Towards Relay: a New IR for Machine Learning Frameworks

Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, Zachary Tatlock
Tension between performance and flexibility
Tension between performance and flexibility
Tension between performance and flexibility
Tension between performance and flexibility
From OpenAI’s recent blog post: https://blog.openai.com/ai-and-compute/
“We believe the largest training runs today employ hardware that cost in the single digit millions of dollars to purchase (although the amortized cost is much lower).”

--- Open AI Blog
Growing compute

- The community is addressing need for cost effective compute with new hardware designs.

- TPU, Trillium, A10 Bionic, Brainwave, etc

- Hardware landscape is becoming very heterogeneous, mix of CPUs, GPUs, custom accelerators.
Growing compute

- Different operating environments; ex. can be memory hungry in cloud, but not edge devices.

- Introducing new compute may increase runtime efficiency.

- Doesn’t account for programming and porting costs.
  
  - For ex. Cloud FPGAs
Leveraging diversity

• Current state of the art is to port and tweak models by hand for each hardware platform until they work.

• How to write programs for many different devices, optimize for:
  • Memory
  • Quantization
  • New numeric representations
  • Model transforms
  • Layout change
  • Device scheduling
VTA

- Take our friend Thierry who has been building new hardware accelerators for ML.
- How to program the hardware?
- How to port existing models?
- How to adapt software for different HW designs?
Portability + Flexibility

• We need models that can be effectively optimized and run on a variety of devices.

• We want generic models, but tuned implementations

• Can we build custom hardware from directly from model descriptions?

• “Write once, run everywhere”
TVM

• An end-to-end compiler stack for deep learning.

• Hierarchal intermediate representations, tightly integrated for tuning models for specific hardware targets.

• TVM is currently focused on producing high performance operator implementations.

• TVM is bottom-up.
Relay

• We contribute a new high level IR for TVM named Relay.

• Generalize computation graphs to differentiable programs.

• Write Python (in the style of PyTorch) but apply end-to-end optimizations.

• Composed of new front-end, IR, auto-diff, optimizer, backend, and runtime.

• Relay is top-down.
graph, lib, params = 
   t.compiler.build(graph, target, params)

module = runtime.create(graph, lib, t.cuda(0))
module.set_input(**params)
module.run(data=data_array)
output = t.nd.empty(out_shape, ctx=t.cuda(0))
module.get_output(0, output)
What Relay will replace

graph, lib, params = t.compiler.build(graph, target, params)

module = runtime.create(graph, lib, t.cuda(0))
module.set_input(**params)
module.run(data=data_array)
output = t.nd.empty(out_shape, ctx=t.cuda(0))
module.get_output(0, output)
Why not current frameworks or IRs?

• We believe the key to being able to optimize programs effectively is a typed, whole program representation of machine learning models.

• We will show how current framework’s IRs are lacking, then examine how Relay addresses these challenges.
DL Frameworks Compilers

• We are at the dawn of the compiler age for deep learning.

• Framework designers realize performance is being left on the table, and frameworks are converging on compilation pipelines.
  
  • XLA for TF, Glow for PyTorch, NNVM/TVM for MxNet

• Other IRs are framework first, we want to be IR first!

• Need “whole model” to do certain classes of optimization, analogous to “whole program” in traditional compilers.
  
  • But we want flexibility, portability, and performance!
Advantages:

+ Embedded domain specific language
+ Dataflow graph gives rise to straightforward execution and scheduling.
+ The graph is easy to optimize and compile, for example static memory planning.
+ XLA style compilation is straightforward.

Disadvantages:

- Embedded domain specific language
- Users write programs to build graph and later execute.
- Staging can be complex and confusing.
- IR is computation graph (i.e a data flow graph) with embedded control and mutation.
- Ex. What does a gradient of an impure function mean?
x = tf.placeholder(tf.float32, shape=(None, D_in))
y = tf.placeholder(tf.float32, shape=(None, D_out))

w1 = tf.Variable(tf.random_normal((D_in, H)))
w2 = tf.Variable(tf.random_normal((H, D_out)))

h = tf.matmul(x, w1)
h_relu = tf.maximum(h, tf.zeros(1))
y_pred = tf.matmul(h_relu, w2)

loss = tf.reduce_sum((y - y_pred) ** 2.0)

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)

for _ in range(500):
    loss_value, _, _ = sess.run([loss, new_w1, new_w2], feed_dict={x: x_value, y: y_value})

Adapted from: https://github.com/jcjohnson/pytorch-examples/blob/master/autograd/two_layer_net_autograd.py
Advantages:

+ Shallow embedding, users just interact with normal Python APIs
+ Expressive, can use all of Python to interact with PyTorch, as it is the execution layer upto tensors.
+ Trace based auto-diff over a subset of Python, can handle arbitrary control flow.
+ Can accelerate pieces using Glow and Tensor Comprehensions

Disadvantages:

- Trace based JIT and exporting, only capture specific execution traces.
- Not “whole model”
- Python is “control plane”
- C extensions are “data plane”; requires C extensions
- Incredibly limited and brittle export functionality.
Tracing based tools fail if traces change at all (i.e. essentially static graph)

Limitations

- The ONNX exporter is a trace-based exporter, which means that it operates by executing your model once, and exporting the operators which were actually run during this run. This means that if your model is dynamic, e.g., changes behavior depending on input data, the export won't be accurate. Similarly, a trace is likely to be valid only for a specific input size (which is one reason why we require explicit inputs on tracing.) We recommend examining the model trace and making sure the traced operators look reasonable.

- PyTorch and Caffe2 often have implementations of operators with some numeric differences. Depending on model structure, these differences may be negligible, but they can also cause major divergences in behavior (especially on untrained models.) In a future release, we plan to allow Caffe2 to call directly to Torch implementations of operators, to help you smooth over these differences when precision is important, and to also document these differences.
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()
    print(t, loss.item())
    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad

        w1.grad.zero_()
        w2.grad.zero_()

Adapted from: https://github.com/jcjohnson/pytorch-examples/blob/master/autograd/two_layer_net_autograd.py
A graph is created on the fly

\[
x = \text{torch.randn}(1, 10) \\
\text{prev}_h = \text{torch.randn}(1, 20) \\
W_h = \text{torch.randn}(20, 20) \\
W_x = \text{torch.randn}(20, 10)
\]
A graph is created on the fly

\[
x = \text{torch.randn}(1, 10) \\
\text{prev}_h = \text{torch.randn}(1, 20) \\
W_h = \text{torch.randn}(20, 20) \\
W_x = \text{torch.randn}(20, 10)
\]
System Design

- Computational Graph
- High level Data-flow Rewriting
- Tensor Operator Description
- Schedule

Accelerators, LLVM, CUDA/Metal/OpenCL

Frameworks: Caffe, CNTK, CoreML
System Design

Relay

Fusion, Layout Change, Partial Eval, Traditional Optimizations

Control

Relay runtime system

Tensor Operator Description

Schedule

Hardware Implementation

Operators

Relay Python Decorator

Frameworks

Caffe²

CoreML

CNTK
Language

- Functional higher order language
  - Closures
  - Tensors
  - Control flow
  - References
  - Shape dependent type system
- Differentiable
Language

- Functional higher order language
  - Closures
  - Tensors
  - Control flow
  - References
  - Shape dependent type system
  - Differentiable

Old PL you know and love
Language

• Functional higher order language

• Closures

• Tensors

• Control flow

• References

• Shape dependent type system

• Differentiable

Old PL you know and love

New challenges
Frontend

• Our current frontend is a subset of Python.

• We use AST rewriting to transform the Python program into our IR directly.

• We can statically analyze this subset, and type check it.

• We rely on MyPy’s infrastructure (annotations, and typed_ast).

```python
@relay
def linear_loss(a, b, x, y):
y_hat = a * x + b
return (y - y_hat)**2
```
If we remove all syntactic sugar we can see a little more what’s going on:

```python
@relay
def linear_loss(
a: Tensor[Float, (1, 1)],
b: Tensor[Float, (1, 1)],
x: Tensor[Float, (1, 1)],
y: Tensor[Float, (1, 1)]) -> Tensor[Float, (1, 1)]:
y_hat = relay.broadcast_add(relay.broadcast_mul(a, x), b)
diff = relay.broadcast_sub(y, y_hat)
return relay.broadcast_mul(diff, diff)
```
If we remove all syntactic sugar we can see a little more what’s going on:

We can use Python’s type annotations to provide type info.

```python
@relay
def linear_loss(a: Tensor[Float, (1, 1)], b: Tensor[Float, (1, 1)], x: Tensor[Float, (1, 1)], y: Tensor[Float, (1, 1)])-> Tensor[Float, (1, 1)]:
y_hat = relay.broadcast_add(relay.broadcast_mul(a, x), b)
diff = relay.broadcast_sub(y, y_hat)
return relay.broadcast_mul(diff, diff)
```
If we remove all syntactic sugar we can see a little more what’s going on:

We can use Python’s type annotations to provide type info.

```python
@relay
def linear_loss(a: Tensor[Float, (1, 1)], b: Tensor[Float, (1, 1)], x: Tensor[Float, (1, 1)], y: Tensor[Float, (1, 1)]) -> Tensor[Float, (1, 1)]:
    y_hat = relay.broadcast_add(relay.broadcast_mul(a, x), b)
    diff = relay.broadcast_sub(y, y_hat)
    return relay.broadcast_mul(diff, diff)
```

Relay is extensible with user defined operators; these are implemented in a library, and map to TVM operators.
If we remove all syntactic sugar we can see a little more what’s going on:

Decorator does the magic!

