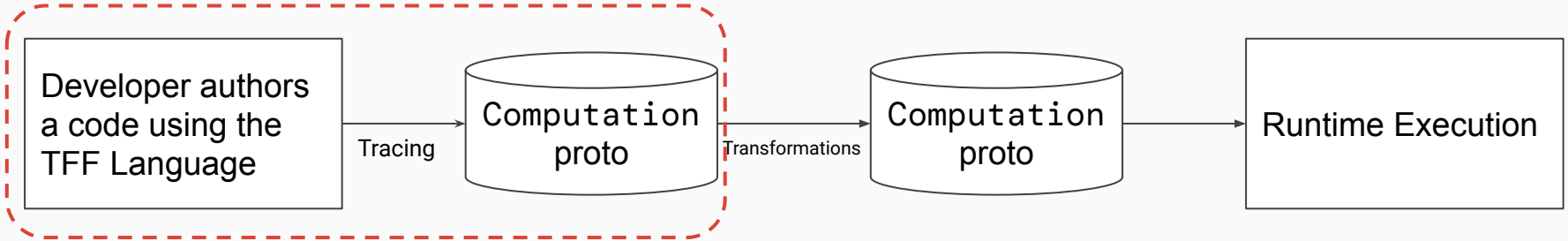
TFF 101: Lingua Federata, Tracing, and Abstract Syntax Trees

Last updated: 2022-07-15

zachgarrett@google.com, krush@google.com

TFF introduction series

- **[this talk]** TFF 101: Lingua Federated, Training, and ASTs
- TFF 102: Executors, Executor Stacks, and Execution Contexts
- TFF 103: Transformations, Compiler Pipelines, and *Forms*



Life of a TFF computation

A lingua franca for federated computation

lingua federata if you will.

What are our goals for a "Language of Federation"

From https://www.tensorflow.org/federated/federated_core

*"... **compact representation** ... **of program logic ... that's** **executable in a variety** **of target environments**.*

*The key **defining characteristic** of the kinds of algorithms that FC is designed to express is that actions of **system participants are described in a collective manner**. Thus, we tend to talk about each device **locally transforming data**, and the devices coordinating work by a centralized coordinator **broadcasting** and **aggregating** their results."*

TFF Design Principles

1. Design composable pieces that can be built-up into larger structures that perform more powerful operations.
    a. Expressible interfaces, limited implementations: prefer that a single object or function does a single thing.
2. Flexible, or even better functional, parameterizations.
    a. Extend functionality via parameterizations.

The Abstract Syntax Tree (AST)

Why even make an AST?

*"... **compact representation** ... **of program logic ... that's executable in a variety of target environments**."*

Not all environments have a Python interpreter (e.g Smartphones).

To execute in environments *without Python*, we need *something* independent of Python.

Why even make an AST?

"… compact representation … of program logic … that's executable in a variety of target environments."

To provide *the same behavior* in these environments (and avoid nasty surprises*), TFF makes the guarantee that it will **only ever execute code which is logically equivalent to what will run "in production"†.**

TFF begins with this guarantee as an absolute, unbreakable requirement; everything else TFF assumes to be addressable by enough effort.

* A common one being: 'oops, the logic I was running implicitly relied on being executed in the same python process'
† "production" means any execution platform that binds against non-simulation data for federated computation, be it cross-device, cross-silo, or cross-user.

Why even make an AST?

"... compact representation ... of program logic ... that's executable in a variety of target environments."

When TFF gives you back an object (e.g. a callable computation), it **must** already be logically equivalent to this platform-agnostic representation.

An AST is one natural method of guaranteeing this.

TFF is not Python

Writing TFF code should *not* be thought of as writing Python code. This will make life harder.

TFF is better thought of its own language, which we happen to use Python to write. Perhaps similar to using natural language to write the language of mathematics.

The TFF language contains lambda calculus, allowing for the expression of functions and applications of those functions.

```
(let
  var18=<
    federated_value_at_server(<>),
    comp#f52c47be()
  >
in (_var3 -> (let
  var1=<
    federated_apply(<
      (_var4 -> _var4.model_broadcast_state),
      _var3[0]
    >),
    federated_apply(<
      (_var6 -> _var6.model),
      _var3[0]
    >),
    federated_apply(<
      (_var12 -> _var12.delta_aggregate_state),
      _var3[0]
    >),
    federated_apply(<
      (_var20 -> _var20.optimizer_state),
      _var3[0]
    >)
  >,
  var2=federated_broadcast(var1[1]),
```

The beginning of a FedAvg computation in TFF "code" (possibly different today)

How do we represent a portable, cross-language, cross-platform AST?

