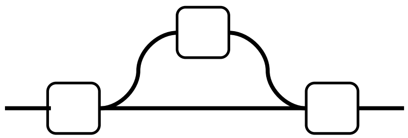
Introduction to JINT Functional Pipeline



*'child is my brother' problem

Kiyoung Kim

grasshoppertrainer@gmail.com

Just In Need, Time Functional Pipeline(JINTFP) is a pattern and framework that combines best of object oriented and functional programing patterns, which makes creating efficient functional pipeline easy.

Table of contents

Concept

Node

Node is a cell which processes input to export output. By using Node core, developer can define a concrete class with other attributes to aid, and control calculation.

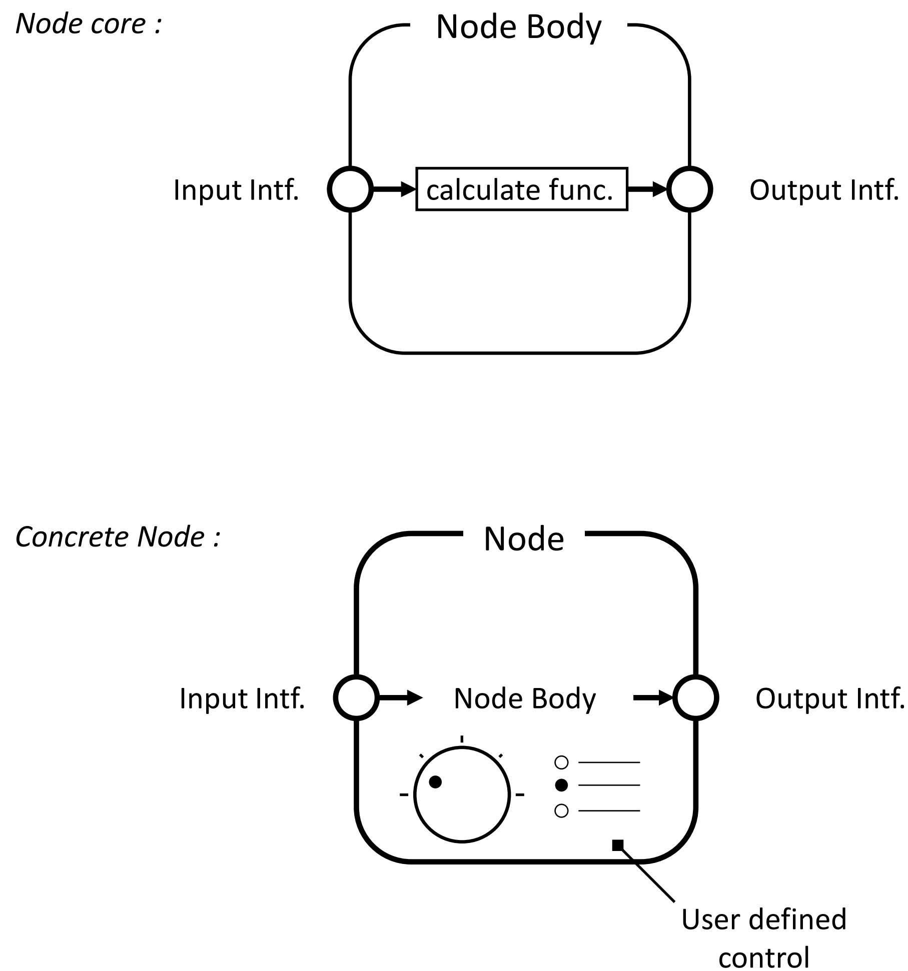


fig1. Structure of Node

Graph

JINTFP consists of a set of **Node**s and relationship among them, which can be perceived as a **none-binary graph**. This graph ensures unidirectional data flow; from upstream to downstream.

