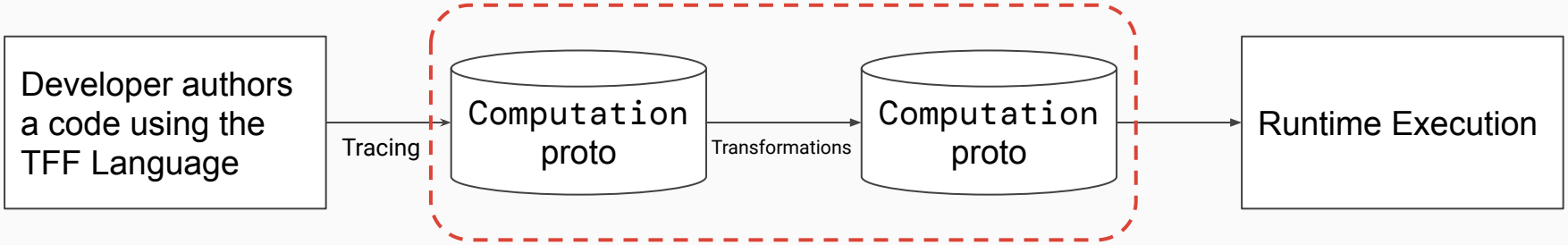
TFF 103: Types, Transformations, and *Forms*

Last updated: 2022-07-29

zachgarrett@google.com, krush@google.com

TFF introduction series

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



Life of a TFF computation

Recall: 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.

TFF type system

Can we statically (ahead of time) reason about valid federated computations?

Why have a type system?

Type systems are formal logical systems used to dictate the space of valid programs.

Type systems enable tooling to assert predicates and statically reason about properties of the computation.

- Enables TFF to find program errors early in development; the type system aides program authors in writing *correct* programs.
- Help developers of TFF to *reason about extensions*; types provide a simple system that ensures coverage of (well-typed) TFF computations.

TFF types that may already be familiar

TensorType
Represents scalars and n-D arrays (e.g. TensorFlow tensors). Has a *shape* and a *dtype* properties.
Shorthand: `str[?], int32, float64[10, 20]`

FunctionType
A type for parameterized functions (lambdas). Composes a *parameter* and *result* type.
Shorthand: `(X -> Y)`

StructType
A container type similar to both `tuple` and `dict` (but not truly either). Has a *children* property for the types of its elements.
Shorthand: `<X,Y>`

SequenceType
A type for a iterable stream of values. Contains a *element* type.
Shorthand: `X*`

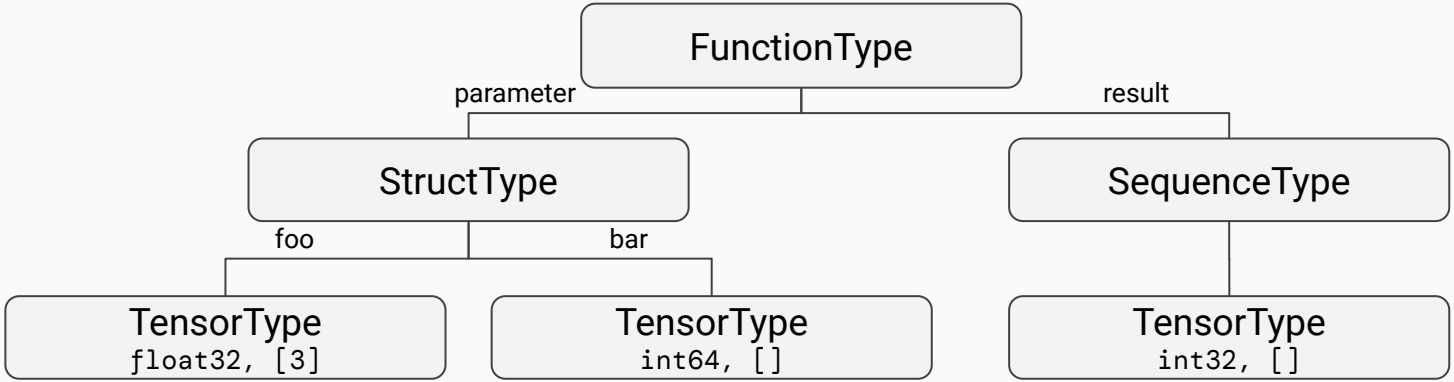
AbstractType
A placeholder for any type. Often used when talking about composite types (Function, Struct, Sequence) in general.
Shorthand: `single uppercase letter (e.g. X)`

