$f(n_1, n_2) = n_2 \quad \pi_2^2$

(start, #) → (R, 1, erasing1)
(erasing1, 1) → (R, #, erasing1)
(erasing1, #) → (R, forward2)
(forward2, 1) → ~~(forward2,~~ (R, forward2)
(forward2, #) → (L, backtrack)
(backtrack, 1) → (L, #, ~~ba~~ carry1)
(backtrack, #) → (
(carry1, 1) → (L, carry1)
(carry1, #) → (~~L~~, checknextBlank,)
(checkNextBlank, 1) → (R, #, finalCleaning)
(finalCleaning, #) → (finalMove, 1, L)
(finalMove, #) → halt

(checknextBlank, #) → (goRight, R)
(goRight, #) → (R, 1, forward2)

π_2^3 : Define a TM for $(n_1, n_2, n_3) = n_2$

square Checker ⟺ $\forall x \; (q(x) = x^2)$
zero Checker ⟺ $\forall x \; (p(x) = 0)$

zeroChecker (x) = squareChecker $(p(x) + x^2)$
($p(x)$ = 0, x^2 = x^2)

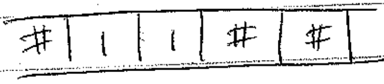
Since $\forall x$, $p(x) + x^2 = x^2$
$\Rightarrow p(x) = 0$.

9, 2009 why do we have to rewind? // composition.
- computable → have an algorithm
- PRecursive. $\langle 0, \sigma, \pi^k$, o, PR, μ-recursion

(+) TM → combination ⓒ

$n \to n+1$ (f)

$n \to n+2$ (g)



(start$_f$, #) → (right$_f$, R)
(right$_f$, 1) → (right$_f$, R)
(right$_f$, #) → (back$_f$, 1, L)
(back$_f$, 1) → (back$_f$, L)
(back$_f$, #) → halt.

(start$_g$, #) → (right$_g$, R)
(right$_g$, 1) → (right$_g$, R)
(right$_g$, #) → (putting 1$_g$, 1, R)
(putting 1$_g$, #) → (back$_g$, 1, L)
(back$_g$, 1) → (back$_g$, L)
(back$_g$, #) → halt.

$h(n) = g(f(n))$ composition.

+ Idea: first run TM for f and then
  run TM for g

We can't combine the rules of f and g
1) we have different conclusions for the same
2) Same names can be used on both TM with slightly different things
3) we can't have "halts" after the 1st TM.

+ Solution.
1) Rename states: start → start$_f$ (1st TM)
   start → start$_g$ ... (2nd)

2- change halt$_f$ → start$_g$
⇒ This is a combination. $f \circ g$.

(HW) 1) Take any two TMs and make a combination

(+) TM → Primitive Recursion (PR)

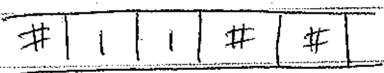
$h = PR(f, g)$

```
h = f
for (int i=0;  i<n;  i++)
{  h = g (x, f(x)) }
```

(+) TM → combination (c)

$n \to n+1$ (f)

$n \to n+2$ (g)



$(start_f, \#) \to (right_f, R)$
$(right_f, 1) \to (right_f, R)$
$(right_f, \#) \to (back_f, 1, L)$
$(back_f, 1) \to (back_f, L)$
$(back_f, \#) \to$ halt.

$(start_g, \#) \to (right_g, R)$
$(right_g, 1) \to (right_g, R)$
$(right_g, \#) \to (putting\,1_g, 1, R)$
$(putting\,1_g, \#) \to (back_g, 1, L)$
$(back_g, 1) \to (back_g, L)$
$(back_g, \#) \to$ halt.

$h(n) = g(f(n))$ composition.

+ Idea : first run TM for f and then run TM for g

We can't combine the rules of f and g
1) we have different conclusions for the same
2) Same names can be used on both TM with slightly different things.
3) we can't have "halts" after the 1st TM.

+ Solution.
1) Rename states : — start → $start_f$ (1st TM)
   start → $start_g$ ... (2nd)
2. change $halt_f \to start_g$
⇒ This is a combination. $f \circ g$.

