# Advanced Python Features Gone Bad

Clément Rouault — Franck Michea
hakril@lse.epita.fr — kushou@lse.epita.fr

**LSE**, EPITA Systems Lab.
http://lse.epita.fr/

# Who's there?

- Clément Rouault
- hakril@lse.epita.fr
- twitter: hakril

- Franck Michea
- kushou@lse.epita.fr
- twitter: kushou_

- Both GISTRE and LSE 2014.
- Interested in Python and Security.

# What this talk is about.

- How to write horrible code with style.
- Using advanced features the wrong way to understand how they work.
- We won't directly talk about inner workings of the interpreter a lot, but some of the points are closely related to it.
- **Note:** Even though these technics could be used to obfuscate code, we didn't write this talk with it in head. We were mostly just playing with the language!

# Table of Contents

# Simple function call

```
1  def f(a, b, c):
2      return (a, b, c)
3
4  f(1, 2, 3)           # (1, 2, 3)
5
6  f(c=3, a=1, b=2)     # (1, 2, 3)
```

# defaults arguments

- func_defaults

```
1  def f(a, b=0, c=42):
2      return (a, b, c)
3
4  f(11)                # (11, 0, 42)
5
6  f.func_defaults      # (0, 42)
```

# Reference Hell

```
1  def f(a, c=[]):
2      c.append(a)
3      return (a, c)
4
5  print f.func_defaults            # ([],)
6
7  print f(42)                      # (42, [42])
8
9  print f.func_defaults            # ([42],)
10
11 print f("Test.")                 # ('Test.', [42, 'Test.'])
12
13 print f.func_defaults            # ([42, 'Test.'],)
```

# Writable attribute

Let the magic append!

```
1  def f(a, b = "STR"):
2      return (str(a) + " : " + str(b))
3
4  f()      # TypeError: f() takes at least 1 argument (0 given)
5
6  f(1)     # "1 : STR"
7
8  f.func_defaults = ("NOP",)
9  f(42)    # "1 : NOP"
```

# MORE!

Why stop here?

```
1  def f(a, b = "STR"):
2      return (str(a) + " : " + str(b))
3
4  f(1)     # 1 : STR
5
6  f.func_defaults = ()
7  f(1)     # TypeError: f() takes at least 2 argument (1 given)
8
9  f.func_defaults = ("NOP", 42)
10 f()      # "NOP : 42"
```

# pause closure

```
1  def plusn(n):
2      def plus(x):
3          return x + n
4      return plus
5
6  plus_1 = plusn(1)
7  print plus_1.func_closure
8  # (<cell at 0x7fd50074c788: int object at 0xa34c68>,)
9  print plus_1.func_closure[0].cell_contents
10 # 1
```

# Base of late binding

```
1  v = 54
2  gen = (v for _ in range(1000))
3
4  print next(gen)          # 54
5
6  v = "POP"
7  print next(gen)          # "POP"
8
9  print next(gen)          # "POP"
10
11 v = "RET"
12 print next(gen)          # "RET"
```

# Warning!

```
1  v = "Test."
2  gen = (v for _ in range(1000))
3
4  def mprint(g, v):
5      print next(g)
6
7  mprint(gen, "FAKE")      # "Test."
```

# Tricks on late binding!

```
1  ll = [[x + y for x in 'AB'] for y in '12']
2  ll2 = [(x + y for x in 'AB') for y in '12']
3
4  for l, l2 in zip(ll, ll2):
5      for t, t2 in zip(l, l2):
6          print t, 'vs.', t2
```

# Solution

```
1  ll = [[x + y for x in 'AB'] for y in '12']
2  ll2 = [(x + y for x in 'AB') for y in '12']
3
4  for l, l2 in zip(ll, ll2):
5      for t, t2 in zip(l, l2):
6          print t, 'vs.', t2
7
8  # A1 vs. A2
9  # B1 vs. B2
10 # A2 vs. A2
11 # B2 vs. B2
```

# New type at runtime

```
1  class A(object):
2      pass
3
4  class B(object):
5      def f(self):
6          print "Call on f for {0}".format(self)
7
8  a = A()
9  a.f()    # AttributeError: 'A' object has no attribute 'f'
10
11 a.__class__ = B
12 a.f()    # Call on f for <__main__.B object at 0x7ffb659c4690>
```

# New type for type!

```
1  class Meta(type):
2      pass
3
4  class OtherMeta(type):
5      def __call__(self):
6          print "META CALL"
7          return 42
8
9  class A(object):
10     __metaclass__ = Meta
11
12 print A()
13 # <__main__.A object at 0x7fe477a92690>
14
15 A.__class__ = OtherMeta
16 print A()
17 # META CALL
18 # 42
```

# Control-Flow Manipulation Introduction

```
1  static long
2  int_hash(PyIntObject *v)
3  {
4      /* XXX If this is changed, you also need to change the way
5         Python's long, float and complex types are hashed. */
6      long x = v -> ob_ival;
7      if (x == -1)
8          x = -2;
9      return x;
10 }
```

```
1  if hash(-2) == hash(-1):
2      print(':)')
```

# Easiest way to write unreadable code...

- Doesn't only apply to dynamic languages; can be done with languages where operator overloading is possible.
- Biggest limitation is probably what allows the syntax of your language and the number of operators you can override.
- You obviously need to give out the code of your VM, though you then can encode your whole code.

# What we used

- You will mostly be restrained by the syntax and the number of operators you will be able to overload.
- Operator overloading is done in python using special functions:
  - Arithmetics are done with `__add__` and `__sub__`
  - You can play with [] by overloading `__getitem__`
  - **dot** can be with properties, and **comma** by translating tuples.
  - `__neg__` and `__pos__` were useful too.
  - You can return objects from comparisons.
- You can't write exactly what you want. Syntax + what applies on what.

# Code example

```
1  class B:
2      def __init__(self, *args):
3          self._actions = []
4          for other in args:
5              self._actions.extend(other._actions)
6              self._actions.append(input_op)
7          if self._actions:
8              self._actions.pop()
9
10     def __neg__(self):
11         self._actions.insert(0, decd_op)
12         return self
13
14     def __pos__(self):
15         self._actions.insert(0, incd_op)
16         return self
```

# Code example

```
1       def __getitem__(self, item):
2           if isinstance(item, tuple):
3               c = B(*item)
4           else:
5               c = B(item)
6           self._actions.append(while_op(list(c._actions)))
7           return self
```

# Example with brainfuck

```
1  b = B(++++++++++B()[B()>+++++++B()>++++++++++B()>+++B()>+B() \
2      <B()<B()<B()<-B()]+-B()>++B()._+-B()>+B()._+++++++ \
3      B()._+-B()._+-+++B()._+-B()>++B()._+-B()<B()< \
4      +++++++++++++++B()._+-B()>B()._+-+++B()._+-------B(). \
5      _+---------B()._+-B()>+B()._+-B()>B()._+-B())
6  b.execute()
7  # prints "Hello World!"
```

# decorators

```
1  @decorator
2  def myfunc(x):
3      return x
4
5  myfunc = decorator(myfunc)
6
7  # decorator : f -> f
8
9  def decorator(f):
10     def wrap(x):
11         print "HELLO"
12         return f(x)
13     return wrap
```

# more decorators

```
1  @log_into("/tmp/file")
2  def myfunc(x):
3      return x * 2
4
5  myfunc = log_into("/tmp/file")(myfunc)
6
7  #loginto : log_into() return a decorator
8
9  def log_into(path):
10     def decorator(f):
11         def function(x):
12             print(path)
13             return f(x) * 2
14         return function
15     return decorator
```

# more decorators

```
1  class Decorator(object):
2      def __init__(self, args):
3          self.args = args
4
5      def __call__(self, f):
6          self.f = f
7          return self.func
8
9      def func(self, *args, **kargs):
10         print self.args
11         return self.f(*args, **kargs)
12
13 @Decorator("MSG")
14 def myf(x):
15     return x
16 print myf(42)
17 # "MSG"
18 # 42
```

