

## 1.2 TM Language Conversion

Let  $M = (\Gamma, Q, \delta)$  and  $\tilde{M} = (\tilde{\Gamma}, \tilde{Q}, \tilde{\delta})$ , and suppose  $M$  has  $k$  tapes. For each transition  $\delta(q, a_1, \dots, a_k) = (q', a'_2, \dots, a'_k, D_1, \dots, D_k)$ , we put states and transitions into  $\tilde{M}$  to create a “read”, “write”, and “move” section.

- The “read” section reads the binary representations of  $a_1, \dots, a_k$  from the tape heads while moving left to right. The states have the form  $q_{z_1 \dots z_k}$ , where each  $z_i$  is a binary string of length at most  $\log |\Gamma|$  representing the string that’s been read from tape  $i$  so far. The transitions are:

$$\tilde{\delta}(q_{z_1 \dots z_k}, b_1, \dots, b_k) = (q_{y_1 \dots y_k}, b_2, \dots, b_k, R, \dots, R), \text{ where } y_i = z_i b_i \text{ (when } |z_1| = \log |\Gamma| - 1, \text{ the } R\text{'s actually become } S\text{'s)}$$

- The “write” section writes the binary representations of  $a'_2, \dots, a'_k$  to the  $k - 1$  work/output tapes from right to left. These states have the form  $q_{a'_2 \dots a'_k n}$ , where  $n$  indicates the number of bits written so far. The transitions are:

$$\tilde{\delta}(q_{a'_2 \dots a'_k n}, -, \dots, -) = (q_{a'_2 \dots a'_k (n+1)}, b_2, \dots, b_k, L, \dots, L), \text{ where } b_i = j^{\text{th}} \text{ bit of } a'_i \text{ and } j = \log |\Gamma| - n \text{ (when } n = \log |\Gamma| - 1, \text{ the } L\text{'s actually become } S\text{'s)}$$

- The “move” section moves the tape heads left or right or nowhere as directed by the  $D_k$ ’s. Note that since the tape heads are at the start of the binary representations of the  $a_i$ ’s at this point, we need to either stay put, move  $\log |\Gamma|$  cells left, or move  $\log |\Gamma|$  cells right. The states have the form  $q_{D_1 \dots D_k n}$ , where  $n$  is a counter up to  $\log |\Gamma|$ . The transitions are:

$$\tilde{\delta}(q_{D_1 \dots D_k n}, b_1, \dots, b_k) = (q_{D_1 \dots D_k (n+1)}, b_2, \dots, b_k, D_1, \dots, D_k)$$

Finally, we add transitions to connect the different sections together:

- Read  $\rightarrow$  Write

$$\tilde{\delta}(q_{z_1 \dots z_k}, b_1, \dots, b_k) = (q_{a'_2 \dots a'_k 0}, b_1, \dots, b_k, S, \dots, S), \text{ where } |z_1| = \log |\Gamma|$$

- Write  $\rightarrow$  Move

$$\tilde{\delta}(q_{a'_2 \dots a'_k \log |\Gamma|}, b_1, \dots, b_k) = (q_{D_1 \dots D_k 0}, b_1, \dots, b_k, S, \dots, S)$$

- Move  $\rightarrow$  Read

$$\tilde{\delta}(q_{D_1 \dots D_k \log |\Gamma|}, b_1, \dots, b_k) = (q_{\emptyset \dots \emptyset}, b_1, \dots, b_k, S, \dots, S)$$

Thus we see that if  $M$  runs in time  $T(n)$ , then  $\tilde{M}$  simulates  $M$  in time  $3(\log |\Gamma|+1)T(n)$  ( $\log |\Gamma|+1$  steps for each of the “read”, “write”, and “move” sections).

