CHAPTER 5

What's a computer, anyway? (Cont'd)

2. Recursive functions are defined one step at a time (Cont'd)

2.5. Full on recursive functions. So (if you were brave enough to read through the previous section) now you know of a function that we can see how to compute (and given enough time we could really write a register machine program for) which is not primitive recursive. This justifies the word *primitive* in the name! Now we'll define the actual recursive functions. The definition will start of rather similar, but there is a small caveat:

Caveat. We'll widen the discussion to *partial* functions from $\mathbb{N}^n \to \mathbb{N}$ (i.e. functional relations whose domain is a subset, possibly proper, of $\mathbb{N}^n$). Extending our previous notation we'll write $\mathcal{F}^\star_n$ for the n-ary partial functions on $\mathbb{N}$ and $\mathcal{F}^\star$ for $\bigcup_{n\in\mathbb{N}} \mathcal{F}^\star_n$.

The first three parts of the next definition are pretty much identical to the definition of primitive recursive functions (with the caveat that now we're closing larger sets of functions). The fourth part is where the magic happens and it's why life is a little easier if functions are allowed to be partial.

Definition 2.5.1. The set of *recursive functions* $\mathcal{R} \subseteq \mathcal{F}^\star$ is defined inductively as follows:

(1) $\mathcal{B} \subseteq \mathcal{R}$.

(2) If $\mathcal{R}$ is closed under composition.

(3) $\mathcal{R}$ is closed under primitive recursion.

(4) Given $f \in \mathcal{F}^\star_{n+1} \cap \mathcal{R}$ the function $g \in \mathcal{F}^\star_n$ given by:

- If there is some $z \in \mathbb{N}$ such that $f(\bar{x}, z) = 0$ and $(\bar{x}, z') \in \mathsf{dom}(f)$ for all $z' \leq z$ then $g(\bar{x})$ is the minimal such z.
- Otherwise $g(\bar{x})$ is undefined.

(5) That's it.

Here's a little bit more terminology:

- We say that a set of functions $\mathcal{S}$ is **closed under minimalisation** if is satisfies (4) in the definition above.
- In (4), we write $g(\bar{x}) = \mu y.(f(\bar{x}, y) = 0)$. Another way of defining this function is the following:

$$\mu y.(f(\bar{x}, y) = 0) := \begin{cases} z & \text{if } f(\bar{x}, z) = 0 \text{ and } f(\bar{x}, z') > 0 \text{ for all } z' < z \\ \text{undefined} & \text{othersiwe.} \end{cases}$$

Remark 2.5.2. Okay, you caught me trying to hide things under the rug. We should be a little more careful when we discuss closure under composition and primitive recursion for partial functions. If you think you can handle the truth, here it is:

(2) Given partial functions $f_1, \ldots, f_n \in \mathcal{F}^\star_m \cap \mathcal{R}$ and $h \in \mathcal{F}^\star_n \cap \mathcal{R}$ the partial function $h(f_1, \ldots, f_n)$ which maps $\bar{x}$ to $h(f_1(\bar{x}), \ldots, f_n(\bar{x}))$ if $\bar{x} \in \mathsf{dom}(f_i)$ for each i and $(f_1(\bar{x}), \ldots, f_n(\bar{x})) \in \mathsf{dom}(h)$ and is otherwise undefined.

(3) Given partial functions $f \in \mathcal{F}^\star_k \cap \mathcal{R}$ and $g \in \mathcal{F}^\star_k \cap \mathcal{R}$ the function h defined by:

- $h(\bar{x}, 0) = f(\bar{x})$, if $\bar{x} \in \mathsf{dom}(f)$ and otherwise h is undefined for $(\bar{x}, 0)$.
- $h(\bar{x}, y{+}1) = g(\bar{x}, y, h(\bar{x}, y))$ if h is defined for $(\bar{x}, y)$ and if $(\bar{x}, y, f(\bar{x}, y)) \in \mathsf{dom}(g)$ and otherwise h is undefined for $(\bar{x}, y + 1)$

Right that enough abstract business, we should connect this with our machines. Ah, there's a small issue, when we defined what it means for a function to be computable by a register machine, we only cared about total functions. Since we're now interested in partial functions, we need to extend our definition.

