Learning Objectives:

- Learn the model of NAND++ program that involve loops.
- See some basic syntactic sugar for NAND++
- Get comfort with switching between representation of NAND++ programs as code and as tuples.
- Learn the notion of *configurations* for NAND++ programs.
- Understand the relation between NAND++ and NAND programs.

7

Loops and infinity

> *"We thus see that when $n = 1$, nine operation-cards are used; that when $n = 2$, fourteen Operation-cards are used; and that when $n > 2$, twenty-five operation-cards are used; but that no more are needed, however great n may be; and not only this, but that these same twenty-five cards suffice for the successive computation of all the numbers"*, Ada Augusta, countess of Lovelace, 1843 [1]

> *"It is found in practice that (Turing machines) can do anything that could be described as 'rule of thumb' or 'purely mechanical'... (Indeed,) it is now agreed amongst logicians that 'calculable by means of (a Turing Machine)' is the correct accurate rendering of such phrases."*, Alan Turing, 1948

[1] Translation of "Sketch of the Analytical Engine" by L. F. Menabrea, Note G.

The NAND programming language has one very significant drawback: a finite NAND program P can only compute a finite function F, and in particular the number of inputs of F is always smaller than the number of lines of P. This does not capture our intuitive notion of an algorithm as a *single recipe* to compute a potentially infinite function. For example, the standard elementary school multiplication algorithm is a *single* algorithm that multiplies numbers of all lengths, but yet we cannot express this algorithm as a single NAND program, but rather need a different NAND program for every input length.

Let us consider the case of the simple *parity* or *XOR* function $XOR : \{0,1\}^* \rightarrow \{0,1\}$, where $XOR(x)$ equals 1 iff the number of 1's in x is odd. As simple as it is, the *XOR* function cannot be computed by a NAND program. Rather, for every n, we can compute XOR_n (the restriction of *XOR* to $\{0,1\}^n$) using a different NAND program.

For example, here is the NAND program to compute XOR_5:

```
u := x_0 NAND x_1
v := x_0 NAND u
w := x_1 NAND u
s := v NAND w
u := s NAND x_2
v := s NAND u
w := x_2 NAND u
s := v NAND w
u := s NAND x_3
v := s NAND u
w := x_3 NAND u
s := v NAND w
u := s NAND x_4
v := s NAND u
w := x_4 NAND u
y_0 := v NAND w
```

This is rather repetitive, and more importantly, does not capture the fact that there is a *single* algorithm to compute the parity on all inputs. Typical programming language use the notion of *loops* to express such an algorithm, and so we might have wanted to use code such as:

```
# s is the "running parity", initalized to 0
while i < length(x):
   u := x_i NAND s
   v := s NAND u
   w := x_i NAND u
   s := v NAND w
   i++
ns := s NAND s
y_0 := ns NAND ns
```

We will now discuss how we can extend the NAND programming language so that it can capture this kind of a construct.

7.1 The NAND++ Programming language

Keeping to our minimalist form, we will not add a `while` keyword to the NAND programming language. But we will extend this language in a way that allows for executing loops and accessing arrays of arbitrary length.

The main new ingredients are the following:

- We add a special variable `loop` with the following semantics: after executing the last line of the program, if `loop` is equal to one, then instead of halting, the program goes back to the first line. If `loop` is equal to zero after executing the last line then the program halts as is usual with NAND.[2]

- We add a special *integer valued* variable `i`, and allow expressions of the form `foo_i` (for every variable identifier `foo`) which are evaluated to equal `foo_`$\langle i \rangle$ (where $\langle i \rangle$ denotes the current value of the variable `i`). For example, if the current value of `i` is equal to 15, then `foo_i` corresponds to `foo_15`.[3] In the first loop of the program, `i` is assigned the value 0, but each time the program loops back to the first line, the value of `i` is updated in the following manner: in the k-th iteration the value of `i` equals $I(k)$ where $I = (I(0), I(1), I(2), \ldots)$ is the following sequence (see Fig. 7.1):[4]

$$0, 1, 0, 1, 2, 1, 0, 1, 2, 3, 2, 1, 0, \ldots \tag{7.1}$$

- Because the input to NAND++ programs can have variable length, we also add a special read-only array `validx` such that `validx_`$\langle n \rangle$ is equal to 1 if and only if the n is smaller than the length of the input. In particular, `validx_i` will equal to 1 if and only if the value of `i` is smaller than the length of the input.

- Like NAND programs, the output of a NAND++ program is the string `y_0`, ..., `y_`$\langle k \rangle$ where k is the largest integer such that `y_`$\langle k \rangle$ was assigned a value.[5]

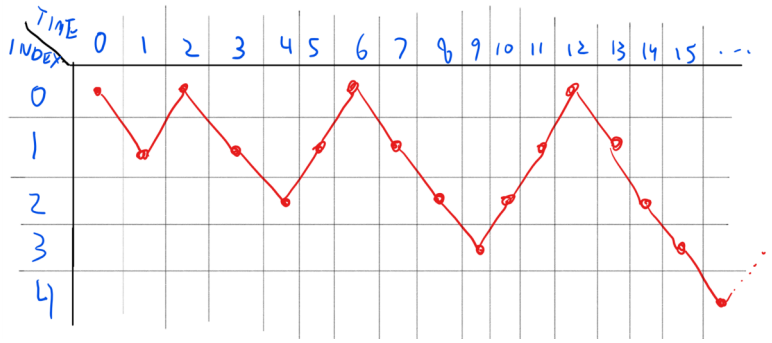

Figure 7.1: The value of `i` as a function of the current iteration. The variable `i` progresses according to the sequence $0, 1, 0, 1, 2, 1, 0, 1, 2, 3, 2, 1, 0, \ldots$. At the k-th iteration the value of `i` equals $k - r(r+1)$ if $k \leq (r+1)^2$ and $(r+1)(r+2) - k$ if $k < (r+1)^2$ where $r = \lfloor \sqrt{k + 1/4} - 1/2 \rfloor$.