# MORE decorators!

## Generating a decorator class using MetaClass

```
1  class DecInstr(type):
2      current_state = None
3
4      def __new__(cls, name, bases, attrs):
5          if '__next__' not in attrs:
6              f = lambda self, simple_next, state: simple_next
7              attrs['__next__'] = f
8          if '__do__' not in attrs:
9              attrs['__do__'] = lambda *args: None
10         attrs['__call__'] = cls.sub_call
11         return type.__new__(cls, name, bases, attrs)
```

# MORE META decorators!

```
1     def sub_call(self, next_func):
2         if not hasattr(next_func, "dad"):
3             type(self.__class__).current_state = state()
4         cstate = type(self.__class__).current_state
5         do = self.__do__
6         self.__do__  = lambda args: do(args, cstate)
7         ne = self.__next__
8         self.__next__ = lambda next_func: ne(next_func, cstate)
9         def n(*args):
10            res = self.__do__(args)
11            if res:
12                args = res
13            return self.__next__(next_func)(*args)
14        n.dad = self
15        type(self.__class__).current_state.flow.append(n)
16        return n
```

# decorython

```
1  __metaclass__ = DecInstr
2  class Loop(object):
3      def __init__(self, rep):
4          self.rep, self.maxrep = rep, rep
5
6      def __next__(self, simple_next, state):
7          self.rep -= 1
8          if self.rep == 0:
9              self.rep = self.maxrep
10             return simple_next
11         return state.flow[-1]
12
13 class DPrint(object):
14     def __init__(self, msg):
15         self.msg = msg
16
17     def __do__(self, args, state):
18         print self.msg
```

# decorython

```
1  def blink(state):
2      blink.x = not blink.x
3      return blink.x
4  blink.x = True
5
6  def onif(*args):
7      return (args[0] + "T",) +   args[1:]
8
9  @DPrint('POP')
10 @Dif(blink, onif)
11 @Loop(2)
12 @Duper()
13 def myfunc(string):
14     print string
15     return string
16
17 myfunc("re")          # POP \n POP \n RET
```

# super()

```
super(klass, instance).func(*args, *kwargs)
```

- `klass` represents the class from which we want to jump
- `instance` is the object on which is applied `func`.
- Syntactic sugar available to skip these parameters in a class declaration.

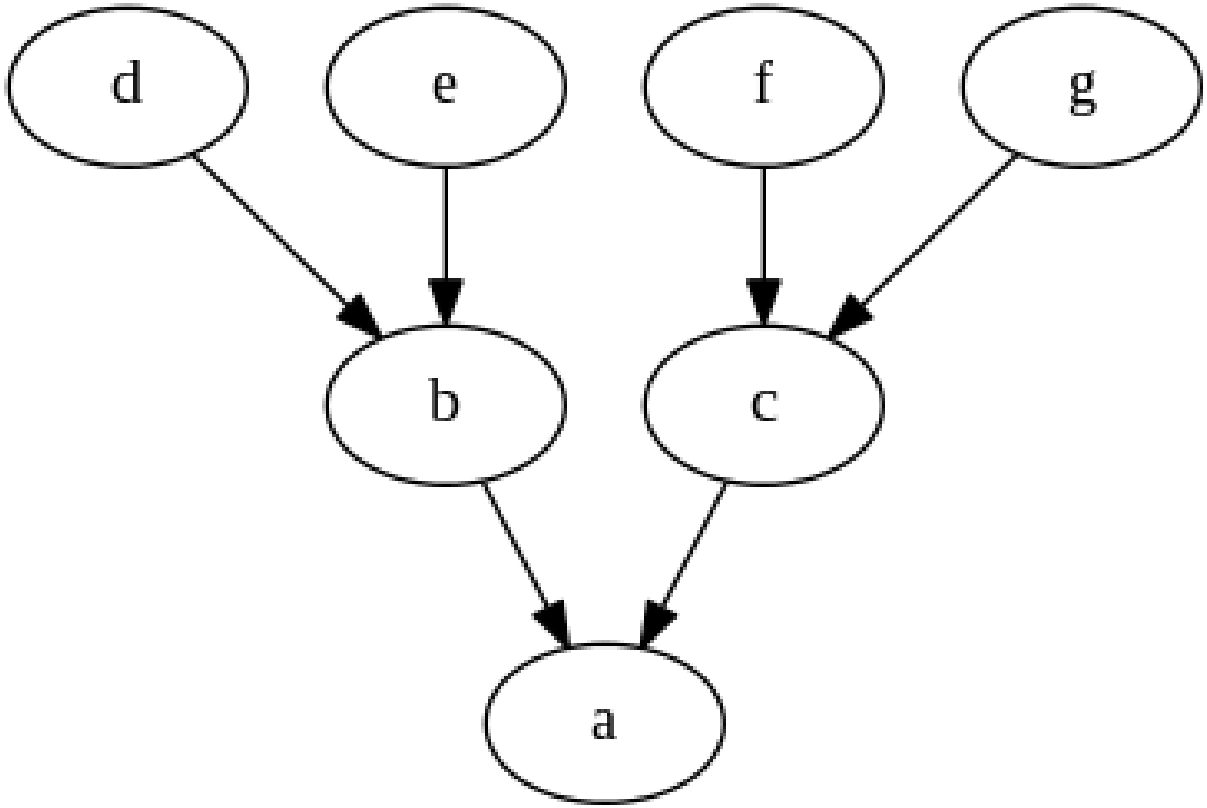
```
class B:
    def foo(self):
        print('Yay!')

class A(B):
    def foo(self):
        super().foo() # B.foo(self)
```

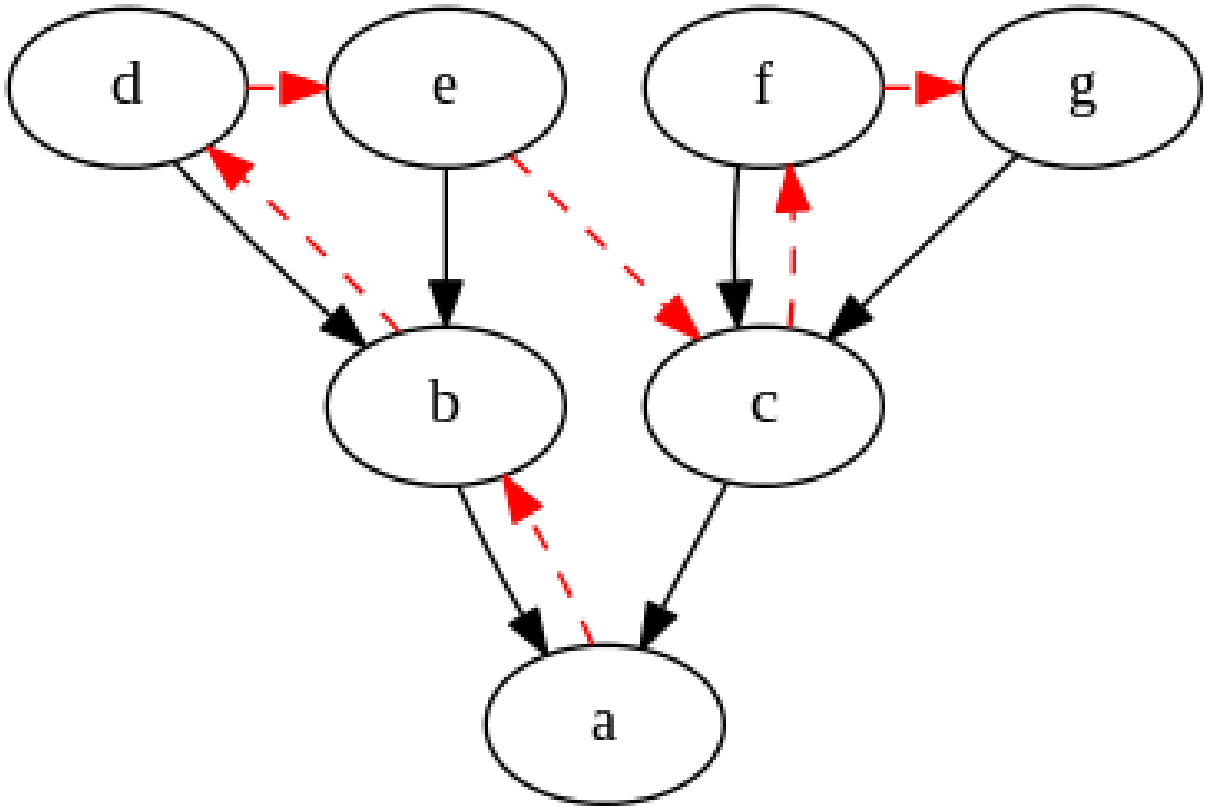
# Why use super()?

