DiFlow

Graphical design and cover picture by Anneleen Maelfeyt (http://anneleenmaelfeyt.be/).

Typeset using LaTeX. More info to be found at http://www.data-intuitive.com.

Contents

Introduction

DiFlow[1,2] is an abstraction layer on top of NextFlow[3]'s DSL2[4]. DiFlow is a set of principles and guidelines for building NextFlow pipelines that allow the developer to declaratively define processing components and the user to declare the pipeline logic in a clean and intuitive way.

Viash[5] is a tool that (among other things) allows us to *use* DiFlow and make it practical, without the burden of maintaining boilerplate or *glue* code.

[1] https://pointer

[2] DiFlow stands for `[NextFlow] D[SL2] I[mprovement] Flow` or maybe also `D[ata] I[ntuitive] Flow`?

[3] https://www.nextflow.io/

[4] https://www.nextflow.io/docs/latest/dsl2.html

[5] http://data-intuitive.com/viash_docs

Functional Reactive Programming (FRP)

If you're new to Functional Reactive Programming (FRP), here are a few pointers to posts and a video that introduce the concepts:

- An excellent Medium post[6] from Timo Stöttner
- The introduction[7] to Reactive Programming you've been missing from André Staltz.
- A very insightful presentation[8] by Staltz where he introduces FRP from first principles (with live coding).

In what follows, we will refer to *streams* in line with those authors but if you're used to working with Rx[9] you would call this an observable.

[6] https://itnext.io/demystifying-functional-reactive-programming-67767dbe520b

[7] https://gist.github.com/staltz/868e7e9bc2a7b8c1f754

[8] https://www.youtube.com/watch?v=fdol03pcvMA

[9] http://reactivex.io/

FRP for pipelines

Other initiatives have recognized that FRP is a good fit for pipeline development. Recent research and development also confirms this:

- Skitter[10]
- Krews[11]

[10] https://soft.vub.ac.be/~mathsaey/skitter/

[11] https://github.com/weng-lab/krews

NextFlow

Introduction

For a step-by-step introduction to NextFlow (both the original DSL as well as DSL2), please refer to this repository[12] created and maintained by the VIB[13].

Another very interesting resource is this cheatsheet[14] by Daniele Cook. It not only contains a handy cheat sheet, but also some conventions one should know about as well as some pitfalls.

[12] https://github.com/vibbits/nextflow-course
[13] https://vib.be/
[14] https://github.com/danrlu/Nextflow_cheatsheet

FRP in NextFlow

The `Channel`[15] class used by NextFlow, itself based on the DataFlow Programming Model[16] can in fact be regarded as an implementation of a Functional Reactive Programming library. Having said that, NextFlow allows one to mix functional and imperative programming to the point that a developer is able to shoot its own foot.

Furthermore, `Channel`s can not be nested which complicates certain operations on the streams.

[15] https://www.nextflow.io/docs/latest/channel.html
[16] https://en.wikipedia.org/wiki/Dataflow_programming

NextFlow DSL(2)

DSL2[17] is a crucial development in NextFlow because it avoid having to maintain large, monolithic pipeline definitions in one file. With DSL2, developer can spin off functionality in separate files and `import` what is needed.

This also potentially opens up ways to build (reusable) modules that could be used in different projects. That is exactly what a lot of organizations need.

[17] https://www.nextflow.io/docs/latest/dsl2.html

DiFlow

The NoPipeline approach

For developing the pipeline, we set out with a few goals in mind:

- Build modules where each modules deals with a specific (computational) task
- Make sure those modules can be reused
- Make sure the module functionality can be tested and validated
- Make sure modules have a consistent API, so that
  a. calling a module is straightforward
  b. including a module in a pipeline is transparent and seamless

Please note that nothing in these requirements has to do with running a pipeline itself. Rather, we consider this a bottom-up system whereby we first focus on a solid foundation before we actually start to tie things together.

That's why we call this the NoPipeline approach, similar to NoSQL where 'No' does not stand for *not*, but rather 'Not Only'. The idea is to focus on the pipeline aspect *after* the steps are properly defined and tested.

General Requirements and design principles

Reproducibility

I originally did not include it as a design principle for the simple reason that I think it's obvious. This should be every researcher's top priority.

Pipeline Parameters vs Runtime Parameters

We make a strict distinction between parameters that are defined for the *FULL* pipeline and those that are defined at runtime.

Pipeline Parameters We currently have 4 pipeline parameters: Docker prefix, `ddir`, `rdir` and `pdir`.

Runtime Parameters Runtime parameters differ from pipeline parameters in that they may be different for parallel runs of a process. A few examples:

- Some samples may require different filter threshold than others
- After concatenation, clustering may be run with different cluster parameters

- etc.

In other words, it does not make sense to define those parameters for the full pipeline because they are not static.

Consistent API

When we started out with the project and chose to use NextFlow as a workflow engine, I kept on thinking that the level of abstraction should have been higher. With DSL1, all you could do was create one long list of NextFlow code, tied together by `Channel`s.

With DSL2, it became feasible to *organise* stuff in separate NextFlow files and import what is required. But in larger codebases, this is not really a benefit because every modules/workflow may have its own parameters and output. No structure is imposed. `Workflow`s are basically functions taking parameters in and returning values.

I think it makes sense to define an API and to stick to it as much as possible. This makes using the modules/workflows easier…

Flat Module Structure

We want to avoid having nested modules, but rather support a pool of modules to be mixed and matched.

As a consequence, this allows a very low threshold for including third-party modules: just add it to the collection of modules and import it in the pipeline. In order to facilitate the inclusion of such third-party modules that are developed in their own respective repositories, we added one additional layer in the hierarchy allowing for such a splitting.

Job Serialization

We avoid requiring the sources of the job available in the runtime environment, i.e., the Docker container. In other words, all code and config is serialized and sent with the *process*.

An abstract computation step

The module concept inspired us to think of an abstract way to represent a computation step and implement this in NextFlow. We wrote [Portash] to this end. But Portash had its shortcomings. The most important of which was that it did not adhere to separation of concerns: execution definition (what?) where mixed up with execution context (how?/where?). Moreover, dynamic nature of Portash lends itself well to running a tool as a service, but not so much in a batch process.

Nevertheless, we were able to express a generic NextFlow step as pure *configuration* that is passed to a process at runtime. This allows for some very interesting functionality. Some prototypes were developed, the last one of which could run a single-cell RNA pipeline from mapping to generating an integrated dataset combining different samples.

The run-configuration was provided by means of a Portash YAML spec residing in the module directory. It must be stressed that not requiring the component *code* to be already available inside the container is a big plus. It means a container contains dependencies, not the actual run script so the latter can be updated more frequently. This is especially useful during component and pipeline development.

Our first implementation had a few disadvantages:

- It contained a mix of what to run and how to run it, but it did not contain information on the container to run in. This had to be configured externally, but then the module is not an independent entity anymore.
- Specifying and overriding YAML content in Groovy is possible, but not something that is intuitive. We worked around that by letting the user specify custom configuration using a Groovy nested `Map`.
- The module functionality was abstracted with a consistent API and the difference between 2 modules was just a few lines of code with a different name or pointer. But still, one had to maintain that and making a similar change in a growing set of module files is a recipe for mistakes.

But overall, the concept of an abstract computation step proved to work, it was just that a few ingredients were still missing it seemed. On the positive side, we showed that it's possible to have an abstract API for (NextFlow) modules that keeps the underlying implementation hidden while improving the readability of the pipeline code.

Toward implementation

What is needed as information in order to run a computation step in a pipeline?