See the appendix for a more formal specification of the NAND++ programming language, and the website http://nandpl.org for an

[2] This corresponds to wrapping the entire program in one big loop that is executed at least once and continues as long as `loop` is equal to 1. For example, in the C programming language this would correspond with wrapping the entire program with the construct `do ... while (loop);`.

[3] Note that the variable `i`, like all variables in NAND, is a *global* variable, and hence all expressions of the form `foo_i`, `bar_i` etc. refer to the same value of `i`.

[4] TODO: Potentially change in the future to Salil's sequence $INDEX(\ell) = min\ell - floor(sqrt(\ell))^2, ceiling(sqrt(\ell))^2 - \ell$ which has the form $0, 0, 1, 1, 0, 1, 2, 2, 1, 0, 1, 2, 3, 3, 2, 1, 0, 1, 2, 3, 4, 4, 3, 2, 1, 0, \ldots$.

[5] To allow control of the output length, we also add a write-only array `invalidy`. If there exist $j < k$ such that `invalidy_`$\langle j \rangle$=1 then we reduce the output length to $j - 1$. However, we will hardly use this array in this course, since we will almost always be interested in programs with a fixed output length (and in fact most often in programs with one bit of output).

implementation. Here is the NAND++ program to compute parity of arbitrary length: (It is a good idea for you to see why this program does indeed compute the parity)

```
# compute sum x_i (mod 2)
# s = running parity
# seen_i = 1 if this index has been seen before

# Do val := (NOT seen_i) AND x_i
tmp_1 := seen_i NAND seen_i
tmp_2 := x_i NAND tmp_1
val := tmp_2 NAND tmp_2

# Do s := s XOR val
ns := s NAND s
y_0 := ns NAND ns
u := val NAND s
v := s NAND u
w := val NAND u
s := v NAND w

seen_i := zero NAND zero
stop := validx_i NAND validx_i
loop := stop NAND stop
```

When we invoke this program on the input 010, we get the following execution trace:

```
... (complete this here)
End of iteration 0, loop = 1, continuing to iteration 1
...
End of iteration 2, loop = 0, halting program
```

7.1.1 *Computing the index location*

We say that a NAND program completed its *r-th round* when the index variable `i` completed the sequence:

$$0,1,0,1,2,1,0,1,2,3,2,1,0,\ldots,0,1,\ldots,r,r-1,\ldots,0 \tag{7.2}$$

This happens when the program completed

$$1+2+4+6+\cdots+2r = r^2+r+1 \tag{7.3}$$

iterations of its main loop. (The last equality is obtained by applying the formula for the sum of an arithmetic progression.) This means that if we keep a "loop counter" k that is initially set to 0 and increases by one at the end of any iteration, then the "round" r is the largest integer such that $r(r+1) \leq k$, which (as you can verify) equals $\lfloor \sqrt{k+1/4} - 1/2 \rfloor$.

Thus the value of `i` in the k-th loop equals:

$$index(k) = \begin{cases} k - r(r+1) & k \leq (r+1)^2 \\ (r+1)(r+2) - k & \text{otherwise} \end{cases} \tag{7.4}$$

where $r = \lfloor \sqrt{k+1/4} - 1/2 \rfloor$. (We ask you to prove this in Exercise 7.1.)

R **Variables as arrays** In NAND we allowed variables to have names such as `foo_17` but the numerical part of the identifier played essentially the same role as alphabetical part. In particular, NAND would be just as powerful if we didn't allow any numbers in the variable identifiers. With the introduction of the special index variable `i`, in NAND++ things are different. It is best to think of each NAND++ variable `foo` as an *array*, with its j-th position corresponding to `foo_`$\langle j \rangle$ (which in other programming languages would often be written as `foo[`$\langle j \rangle$`]`). Recall also our convention that a variable without an index such as `bar` is equivalent to `bar_0`, or the first position of the corresponding array. Of course we can think of variables as arrays in NAND as well, but since in NAND all indices are absolute numerical constants, this viewpoint does not make much of a difference as it does in NAND++.

7.1.2 *Infinite loops and computing a function*

One crucial difference between NAND and NAND++ programs is the following. Looking at a NAND program P, we can always tell how many inputs and how many outputs it has (by simply counting the number of `x_` and `y_` variables). Furthermore, we are guaranteed that if we invoke P on any input then *some* output will be produced.

In contrast, given any particular NAND++ program P', we cannot determine a priori the length of the output. In fact, we don't even know if an output would be produced at all! For example, the follow-

ing NAND++ program would go into an infinite loop if the first bit of the input is zero:

```
loop := x_0 NAND x_0
```

For a NAND++ program P and string $x \in \{0,1\}^*$, if P produces an output when executed with input x then we denote this output by $P(x)$. If P does not produce an output on x then we say that $P(x)$ is *undefined* and denote this as $P(x) = \perp$.

Definition 7.1 — Computing a function. We say that a NAND++ program P *computes* a function $F : \{0,1\}^* :\to \{0,1\}^*$ if $P(x) = F(x)$ for every $x \in \{0,1\}^*$.

If F is a partial function then we say that P *computes* F if $P(x) = F(x)$ for every x on which F is defined.

We say that a function F is *NAND++ computable* if there is a NAND++ program that computes it.

We will often drop the "NAND++" qualifier and simply call a function *computable* if it is NAND++ computable. This may seem "reckless" but, as we'll see in future lectures, it turns out that being NAND++-computable is equivalent to being computable in essentially any reasonable model of computation.

