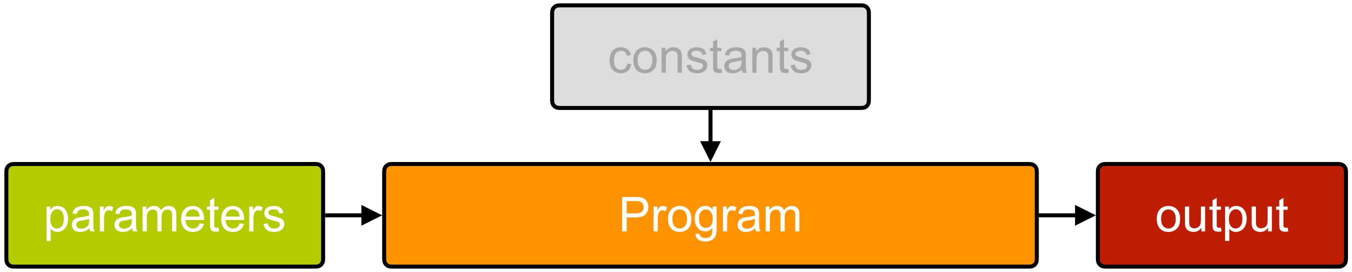
Distributed Gradients

Lukas Heinrich

Differentiable Programming



$$f : \mathbb{R}^m \rightarrow \mathbb{R}^n$$

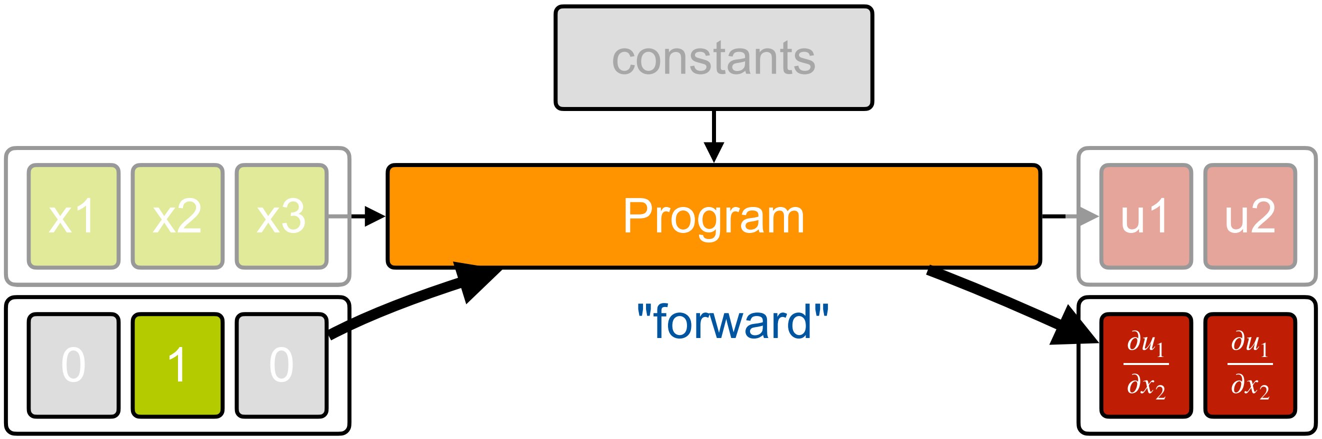
differentiable programming:

- **enable not only evaluation of the program** $f(x)$
- **but also efficient computation of gradients / jacobians**

$$J_{ij} = \begin{bmatrix} \frac{\partial u_1}{\partial x_1} & \frac{\partial u_1}{\partial x_2} & \frac{\partial u_1}{\partial x_3} \\ \frac{\partial u_2}{\partial x_1} & \frac{\partial u_2}{\partial x_2} & \frac{\partial u_2}{\partial x_3} \end{bmatrix}$$

