Equivalence of Turing Machine and μ-Recursive Functions

Motivations. We have mentioned earlier that in the 1930s, two different definitions of computability were proposed:

- Alan Turing described computability as computability on a Turing machine, while
- Alonzo Church defined computability in terms of what we now call μ-recursive functions.

Later on, Church proved that these two definitions are equivalent – i.e., that they describe the exact same notion of computability. This is what we will prove now.

Theorem. *A (partially defined) function is computable on a Turing machine if and only if it is μ-recursive.*

Comment. We have already argued that computability by a μ-recursive function is equivalent to computability by a Java program. Thus, what we are proving is that a function is computable by a Java program if and only if it is computable by a Turing machine.

As a result, in the description of the Church-Turing thesis, we can replace computability by a Java program with computability on a Turing machine.

Proof: structure. Every Turing machine can be relatively easily simulated by a Java program: this is what students from the Automata class do every semester. So, clearly, whatever function can be computed by a Turing machine can also be computed by a Java program – and thus, is μ-recursive.

So, to prove the theorem, it is sufficient to prove that, vice versa, every μ-recursive function can be computed on a Turing machine.

By definition, a μ-recursive function is any function that can be obtained from 0, σ, and π_i^k by using composition, primitive recursion, and μ-recursion. So, to prove that every μ-recursive function is computable on a Turing machine, it is sufficient to prove the following:

- that 0 can be computed on a Turing machine;
- that $\sigma(n) = n + 1$ can be computed on a Turing machine;

- that each projection $\pi_i^k(n_1, \ldots, n_i, \ldots, n_k) = n_i$ can be computed on a Turing machine;
- that if two functions $f(n)$ and $g(n)$ can be computed on a Turing machine, then their composition $(f \circ g)(n) = f(g(n))$ can also be computed on a Turing machine;
- that if function f and g can be computed on a Turing machine, then $PR(f, g)$ can be computed on a Turing machine; and
- that if a function $P(n_1, \ldots, n_k, m)$ can be computed on a Turing machine, then the function $\mu m.P(n_1, \ldots, n_k, m)$ can also be computed on a Turing machine.

Let us prove these statements one by one.

Proof that 0 can be computed on a Turing machine. We want a Turing machine that, no matter what is had originally on its tape, returns 0.

For simplicity, let us assume that all the numbers are presented in a unary code. Then, no matter what we start with:

$\underline{-}$	1	1	–	–	...	start

the desired final configuration should have the following form:

$\underline{-}$	–	–	–	–	...	halt

The idea is straightforward:

- We go right while we see 1s.
- When we see the first blank, we go into the state erasing and start going back and erasing all the 1s.
- Once we reach the initial blank cell, we stop.

Here are the corresponding rules:

start, – → R, moving
moving, 1 → R
moving, – → L, erasing
erasing, 1 → –, L
erasing, – → halt

Example. Let us show, step-by-step, how this Turing machine works in the above example:

$\underline{-}$	1	1	–	–	...	start
–	$\underline{1}$	1	–	–	...	moving
–	1	$\underline{1}$	–	–	...	moving

–	1	1	$\underline{-}$	–	...	moving
–	1	$\underline{1}$	–	–	...	erasing
–	$\underline{1}$	–	–	–	...	erasing
$\underline{-}$	–	–	–	–	...	erasing
$\underline{-}$	–	–	–	–	...	halt

Proof that σ can be computed on a Turing machine: we already showed it when we studied Turing machines.

How can we represent tuples. Projection π_i^k involves tuples $(n_1, \ldots, n_k)$. How can we represent a tuple on a Turing machine? The idea is straightforward:

- we place the number n_1,
- then we leave a blank space to separate it from the next number n_2,
- then we place the number n_2,
- then we leave a blank space to separate it from the next number n_3, etc.

For example, in the unary code, a tuple $(1, 2)$ is represented as follows:

–	1	–	1	1	–	...