R **Notation** If $F : \{0,1\}^* \to \{0,1\}$ is a Boolean function, then computing F is equivalent to deciding membership in the set $L = \{x \in \{0,1\}^* \mid F(x) = 1\}$. Subsets of $\{0,1\}^*$ are known as *languages* in the literature. Such a language $L \subseteq \{0,1\}^*$ is known as *decidable* or *recursive* if the corresponding function F is computable.

7.2 *A spoonful of sugar*

Just like NAND, we can add a bit of "syntactic sugar" to NAND++ as well. These are constructs that can help us in expressing programs, though ultimately do not change the computational power of the model, since any program using these constructs can be "unsweetened" to obtain a program without them.

7.2.1 *Inner loops via syntactic sugar*

While NAND+ only has a single "outer loop", we can use conditionals to implement inner loops as well. That is, we can replace code such as

```
PRELOOP_CODE
while (cond) {
 LOOP_CODE
}
POSTLOOP_CODE
```

by

```
// startedloop is initialized to 0
// finishedloop is initalized to 0
if NOT(startedloop) {
   PRELOOP_CODE
   startedloop := 1
   temploop := loop
}
if NOT(finishedloop) {
   if (cond) {
      LOOP_CODE
      loop :=1
   }
   if NOT(cond) {
      finishedloop := 1
      loop := temploop
   }
}
if (finishedloop) {
   POSTLOOP_CODE
}
```

(Applying the standard syntactic sugar transformations to convert the conditionals into NAND code.) We can apply this transformation repeatedly to convert programs with multiple loops, and even nested loops, into a standard NAND++ program.

P Please stop and verify that you understand why this transformation will result in a program that computes the same function as the original code with an inner loop.

7.2.2 Controlling the index variable

NAND++ is an *oblivious* programming model, in the sense that it gives us no means of controlling the index variable `i`. Rather to read, for example, the 1017-th index of the array `foo` (i.e., `foo_1017`) we need to wait until `i` will equal 1017.[6] However we can use syntactic sugar to simulate the effect of incrementing and decrementing `i`. That is, rather than having `i` move according to a fixed schedule, we can assume that we have the operation `i++ (foo)` that increments `i` if `foo` is equal to 1 (and otherwise leaves `i` in place), and similarly the operation `i- (bar)` that decrements `i` if `bar` is 1 and otherwise leaves `i` in place.

[6] Note that we *can* use variables with absolute numerical indices in the program, but they can only let us access a fixed number of locations (in particular smaller than the number of lines in the program). Since in NAND++ we typically think of inputs that are much longer than the number of lines, in general we will have to use the index variable `i` to access most of the memory locations.

To achieve this, we start with the observation that in a NAND++ program we can know whether the index is increasing or decreasing. We achieve this using the Hansel and Gretel technique of leaving *"breadcrumbs"*. Specifically, we create an array `atstart` such that `atstart_0` equals 1 but `atstart_`$\langle j \rangle$ equals 0 for all $j > 0$, and an array `breadcrumb` where we set `breadcrumb_i` to 1 in every iteration. Then we can setup a variable `indexincreasing` and set it to 1 when we reach the zero index (i.e., when `atstart_i` is equal to 1) and set it to 0 when we reach the end point (i.e., when we see an index for which `breadcrumb_i` is 0 and hence we have reached it for the first time). We can also maintain an array `arridx` that contains 0 in all positions except the current value of `i`.

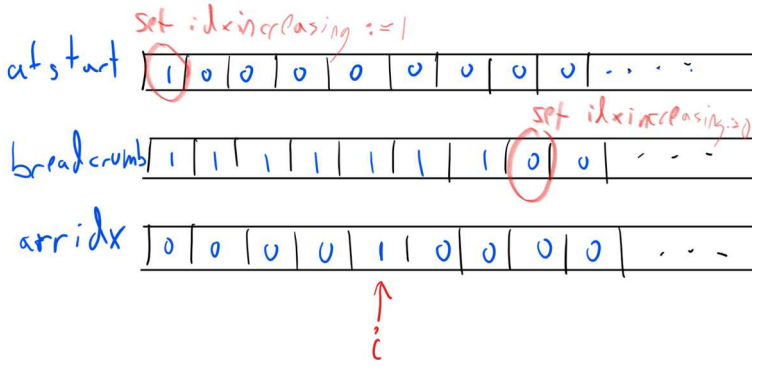

Figure 7.2: We can simulate controlling the index variable `i` by keeping an array `atstart` letting us know when `i` reaches 0, and hence `i` starts increasing, and `breadcrumb` letting us know when we reach a point we haven't seen before, and hence `i` starts decreasing. If we are at a point in which the index is increasing but we want it to decrease then we can mark our location on a special array `arridx` and enter a loop until the time we reach the same location again.

Now we can simulate incrementing and decrementing `i` by one by simply waiting until our desired outcome happens naturally. (This is similar to the observation that a bus is like a taxi if you're willing to wait long enough.) That is, if we want to increment `i` and

`indexincreasing` equals 1 then we simply wait one step. Otherwise (if `indexincreasing` is 0) then we go into an inner loop in which we do nothing until we reach again the point when `arridx_i` is 1 and `indexincreasing` is equal to 1. Decrementing `i` is done in the analogous way.[7]

[7] It can be verified that this transformation converts a program with T steps that used the `i++ (foo)` and `i- (bar)` operations into a program with $O(T^2)$ that doesn't use them.

7.2.3 *"Simple" NAND++ programs*

When analyzing NAND++ programs, it will sometimes be convenient for us to restrict our attention to programs of a somewhat nicer form.

