

B

The NAND++ Programming Language

The NAND++ Programming language

Version: 0.2

The NAND++ programming language was designed to accompany the upcoming book ["Introduction to Theoretical Computer Science"](#). This is an appendix to this book, which is also available online as a Jupyter notebook in the [boazbk/nandnotebooks](#) on Github. You can also try the [live binder version](#).

The NAND++ programming language is defined in [Chapter 6: "Loops and Infinity"](#)

The NAND programming language we saw before corresponds to *non uniform, finite* computation.

NAND++ captures uniform computation and is equivalent to Turing Machines.

One way to think about NAND++ is

$$NAND++ = NAND + \text{loops} + \text{arrays}$$

Enhanced NAND++

We start by describing "enhanced NAND++ programs", and later we will describe "vanilla" or "standard" NAND++.

Enhanced NAND++ programs have the following form: every line is either of the form

```
foo = NAND(bar,blah)
```

or

```
i += foo
```

or

```
i -= foo
```

where `foo` is a variable identifier that is either a *scalar variable*, which is a sequence of letters, numbers and underscores, or an *array element*, which starts with a capital letter, and ends with `[i]`

We have a special variable `loop`. If `loop` is set to `1` at the end of the program then execution goes back to the beginning.

We have the special input and output arrays $X[.]$ and $Y[.]$ but because their length is not fixed in advance, we also have $X_{\text{valid}}[.]$ and $Y_{\text{valid}}[.]$ arrays. The input is $X[0], \dots, X[n-1]$ where n is the smallest integer such that $X_{\text{valid}}[n] = 0$. The output is $Y[0], \dots, Y[m-1]$ where m is the smallest integer such that $Y_{\text{valid}}[m] = 0$.

The default value of every variable in NAND++ is zero.

Ignore in first read: utility code

We use some utility code, which you can safely ignore in first read, to allow us to write NAND++ code in Python

```
In [1]: # utility code
%run "NAND programming language.ipynb"
from IPython.display import clear_output
clear_output()
```

```
In [2]: # Ignore this utility function in first and even second and third read
import inspect
import ast
import astor

def noop(f):
    return f;

def runwithstate(f):
    """Modify a function f to take and return an argument state and make a
    ll names relative to state."""
    tree = ast.parse(inspect.getsource(f))
    tmp = ast.parse("def _temp(state):\n    pass\n").body[0]
    args = tmp.args
    name = tmp.name
    tree.body[0].args = args
    tree.body[0].name = name
    tree.body[0].decorator_list = []

    class AddState(ast.NodeTransformer):
        def visit_Name(self, node: ast.Name):
            if node.id == "enandpp": return ast.Name(id="noop", ctx=Load
            ())
            new_node = ast.Attribute(ast.copy_location(ast.Name('state', a
            st.Load()), node), node.id,
                                     ast.copy_location(ast.Load(), node))
            return ast.copy_location(new_node, node)

    tree = AddState().visit(tree)
    tree.body[0].body = tree.body[0].body + [ast.parse("return state")]
    tree = ast.fix_missing_locations(tree)
    src = astor.to_source(tree)
    # print(src)
    exec(src, globals())
    _temp.original_func = f
    return _temp
```

```

def enandpp(f):
    g = runwithstate(f)
    def _temp1(X):
        nonlocal g
        return ENANDPPEVAL(g,X)
    _temp1.original_func = f
    _temp1.transformed_func = g
    return _temp1

```

In [3]: *# ignore utility class in first and even second or third read*

```

from collections import defaultdict
class NANDPPstate:
    """State of a NAND++ program."""

    def __init__(self):
        self.scalars = defaultdict(int)
        self.arrays = defaultdict(lambda: defaultdict(int))
        # eventually should make self.i non-negative integer type

    def __getattr__(self,var):
        g = globals()
        if var in g and callable(g[var]): return g[var]
        if var[0].isupper():
            return self.arrays[var]
        else:
            return self.scalars[var]

```

In [4]:

```

def ENANDPPEVAL(f,X):
    """Evaluate an enhanced NAND++ function on input X"""
    s = NANDPPstate()
    for i in range(len(X)):
        s.X[i] = X[i]
        s.Xvalid[i] = 1
    while True:
        s = f(s)
        if not s.loop: break
    res = []
    i = 0
    while s.Yvalid[i]:
        res += [s.Y[i]]
        i+= 1
    return res

```

In [5]:

```

def rreplace(s, old, new, occurrence=1): # from stackoverflow
    li = s.rsplit(old, occurrence)
    return new.join(li)

def ENANDPPcode(P):
    """Return ENAND++ code of given function"""