We can use Python’s type annotations to provide type info.

```
@relay
def linear_loss(
    a: Tensor[Float, (1, 1)],
    b: Tensor[Float, (1, 1)],
    x: Tensor[Float, (1, 1)],
    y: Tensor[Float, (1, 1)])-> Tensor[Float, (1, 1)]:
    y_hat = relay.broadcast_add(relay.broadcast_mul(a, x), b)
    diff = relay.broadcast_sub(y, y_hat)
    return relay.broadcast_mul(diff, diff)
```

Relay is extensible with user defined operators; these are implemented in a library, and map to TVM operators.
TVM FFI

- TVM has a powerful system which allows us to export C++ infrastructure to Python
  - Able to pass data (incl. closures) back and forth.
- Python frontend is a vanilla decorator, calls into C++
  - We will use this trick many times.
- C++ AST inherits from a special super class, gets Python interoperability for cheap with some boilerplate.

```cpp
class IfNode : public Node {
    public:
        Expr guard;
        Expr true_b;
        Expr false_b;
    ...
};
```

Enables interoperability

🐇
Produce a single function’s Relay representation

```python
def compile_func(f):
    """Compile a single Python function to a Relay function.
    ..."
    source = inspect.getsource(f)
    func = ast.parse(source)
    ...  
    return compile_func_to_defn(func)

def relay(func):
    """The Relay decorator."""
    env = get_env()
    defn = compile_func(func)
    env.add(defn)
    ...
```
Type System

• Stratified type system with type, shape, and base type dependency.

• Type system has a limited form of dependency, possible to write functions from shapes to types.

• We implement type checking, and inference over the shape inference relied on by traditional ML frameworks.

• i.e capture ad-hoc shape inference formally
For example we can type operators which work over all base types and shapes.

```plaintext
relay.tensor_add :
  forall (bt : BaseType) (s : Shape),
  Tensor bt s ->
  Tensor bt s ->
  Tensor bt s

relay.tensor_mul :
  forall (bt : BaseType) (s1 s2 : Shape),
  Tensor bt s1 ->
  Tensor bt s1 ->
  Tensor bt (mul_output_shape s1 s2)
```
\[ i \in \mathbb{Z} \]
\[ \Delta; \Gamma \vdash i : \text{Tensor(IntType(32), Shape()}) \]  (Type-Int-Literal)

\[ f \in \mathbb{R} \]
\[ \Delta; \Gamma \vdash f : \text{Tensor(FloatType(32), Shape())} \]  (Type-Float-Literal)

\[ b \in \{\text{True}, \text{False}\} \]
\[ \Delta; \Gamma \vdash b : \text{Tensor(BoolType, Shape())} \]  (Type-Bool-Literal)

\[\Delta \vdash s = \text{Shape}(d_1, d_2, \ldots, d_n) \]
\[\Delta \vdash b : \text{BaseType} \]
\[\Delta; \Gamma \vdash t_1 : \text{Tensor}(b, s) \]
\[\Delta; \Gamma \vdash t_2 : \text{Tensor}(b, s) \]
\[\Delta; \Gamma \vdash \text{Shape}(m, d_1, d_2, \ldots, d_n) \]  (Type-Tensor-Literal)

\[\Delta; \Gamma \vdash p_1 : T_1 \]
\[\Delta; \Gamma \vdash p_2 : T_2 \]
\[\ldots \]
\[\Delta; \Gamma \vdash p_n : T_n \]
\[\Delta; \Gamma \vdash (p_1, p_2, \ldots, p_n) : T_1 \times T_2 \times \cdots \times T_n \]  (Type-Product)

\[\Delta; \Gamma \vdash p : T_1 \times T_2 \times \cdots \times T_n \]
\[\Delta; \Gamma \vdash p[i] : T_i \]  (Type-Projection)

\[\Delta; \Gamma \vdash d : T \]
\[\Delta; \Gamma \vdash \text{let id = d in } b : T' \]  (Type-Let)

\[\Delta \vdash b : \text{BaseType} \]
\[\Delta \vdash s : \text{Shape} \]
\[\Delta; \Gamma \vdash t : \text{Tensor}(b, s) \]
\[\Delta; \Gamma \vdash \text{UnaryOp}(b, t) : \text{Tensor}(b, s) \]  (Type-UnaryOp)

