, and then an InterpreterState is constructed and run.

```
graph(%x : Float(*, *),
      %hx : Float(*, *),
      %cx : Float(*, *),
      %w_ih : Float(*, *),
```

```
      %w_hh : Float(*, *),
      %b_ih : Float(*),
      %b_hh : Float(*)):
  %9 : Float(*, *) = aten::t(%w_ih)
  %10 : Float(*, *) = aten::mm(%x, %9)
  %11 : Float(*, *) = aten::t(%w_hh)
  %12 : Float(*, *) = aten::mm(%hx, %11)
  %77 : Tensor[] = prim::ListConstruct(%b_hh, %b_ih, %10, %12)
  %78 : Tensor[] = aten::broadcast_tensors(%77)
  %79 : Tensor, %80 : Tensor, %81 : Tensor, %82 : Tensor = prim::ListUnpack(%78)
  %hy : Float(*, *), %cy : Float(*, *) = prim::FusionGroup_0(%cx, %82, %81, %80, %79)
  %30 : (Float(*, *), Float(*, *)) = prim::TupleConstruct(%hy, %cy)
  return (%30);

with prim::FusionGroup_0 = graph(%13 : Float(*, *),
      %71 : Tensor,
      %76 : Tensor,
      %81 : Tensor,
      %86 : Tensor):
  %87 : Float(*, *), %88 : Float(*, *), %89 : Float(*, *), %90 : Float(*, *) =
prim::ConstantChunk[chunks=4, dim=1](%86)
  %82 : Float(*, *), %83 : Float(*, *), %84 : Float(*, *), %85 : Float(*, *) =
prim::ConstantChunk[chunks=4, dim=1](%81)
  %77 : Float(*, *), %78 : Float(*, *), %79 : Float(*, *), %80 : Float(*, *) =
prim::ConstantChunk[chunks=4, dim=1](%76)
  %72 : Float(*, *), %73 : Float(*, *), %74 : Float(*, *), %75 : Float(*, *) =
prim::ConstantChunk[chunks=4, dim=1](%71)
  %69 : int = prim::Constant[value=1]()
  %70 : Float(*, *) = aten::add(%77, %72, %69)
  %66 : Float(*, *) = aten::add(%78, %73, %69)
  %62 : Float(*, *) = aten::add(%79, %74, %69)
  %58 : Float(*, *) = aten::add(%80, %75, %69)
  %54 : Float(*, *) = aten::add(%70, %82, %69)
  %50 : Float(*, *) = aten::add(%66, %83, %69)
  %46 : Float(*, *) = aten::add(%62, %84, %69)
  %42 : Float(*, *) = aten::add(%58, %85, %69)
  %38 : Float(*, *) = aten::add(%54, %87, %69)
  %34 : Float(*, *) = aten::add(%50, %88, %69)
  %30 : Float(*, *) = aten::add(%46, %89, %69)
  %26 : Float(*, *) = aten::add(%42, %90, %69)
  %ingate : Float(*, *) = aten::sigmoid(%38)
  %forgetgate : Float(*, *) = aten::sigmoid(%34)
  %cellgate : Float(*, *) = aten::tanh(%30)
  %outgate : Float(*, *) = aten::sigmoid(%26)
  %14 : Float(*, *) = aten::mul(%forgetgate, %13)
  %11 : Float(*, *) = aten::mul(%ingate, %cellgate)
  %cy : Float(*, *) = aten::add(%14, %11, %69)
  %4 : Float(*, *) = aten::tanh(%cy)
  %hy : Float(*, *) = aten::mul(%outgate, %4)
  return (%hy, %cy)
```

### Derivate Splitting

Many `Graphs` will require gradients (i.e. one of the inputs will have a `requires_grad` property set). In this case, it is unsafe to run post-derivative optimizations directly on the `Graph`. Instead, our approach is to first *split* the `Graph` into sub-Graphs where symbolic gradient formulas are known and produce an explicit `Graph` for the forward pass along with a complementary `Graph` that implements the backwards pass using some of the values computed in the forward pass. We can then apply post-derivative optimization to the forward graph. The "gradOutputs" for the backwards graph are only known when the backward pass runs, so we cannot fully optimize it at this time. For instance, we do not know if some of those gradOutputs will also `require_grad` meaning that a gradient-of-gradient situation exists. Instead the backward pass will use a new GraphExecutor object to run and optimize its execution. In this way, we can handle an indefinite number of recursive gradient calculations.