- Helps support cooperative multiple inheritance in a dynamic execution environment.
- It'll make sure that everything is visited in the right order (the same order as `getattr`)

# Simple inheritance



# Super on simple inheritance



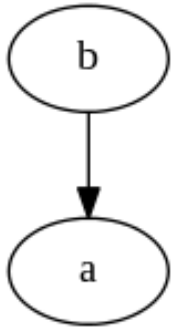
# MRO

```
1  class D: pass              # D   E
2  class E: pass              #  \ /
3  class B(D, E): pass        #   B
4
5  class F: pass              # F   G
6  class G: pass              #  \ /
7  class C(F, G): pass        #   C
8
9  class A(B, C): pass
10
11 # mro: (<class '__main__.A'>, <class '__main__.B'>,
12 #       <class '__main__.D'>, <class '__main__.E'>,
13 #       <class '__main__.C'>, <class '__main__.F'>,
14 #       <class '__main__.G'>, <class 'object'>)
15 print('mro:', A.__mro__)
```

# Common mistake with super()

- Sometimes you need to provide the parameters though, for example if you want a more complicated behavior, or are not in a class definition.

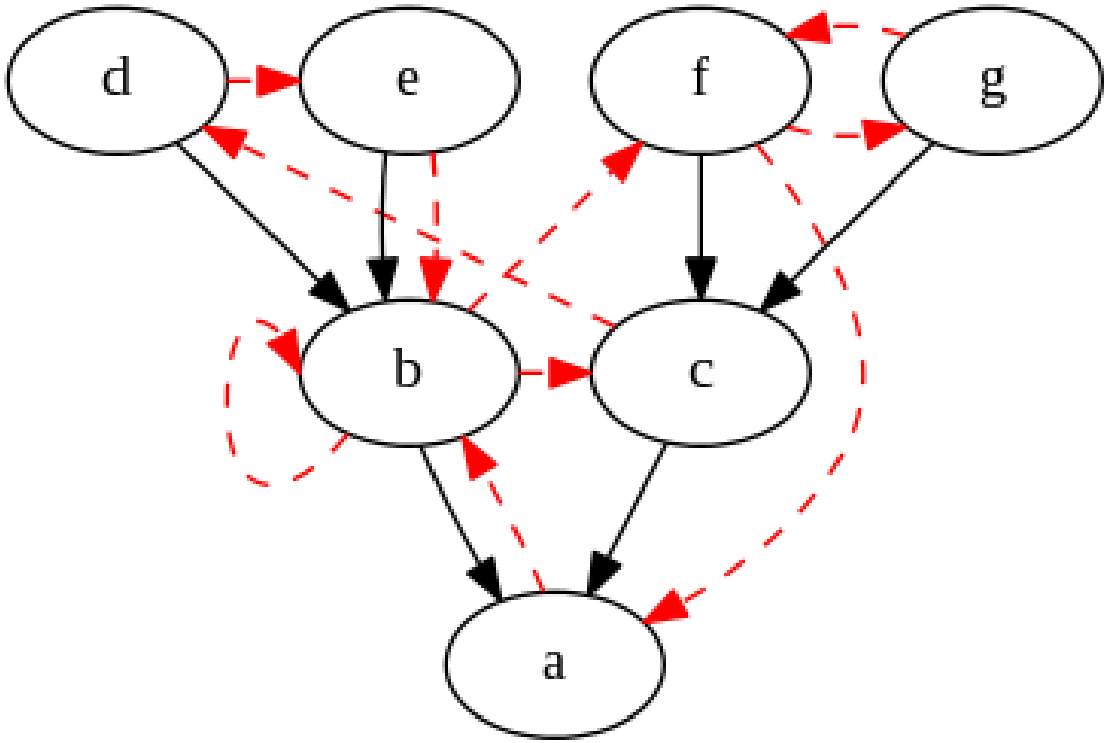
```
1  super(obj.__class__, obj).foo()
```



# What you can actually do

```
super(klass, instance).func(*args, *kwargs)
```

- If you control the `klass` argument, you can jump whereever you want in the inheritance tree.



# Some important info on inheritance

- You can have an instance of a class object in your inheritance tree only once.
  - Not a problem, since we can create a class and return it in a function.

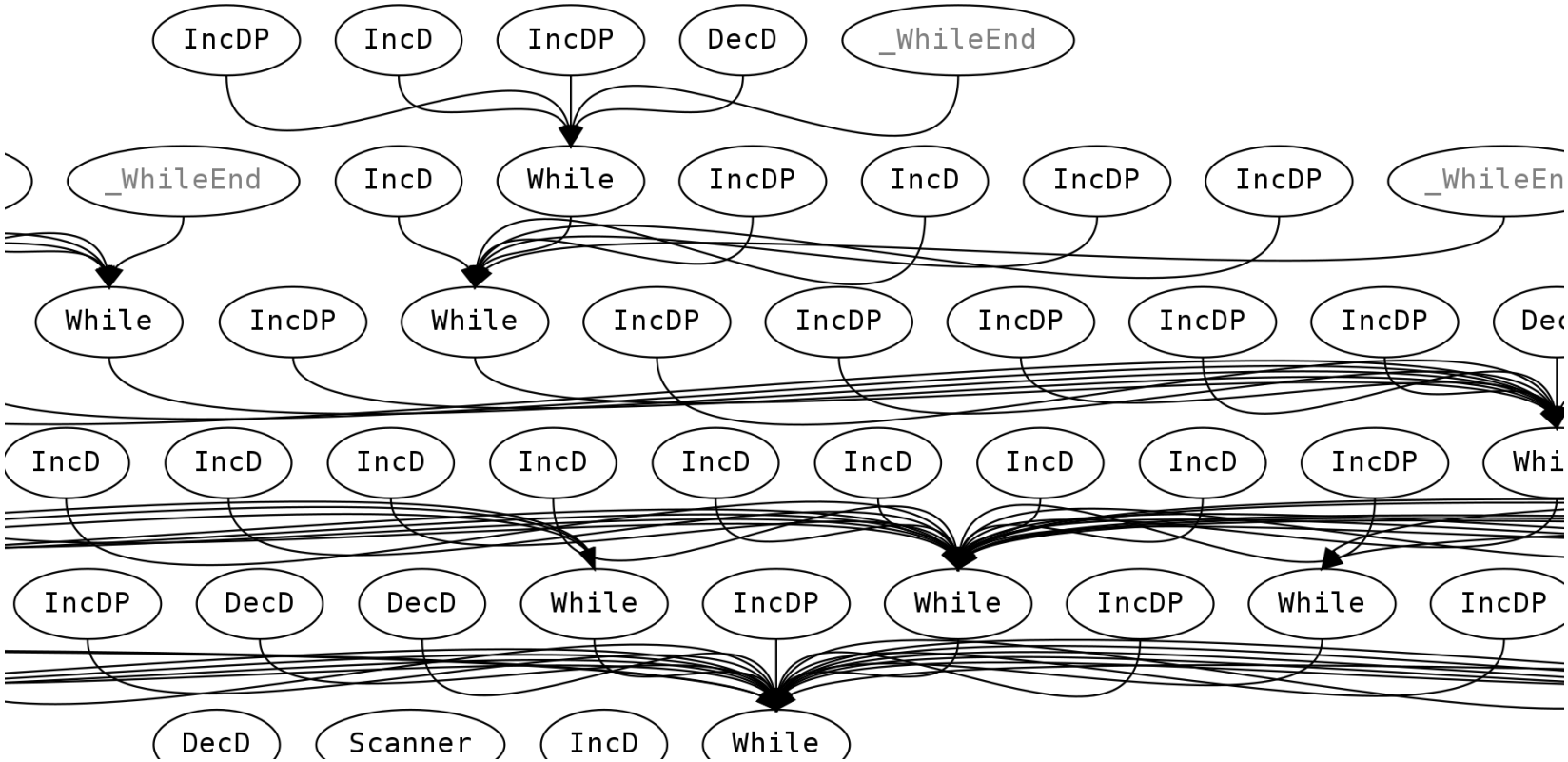
# Some important info on inheritance

- You can have an instance of a class object in your inheritance tree only once.
  - Not a problem, since we can create a class and return it in a function.
- Tail recursions are not optimized in python (choice of the BDFL) so you are limited to 1000 embedded calls. (Reached easily with loops)
  - Hardens the way to implement loops, but not that hard, and it's still possible to detect it while constructing the inheritance tree.