1) Take any two TMs and make a combination.

(+) TM → Primitive Recursion (PR)

$h = PR(f, g)$

```
h = f
for (int i=0; i<n; i++)
{ h = g(x, f(x)) }
```

$$h(\vec{a}, 0) = f(\vec{a})$$
$$h(\vec{a}, n+1) = g(\vec{a}, h(\vec{a}, n))$$

#	$\vec{a}$	#	n	$\vec{a}$	n-1	#	$\vec{a}$	#	n-2	....	#	$\vec{a}$	#	0

#	$\vec{a}$	#	1	1	1	#	$\vec{a}$	#	1	1	#	$\vec{a}$

(+) TM → μ-recursion

$\mu m. P(\vec{a}, m)$: P: property returns < 1, 0

↓

apply while loop → TM.

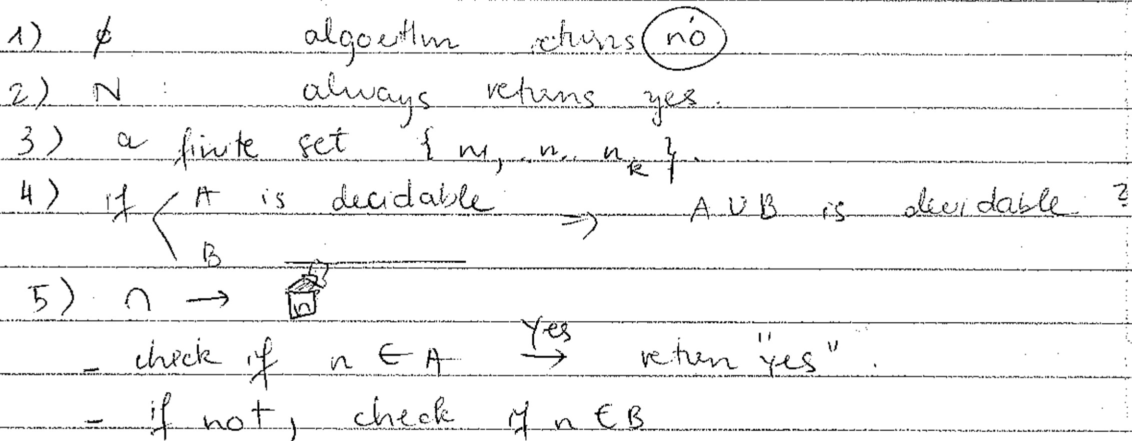
So far, we have two languages:
- Recursive fn
- TM : bitwise

There is another intuition: (sets)

$N = \{0, 1, 2, 3 \dots\}$

$A \subseteq N$

Def: A set 'A' is called "decidable" if there is an algorithm that given a natural # decides whether $n \in A$.

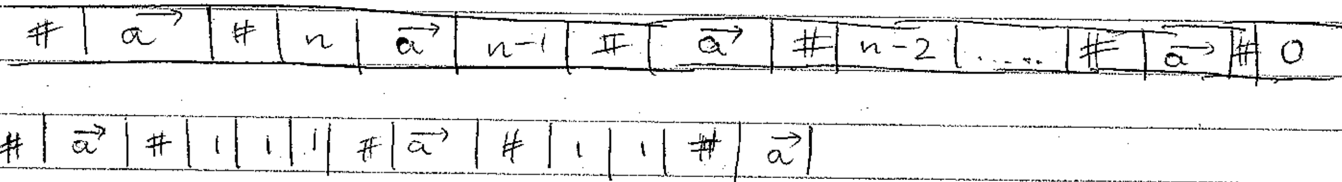


1) ϕ algorithm returns (no)
2) N : always returns yes.
3) a finite set $\{n_1, n_2 \dots n_k\}$.
4) if < A, B is decidable ⇒ $A \cup B$ is decidable ?
5) $\cap$ →
- check if $n \in A$ →(yes) return "yes".
- if not, check if $n \in B$

2. Prove if $A \cap B$ is decidable if < A, B is decidable

6) complement : if A is decidable.
is A^c is decidable ?

$$h(\vec{a}, 0) = f(\vec{a})$$
$$h(\vec{a}, n+1) = g(\vec{a}, h(\vec{a}, n))$$



(+) TM $\rightarrow$ μ-recursion

$\mu m. P(\vec{a}, m)$: P: property returns 1 / 0

↓

apply while loop $\rightarrow$ TM.