Definition 7.2 — Simple NAND++ programs. We say that a NAND++ program P is *simple* if it has the following properties:

- The only output variable it ever writes to is `y_0` (and so it computes a Boolean function).
- The last line of the program has the form `halted := loop NAND loop` and so the variable `halted` gets the value 1 when the program halts. Moreover, there is no other line in the program that writes to the variable `halted`.
- All lines that write to the variable `loop` or `y_0` are "guarded" by `halted` in the sense that we replace a line of the form `y_0 := foo NAND bar` with the (unsweetened equivalent to) `if NOT(halted) y_0 := foo NAND bar` and similarly `loop := blah NAND baz` is replaced with `if NOT(halted) loop := blah NAND baz .`
- It has an `indexincreasing` variable that is equal to 1 if and only if in the next iteration the value of `i` will increase by 1.
- It contains variables `zero` and `one` that are initialized to be 0 and 1 respectively, by having the first line be `one := zero NAND zero` and having no other lines that assign values to them.

Note that if P is a simple program then even if we continue its execution beyond the point it should have halted in, the value of the `y_0` and `loop` variables will not change. The following theorem shows that, in the context of Boolean functions, we can assume that every program is simple:[8]

Theorem 7.3 — Simple program. Let $F : \{0,1\}^* \to \{0,1\}$ be a (possibly partial) Boolean function. If there is a NAND++ program that computes F then there is a simple NAND++ program P' that computes

[8] The restriction to Boolean functions is not very significant, as we can always encode a non Boolean function $F : \{0,1\}^* \to \{0,1\}^*$ by the Boolean function $G(x,i) = F(x)_i$ where we treat the second input i as representing an integer. The crucial point is that we still allow the functions to have an unbounded *input length* and hence in particular these are functions that cannot be computed by plain "loop less" NAND programs.

F as well.

Proof. We only sketch the proof, leaving verifying the full details to the reader. We prove the theorem by transforming the code of the program P to achieve a simple program P' without modifying the functionality of P. If P computes a Boolean function then it cannot write to any $\texttt{y\_}\langle j \rangle$ variable other than `y_0`. If P already used a variable named `halted` then we rename it. We then we add the line `halted := loop NAND loop` to the end of the program, and replace all lines writing to the variables `y_0` and `loop` with their "guarded" equivalents. Finally, we ensure the existence of the variable `indexincreasing` using the "breadcrumbs" technique discussed above. ■

7.3 *Uniformity, and NAND vs NAND++*

While NAND++ adds an extra operation over NAND, it is not exactly accurate to say that NAND++ programs are "more powerful" than NAND programs. NAND programs, having no loops, are simply not applicable for computing functions with more inputs than they have lines. The key difference between NAND and NAND++ is that NAND++ allows us to express the fact that the algorithm for computing parities of length-100 strings is really the same one as the algorithm for computing parities of length-5 strings (or similarly the fact that the algorithm for adding n-bit numbers is the same for every n, etc.). That is, one can think of the NAND++ program for general parity as the "seed" out of which we can grow NAND programs for length 10, length 100, or length 1000 parities as needed. This notion of a single algorithm that can compute functions of all input lengths is known as *uniformity* of computation and hence we think of NAND++ as *uniform* model of computation, as opposed to NAND which is a *nonuniform* model, where we have to specify a different program for every input length.

Looking ahead, we will see that this uniformity leads to another crucial difference between NAND++ and NAND programs. NAND++ programs can have inputs and outputs that are longer than the description of the program and in particular we can have a NAND++ program that "self replicates" in the sense that it can print its own code.

This notion of "self replication", and the related notion of "self reference" is crucial to many aspects of computation, as well of course to life itself, whether in the form of digital or biological programs.

7.3.1 Growing a NAND tree

If P is a NAND++ program and $n, T \in \mathbb{N}$ are some numbers, then we can easily obtain a NAND program $P' = expand_{T,n}(P)$ that, given any $x \in \{0,1\}^n$, runs T loop iterations of the program P and outputs the result. If P is a simple program, then we are guaranteed that, if P does not enter an infinite loop on x, then as long as we make T large enough, $P'(x)$ will equal $P(x)$. To obtain the program P' we can simply place T copies of the program P one after the other, doing a "search and replace" in the k-th copy of any instances of `_i` with the value $index(k)$, where the function $index$ is defined as in Eq. (7.4). For example, Fig. 7.3 illustrates the expansion of the NAND++ program for parity.

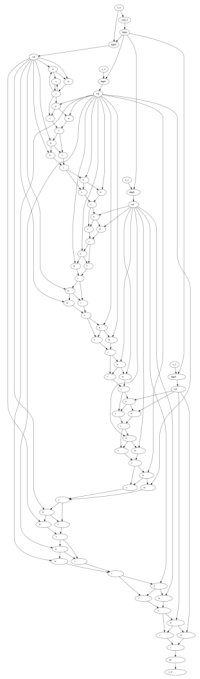

Figure 7.3: The circuit corresponding to a NAND program for parity obtained by expanding the NAND++ program

We can also obtain such an expansion by using the `for .. do ..` syntactic sugar. For example, the NAND program below corresponds to running the parity program for 17 iterations, and computing $XOR_5 : \{0,1\}^5 \to \{0,1\}$. Its standard "unsweetened" version will have $17 \cdot 10$ lines.[9]

```
for i in [0,1,0,1,2,1,0,1,2,3,2,1,0,1,2,3,4] do {
tmp1 := seen_i NAND seen_i
tmp2 := x_i NAND tmp1
val := tmp2 NAND tmp2
```