### 1.5 Oblivious TM

Use the same construction as in Claim 1.6 of the text: “The TM  $M'$  encodes the  $k$  tapes of  $M$  (including its input and output tapes) on a single tape by using locations  $1, k+1, 2k+1, \dots$  to encode the first tape, locations  $2, k+2, 2k+2, \dots$  to encode the second tape etc. For every symbol  $a$  in  $M$ 's alphabet,  $M'$  will contain both the symbol  $a$  and the symbol  $\hat{a}$ . In the encoding of each tape, exactly one symbol will be of the “hat type”, indicating that the corresponding head of  $M$  is positioned in that location.  $M'$  uses the input and output tape in the same way  $M$  does. To simulate one step of  $M$ , the machine  $M'$  makes two sweeps of its work tape: first it sweeps the tape in the left-to-right direction and records to its state register the  $k$  symbols that are hatted. Then  $M'$  uses  $M$ 's transition function to determine the new state, symbols, and head movements and sweeps the tape back in the right-to-left direction to update the encoding accordingly.” This can be accomplished by creating, for every state  $q$  in  $M$  a set of states  $q_{rp}^{s_1, \dots, s_k}$ , where the  $s_i$  encode each possible combination of the  $k$  tape head positions,  $r$  indicates that the machine is in the moving right phase, and  $p$  encodes what tape it is currently reading. At step  $i$ ,  $M$  could have in the worst case accessed the  $i^{\text{th}}$  position on all of its tapes. Therefore, to ensure that on all inputs of the same length that the machine behaves the same, we'll use a special symbol  $*$  to denote the  $k * (i - 1)$  position on the tape and on every left to right pass we'll move the symbol  $k$  steps to the right, thereby ensuring that at time  $i$  on all inputs  $|x|$ ,  $M'$  moves exactly the same. This is true only up to the halting of the machine, i.e.  $M$  may take a different number of steps to halt on two different inputs of equal length. To correct this, we know that  $M$  is computable in time  $T(n)$  for some *time constructible*  $T$ . This means that there exists a machine  $M_T$  such that on all inputs of length  $n$ ,  $M_T$  halts in exactly  $T(n)$  steps. We can simulate  $M$  and  $M_T$  simultaneously on the same input by adding  $m$  tapes (this includes input/output tapes) for simulation of  $M_T$  to  $M'$ . States of the machine  $M'$  can then be considered as pairs of states of the above form corresponding to  $M$  and  $M_T$ .  $M'$  will continue sweeping back and forth in the oblivious manner described above until  $M_T$  halts.

### 1.15 Unary Numbers

a.) To convert  $x$  from base  $b$  to base  $b'$  simply compute  $y = \sum_{i=0}^{|x|-1} b^i x_i$  using base  $b'$  arithmetic. We know that addition/multiplication (irrespective of base) is polynomial time by simply using the grade school algorithms. Thus, since we're doing a polynomial number of additions, the whole summation can be computed in polynomial time. Hence it's clear from the definition of reducibility that  $L_S^b \leq L_S^2$  and vice-versa, so  $L_S^b \in P \iff L_S^2 \in P$ .

b.)  $(n, l, k) \in \text{UNARYFACTORING}$  if there exists a prime  $p \in (l, k)$  such that  $p$  divides  $n$ . If the integers in the input are represented in unary then the input has length  $O(n + k + l)$ . We can test membership in the language in polynomial time by enumerating all integers in  $(l, k)$ , testing them for primality, and then testing if they divide  $n$ . Notice that testing if  $p$  is prime requires checking that the  $p - 1$  integers less than  $p$  do not divide it and divisibility can be tested via repeated subtraction. Both of these require time polynomial in  $n$ ,  $k$ , and  $l$ .

### 2.5 PRIMES $\in NP$

We construct the following  $NP$  machine for the problem: On input  $n$ ,

- Non-deterministically choose integers  $q_1, \dots, q_k$  and  $a_1, \dots, a_k$
- Test that  $\prod_i q_i = n - 1$
- For all  $i$ , check that  $a^{n-1} = 1 \pmod n$  and  $a^{(n-1)/q_i} \neq 1 \pmod n$ .
- Recursively test that  $q_i$  is prime until we get to known primes (e.g. 2, 3).

Notice that  $k$  is at most  $\log n$  since the least any factor can be is 2. Because modular exponents can be computed in  $\log n$  time using repeated squaring, the running time of the machine is given by the following recurrence:

$$T(n) = \sum_i T(q_i) + c \log n$$

I claim that  $T(n)$  is  $O(\log n)$  (this can be made formal with induction, but here is the inductive step):

$$\begin{aligned} T(n) &= \sum_i T(q_i) + c \log n \\ &= \sum_i O(\log q_i) + c \log n \\ &= O(\log(n-1)) + c \log n \\ &= O(\log n) \end{aligned}$$

[Note that this analysis assumes unit time arithmetic to simplify the presentation. This assumption can be removed by using the observation that  $\sum_i \log^x q_i \leq \log^x(\prod_i q_i)$ .]