\[\Delta \vdash \text{op} \in \{\text{+, -, \times, /}\} \]
\[\Delta \vdash b : \text{BaseType} \]
\[\Delta \vdash s : \text{Shape} \]
\[\Delta; \Gamma \vdash t_1 : \text{Tensor}(b, s) \]
\[\Delta; \Gamma \vdash t_2 : \text{Tensor}(b, s) \]
\[\Delta; \Gamma \vdash \text{BinaryOp}(\text{op}, t_1, t_2) : \text{Tensor}(b, s) \]  (Type-NonComp-BinaryOp)

\[\Delta \vdash \text{op} \in \{\text{\texttt{>, <, >=, <=}}\} \]
\[\Delta ; \vdash b : \text{BaseType} \]
\[\Delta \vdash s : \text{Shape} \]
\[\Delta; \Gamma \vdash t_1 : \text{Tensor}(b, s) \]
\[\Delta; \Gamma \vdash t_2 : \text{Tensor}(b, s) \]
\[\Delta; \Gamma \vdash \text{BinaryOp}(\text{op}, t_1, t_2) : \text{Tensor}(b, s) \]  (Type-Comp-BinaryOp)

\[\Delta; \Gamma \vdash \text{def } f(p_1, p_2, \ldots, p_n : T_n) \Rightarrow T', \text{body} : (T_1 \times T_2 \times \cdots \times T_n) \Rightarrow T' \]  (Type-Function-Definition)

\[\Delta \vdash a_1 : T_1 \]
\[\Delta \vdash a_2 : T_2 \]
\[\ldots \]
\[\Delta \vdash a_n : T_n \]
\[\Delta; \Gamma \vdash f(a_1, a_2, \ldots, a_n) : T' \]  (Type-Call)

\[\Delta; \Gamma \vdash \text{let } \text{if } c \text{ then } b_1 \text{ else } b_2 : T \]  (Type-If)

\[\Delta \vdash c : \text{Tensor(BoolType, Shape())} \]
\[\Delta; \Gamma \vdash b_1 : T \]
\[\Delta; \Gamma \vdash b_2 : T \]  (Type-If)

\[\Delta \vdash b : \text{BaseType} \]
\[\Delta \vdash s : \text{Shape} \]
\[\Delta; \Gamma \vdash \text{Zero} : \text{Tensor}(b, s) \]  (Type-Zero)

\[\Delta; \Gamma \vdash \text{autodiff}(e) : T \]  (Type-Gradient)

\[\Delta; \Gamma \vdash \text{Grad } e : T \]  (Type-Grad)

\[\Delta; \Gamma \vdash \text{n} : T \]  (Type-Ref)

\[\Delta; \Gamma \vdash \text{Ref } \text{n} : \text{RefType}(T) \]  (Type-Val-Ref)

\[\Delta; \Gamma \vdash \text{ref } t : T \]  (Type-Val-Ref)

\[\Delta \vdash r : \text{RefType}(T) \]
\[\Delta; \Gamma \vdash v : T \]
\[\Delta; \Gamma \vdash r := v : () \]  (Type-Set-Ref)

Figure 6. Rules for deriving types of expressions and definitions. The unit type, (), is syntactic sugar for a product type with zero members. Note that these type rules assume that all type variables in quantifiers have already been concretely instantiated. Additionally, in the rule for gradient, “autodiff” is the automatic differentiation AST transformation on expression e; rather than attempt to capture the entire semantics of the transformation in that inference rule, we explain the transformation in 4.2.
$i \in \mathbb{Z}$
\[ \Delta; \Gamma \vdash i : \text{Tensor(32, Shape())} \]  
\( \text{(Type-Int-Literal)} \)

$f \in \mathbb{R}$
\[ \Delta; \Gamma \vdash f : \text{Tensor(32, Shape())} \]  
\( \text{(Type-Float-Literal)} \)

$b \in \{\text{True}, \text{False}\}$
\[ \Delta; \Gamma \vdash b : \text{Tensor(BoolType, Shape())} \]  
\( \text{(Type-Bool-Literal)} \)

\[ \begin{align*}
\Delta; \Gamma \vdash s &= \text{Shape}(d_1, d_2, \ldots, d_a) \\
\Delta; \Gamma \vdash b &= \text{BaseType} \\
\Delta; \Gamma \vdash t_1 &= \text{Tensor}(b, s) \\
\Delta; \Gamma \vdash t_2 &= \text{Tensor}(b, s) \\
\ldots \\
\Delta; \Gamma \vdash t_m &= \text{Tensor}(b, s)
\end{align*} \]
\[ \Delta; \Gamma \vdash [t_1, t_2, \ldots, t_m] : \text{Tensor}(b, \text{Shape}(d_1, d_2, \ldots, d_a)) \]
\( \text{(Type-Comp-Binary)} \)