```

```

code = ...
counter = 0

class CodeENANDPPcounter:
    def __init__(self, name="i"):
        self.name = name

    def __iadd__(self, var):
        nonlocal code
        code += f'\ni += {var}'
        return self

    def __isub__(self, var):
        nonlocal code
        code += f'\ni -= {var}'
        return self

    def __str__(self): return self.name

class CodeNANDPPstate:

    def __getattr__(self, var):
        # print(f"getting {var}")
        if var=='i': return CodeENANDPPcounter()
        g = globals()
        if var in g and callable(g[var]): return g[var]
        if var[0].isupper():
            class Temp:
                def __getitem__(self, k): return f"{var}[{str(k)}]"
                def __setitem__(s, k, v): setattr(self, f"{var}[{str
(k)}}]", v)
            return Temp()
        return var

    def __init__(self):
        pass

    def __setattr__(self, var, val):
        nonlocal code
        if var=='i': return
        if code.find(val)==-1:
            code += f'\n{var} = {val}'
        else:
            code = rreplace(code, val, var)

s = CodeNANDPPstate()

def myNAND(a, b):
    nonlocal code, counter
    var = f'temp_{counter}'
    counter += 1
    code += f'\n{var} = NAND({a}, {b})'
    return var

```

```
s = runwith(lambda : P.transformed_func(s), "NAND", myNAND)

return code
```

Our first NAND++ program

Here is an enhanced NAND++ program to increment a number:

```
In [6]: @enandpp
def inc():
    carry = IF(started, carry, one(started))
    started = one(started)
    Y[i] = XOR(X[i], carry)
    carry = AND(X[i], carry)
    Yvalid[i] = one(started)
    loop = COPY(Xvalid[i])
    i += loop
```

```
In [7]: inc([1,1,0,0,1])
```

```
Out[7]: [0, 0, 1, 0, 1, 0]
```

```
In [8]: print(ENANDPPcode(inc))

temp_0 = NAND(started, started)
temp_1 = NAND(started, temp_0)
temp_2 = NAND(started, started)
temp_3 = NAND(temp_1, temp_2)
temp_4 = NAND(carry, started)
carry = NAND(temp_3, temp_4)
temp_6 = NAND(started, started)
started = NAND(started, temp_6)
temp_8 = NAND(X[i], carry)
temp_9 = NAND(X[i], temp_8)
temp_10 = NAND(carry, temp_8)
Y[i] = NAND(temp_9, temp_10)
temp_12 = NAND(X[i], carry)
carry = NAND(temp_12, temp_12)
temp_14 = NAND(started, started)
Yvalid[i] = NAND(started, temp_14)
temp_16 = NAND(Xvalid[i], Xvalid[i])
loop = NAND(temp_16, temp_16)
i += loop
```

And here is an enhanced NAND++ program to compute the XOR function on unbounded length inputs (it uses XOR on two variables as a subroutine):

```
In [9]: @enandpp
def UXOR():
    Yvalid[0] = one(X[0])
    Y[0] = XOR(X[0], Y[0])
```

```
loop = Xvalid[i]
i += Xvalid[i]
```

```
In [10]: UXOR([1,1,0,0,1,1])
```

```
Out[10]: [0]
```

```
In [11]: print(ENANDPPcode(UXOR))
```

```
temp_0 = NAND(X[0],X[0])
Yvalid[0] = NAND(X[0],temp_0)
temp_2 = NAND(X[i],Y[0])
temp_3 = NAND(X[i],temp_2)
temp_4 = NAND(Y[0],temp_2)
Y[0] = NAND(temp_3,temp_4)
loop = Xvalid[i]
i += Xvalid[i]
```

"Vanilla" NAND++

In "vanilla" NAND++ we do not have the commands `i += foo` and `i -= foo` but rather `i` travels obliviously according to the sequence `0, 1, 0, 1, 2, 1, 0, 1, 2, 3, 2, 1, 0, 1, 2, ...`

```
In [12]: def index():
         """Generator for the values of i in the NAND++ sequence"""
         i = 0
         last = 0
         direction = 1
         while True:
             yield i
             i += direction
             if i > last:
                 direction = -1
                 last = i
             if i==0: direction = +1

         a = index()
         [next(a) for i in range(20)]
```

```
Out[12]: [0, 1, 0, 1, 2, 1, 0, 1, 2, 3, 2, 1, 0, 1, 2, 3, 4, 3, 2, 1]
```

```
In [13]: def NANDPPEVAL(f,X):
         """Evaluate a NAND++ function on input X"""
         s = NANDPPstate() # intialize state

         # copy input:
         for i in range(len(X)):
             s.X[i] = X[i]
             s.Xvalid[i] = 1

         # main loop:
         for i in index():
             s.i = i
```

```

        s = t(s)
        if not s.loop: break

    # copy output:
    res = []
    i = 0
    while s.Yvalid[i]:
        res += [s.Y[i]]
        i+= 1
    return res

def nandpp(f):
    """Modify python code to obtain NAND++ program"""
    g = runwithstate(f)
    def _temp1(X):
        return NANDPPEVAL(g,X)
    _temp1.original_func = f
    _temp1.transformed_func = g
    return _temp1