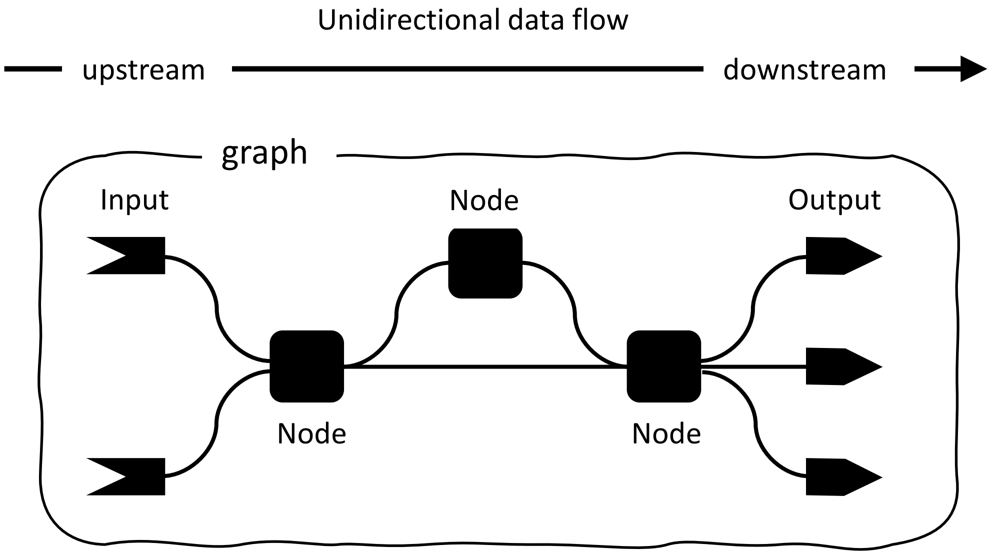


fig2. Structure of Node

Essence of JINTFP

As a derivative of JIT, JINT stands for Just In Need, Time. Not only JINTFP executes calculation only when output is asked, but with only necessary sub-calculation.

1. Just In Time calculation

Output is not calculated when its built nor when new input is set. It is calculated only when it is actually asked. Recalculation stage will recalculate up-to-date output and then pipeline goes back to idle stage. This way, calculation can be executed only once in spite of multiple input set at different moment of runtime.

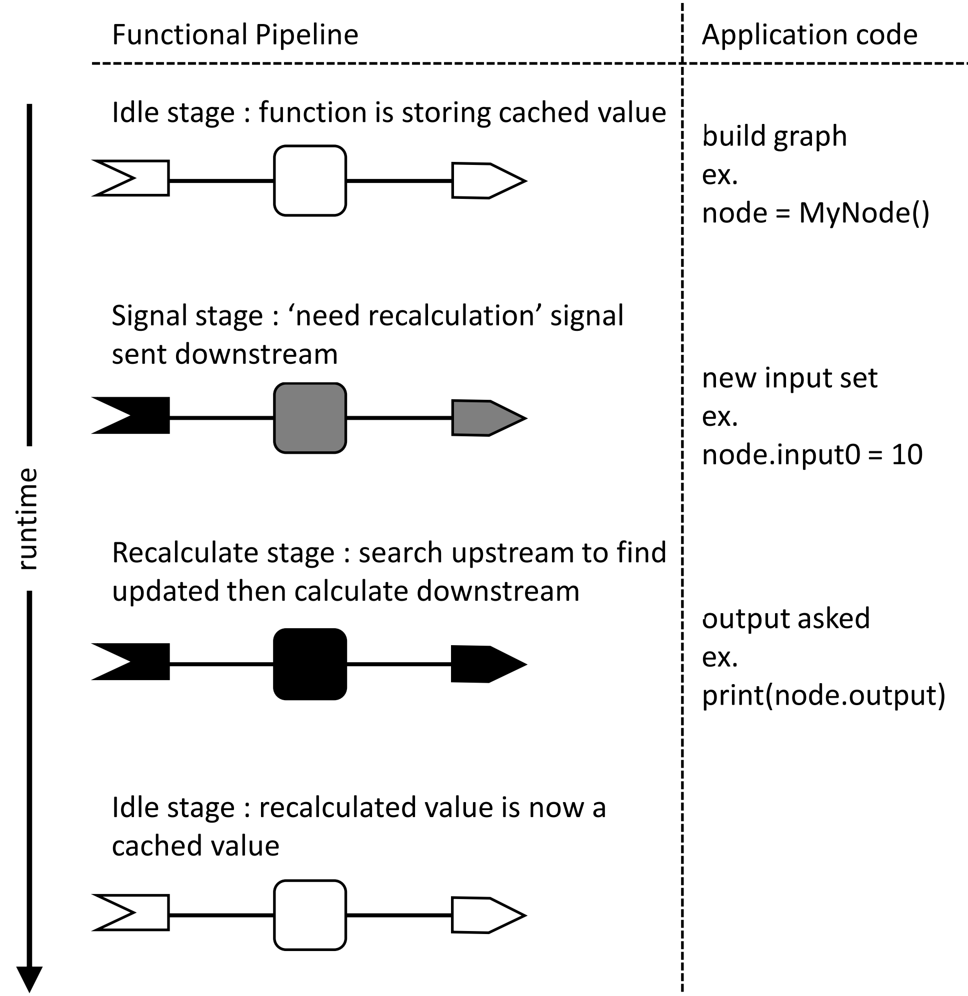


fig3. JIT in runtime

Essence of JINTFP

2. Just In Need calculation

JINTFP is porous pipeline; meaning graph can have multiple input and output at any part, depth. So, in JINTFP, output is not dependent to whole graph. JINTFP graph is able to search sub-graph for calculating specific output. Meaningless calculation is drastically removed this way.