Similarly, we can represent the pair $(1, 0)$. The only difference is that after the blank space separating n_1 and n_2, in this case, we place 0 – which in unary code means nothing (an empty string):

–	1	–	–	–	–	...

What about the pair $(0, 1)$? This may be somewhat less intuitive. To understand the resulting representation, let us follow the above procedure step-by-step. First, we place blank in the very first cell – as usual in Turing machines:

–						...

Then, according to the general algorithm, we place the number n_1. In our case, $n_1 = 0$, this means that we do not place anything and thus, end up with the same configuration as before:

–						...

After that, we place a blank space separating n_1 from n_2:

–	–					...

Finally, we place the number $n_2 = 1$:

–	–	1	–			...

Comment. One can see that the resulting representation of the tuple $(0, 1)$ is different from the above representation of the tuple $(1, 0)$ – which is good, since

this shows that we can uniquely reconstruct the tuple from its Turing machine representation.

Another example. Let us show how we can represent a triple $(1, 0, 2)$. First, we place blank in the very first cell – as usual in Turing machines:

–						...

Then, we place the number $n_1 = 1$:

–	1					...

Then, we place a blank space separating n_1 from n_2:

–	1	–				...

Then, we place the number $n_2 = 0$ – which means that we do not add anything:

–	1	–				...

After that, we place a blank space separating n_2 from n_3:

–	1	–	–			...

Finally, we place the number $n_3 = 2$:

–	1	–	–	1	1	– ...

Comment. Note that this is different from the representation of the triple $(1, 2, 0)$, that looks like this:

–	1	–	1	1	–	– ...

Proof that π_1^2 can be computed on a Turing machine. We need to prove that projections π_i^k are computable on a Turing machine.

We start with the case when $k = 2$ and $i = 1$. In this case, we start with two numbers – separated by a blank space – and return the first of the two numbers. Let us illustrate this on the example when $n_1 = 2$ and $n_2 = 3$. In this case, we start with the state

$\underline{-}$	1	1	–	1	1	1	–	...

start

and we need to return only the first number $n_1 = 2$:

$\underline{-}$	1	1	–	–	–	–	–	...

halt

The idea is straightforward:

- first, we go through the 1st number; we will call the corresponding state in1st;
- when we see blank, this means that we will start going through a second number; we will call the corresponding state in2nd;
- we go through the second state; once we are done, i.e., once we see the blank space again, this means that we can start erasing;

- we erase the second number one digit at a time;
- when we see the blank space separating the two numbers, this means that we will now go to the 1st number; so we stop erasing and go to the state back;
- in this state back, we go back one position at a time until we see the starting blank state, then we halt.

This can be described by the following Turing machine:

start, – → R, in1st
in1st, 1 → R
in1st, – → R, in2nd
in2nd, 1 → R
in2nd, – → L, erasing
erasing, 1 → –, L
erasing, – → back, L
back, 1 → L
back, – → halt

Example. Let us show, step-by-step, how this Turing machine works in the above example:

$\underline{-}$	1	1	–	1	1	1	–	...	start
–	$\underline{1}$	1	–	1	1	1	–	...	in1st
–	1	$\underline{1}$	–	1	1	1	–	...	in1st
–	1	1	$\underline{-}$	1	1	1	–	...	in1st
–	1	1	–	$\underline{1}$	1	1	–	...	in2nd
–	1	1	–	1	$\underline{1}$	1	–	...	in2nd
–	1	1	–	1	1	$\underline{1}$	–	...	in2nd
–	1	1	–	1	1	1	$\underline{-}$	...	in2nd
–	1	1	–	1	1	$\underline{1}$	–	...	erasing
–	1	1	–	1	$\underline{1}$	–	–	...	erasing
–	1	1	–	$\underline{1}$	–	–	–	...	erasing
–	1	1	$\underline{-}$	–	–	–	–	...	erasing
–	1	$\underline{1}$	–	–	–	–	–	...	back
–	$\underline{1}$	1	–	–	–	–	–	...	back
$\underline{-}$	1	1	–	–	–	–	–	...	back

$\underline{-}$	1	1	–	–	–	–	–	...	halt

Proof that π_2^2 can be computed on a Turing machine. Let us now consider the case when $k = 2$ and $i = 2$.