Auto-Diff: computes jacobian either row or column-wise

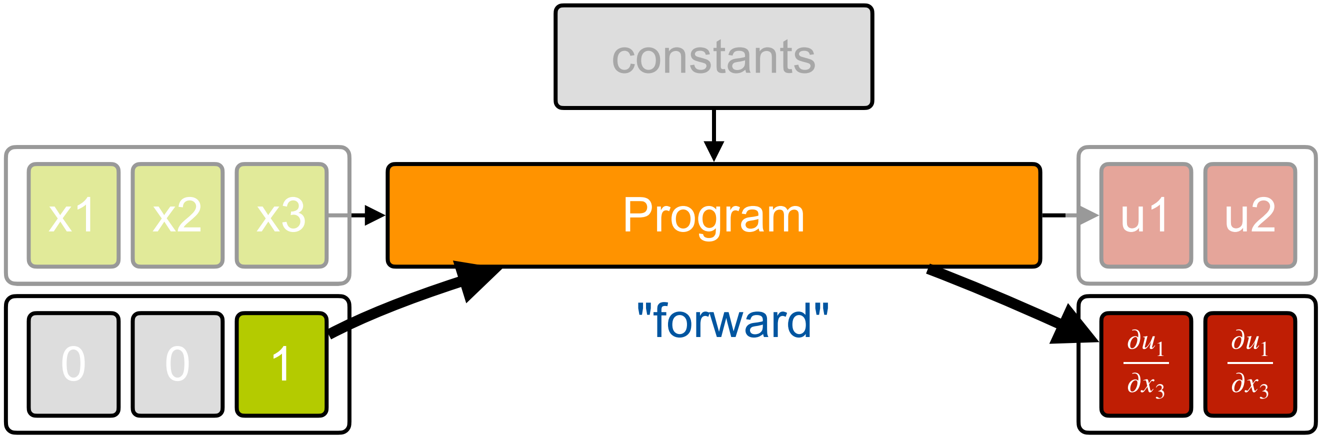
- execution of program requires double to storage for each input and output
- pass special inputs either at beginning or end



$$J_{ij} = \begin{bmatrix} \frac{\partial u_1}{\partial x_1} & \frac{\partial u_1}{\partial x_2} & \frac{\partial u_1}{\partial x_3} \\ \frac{\partial u_2}{\partial x_1} & \frac{\partial u_2}{\partial x_2} & \frac{\partial u_2}{\partial x_3} \end{bmatrix}$$

Auto-Diff: computes jacobian either row or column-wise

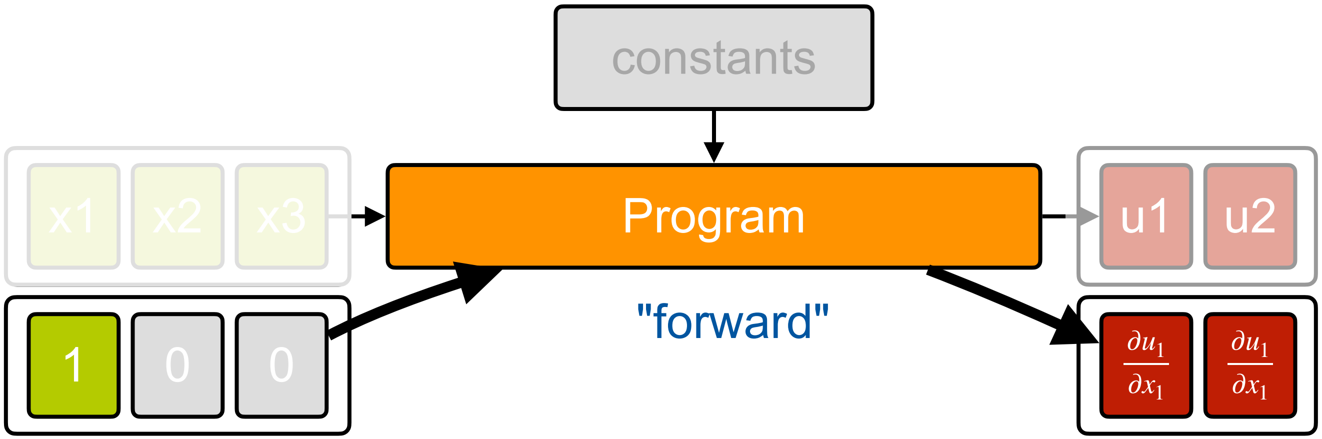
- execution of program requires double to storage for each input and output
- pass special inputs either at beginning or end