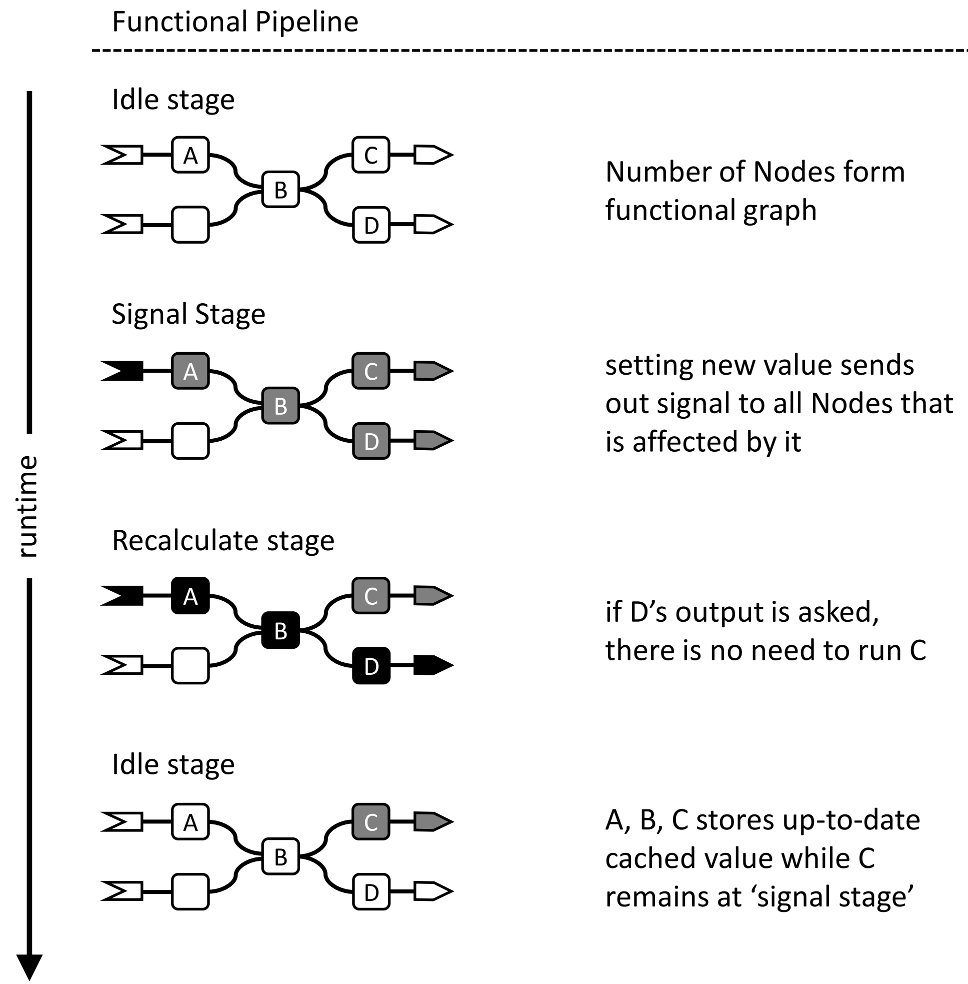


fig4. JIN in runtime

Essence of JINTFP

Functional pipeline is a set of fixed execution that produces outputs from inputs. JINT Functional Pipeline can be modified in runtime. Moreover, number and when to put in input values has no limitation. This characteristics is also true with outputs, making JINT Functional Pipeline highly flexible and reusable.