\[ \Delta; \Gamma \vdash c : \text{Tensor(BoolType, Shape())} \]
\[ \Delta; \Gamma \vdash b_1 : T \quad \Delta; \Gamma \vdash b_2 : T \]
\[ \Delta; \Gamma \vdash \text{if } c \text{ then } b_1 \text{ else } b_2 : T \]  
\( \text{(Type-If)} \)

\[ \Delta; \Gamma \vdash \text{let } id = d \text{ in } b : T' \]  
\( \text{(Type-Let)} \)

\[ \Delta; \Gamma \vdash \text{Grad } e : T \]  
\( \text{(Type-Gradient)} \)

\[ \begin{align*}
op &\in \{\neg, \text{sq}\} \\
\Delta; \Gamma \vdash b &= \text{BaseType} \\
\Delta; \Gamma \vdash s &= \text{Shape} \\
\Delta; \Gamma \vdash t &= \text{Tensor}(b, s) \\
\Delta; \Gamma \vdash \text{UnaryOp}(\mathit{op}, t) &= \text{Tensor}(b, s) \\
\end{align*} \]
\( \text{(Type-UnaryOp)} \)

\[ \begin{align*}
op &\in \{\times, \div, /\} \\
\Delta; \Gamma \vdash b &= \text{BaseType} \\
\Delta; \Gamma \vdash s &= \text{Shape} \\
\Delta; \Gamma \vdash t_1 &= \text{Tensor}(b, s) \\
\Delta; \Gamma \vdash t_2 &= \text{Tensor}(b, s) \\
\Delta; \Gamma \vdash \text{BinaryOp}(\mathit{op}, t_1, t_2) &= \text{Tensor}(b, s) \\
\end{align*} \]
\( \text{(Type-NonComp-BinaryOp)} \)

\[ \Delta; \Gamma \vdash \text{Ref } n : \text{RefType}(T) \]  
\( \text{(Type-Ref)} \)

\[ \Delta; \Gamma \vdash r : \text{RefType}(T) \]
\[ \Delta; \Gamma \vdash !r : T \]  
\( \text{(Type-Val-Ref)} \)

\[ \Delta; \Gamma \vdash r : \text{RefType}(T) \quad \Delta; \Gamma \vdash v : T \quad \Delta; \Gamma \vdash r := v : () \quad \text{ TYPE-SET-REF} \]

**Figure 6.** Rules for deriving types of expressions and definitions. The unit type, (), is syntactic sugar for a product type with zero members. Note that these type rules assume that all type variables in quantifiers have already been concretely instantiated. Additionally, in the rule for gradient, “autodiff” is the automatic differentiation AST transformation on expression e; rather than attempt to capture the entire semantics of the transformation in that inference rule, we explain the transformation in 4.2.
Figure 6. Rules for deriving types of expressions and definitions. The zero members. Note that these type rules assume that all type variables. Additionally, in the rule for gradient, "autodiff" is the automatic differentiation mechanism than attempt to capture the entire semantics of the transformation in
All values are Tensor typed in Relay

\[
\Delta; \Gamma \vdash i : \text{Tensor(}\text{IntType}(32), \text{Shape}())
\]

\[
\Delta; \Gamma \vdash f : \text{Tensor(}\text{FloatType}(32), \text{Shape}())
\]

\[
b \in \{\text{True}, \text{False}\}
\]

\[
\Delta; \Gamma \vdash b : \text{Tensor(}\text{BoolType}, \text{Shape}())
\]

\[
\Delta; \Gamma \vdash s = \text{Shape}(d_1, d_2, \ldots, d_n)
\]

\[
\Delta; \Gamma \vdash b : \text{BaseType}
\]

\[
\Delta; \Gamma \vdash t_1 : \text{Tensor}(b, s)
\]

\[
\Delta; \Gamma \vdash t_2 : \text{Tensor}(b, s)
\]

\[
\Delta; \Gamma \vdash [t_1, t_2, \ldots, t_m] : \text{Tensor}(b, \text{Shape}(m, d_1, d_2, \ldots, d_n))
\]

\[
\Delta; \Gamma \vdash c : \text{Tensor(}\text{BoolType}, \text{Shape}())
\]

\[
\Delta; \Gamma \vdash b_1 : T
\]

\[
\Delta; \Gamma \vdash b_2 : T
\]

\[
\Delta; \Gamma \vdash \text{if } c \text{ then } b_1 \text{ else } b_2 : T
\]

\[
\Delta; \Gamma \vdash i \in \mathbb{Z}
\]

\[
\Delta; \Gamma \vdash b_1 : T
\]