$$J_{ij} = \begin{bmatrix} \frac{\partial u_1}{\partial x_1} & \frac{\partial u_1}{\partial x_2} & \frac{\partial u_1}{\partial x_3} \\ \frac{\partial u_2}{\partial x_1} & \frac{\partial u_2}{\partial x_2} & \frac{\partial u_2}{\partial x_3} \end{bmatrix}$$

Auto-Diff: computes jacobian either row or column-wise

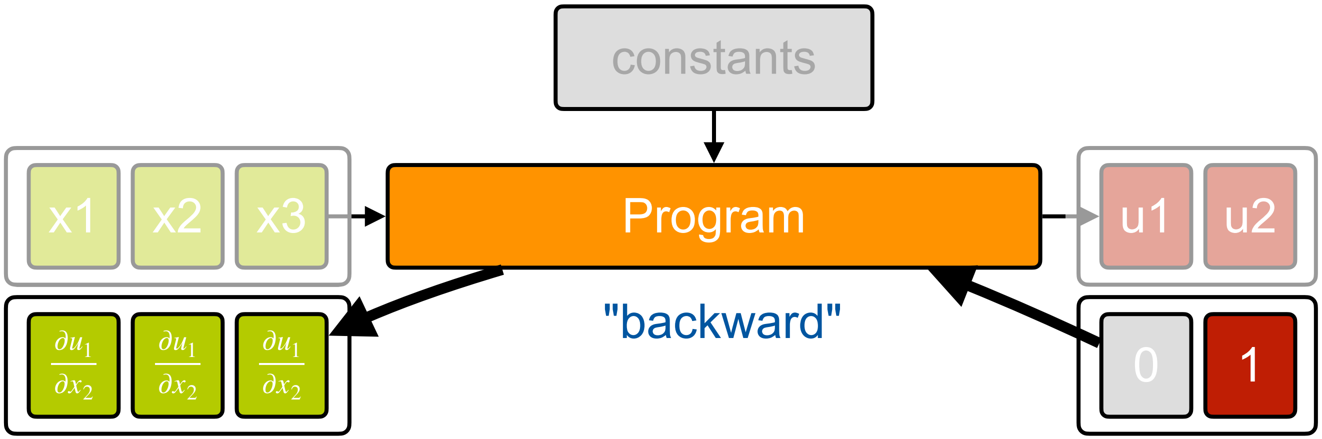
- execution of program requires double to storage for each input and output
- pass special inputs either at beginning or end



$$J_{ij} = \begin{bmatrix} \frac{\partial u_1}{\partial x_1} & \frac{\partial u_1}{\partial x_2} & \frac{\partial u_1}{\partial x_3} \\ \frac{\partial u_2}{\partial x_1} & \frac{\partial u_2}{\partial x_2} & \frac{\partial u_2}{\partial x_3} \end{bmatrix}$$