Multiple ways to represent the AST

Protocol buffer

Within the system TFF serializes/deserializes ASTs using protocol buffer messages, namely the `Computation` proto.

These slides will frequently use this representation.

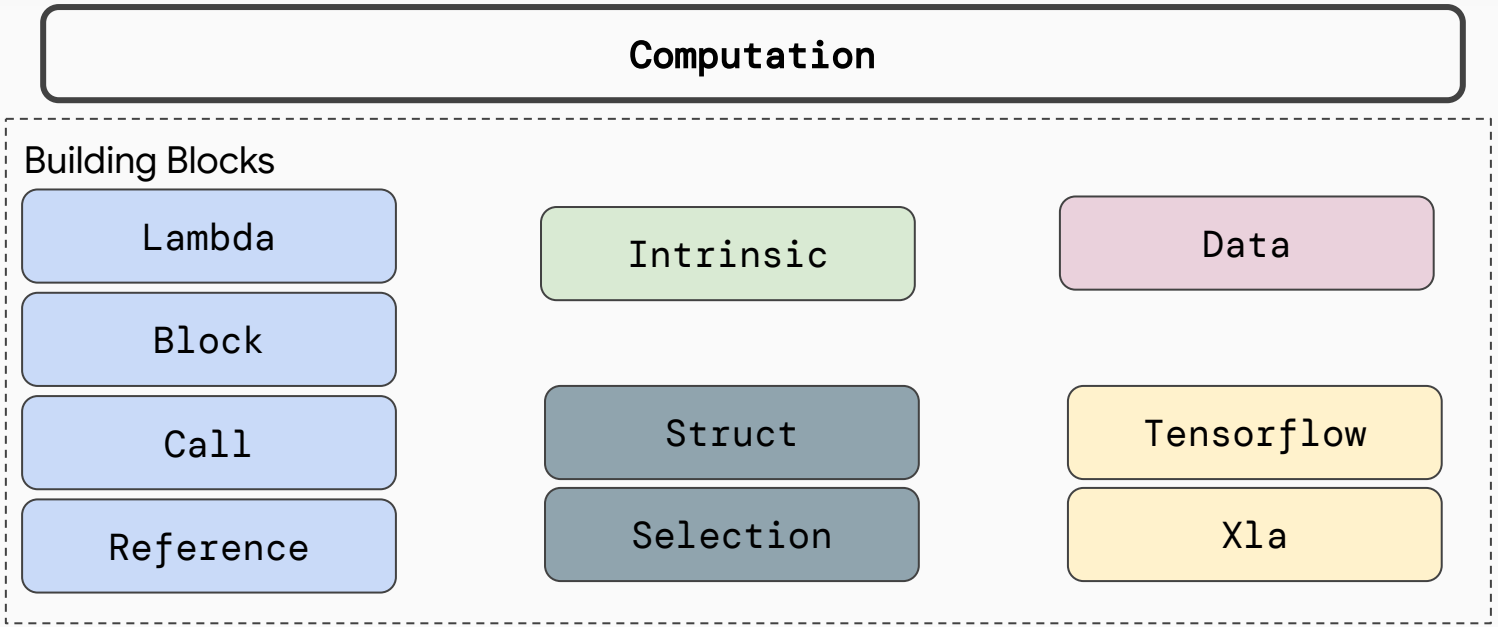
Symbolic Expressions (S-expressions)

TFF has a compact form of representing computations that uses Lisp S-Expressions.

This was demonstrated in a previous slide showing FedAvg, and is often used within the runtime for print debug statements about currently executing computations.

The `Computation` proto (TFF's AST)

https://github.com/tensorflow/federated/blob/main/tensorflow_federated/proto/v0/computation.proto



TFF Python API and the `Computation` proto

```
@tff.federated_computation(
  tff.types.at_server(tf.int32))
def simple(server_value):
  client_values = tff.federated_broadcast(
    server_value)
  client_values = tff.federated_map(
    add_one, client_values)
  server_result = tff.federated_sum(
    client_values)
  return server_result
```

```
Computation {
  type { function { … } }
  lambda {
    parameter_name: "simple_arg"
    result {
      …
      }
    }
  }
}
```

This computation is a `lambda`, which always has a `function` type signature. Lambdas are how we define computations that are parameterized.

