IACS CS 207 | Systems Development for Computational Science

PyPE: a Python Pipeline Engine, Part 4

Part 4: Execution

Skeleton Updates

As usual, your first task is to update your PyPE implementation with some new code. Like the other PyPE labs, be careful copying functions and code over. We've put these files in the same skeleton repository as before: https://github.com/rdadolf/pype-skeleton (https://github.com/rdadolf/pype-skeleton)

- `fgir.py` : the new `forward` enum in FGNodeType, and `topological_flowgraph_pass` .
- `optimize.py` : the new `TopologicalFlowgraphOptimization` and `InlineComponents` functions.
- `pcode.py` : the whole file.
- `pipeline.py` : updates to the imports, the `compile` method, and a new `__getitem__` method
- `translate.py` : minor fix to `LoweringVisitor`

Back to top ↑

A PyPE Interpreter

Today you'll be closing the loop on PyPE: we'll finally be able to actually execute PyPE programs and get real answers out. Our goal is to build an interpreter. You've already had some experience with the internals of the Python interpreter, but ours will work slightly differently, due to language differences.

Back to top ↑

Execution Model

PyPE programs describe computational pipelines. Unlike normal imperative languages, computational pipelines don't have an implicit notion of time or sequentiality. So instead of executing a stream of high-level instructions like the Python interpreter, our interpreter will execute operations whenever the data they require is available. As we discussed a while ago, this is called a dataflow model.

More specifically, we are going to construct a set of asynchronously-scheduled functions connected by data buffers. This means that we'll launch *every function* at once and let them wait for data. As the inputs percolate through the pipeline, each function will trigger in turn.

You've already seen how to create asynchronous functions using coroutines, and we'll introduce queues in a little bit. First, however, we need to take care of a little wrinkle with component composition.

Back to top ↑

Inlining

PyPE allows components to be invoked from other components. This feature greatly improves modularity and reuse, but the latter aspect (reuse) also causes us some headaches for execution. Consider the following PyPE program:

```
(import some_functions)
{ mul (input x y) (:= z (* x y)) (output z) }
{ dist (input a b) (:= c (+ (mul a b) (mul b a))) (output c) }
```

Assuming we have the appropriate functions defined, this program uses the `sq` component twice. While this is convenient to write, it means that we have to be able to push two separate sets of parameters through the `mul` component. Moreover, because PyPE has no notion of ordering besides data dependences, it's entirely possible that the two invocations of `mul` happen *concurrently*. This is a problem. To see exactly why, let's give some imaginary FGIR node numbers to specific computational nodes:

- `(* x y)` in `mul` will be `@N1`
- `(input a b)` will be `@N2` and `@N3` for `a` and `b`, respectively
- `(input x y)` will be `@N4` and `@N5` for `x` and `y`, respectively

Recall that our approach is to launch all of the nodes at once, so an asynchronous function for `@N1` through `@N5` will start and wait for data. Now let's imagine two possible orderings for how these functions actually trigger (omitting all other nodes in between):

ordering 1: `@N2, @N3 ... @N4(a), @N5(b), @N1(a,b) ... @N4(b), @N5(a), @N1(b,a)`

ordering 2: `@N2, @N3 ... @N4(a), @N5(a), @N1(a,a) ... @N4(b), @N5(b), @N1(b,b)`

The letters in parentheses are reminders to help you keep track of what data values are being processed. In the first example, all the coroutines for `(mul a b)` fire before all the coroutines for `(mul b a)`. In the second order, we simply swap the execution order of the `@N5` nodes, so that the `@N5` node invoked by the *second* `mul` happens first. This is valid, because there is no data dependence. Unfortunately, that means that instead of multiplying `a` and `b` twice, we actually multiply `b` and `b`, then `a` and `a`, and we get a different answer.

This is a direct result of creating only one instance of the `mul` component, even though there are two uses.