Auto-Diff: computes jacobian either row or column-wise

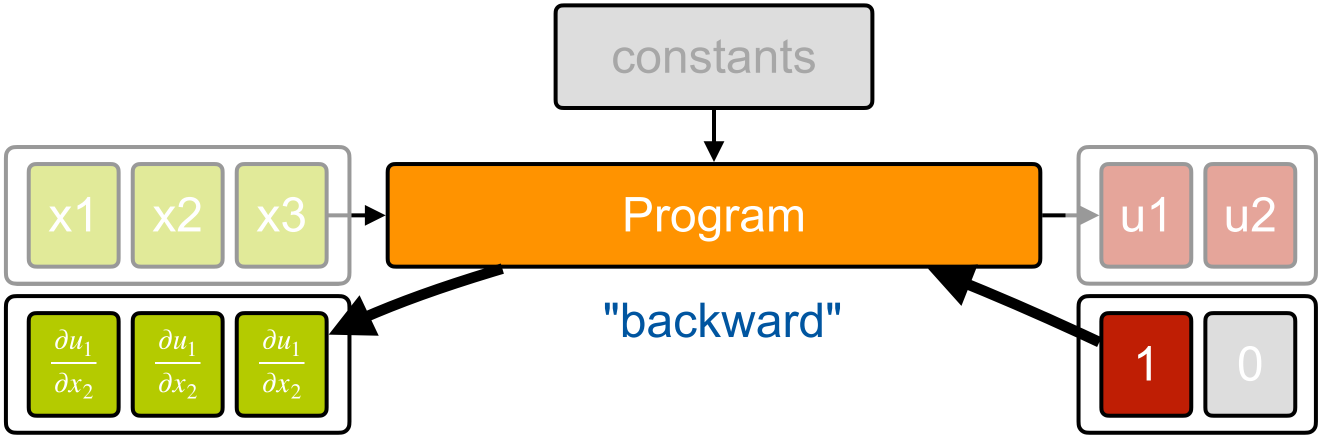
- execution of program requires double to storage for each input and output
- pass special inputs either at beginning or end



$$J_{ij} = \begin{bmatrix} \frac{\partial u_1}{\partial x_1} & \frac{\partial u_1}{\partial x_2} & \frac{\partial u_1}{\partial x_3} \\ \frac{\partial u_2}{\partial x_1} & \frac{\partial u_2}{\partial x_2} & \frac{\partial u_2}{\partial x_3} \end{bmatrix}$$

Auto-Diff: computes jacobian either row or column-wise

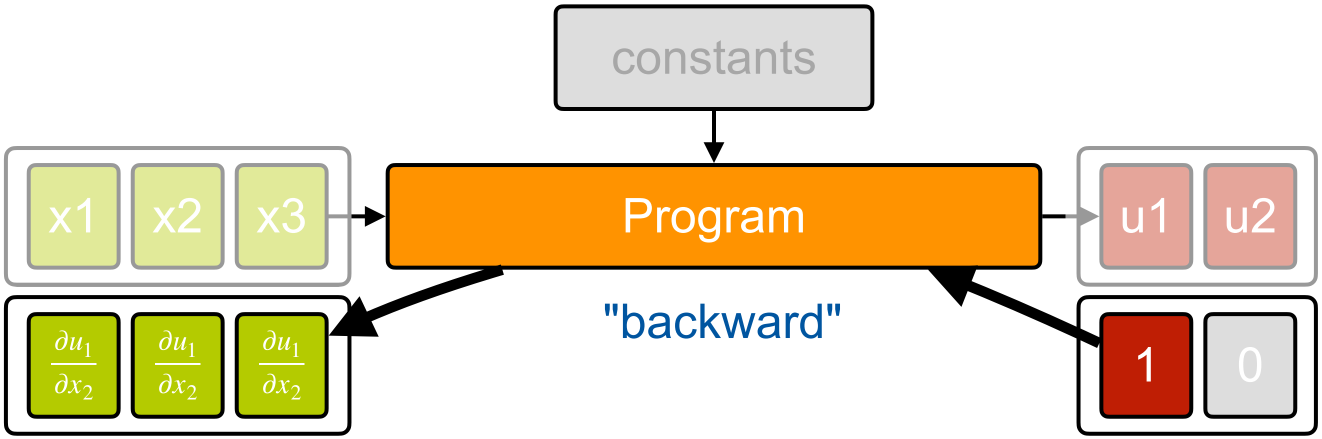
- execution of program requires double to storage for each input and output
- pass special inputs either at beginning or end



$$J_{ij} = \begin{bmatrix} \frac{\partial u_1}{\partial x_1} & \frac{\partial u_1}{\partial x_2} & \frac{\partial u_1}{\partial x_3} \\ \frac{\partial u_2}{\partial x_1} & \frac{\partial u_2}{\partial x_2} & \frac{\partial u_2}{\partial x_3} \end{bmatrix}$$