1. First, we need data or generally speaking, **input**. Components/modules and pipelines should run zero-touch, so input has to be provided at startup time.
2. Secondly, we need to know what to run en how to run it. This is in effect the definition of a modules or pipeline step.
3. Thirdly, in many cases we will require the possibility to change parameters for individual modules in the pipeline, for instance cutoff values for a filter, or the number of clusters for a clustering algorithm. The classical way to do that is via the `params` object.

One might wonder if there is a difference between input and parameters pointing to input is also a kind of parametrization. The reason those are kept apart is that additional validation steps are necessary for the data. Most pipeline systems trace input/output closely whereas parameters are ways to configure the steps in the pipeline.

In terms of FRP, and especially in the DataFlow model, we also have to keep track of the *forks* in a parallel execution scenario. For instance, if 10 batches of data can be processed in parallel we should give all 10 of them an ID so that individual forks can be distinguished. We will see that those IDs become crucial in most pipelines.

We end up with a model for a stream/channel as follows (conceptually):

```
[ ID, data, config ]
```

were

- `ID` is just a string or any object for that matter that can be compared later. We usually work with strings.
- `data` is a pointer to the (input) data. With NextFlow, this should be a `Path` object, ideally created using the `file()` helper function.
- `config` is a nested `Map` where the first level keys are chosen to be simply an identifier of the pipeline step. Other approaches can be taken here, but that's what we did.

This can be a triplet, or a list with mixed types. In Groovy, both can be used interchangeably.

The output of a pipeline step/mudules adheres to the same structure so that pipeline steps can easily be chained.

Step by step

Let us illustrate some key features of NextFlow together with how we use them in DiFlow and approach this step by step.

The code blocks below are run with the following version of NextFlow:

```
nextflow -v

nextflow version 20.10.0.5430
```

Step 1 - Operate on a *stream*

Let us illustrate the stream-like nature of a NXF `Channel` using a very simple example: computing $1 + 1$.

```
// Step - 1
workflow step1 {
  Channel.from(1) \
    | map{ it + 1 } \
    | view{ it }
}
```

This chunk is directly taken from `main.nf`, running it can be done as follows:

```
> nextflow -q run . -entry step1
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
2
```

Step 2 - Operate on a stream in parallel

NextFlow (and streams in general) are supposed to be a good fit for parallel execution. Let's see how this can be done:

```
// Step - 2
workflow step2 {
  Channel.from( [ 1, 2, 3 ] ) \
    | map{ it + 1 } \
    | view{ it }
}
```

Running it can be done using:

```
> nextflow -q run . -entry step2
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
2
3
4
```

Step 3 - Operate on a stream using a `process`

In the previous example, we ran 3 parallel executions each time applying the same simple function: adding one. Let us simulate now a more real-life example where parallel executions will not take the same amount of time. We do this by defining a `process` and `workflow` that uses this process. The rest is similar to our example before.

```
// Step - 3
process add {
  input:
    val(input)
  output:
    val(output)
  exec:
    output = input + 1
}
workflow step3 {
  Channel.from( [ 1, 2, 3 ] ) \
    | add \
    | view{ it }
}
```

Running it is again the same.

```
> nextflow -q run . -entry step3
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
2
4
3
```

The result will be a permutation of 2,3 and 4. Try it multiple times to verify for yourself that the order is not guaranteed to be the same. Even though the execution times will not be that much different! In other words, a `Channel` does not guarantee the order, and that's a good thing.

Step 4 - How `map` is synchronous

An illustrative test is one where we do not use a `process` for the execution, but rather just `map` but such that one of the inputs *takes longer* to process, i.e.:

```
// Step - 4
def waitAndReturn(it) { sleep(2000); return it }
workflow step4 {
  Channel.from( [ 1, 2, 3 ] ) \
    | map{ (it == 2) ? waitAndReturn(it) : it } \
    | map{ it + 1 } \
    | view{ it }
}
```

Running it:

```
> nextflow -q run . -entry step4
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
2
3
4
```

The result may be somewhat unexpected, the order is retained even though there's a 2 second delay between the first entry and the rest. The `sleep` in other words blocks all the parallel execution branches.

This is a clear indication of why it's better to use a `process` to execute computations. On the other hand, as long as we *stay* inside the `map` and don't run a `process`, the order is retained. This opens up possibilities that we will exploit in what follows.

Step 5 - Introduce an `ID`

If we can not guarantee the order of the different parallel branches, we should introduce a *branch ID*. This may be a label, a sample ID, a batch ID, etc. It's the unit of parallelization.

```
// Step - 5
process addTuple {
  input:
    tuple val(id), val(input)
  output:
    tuple val("${id}"), val(output)
  exec:
    output = input + 1
}
workflow step5 {
  Channel.from( [ 1, 2, 3 ] ) \
    | map{ el -> [ el.toString(), el ]} \
    | addTuple \
    | view{ it }
}
```

We can run this code sample in the same way as the previous examples:

```
> nextflow -q run . -entry step5
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[3, 4]
[1, 2]
[2, 3]
```

Please note that the function to add 1 remains exactly the same, we only added the `id` as the first element of the tuple in both input and output. As such we keep a handle on which sample is which, by means of the *key* in the tuple.

Note: Later, we will extend this tuple and add configuration parameters to it… but this was supposed to go step by step.

Step 6 - Add a process parameter

What if we want to be able to configure the term in the sum? This would require a parameter to be sent with the process invocation. Let's see how this can be done.

```
// Step - 6
process addTupleWithParameter {
  input:
    tuple val(id), val(input), val(term)
  output:
    tuple val("${id}"), val(output)
  exec:
    output = input + term
}
workflow step6 {
  Channel.from( [ 1, 2, 3 ] ) \
    | map{ el -> [ el.toString(), el, 10 ]} \
    | addTupleWithParameter \
    | view{ it }
}
```

The result is:

```
> nextflow -q run . -entry step6
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[2, 12]
[1, 11]
[3, 13]
```

This works, but is not very flexible. What if we want to configure the operator as well? What if we want to have branch-specific configuration? We can add a whole list of parameters, but that means that the `process` signature may be different for every `process` that we define. That is not a preferred solution.

Step 7 - Use a `Map` to store parameters

Let us use a simple `Map` to add 2 configuration parameters:

```
// Step - 7
process addTupleWithMap {
  input:
    tuple val(id), val(input), val(config)
  output:
    tuple val("${id}"), val(output)
  exec:
    output = (config.operator == "+")
                ? input + config.term
                : input - config.term
}
workflow step7 {
  Channel.from( [ 1, 2, 3 ] ) \
    | map{ el ->
      [
        el.toString(),
        el,
        [ "operator" : "-", "term" : 10 ]
      ] } \
    | addTupleWithMap \
    | view{ it }
}
```

The result is:

```
> nextflow -q run . -entry step7
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[2, -8]
[3, -7]
[1, -9]
```

Step 8 - Use a `Map` with a process-key

Step 7 provides a way to use a consistent API for a process. Ideally, however, we would like different `process` invocation to be chained rather than to explicitly add the correct configuration all the time. Let us add an additional key to the map, so that a process knows *it's scope*.

```
// Step - 8
process addTupleWithProcessHash {
  input:
    tuple val(id), val(input), val(config)
  output:
    tuple val("${id}"), val(output)
  exec:
    def thisConf = config.addTupleWithProcessHash
    output = (thisConf.operator == "+")
                ? input + thisConf.term
                : input - thisConf.term
}
workflow step8 {
  Channel.from( [ 1, 2, 3 ] ) \
    | map{ el ->
      [
        el.toString(),
        el,
        [ "addTupleWithProcessHash" :
          [
            "operator" : "-",
            "term" : 10
          ]
        ]
      ] } \
    | addTupleWithProcessHash \
    | view{ it }
}
```

Which yields:

```
> nextflow -q run . -entry step8
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[3, -7]
[2, -8]
[1, -9]
```

Please note that we used the process name as a key in the map, so that each process can tell what configuration parameter are relevant for its own scope. We call this `Map` a `ConfigMap`.