The Turing machine for computing π_2^2 is based on the following idea:

- first, we place 1 in the very first cell (to make sure that we will know when to stop when we get back),
- then, one by one, we eliminate all the ones from the 1st number,
- then, we go to the second number and continue going until we reach the first blank space after its end,
- we need to move the second number closer to the starting cell,
- moving a unary number one step to the left means that we erase the last 1, and add a 1 before this number; this will keep the same number of ones, but we get one step closer to the starting cell of the Turing machine,
- so, once we reach the blank space after the second number, we go back one step, erase the 1 symbol, and start going left,
- we go left until we reach the end of the number, i.e., the first blank space, which we replace by 1,
- if directly to the left of the replaced black space is a symbol 1, this means that we are at the starting cell of the Turing machine, thus we have moved the number already; now all we need to do is replace this symbol 1 with blank space and stop,
- on the other hand, if directly to the left of the replaced blank space is an empty cell, this means that we need to again go right and repeat the same move-one-step-to-the-left procedure.

A special care needs to be taken for a special case when the second component of the original pair is number 0. In this case, once we erase the 1st number, there is nothing left to erase, so we simply go back (and replace 1 back to blank when we reach the starting cell).

This Turing machine has the following main rules:

start, – $\to$ 1, erase1st, R,
erase1st, 1 $\to$ –, R,
erase1st, – $\to$ right, R,
right, 1 $\to$ R,
right, – $\to$ erase, L,
erase, 1 $\to$ –, left, L,
left, 1 $\to$ L,
left, – $\to$ 1, checking, L,
checking, – $\to$ right, R,
checking, 1 $\to$ –, halt,

The following three additional rules take care of the case when the second number is 0:

erase, – → finish, L,
finish, – → L,
finish, 1 → –, halt.

First example. Let us show, step-by-step, how this Turing machine works in the above example:

$\underline{-}$	1	1	–	1	1	1	–	...	start

First, we place 1 in the very first cell (to make sure that we will know when to stop when we get back), move to the next cell and prepare to start erasing the 1st number:

1	$\underline{1}$	1	–	1	1	1	–	...	erase1st

Then, we erase the first number symbol by symbol until we reach a black space:

1	–	$\underline{1}$	–	1	1	1	–	...	erase1st
1	–	–	$\underline{-}$	1	1	1	–	...	erase1st

Now, we move through the 2nd number:

1	–	–	–	$\underline{1}$	1	1	–	...	right
1	–	–	–	1	$\underline{1}$	1	–	...	right
1	–	–	–	1	1	$\underline{1}$	–	...	right
1	–	–	–	1	1	1	$\underline{-}$	...	right

We have reached the blank space after the second number, we go back one step, erase the 1 symbol, and start going left, until we reach the end of the number, i.e., the first blank space, which we replace by 1:

1	–	–	–	1	1	$\underline{1}$	–	...	erase
1	–	–	–	1	$\underline{1}$	–	–	...	left
1	–	–	–	$\underline{1}$	1	–	–	...	left
1	–	–	$\underline{-}$	1	1	–	–	...	left
1	–	$\underline{-}$	1	1	1	–	–	...	checking

Here, directly to the left of the replaced blank space is an empty cell. This means that we need to again go right and repeat the same move-one-step-to-the-left procedure:

1	–	–	$\underline{1}$	1	1	–	–	...	right

1	–	–	1	$\underline{1}$	1	–	–	...	right
1	–	–	1	1	$\underline{1}$	–	–	...	right
1	–	–	1	1	1	$\underline{–}$	–	...	right
1	–	–	1	1	$\underline{1}$	–	–	...	erase
1	–	–	1	$\underline{1}$	–	–	–	...	left
1	–	–	$\underline{1}$	1	–	–	–	...	left
1	–	$\underline{–}$	1	1	–	–	–	...	left
1	$\underline{–}$	1	1	1	–	–	–	...	checking