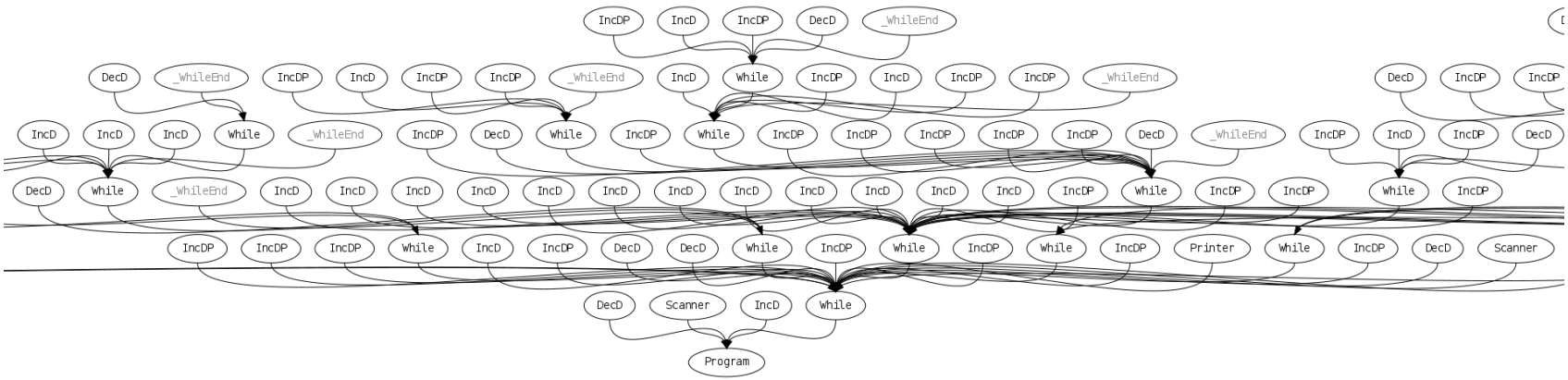
# VM implementation in the inheritance tree

- We will implement a VM with opcodes as classes and use the inheritance tree as a memory.
- All simple opcodes will just call the next instruction with super.
- Any opcode that wants to do complicated jumps can do it by calling `super()` themselves.
- Note: they can also add multiple classes in the inheritance tree, to use as "addresses".

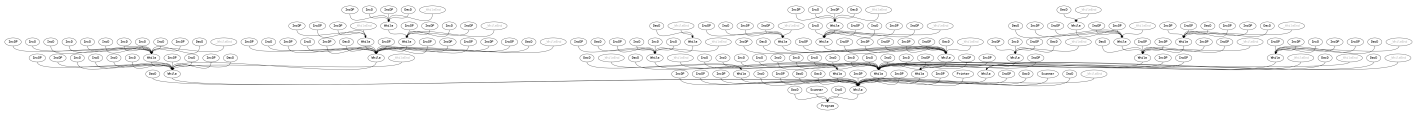
# Fuck me right!? [1/3]



# Fuck me right!? [2/3]



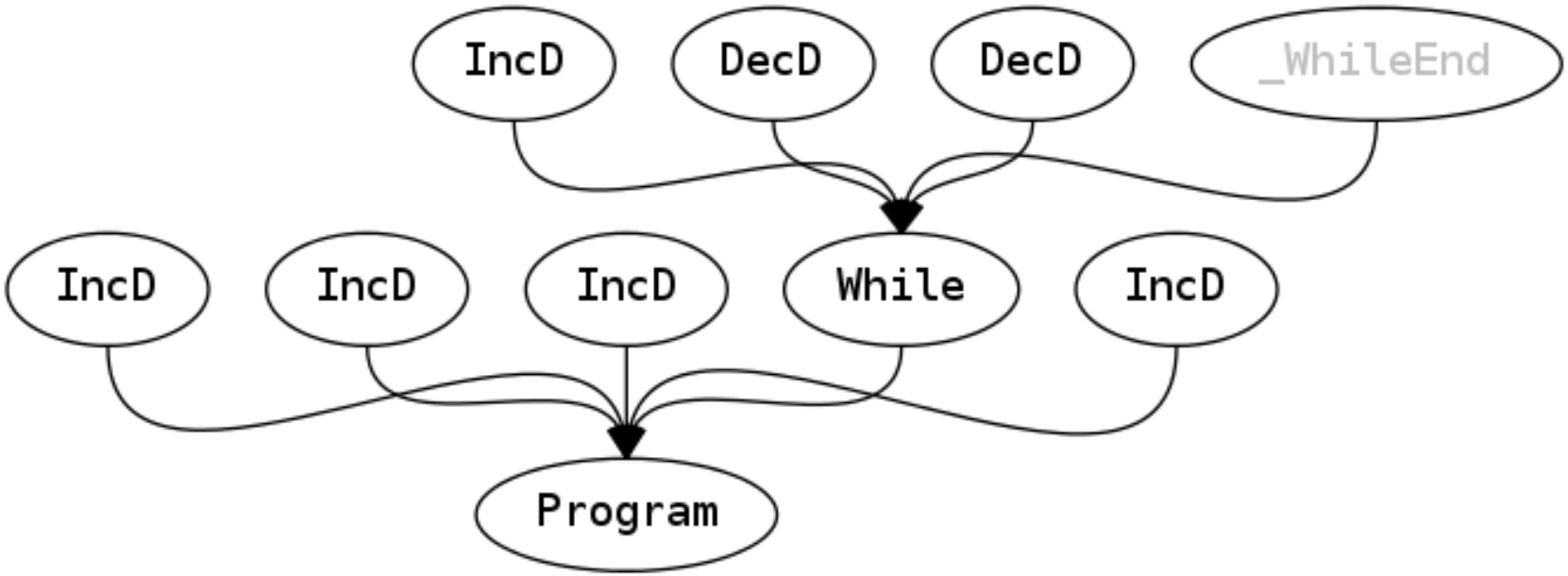
# Fuck me right!? [3/3]