\[
\Delta; \Gamma \vdash b_2 : T
\]

\[
\Delta; \Gamma \vdash \text{let } id = d \text{ in } b : T'
\]

\[
op \in \{-, \text{sq}\}
\]

\[
\Delta; \Gamma \vdash b : \text{BaseType}
\]

\[
\Delta; \Gamma \vdash s : \text{Shape}
\]

\[
\Delta; \Gamma \vdash \text{UnaryOp}(op, t) : \text{Tensor}(b, s)
\]

\[
op \in \{+, -, \times, /\}
\]

\[
\Delta; \Gamma \vdash t_1 : \text{Tensor}(b, s)
\]

\[
\Delta; \Gamma \vdash t_2 : \text{Tensor}(b, s)
\]

\[
\Delta; \Gamma \vdash \text{BinaryOp}(op, t_1, t_2) : \text{Tensor}(b, s)
\]

\[
\Delta; \Gamma \vdash \text{projection}(i, n) : T_n
\]

\[
\Delta; \Gamma \vdash \text{Product}(T_2, \ldots)
\]

\[
\Delta; \Gamma \vdash \text{Shape}(d_1, d_2, \ldots, d_n)
\]

Figure 6. Rules for deriving types of expressions and definitions. This includes zero members. Note that these type rules assume that all type variables are unique. Additionally, in the rule for gradient, "autodiff" is the automatic differentiation system, which we will attempt to capture the entire semantics of the transformation in the future.
All values are Tensor typed in Relay

\[ i \in \mathbb{Z} \]
\[ \Delta; \Gamma \vdash i : \text{Tensor(\text{IntType}(32), Shape())} \]  

\[ f \in \mathbb{R} \]
\[ \Delta; \Gamma \vdash f : \text{Tensor(\text{FloatType}(32), Shape())} \]  

\[ b \in \{\text{True, False}\} \]
\[ \Delta; \Gamma \vdash b : \text{Tensor(\text{BoolType}, Shape())} \]  

\[ \Delta \vdash s = \text{Shape}(d_1, d_2, \ldots, d_n) \]
\[ \Delta \vdash b : \text{BaseType} \]
\[ \Delta; \Gamma \vdash t_1 : \text{Tensor}(b, s) \]
\[ \cdots \]
\[ \Delta; \Gamma \vdash t_m : \text{Tensor}(b, s) \]
\[ \Delta; \Gamma \vdash [t_1, t_2, \ldots, t_m] : \text{Tensor}(b, \text{Shape}(m, d_1, d_2, \ldots, d_n)) \]  

\[ \Delta; \Gamma \vdash c : \text{Tensor(\text{BoolType}, Shape())} \]
\[ \Delta; \Gamma \vdash b_1 : T \]
\[ \Delta; \Gamma \vdash b_2 : T \]
\[ \Delta; \Gamma \vdash \text{if } c \text{ then } b_1 \text{ else } b_2 : T \]  

\[ \Delta \vdash t_2 \cdots \]
\[ \Delta; \Gamma \vdash \rho_n : T_n \]  

\[ \rho_n : T_1 \times T_2 \times \cdots \times T_n \]  

\[ \text{(Type-Product)} \]
\[ \Delta \vdash s \]
\[ \Delta \vdash \rho \]  

\[ \Delta; \Gamma \vdash \rho_1, \ldots, \rho_n : T_n \]  

\[ \text{(Type-Projection)} \]
\[ \Delta; \Gamma \vdash \{i\} : T_i \]  

\[ \Delta; \Gamma \vdash \text{id} : T \vdash b : T' \]  

\[ \text{(Type-Let)} \]
\[ \text{let } id = d \text{ in } b : T' \]

\[ op \in \{-, \cdot, \circ, /\} \]
\[ \Delta \vdash b : \text{BaseType} \]
\[ \Delta \vdash s : \text{Shape} \]
\[ \Delta; \Gamma \vdash t : \text{Tensor}(b, s) \]
\[ \Delta; \Gamma \vdash \text{UnaryOp}(op, t) : \text{Tensor}(b, s) \]  

\[ \text{(Type-UnaryOp)} \]
\[ op \in \{+, \cdot, \circ, /\} \]
\[ \Delta \vdash b \in \text{BaseType} \]
\[ \Delta \vdash s : \text{Shape} \]
\[ \Delta; \Gamma \vdash t_1 : \text{Tensor}(b, s) \]
\[ \Delta; \Gamma \vdash t_2 : \text{Tensor}(b, s) \]
\[ \Delta; \Gamma \vdash \text{BinaryOp}(op, t_1, t_2) : \text{Tensor}(b, s) \]  

\[ \text{(Type-NonComp-BinaryOp)} \]