The creating of derivative subgraphs is done using a similar approach to finding fusion groups: adjacent operations with known gradient formulas are grouped together into `prim::DifferentiableGraph` nodes. We only generate these nodes if we can find a large enough subgraph where optimization is likely to be profitable since there is some overhead involved in entering and exiting a differentiable subgraph.

```
graph(%x : Float(*, *),
      %hx : Float(*, *),
      %cx : Float(*, *),
      %w_ih : Float(*, *),
      %w_hh : Float(*, *),
      %b_ih : Float(*),
      %b_hh : Float(*)):
  %8 : int = prim::Constant[value=1]()
  %hy : Float(*, *), %cy : Float(*, *) = prim::DifferentiableGraph_0(%cx, %b_hh,
%b_ih, %hx, %w_hh, %x, %w_ih)
  %30 : (Float(*, *), Float(*, *)) = prim::TupleConstruct(%hy, %cy)
  return (%30)
with prim::DifferentiableGraph_0 = graph(%13 : Float(*, *),
      %29 : Float(*),
      %33 : Float(*),
      %40 : Float(*, *),
      %43 : Float(*, *),
      %45 : Float(*, *),
      %48 : Float(*, *)):
  %49 : Float(*, *) = aten::t(%48)
  %47 : Float(*, *) = aten::mm(%45, %49)
  %44 : Float(*, *) = aten::t(%43)
  %42 : Float(*, *) = aten::mm(%40, %44)
  %38 : int = prim::Constant[value=1]()
  %39 : Float(*, *) = aten::add(%47, %42, %38)
  %35 : Float(*, *) = aten::add(%39, %33, %38)
  %gates : Float(*, *) = aten::add(%35, %29, %38)
  %24 : Float(*, *), %25 : Float(*, *), %26 : Float(*, *), %27 : Float(*, *) =
prim::ConstantChunk[chunks=4, dim=1](%gates)
  %ingate : Float(*, *) = aten::sigmoid(%24)
  %forgetgate : Float(*, *) = aten::sigmoid(%25)
  %cellgate : Float(*, *) = aten::tanh(%26)
  %outgate : Float(*, *) = aten::sigmoid(%27)
  %14 : Float(*, *) = aten::mul(%forgetgate, %13)
```

```
  %11 : Float(*, *) = aten::mul(%ingate, %cellgate)
  %cy : Float(*, *) = aten::add(%14, %11, %38)
  %4 : Float(*, *) = aten::tanh(%cy)
  %hy : Float(*, *) = aten::mul(%outgate, %4)
  return (%hy, %cy)
```

## JIT Logging

jit_log.h

Logging is a very useful debugging technique, especially in the context of compilers. Compilers perform a series of passes and analyses and logging can help to trace issues such as wrong results or segmentation faults all the way back to the original erroneous transformation.

`TorchScript` offers a simple logging facility that can enabled by setting an environment variable `PYTORCH_JIT_LOG_LEVEL`.

Logging is enabled on a per file basis. To enable logging in `dead_code_elimination.cpp`, `PYTORCH_JIT_LOG_LEVEL` should be set to `dead_code_elimination.cpp` or, simply, to `dead_code_elimination` (i.e. `PYTORCH_JIT_LOG_LEVEL=dead_code_elimination`).

Multiple files can be logged by separating each file name with a colon `:` as in the following example, `PYTORCH_JIT_LOG_LEVEL=dead_code_elimination:guard_elimination`

There are 3 logging levels available for your use ordered by the detail level from lowest to highest.