Auto-Diff: computes jacobian either row or column-wise

- **execution of program requires double to storage for each input and output**
- **pass special inputs either at beginning or end**

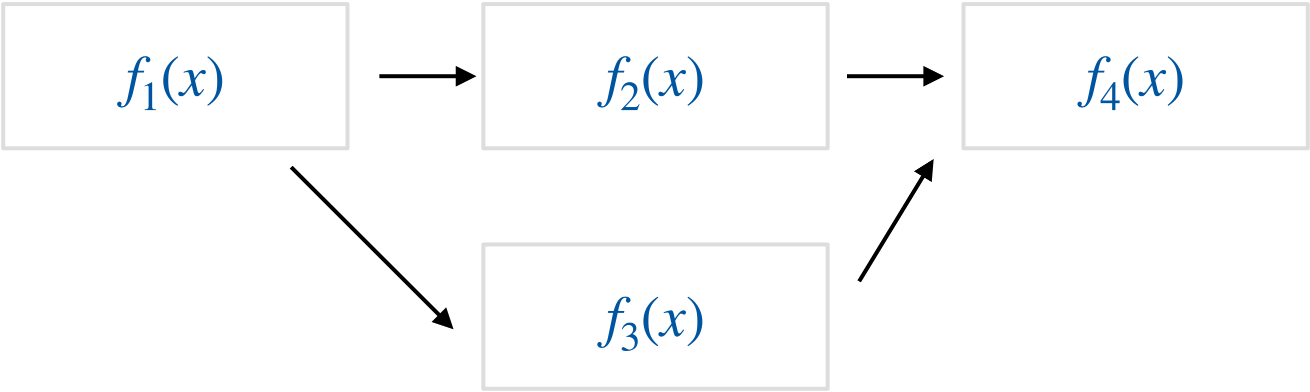


Which path
is better depends
on dimensions of
inputs / outputs

$$J_{ij} = \begin{bmatrix} \frac{\partial u_1}{\partial x_1} & \frac{\partial u_1}{\partial x_2} & \frac{\partial u_1}{\partial x_3} \\ \frac{\partial u_2}{\partial x_1} & \frac{\partial u_2}{\partial x_2} & \frac{\partial u_2}{\partial x_3} \end{bmatrix}$$