Remember TFF Design Principle #1: Composition

Types do not *inherit*, they **compose**!

```
FunctionType(
  parameter=StructType(
    foo=TensorType(float32, [5])
    bar=TensorType(int64, [])),
  result=SequenceType(
    element=TensorType(int32, []))
```

```
(<foo=float32[5],bar=int64> -> int32*)
```



Types that are unique to TFF

FederatedType
Represents *placed* data (data at a particular location in the system). Has a *member* (the inner type), *placement*, and a boolean *all_equal* property.

Today, only `CLIENTS` and `SERVER` placement exist, but the language does have extension points for more.

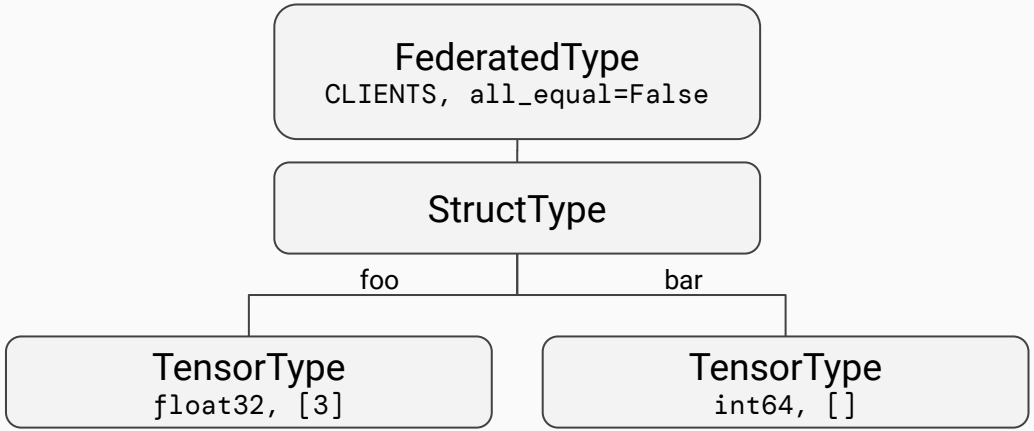
Shorthand:

```
X@CLIENTS (all_equal=True)
{X}@CLIENTS (all_equal=False)
Y@SERVER
```

Example: Composition with FederatedType

```
FederatedType(
  member=StructType(
    foo=TensorType(float32, [5])
    bar=TensorType(int64, [])),
  placement=CLIENTS,
  all_equal=False)
```

```
{<foo=float32[5],bar=int64>}@CLIENTS
```



All intrinsics are typed

Intrinsics are typed, functional building blocks that usually are implemented by the platform.

The type definitions of all intrinsics can be found at:
https://github.com/tensorflow/federated/blob/main/tensorflow_federated/python/core/impl/compiler/intrinsic_defs.py

A few notable examples:

`federated_broadcast`	`(T@S → T@C)`
`federated_map`	`(<(T→U),{T}@C> → {U}@C)`
`federated_sum`	`(<{V}@C> → V@S)`
`federated_secure_sum`	`(<{V}@C, M> → V@S)`
`federated_aggregate`	`(<T@C, U, (<U,T>→U), (<U,U>→U), (U→R)> → R@S)`
`federated_select`	`(<{K}@C, int32@S, T@S, (<T, int32> → U)> → U*@C)`

A few rules to remember on allowed composite types

Structures-of-federated values at the same placement and federated-structure-of-values are ***not*** the same type in TFF's type system! (Though frequently can be considered isomorphic and can be converted).