[9] This is of course not the most efficient way to compute XOR_5. Generally, the NAND program to compute XOR_n obtained by expanding out the NAND++ program will require $\Theta(n^2)$ lines, as opposed to the $O(n)$ lines that is possible to achieve directly in NAND. However, in most cases this difference will not be so crucial for us.

```
ns := s NAND s
y_0 := ns NAND ns
u := val NAND s
v := s NAND u
w := val NAND u
s := v NAND w
seen_i := zero NAND zero
}
```

In particular we have the following theorem

Theorem 7.4 — Expansion of NAND++ to NAND. For every simple NAND++ program P and function $F : \{0,1\}^* \to \{0,1\}$, if P computes F then for every $n \in \mathbb{N}$ there exists $T \in \mathbb{N}$ such that $expand_{T,n}(P)$ computes F_n.

```
# Expand a NAND++ program and a given time bound T and n to
    an n-input T-line NAND program
def expand(P,T,n):
   result = ""

   for k in range(T):
      i=index(k)
      validx = ('one' if i<n else 'zero')
      result += P.replace('validx_i',validx).replace('x_i',(
          'x_i' if i<n else 'zero')).replace('_i','_'+str(i))

   return result


def index(k):
   r = math.floor(math.sqrt(k+1/4)-1/2)
   return (k-r*(r+1) if k <= (r+1)*(r+1) else (r+1)*(r+2)-k)
```

Proof. We'll start with a "proof by code". Above is a Python program `expand` to compute $expand_{T,n}(P)$. On input the code P of a NAND++ program and numbers T, n, `expand` outputs the code of the NAND program P' that works on length n inputs and is obtained by running T iterations of P:

If the original program had s lines, then for every $\ell \in [sT]$, line ℓ in the output of `expand(P,T,n)` corresponds exactly to the line executed in step ℓ of the execution $P(x)$.[10] Indeed, in step ℓ of the execution of $P(x)$, the line executed is $k = \ell \bmod s$, and line ℓ in the

[10] In the notation above (as elsewhere), we index both lines and steps from 0.

output of `expand(P,T,n)` is a copy of line k in P. If that line involved unindexed variables, then it is copied as is in the returned program `result`. Otherwise, if it involved the index `_i` then we replace `i` with the current value of i. Moreover, we replace the variable `validx_i` with either `one` or `zero` depending on whether $i < n$.

Now, if a simple NAND++ program P computes some function $F : \{0,1\}^* \to \{0,1\}$, then for every $x \in \{0,1\}^*$ there is some number $T_P(x)$ such that on input x halts within $T(x)$ iterations of its main loop and outputs $F(x)$. Moreover, since P is simple, even if we run it for more iterations than that, the output value will not change. For every $n \in \mathbb{N}$, define $T_P(n) = \max_{x \in \{0,1\}^n} T(x)$. Then $P' = expand_{T_P(n),n}(P)$ computes the function $F_n : \{0,1\}^n \to \{0,1\}$ which is the restriction of F to $\{0,1\}^n$. ■

7.4 NAND++ Programs as tuples

Just like we did with NAND programs, we can represent NAND++ programs as tuples. A minor difference is that since in NAND++ it makes sense to keep track of indices, we will represent a variable `foo_`$\langle j \rangle$ as a pair of numbers (a, j) where a corresponds to the identifier `foo`. Thus we will use a 6-tuple of the form (a, j, b, k, c, ℓ) to represent each line of the form `foo_`$\langle j \rangle$ `:= bar_`$\langle k \rangle$ `NAND baz_`$\langle \ell \rangle$, where a, b, c correspond to the variable identifiers `foo`, `bar` and `baz` respectively.[11] If one of the indices is the special variable `i` then we will use the number s for it where s is the number of lines (as no index is allowed to be this large in a NAND++ program). We can now define NAND++ programs in a way analogous to Definition 3.1:

[11] This difference between three tuples and six tuples is made for convenience and is not particularly important. We could have also represented NAND programs using six-tuples and NAND++ using three-tuples. Also recall that we use the convention that an unindexed variable identifier `foo` is equivalent to `foo_0`.

Definition 7.5 — NAND++. A *NAND++ program* is a 6-tuple $P = (V, X, Y, VALIDX, LOOP, L)$ of the following form:

- V (called the *variable identifiers*) is some finite set.
- $X \in V$ is called the *input identifier*.
- $Y \in V$ is called the *output identifier*.
- $VALIDX \in V$ is the *input length identifier*.
- $LOOP \in V$ is the *loop variable*.
- $L \in (V \times [s+1] \times V \times [s+1] \times V \times [s+1])^*$ is a list of 6-tuples of the form (a, j, b, k, c, ℓ) where $a, b, c \in V$ and $j, k, \ell \in [s+1]$ for $s = |L|$. That is, $L = ((a_0, j_0, b_0, k_0, c_0, \ell_0), \ldots, (a_{s-1}, j_{s-1}, b_{s-1}, k_{s-1}, c_{k-1}, \ell_{s-1}))$ where for every $t \in \{0, \ldots, s-1\}$, $a_t, b_t, c_t \in V$ and $j_t, k_t, \ell_t \in$

$[s+1]$. Moreover $a_t \notin \{X, VALIDX\}$ for every $t \in [s]$ and $b_t, c_t \notin \{Y, LOOP\}$ for every $t \in [s]$.

P This definition is long but ultimately translating a NAND++ program from code to tuples can be done in a fairly straightforward way. Please read the definition again to see that you can follow this transformation. Note that there is a difference between the way we represent NAND++ and NAND programs. In NAND programs, we used a different element of V to represent, for example, `x_17` and `x_35`. For NAND++ we will represent these two variables by $(X, 17)$ and $(X, 35)$ respectively where X is the input identifier. For this reason, in our definition of NAND++, X is a single element of V as opposed to a tuple of elements as in Definition 3.1. For the same reason, Y is a single element and not a tuple as well.