TFF Python API and the `Computation` proto

```python
@tff.federated_computation(
  tff.types.at_server(tf.int32))
def simple(server_value):
  client_values = tff.federated_broadcast(
    server_value)
  client_values = tff.federated_map(
    add_one, client_values)
  server_result = tff.federated_sum(
    client_values)
  return server_result
```

The `lambda`'s *result* is also a computation. The result of this lambda is a `block`, which defines an *ordered sequence* of computations.

```
(
      type { … }
      block {
        local { … }
        local { … }
        local { … }
        local { … }
        result { … }
      }
    }
  }
}
```

TFF Python API and the Computation proto

```python
@tff.federated_computation(
  tff.types.at_server(tf.int32))
def simple(server_value):
  client_values = tff.federated_broadcast(
    server_value)
  client_values = tff.federated_map(
    add_one, client_values)
  server_result = tff.federated_sum(
    client_values)
  return server_result
```

```
Computation {
  Block {
    local { … }
    local { … }
    local { … }
    result { … }
  }
}
```

A Block has Locals which express partial computation graphs. Locals are assigned/bound to References which can be used to refer to the local's result in other Locals or the Block's result.

TFF Python API and the `Computation` proto

```
@tff.federated_computation(
  tff.types.at_server(tf.int32))
def simple(server_value):
  client_values = tff.federated_broadcast(
    server_value)
  client_values = tff.federated_map(
    add_one, client_values)
  server_result = tff.federated_sum(
    client_values)
  return server_result
```

```
Computation {
  type { function { … } }
  lambda {
    parameter_name: "simple_arg"
    result {
      type { … }
      block {
        local { … }
        local { … }
        local { … }
        result { … }
      }
    }
  }
}
```

TFF Python API and the Computation proto

```
@tff.federated_computation(
  tff.types.at_server(tf.int32))
def simple(server_value):
  client_values = tff.federated_broadcast(
    server_value)
  client_values = tff.federated_map(
    add_one, client_values)
  server_result = tff.federated_sum(
    client_values)
  return server_result
```

```
Computation {
  type { function { … } }
  lambda {
    parameter_name: "simple_arg"
    result {
      type { … }
      block {
        local { … }
        local { … }
        local { … }
        result { … }
      }
    }
  }
}
```

TensorFlow is a computation type

```
@tff.tf_computation(tf.int32)
def simple(value):
  return value + 1
```

```
Computation {
  type { function { … } }
  tensorflow {
    graph_def { … }
    parameter {
      tensor { tensor_name: "arg:0" }
    }
  }
}
```

Tensorflow computations define a graph, input, and output tensor bindings (values to use in a Session.run() call).

Struct and Selection: a "fun" example

```
@tff.federated_computation(
  tf.int32, tf.int32)
def combine(a, b):
  return (a, b)
```

```
Computation {
  type {
    function {
      parameter {
        struct {
          element { name: "a" value { tensor {…} } }
          element { name: "b" value { tensor {…} } }
        }
      }
      result { struct { … } }
    }
  }
  …
  }
  }
}
```

All Lambdas have only a single parameter (or None) under-the-hood. All Python parameters are packed into a single Struct going into TFF, and unpacked on the way back out. This greatly simplifies the backend. (more on this later)

Struct and Selection: a "fun" example

```
@tff.federated_computation(
  tf.int32, tf.int32)
def combine(a, b):
  return (a, b)
```

The `Selection` computation allows unpacking the elements of a struct.

```
lambda {
  parameter_name: "combine_arg"
  result {
    struct {
      element {
        selection {
          source { "combine_arg" }
          index: 0
        }
      }
      element {
        selection {
          source { "combine_arg" }
          index: 1
        }
      }
    }
  }
}
```

Struct and Selection: a "fun" example

```
@tff.federated_computation(
   t
de
```

If we look closely, this `Computation` could *almost* be simplified to avoid the selections and just pass through the argument (note the argument has names, the result does not).

```
Computation {
  type {
    function {
      parameter {
        struct {
          element { name: "a" value { tensor {…} } }
          element { name: "b" value { tensor {…} } }
        }
      }
      result { struct { … } }
    }
  }
  lambda {
    parameter_name: "combine_arg"
    result {
      struct {
        element {
          selection {
            source { "combine_arg" }
            index: 0
          }
        }
        element {
          selection {
            source { "combine_arg" }
            index: 1
          }
        }
      }
```

Tracing

Building Abstract Syntax Trees from Python

Remember that TFF is not Python?