- `<X@CLIENTS,Y@CLIENTS> != <X,Y>@CLIENTS`
- Selections on `FederatedType` applies the placement to all children.
  - ```
    A = <X,<Y,Z>>@CLIENTS
    A[0] # X@CLIENTS
    A[1] # <Y,Z>@CLIENTS
    ```

Only `StructType`, `SequenceType`, and `TensorType` are allowed underneath a `FederatedType` (no federated functions, or federated-federated types).

- The type tree *cannot* nest placements
  ✗ `<X@CLIENTS,Y>@SERVER`
- "Only *data* is federated"
  ✗ `(X -> Y)@CLIENTS`
  ✔ `(X@CLIENTS -> Y@CLIENTS)`
    ```

A few rules to remember on allowed composite types

`StructType` without names can be used in places `StructType` (with the same structure) has names, but not the other way around.

```
a = <X, Y>
b = <foo=X, bar=Y>

a.is_assignable_from(b)  # False
b.is_assignable_from(a)  # True
```

Okay:

```
@tff.tf_computation(b)
def foo(...):
 …

foo(a)  # Can add names.
```

Not Okay:

```
@tff.tf_computation(a)
def foo(…):
 …

foo(b)  # Can't drop names
```

Why do we have these types again?

To help humans write federated computations free of errors!

To enable TFF to analyze for some kinds of mistakes sooner, without needing to wait until runtime.

```
@tff.federated_computation(
  tff.types.at_server(tf.int32))
def foo(value):
  summed_value = tff.federated_sum(round_num)
  return summed_value
```

Oops! Can't aggregate a value already on the server!