# Program declaration

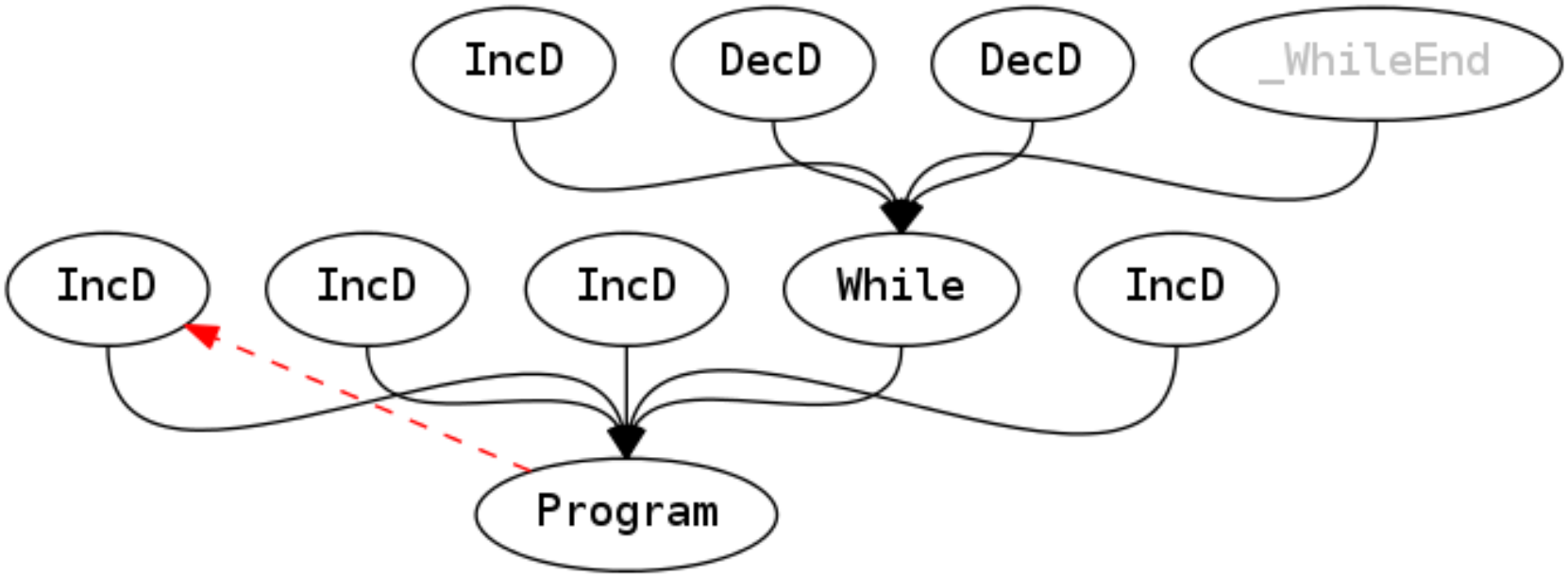
```
1  class Program(*str2inheritancetree(prog), metaclass=ProgramMeta):
2      def init_proc(self):
3          class Proc:
4              def __init__(self):
5                  self.dp, self.mem = 0, [0]
6
7              def __str__(self):
8                  msg = ['DP = {}'.format(self.dp),
9                         'MEM = {}'.format(self.mem)]
10                 return '; '.join(msg)
11         return Proc()
12
```