## 2.10 Combining NP Languages

Since  $L_1, L_2 \in NP$ :

$$x \in L_1 \iff \exists u \in \{0, 1\}^{p_1(|x|)}. M_1(x, u) = 1$$

$$x \in L_2 \iff \exists u \in \{0, 1\}^{p_2(|x|)}. M_2(x, u) = 1$$

Therefore:

$$x \in L_1 \cup L_2 \iff \exists u \in \{0, 1\}^{p_1(|x|)+p_2(|x|)}. M_1(x, u_1) = 1 \vee M_2(x, u_2) = 1$$

$$x \in L_1 \cap L_2 \iff \exists u \in \{0, 1\}^{p_1(|x|)+p_2(|x|)}. M_1(x, u_1) = 1 \wedge M_2(x, u_2) = 1$$

where  $u_1$  is the first  $p_1(|x|)$  bits of  $u$ , and  $u_2$  is the rest. Since  $M_1$  and  $M_2$  are both deterministic poly-time machines, clearly we can combine them using  $\vee$  or  $\wedge$  to form another deterministic poly-time machine. Hence  $L_1 \cup L_2 \in NP$  and  $L_1 \cap L_2 \in NP$ .

## 2.13b Parsimonious Reduction from SAT to 3SAT

We define parsimonious reduction  $f : SAT \rightarrow 3SAT$  as a repeated application of function  $g$ , where  $g$  maps each clause  $C = x_1 \vee \dots \vee x_k$  in  $SAT$  instance  $x$  to a set of clauses as follows:

- If  $k < 4$ ,  $g(C) = C$
- Else,  $g(C) = (z \vee x_1 \vee x_2) \wedge (x_1 \rightarrow \bar{z}) \wedge (x_2 \rightarrow \bar{z}) \wedge g(\bar{z} \vee x_3 \vee \dots \vee x_k)$   
 $= (z \vee x_1 \vee x_2) \wedge (\bar{z} \vee \bar{x}_1) \wedge (\bar{z} \vee \bar{x}_2) \wedge g(\bar{z} \vee x_3 \vee \dots \vee x_k)$

The idea here is that it's just like the reduction in the book, except we force  $z$  to be false whenever either  $x_1$  or  $x_2$  is true. This guarantees that whenever  $x$  is satisfied by some assignment of variables, there is a unique extension to that assignment that satisfies  $f(x)$ . If  $x_1$  and  $x_2$  are both false,  $z$  is forced to true; otherwise,  $z$  is forced to false. Furthermore, if we take any satisfying assignment of  $g(C)$  and remove all the assignments to new variables, it's clear by induction that the resulting assignment satisfies  $C$  (if  $z$  is false, then  $x_1 \vee x_2$  is true; if  $z$  is true, then  $x_3 \vee \dots \vee x_k$  is true by induction).

## 2.20 Berman's Theorem

Consider a Karp-reduction from  $3SAT$  instances to instances of unary language  $L$ . Because the reduction must be polynomial time, there's a  $c$  such that every  $3SAT$  instance of size  $n$  is mapped to a unary string  $1^i$  where  $i \leq n^c$ . Notice that if two instances of  $3SAT$  are both mapped to the same unary string then they are either both unsatisfiable or both satisfiable. Let  $R(\phi)$  be the unary string produced by the reduction.

Algorithm 1: *sat*

Input:  $\phi$  and an array of known results  $A$  (initialized to unknown before the first call)

1. if  $n = 0$  then return  $\phi$  (which must be either true or false)
2. if  $A[|R(\phi)|]$  is not equal to unknown then return  $A[|R(\phi)|]$
3. if  $\text{sat}(\phi(0, x_2, \dots, x_n), A) = \text{true}$  or  $\text{sat}(\phi(1, x_2, \dots, x_n), A) = \text{true}$  then
4.     set  $A[|R(\phi)|] = \text{true}$
5.     return true
6. else
7.     set  $A[|R(\phi)|] = \text{false}$
8.     return false

Because *SAT* is self-reducible, to see if a formula  $\phi(x_1, \dots, x_n)$  is satisfiable, we can check if either  $\phi(0, x_2, \dots, x_n)$  or  $\phi(1, x_2, \dots, x_n)$  is satisfiable. Algorithm 1 performs a depth first search of the search space for an instance of the *3SAT* problem. In the worst case, every  $2n$  recursive calls, we are guaranteed to learn the satisfiability of one more of the  $n^c$  possible unary strings because if one child of every node is known to be false and the other is unknown it will take  $2n$  steps to reach a leaf node so that the search can no longer advance down the tree. Therefore, after  $O(n^{c+1})$  recursive calls we are guaranteed to know which unary strings correspond to satisfying assignments and hence will then deduce the corresponding satisfiability of the original problem within  $n$  additional recursive calls. This algorithm solves *3SAT* in polynomial time and *3SAT* is NP-complete, so  $P = NP$ .