```
@tff.federated_computation(
  tff.types.at_server(tf.int32),
  tff.types.at_clients(
    tff.types.SequenceType(tf.int32)))
def foo(value, datasets):
  return tff.federated_map(
    count_examples, (value, datasets))
```

Oops! Can't map a function to mixed placements, must communicate first!

Transforming the AST

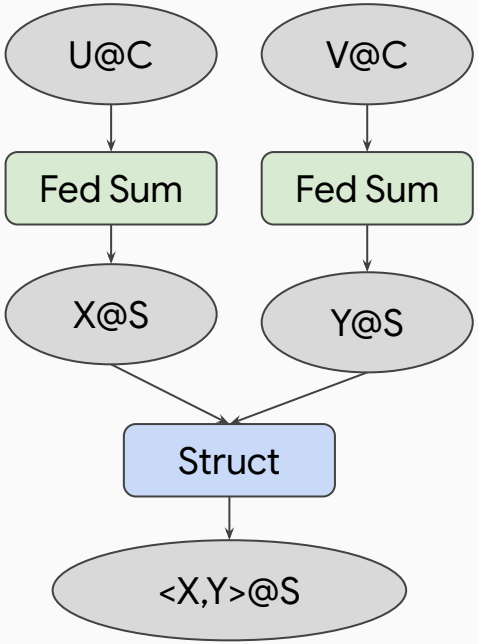
Why do we want transformations?

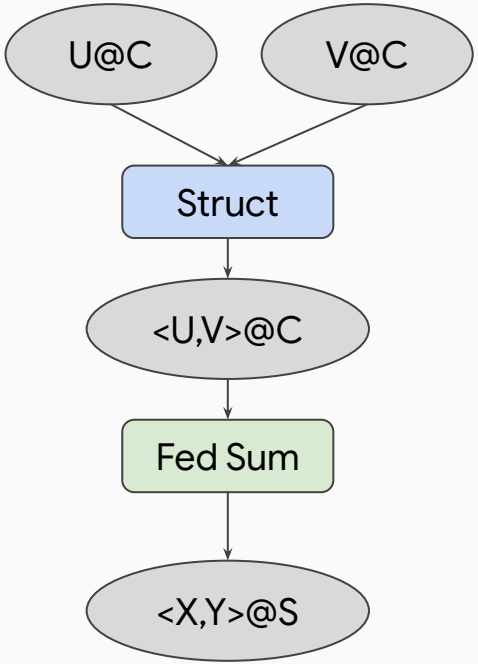
Transformations provide guarantees (invariants) about the output `tff.Computation`.

By enforcing invariants we can reduce the complexity of the execution stacks (remember TFF 102); this is a general trade-off.

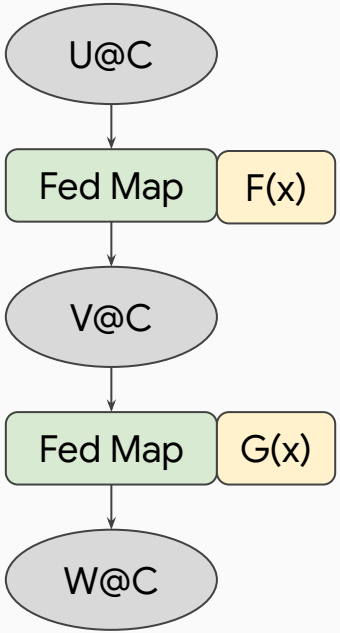
This bridges the gap between TFF computation author's flexibility and the execution stack's complexity.

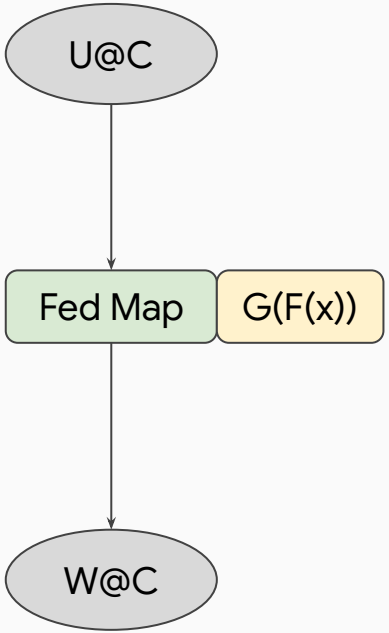
What kinds of things can you do with the AST? (What is a "transformation")



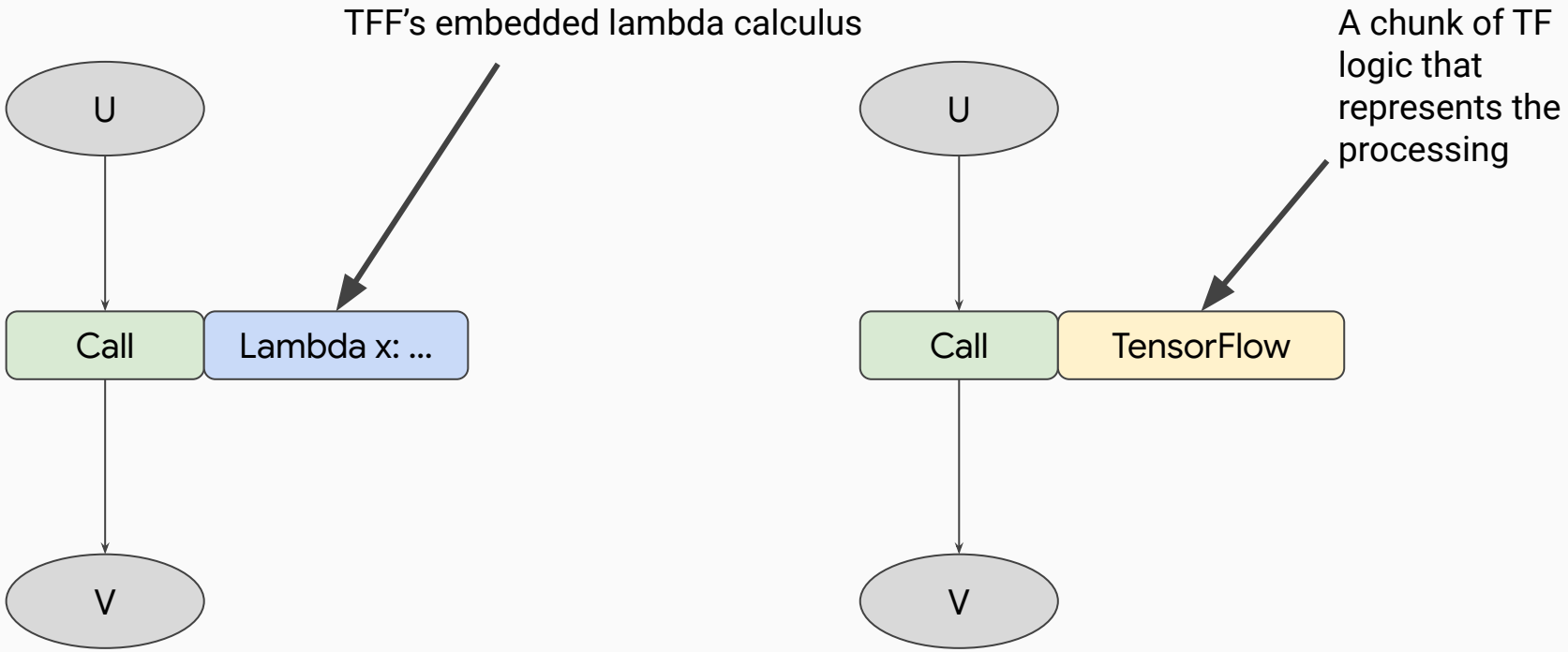


What kinds of things can you do with the AST? (What is a "transformation")





What kinds of things can you do with the AST? (What is a "transformation")

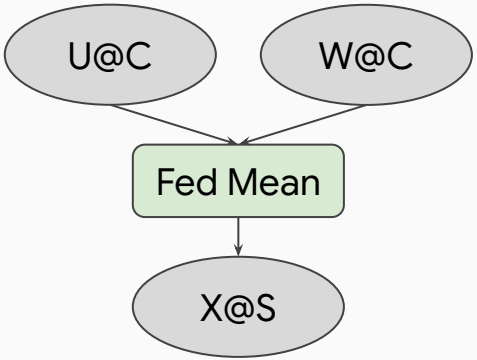