Here still, directly to the left of the replaced blank space is an empty cell. This means that we need to again go right and repeat the same move-one-step-to-the-left procedure:

1	–	$\underline{1}$	1	1	–	–	–	...	right
1	–	1	$\underline{1}$	1	–	–	–	...	right
1	–	1	1	$\underline{1}$	–	–	–	...	right
1	–	1	1	1	$\underline{–}$	–	–	...	right
1	–	1	1	$\underline{1}$	–	–	–	...	erase
1	–	1	$\underline{1}$	–	–	–	–	...	left
1	–	$\underline{1}$	1	–	–	–	–	...	left
1	$\underline{–}$	1	1	–	–	–	–	...	left
$\underline{1}$	1	1	1	–	–	–	–	...	checking

Now, directly to the left of the replaced black space is a symbol 1. This means that we are at the starting cell of the Turing machine, thus we have moved the number already. Now all we need to do is replace this symbol 1 with blank space and stop.

$\underline{–}$	1	1	1	–	–	–	–	...	halt

Second example. Let us now consider the case when the second number is 0:

$\underline{–}$	1	1	–	–	–	–	–	...	start

First, we place 1 in the very first cell (to make sure that we will know when to stop when we get back), move to the next cell and prepare to start erasing the 1st number:

1	$\underline{1}$	1	–	–	–	–	–	...	erase1st

Then, we erase the first number symbol by symbol until we reach a black space:

1	–	$\underline{1}$	–	–	–	–	–	...	erase1st
1	–	–	$\underline{-}$	–	–	–	–	...	erase1st
1	–	–	–	$\underline{-}$	–	–	–	...	right
1	–	–	$\underline{-}$	–	–	–	–	...	erase

The main rules – that we used earlier – did not even consider the situation when we want to erase, but there is nothing to erase. This simply means that the second word was empty, so we need to apply auxiliary rules:

erase, – $\to$ finish, L,
finish, – $\to$ L,
finish, 1 $\to$ –, halt.

By applying these rules, we get the following:

1	–	$\underline{-}$	–	–	–	–	–	...	finish
1	$\underline{-}$	–	–	–	–	–	–	...	finish
$\underline{1}$	–	–	–	–	–	–	–	...	finish
$\underline{-}$	–	–	–	–	–	–	–	...	halt

In other words, we get the desired value 0.

Composition. Composition $f(g(n))$ means that:

- we first apply the Turing machine for computing $g(n)$ to compute $m = g(n)$, i.e., want to go from the original state

$\underline{-}$	n	–	...

  to the state

$\underline{-}$	$g(n)$	–	...

- then, we apply the Turing machine for computing $f(n)$ to the resulting value m, resulting in $f(m) = f(g(n))$, i.e., we need to go from

$\underline{-}$	$g(n)$	–	...

  to the state

$\underline{-}$	$f(g(n))$	–	...

So, we do not want the g-Turing machine to stop after finishing its computations. Thus, what previously was the halt state will now be the start state of the f-Turing machine.

Comments.

- The descriptions of the two Turing machines may use the same name for two different states. To avoid confusion, a good idea is:
  - to mark all non-start states of the first Turing machine – for computing $g(n)$ – by a subscript 1; e.g., moving becomes moving_1; and
  - to mark all non-halt states of the second machine – for computing $f(n)$ – by a subscript 2.
- The need for composition explains why we need to "rewind" the Turing machine after its computations, i.e., why we need to make sure that at the end, the head is pointing to the very first blank cell: without this rewinding, composition would be difficult.

So, we arrive at the following rules for computing composition.

How to compute composition. To compute composition $f(g(n))$, we:

- rename all the non-start states of the g-machine by giving them all a subscript 1,
- rename all the non-halt states of the f-machine by giving them all a subscript 2, and
- replace the halt state of the g-machine with the start state of the f-machine.