We'd love to say that a partial function $f : A \subseteq \mathbb{N}^n \to \mathbb{N}$ is **register machine computable** if there is a register machine R such that on $\mathsf{dom}(f) = A$, R computes f, in the sense discussed previously, while on $\mathbb{N}^n \setminus A$, R may or may not halt. The situation could get a little fussy, but let's not be too pedantic. We'll adopt the "obvious" definition:

Definition 2.5.3. Let $f : \mathbb{N}^m \to \mathbb{N}$ be a partial function. We say that f is *register machine computable* if there is a register machine $R = (\mathtt{L}_1, \ldots, \mathtt{L}_n)$ with input registers $R_1, \ldots, R_m$ such that for all $x_1, \ldots, x_m \in \mathbb{N}^m$ we have:

- If $(x_1, \ldots, x_m) \in \mathsf{dom}(f)$ then the run of R with initial state $(x_1, \ldots, x_m, 0, 0, \ldots)$ terminates with final state $(f(x_1, \ldots, x_m), 0, 0, \ldots)$.

- If $(x_1, \ldots, x_m) \notin \mathsf{dom}(f)$ then the run of R with initial state $(x_1, \ldots, x_m, 0, 0, \ldots)$ never terminates.[1].

From the way that we defined composition and recursion for partial functions, the following remark is immediate:

Remark 2.5.4. All the results concerning closure properties of total register machine computable functions still carry through for partial register machine computable functions.

Now that we've allowed ourselves partial functions, we'll prove one more closure property:

Proposition 2.5.5 (Closure under minimalisation)**.** *Let $f : \mathbb{N}^{n+1} \to \mathbb{N}$ be a partial function that is register machine computable. Then, the partial function $g : \mathbb{N}^n \to \mathbb{N}$ given by $g(\bar{x}) = \mu y.(f(\bar{x}, y) = 0)$ is also register machine computable.*

PROOF. To compute $g(\bar{x})$, we start running the register machine that computes f with consecutive inputs for y, that is, we run $f(\bar{x}, 0)$, $f(\bar{x}, 1)$ etc. and we end the computation precisely when we hit some element y which makes $f(\bar{x}, y) = 0$. If the machine never terminates, then $g(\bar{x})$ is undefined (either because at some point we had that $f(\bar{x}, y)$ was undefined or because $f(\bar{x}, y)$ is never 0), and if it does terminate, then, it does so at the smallest value of y. □

We have in fact, pretty much proved the following:

Corollary 2.5.6. *Every partial recursive function is register machine computable.*

PROOF. By Remark 2.5.4, we have closure under composition and primitive recursion. The previous proposition shows the remaining closure property, □

Our next goal is to show the converse.

[1]Some authors allow R to terminate with input not in the domain, provided that then it "lets us know" that the output is rubbish, i.e. there is no $n \in \mathbb{N}$ such that the final state is $(n, 0, 0, \ldots)$

2.6. Recursive functions are like pretty cool, actually. Fix, for the minute a register machine $R = (\mathtt{L}_1, \dots, \mathtt{L}_n)$ which computes a partial function $f \in \mathcal{F}_p^\star$ (i.e. has p input registers).[2] Our goal will be to prove that f is a recursive function. To do so, we will proceed in two steps:

Step 1. We will find an effective (i.e. primitive recursive) way of coding register machine configurations into natural numbers.

Step 2. We will define a primitive recursive function that given a register machine configuration, computes the next configuration.

Step 3. Using our now unbounded μ operator, we simulate the computation.

Well, Step 1 is as good a place to start as any. Recall that a configuration of a register machine is but a pair:

$$(\underline{s}_i, \mathtt{L}_{j_i}),$$

where:

$$\underline{s}_i = (s_1^i, s_2^i, \dots)$$

is a sequence of natural numbers, so that s_1^i is the contents of register R_j at the i-th step of the computation (R of course only uses finitely many registers) and $\mathtt{L}_{j_i}$ is one of the instructions of our register machine. We shall thus devise effective ways of coding these two objects into natural numbers.

Step 1(a). Code register machine instructions into numbers.

First of all, we want a way to code a register machine R (i.e. its instruction set and arity) in a single number. A way of doing this, using Gödel numbers is presented below. This way is neither unique nor optimal.

First, we figure out how to code instructions into single numbers. We can do so in the following way:

- An instruction of the form

  $$\mathtt{L}_i : (R_j, +, \mathtt{L}_k)$$

  is coded by the natural number:

  $$\mathsf{Code}(\mathtt{L}_i) = \mathsf{pair}(0, \mathsf{pair}(j, k)) + 1.$$

[2] This is the point, where, in order to make our coding easier, we will assume that R only ever touches registers R_1 up to R_n – This is a harmless assumption, as any register machine using exactly n registers in its computation can be assumed (after relabelling registers) to be using the first n ones.