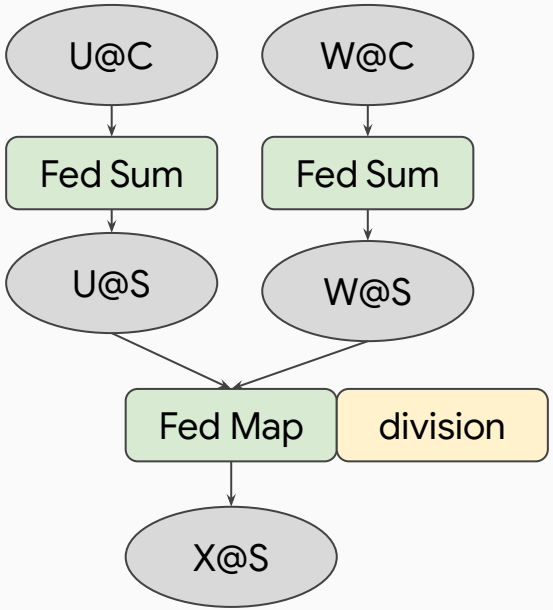
Example transformation: Lowering

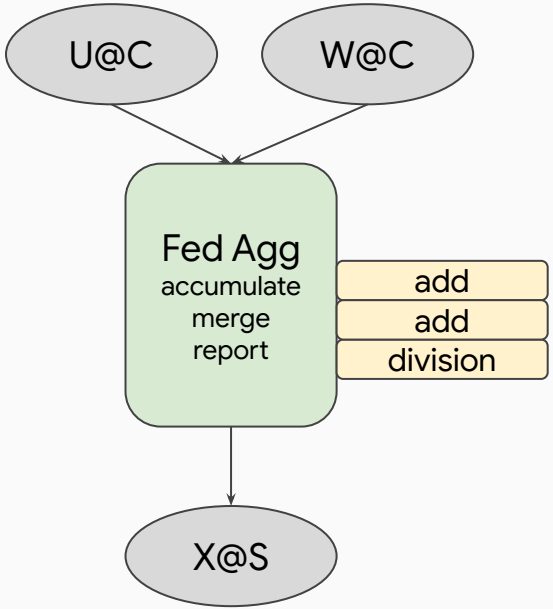
By "lowering" we mean replace "complex" (high level) operations with (often multiple) more primitive (low level) operations.

> 👍 This reduces the size of the instruction (intrinsic) set that a runtime must support.
>
> 👎 This process is difficult to reverse, and can make reasoning about properties of the AST very difficult.

Example: Lowering `federated_mean`







Forms

What do we mean when we say *Forms*?

We really mean **invariants**.

Specifically **structural invariants**, which give us **stronger** capacity to reason about "how things are executed" than we can find by only inspecting the type system.

Why do we want structural invariants?

We often want to ‘read’ some higher-level ‘fact’ from a computation.

- Example: “How many times does placement change?”

We will swap this question for a ‘simpler’ question:
How many aggregations are present in this computation?

- Next we will see that this already contains a lot of subtlety.

Q: how many aggregations are in this computation?