TFF **traces** the logic, and represents the logic from the Python code as a "TFF AST" (a `Computation` proto).

- Very analogous to the behavior of `tf.function` tracing to `tf.FunctionGraph` proto.
- When we use one of TFF's decorators, the thing later invoked (that looks like a Python function) is **not** a Python function.
  - Remember, to support the goal of running **logically identical** code in your test, simulations, cross-silo and cross-device deployments, it **can't be**.

Tracing TFF computations

```
@tff.federated_computation(
  tff.types.at_server(tf.int32))
def simple(server_value):
  client_values = tff.federated_broadcast(
    server_value)
  client_values = tff.federated_map(
    add_one, client_values)
  server_result = tff.federated_sum(
    client_values)
  return server_result
```

How do we go from this to this?

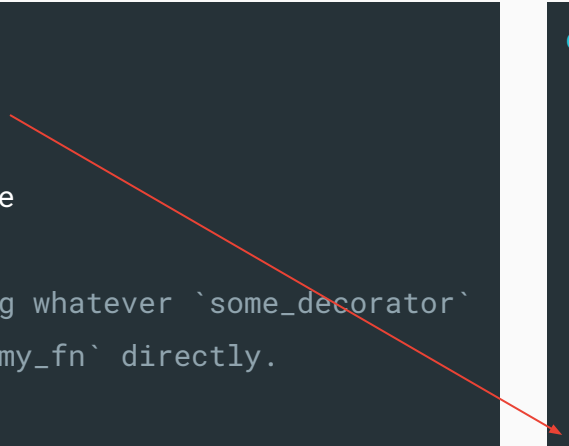
```
Computation {
  type { function { … } }
  lambda {
    parameter_name: "simple_arg"
    result {
      type { … }
      block {
        local { … }
        local { … }
        local { … }
        result { … }
      }
    }
  }
}
```

Interlude: Python decorator magic

Decorator take objects (often functions) as arguments, and return new ("wrapped") objects.

```
@some_decorator
def my_fn(value):
  # do stuff
  return new_value

# Actually calling whatever `some_decorator`
# returned, not `my_fn` directly.
my_fn(value)
```



```
def some_decorator(fn):

  def new_fn(*args, **kwargs):
    # probably do something fancy
    return fn(*args, **kwargs)

  return new_fn

my_fn = some_decorator(my_fn)
my_fn(value)
```

TFF's decorator **intercepts**!

```
@tff.federated_computation(SOME_TFF_TYPE)
def my_fn(value):
  # do stuff
  return new_value
```

(conceptually)

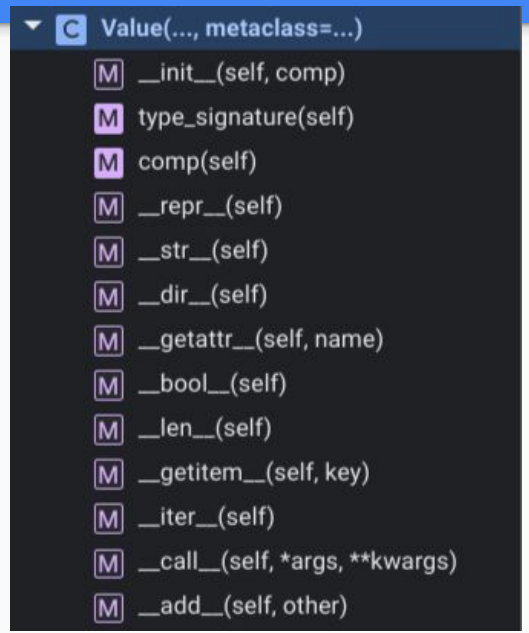
```
tracer_value = make_reference_value(SOME_TFF_TYPE)
traced_result = my_fn(tracer_value)
tff_ast = Lambda(
  param=tracer_value, result=traced_result)
# This Computation is the thing the decorated
# function becomes. 'Looks like' a Python callable,
# because it implements __call__. But not a
# 'normal' Python function!
return Computation(tff_ast)
```

Whoah! What happened here?

```
tracer_value = make_reference_value(SOME_TFF_TYPE)
traced_result = my_fn(tracer_value)
tff_ast = Lambda(param=tracer_value, result=traced_result)
return Computation(tff_ast)
```

- We know the TFF type of the argument the function is written to expect*.
- We construct a TFF Value of this type, representing a reference to the parameter of this function.
  - At the inner core, TFF computations only accept **one parameter** (next slide).
- TFF Value implements every Python dunder method we could think of (and would be reasonable to support)

All these things!