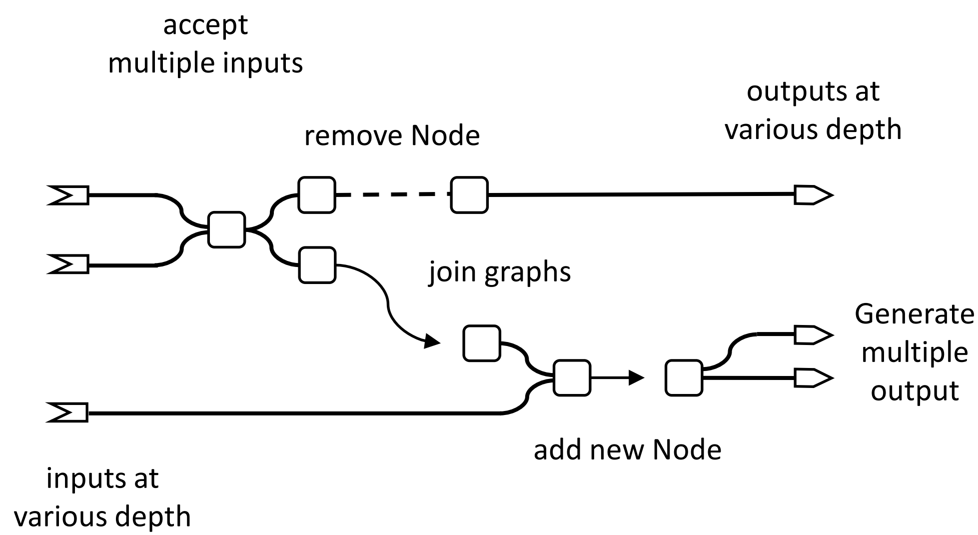


fig5. Porous, editable pipeline

Characteristics Summery

1. Build a pipeline as you picture in mind

User doesn’t have to build a graph from top to bottom. As imagination expands graph can be expanded back and forth. Only by few means; removing or adding Node, develop pictured pipeline into working code.

2. caching, JINT efficiency

Using cached value prevents duplicated calculation. Once output is calculated, there is no cost calling it again before another update is made.

3. Monomorphic abstraction

Node that runs other Nodes inside its calculation is a compound node. Compound node acts the same as a simple node, which makes a pipeline to be structured with multiple level of abstraction.

4. Possibility of multi processing

All Nodes are subclass of parent class that defines meta-function and interface. As so, multiprocessing can be implemented inside recalculation meta-function out of actual Nodes’ calculations

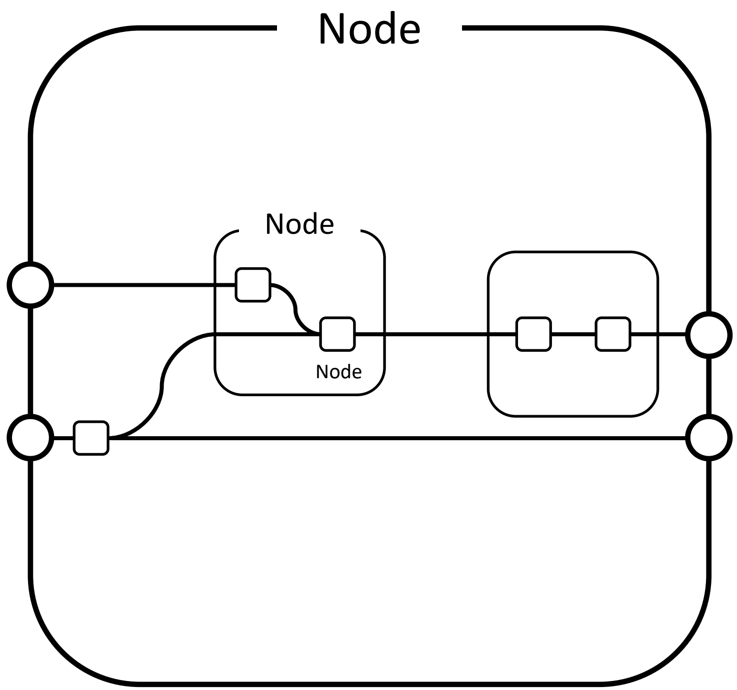


fig6. Compound Node – monomorphic abstraction

Implementation

Structure

Input Interface, Node Body and Output Interface forms ‘Node’. These three partition responsibilities of Node; what and how to process value. They are all subclasses of `NodeMember`. This design decision was made not only to use none-binary graph to maintain relationship between Nodes, but within Node’s components too. Thus making calculating process concise.