```
1 @tff.federated_computation(tff.type_at_clients(tf.int32))
2 def return_one(arg):
3   val = tff.federated_sum(arg)
4
5   @tff.federated_computation(val.type_signature)
6   def tuple_with_1(x):
7     return [x, 1]
8
9   t = tuple_with_1(val)
10  return t[1]
```

A: It depends

It depends on what the platform guarantees.

Runtime	# executed aggregations
TFF, no CDF	1
TFF w/ CDF	0

Note: the number of aggregations is about *how* the result is computed. *What* result is computed *must be* the same across all runtimes; the runtime that differs violates the language contract.

How can I wrap my brain around this?

Certain properties one might want to ‘read’ out of a TFF computation are, in general, *not* well-specified by the computation’s logic.

- That is, we can ‘rearrange’ the computation in logically equivalent ways and change these properties.

Limiting this situation is the purpose of structural invariants.

- If we define ‘the correct structural invariants, the logic of the computation becomes sufficient to specify properties that downstream systems may want to ‘read out’ directly, and rely on.

Call-Dominant-Form

A TFF computation in call-dominant form has two* important properties:

1. Every intrinsic which will be invoked to execute the computation appears as a top-level let binding (modulo an encapsulating global lambda).
2. Each of these intrinsics is depended upon by the computation output.
    - Note: since we view TF invocations as black boxes, all of their results depend on all of their inputs.

Together, these two properties (plus the TFF typesystem) allow us to reason about many higher-level properties of a computation.

Call-Dominant-Form

TLDR: the communication operators (plus some more stuff) are extracted to form a ‘spine’ of the computation.