- An instruction of the form
$$\mathtt{L}_i : (R_j, -, \mathtt{L}_k, \mathtt{L}_l)$$
is coded by the natural number
$$\mathsf{Code}(\mathtt{L}_i) = \mathsf{pair}(1, \mathsf{tuple}^3(j, k, l)) + 1.$$
- An instruction of the form:
$$\mathtt{L}_i : \textsc{Halt}$$
is coded by the natural number
$$\mathsf{Code}(\mathtt{L}_i) = 0.$$

It is clear from all our work on primitive recursive functions, that all of this can be done and undone primitively recursively.

Step 1(b). We will code $\underline{s}_i$ into a natural number in a similar way, namely we will take:
$$\mathsf{Code}(\underline{s}_i) = \mathsf{pair}(n, \langle s_i^1, s_i^2, \dots \rangle),$$
where n is the number of instructions of R, so it's an upper bound on the number of registers that R will ever touch. Again, both encoding and decoding this is primitive recursive.

Remark 2.6.1. Once again, R only uses finitely many registers (and without loss of generality, we can always assume it only uses registers $R_1, \dots, R_{2n}$, but note that the Gödel number of an infinite sequence that is eventually the constant sequence zero makes sense, so we may well just write the expression above to make life easier.

Finally, given a state $(\underline{s}_i, \mathtt{L}_{j_i})$ we do what the notation would suggest, and set:
$$\mathsf{Code}(\underline{s}_i, \mathtt{L}_{j_i}) := \mathsf{pair}(\mathsf{Code}(\underline{s}_i), \mathsf{Code}(\mathtt{L}_{j_i})).$$
Of course, the final pairing is also done primitively recursively. That was way too many definitions, and honestly, I'll never ask you to write out the code of something, although you could and it would be very instructive. I'll do an example here, just to make sure we're all on the same page.

Recall that:
$$\mathsf{pair}(x, y) = \frac{1}{2}(x + y)(x + y + 1) + y$$
and, more generally:
$$\mathsf{tuple}^2(x, y) = \mathsf{pair}(x, y), \quad \mathsf{tuple}^{n+1}(x_1, \dots, x_{n+1}) = \mathsf{pair}(\mathsf{tuple}^n(x_1, \dots, x_n), x_{n+1})$$

Also, recall that:

$$\langle x_0, \dots, x_{n-1} \rangle := \mathsf{pr}(0)^{x_0} \times \cdots \times \mathsf{pr}(n-2)^{x_{n-2}} \times \mathsf{pr}(n-1)^{x_{n-1}}.$$

Thus, if we go back to an easy register machine from before, say:

- $\mathtt{L}_1$: If $R_2 = 0$ then "Go to $\mathtt{L}_3$" else "Let $R_2 := R_2 - 1$" and go to $\mathtt{L}_2$.
- $\mathtt{L}_2$: Let $R_1 := R_1 + 1$ then "Go to $\mathtt{L}_1$".
- $\mathtt{L}_3$: HALT.

Then, these instructions are coded as follows:

- $\mathtt{L}_1$: $\mathsf{pair}(1, \mathsf{pair}(\mathsf{pair}(2,2),3))) + 1$
- $\mathtt{L}_2$: $\mathsf{pair}(0, \mathsf{pair}(2,1)) + 1$
- $\mathtt{L}_3$: 0

For the initial configuration $(2, 2, 0, 0, \dots)$, we have that:

$$\langle 2, 2, 0, 0, \dots \rangle = 2^2 \times 3^2 = 36.$$

So, here, for example the state $((2, 2, 0, 0, \dots), \mathtt{L}_1)$ would simply be coded as:

$$\mathsf{pair}(\mathsf{pair}(3, 36), 108) = 10476.$$

This is extremely tedious, but the point is that it is a fully formal procedure that we can write a little program to do for us. We will assume for now that when a register machine reaches a configuration with the halting instruction it just stays forever at that configuration (i.e. the next configuration is the same).