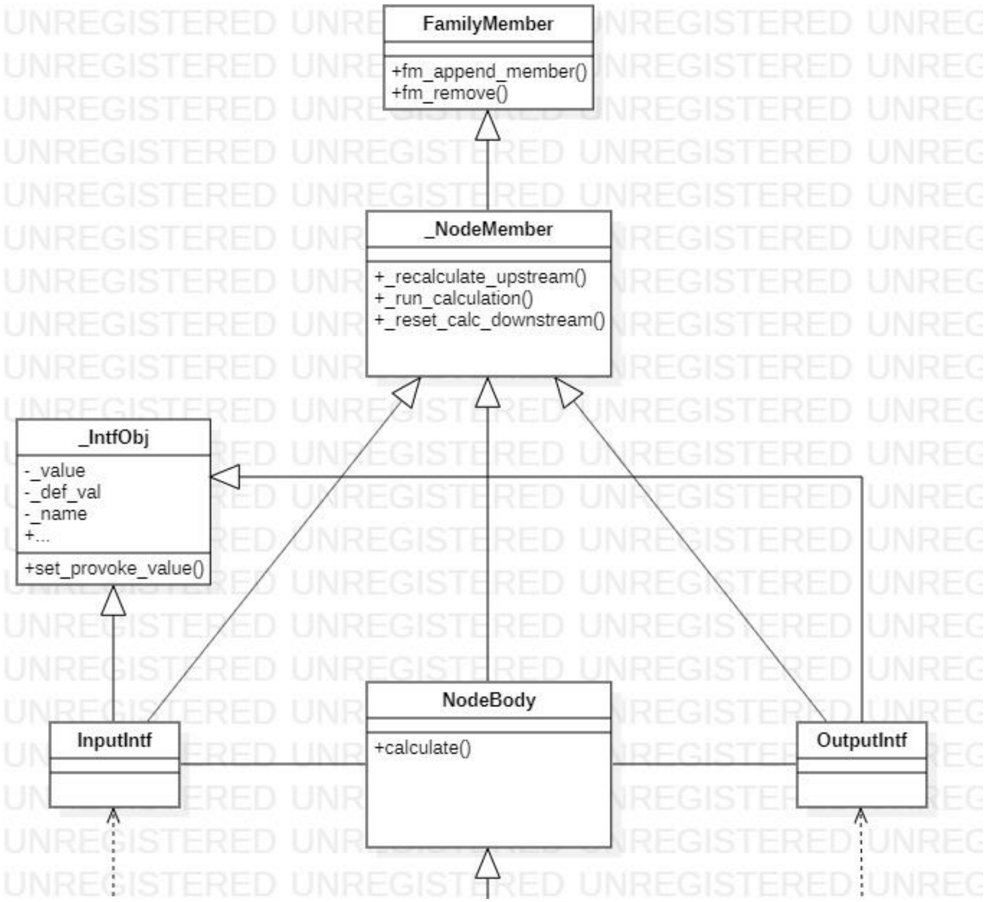


fig7. UML diagram of JINTFP

User interface

`Input` and `Output` is a Descriptors for each type of interface object. `_IntfObj` is the one that holds cached value, while `_IntfDescriptor` is a means for structuring `Concrete  Node`. Nodes communicates with each other via `_IntfObj`. `_IntfDescriptor` types are for developer(user) to control relationship between Nodes.

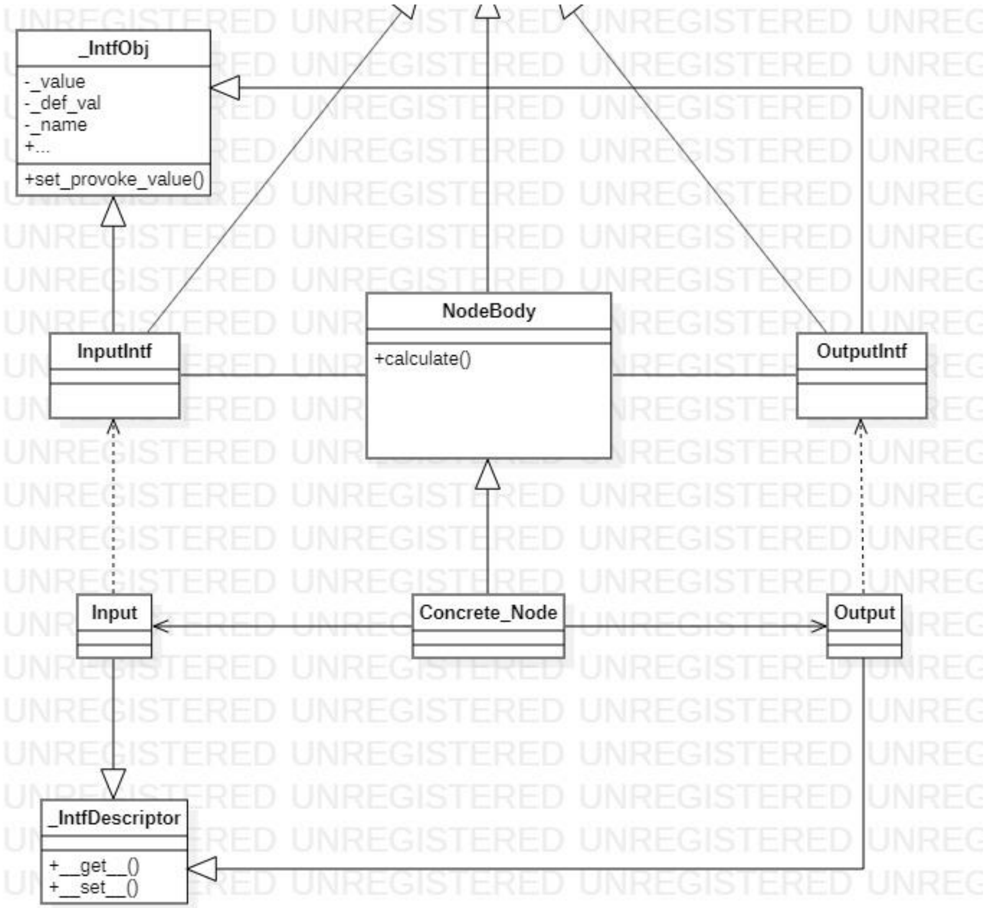


fig7. UML diagram of JINTFP

User Interface __set__

1. Simply assign value into an interface. It will reset ‘calculated’ signal all the way to the leaf of the graph.

If given value is another interface, new relationship between this interface and given interface will be stored.