# +++[+--]+



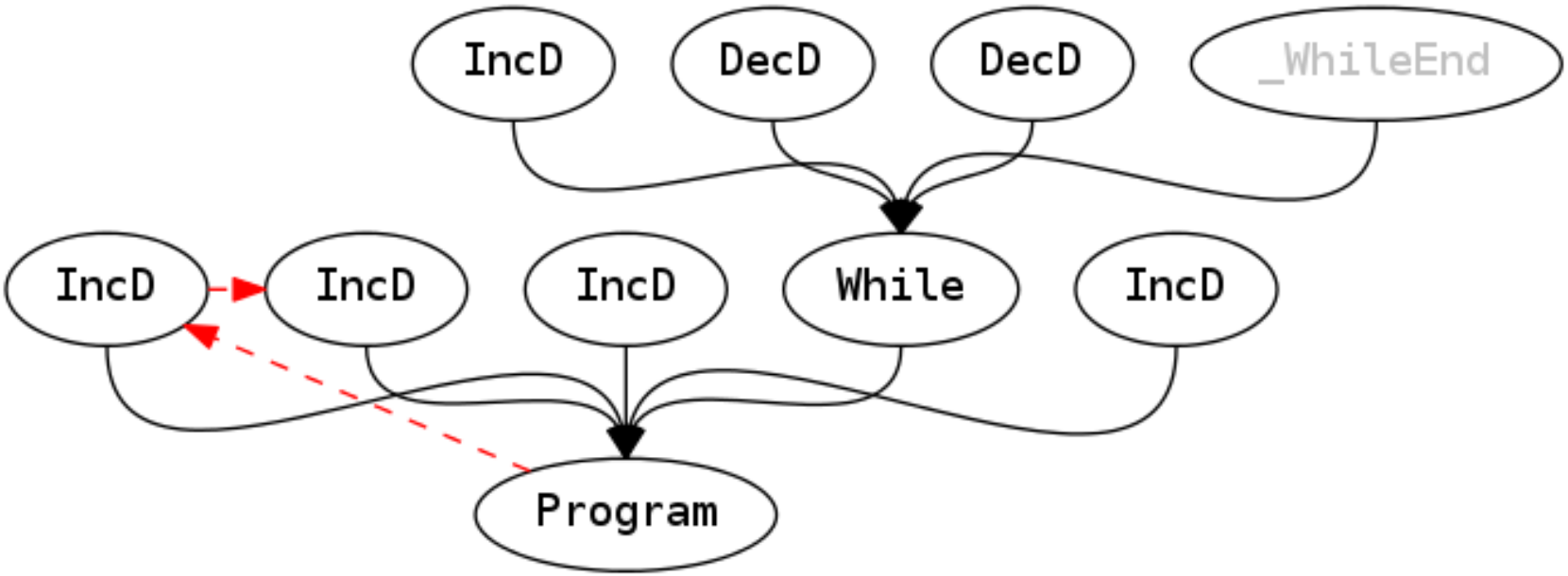

- `DP = 0; MEM = [0]`

# +++[+--]+



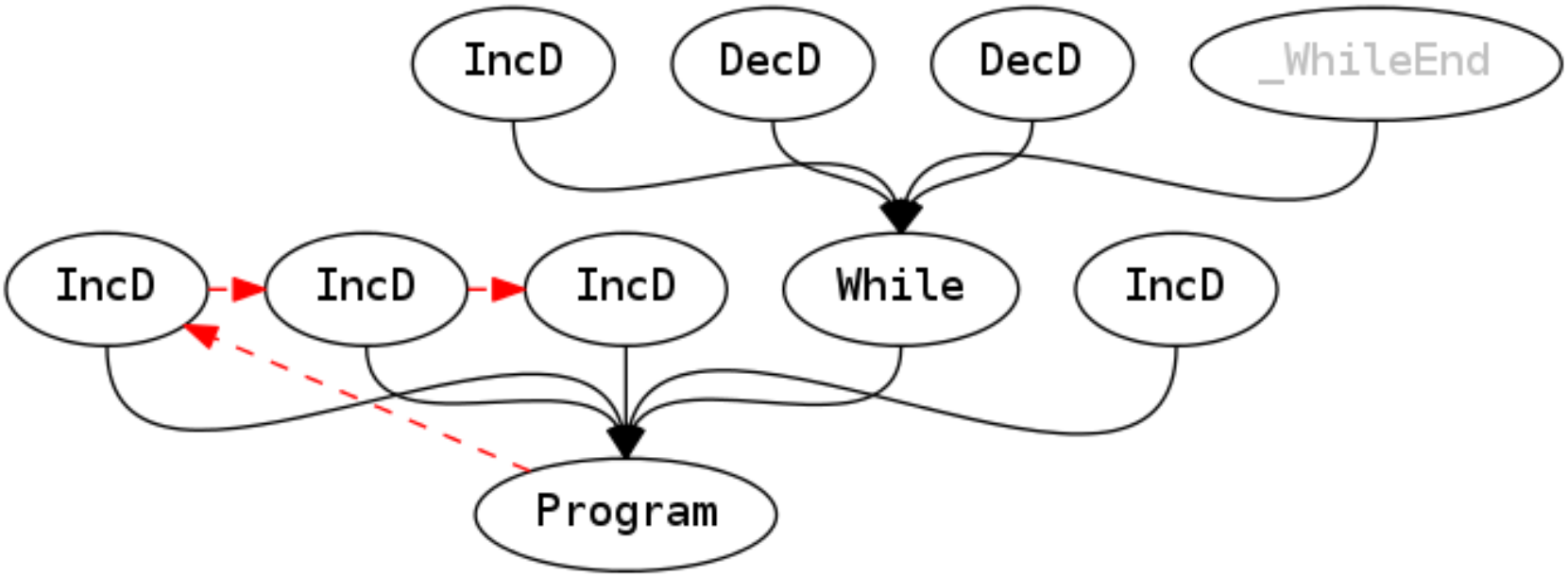

- `DP = 0; MEM = [1]`

# +++[+--]+



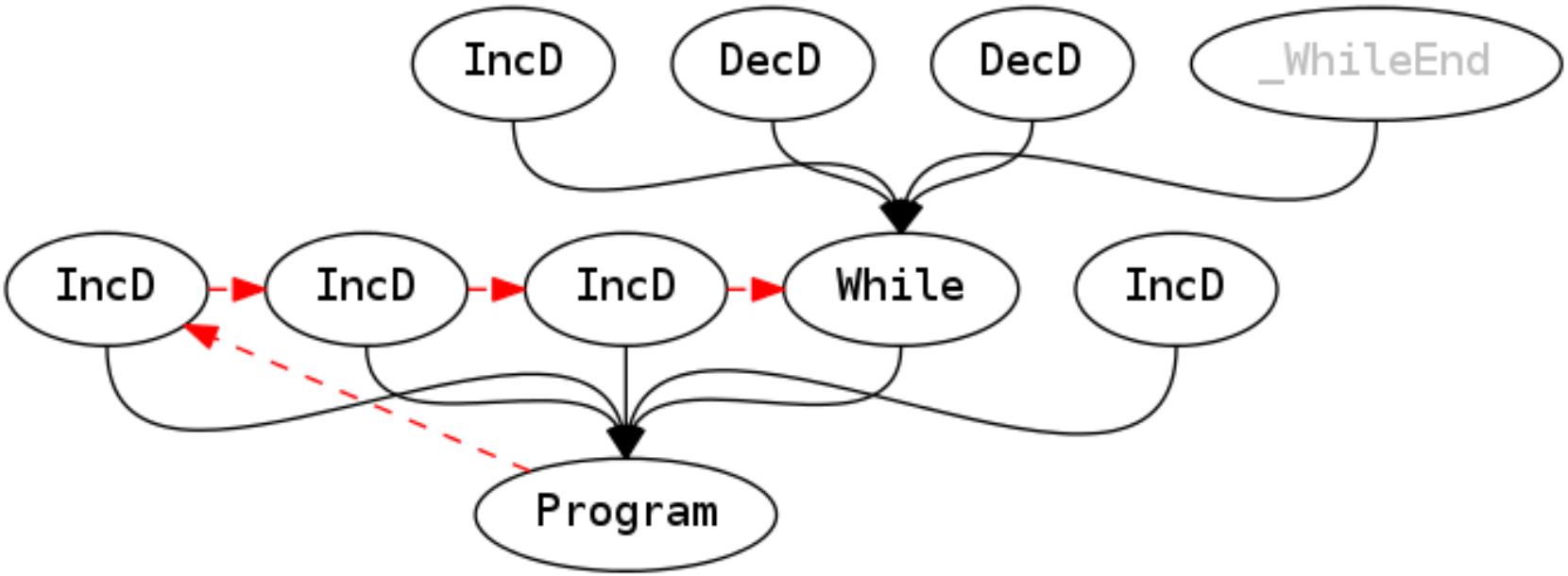

- `DP = 0; MEM = [2]`

# +++[+--]+



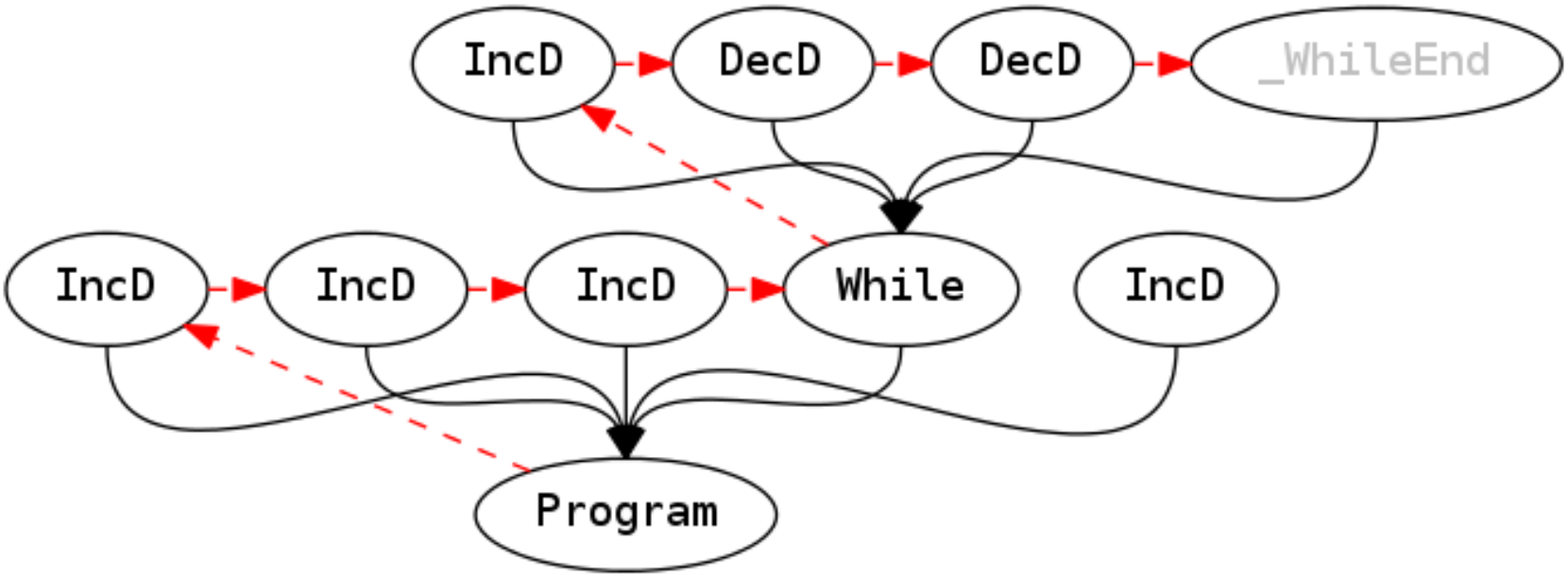

- `DP = 0; MEM = [3]`

# +++[+--]+



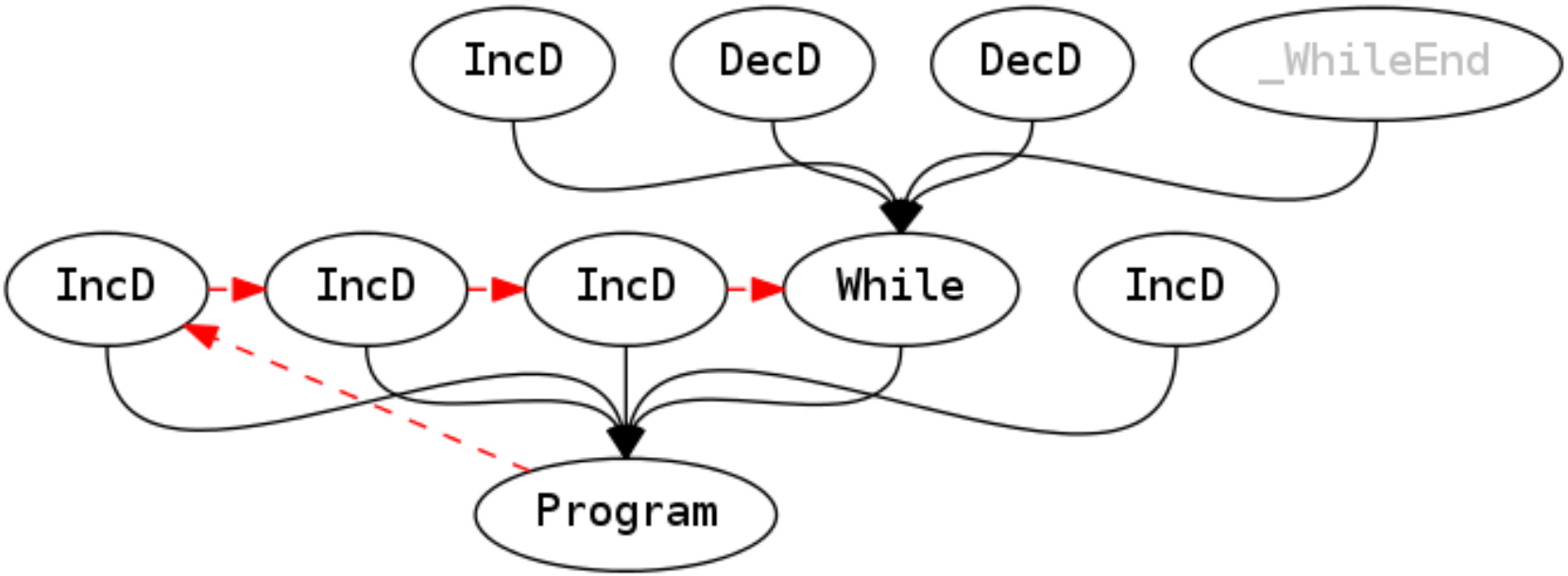