Lemma 2.6.2. *Let $R = (\mathtt{L}_1, \dots, \mathtt{L}_n)$ be a fixed register machine. There is a primitive recursive function $g_R \in \mathcal{F}_1$ such that if x is the code of the configuration of R at instant t, then $g_R(x)$ is the code of the configuration of R at instant $t + 1$.*

PROOF (SKETCH). This follows because primitive recursion is closed under definition by cases. Indeed, we define g by based on each instruction. We first decode the configuration, then based on the instruction we alter the state and encode the appropriate instruction number □

We've gotten pretty far coding machines into numbers. Funny, we can go even further. Let's denote by next_R the function $g_R \in \mathcal{F}_1$ from the previous lemma.

Lemma 2.6.3. *Let $R = (L_1, \ldots, L_n)$ be a fixed register machine. Let* $\mathsf{Run\text{-}at}(t, x_1, \ldots, x_p)$ *be defined by recursion, as follows:*

$$\mathsf{Run}_R\text{-}\mathsf{at}(t, x_1, \ldots, x_p) := \begin{cases} \mathsf{pair}(\mathsf{Code}(x_1, \ldots, x_p), \mathsf{Code}(L_1)) & \text{if } i = 0 \\ \mathsf{next}_R(\mathsf{Run}_R\text{-}\mathsf{at}(t-1, x_1, \ldots, x_p)) & \text{otherwise.} \end{cases}$$

Then:

- $\mathsf{Run}_R\text{-}\mathsf{at}$ *is primitive recursive.*
- *For all $t, x_1, \ldots, x_n \in \mathbb{N}$, we have that* $\mathsf{Run}_R\text{-}\mathsf{at}(t, x_1, \ldots, x_n)) = \mathsf{Code}(\underline{s}_t, L_{j_t})$, *where $(\underline{s}_t, L_{j_t})$ is the t-th configuration of R on input $(x_1, \ldots, x_t)$.*

PROOF. At this point, this is trivial, since next is primitive recursive and so are all the functions involved in the computation of $\mathsf{pair}(\mathsf{Code}(x_1, \ldots, x_p), \mathsf{Code}(\mathtt{L}_1))$. □

So, to recapitulate, our goal was to show that given a partial function $f : \mathbb{N}^m \to \mathbb{N}$ for which there is a register machine $R = (\mathtt{L}_1, \ldots, \mathtt{L}_n)$, that computes f, then there is a recursive function $g : \mathbb{N}^m \to \mathbb{N}$ such that $\mathsf{dom}(f) = \mathsf{dom}(g)$ and for all $\bar{x} \in \mathsf{dom}(f)$, $f(\bar{x}) = g(\bar{x})$. We're almost there. Now is the point where we need to actually move one step above primitive recursion, to actual recursion. Recall that we have coded HALT to be 0, and by definition, the codes of all other instructions are non-zero. Then, we can define the following recursive function:

$$\mathsf{term\text{-}time}_R(x_1, \ldots, x_p) := \mu t.(\mathsf{unpair}_2(\mathsf{Run}_R\text{-}\mathsf{at}(t, x_1, \ldots, x_p) = 0)$$

which, as the name suggests gives us the **termination time** of R, i.e. the first instant that R reaches the halting state. Observe that:

$(x_1, \ldots, x_p) \in \mathsf{dom}(\mathsf{term} - \mathsf{time})$ if and only if R terminates on input $(x_1, \ldots, x_p)$.

We're now so close. Let:

$$\mathsf{return}(x) = \mathsf{untuple}_1(\mathsf{unpair}_1(x)),$$

i.e. the function that when given a pair, the first element of which is a sequence, returns the first element of that sequence. Finally, set:

$$g(x_1, \ldots, x_p) = \mathsf{return}(\mathsf{Run}_R\text{-}\mathsf{at}(\mathsf{term\text{-}time}_R(x_1, \ldots, x_p), x_1, \ldots, x_p)).$$

Then, g is recursive. By all the previous discussion, $\mathsf{dom}(f) = \mathsf{dom}(g)$ and for all $(x_1, \ldots, x_p) \in \mathsf{dom}(f)$ we have that $f(x_1, \ldots, x_p) = g(x_1, \ldots, x_p)$. This is so cool it should be in a theorem environment:

THEOREM 2.6.4. *Let $f \in \mathcal{F}^\star$. Then, the following are equivalent:*

(1) f is a partial recursive function.

(2) f is register machine computable.

A similar argument (with the fiddling happening in different ways) let's us prove the following:

FACT. *A partial function is recursive if and only if it is Turing machine computable.*

2.7. Church's Thesis. A function is recursive if and only if we can think up of a mechanised procedure that computes it (in ANY kind of way).