Example. Let us illustrate this algorithm on the example of a Turing machine for computing $f(n) = n + 1$ that we had in the Turing machine lecture:

start, – → R, moving
moving, 1 → R
moving, – → 1, L, back
back, 1 → L
back, – → halt

and a similar Turing machine for computing $g(n) = n \dot{-} 1$:

start, – → R, testing
testing, – → L, halt
testing, 1 → R, moving
moving, 1 → R
moving, – → L, erasing
erasing, 1 → –, L, back
back, 1 → L
back, – → halt

According to the above general algorithm, we get the following Turing machine for computing the composition $f(g(n))$:

$$\begin{gathered}
\text{start}, - \to \text{R}, \text{testing}_1 \\
\text{testing}_1, - \to \text{L}, \text{start}_2 \\
\text{testing}_1, 1 \to \text{R}, \text{moving}_1 \\
\text{moving}_1, 1 \to \text{R} \\
\text{moving}_1, - \to \text{L}, \text{erasing}_1 \\
\text{erasing}_1, 1 \to -, \text{L}, \text{back}_1 \\
\text{back}_1, 1 \to \text{L} \\
\text{back}_1, - \to \text{start}_2 \\
\text{start}_2, - \to \text{R}, \text{moving}_2 \\
\text{moving}_2, 1 \to \text{R} \\
\text{moving}_2, - \to 1, \text{L}, \text{back}_2 \\
\text{back}_2, 1 \to \text{L} \\
\text{back}_2, - \to \text{halt}
\end{gathered}$$

Example. Let us trace this new machine on the example of $n = 2$:

$\underline{-}$	1	1	$-$	...	start
$-$	$\underline{1}$	1	$-$	...	testing_1
$-$	1	$\underline{1}$	$-$	...	moving_1
$-$	1	1	$\underline{-}$	...	moving_1
$-$	1	$\underline{1}$	$-$	...	erasing_1
$-$	$\underline{1}$	$-$	$-$	...	back_1
$\underline{-}$	1	$-$	$-$	...	back_1
$\underline{-}$	1	$-$	$-$	...	start_2
$-$	$\underline{1}$	$-$	$-$	...	moving_2
$-$	1	$\underline{-}$	$-$	...	moving_2
$-$	$\underline{1}$	1	$-$	...	back_2
$\underline{-}$	1	1	$-$	...	back_2
$\underline{-}$	1	1	$-$	...	halt

Copying. Implementation of primitive recursion and μ-recursion requires copying, so let us first describe how we can copy a number. Copying means that we start with a number n:

$\underline{-}$	n	$-$	$-$	$-$	... start

and we want to return a pair (n, n):

$\underline{=}$	n	–	n	–	...

halt

The main idea is that first, we copy symbols one by one. Every time we create a copy of a symbol, we mark this symbol as copied – e.g., by replacing each 1 with $\widehat{1}$. After copying a symbol, we:

- go back,
- find the first un-marked symbol,
- carry it to the first blank place,
- copy it there, etc.

Once we have copied all the symbols, we delete the copying marks, i.e., replace each marked symbol $\widehat{1}$ with the unmarked one 1.

Here are the corresponding rules. First, we start moving:

$$\text{start}, - \to \text{R}, \text{in1st}$$

If we see blank, this means that we had nothing to copy – the original number was 0. So, we go back and halt.

$$\text{in1st}, - \to \text{L}, \text{halt}$$

If we see 1, we mark it and go to the state carry1in1st:

$$\text{in1st}, 1 \to \widehat{1}, \text{R}, \text{carry1in1st}$$

We move step by step inside the 1st (original) number:

$$\text{carry1in1st}, 1 \to \text{R}$$

Once we reach a blank space, we know that after moving 1 step to the right we will be in the 2nd number:

$$\text{carry1in1st}, - \to \text{R}, \text{carry1in2nd}$$

As long as we see 1s, we continue going through the 2nd number:

$$\text{carry1in2nd}, 1 \to \text{R}$$

Once we see the first blank space, we drop 1 there and start going back:

$$\text{carry1in2nd}, - \to 1, \text{L}, \text{backin2nd}$$

We go left through the second number:

$$\text{backin2nd}, 1 \to \text{L}$$

Once we read the blank space separating the second number from the first one, we need to check if we are done:

$$\text{backin2nd}, - \to \text{L}, \text{checkIfDone}$$

If the symbol that we see in the 1st number is an unmarked symbol, this means that we are not done, so we need to go back and start finding the first unmarked symbols:

$$\text{checkIfDone}, 1 \to \text{L}, \text{backIn1st}$$

As we go left, if we see an unmarked symbol, we continue going left:

$$\text{backIn1st}, 1 \to \text{L}$$

Once we meet a marked symbol, this means that next one to the right if the first unmarked on. So we go to the state in1st to repeat the whole procedure:

$$\text{backIn1st}, \widehat{1} \to \text{R}, \text{in1st}$$

If the first symbol we see after getting into the 1st number is marked, this means that there are no more unmarked symbols in the 1st number. So, we unmark the marked symbol that we see and go to the unmark state:

$$\text{checkIfDone}, \widehat{1} \to 1, \text{L}, \text{unmark}$$

Then, we go left step by step and unmark all the symbols of the first number one by one:

$$\text{unmark}, \widehat{1} \to 1, \text{L}$$

Once we reach the very first (blank) cell of the Turing machine, this means that we are done. So we halt:

$$\text{unmark}, - \to \text{halt}$$

Let us show how this Turing machine will copy number 2.

$\underline{-}$	1	1	–	–	–	–	...	start
–	$\underline{1}$	1	–	–	–	–	...	in1st
–	$\widehat{1}$	$\underline{1}$	–	–	–	–	...	carry1in1st
–	$\widehat{1}$	1	$\underline{-}$	–	–	–	...	carry1in1st
–	$\widehat{1}$	1	–	$\underline{-}$	–	–	...	carry1in2nd
–	$\widehat{1}$	1	$\underline{-}$	1	–	–	...	backIn2nd
–	$\widehat{1}$	$\underline{1}$	–	1	–	–	...	checkIfDone
–	$\widehat{\underline{1}}$	1	–	1	–	–	...	backIn1st
–	$\widehat{1}$	$\underline{1}$	–	1	–	–	...	in1st
–	$\widehat{1}$	$\widehat{1}$	$\underline{-}$	1	–	–	...	carry1in1st

–	$\widehat{1}$	$\widehat{1}$	–	$\underline{1}$	–	–	...	carry1in2nd
–	$\widehat{1}$	$\widehat{1}$	–	1	$\underline{-}$	–	...	carry1in2nd
–	$\widehat{1}$	$\widehat{1}$	–	$\underline{1}$	1	–	...	backIn2nd
–	$\widehat{1}$	$\widehat{1}$	$\underline{-}$	1	1	–	...	backIn2nd
–	$\widehat{1}$	$\underline{\widehat{1}}$	–	1	1	–	...	checkIfDone
–	$\underline{\widehat{1}}$	1	–	1	1	–	...	unmark
$\underline{-}$	1	1	–	1	1	–	...	unmark
$\underline{-}$	1	1	–	1	1	–	...	halt

Comment.

- If we want to carry a binary number, we need to indicate which symbol we are carrying, i.e.:
  - instead of a state carry1in1st, we need two states: carry0in1st and carry1in1st;
  - instead of a state carry1in2nd, we need two states: carry0in2nd and carry1in2nd.
- Similarly, we can copy a tuple $(n_1, \ldots, n_k)$.