Just as was the case for NAND programs, we can define a *canonical form* for NAND++ variables. Specifically in the canonical form we will use $V = [t]$ for some $t > 3$, $X = 0$,$Y = 1$,$VALIDX = 2$ and $LOOP = 3$. Moreover, if P is *simple* in the sense of Definition 7.2 then we will assume that the `halted` variable is encoded by 4, and the `indexincreasing` variable is encoded by 5. The canonical form representation of a NAND++ program is specified simply by a length s list of 6-tuples of natural numbers (a, j, b, k, c, ℓ) where $a, b, c \in [t]$ and $j, k, \ell \in [s+1]$.

Here is a Python code to evaluate a NAND++ program given the list of 6-tuples representation:

```
# Evaluates a NAND++ program P on input x
# P is given in the list of tuples representation
# untested code
def EVALpp(P,x):
    vars = { 0:x , 2: [1]*len(x) } # vars[var][idx] is value
        of var_idx.
    # special variables: 0:X, 1:Y, 2:VALIDX, 3:LOOP
    t = len(P)

    def index(k): # compute i at loop j
        r = math.floor(math.sqrt(k+1/4)-1/2)
        return (k-r*(r+1) if k <= (r+1)*(r+1) else (r+1)*(r+2)
            -k)
```

```
def getval(var,idx): # returns current value of var_idx
   if idx== t: idx = index(k)
   l = vars.getdefault(var,[])
   return l[idx] if idx<len(l) else 0

def setval(var,idx,v): # sets var_idx := v
   l = vars.setdefault(var,[])
   l.append([0]*(1+idx-len(l)))
   l[idx]=v
   vars[var] = l

k = 0
while True:
   for t in P:
      setval(t[0],t[1], 1-getval(t[2],t[3])*getval(t[4],t
          [5]))
   if not getval(3,0): break
   k += 1

return vars[1]
```

7.4.1 *Configurations*

Just like we did for NAND programs, we can define the notion of a *configuration* and a *next step function* for NAND++ programs. That is, a configuration of a program P records all the state of P at a given point in the execution, and contains everything we need to know in order to continue from this state. The next step function of P maps a configuration of P into the configuration that occurs after executing one more line of P.

Before reading onwards, try to think how *you* would define the notion of a configuration of a NAND++ program.

While we can define configurations in full generality, for concreteness we will restrict our attention to configurations of "simple" programs NAND++ programs in the sense of Definition 7.2, that are given in a canonical form. Let P be a canonical form simple program, represented as a list of 6 tuples $L = ((a_0, j_0, b_0, k_0, c_0, \ell_0), \ldots, (a_{s-1}, j_{s-1}, b_{s-1}, k_{s-1}, c_{s-1}, \ell_{s-1}))$. Let s be the number of lines and t be one more than the largest number appearing among the a's, b's or c's.

Just like we did for NAND, a *configuration* of the program P will denote the current line being executed and the current value of all variables. For our convenience we will use a somewhat different encoding than we did for NAND. We will encode the configuration as a string $\sigma \in \{0,1\}^*$, which is composed of *blocks*, that is, σ will be the concatenation of $\sigma^0, \ldots, \sigma^{r-1}$ for some $r \in \mathbb{N}$ (that will represent the maximum among $n-1$, where n is the input length, the largest numerical index appearing in the program, and the largest index that the program has ever reached in the execution). Each block σ^i will be a string of length B (for some constant B depending on t, s) that encodes the following:

- The values of variables indexed by i (e.g., `foo_`$\langle i \rangle$, `bar_`$\langle i \rangle$, etc.).
- Whether or not the block is "active" (i.e., whether the current value of the index variable `i` is i), and in the latter case, the current line that is being executed.
- Whether this is the first or last block.

> R **High level points about configurations** For the sake of completeness, we will describe below precisely how configurations of NAND++ programs and the next-step function are defined. However, the details are as important as the high level points, which are the following: A configuration encodes all the information of the state of the program at a given step in the computation, including the values of all variables (both the Boolean variables and the special index variable `i`) and the current line number that is to be executed. The next step function of a program P updates that configuration by computing one line of the program, and updating the value of the variable that is assigned a value in this program. The variables involved in that line either have absolute numerical indices (in which case they are encoded in one of the first s blocks, as numerical indices can't be larger than the number of lines) or are indexed by the special variable `i` (in which case they are encoded in the active block). If the line is the last one in the program, the next step function also determines whether to halt based on the `loop` variable, and updates the active block based on whether the index will be increasing or decreasing.

We now describe a precise encoding for the configurations of a NAND++ program. Many of the choices below are made for convenience and other choices would be just as valid. We will think of encoding each block as using the alphabet $\Sigma = \{\texttt{BB}, \texttt{EB}, 0, 1\}$. (`BB` and `EB` stand for "begin block" and "end block" respectively; we can later

encode this as a binary string using the map $0 \mapsto 00, 1 \mapsto 11, \texttt{BB} \mapsto 01, \texttt{EB} \mapsto 10$.) In this alphebt Σ, every block σ^i will have the form

$$\sigma^i = \texttt{BB}\ \hat{\sigma}^i\ first\ last\ active\ p\ \texttt{EB} \tag{7.5}$$

where $\hat{\sigma}^i$ is a string in $\{0,1\}^t$ that encodes the values of all the variables in the program indexed by i. That is, the a-th coordinate of $\hat{\sigma}^i$ corresponds to the value of the variable represented by (a, i). For example, if we encode `foo` by the number 11 then $\hat{\sigma}^{17}_{11}$ corresponds to the value of `foo_17` at the given point in the execution. We use the same indexing of variables as in representations and so in particular coordinates $0, 1, 2, 3, 4, 5$ of $\hat{\sigma}^i$ correspond to the variables `x_i`,`y_i`,`validx_i`,`loop_i`,`halted_i`,`indexincreasing_i` respectively.[12]

[12] Recall that we identify an unindexed variable identifier such as `foo` with `foo_0`, and so in particular the values of `loop`, `halted` and `indexincreasing` are encoded in the block σ^0.