- `GRAPH_DUMP` should be used for printing entire graphs after optimization passes
- `GRAPH_UPDATE` should be used for reporting graph transformations (i.e. node deletion, constant folding, etc)
- `GRAPH_DEBUG` should be used for providing information useful for debugging the internals of a particular optimization pass or analysis

The current logging level is `GRAPH_UPDATE` meaning that both `GRAPH_DUMP` and `GRAPH_UPDATE` will be enabled when one specifies a file(s) in `PYTORCH_JIT_LOG_LEVEL`.

`GRAPH_DEBUG` can be enabled by prefixing a file name with an `>` as in `>alias_analysis`. `>>` and `>>>` are also valid and **currently** are equivalent to `GRAPH_DEBUG` as there is no logging level that is higher than `GRAPH_DEBUG`.

By default, types in the graph are printed with maximum verbosity. The verbosity level can be controlled via the environment variable `PYTORCH_JIT_TYPE_VERBOSITY`. The available settings are:

- `0`: No type information
- `1`: Types and shapes only
- `2`: Also print strides
- `3`: Also print device type and whether gradient is required

## JIT Optimization Limiter

jit_opt_limit.h

Often times, we need to limit the number of optimizations for any lowering passes for debugging purposes.

`TorchScript` offers a simple optimization limit checker that can be configured through environment variable `PYTORCH_JIT_OPT_LIMIT`. The purpose is to limit how many optimization you can make per pass. This is useful for debugging any passes.

Opt limit checker is enabled on a per file basis (hence per pass). For example, in `constant_propagation.cpp`, `PYTORCH_JIT_OPT_LIMIT` should be set to `constant_propagation=<opt_limit>` where `<opt_limit>` is the number of optimizations you want to make for the pass. (i.e. `PYTORCH_JIT_OPT_LIMIT="constant_propagation=<opt_limit>"`).

Multiple files can be configured by separating each file name with a colon `:` as in the following example, `PYTORCH_JIT_OPT_LIMIT="constant_propagation=<opt_limit>:dead_code_elimination=<opt_limit>"`

You can call opt limiter by calling a macro `JIT_OPT_ALLOWED`. It will return true if we haven't reached the optimization limit yet. Typical usage:

```
if (!JIT_OPT_ALLOWED) {
    GRAPH_DUMP(...); //supplied from jit_log
    return;
}
```

## DifferentiableGraphOp

runtime/graph_executor.cpp

A DifferentiableGraphOp combines an explicit forward `Graph` `f` with a paired backward graph `df`. When it runs, the input `Tensors` to `f` are detached from the autograd, the body of `f` is run, and then the autograd graph for the outputs of `f` are hooked up to the `df` function. The `df` function's outputs are also hooked up to the autograd graph.

## Handling Mutability

### Aliasing and mutation in the PyTorch API

In PyTorch, `Tensors` are reference types. Operators can return "views" of the input `Tensor`, creating a new `Tensor` object that shares the same underlying storage as the original:

```
a = torch.rand(2, 3)
b = a
# At this point, `a` and `b` share their storage.
c = b[0]
# `c` shares storage with `a` and `b`, but only sees a slice of the allocated
memory.
```

Some operators will *mutate* one or more of their operands in-place. These are typically denoted with a trailing underscore, or by taking an `out` argument as input:

```
a = torch.zeros(2, 3)
b = torch.ones(2, 3)
```

```
a.add_(b)  # in-place add, so `a` is modified.
torch.add(a, b, out=a) # another way to express the same thing
```

### Aliasing and mutation annotations in FunctionSchema

The JIT's `FunctionSchema` allows operator writers to add annotations specifying the aliasing and mutation behavior of an operator. Optimization passes will use this information to determine whether transformations are semantics-preserving. This section provides a description of the alias annotation language, assuming that the reader already knows what `FunctionSchema` looks like.