Step 9 - Use a `ConfigMap` with a shell script

We used native Groovy code in the `process` examples above. Let us now use a shell script:

```
// Step - 9
process addTupleWithProcessHashScript {
  input:
    tuple val(id), val(input), val(config)
  output:
    tuple val("${id}"), stdout
  script:
    def thisConf = config.addTupleWithProcessHashScript
    def operator = thisConf.operator
    def term = thisConf.term
    """
    echo \$( expr $input $operator ${thisConf.term} )
    """
}
workflow step9 {
  Channel.from( [ 1, 2, 3 ] ) \
    | map{ el ->
      [
        el.toString(),
        el,
        [ "addTupleWithProcessHashScript" :
          [
            "operator" : "-",
            "term" : 10
          ]
        ]
      ] } \
    | addTupleWithProcessHashScript \
    | view{ it }
}
```

Running this (in the same way as before), we get something along these lines:

```
> nextflow -q run . -entry step9
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[2, -8
]
[1, -9
]
[3, -7
]
```

This is because the `stdout` qualifier captures the newline at the end of the code block. We could look for ways to circumvent that, but that is not the point here.

What's important to notice here:

1. We can not just retrieve individual entries in `config` in the shell, we have to let Groovy do that.
2. That means we either first retrieve individual values and store them in a variable,
3. or we use the `${...}` notation for that.

4. If we want to use `bash` variables, the `$` symbol has to be escaped.

Obviously, passing config like this requires a lot of typing (especially as additional parameters are introduced) and is error prone.

> A DiFlow module selects the appropriate key from a `ConfigMap` and uses that as its configuration settings. In a sense, we *scope* the global `ConfigMap` and use it as a local variable within a module. A module could also update the global `ConfigMap` and there may be cases where this is necessary, but care should be taken to update the global state like this.

The scoping done can be seen as a Functional Lens[18], although it's a poor man's implementation at that. Furthermore, in some FRP frameworks, so-called *reducers* or *transducers* are used for transforming state in an application. We did not (yet) consider the further extension in that direction.

[18] https://medium.com/@dtipson/functional-lenses-d1aba9e52254

Step 10 - Running a *pipeline*

We used the pipe `|` symbol to combine different steps in a *pipeline* and we noticed that a `process` can do computations on parallel branches. That's nice, but we have not yet given an example of running 2 processes, one after the other.

There are a few things we have to note before we go to an example:

1. It's not possible to call the same process twice, a strange error occurs in that case[19].
2. If we want to pipe the output of one process as input of the next, the I/O signature needs to be exactly the same, so the `output` of the `process` should be a triplet as well.

[19] It *is* possible in some cases however to manipulate the `Channel` such that a process is effectively run twice on the same data, but that is a more advanced topic.

```
// Step - 10
process process_step10a {
  input:
    tuple val(id), val(input), val(term)
  output:
    tuple val("${id}"), val(output), val("${term}")
  exec:
    output = input.toInteger() + term.toInteger()
}
process process_step10b {
  input:
    tuple val(id), val(input), val(term)
  output:
    tuple val("${id}"), val(output), val("${term}")
  exec:
    output = input.toInteger() - term.toInteger()
}
workflow step10 {
  Channel.from( [ 1, 2, 3 ] ) \
    | map{ el -> [ el.toString(), el, 10 ] } \
    | process_step10a \
    | process_step10b \
    | view{ it }
}
```

The result of this is that first 10 is added and then the same 10 is subtracted again, which results in the same as the original. Please note that the output contains 3 elements, also the `term` passed to the `process`:

```
> nextflow -q run . -entry step10
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[1, 1, 10]
[3, 3, 10]
[2, 2, 10]
```

We can configure the second `process` (subtraction) by adding an additional `map` in the mix:

```
// Step - 10a
workflow step10a {
  Channel.from( [ 1, 2, 3 ] ) \
    | map{ el -> [ el.toString(), el, 10 ] } \
    | process_step10a \
    | map{ id, value, term -> [ id, value, 5 ] } \
    | map{ [ it[0], it[1], 5 ] } \
    | map{ x -> [ x[0], x[1], 5 ] } \
    | process_step10b \
    | view{ it }
}
```

Resulting in:

```
> nextflow -q run . -entry step10a
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[1, 6, 5]
[2, 7, 5]
[3, 8, 5]
```

Please note that we define the closure in a different manner here, using the special variable `it`. We could also write (to the same effect):

```
  ...
  | map{ x -> [ x[0], x[1], 5 ] } \
  ...
```

or even

```
  ...
  | map{ id, value, term -> [ id, value, 5 ] } \
  ...
```

Step 11 - A more generic process

What if we rewrite the previous using some of the techniques introduced earlier. Let us specify the operator as a parameter and try to stick to just 1 `process` definition.

```
// Step - 11
process process_step11 {
    input:
        tuple val(id), val(input), val(config)
    output:
        tuple val("${id}"), val(output), val("${config}")
    exec:
        if (config.operator == "+")
           output = input.toInteger() + config.term.toInteger()
        else
           output = input.toInteger() - config.term.toInteger()
}
workflow step11 {
  Channel.from( [ 1, 2, 3 ] ) \
    | map{ el -> [ el.toString(), el, [ : ] ] } \
    | process_step11 \
    | map{ id, value, config ->
      [
        id,
        value,
        [ "term" : 11, "operator" : "-" ]
      ] } \
    | process_step11 \
    | view{ [ it[0], it[1] ] }
}
```

This little workflow definition results in an error, just like we warned before:

```
> nextflow -q run . -entry step11
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
assert processConfig==null
       |
       ['echo':false, 'cacheable':true, 'shell':['/bin/bash', '-ue'], 'validExitStatus':[0], 'maxRetries':0, 'max

 -- Check script 'main.nf' at line: 248 or see '.nextflow.log' file for more details
```

There is, however, one simple way around this: `include { ... as ...}`. Let us see how this works.

First, we store the `process` in a file `examples/step/step11.nf`:

```
process process_step11 {
  input:
    tuple val(id), val(input), val(config)
  output:
    tuple val("${id}"), val(output), val("${config}")
  exec:
    if (config.operator == "+")
      output = input.toInteger() + config.term.toInteger()
    else
      output = input.toInteger() - config.term.toInteger()
}
```

The `workflow` definition becomes:

```
// Step - 11a
include { process_step11 as process_step11a } \
  from './examples/modules/step11.nf'
include { process_step11 as process_step11b } \
  from './examples/modules/step11.nf'
workflow step11a {
  Channel.from( [ 1, 2, 3 ] ) \
    | map{ el -> [ el.toString(), el, [ : ] ] } \
    | map{ id, value, config ->
      [
        id,
        value,
        [ "term" : 5, "operator" : "+" ]
      ] } \
    | process_step11a \
    | map{ id, value, config ->
      [
        id,
        value,
        [ "term" : 11, "operator" : "-" ]
      ] } \
    | process_step11b \
    | view{ [ it[0], it[1] ] }
}
```

Running this yields an output similar to this:

```
> nextflow -q run . -entry step11a
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[1, -5]
[2, -4]
[3, -3]
```

We made a few minor changes to the workflow code in the meanwhile:

1. Splitting the conversion from an array of items to the triplet is now done explicitly and separate from specifying the configuration for the `process` itself.
2. The `view` now only contains the relevant parts, not the configuration part for the last `process`.

The above example illustrates the `include` functionality of NextFlow DSL2. This was not possible with prior versions.