Input and Output behaves slightly different to maintain graph unidirectional.

```
# assignment will call descriprot's `__set__`
# then Interface buffer's `set_provoke_value`
# then `_signal_downstream`
some_node.some_input = another_node.some_output

class _IntfDescriptor:
    ...
    def __set__(self, instance: NodeBody, value):
        # retrive stored interface buffer
        intf = instance.get_intf(self._name)
        intf.set_provoke_value(value)
    ...


class _InputBffr(_IntfBffr):
    ...
    def set_provoke_value(self, value):
        """
        Set interface value

        whilst building relationship and resetting calculated flag
        :param value:
        :return:
        """
        super().set_provoke_value(value)
        # by default relationship is monopoly so clear
        self.fm_clear_parent()
        # make relationship
        if isinstance(value, _IntfBffr):
            # set relationship between interface
            self.fm_append_member(parent=value, child=self)
```

```
        value = value._value
    self._value = value
    self._signal_downstream()

def _signal_downstream(self, visited=None, debug=''):
    """
    Reset children's calculated sign
    :return:
    """
if visited == None:
    visited = set()
if self._is_calculated():
    self._reset_calculated()
    for child in self.fm_all_children():
        if child not in visited:
            visited.add(child)
            child.reset_downstream(visited, debug + ' ' * 4)
...
```

User Interface __get__

Getting value is a bit more complicated than setting as it involves executing Nodes' calculation in correct order.

1. Use DFS algorithm to search upstream until calculated NodeMember(Interface or node body) is met. 'child is my brother' problem can be avoided this way.

```
class _NodeMember(FamilyMember):
    ...
    def _recalculate_upstream(self, _visited=None, debug=''):
        """
        recalculated upstream to get up to date result
        """
        if _visited is None: # initiate recursion
            _visited = set()
        if self._is_calculated(): # base condition
            return True
        is_parent_recalculated = True # flag for checking permanent recalculation
        # recursivly calculate upstream before calculating current
        for member in self.fm_all_parents():
            if not isinstance(member, _NodeMember):
                continue
            if member not in _visited:
                _visited.add(member)
                is_parent_recalculated &= member._recalculate_upstream(_visited,
debug+' '*4)
        self._set_calculated()
        self._run_calculation()

        # if this node is set to permanently recalculate
        # or one of parent is set so, pass this signal downstream
        if self._calculate_permanent or not is_parent_recalculated:
            self._reset_calculated()
        return self._is_calculated()
    ...
```

User Interface __get__

2. After upstream search is done, run calculation function.

3.After calculation, push values downstream.

Please pay attention how `NodeBody`, `InputBffr` and `OutputBffr` override `run_calculation` function differently.

```
class NodeBody(_NodeMember):
  ...
  def _run_calculation(self):
    """
    Execute concrete function and push value downstream
    """
    # collect input
    input_vs = OrderedDict()
    for intf in self.input_intfs:
      if intf.sibling_intf_allowed:
        input_vs.setdefault(intf.family_name,
[]).append(intf.get_calculated_value())
      else:
        input_vs[intf] = intf.get_calculated_value()
    try:
      results = self.calculate(*input_vs.values()) # run concrete function

    except Exception as e:
      results = [NullValue(f"calculation fail of {self}")] * len(self.output_intfs)
      # record and print error status
      self._calculation_status = e
      self.print_status()
    else:
      results = [results] if not isinstance(results, (list, tuple)) else list(results)
      results += [NullValue("not enough value")] * (len(self.output_intfs) -
len(results))
      self._calculation_status = ''
    finally:
      for result, intf in zip(results, self.output_intfs): # push result downwards
        intf.set_provoke_value(result)
```

```python
class _InputBffr(_IntfBffr):
    ...
    def _run_calculation(self):
        """
        Nothing to do
        """
        pass
    ...


class _OutputBffr(_IntfBffr):
    ...
    def _run_calculation(self):
        """
        Just push value downstream
        """
        for child in self.fm_all_children():
            if isinstance(child, _IntfBffr):
                child._value = self._value
    ...
```