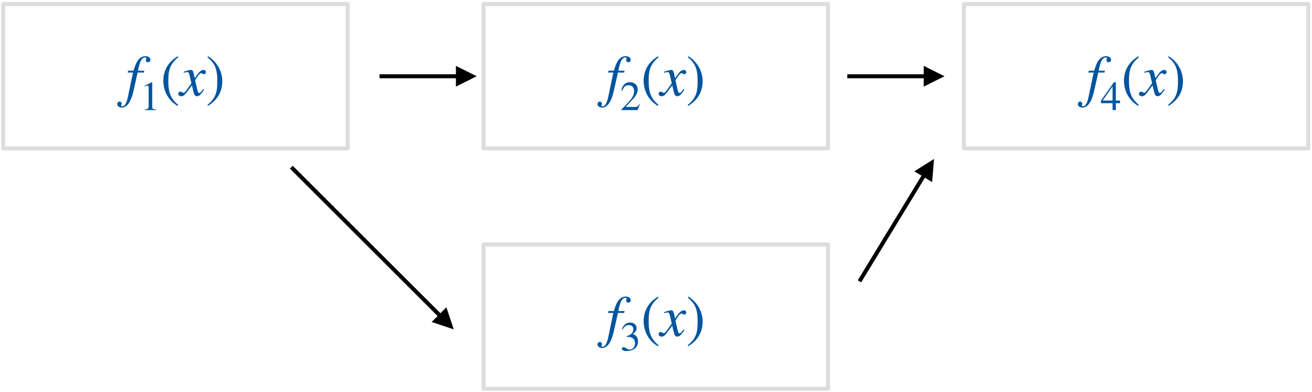
For HEP we have these challenges

- our computations are very complex chains (e.g. O(minutes/per event) vs O(µs)
- not implement(able) in a single AD framework
  - millions LOC of existing C++
- asynchronous, multi step procedure.
- needs distributed computing



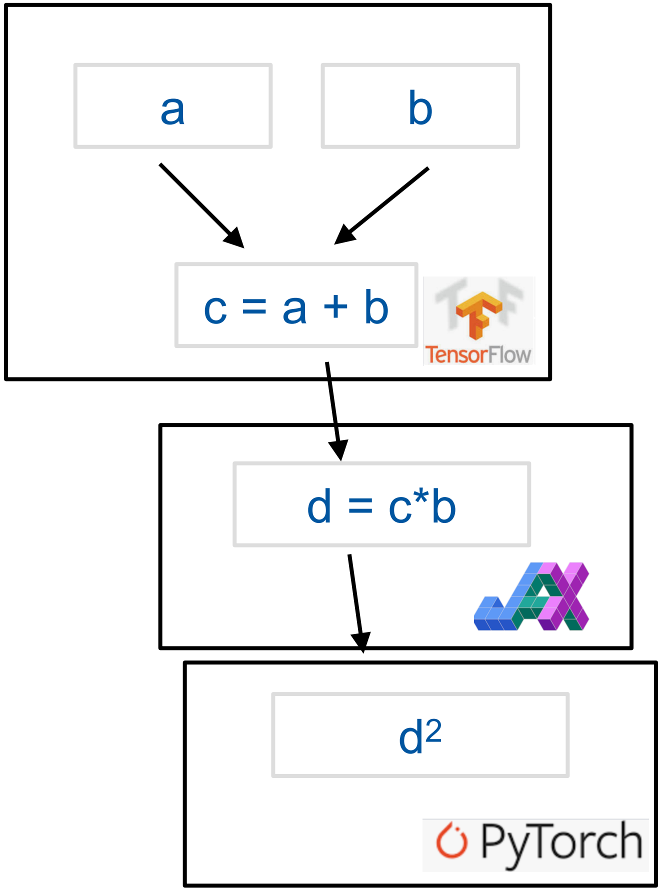
Needed infrastructure:

- workflow to chain functions (perhaps function-as-a-service)
  - track provenance of full workflow (parent child)
- ability to register "closures" for backward functions
- individual functions must be differentiable but infrastructure can be completely agnostic to which frameworks are used