While there are a number of different solutions to this issue, we'll take the easiest one: not sharing instances. Instead, we'll be creating as many instances of components as there are uses. In other words, whenever a component is called, we're going to make a copy of the component and insert it directly into the calling component. So our program is going to look something like this:

```
(import some_functions)
{ mul (input x y) (:= z (* x y)) (output z) }
{ dist (input a b)
  (:= c (+
    (let (x1 <- a) (y1 <- b)
      (* x1 y1))
    (let (x2 <- b) (y2 <- a)
      (* x2 y2))))
  (output c) }
```

Where the `let` and `<-` notation is not real PyPE syntax, just representation of the basic idea that we're trying to accomplish.

In languages with functions, this is called *inlining*.

Writing an inliner is a little tricky, and we have other things for you to write. So instead, we'd like you to prove that the version we gave you in `optimize.py` does what we say it does. In other words, we'd like you to write test cases (with **100%** coverage) for the `visit()` method of `InlineComponents` .

There's two main invariants to check: first, that the function behavior isn't broken (all the inputs and outputs are still there, etc.), and second, that there are *no* component nodes in the graph anymore. If there are, you know that the inliner didn't complete its task.

Back to top ↑

`asyncio` and Queues

Now that we've smoothed out the issue with composed components, we'll move on to constructing a network or communicating coroutines. The major new piece you'll need is the `asyncio.Queue` object. A queue (in `asyncio`) is just a buffer that has asynchronous methods for getting and putting data from it. The key is that this allows us to attempt to get data into the queue before we put data into it. Under normal circumstances, this would cause an error. Using `asyncio.Queue` , however, it just causes the `get()` coroutine to block until data is put into the queue.

This is a much simpler abstraction for PyPE programs because it changes from a control-flow-based approach (moving data between coroutines by directly calling them with `yield from` / `await`) to a data-flow approach (moving data through asynchronous storage buffers).

It's probably worthwhile to glance over the documentation for `asyncio.Queue` (https://docs.python.org/3/library/asyncio-queue.html) before trying to use them.

Back to top ↑

Building a Producer/Consumer Pipeline

There's two major parts to building an asynchronous computational pipeline from FGIR:

- a coroutine for every node in every flowgraph
- a queue between every pair of dependent nodes

We'll solve the first one by using an abstract interface to handle creating coroutines for us. This class, `PCodeOp` , has two parts: first is a helper function (`_node`) to create a coroutine that waits for all of its input queues, computes a function on the values, then writes that value on all of its output queues. You'll need to write this function. The second part is a set of functions corresponding to the type of nodes we find in FGIR. Each of these are slightly different, but most do very little fancy stuff. We've provided most of them, but we'd like you to write the `libraryfunction` method. It should be very similar to the `librarymethod` method. You should look back at the `.ref` attribute of FGIR nodes if you're lost.

The second major part is constructing queues between all dependent nodes. We'll deal with this in two steps: first, by iterating over all the edges in a flowgraph, creating a queue for each edge, and storing them in a table. Second, when we iterate of all the nodes in the flowgraph, we'll pass these queues into the `PCodeOp` methods that we wrote above. This will effectively link the queues to the coroutines.

All of this code will be run for us in a `FlowgraphOptimization` .

Note: This is tricky code. It will take a couple of iterations to get right. On the upside, once you have it working, your PyPE compiler should be complete!

Back to top ↑

Running PyPE Programs

Now comes the moment of truth. To run a pype program, just build a `Pipeline` object from a .ppl file, select a component to run, and provide it inputs:

```
pipeline = pype.Pipeline('file.ppl')
value = pipeline['component_name'].run(input1, input2)
print(value)
```

In a test case, we'd like you to create a `TimeSeries` object for the numbers -50 through 49 at time points 0 through 99 and standardize (mean 0, std 1) using the example0.ppl and example1.ppl pipelines. Show us that the resulting output has mean 0 and standard deviation 1.

(Note: you'll have to have most everything else we've asked you to do in class complete before this will work.)

Back to top ↑