Integrated workflow example

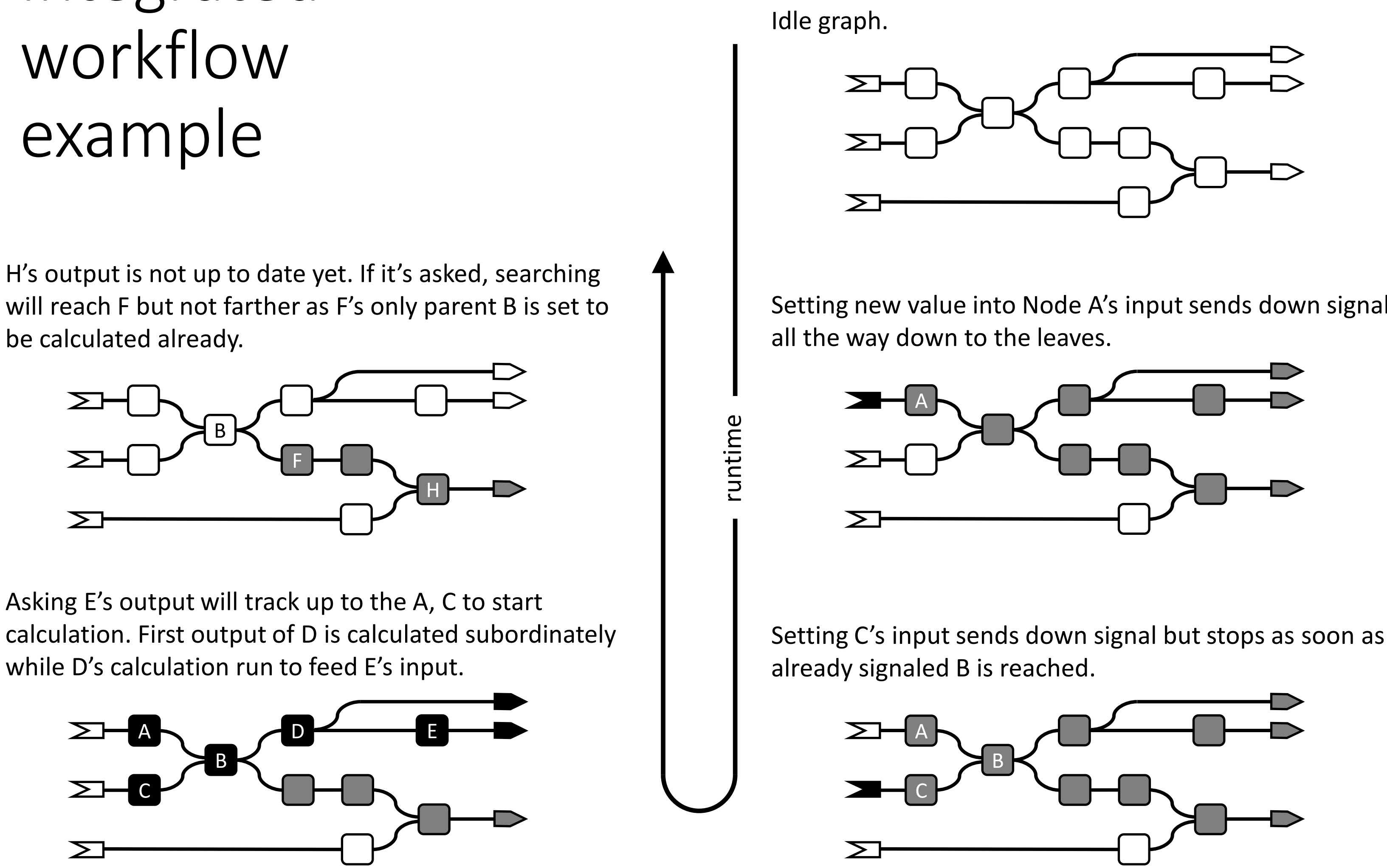


fig8. integrated workflow

Beginner’s User Guide

Inherit to create your own Node

Simply inherit `NodeBody` to create concrete Node. Add interfaces by assigning class attribute.

```
from JINTFP import NodeBody, Input, Output


class MyNode(NodeBody):
    # input descr. will only accept instance of `typs`
    in0 = Input(def_val='', typs=str)
    in1 = Input(def_val='', typs=str)

    out = Output()

    def __init__(self):
        # NodeBody's `__init__` has to be called
        # to initiate instance as a Node
        super().__init__()
        self._do_upper = False

    def calculate(self, in0, in1):
        # user should provide matching number of
        # attribute to use them in calculation
        if self._do_upper:
            return (in0 + in1).upper()
        return in0 + in1

    def set_upper(self):
        # except inheriting and initiating,
        # concrete Node acts the same as any other class
        # user can extend its functionality as they wish to
        self._do_upper = True
```

Build Graph with Assignment

If another Node’s interface is given to assign with, Node will automatically build relationship with it, meaning building graph. If else value is given, Node will simply cache that value.