```
_var1=federated_eval_at_server(comp#1c4fc665),
_var3=federated_value_at_server(<>),
_var5=federated_value_at_server(<>),
_var42=federated_eval_at_server(comp#69c99fcc),
_var43=federated_value_at_server(<>),
_var44=federated_zip_at_server(<
  _var42,
  _var43
>),
_var45=federated_eval_at_server(comp#ea88a151),
_var46=federated_value_at_server(<>),
_var47=federated_zip_at_server(<
  _var45,
  _var46
>),
```

(partial) ‘spine’ of an initialization computation

Assume we guarantee the generated AST will be in CDF. Now how many aggregations?

```
1 @tff.federated_computation(tff.type_at_clients(tf.int32))
2 def return_one(arg):
3   val = tff.federated_sum(arg)
4
5   @tff.federated_computation(val.type_signature)
6   def tuple_with_1(x):
7     return [x, 1]
8
9   t = tuple_with_1(val)
10  return t[1]
```

A: Zero

Why?

Recall:

1. Every intrinsic which will be invoked to execute the computation appears as a top-level let binding.
2. Each of these intrinsics is depended upon by the output.

Since the only dependency between the output and the aggregation is through TFF's tupling/untupling, CDF makes the guarantee that this aggregation will be **eliminated** (optimized away).

Indeed, the generated CDF literally just ‘returns 1’

```
(return_one_arg -> (let
  _var3=comp#b017312c()
 in _var3))
```

Assume we guarantee the generated AST will be in CDF. Now how many aggregations?

```
@tff.federated_computation(tff.type_at_clients(tf.int32))
def return_one_with_black_box(arg):
  val = tff.federated_sum(arg)

  @tff.tf_computation(tf.int32)
  def tuple_with_1(x):
    return [x, 1]

  t = tff.federated_map(tuple_with_1, val)
  return t[1]
```

A: One

Why?

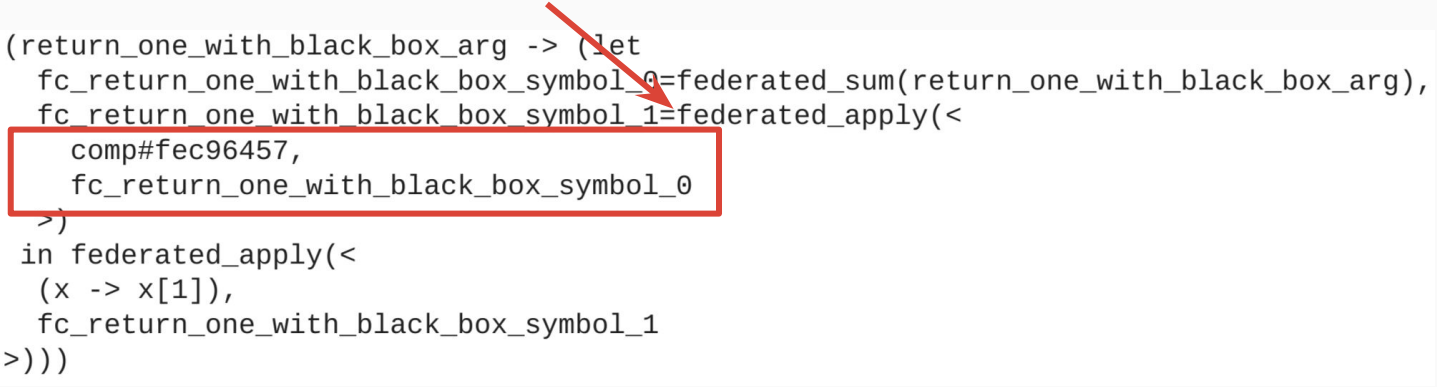
Recall:

1. Every intrinsic which will be invoked to execute the computation appears as a top-level let binding.
2. Each of these intrinsics is depended upon by the output.

But we consider the result of every TF function invocation to depend on all the inputs, to avoid inspecting data flow inside TF graphs. So CDF **cannot** eliminate this aggregation because the true dependency chain is opaque.

Generated CDF

TFF can’t “see through” or “destructure” this function call;
the output now depends on the aggregation!