Figure 6. Rules for deriving types of expressions and definitions. The zero members. Note that these type rules assume that all type variables. Additionally, in the rule for gradient, “autodiff” is the automatic differentiation instead of attempt to capture the entire semantics of the transformation in.
\[
\begin{align*}
\frac{\text{width} \in \mathbb{N}}{\Delta \vdash \text{IntType}(\text{width}) : \text{BaseType}} & \quad \text{(BASE-TYPE-T)} \\
\Delta \vdash \text{FloatType}(\text{width}) : \text{BaseType} \\
\Delta \vdash \text{UIntType}(\text{width}) : \text{BaseType} \\
\Delta \vdash \text{BoolType} : \text{BaseType} \\
\frac{d_1, d_2, \ldots, d_n \in \mathbb{N}}{\Delta \vdash \text{Shape}(d_1, d_2, \ldots, d_n) : \text{Shape}} & \quad \text{(SHAPE-T)} \\
\Delta \vdash \text{bt} : \text{BaseType} \\
\Delta \vdash \text{sh} : \text{Shape} \\
\frac{\Delta \vdash \text{Tensor}(\text{bt}, \text{sh}) : \text{Type}}{\Delta \vdash \text{Tensor}(\text{bt}, \text{sh}) : \text{Type}} & \quad \text{(TENSOR-T)} \\
\Delta \vdash T : \text{Type} \\
\Delta \vdash U : \text{Type} \\
\frac{\Delta \vdash T \rightarrow U : \text{Type}}{\Delta \vdash T \rightarrow U : \text{Type}} & \quad \text{(ARROW-T)} \\
\end{align*}
\]

\[\frac{K \in \{\text{Shape}, \text{Type}, \text{BaseType}\}}{\Delta, T : K \vdash \text{body} : \text{Type}} \quad \text{(QUANTIFIER-T)} \]

\[\frac{\Delta \vdash \text{forall} (T : K), \text{body} : \text{Type}}{\Delta \vdash \text{forall} (T : K), \text{body} : \text{Type}} \quad \text{(PRODUCT-T)} \]

\[\frac{\Delta \vdash T_1 : \text{Type}}{\Delta \vdash T_1 : \text{Type}} \quad \frac{\Delta \vdash T_2 : \text{Type}}{\Delta \vdash T_2 : \text{Type}} \quad \ldots \quad \frac{\Delta \vdash T_n : \text{Type}}{\Delta \vdash T_n : \text{Type}} \quad \text{(PRODUCT-T)} \]

\[\Delta \vdash (T_1 \times T_2 \times \cdots \times T_n) : \text{Type} \]

\[\Delta \vdash T : \text{Type} \quad \text{(REF-T)} \]

\[\Delta \vdash \text{RefType}(T) : \text{Type} \]

Figure 5. Rules for constructing types, indicating kinds. Reference types are only generated internally by reverse-mode automatic differentiation and cannot be given in frontend user code. Also note we will eventually define a more complex AST for shapes.
Figure 5. Rules for constructing types, indicating kinds. Reference types are only generated internally by reverse-mode automatic differentiation and cannot be given in frontend user code. Also note we will eventually define a more complex AST for shapes.
Figure 5. Rules for constructing types, indicating kinds. Reference types are only generated internally by reverse-mode automatic differentiation and cannot be given in frontend user code. Also note we will eventually define a more complex AST for shapes.

Tensors can only contain base types and not tensors or other functions.
Tensors can only contain base types and not tensors or other functions.

Figure 5. Rules for constructing types, indicating kinds. Reference types are only generated internally by reverse-mode automatic differentiation and cannot be given in front-end user code. Also note we will eventually define a more complex AST for shapes.
Automatic Differentiation

• Automatic differentiation as a source code transformation, exposed to users, can be invoked on arbitrary functions.

• Experimenting with different implementation strategies to address shortcomings from other attempts.

• Goal is to provide higher-order, higher order reverse mode.

```python
@relay
def grad_of(f):
    return relay.grad(f)
```
Evaluator

- An interpreter for Relay programs, implements reference semantics and runtime data structures.
- Supports just-in-time compilation of type specialized operators.
- Interoperability with Python data types, just call Relay functions like normal Python functions.

```python
a = np.zeros((1, 1))
b = np.zeros((1, 1))
x = np.ones((1, 1))
y = np.array([5])

loss = linear_loss(a, b, x, y)
```
# Register operator in env.
register_op(
    env,
    'broadcast_add',
    badd_type,
    broadcast_add_compiler)
shape = TypeId("s", ir.Kind.Shape)
in_out_type = TensorType(FloatType(32), shape)

# We take two tensors of identical shape, 
# and return one of identical shape.
input_types = [in_out_type, in_out_type]
output_type = in_out_type