Step 12 - Map/reduce in NextFlow

Let's implement a simple map/reduce schema with what we developed above. Until now, we basically covered the mapping stage: starting from 3 independent number, execute a function on each *branch* individually. Now, we want to calculate the sum at the end (reduce phase).

We do this by adding a `process` to the example in Step 10

```
// Step - 12
process process_step12 {
  input:
    tuple val(id), val(input), val(term)
  output:
    tuple val("${id}"), val(output), val("${term}")
  exec:
    output = input.sum()
}
workflow step12 {
  Channel.from( [ 1, 2, 3 ] ) \
    | map{ el -> [ el.toString(), el, 10 ] } \
    | process_step10a \
    | toList \
    | map{
      [
        "sum",
        it.collect{ id, value, config -> value },
        [ : ]
      ] } \
    | process_step12 \
    | view{ [ it[0], it[1] ] }
}
```

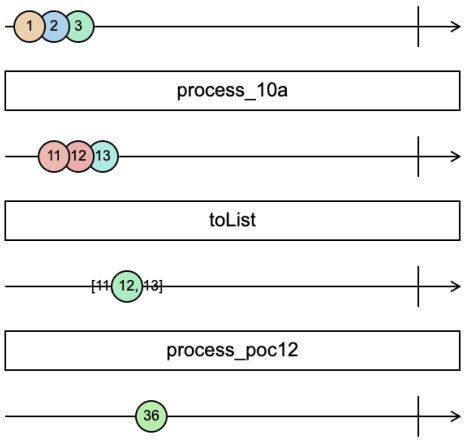
Running this yields:

```
> nextflow -q run . -entry step12
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[sum, 36]
```

A few remarks are in order here:

1. We use the `toList` operator on the output of `process_step10a`. This can be regarded as merging the 3 parallel branches into one branch. The result has the signature `Channel[List[Triplet]]`. This `toList` operator only *outputs* something on the output `Channel` when all incoming branches have finished and the *merge* can effectively be performed.
2. It's important to note that what is passed through the pipe is still a `Channel`, only the number of *branches*, *nodes*, or whatever you want to call it differs.
3. The long `map{ [ "sum", ... }` line may seem complex at first, but it's really not. We take in `List[Triplet]` and convert this to `Triplet`. The first element of the triplet is just an identifier (`sum`). The last is the configuration map, but we don't need configuration for the sum. As the second element we want to obtain `List[Int]`, where the values are the 2nd element from the original triplets. The Groovy function `collect` on an array is like `map` in many other languages.

The marble diagram can be depicted conceptually as follows, where we note that in effect it's triplets rather than numbers that are contained in the marbles:



Please note that though we define the *pipeline* sequentially, the 3 numbers are first handled in parallel and only combined when calling `toList`. Stated differently, parallelism comes for free when defining workflows like this.

Step 13 - Files as input/output

Let us tackle a different angle now and start to deal with files as input and output. In order to do this, we will mimic the functionality from earlier and modify it such that a file is used as input and output is also written to a file.

The following combination of `process` and `workflow` definition does exactly the same as before, but now from one or more files containing just a single integer number:

```
// Step - 13
process process_step13 {
  input:
    tuple val(id), file(input), val(config)
  output:
    tuple val("${id}"), file("output.txt"), val("${config}")
  script:
    """
    a=`cat $input`
    let result="\$a + ${config.term}"
    echo "\$result" > output.txt
    """
}
workflow step13 {
  Channel.fromPath( params.input ) \
    | map{ el ->
      [
        el.baseName.toString(),
        el,
        [ "operator" : "-", "term" : 10 ]
      ]} \
    | process_step13 \
    | view{ [ it[0], it[1] ] }
}
```

While doing this, we also introduced a way to specify parameters via a configuration file (`nextflow.config`) or from the CLI. In this case `params.input` points to an argument we should provide on the CLI, for instance:

```
> nextflow -q run . -entry step13 --input data/input1.txt
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[input1, <...>/work/17/bde087db08bba02e1a2dcce41c8892/output.txt]
```

Let's dissect what is going on here…

1. We provide the input file `data/input1.txt` as input which gets automatically added to the `params` map as `params.input`.
2. The content of `input1.txt` is used in the simple sum just as before.
3. The output `Channel` contains the known triplet but this time the second entry is not a value, but rather a filename.

Please note that the file is `output.txt` is automatically stored in the (unique) `work` directory. We can take a look inside to verify that the calculation succeeded:

```
> cat $(nextflow log | cut -f 3 | tail -1 | xargs nextflow log)/output.txt
11
```

It seems the calculation went well, although one might be surprised by two things:

1. The output of the calculation is stored in some randomly generated `work` directory whereas we might want it somewhere more *findable*.
2. The `process` itself defines the value of the output filename, which may seem odd… and it is.

Taking our example a bit further and exploiting the fact that parallelism is natively supported by NextFlow as we've seen before, we can pass multiple input files to the same workflow defined above.

```
> nextflow -q run . -entry step13 --input "data/input?.txt"
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[input3, <...>/work/7b/1d1d756ff5c66dbc56daa6793ec324/output.txt]
[input1, <...>/work/6c/c09325a112f2a749ec10cacd222e47/output.txt]
[input2, <...>/work/0e/7b5a91a27313b0218ac5fcf8f2b09a/output.txt]
```

Please note that we

1. provide the absolute path to the file
2. use a wildcard `*` to select multiple files
3. enclose the path (with wildcard) in double quotes to avoid shell globbing.

In the latter case, we end up with 3 output files, each named `output.txt` in their own respective (unique) `work` directory.

```
> nextflow log | cut -f 3 | tail -1 | xargs nextflow log | xargs ls
<...>/work/0e/7b5a91a27313b0218ac5fcf8f2b09a:
input2.txt
output.txt

<...>/work/6c/c09325a112f2a749ec10cacd222e47:
input1.txt
output.txt

<...>/work/7b/1d1d756ff5c66dbc56daa6793ec324:
input3.txt
output.txt
```

Step 14 - *Publishing* output

Let us tackle one of the pain points of the previous example: output files are hidden in the `work` directory. One might be tempted to specify an output file in the `process` definition as such `file("<somepath>/output.txt")` but when you try this, it will quickly turn out that this does not work in the long run (though it may work for some limited cases).

NextFlow provides a better way to achieve the required functionality: `publishDir`[20]. Let us illustrate its use with an example again and just adding the `publishDir` directive:

[20] https://www.nextflow.io/docs/latest/process.html?highlight=publish#publishdir

```
// Step - 14
process process_step14 {
    publishDir "output/"
    input:
        tuple val(id), file(input), val(config)
    output:
        tuple val("${id}"), file("output.txt"), val("${config}")
    script:
        """
        a=`cat $input`
        let result="\$a + ${config.term}"
        echo "\$result" > output.txt
        """
}
workflow step14 {
    Channel.fromPath( params.input ) \
        | map{ el ->
          [
            el.baseName.toString(),
            el,
            [ "operator" : "-", "term" : 10 ]
          ]} \
        | process_step14 \
        | view{ [ it[0], it[1] ] }
}
```

This single addition yields:

```
> nextflow -q run . -entry step14 --input data/input1.txt
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[input1, <...>/work/1a/9fe9ead2ecfcf5db71d5c6a3acfeda/output.txt]
```

This example shows us a powerful approach to publishing data. There is a similar drawback as for the output filenames, however, and that is that the `process` defines the output directory explicitly. But there is a different problem as well, which can be observed when running on multiple input files:

```
> nextflow -q run . -entry step14 --input "data/input?.txt"
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[input3, <...>/work/87/81d9a9c9a7ab73110834aa6d1cffcc/output.txt]
[input2, <...>/work/5f/85dfd25d7b0156782200fede6d9bc3/output.txt]
[input1, <...>/work/8e/028d5c3c1f072f5edc6d306f998ad2/output.txt]
```

```
> cat output/output.txt
11
```

What do you think happens here? Yes, sure, we *publish* the same `output.txt` file three times and each time overwriting the same file. The last one is the one that persists.