How to compute primitive recursion. Suppose that we have Turing machines computing functions $f(n_1, \ldots, n_k)$ and $g(n_1, \ldots, n_k, m, h)$. Let us show how to build a Turing machine that compute the function $h = PR(f, g)$, i.e., the function $h(n_1, \ldots, n_k, a)$ that is defined by the following equalities:

$$h(n_1, \ldots, n_k, 0) = f(n_1, \ldots, n_k);$$

$$h(n_1, \ldots, n_k, m+1) = g(n_1, \ldots, n_k, m, h(n_1, \ldots, n_k, m)).$$

We start with the state

$\underline{-}$	n_1	...	n_k	–	a	–	...	start

and we want to end up in the state

$\underline{-}$	$h(n_1, \ldots, n_k, a)$	–	...	halt

This can be done as follows. First, we copy a, add 0, then copy the tuple $(n_1, \ldots, n_k)$, and move the head into the cell right before the second copy of n_1:

–	n_1	...	n_k	–	a	–	a	–	0	$\underline{-}$	n_1	...	n_k	–	...

Then, we apply the Turing machine f. Since a Turing machine never goes beyond the cell where it starts, it will compute the value

$$h(n_1, \ldots, n_k, 0) = f(n_1, \ldots, n_k),$$

so we will have the following state of the tape:

$-$	n_1	$\ldots$	n_k	$-$	a	$-$	a	$-$	0	$\underline{-}$	$h(n_1, \ldots, n_k, 0)$	$-$	$\ldots$

Now, we copy $n_1, \ldots, n_k$, and 0 before h, and get

$-$	n_1	$\ldots$	n_k	$-$	a	$-$	a	$-$	0	$\underline{-}$	n_1	$\ldots$	n_k	$-$	0	$-$	$h(n_1, \ldots, n_k, 0)$	$-$	$\ldots$

Then, we apply the Turing machine for computing the function g, and get $h(n_1, \ldots, n_k, 1) = g(n_1, \ldots, n_k, 0, h(n_1, \ldots, n_k, 0))$. So, the tape has the form:

$-$	n_1	$\ldots$	n_k	$-$	a	$-$	a	$-$	0	$\underline{-}$	$h(n_1, \ldots, n_k, 1)$	$-$	$\ldots$

After that, we decrease the second copy of a by 1, increase 0 by 1, and get the following:

$-$	n_1	$\ldots$	n_k	$-$	a	$-$	$a-1$	$-$	1	$\underline{-}$	$h(n_1, \ldots, n_k, 1)$	$-$	$\ldots$

and we repeat a similar procedure.

In general, for each $m \leq a$, we get the following state of the tape:

$-$	n_1	$\ldots$	n_k	$-$	a	$-$	$a-m$	$-$	m	$\underline{-}$	$h(n_1, \ldots, n_k, m)$	$-$	$\ldots$

Then, we copy $n_1, \ldots, n_k$, and m before h, and get

$-$	n_1	$\ldots$	n_k	$-$	a	$-$	$a-m$	$-$	m	$\underline{-}$	n_1	$\ldots$	n_k	$-$	m	$-$	$h(n_1, \ldots, n_k, m)$	$-$	$\ldots$

Now, we apply the Turing machine for computing the function g, and get

$$h(n_1, \ldots, n_k, m+1) = h(n_1, \ldots, n_k, m, h(n_1, \ldots, n_k, m)).$$

So, the tape has the form:

–	n_1	…	n_k	–	a	–	$a-m$	–	m	$\underline{=}$	$h(n_1,\ldots,n_k,m+1)$	–	…

Then, we check whether $a - m = 0$. If $a - m > 0$, we decrease $a - m$ by 1, increase m by 1, and get the following:

–	n_1	…	n_k	–	a	–	$a-(m+1)$	–	$m+1$	$\underline{=}$	$h(n_1,\ldots,n_k,m+1)$	–	…

and we repeat a similar procedure.

Once we get $a - m = 0$ i.e., $m = a$, the state of the tape takes the form

–	n_1	…	n_k	–	a	–	0	–	a	$\underline{=}$	$h(n_1,\ldots,n_k,a)$	–	…

Here, we have $k + 4$ numbers:

- k numbers $n_1, \ldots, n_k$, and
- four numbers a, 0, a, and $h(n_1, \ldots, n_k, a)$.