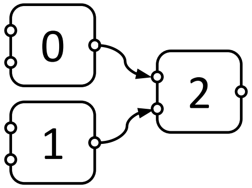
```
node0 = MyNode(0)
node1 = MyNode(1)
node2 = MyNode(2)
print('>>> building graph')
node2.in0 = node0.out
node2.in1 = node1.out
```

console :

```
>>> building graph
RUNNING Node : 0
RUNNING Node : 1
```

Pay attention to ‘RUNNING’ log. Outputs had to be refreshed while assigning it into ‘node2’s inputs

current graph :



Complicate Graph

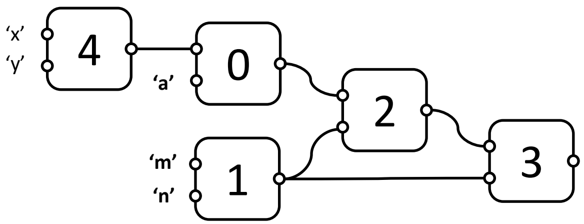
Make graph none-binary. Put inputs whilst building the graph. Ask for intermediate, and end outputs at any time you wish.

```
...
print('>>> growing graph')
node3, node4 = MyNode(3), MyNode(4)
node3.in0, node3.in1 = node2.out, node1.out
node0.in0 = node4.out
# feeding inputs
node4.in0, node4.in1 = 'x', 'y'
node0.in1 = 'a'
node1.in0, node1.in1 = 'm', 'n'
```

console :

```
>>> growing graph
RUNNING Node : 2
RUNNING Node : 4
```

current graph :



Get up to date output

Simply call output. Graph will run lazy calculation.

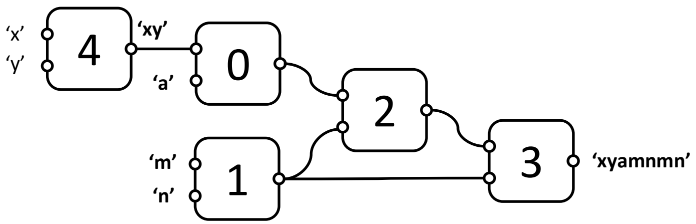
```
...
print('>>> getting outputs')
print(node3.out, node4.out)
```

console :

```
>>> getting outputs
RUNNING Node : 4
RUNNING Node : 0
RUNNING Node : 1
RUNNING Node : 2
RUNNING Node : 3
<output_intf 'out' : xyamnmn> <output_intf 'out' : xy>
```

Calculation is run in correct order despite ‘child is my brother’ problem, and not run when value is already cached.

current graph :



Tweak calculation

Interact with node like one would do with a class instance.

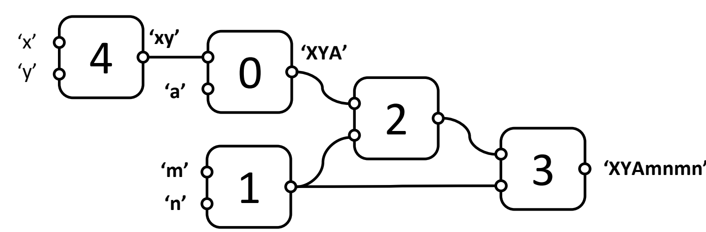
```
...
print('>>> tweaking calculation')
node0.set_upper()
print('>>> getting unaffected')
print(node1.out)
print('>>> getting affected')
print(node3.out)
```

console :

```
>>> tweaking calculation
>>> getting unaffected
<output_intf 'out' : mn>
>>> getting affected
RUNNING Node : 0
RUNNING Node : 2
RUNNING Node : 3
<output_intf 'out' : XYAmnmn>
```

No calculation is called when getting unaffected Node’s output.

current graph :



Carry on

No need to bother casting output into wrapped type. Simply call real value and use it as you wish.

```
...
print('>>> getting real value')
print(node3.out.r, type(node3.out.r))
```

‘r’ for ‘real’

console :

```
>>> getting real value
XYAmnmn <class 'str'>
```

Roadmap

JINTF’s real power is revealed when using it to constantly process mass data of same kind; like geometry processing for interactive visualization.

Please be updated with newest package :

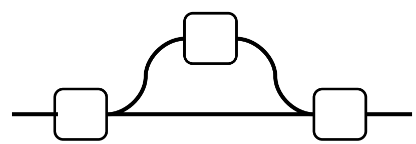
v 0.1.0   initial publication

v 0.2.0   update on parallel calculation

v 0.3.0   update on multi threading, multi processing

V 1.0.0   alpha lunch

Thank you.



Kiyoung Kim

grasshoppertrainer@gmail.com