The values *active*, *first*, and *last* are each bits that are set to 1 or 0 depending on whether the current block is *active* (i.e. the current value of `i` is i), is the *first* block in the configuration and the *last* block, respectively. The parameter p is a string in $\{0,1\}^{\lceil \log(s+1) \rceil}$, which (via the binary representation) we think of also as number in $[s+1]$. The value of p is equal to the current line that is about to be executed if the block is active, and to 0 if the block is not active. If $p = s$ then this means that we have halted.

Note that in the alphabet Σ, our encoding takes 2 symbols for `BB` and `EB`, t symbols for $\hat{\sigma}^i$, three symbols for *first*,*last*,*active*, and $\log\lceil s+1 \rceil$ symbols for encoding p. Hence in the binary alphabet, each block σ^i will be encoded as a string of length $B = 2(5 + t + \log(\lceil s+1 \rceil))$ bits, and a configuration will be encoded as a binary string of length $(r+1)B$ where r is the largest index that the variable `i` has reached so far in the execution. See Fig. 7.4 for an illustration of the configuration.

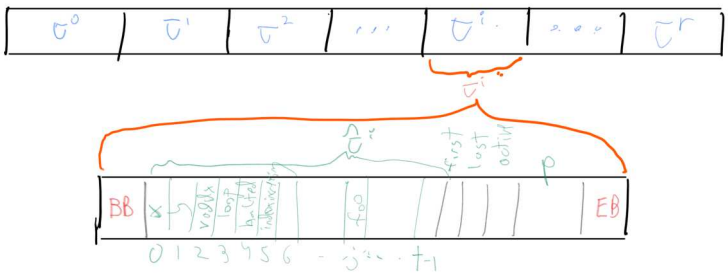

Figure 7.4: A configuration of an s-line t-variable simple NAND++ program can be encoded as a string in $\{0,1\}^{rB}$, the i-th block encodes the value of all variables of the form `foo_`$\langle i \rangle$, as well as whether the block is first, last or active in the sense that `i`=i and in the latter case, also the index of the current line being executed.

For a simple s-line t-variable NAND++ program P the *next configuration function* $NEXT_P : \{0,1\}^* \to \{0,1\}^*$ is defined in the natural way.[13] That is, on input a configuration σ, one can compute $\sigma' = NEXT_P(\sigma)$ as follows:

1. Scan the configuration σ to find the index i of the active block (block where the *active* bit is set to 1) and the current line p that needs to be executed (which is enc). We denote the new active block and current line in the configuration σ' by (i', p').

2. If $p = s$ then this σ a halting configuration and $NEXT_p(\sigma) = \sigma$. Otherwise we continue to the following steps:

3. Execute the line p: if the p-th tuple in the program is (a, j, b, k, c, ℓ) then we update σ to σ' based on the value of this program. That is, in the configuration σ', we encode the value of of the variable corresponding to (a, j) as the NAND of the values of variables corresponding to (b, k) and (c, ℓ).[14]

4. Updating the value of i: if $p = s - 1$ (i.e., p corresponds to the last line of the program), then we check whether the value of the `loop` or `loop_0` variable (which by our convention is encoded as the variable with index 3 in the first block) and if so set in σ' the value $p' = s$ which corresponds to a halting configuration. Otherwise, i is either incremented and decremented based on `indexincreasing` (which we can read from the first block). That is, we let i' be either $i + 1$ and $i - 1$ based on `indexincreasing` and modify the active block in σ' to be i'. (If i is the final block and $i' = i + 1$ then we create a new block and mark it to be the last one.)

5. We update $p' = p + 1 \mod s$, and encode p' in the active block of σ'.

One important property of $NEXT_P$ is that to compute it we only need to access the blocks $0, \ldots, s - 1$ (since the largest absolute numerical index in the program is at most $s - 1$) as well as the current active block and its immediate neighbors. Thus in each step, $NEXT_P$ only reads or modifies a constant number of blocks.

Here is some Python code for the next step function:

```
# compute the next-step configuration
# Inputs:
# P: NAND++ program in list of 6-tuples representation (
    assuming it has an "indexincreasing" variable)
# conf: encoding of configuration as a string using the
    alphabet "B","E","0","1".
def next_step(P,conf):
```

[13] We define $NEXT_P$ as a *partial* function, that is only defined on strings that are valid encoding of a configuration, and in particular have only a single block with its active bit set, and where the initial and final bits are also only set for the first and last block respectively. It is of course possible to extend $NEXT_P$ to be a total function by defining it on invalid configurations in some way.