So far, we have two languages:

- Recursive fn
- TM : bitwise

There is another intuition: (sets)

$N = \{0, 1, 2, 3 \ldots\}$

$A \subseteq N$

Def: A set 'A' is called "decidable" if there is an algorithm that given a natural # decides whether $a \in A$.

1) ϕ algorithm returns (no)
2) N : always returns yes.
3) a finite set $\{n_1, \ldots n_2 \ldots n_k\}$.
4) if A, B is decidable $\Rightarrow$ $A \cup B$ is decidable ?
5) $\cap \rightarrow$ [house]

- check if $n \in A$ $\xrightarrow{Yes}$ return "yes".
- if not, check if $n \in B$

2. Prove if $A \cap B$ is decidable if A, B is decidable

6) complement : if A is decidable

is A^c is decidable ?

7) Is there any set that is not decidable ?

$\{ \langle p, d \rangle : p \text{ halts on } d \}$ → not decidable.

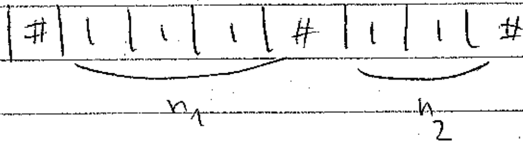
$\{ p : p \text{ always return } 0 \}$ → not decidable.

<u>Def</u>: A set 'A' is called recursively enumerable if there is an algorithm that eventually prints all elements of A.

```
n = 0;
while (true) {
        System.out.println();
    }
```

1. Design a TM that computes $f(n) = n+2$.
   // Use the 'state' wisely.

+ Function projection $\pi_1^2 = (n_1, n_2) \rightarrow n_1$



(start, #) → (R, in1right)
(in1right, 1) → (R, in1right)
(in1right, #) → (R, in2right)
(in2right, 1) → (R, in2right)
(in2right, #) → (L, #, erase2no)
replace

(erase2no, 1) → (L, erase2no)
(erase2no, #) → (L, in1left)
(in1left, 1) → (L, in1left)
(in1left, #) → halt

2. TM: π_1^3 $f(n_1, n_2, n_3) = n_1$
3. Extra credit: π_2^2 $f(n_1, n_2) = n_2$

(+)

Feb 17, 2009

What is the code of a Java program?
- Java program - is a sequence of ASCII symbols.
- In machine code, every ASCII symbols is a sequence of 0s and 1s.
- Put 1 in front
  Get a natural #, this # is called the code of a Java program.

What f_c?
let's start with a natural # c. We strip off 1s in front.

$f_c \rightarrow$ Java program corresponding to # c.

$$f(n) = \begin{cases} f_n(n) + 1 & \text{if } n \text{ is a valid code and } f_n \text{ halts on } n. \\ 0 & \text{otherwise} \end{cases}$$

check if :
Java program :

+ Assumption. A zero Checker exists.
− Proof : Build a ~~zero~~ halt checker based on a zero Checker.

$$\text{halts}(p, d, t) = \begin{cases} 1 : \text{if } p \text{ halts on } d \text{ by time } t \\ 0 \end{cases}$$

$$f_{p,d}(t) = \text{halts}(p, d, t)$$

$$!\,\text{haltChecker}(p,d) = \begin{cases} 1, & \text{if } f_{p,d}(t) = 0 \\ 0 & \text{otherwise.} \end{cases}$$

→ We need to use negation of halt Checker since we can't compute when a program does halt? Hence, we have to compute when a program ~~never~~ does not halts by some time t.

$$\Rightarrow \text{haltChecker}(p, d) = \begin{cases} !\,\text{zeroChecker}(f_{p,d}) \\ \end{cases}$$

$$!\,\text{haltChecker}(p, d) = \begin{cases} 1 \\ 0 \end{cases}$$