First, here is a pure function which always returns new memory:

```
add(Tensor a, Tensor b) -> Tensor
```

The type `Tensor` with no annotations is sugar for "fresh, read-only `Tensor`". So since there are no annotations on anything, we know that this operator creates no aliases and mutates no inputs.

Next, a function that returns an alias to one of the inputs.:

```
view(Tensor(a) self, int[] size) -> Tensor(a)
```

The shared `(a)` annotation on `self` and the output signify that the `Tensors` will share the same storage. Another way to say is that `self` and the output belong to the same "alias set" `a`.

Now a function that writes in-place to one of the inputs (note the trailing underscore):

```
add_(Tensor(a!) self, Tensor other) -> Tensor(a!)
```

The `!` annotation means that this operator writes to the specified alias set (in this case `a`).

Sometimes we don't have enough information to provide an exact alias annotation. For example, here is the operator to extract an element from a list:

```
list_select(Tensor[] list, int idx) -> Tensor(*)
```

Note the alias set `*`. This is the **wildcard set**. These are values which we conservatively analyze. Containers, such as lists and dictionaries, Graph inputs, and class attributes are conservatively analyzed to all alias. In most cases, people shouldn't be writing operators with wildcard annotations. They are used as temporary workaround for when our alias analysis isn't sophisticated enough to understand something yet but we don't want to block feature development.

Similarly, we have operators which result in Tensors being contained in a list. In this case, to preserve the relationship between output list and input, we annotate that the input enters the wildcard set with the `(a -> *)` syntax.

```
func: chunk(Tensor(a -> *) self, int chunks, int dim=0) -> Tensor(a)[]
```

This annotation language is consumed by the `FunctionSchema` parser, which produces `AliasInfo` objects summarizing the aliasing relationships for each schema `Argument`.

### Alias Analysis in the IR

ir/alias_analysis.h

An alias analysis pass consumes the per-operator aliasing information to construct a database of aliasing and mutation relationships in a graph, called `AliasDb`. This section focuses on the alias analysis pass; the public interface to `AliasDb` will be described later.

The core data structure in the AliasDb is called `MemoryDAG`, which is a DAG where the edges are "may point to" relationships and the vertices are aliasing `Element`s. The most common kind of `Element` is an IR `Value`, but there are other kinds of things that can alias that aren't first-class `Value`s in the IR, like wildcards or contained types (such as in a list or tuple).

The alias analysis pass walks through the nodes in a graph, examining schema `AliasInfo` objects and adding edges in the `MemoryDAG` accordingly. For example, for the node:

```
%output : Tensor = aten::view(%self, %size)
```

the analyzer will examine the schema for `view()`:

```
view(Tensor(a) self, int[] size) -> Tensor(a)
```

and add an edge from `%output` to `%self`. The alias analysis pass is flow-insensitive, as we are only adding "points-to" edges when processing a node.

As a more involved example, the following TorchScript snippet:

```
@torch.jit.script
def foo(a : Tensor, b : Tensor):
  c = 2 * b
  a += 1
  if a.max() > 4:
    r = a[0]
  else:
    r = b[0]
  return c, r
```

Will produce a graph like this:

 AliasTracker graph

A few things to note:

- "Graph Input Element" is an example of an `Element` that isn't a first-class `Value`. Alias analysis happens on a per-function level, so we don't necessarily know the aliasing relationships of the inputs. The only safe assumption is that `a` and `b` may alias each other, so they point to a special `Element` that describes "the world outside of this function".
- `r` may point to either `a` or `b`, depending on the runtime value of `a.max()`. A given `Element` may point to multiple other `Element`s. This can happen if there is branching control flow (like in this example), or with certain ops like `contiguous()`, which either returns an alias to the input or a fresh `Tensor`, depending on the runtime characteristics of the input.
- `c` is a fresh `Tensor` (i.e. it doesn't point to anything) since it was created using the pure operation `2 * b`.