[14] Recall that according to the way we represent NAND++ programs as 6-tuples, if a is the number corresponding to the identifier `foo` then (a, j) corresponds to `foo_`$\langle j \rangle$ if $j < s$, and corresponds to `foo_`$\langle i \rangle$ if $j = s$ where i is the current value of the index variable `i`.

```
s = len(P) # numer of lines
t = max([max(tup[0],tup[2],tup[4]) for tup in P])+1 #
    number of variables
line_enc_length = math.ceil(math.log(s+1,2)) # num of
    bits to encode a line
block_enc_length = t+3+line_enc_length # num of bits to
    encode a block (without bookends of "E","B")
LOOP = 3
INDEXINCREASING = 5
ACTIVEIDX = block_enc_length -line_enc_length-1 #
    position of active flag
FINALIDX = block_enc_length -line_enc_length-2 # position
     of final flag

def getval(var,idx):
   if idx<s: return int(blocks[idx][var])
   return int(active[var])

def setval(var,idx,v):
   nonlocal blocks, i
   if idx<s: blocks[idx][var]=str(v)
   blocks[i][var]=str(v)

blocks = [list(b[1:]) for b in conf.split("E")[:-1]] #
    list of blocks w/o initial "B" and final "E"

i = [j for j in range(len(blocks)) if blocks[j][ACTIVEIDX
    ]=="1" ][0]
active = blocks[i]

p = int("".join(active[-line_enc_length:]),2) # current
    line to be executed

if p==s: return conf # halting configuration

(a,j,b,k,c,l) = P[p] # 6-tuple corresponding to current
    line# 6-tuple corresponding to current line
setval(a,j,1-getval(b,k)*getval(c,l))

new_p = p+1
new_i = i
if p==s-1: # last line
   new_p = (s if getval(LOOP,0)==0 else 0)
   new_i = (i+1 if getval(INDEXINCREASING,0) else i-1)
```

```
        if new_i==len(blocks): # need to add another block and
             make it final
            blocks[len(blocks)-1][FINALIDX]="0"
            new_final = ["0"]*block_enc_length
            new_final[FINALIDX]="1"
            blocks.append(new_final)

        blocks[i][ACTIVEIDX]="0" # turn off "active" flag in
            old active block
        blocks[i][ACTIVEIDX+1:ACTIVEIDX+1+line_enc_length]=["0
            "]*line_enc_length # zero out line counter in old
            active block
        blocks[new_i][ACTIVEIDX]="1" # turn on "active" flag
            in new active block
    new_p_s = bin(new_p)[2:]
    new_p_s = "0"*(line_enc_length-len(new_p_s))+new_p_s
    blocks[new_i][ACTIVEIDX+1:ACTIVEIDX+1+line_enc_length] =
        list(new_p_s) # add binary representation of next line
         in new active block

    return "".join(["B"+"".join(block)+"E" for block in
        blocks]) # return new configuration
```

7.4.2 *Deltas*

Sometimes it is easier to keep track of merely the *changes* (sometimes known as "deltas") in the state of a NAND++ program, rather than the full configuration. Since every step of a NAND++ program assigns a value to a single variable, this motivates the following definition:

Definition 7.6 — Modification logs of NAND++ program. The *modification log* (or "deltas") of an s-line simple NAND++ program P on an input $x \in \{0,1\}^n$ is the string Δ of length $sT + n$ whose first n bits are equal to x and the last sT bits correspond to the value assigned in each step of the program. That is, for every $i \in [n]$, $\Delta_i = x_i$ and for every $\ell \in [sT]$, $\Delta_{\ell+n}$ equals to the value that is assigned by the line executed in step ℓ of the execution of P on input x, where T is the number of iterations of the loop that P does on input x.

If Δ is the "deltas" of P on input $x \in \{0,1\}^n$, then for every $\ell \in [Ts]$, Δ_ℓ is the same as the value assigned by line ℓ of the NAND

program $expand_{T',n}(P)$ where s is the number of lines in P, and for every T' which is at least the number of loop iterations that P takes on input x.

R **Snapshots and deltas: what you need to remember**

The details of the definitions of configuration and deltas are not as important as the main points which are:

* A *configuration* is the full state of the program at a certain point in the computation. Applying the $NEXT_P$ function to the current configuration yields the next configuration.

* Each configuration can be thought of as a string which is a sequence of constant-size *blocks*. The $NEXT_P$ function only depends and modifies a constant number of blocks: the t first ones, the current active block, and its two adjacent neighbors.

* The "delta" or "modification log" of computation is a succinct description of how the configuration changed in each step of the computation. It is simply the string Δ of length T such that for every $\ell \in T$, Δ_ℓ is denotes the value assigned in the ℓ-th step of the computation.

Both configurations and Deltas are technical ways to capture the fact that computation is a complex process that is obtained as the result of a long sequence of simple steps.

7.5 *Lecture summary*

- NAND++ programs introduce the notion of *loops*, and allow us to capture a single algorithm that can evaluate functions of any input length.

- Running a NAND++ program for any finite number of steps corresponds to a NAND program. However, the key feature of NAND++ is that the number of iterations can depend on the input, rather than being a fixed upper bound in advance.

- A *configuration* of a NAND++ program encodes the state of the program at a given point in the computation. The *next step function* of the program maps the current configuration to the next one.

7.6 Exercises

Exercise 7.1 — Compute index. Suppose that t is the "iteration counter" of a NAND++ program, in the sense that t is initialized to zero, and is incremented by one each time the program finishes an iteration and goes back to the first line. Prove that the value of the variable `i` is equal to $t - r(r+1)$ if $t \leq (r+1)^2$ and equals $(r+2)(r+1) - t$ otherwise, where $r = \lfloor \sqrt{t + 1/4} - 1/2 \rfloor$. ■

7.7 Bibliographical notes

The notion of "NAND++ programs" we use is nonstandard but (as we will see) they are equivalent to standard models used in the literature. Specifically, NAND++ programs are closely related (though not identical) to *oblivious one-tape Turing machines*, while NAND« programs are essentially the same as RAM machines. As we've seen in these lectures, in a qualitative sense these two models are also equivalent to one another, though the distinctions between them matter if one cares (as is typically the case in algorithms research) about polynomial factors in the running time.

7.8 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

7.9 Acknowledgements