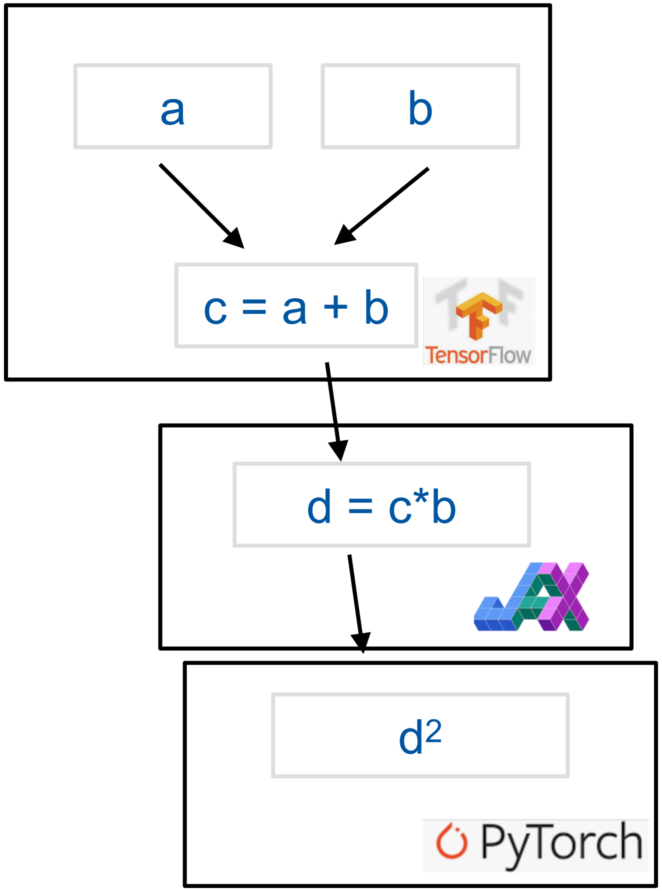


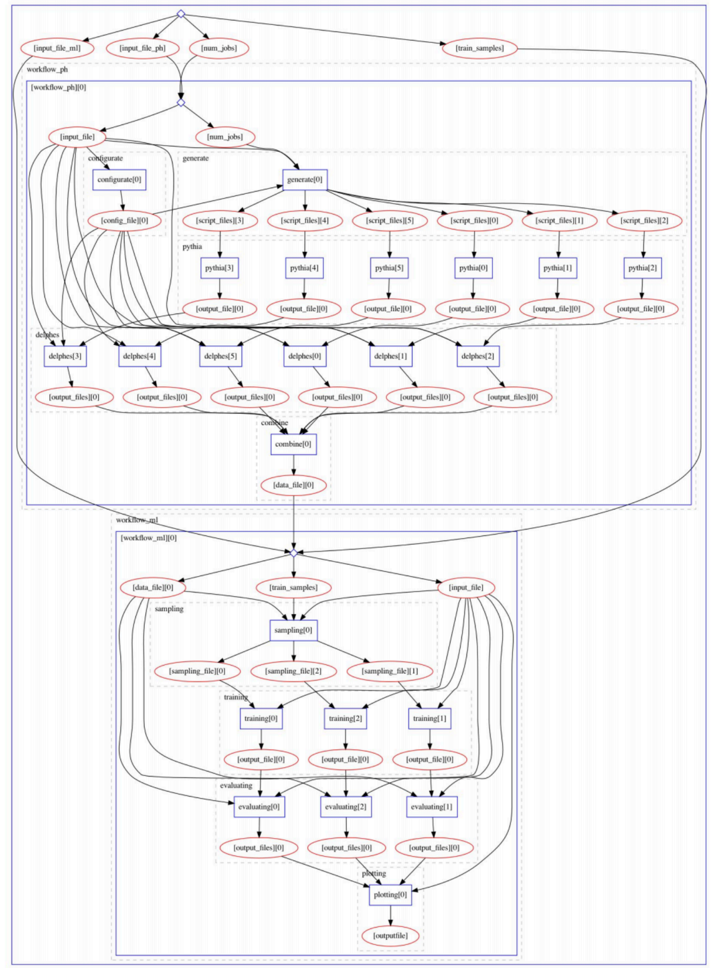
Prototype: differentiating through PyTorch + JAX + Tensorflow using functions as a service

```
In [185]: @register_function('step1')
          @instrument_torch
          def torch_function(x):
              a,b = x
              return torch.add(a,b),b

          @register_function('step2')
          @instrument_jax
          def jax_function(a,b):
              return jax.numpy.multiply(a,b)

          @register_function('step3')
          @instrument_tf
          def tensorflow_function(a):
              return tf.pow(a,2)

In [188]: def forward(x):
              (r1a, r1b) = funcx['step1']('step1_bwd',x)     ##run through torch
              r2 = funcx['step2']('step2_bwd',r1a, r1b)      ##run through jax
              r3 = funcx['step3']('step3_bwd',r2)            ##run through TF

              return r3

          def backward(adj_in_np, jvp_funcs):
              ## backprop through TF
              adj_in_np = funcx['step3_bwd'](adj_in_np)
              adj_in_np = funcx['step2_bwd'](adj_in_np)
              adj_in_np = funcx['step1_bwd'](adj_in_np)
              return adj_in_np

In [189]: result = forward([1,2])
          gradients = backward((1.0,), jvp_funcs)
          print(result, gradients)

          36.0 (array([24., 60.], dtype=float32),)
```