The last point demonstrates a key concept: *leaf elements uniquely describe memory locations*. Since a leaf element doesn't point to anything, the memory that backs it must have been freshly allocated by some op. Thus we can use leaf elements to represent disjoint memory locations.

So to determine whether `a` and `b` may alias, we traverse the `MemoryDAG` DAG and figure out if `a` and `b` share any leaf nodes. If they do, then we know `a` and `b` might point to the same memory location, i.e. `a` and `b` may alias. This kind of query is common enough that `MemoryDAG` does path compression to speed up leaf-finding, so that aliasing queries can be serviced in amortized constant time.

### Writing optimization passes with `AliasDb`

`AliasDb` provides a high-level interface to help people write mutability-safe optimization passes.

In particular, `moveAfterTopologicallyValid()` (and its `moveBefore` variant) will reorder nodes in a way that preserves data dependencies and avoids any data hazards. The rules for this are that all mutable *writes* to a given memory location must occur in the same order (avoid WAW hazards), and that no reads can be reordered before or after any write (WAR, RAW hazards).

However, reordering of reads across writes *is allowed* if we can prove that the read cannot alias the thing being written. This happens whenever we have `Tensors` that come from functions that produce fresh results (common) inside of the function. It also happens whenever the creation of the mutable `Tensor` is seen in the function (so it gets assigned a fresh variable), and all of its writes occur in that function.

The intention is that if you only mutate the graph through `AliasDb`, you don't have to think about mutability/aliasing at all in your pass. As we write more passes, the interface to `AliasDb` will get richer (one example is transforming an in-place operation to its pure equivalent if we can prove it's safe).

`AliasDb` also provides lower level APIs that users of LLVM's alias analysis pass would be familiar with, such as querying whether any two `Value`s may alias.

TODO: differentiation, symbolic autograd, fusion, operators

# Profiling Programs

`prim::profile` nodes are inserted on every **use** of a value by `ProfilingRecord::instrumentBlock`. Every `prim::profile` node runs a lambda that uses a captured, initial type value and the type of an incoming `Tensor` and merges the two into a refined `TensorType`.

`prim::profile` nodes are replaced with `prim::Guard` nodes by `InsertGuards`. `prim::Guard` nodes are inserted to guarantee that beyond the guard a guarded `Tensor` will always be of the profiled shape. This guarantee will enable optimizations and code generators to generate more efficient code.

We attempt to reduce the number of `prim::Guard` nodes as these nodes may interfere with optimizations.

- First, `GuardElimination::moveGuardsToDefs` tries to move `prim::Guards` to their definitions, so the guards guarding the same `Tensor` follow the definition directly or another guard on the same `Tensor`.
- This ordering allows us to **coalesce** (done in `GuardElimination::coalesceGuards`) multiple guards into a single one.
- After guards are **coaslesced**, `GuardElimination::eliminateGuards` attempts to eliminate more guards as follows: it inspects each operation and its inputs. It checks if inputs to the operation are guarded and also if the operation produces the consistent shapes given the guarded inputs. For example, if two

inputs to `add` are guaranteed to be of shape `(2, 3)`, the output shape will also always be `(2, 3)`. If this property holds, we are allowed to remove the guard guarding operation's output.

Lastly, we need to be handle cases when the assumptions about `Tensor` shapes fail at runtime. To handle guard failures, we need to be able to run the original code i.e. the code that doesn't rely on assumptions about shapes. As guards can be inserted and moved (by Optimizer) at/to arbitrary points in a computational graph, we need to be able to resume execution starting from those arbitrary points onward.

`InsertBailoutNodes` builds deoptimized versions of the original computational graph, that contain the rest of computations starting from their corresponding guard failure points and also captures live values needed to execute those deoptimized graphs. In other words, the pass replaces `prim::Guard` nodes with `prim::BailOut` nodes which have the `attr::Subgraph` attributes set to the deoptimized versions of the remaining computations at their corresponding `prim::Guard` s.

# Saving Programs

See the serialization docs.