* Polymorphism's implementation is not interestingly different today

Good to know: One argument?

```
tracer_value = make_reference_value(SOME_TFF_TYPE)
traced_result = my_fn(tracer_value)
tff_ast = Lambda(param=tracer_value, result=traced_result)
return Computation(tff_ast)
```

To simplify the execution, TFF represents any function arguments as a **single** structure of arguments. We'll see this come up in TFF 102 presentation.

In Python if it looks like we have multiple arguments, at the inner levels of TFF this is represented as a `tff.Struct` for ease of passing around the values.

TFF handles all the packing of multiple arguments into a single structure, as well as unpacking the structure when handing values back to the user.

What happened here? continued

```
tracer_value = make_reference_value(SOME_TFF_TYPE)
traced_result = my_fn(tracer_value)
tff_ast = Lambda(param=tracer_value, result=traced_result)
return Computation(tff_ast)
```

So what happens when we call `my_fn`?

- Well, Python duck-types, so it **starts interpreting the body of `my_fn` with `tracer_value` argument.**
- This interpretation process **calls the dunder methods implemented by Value**.

```
@tff.federated_computation(StructType...)
def make_selection(arg):
  return arg[0]
```

is effectively

```
return arg_value.__getitem__(0) # type(arg_value) == tff.Value
```

And recall `tracer_value` has implemented `__getitem__`, and can record this interaction.

What do these `__fn__` do?

```
tracer_value = make_reference_value(SOME_TFF_TYPE)
traced_result = my_fn(tracer_value)
tff_ast = Lambda(param=tracer_value, result=traced_result)
return Computation(tff_ast)
```

Each of them **returns a new `tracer_value` (an instance of `tff.Value`).**

Python calls `a.__add__(b)` when it sees `a + b`

Make a proto representing "call *plus* on two arguments" `(x, y -> x+y)(a, b)`

Make a reference so this operation can be reused, and return a new `Value` backed by the reference.

```
def __add__(self, other):
  other = to_value(other, None)
  if not self.type_signature.is_equivalent_to(other.type_signature):
    raise TypeError('Cannot add {} and {}.'.format(self.type_signature,
                                                   other.type_signature))
  call = building_blocks.Call(
      building_blocks.Intrinsic(
          intrinsic_defs.GENERIC_PLUS.uri,
          computation_types.FunctionType(
              [self.type_signature, self.type_signature],
              self.type_signature)),
      to_value([self, other], None).comp)
  ref = _bind_computation_to_reference(call, 'adding a tff.Value')
  return Value(ref)
```

OK, but where is the AST?

```
tracer_value = make_reference_value(SOME_TFF_TYPE)
traced_result = my_fn(tracer_value)
tff_ast = Lambda(param=tracer_value, result=traced_result)
return computation(tff_ast)
```

The AST is the chain/dependency tree of `Value`s after tracing!

But you said protos? Every instance of `Value` contains an instance of the Computation proto we saw earlier.

To get at that computation proto, we only need to ask for it:

```
proto = value.comp.proto
type(proto)  # computation_pb2.Computation
```

TFF Tracing TL;DR

```
tracer_value = make_reference_value(SOME_TFF_TYPE)
traced_result = my_fn(tracer_value)
tff_ast = Lambda(param=tracer_value, result=traced_result)
return Computation(tff_ast)
```

"TFF creates a tracer object and invokes your function on the tracer object, creating a `Computation` proto in the process."

`tff.tf_computation` logic works similarly, but the tracing object is a `tf.placeholder` (instead of a `tff.Value`), and a `tf.Graph` captures the logic during tracing.

- XLA as well! Just another serialization method.
- `tf.function` and `jax.jit` all work in a very similar manner.

This invocation *defines* an AST; it is the AST defined by the invocation that TFF hands back to you, the user.

What did we learn?

TFF: the big ideas in the lingua federata

- TFF's goal is to be a cross-platform language for federated computation.
  - "Cross-platform" means the Python environment that created the computation is **not available**.

- TFF represents the logic of the computation using the `Computation` protocol buffer message which is created by tracing Python functions using "intercepter" arguments.

- The `Computation` protocol buffer describes an Abstract Syntax Tree built up of *building blocks* that embeds a lambda calculus that enables defining a (very) large space of computations.

- Once we create a `Computation`, how do we execute it? Check out TFF 102.