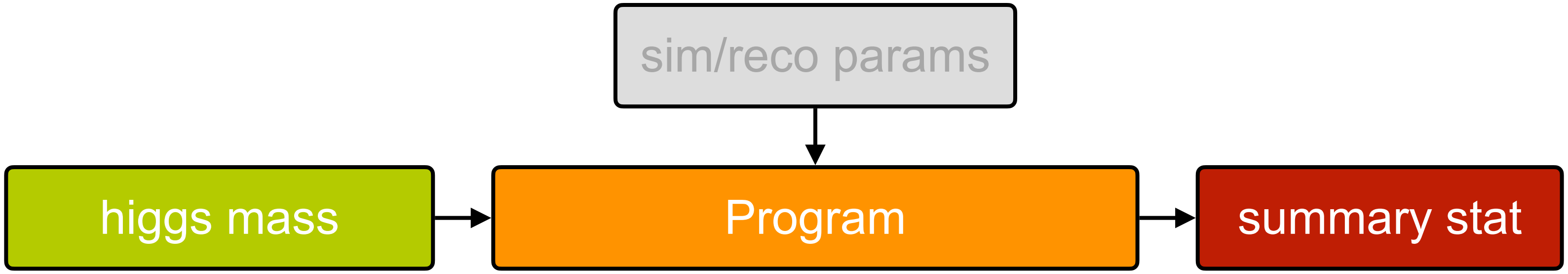


Nested structure with boundaries between semantic steps of computation similar to what we see in our workflow systems:

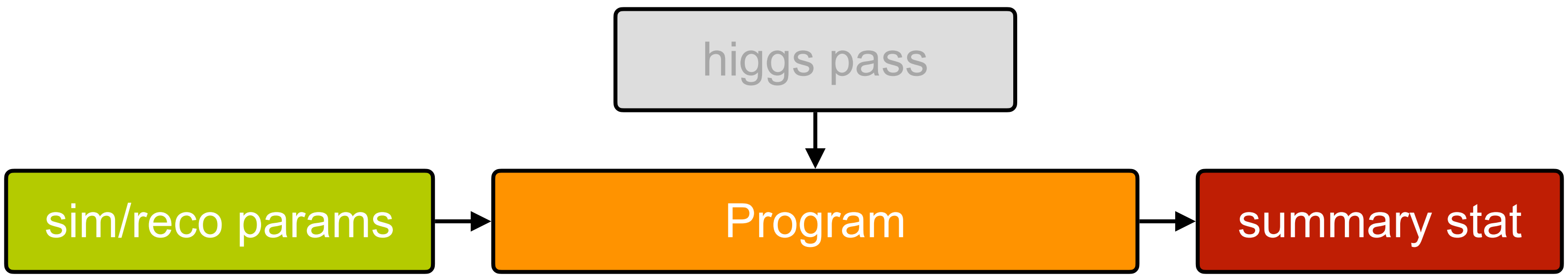




For HEP: examples of how to draw the diagram:



$$q(x, \theta_{\text{physics}}, \theta_{\text{sim,reco}}) \quad \partial q / \partial \theta_{\text{physics}}$$



$$q(x, \theta_{\text{physics}}, \theta_{\text{sim,reco}}) \quad \partial q / \partial \theta_{\text{sim,reco}}$$