Step 15 - Make output files/paths unique

Let us describe a way to avoid the above issue. There are other approaches to resolve this issue, but let us for the moment look at one that can easily be reused.

```
// Step - 15
process process_step15 {
    publishDir "output/${config.id}"
    input:
        tuple val(id), file(input), val(config)
    output:
        tuple val("${id}"), file("output.txt"), val("${config}")
    script:
        """
        a=`cat $input`
        let result="\$a + ${config.term}"
        echo "\$result" > output.txt
        """
}
workflow step15 {
    Channel.fromPath( params.input ) \
        | map{ el ->
            [
              el.baseName,
              el,
              [
                "id": el.baseName,
                "operator" : "-",
                "term" : 10
              ]
            ] } \
        | process_step15 \
        | view{ [ it[0], it[1] ] }
}
```

This results in the following:

```
> nextflow -q run . -entry step15 --input "data/input?.txt"
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[input3, <...>/work/a1/0947168c069ee6f4e1e7a850f268ea/output.txt]
[input1, <...>/work/d2/41f7e0ec2fb3f8e4204aca6bf952ab/output.txt]
[input2, <...>/work/f5/845a17a4e548c531b7c63a06e50304/output.txt]
```

With the following result:

```
> cat output/input?/output.txt
11
12
13
```

In other words, since (in this case 3) parallel branches each write to the same output location we have to make sure that we add something unique for every of the parallel branches. Another approach is to tweak the name

of the output file in the `process`, but for the moment it is still fixed and defined in the `process` itself. Let us take a look at that aspect next.

Step 16 - Where to put `params`?

We want the output filename to be configurable. That means that we either use the `params` map for this (and take care it is available in modules that are `included`) or we pass it to the `process` as part of the input. Let us explore both scenarios.

But first, we need to understand a bit better where the contents of `params` comes from. We already covered a few examples where we specify a `params` key on the CLI. There is another way as well, via `nextflow.config`. In it, we can add a scope `params` and add the configuration there.

Let us reconsider the previous example (`step15`) but this time add a `nextflow.config` file like this (please update the `<...>` part according to your situation):

params.input = "$PWD/data/input?.txt"

Let us illustrate the effect by means of two examples:

```
> nextflow -q run . -entry step15
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[input3, <...>/work/d3/3b7016780d0a10bc43a4a1e303edbc/output.txt]
[input1, <...>/work/e8/52a69a6234d9008041151f0ddaea06/output.txt]
[input2, <...>/work/c9/ff527e7934c309ec34ee4fabdf0654/output.txt]
```

```
> nextflow -q run . -entry step15 --input data/input1.txt
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[input1, <...>/work/4a/9479b71506cc1ca0c536ad587498af/output.txt]
```

In other words, `params` can be defined in `nextflow.config` but if it appears on the CLI then the latter gets priority. Please be reminded that `params` is a map, the following is equivalent:

```
params {
    input = "/.../diflow/data/*.txt"
}
```

Step 17 - Add the output file to `params`

In this case, we would add a `output = ...` key to `nextflow.config` or provide `--output ...` on the CLI. This is done in the following example:

```
// Step - 17
process process_step17 {
    publishDir "output"
    input:
        tuple val(id), file(input), val(config)
    output:
        tuple val("${id}"), file(params.output), val("${config}")
    script:
        """
        a=`cat $input`
        let result="\$a + ${config.term}"
        echo "\$result" > ${params.output}
        """
}
workflow step17 {
    Channel.fromPath( params.input ) \
        | map{ el ->
          [
            el.baseName.toString(),
            el,
            [
              "id": el.baseName,
              "operator" : "-",
              "term" : 10
            ]
          ] } \
        | process_step17 \
        | view{ [ it[0], it[1] ] }
}
```

The code that is run:

```
> nextflow -q run . -entry step17 --input data/input.txt --output output.txt
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[input, <...>/work/b1/7f1b50fa5f6e9fdb83bc656e8946e7/output.txt]
```

The result is:

```
> cat output/output.txt
11
```

We note that the `params.output` occurs in the `output:` triplet as well as in the script code itself. That's quite important, otherwise NextFlow will complain the output file can not be found.

This approach does what it is supposed to do: make the output filename configurable. There are a few drawbacks however:

1. We would have to configure the filename for *every* process individually. While this can be done (`params.<process>.output` for instance), it requires additional bookkeeping on the side of the pipeline developer.
2. It does not help much because the output filename for every parallel branch again has the same name. In other words, we still have to have the `publishDir`

In all fairness, these issues only arise when you want to *publish* the output data because in the other case every process *lives* in its own (unique) `work` directory.

Step 18 - Add the output filename to the triplet

The other approach to take is to add the output filename to the triplet provided as input to the `process`. This can be done similarly to what we did with the input filename, i.e.:

```
// Step - 18
process process_step18 {
    publishDir "output"
    input:
        tuple val(id), file(input), val(config)
    output:
        tuple val("${id}"), file("${config.output}"), val("${config}")
    script:
        """
        a=`cat $input`
        let result="\$a + ${config.term}"
        echo "\$result" > ${config.output}
        """
}
workflow step18 {
    Channel.fromPath( params.input ) \
        | map{ el -> [
            el.baseName.toString(),
            el,
            [
                "output" : "output_from_${el.baseName}.txt",
                "id": el.baseName,
                "operator" : "-",
                "term" : 10
            ]
          ]} \
        | process_step18 \
        | view{ [ it[0], it[1] ] }
}
```

In order to make a bit more sense of the (gradually growing) configuration map that is sent to the `process`, we tuned the layout a bit. In this case, the output filename that is configured contains an identifier for the input as well. In this way, the output is always unique.

Since we have configured `params.input` in `nextflow.config`, we are able to just run our new *pipeline*:

```
> nextflow -q run . -entry step18
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[input3, <...>/work/9b/7a08a98e8ebf69b0781e06ac95d4d6/output_from_input3.txt]
[input1, <...>/work/e1/2a333f9af260daaad4b86636426004/output_from_input1.txt]
[input2, <...>/work/30/90bc0238055daacc9fdeea3f232c27/output_from_input2.txt]
```

```
> ls -1 output/output_from*
output/output_from_input1.txt
output/output_from_input2.txt
output/output_from_input3.txt
```

In other words, this allows to distinguish between parallel branches in the pipeline.

Please note that if we add steps to the *pipeline*, because the output is reported as input for the next `process`, it automatically points to the correct filename even though the next process is not aware of the way the output filename has been specified. That's nice.

Step 19 - Use a closure

We mentioned that there are 2 ways to pass an output filename to a `process`. There is a third one, using a closure or function to handle the naming for us.