- `DP = 0; MEM = [3]`

# +++[+--]+



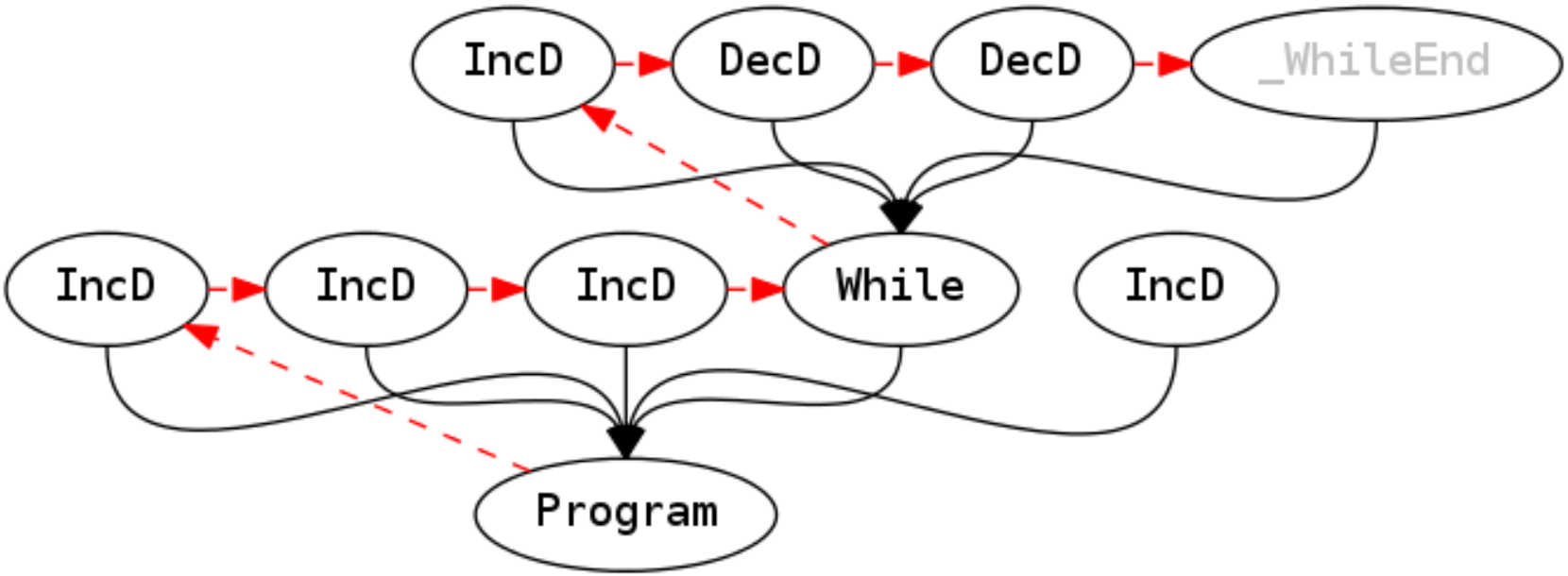

- `DP = 0; MEM = [2]`

# +++[+--]+



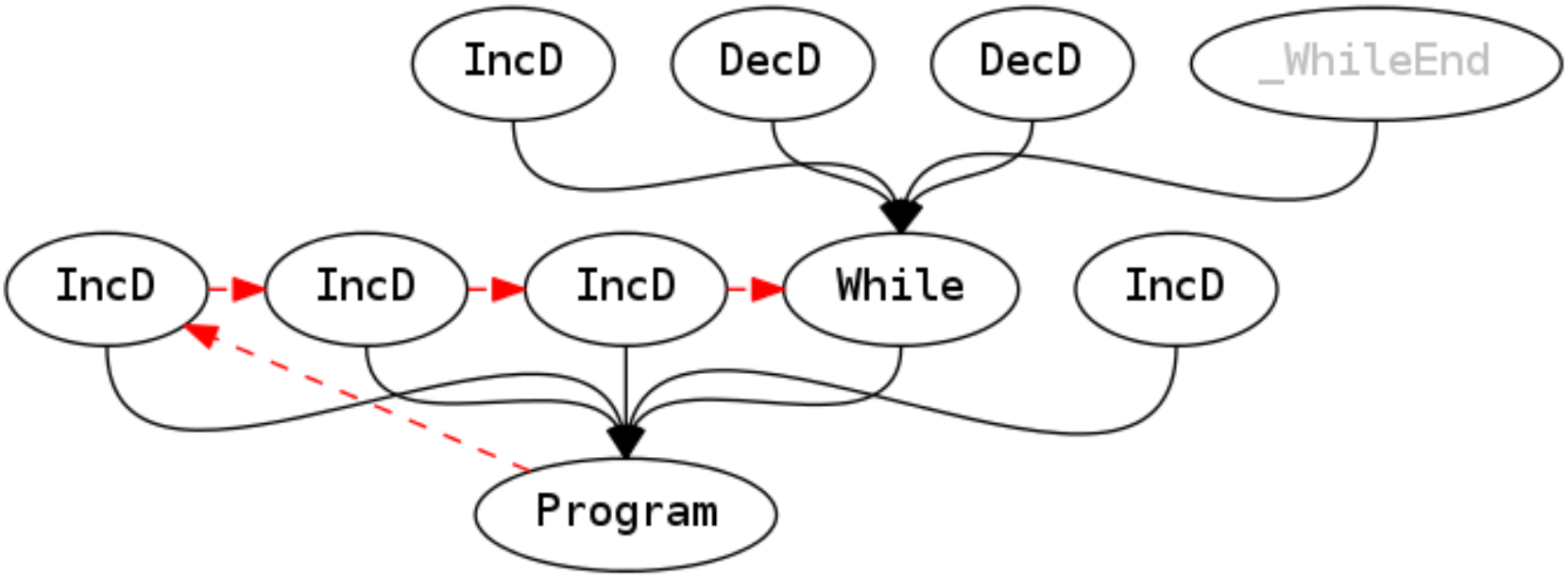

- `DP = 0; MEM = [2]`

# +++[+--]+



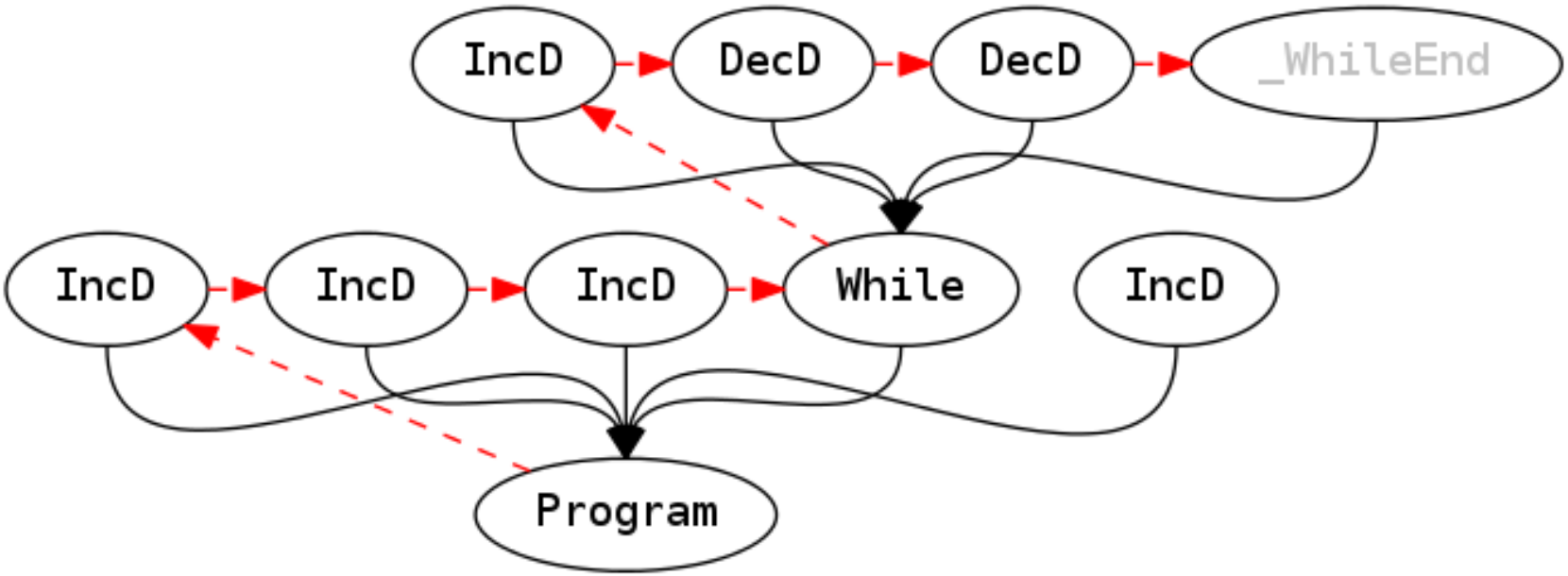

- `DP = 0; MEM = [1]`

# +++[+--]+



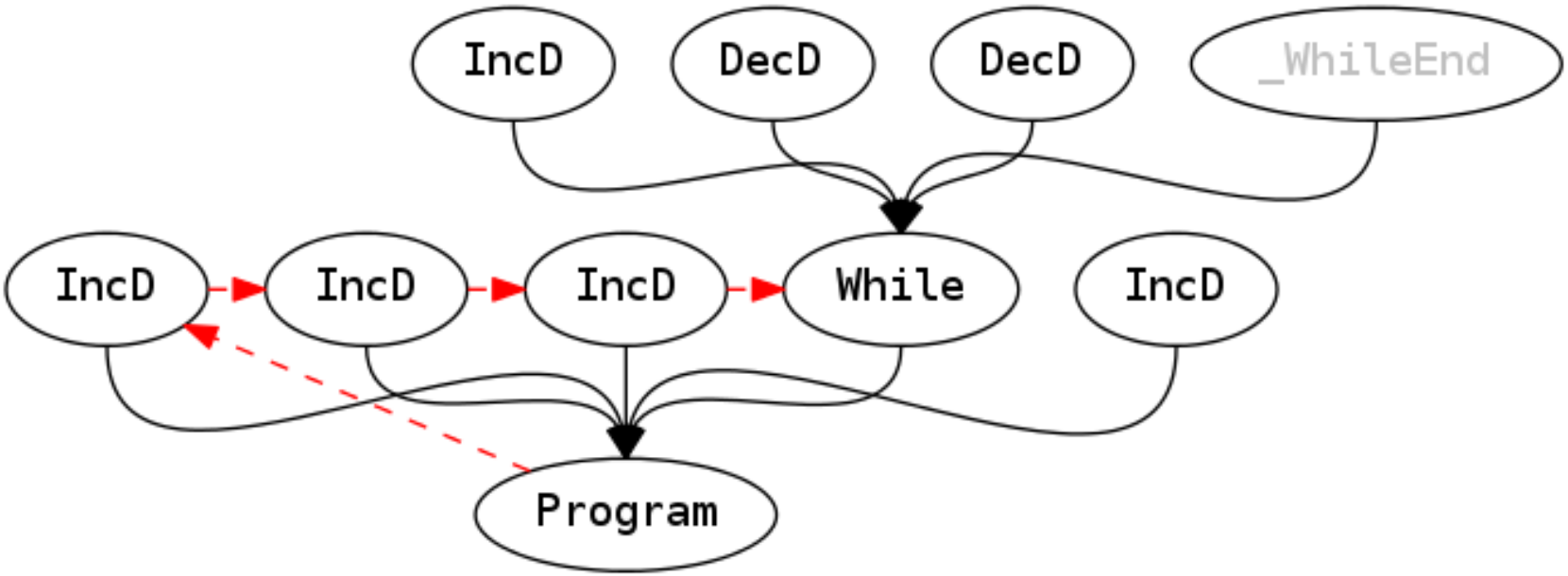

- `DP = 0; MEM = [1]`

# +++[+--]+



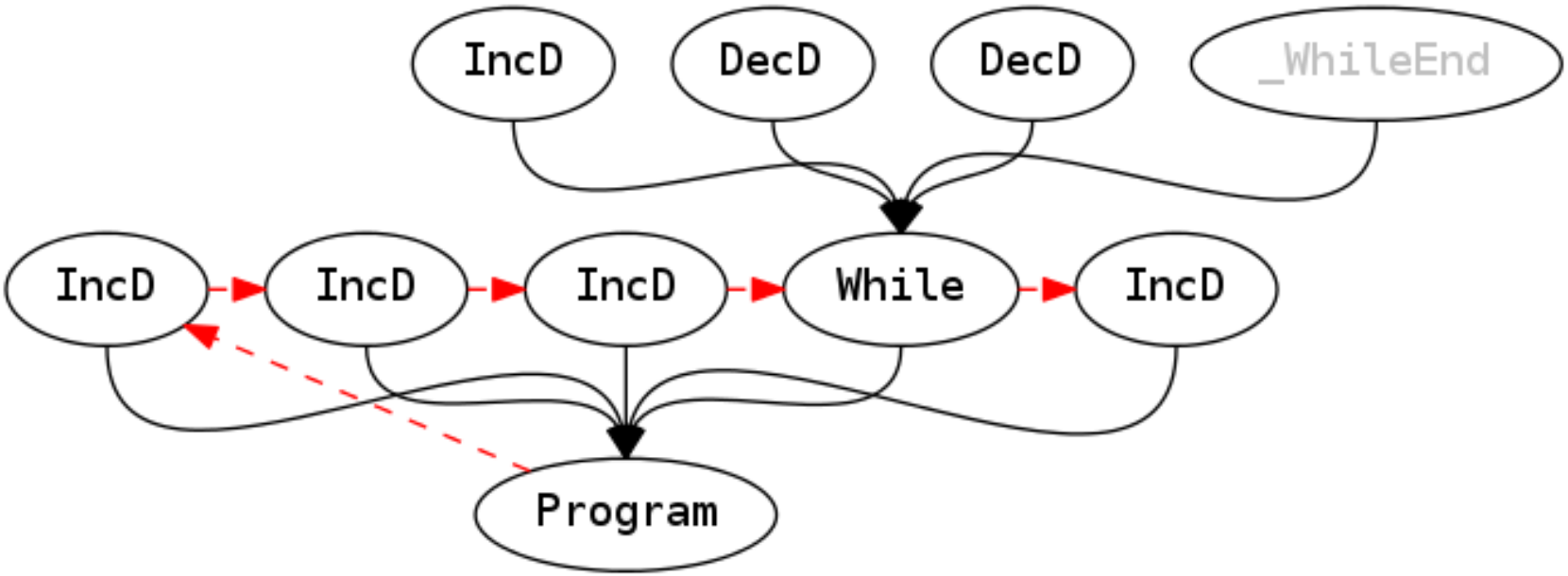

- `DP = 0; MEM = [0]`

# +++[+--]+




- `DP = 0; MEM = [0]`

# +++[+--]+




- `DP = 0; MEM = [1]`

# Conclusion

- Thank you for your attention!
- We have more ideas, and you might have some too, so come talk to us!
- Ask if you want a demo or PoCs :)

- Christian Raoul
- `hakril@lse.epita.fr`
- twitter: `@hakril`

- Quontin Choux
- `kushou@lse.epita.fr`
- twitter: `@kushou_`