The desired value $h(n_1, \ldots, n_k, a)$ is $(k + 4)$-th out of $k + 4$, so we can get it by applying the Turing machine computing the corresponding projection π^{k+4}_{k+4}:

$\underline{=}$	$h(n_1,\ldots,n_k,a)$	–	…	halt

This is exactly what we wanted.

In this construction, we use composition, adding 1, subtracting 1, copying, and projection. We know how to do all this on a Turing machine, so indeed we can thus build a Turing machine for computing the function $PR(f, g)$.

Computing μ-recursion. Suppose that we have a Turing machine for computing a function $P(n_1, \ldots, n_k, m)$. Let us show how we can compute the function

$$f(n_1, \ldots, n_k) = \mu m.P(n_1, \ldots, n_k, m)$$

that, given the tuple $(n_1, \ldots, n_k)$, returns the smallest integer m for which $P(n_1, \ldots, n_k)$ is true – i.e., $P(n_1, \ldots, n_k) = 1$.

We start with the state

$\underline{=}$	n_1	…	n_k	–	…	start

and we want to end up in the state

$\underline{=}$	$f(n_1,\ldots,n_k)$	–	…	halt

Let us show how this can be done. First, we add 0 after the input, copy the whole tuple $(n_1, \ldots, n_k, 0)$, and move the head before the second copy of n_1:

$\underline{=}$	n_1	…	n_k	–	0	$\underline{=}$	n_1	…	n_k	–	0	–	…

Then, we apply the Turing machine computing the function $P(n_1, \ldots, n_k, 0)$. As a result, we get the following state:

$-$	n_1	$\ldots$	n_k	$-$	0	$\underline{-}$	$P(n_1, \ldots, n_k, 0)$	$-$	$\ldots$

If $P(n_1, \ldots, n_k, 0) = 0$ (i.e., if the property $P(n_1, \ldots, n_k, m)$ is false), then we increase 0 by 1, copy the tuple $(n_1, \ldots, n_k, 1)$:

$-$	n_1	$\ldots$	n_k	$-$	1	$\underline{-}$	n_1	$\ldots$	n_k	$-$	1	$-$	$\ldots$

and again apply the Turing machine for computing $P(n_1, \ldots, n_k, m)$, resulting in:

$\underline{-}$	n_1	$\ldots$	n_k	$-$	1	$\underline{-}$	$P(n_1, \ldots, n_k, 1)$	$-$	$\ldots$

In general, at each iteration, we start with the state

$-$	n_1	$\ldots$	n_k	$-$	m	$\underline{-}$	$P(n_1, \ldots, n_k, m)$	$-$	$\ldots$

If $P(n_1, \ldots, n_k, m) = 0$ (i.e., to "false"), then we increase m by 1, copy the tuple $(n_1, \ldots, n_k, m+1)$:

$-$	n_1	$\ldots$	n_k	$-$	$m+1$	$\underline{-}$	n_1	$\ldots$	n_k	$-$	$m+1$	$-$	$\ldots$

and again apply the Turing machine for computing $P(n_1, \ldots, n_k, m+1)$, resulting in:

$-$	n_1	$\ldots$	n_k	$-$	$m+1$	$\underline{-}$	$P(n_1, \ldots, n_k, m+1)$	$-$	$\ldots$

etc.

This continues until we get the first value m for which $P(n_1, \ldots, n_k, m) = 1$ (i.e., "true"). In this case, we get the state

$-$	n_1	$\ldots$	n_k	$-$	m	$\underline{-}$	1	$-$	$\ldots$

Here, the desired value m is $(k+1)$-st out of $k+2$, so it can be found if we apply the corresponding projection π_{k+1}^{k+2}, resulting in:

$\underline{-}$	m	$-$	$\ldots$	halt

where $m = f(n_1, \ldots, n_k) = \mu m.P(n_1, \ldots, n_k, m)$.

This is exactly what we wanted.