Let us illustrate this with an example again:

```
// Step - 19
def out_from_in = { it -> it.baseName + "-out.txt" }
process process_step19 {
    publishDir "output"
    input:
        tuple val(id), file(input), val(config)
    output:
        tuple val("${id}"), file("${out}"), val("${config}")
    script:
        out = out_from_in(input)
        """
        a=`cat $input`
        let result="\$a + ${config.term}"
        echo "\$result" > ${out}
        """
}
workflow step19 {
    Channel.fromPath( params.input ) \
        | map{ el -> [
            el.baseName.toString(),
            el,
            [
                "id": el.baseName,
                "operator" : "-",
                "term" : 10
            ]
          ]} \
        | process_step19 \
        | view{ [ it[0], it[1] ] }
}
```

The result is as follows:

```
> nextflow -q run . -entry step19
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[input2, <...>/work/a3/2c69f0be071c1e2170a19f6f397dda/input2-out.txt]
[input3, <...>/work/0a/e2dc41121ad3b619348d3abc985839/input3-out.txt]
[input1, <...>/work/04/18d42a25a1923a3ff15cf5f6a7aae8/input1-out.txt]
```

We can even add the closure to the configuration map sent to the `process`, but NextFlow complains that this is not serializable so you may miss some features and most importantly it may not work at all times:

```
WARN: Cannot serialize context map. Cause: java.lang.IllegalArgumentException: Unknown workflow parameter definit
- Resume will not work on this process
```

This approach may seem like completely over-engineered but for a lot of use-cases it turns out to be a good fit. Although, not in the way we introduced it here. We come back to that later…

A DiFlow module contains a function definition that takes the input file name as input and *generates* an output filename.

Step 20 - The order of events in a stream

We touch upon a point that we have encountered but not really considered in-depth: the order of *things* in the `Channel` or stream. We've noticed that the order is not predictable and we've discussed that this is to be expected. In general, the duration of a `process` step may depend on the data or the number of resources available at the time of running. Also, the example where we joined the different parallel branches (Step 12 - **REF**) was independent of the order because it just calculated the sum.

Another consequence of the undetermined order of *events* is the fact that during a join or reduce phase (for instance with `toList`), the resulting order is undetermined and this messes up the caching functionality of NextFlow.

Let us give an example with a reduce process that *does* depend on the order of *events*. We divide the first element from the *map* phase by the second one:

```
// Step - 20a
process process_step20 {
    input:
        tuple val(id), val(input), val(term)
    output:
        tuple val("${id}"), val(output), val("${term}")
    exec:
        output = input[0] / input[1]
}
workflow step20a {
    Channel.from( [ 1, 2 ] ) \
        | map{ el -> [ el.toString(), el, 10 ] } \
        | process_step10a \
        | toList \
        | map{ [
                  "sum",
                  it.collect{ id, value, config -> value },
                  [ : ]
                ] } \
        | process_step20 \
        | view{ [ it[0], it[1] ] }
}
```

If you run this code like this, you get something like this when launching multiple times:

```
> nextflow -q run . -entry step20a -with-docker
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[sum, 0.9166666667]
```

```
> nextflow -q run . -entry step20a -with-docker
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[sum, 1.0909090909]
```

As an illustration, I've added the `-with-docker` option.

Luckily, there is a variant of the `toList` channel operator that takes into

account sorting: `toSortedList`. There are other operators as well, but we leave it as an exercise to look those up. The `workflow` code above simply becomes:

```
// Step - 20b
workflow step20b {
    Channel.from( [ 1, 2 ] ) \
        | map{ el -> [ el.toString(), el, 10 ] } \
        | process_step10a \
        | toSortedList{ a,b -> a[0] <=> b[0] } \
        | map{ [ "sum", it.collect{ id, value, config -> value }, [ : ] ] } \
        | process_step20 \
        | view{ [ it[0], it[1] ] }
}
```

In this example, we sort (alphabetically) on the id in the triplet.

Step 21 - Is the *triplet* really necessary?

A `process` can take multiple input `Channel`s. But then why are struggling with triplets above? Why do we make our life harder than it could be? Let us illustrate this with a little example. We define a process that takes two input `Channel`s, one containing integers and the other with strings. We simply concatenate both in the process definition:

```
// Step - 21
process process_step21 {
    input:
        val(in1)
        val(in2)
    output:
        val(out)
    exec:
        out = in1 + in2
}
workflow step21 {
    ch1_ = Channel.from( [1, 2, 3, 4, 5 ] )
    ch2_ = Channel.from( ["a", "b", "c", "d" ] )
    process_step21(ch1_, ch2_) | toSortedList | view
}
```

If we run this, we get the following result:

```
> nextflow -q run . -entry step21 -with-docker
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[1a, 2b, 3c, 4d]
```

This seems fine, it is probably what was expected to happen. If we slightly change the workflow and add a `process` step we defined earlier (`add`):

```
// Step - 21a
workflow step21a {
    ch1_ = Channel.from( [1, 2, 3, 4, 5 ] ) | add
    ch2_ = Channel.from( ["a", "b", "c", "d" ] )
    process_step21(ch1_, ch2_) | toSortedList | view
}
```

Running this two times should reveal the caveat we want to point out;

```
> nextflow -q run . -entry step21a -with-docker
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[2d, 3a, 4c, 5b]
```

```
> nextflow -q run . -entry step21a -with-docker
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[2d, 3a, 4c, 6b]
```

The result is not deterministic. Imagine you want to combine two input `Channel`s like this, but one of the `Channel`s requires some additional processing first (creating an index or a qc report) then relying on the order of operations not consistent.

In other words, we need to stick to adding an explicit `ID` and add it with the config to the triplet.

Step 22 - Toward generic processes

Dealing with computational pipelines, and looking at the examples above and beyond, we note that the `input` of a process is always the same: a triplet. The output should at least contain the `ID` and the path to the output file or directory. We want to provide the `ConfigMap` as output as well, so that input and output become consistent and we can easily chain processes.

Going a step further, we might reflect on the nature of the script-part of a `process` definition. It contains one or more commands, each with options. For the sake of the argument, let's say we need to run one command. We already know how we can provide parameters for input and output. We can now also go a step further.

We could, for instance, provide the full command line instruction via the `ConfigMap`:

```
// Step - 22
process process_step22 {
    publishDir "output"
    input:
        tuple val(id), file(input), val(config)
    output:
        tuple val("${id}"), file("${config.output}"), val("${config}")
    script:
        """
        ${config.cli}
        """
}
workflow step22 {
    Channel.fromPath( params.input ) \
        | map{ el -> [
            el.baseName.toString(),
            el,
            [
                "cli": "cat input.txt > output22.txt",
                "output": "output22.txt"
            ]
          ]} \
        | process_step22 \
        | view{ [ it[0], it[1] ] }
}
//- - -
```

Such that

```
> nextflow -q run . -entry step22 --input data/input.txt
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[input, <...>/work/cf/0822ca4f216cd2d616d25bc279eaf4/output22.txt]
```

Unsurprisingly, the content of `output22.txt` is the same as that of `input.txt`:

```
> cat output/output22.txt
1
```

This may seem silly, but let us make a few remarks anyway:

1. The output file name is specified in two places, that is not a good idea
2. The input file name is specified explicitly in the `cli` definition. We could get around by by pointing to `params.input` instead, keeping in mind correct escaping and such. It could work, but would be error prone.
3. One could be tempted (we were) to indeed create 1 generic process handler, but when looking at a pipeline run, one would not be able to distinguish the different processes from each other because the name of the process is used as an identifier.

So, while this may seem like a stupid thing to do, we use a method that is very similar to this in DiFlow. Keeping into account the above points, that is…

In practice, we do not specify the command line like shown above, but rather by specifying the command and options by means of the `ConfigMap`

for that specific process. Let us give an example:

```
params {
  ...
  cellranger {
    name = "cellranger"
    container = "mapping/cellranger:4.0.0-1"
    command = "cellranger"
    arguments {
      mode {
        name = "mode"
        otype = "--"
        description = "The run mode"
        value = "count"
        ...
      }
      input {
        name = "input"
        otype = "--"
        description = "Path of folder created by mkfastq or bcl2fastq"
        required = false
        ...
      }
      id {
        name = "id"
        otype = "--"
        description = "Output folder for the count files."
        value = "cr"
        required = false
        ...
      }
      ...
    }
    ...
  }
```

In reality, this is still a simplified version because we also use variables in `nextflow.config` but that is only for convenience.