```

Here is the increment function in vanilla NAND++. Note that we need to keep track of an Array Visited to make sure we only add the carry once per location.

```

In [14]: @nandpp
def inc():
    carry = IF(started,carry,one(started))
    started = one(started)
    Y[i] = IF(Visited[i],Y[i],XOR(X[i],carry))
    Visited[i] = one(started)
    carry = AND(X[i],carry)
    Yvalid[i] = one(started)
    loop = Xvalid[i]

```

```

In [15]: inc([1,1,0,1,1])

```

```

Out[15]: [0, 0, 1, 1, 1, 0]

```

And here is the "vanilla NAND++" version of XOR:

```

In [16]: @nandpp
def vuXOR():
    Yvalid[0] = one(X[0])
    Y[0] = IF(Visited[i],Y[0],XOR(X[i],Y[0]))
    Visited[i] = one(X[0])
    loop = Xvalid[i]

```

```

In [17]: vuXOR([1,0,0,1,0,1,1])

```

```

Out[17]: [0]

```

```

In [18]: def NANDPPcode(P):

```

```

    """NAND++ code for P"""

```



```

"""Return NAND++ code of given function"""

code = ''
counter = 0

class CodeNANDPPstate:

    def __getattr__(self,var):
        # print(f"getting {var}")
        g = globals()
        if var in g and callable(g[var]): return g[var]
        if var[0].isupper():
            class Temp:
                def __getitem__(self,k): return var+"[i]"
                def __setitem__(s,k,v):
                    setattr(self,var+"[i]",v)
            return Temp()
        return var

    def __init__(self):
        pass

    def __setattr__(self,var,val):
        nonlocal code
        # print(f"setting {var} to {val}")
        if code.find(val)==-1:
            code += f'\n{var} = {val}'
        else:
            code = rreplace(code,val,var)

s = CodeNANDPPstate()

def myNAND(a,b):
    nonlocal code, counter
    var = f'temp_{counter}'
    counter += 1
    code += f'\n{var} = NAND({a},{b})'
    return var

s = runwith(lambda : P.transformed_func(s),"NAND",myNAND)

return code

# utility code - replace string from right, taken from stackoverflow
def rreplace(s, old, new, occurrence=1):
    li = s.rsplit(old, occurrence)
    return new.join(li)

```

In [19]: `print(NANDPPcode(inc))`

```

temp_0 = NAND(started,started)
temp_1 = NAND(started,temp_0)

```

```

temp_1 = NAND(started, temp_0)
temp_2 = NAND(started, started)
temp_3 = NAND(temp_1, temp_2)
temp_4 = NAND(carry, started)
carry = NAND(temp_3, temp_4)
temp_6 = NAND(started, started)
started = NAND(started, temp_6)
temp_8 = NAND(X[i], carry)
temp_9 = NAND(X[i], temp_8)
temp_10 = NAND(carry, temp_8)
temp_11 = NAND(temp_9, temp_10)
temp_12 = NAND(Visited[i], Visited[i])
temp_13 = NAND(temp_11, temp_12)
temp_14 = NAND(Y[i], Visited[i])
Y[i] = NAND(temp_13, temp_14)
temp_16 = NAND(started, started)
Visited[i] = NAND(started, temp_16)
temp_18 = NAND(X[i], carry)
carry = NAND(temp_18, temp_18)
temp_20 = NAND(started, started)
Yvalid[i] = NAND(started, temp_20)
loop = Xvalid[i]

```

Transforming Enhanced NAND++ to NAND++

Eventually we will have here code to automatically transform an enhanced NAND++ program into a NAND++ program. At the moment, let us just give the high level ideas. See [Chapter 6 in the book](#) for more details.

To transform an enhanced NAND++ program to a standard NAND++ program we do the following:

1. We make all our operations "guarded" in the sense that there is a special variable `noop` such that if `noop` equals 1 then we do not make any writes.
2. We use a `Visited` array to keep track of all locations we visited, and use that to keep track of an decreasing variable that is equal to 1 if and only the value of `i` in the next step will be one smaller.
3. If we have an operation of the form `i += foo` or `i -= foo` at line ℓ then we replace it with lines of code that do the following:
 - a. (Guarded) set `temp ℓ = foo`
 - b. (Unguarded) If `Waitingline ℓ` and `Restart[i]` : set `noop=0` if increasing is equal to `wait_increasing`. (Otherwise `noop` stays the same.)
 - c. (Guarded) set `Restart[i]` to 1.
 - d. (Guarded) set `Waitingline ℓ` to 1.
 - e. (Guarded) set `wait_increasing` to 1 if the operation is `i += foo` and to 0 if it's `i -= foo`

- f. (Guarded) set `noop = temp_ℓ`
- g. (Unguarded) set `temp_ℓ = 0`
- h. (Guarded) set `Restart[i]` to 0.
- i. (Guarded) set `Waitingline_ℓ` to 0.