# Build function type.
arrow_type = mk_arrow(
    input_types,
    output_type)

badd_type = mk_quant(
    shape,
    mk_arrow(input_types, out_type))
shape = TypeId("s", ir.Kind.Shape)
in_out_type = TensorType(FloatType(32), shape)

# We take two tensors of identical shape,
# and return one of identical shape.
input_types = [in_out_type, in_out_type]
output_type = in_out_type

# Build function type.
arrow_type = mk_arrow(
    input_types,
    output_type)

badd_type = mk_quant(
    shape,
    mk_arrow(input_types, output_type))
shape = TypeId("s", ir.Kind.Shape)
in_out_type = TensorType(FloatType(32), shape)

# We take two tensors of identical shape, 
# and return one of identical shape.
input_types = [in_out_type, in_out_type]
output_type = in_out_type

# Build function type.
arrow_type = mk_arrow(
    input_types,
    output_type)

badd_type = mk_quant( 
    shape, 
    mk_arrow(input_types, out_type))
shape = TypeId("s", ir.Kind.Shape)
in_out_type = TensorType(FloatType(32), shape)

# We take two tensors of identical shape, and return one of identical shape.
input_types = [in_out_type, in_out_type]
output_type = in_out_type

# Build function type.
arrow_type = mk_arrow(input_types, output_type)

badd_type = mk_quant(shape, mk_arrow(input_types, output_type))
forall (s : Shape),
  Tensor Float s ->
  Tensor Float s ->
  Tensor Float s

shape = TypeId("s", ir.Kind.Shape)
in_out_type = TensorType(FloatType(32), shape)
# We take two tensors of identical shape, and return one of identical shape.
input_types = [in_out_type, in_out_type]
output_type = in_out_type
# Build function type.
arrow_type = mk_arrow(input_types, output_type)
badd_type = mk_quant(shape, mk_arrow(input_types, output_type))

badd_type = mk_quant(shape, mk_arrow(input_types, output_type))
def broadcast_add_compiler(func_ty: ir.Type) -> Any:
    # Get inputs based on type, and return type.
    Inputs, ret_ty = func_ty_to_placeholders(func_ty)
    
    # Specialize add to Inputs.
    Output = topi.broadcast_add(*Inputs)
    
    schedule = tvm.create_schedule(Output.op)
    
    # Use TVM to compile operator, and return function.
    module = tvm.build(schedule, Inputs + [Output], …)
    
    return module.get_function("broadcast_add_compiler")
Ongoing & Future Work

- Runtime system
- Optimizations
- Numerical Accuracy
- Type system extensions
- Software Engineering
Runtime System

- Evaluator is a reference implementation.

- VM is intended to implement efficient allocation, reclamation, distinction between identity and resources.

- TVM has a runtime system which we are extending with new features to support Relay’s currently non-compilable features (such as control flow).
Optimizations

• We are “whole program” meaning we have control and data flow, enables dynamic networks.

• We want to port some traditional optimizations which have been challenging on current IR such as: fusion, change of layout, parallel, and distributed work scheduling.

• Implement other framework’s optimizations such as auto-batching or TensorFlow fold.
Numerical Accuracy

• ML has proven robust to rounding error

• We are eager to apply ideas from tools like Herbie and Stoke.

• Optimize for performance like Stoke, but “smart” instead of stochastic.

• “Machine oblivious” machine learning

• Try new numeric types, and hardware, quantization and more
Type system extensions

- Partially known shapes, necessary for NLP applications.

- Extend tensor types to track data layout.

- Internal effect system for reasoning about RNG, state, parameters, i/o

\[
\text{Tensor}[\text{Float}, (n, m, \text{Any})]
\]

\[
\text{Tensor}[\text{Float}, (n, m, \text{Any}), \text{Layout}]
\]

\[
A \rightarrow \text{Eff}[B]
\]
Software Engineering

- Our early prototype of Relay had a REPL, step debugger, and differential testing.

- We would like to bring these tools back, including high quality error messages, and more debugging and productivity tools like NaN isolation.
Relay as Research Vehicle

We view Relay as a new research vehicle for *exploring*:

- Differentiable programming

- A backend for new Deep Probabilistic Programming Languages (like Pyro or Edward).

- ML and synthesis guided compiler optimizations, inspired by AutoTVM, Chlorophyll, …

- Collaborations with other researchers!

- More!
This represents months of joint work with lots of great folks:
Conclusion

• Relay is a new high-level IR for TVM.

• Production quality implementation in progress.

• Near term goal is to match TVM’s existing performance, then focus on new optimizations.

• We plan to release a Relay alpha by end of the summer.

• Questions?