The above is a representation of the command-line instruction to be provided inside a container (`mapping/cellranger:4.0.0-1`). The command itself is `cellranger` and the different options are listed as keys under `arguments`.

> Every DiFlow module has its own `nextflow.config` that contains a representation of the CLI instruction to run as well as a pointer to the container to be used for running.

We have a function in NextFlow that takes `params.cellranger` and creates the CLI that corresponds to it. Values for `input` and `output` are set during the pipeline run based on the input provided and a closure for generating the output file name. To give an idea of what this CLI rendering looks like, this is what we use in DiFlow:

A `nextflow.config` file with content like above is created for every *module*, i.e., for every processing step in the pipeline. On the level of the pipeline

those config files are sourced, i.e.:

```
includeConfig '<...>/nextflow.config'
```

It may seem like a daunting task to create a config file like this for every computational step, and it is. We are not doing this manually as that would be too error-prone and frustrating on top of that. We are just laying out the principles here, later we will see how `viash` can create the `nextflow.config` file for us.

Transforming the relevant section in `nextflow.config` to a command-line instruction is done by a Groovy function that simply parses the `ConfigMap`.

Step 23 - More than one input

It may appear in the above example that input can only be provided as one variable or file. NextFlow allows you to specify not only wildcards (`*` for instance) but also arrays. This comes in handy when multiple input file reference need to be provided on the CLI. For instance, when doing a mapping step we usually need to provide a reference file. We could add this reference file as a parameter (and take it up in `nextflow.config` but then it would just be seen as a string value. NextFlow can not know then that it has to check for existence of the file prior to starting to run the process[21].

[21] There is more in fact: NextFlow has some logic on how to deal with input files/directories when using Docker. It will mount the volumes that contain input references. If we just add an input file reference as a string variable to the CLI, it will not be visible inside the Docker container at runtime.

How does a DiFlow module know then what file reference is related to what option on the CLI? We would obviously not want to run `CellRanger count` on a reference file as input and use a `fastq` file as reference (as if that would work?!). That means we have to be sure to somehow let the DiFlow module know what file references correspond to what options on the CLI.

There are three possibilities:

1. There is only one input file: in this case we just have to make sure it is passed as a `Path` object to the DIFlow module.
2. There are multiple input files but they correspond to the same option on the CLI. For instance, `cat`'ing multiple files where order is not relevant. In this case, we can simply pass a `List[Path]` to the DiFlow module.
3. There are multiple input files corresponding to the different options on the CLI, for instance CellRanger with input and a reference file. In this case, we pass something like

```
[ "input": "<fastq-dir>", "reference": "<reference-dir>" ]
```

While there may be still other possibilities that we may encounter in the future, these three are covered by the current implementation of DiFlow.

> A DiFlow module contains the necessary logic to parse three types of datastructures as input file references: `Path`, `List[Path]` and `Map[String, List[Path]]`.

Remark: that the easiest way to create a `Path` object with the proper pointers in a NextFlow context is to use the built-in `file()` function. It simply takes a `String` argument pointing to the file (either relative or absolute).

There's some more magic going on in the background. For starters, if you specify just either `Path` or `List[Path]`, the DiFlow module will retrieve the

appropriate command-line option to associate it with *automagically*. This way, as a pipeline developer you usually should not care about what exact command line option is necessary for your input data to be processed.

Step 24 - `workflow` instead of `process`

We already have quite some helper functionality that should be provided by a DiFlow module:

- Generating an output file name based on input
- Parsing different types of input file references
- Selecting the proper key from the (large) `ConfigMap` stored in the global `params Map`.
- Generating the CLI from `nextflow.config`
- Providing a test case for the module

There is some hidden functionality as well:

- Making sure input and output file references are updated in the `ConfigMap`
- Dealing with per-sample configuration (upcoming feature)

As it turns out, providing all this functionality in a `process` is not the proper way to go, and is even expected not to work. Luckily we can define `workflow`s in NextFlow's DSL2 syntax. Such a `workflow` can be used just like a `process` in the above example as long as we take care that the input/output signatures are aligned with what is expected.

The added benefit of using a `workflow` rather than a `process` is that the underlying `process` can have a different signature. In practice, this is what a DiFlow `process` looks like:

```
process cellranger_process {
  ...
  container "${params.dockerPrefix}${container}"
  publishDir "${params.output}/processed_data/${id}/", mode: 'copy', overwrite: true
  input:
    tuple val(id), path(input), val(output), val(container), val(cli)
  output:
    tuple val("${id}"), path("${output}")
  script:
    """
    export PATH="${moduleDir}:\$PATH"
    $cli
    """
}
```

As can be seen, the `ConfigMap` is not passed to the `process` but instead the information in `params` is used to generate an `output` filename, extract the container image and generate the CLI instruction. Please note that `input` here points to ALL possible input file references as per Step 23.

The `workflow` that points to this `process` is then defined as follows:

```
workflow cellranger {
  take:
    id_input_params_
  main:
    ...
    result_ =  ...
  emit:
    result_
}
```

Step 25 - Custom scripts

The attentive reader may have noticed the instruction in the `script` section in Step 24. It adds the directory of the current *module* to the `$PATH`. This allows us to store scripts or binaries with the NextFlow module and still be able to call those, even if NextFlow runs all processes from their own private `work` directory.

Step 26 - The missing link

Creating and maintaining all the necessary files for a modules, especially with the amount of (duplicate) boilerplate code in each of them may seem like a daunting task. It is… Therefore, we developed a tool that is able to add the boilerplate for us: `viash`.

`viash`[22] takes as input a specification of a command/tool, how to run and test it. `viash` can then turn this specification into runnable script, a containerized executable or a NextFlow module.

[22] To%20be%20open-sourced%20soon!

Putting it all together

In the end, a *module* consists of the following:

- `main.nf` contains the code for `workflow` and `process` definition. We duplicate ALL the parsing code (for CLI, input, `ConfigMap`, etc.) in order for a module to be effectively standalone.
- `nextflow.config` contains the `ConfigMap` for this specific module, scoped properly.
- Executables or scripts required to be on the `$PATH` for this module to run inside the container defined in `nextflow.config`.

In what follows, we will point to an example pipeline in viash_docs[23]. This repository contains the source files needed to *generate* the DiFlow modules.

[23] https://github.com/data-intuitive/viash_docs/blob/master/examples/civ6_postgame/main.nf

Creating a pipeline from these modules is now a matter of:

Generate the modules

Using `viash`, it's easy to go from the component definitions under `src/` to proper DiFlow modules:

```
viash ns build -p docker --setup
viash ns build -p nextflow
```

The first instruction builds the Docker containers needed for the pipeline to work. The second one builds the NextFlow/DiFlow modules.

Please note that `viash` allows you to also export to native or containerized binaries as well as run unit tests for the components. This, though, is covered elsewhere.