# Testing Programs

## Testing Autodiff

runtime/symbolic_script.cpp

When differentiating a graph, each node that has a symbolic gradient will be included in a `prim::DifferentiableGraph`. We fall back to using autograd for the node if there isn't a gradient formula for it. Adding/updating symbolic gradient functions must be tested carefully as it's easy to get CI green by comparing autograd result with itself, but potentially cause an autodiff support regression.

If your PR adds/updates a gradient formula for `torch` / `nn` functions, you **MUST** enable/update the corresponding tests in

- `torch` functions: `method_tests` in common_method_tests.py
- `nn` functions: `nn_functional_tests` in test_jit.py

To turn on autodiff check, you can add an optional `check_ad(should_autodiff_node[bool], nonfusible_nodes[str|list[str]], fusible_nodes[str|list[str]])` tuple after the optional test variant name field. If `should_autodiff_node=True`, the differentiated traced/script forward graph must have a `prim::DifferentiableGraph`.

All nodes in `nonfusible_nodes` should show up at least once in `prim::DifferentiableGraph` subgraphs. When fusion is enabled, all nodes in `fusible_nodes` should show up in one of `prim::FusionGroup` graphs attached to `prim::DifferentiableGraph`, otherwise they're checked as `nonfusible_nodes` as well. On the other hand, if `should_autodiff_node=False`, the graph can still have `prim::DifferentiableGraph` with other nodes, but not `nonfusible_nodes` and `fusible_nodes`.

To make writing tests easier, you only need to write out node names if it's different from the function name. Below are a few examples:

```
('conv1d', ...), # No symbolic gradient formula
('avg_pool2d', ..., (True,)), # Has symbolic gradient formula, only has one
nonfusible node aten::avg_pool2d
('nll_loss', ..., (True, 'aten::nll_loss_forward')), # Is replaced by a different
node in its symbolic gradient formula
('dropout', ..., (True, ['prim::is_CUDA', 'aten::bernoulli_'], ['aten::rand_like',
..., 'aten::div'])), # Some ops are fused when fusion is enabled
```

Note that even for the same function, different tests could trigger different function schemas (e.g `aten::add`) while only a few of them have symbolic gradient formulas. You should only turn on autodiff checks in tests that have symbolic gradients. If you are not sure, uncomment the debugging line in runtime/symbolic_script.cpp to check which function schema the test triggers.

# Python Printer

serialization/python_print.cpp serialization/import_source.cpp

The Python Printer takes a `Graph` and produces Python-like code that represents the same graph. Using some special values in serialization/import_source.cpp, this code can be read back in by the compiler to produce the same `Graph`. In Python a `ScriptModule`'s `code` property shows the Python Printed graph.

The table below shows the graph and code for this small `ScriptModule`:

```
class M(torch.jit.ScriptModule):
    @torch.jit.script_method
    def forward(self, x, y, z):
        # type: (Tensor, int, float) -> Tensor
        if y > 2:
            x = x + z
        else:
            x = x + y
        return x

m = M()
```

`m.graph`

```
graph(%x.1 : Tensor,
      %y : int,
      %z : float):
  %5 : int = prim::Constant[value=1]()
  %3 : int = prim::Constant[value=2]()
  %4 : bool = aten::gt(%y, %3)
  %x : Tensor = prim::If(%4)
    block0():
      %x.2 : Tensor = aten::add(%x.1, %z, %5)
      -> (%x.2)
    block1():
      %x.3 : Tensor = aten::add(%x.1, %y, %5)
```

```
      -> (%x.3)
  return (%x)
```

`m.code`

```
def forward(self,
    x: Tensor,
    y: int,
    z: float) -> Tensor:
  if torch.gt(y, 2):
    x0 = torch.add(x, z, 1)
  else:
    x0 = torch.add(x, y, 1)
  return x0
```

# Python Bindings

TODO: Script Module, torch.jit.trace, **constant** handling, weak script modules