```
(return_one_with_black_box_arg -> (let
  fc_return_one_with_black_box_symbol_0=federated_sum(return_one_with_black_box_arg),
  fc_return_one_with_black_box_symbol_1=federated_apply(<
    comp#fec96457,
    fc_return_one_with_black_box_symbol_0
  >)
 in federated_apply(<
  (x -> x[1]),
  fc_return_one_with_black_box_symbol_1
>)))
```

Motivation for MapReduceForm

CDF is strong enough to let us reason about many properties we want–but not quite as strong as some systems need.

- For example, CDF can express multiple back-and-forths between a single logical cohort, and so can represent computations which cannot go to single "Map followed by Reduce" restricted forms.

We use another data structure expresses these things: MapReduceForm.

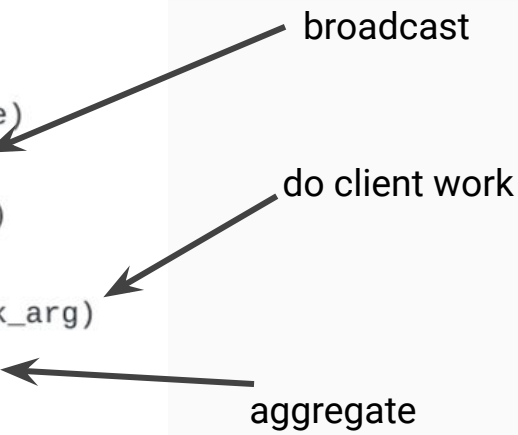
MapReduceForm

- MapReduceForm (MRF) is essentially "chunks of TensorFlow" with some specified meaning connected by communication that is implied by the data structure. MRF can represent a single round of federated learning in a MapReduce-like system.

- It is the job of TFF’s MapReduce compiler to generate these chunks from any TFF computation which can be represented in this way (or raise an error if it can't).

Logic of MRF, alternative representation

```
@federated_computation.federated_computation(next_parameter_type)
def next_computation(arg):
  """The logic of a single MapReduce processing round."""
  server_state, client_data = arg
  broadcast_input = intrinsics.federated_map(mrf.prepare, server_state)
  broadcast_result = intrinsics.federated_broadcast(broadcast_input)
  work_arg = intrinsics.federated_zip([client_data, broadcast_result])
  (aggregate_input, secure_sum_bitwidth_input, secure_sum_input,
   secure_modular_sum_input) = intrinsics.federated_map(mrf.work, work_arg)
  aggregate_result = intrinsics.federated_aggregate(aggregate_input,
                                                    mrf.zero(),
                                                    mrf.accumulate, mrf.merge,
                                                    mrf.report)
```



Logic of MRF, alt representation (con't)

…later, after doing SecAgg stuff

```
update_arg = intrinsics.federated_zip(
    (server_state, (aggregate_result, secure_sum_bitwidth_result,
                    secure_sum_result, secure_modular_sum_result)))
updated_server_state, server_output = intrinsics.federated_map(
    mrf.update, update_arg)
return updated_server_state, server_output
```

finalize results (e.g., perform server optimizer step)

Return the result

Limitations of MRF

- MRF can only represent certain intrinsics; these are hardcoded into the data structure itself (today: federated_aggregate and 3 variants of secure summing).
  - Adding a new intrinsic requires literally extending the data structure each time.
- MRF is in some sense a ‘reduced instruction-set’ data structure; the TFF compiler necessarily ‘throws information away’ in generating one.
  - E.g., all of TFF’s lambda calculus is ‘lowered’ to TF, all simple aggregations are lowered to `federated_aggregate`, etc.
- More general extensions of MRF are highly unclear
  - E.g., for a system which can express multiple back-and-forths, or which don’t naturally break down into phases without parallelism.

Other forms

- Other forms exist (e.g. MergeableCompForm), which make different guarantees about the way they can be processed.

- Remember there is an interesting and subtle tradeoff here; one could imagine computing these guarantees inside the runtime rather than in the compiler. This is a general tradeoff, one TFF must constantly consider and balance.
  - An example of the flexible parameterizations design principle of TFF.

Wrapping up: What did we learn?

TFF Types and Forms: the main takeaways

- The type system is your friend 🫂: it helps ensure correctness/soundness of federated computations!
- In TFF data is federated, computation/logic is not.
- Just like most programming languages, in TFF work in the runtime can be traded for work in the compiler via transformations.
- "Forms" are sets of invariants that constrain the interpretation of a `Computation`, and help us reason about high-level properties.