Pipeline `main.nf`

The pipeline logic is contained in `main.nf`. In order to use the modules defined using DiFlow, they have to be imported. This is the full `main.nf` file for the civ6_postgame pipeline:

```
nextflow.preview.dsl=2

import java.nio.file.Paths

include  plot_map       from  './target/nextflow/civ6_save_renderer/plot_map/main.nf'       params(params)
include  combine_plots  from  './target/nextflow/civ6_save_renderer/combine_plots/main.nf'  params(params)
include  convert_plot   from  './target/nextflow/civ6_save_renderer/convert_plot/main.nf'   params(params)
include  parse_header   from  './target/nextflow/civ6_save_renderer/parse_header/main.nf'   params(params)
include  parse_map      from  './target/nextflow/civ6_save_renderer/parse_map/main.nf'      params(params)
include  rename         from  './src/utils.nf'

workflow {

    if (params.debug == true)
        println(params)

    if (!params.containsKey("input") || params.input == "") {
        exit 1, "ERROR: Please provide a --input parameter pointing to .Civ6Save file(s)"
    }

    def input_ = Channel.fromPath(params.input)

    def listToTriplet = { it -> [ "all", it.collect{ a -> a[1] }, params ] }

    input_ \
        | map{ it -> [ it.baseName , it ] } \
        | map{ it -> [ it[0] , it[1], params ] } \
        | ( parse_header & parse_map ) \
        | join \
        | map{ id, parse_headerOut, params1, parse_mapOut, params2 ->
            [ id, [ "yaml" : parse_headerOut, "tsv": parse_mapOut ], params1 ] } \
        | plot_map \
        | convert_plot \
        | rename \
        | toSortedList{ a,b -> a[0] <=> b[0] }  \
        | map( listToTriplet ) \
        | combine_plots

}
```

Given the steps described above, we estimate that it's possible to understand this pipeline.

Pipeline `nextflow.config`

This is the config file for the pipeline:

```
includeConfig 'target/nextflow/civ6_save_renderer/plot_map/nextflow.config'
includeConfig 'target/nextflow/civ6_save_renderer/combine_plots/nextflow.config'
includeConfig 'target/nextflow/civ6_save_renderer/convert_plot/nextflow.config'
includeConfig 'target/nextflow/civ6_save_renderer/parse_header/nextflow.config'
includeConfig 'target/nextflow/civ6_save_renderer/parse_map/nextflow.config'
```

```
docker {
  runOptions = "-i -v ${baseDir}:${baseDir}"
}
```

Running the pipeline

```
> nextflow run . \
  --input "data/*.Civ6Save" \
  --output "output/" \
  --combine_plots__framerate 1 \

N E X T F L O W  ~  version 20.10.0
Launching `./main.nf` [serene_mercator] - revision: 86da0cc3ec
executor >  local (26)
[2c/970402] process > parse_header:parse_header_process (AutoSave_0158) [100%] 5 of 5 ✔
[7d/c19cfa] process > parse_map:parse_map_process (AutoSave_0162)       [100%] 5 of 5 ✔
[06/4b19be] process > plot_map:plot_map_process (AutoSave_0160)         [100%] 5 of 5 ✔
[fc/3f219c] process > convert_plot:convert_plot_process (AutoSave_0162) [100%] 5 of 5 ✔
[f2/c24399] process > rename (AutoSave_0162)                            [100%] 5 of 5 ✔
[fb/43a707] process > combine_plots:combine_plots_process (all)         [100%] 1 of 1 ✔
```

Please note that we use an option `--combine_plots__framerate 1`. This points to an option of the `combine_plots` module that is called `framerate`. In other words, if a module defines an option (corresponding to a CLI option) it can be overridden from the CLI by using the convention `<module_name>__<module option> <value>`[24].

[24] See below for more information about this and how this is encoded in `nextflow.config`.

What is missing from DiFlow?

Parameter checks

We currently do no checks on variables that are set through the `ConfigMap`.

Multiple output file references

We make an implicit assumption all of the above that the output of a pipeline step is a single file, or a single directory or a set of files that relate to each other. What we don't cover yet is a component that outputs both the results of analysis and a report, for instance. The reason for this is not technical but rather that it breaks the logical flow of a pipeline definition.

We currently output 1 *type* of output from a module, and that allows us to easily chain modules. A module that would output 2 types of data would essentially be a fork in the pipeline process. That means that either the next component knows to expect these two outputs as inputs or we have to explicitly deal with the two branches of the fork. But then we can not longer write pipelines like:

```
step1 | step2 | step3
```

There is a simple workaround for this kind of situation: Make two components/modules that each output either of the outputs. This fits in the API of a modules and those can again be chained easily. Having said that, we are thinking of ways to allow the output of multiple output files because this workaround is not efficient at all.

Per-sample configuration

When running different samples in parallel, one may sometimes want to have parameters specific to a sample. Filter threshold, for instance, may be different from sample to sample. Implementing this in DiFlow is not hard per se, it is more a matter of coming up with a good way to encode this in `nextflow.config` and the `ConfigMap`.

Appendix

Variables in `nextflow.config`

CLI arguments and options for specific components/steps in the pipeline are configured in the respective `nextflow.config` files that are imported in the global one. But that also means that we can not override them from the CLI anymore. For instance, it is not possible to add an argument in the style `--component.input.value <value>`. That means that all options would either be fixed for the whole pipeline, or to be configured in `nextflow.config` explicitly. The latter is possible by means of a custom config file that overrides the other settings.

There is, however, an easier approach to this: variables in `nextflow.config`. The functionality is explained here[25] and is used in DiFlow for allowing us to provide the (scoped) parameter values on the CLI. For instance, this is an excerpt from an existing component's `nextflow.config`:

[25] https://www.nextflow.io/docs/latest/config.html#config-variables

```
params {
  ...
  cellranger_vdj__id = "cr"
  ...
  cellranger_vdj {
    name = "cellranger_vdj"
    container = "mapping/cellranger_vdj/cellranger_vdj:4.0.0-1"
    command = "cellranger_vdj"
    arguments {
      ...
      id {
        name = "id"
        otype = "--"
        value = "${params.cellranger_vdj__id}"
        ...
      }
      ...
```

If you look at this one parameter for the `cellranger_vdj` component, you notice that directly under `params`, we have the key `cellranger_vdj__id`. In other words, the component name followed by a double underscore and then the parameter name. Since all arguments are named here, we do not have to have to specify -'s or --'s.

Reasons for an explicit *flow*

In DiFlow, we do not allow multiple input `Channel`s, but rather make the flow of data explicit by means of the available `Channel` operators. There are a few reasons for this:

1. It's easier to use a consistent API for modules, so that we don't need to *know* how to call a module
2. This makes for cleaner view on what a pipeline does by looking at the pipeline code
3. The asynchronous nature of the computations may cause inconsistencies

Let us illustrate the latter point a bit more in detail. In what follows we define the *same* process in two scenarios: once where we allow two input process and once where we define the flow explicitly using the `join` operator.

First the process that takes two inputs:

```
Such that
```

```sh
> nextflow -q run . -entry join_process -with-dag figures/join_process.png
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[[1, 11/d], [2, 21/c], [3, 31/b], [4, 41/a]]
```

The second implementation *seems* to do the same:

```
Such that
```

```sh
> nextflow -q run . -entry join_stream -with-dag figures/join_stream.png
WARN: DSL 2 IS AN EXPERIMENTAL FEATURE UNDER DEVELOPMENT -- SYNTAX MAY CHANGE IN FUTURE RELEASE
[[1, 11a], [2, 21b], [3, 31c], [4, 41d]]
```

Resources

When you run or export with the `DockerTarget`, resources are automatically added to the running container and stored under `/resources`. In case of the `NativeTarget`, this is not the case and since `NextFlowTarget` uses the `NativeTarget` it's the same there. That does not mean that resources specified in `functionality.yaml` are not available in these cases, we only have to point to them where appropriate.

The following snippet (from `ct/singler`) illustrates this:

```
par = list(
  input = "input.h5ad",
  output = "output.h5ad",
  reference = "HPCA",
  outputField = "cellType",
  pruningMADS = 3,
  outputFieldPruned = "celltype-pruned",
  reportOutputPath = "report.md"
)
par$resources_dir <- resources_dir
```

In other words, `resources_dir` is automatically created by `viash` in all current 3 environments. This means that we can point to the `report.Rmd` file present in the resources like so:

```
rmarkdown::render(paste0(par$resources_dir, "/", "report.Rmd"), output_file = par$reportOutputPath)
```

Default values

In the viash `functionality` spec, no option should have an empty string as value!