Not that I have been all that serious about writing out computations in a formal way, but from now, having full faith in Church's thesis (we all have to have a Church, I suppose), I will be even less formal. When we want to show that a function is recursive rather than try to write out a formal derivation from the closure operations that define recursive functions we'll just describe an algorithm (at a high level) that computes our functions.

Homework 7

3. Halt and catch fire

Now that we've dealt with a fair bit of technical recursion theory, we're about to start using recursion theory to prove some big (negative) results. The main theorem we will prove here is (essentially) due to Turing, and it roughly states that there is no computer program (i.e. by Church's thesis, no recursive function) that given as input the (literal, say) code of a computer program R and an input x for R decides if R on input x halts or not. This is the so called **halting problem** (of the title of the section).

3.1. A God Machine. In the previous section, we simulated the running of a *fixed* register machine R by a recursive function. Our goal is to show that there is a fixed recursive function (i.e. register machine) which can simulate the running of *any* register machine.

We'll start this part of our journey by describing such a machine, which is usually called a **universal machine**. Really, we're all too familiar with universal machines for this to be something magical, but in a sense it really is. It tells us that we need only keep one register machine in our pockets, and it can do any computational task we may need of it.

Anyway, a universal register machine is a register machine $\phi^p : \mathbb{N}^2 \to \mathbb{N}$ which takes as inputs:

(1) ANY register machine R (encoded in some way) with p input registers.

(2) A code for a set of inputs $(x_1, \ldots, x_p)$ for R

and:

- If R on input $(x_1, \ldots, x_p)$ terminates with output y then ϕ^p terminates with output y.
- If R on input $(x_1, \ldots, x_p)$ does not terminate, then neither does ϕ^p.

Of course, our usual computers pretty much are universal machines, since they can simulate the running of any program. It is somewhat an unfortunate artefact of the early days of recursion theory that a lot of the discussion here will have to be kind of low to the ground.

Step 1. Code register machine instructions into numbers.

First of all, we want a way to code a register machine R (i.e. its instruction set and arity) in a single number. Recall that for each instruction $\mathtt{L}_i$ we devised already a way of coding $\mathtt{L}_i$ into a single number, which we denoted by $\mathsf{Code}(\mathtt{L}_i)$. To code a register

machine R whose instructions are $\mathtt{L}_1, \ldots, \mathtt{L}_n$ we just put those codes together:

$$\langle \mathsf{Code}(\mathtt{L}_1), \ldots, \mathsf{Code}(\mathtt{L}_n) \rangle.$$

Finally, we write

$$\ulcorner R \urcorner := \mathsf{tuple}^3(p, n, \langle \mathsf{Code}(\mathtt{L}_1), \ldots, \mathsf{Code}(\mathtt{L}_n) \rangle).$$

We call $\ulcorner R \urcorner$ the **index** of the register machine R.

We can then decode register machines in a primitive recursive way. Indeed, unpair and i-th coordinates are all primitive recursive. The following proposition is central, but also extremely tedious to prove carefully, I hope that at this point you can trust that when I say that something is extremely tedious, it really must be,

Proposition 3.1.1. *The set:*

$$I_p := \{i \in \mathbb{N} : i \textit{ is the index for a register machine of arity } p\}$$

is primitive recursive.

Proof (Sketch). Of course, $i \in I_p$ if and only if it satisfies a Boolean combination of primitive recursive conditions, e.g. its the code of a pair whose first coordinate is the natural number p, whose second coordinate is a natural number n and whose third coordinate is the Gödel code of a sequence of length n, where each entry corresponds to the code of one of the three kinds of instructions. □

If $i \in I_p$ we will write R^i for the register machine such that $\ulcorner R^i \urcorner = i$.

Step 2. Simulate the running of any register machine.

Recall that a state of a register machine R was a pair $(\underline{s}_i, \mathtt{L}_i)$, where $\underline{s}$ consisted of all the inputs of the registers and $\mathtt{L}_i$ was the "current instruction". We provided a way of coding any such configuration, by setting:

$$\mathsf{Code}(\underline{s}_i, \mathtt{L}_{j_i}) := \mathsf{pair}(\mathsf{Code}(\underline{s}_i), \mathsf{Code}(\mathtt{L}_{j_i})).$$

Now we want an analogue of Lemma 2.6.3, where the machine we are working with is not fixed from the beginning.

We will define a function:

$$\mathsf{simul}^p : \mathbb{N}^{p+2} \to \mathbb{N}$$

as follows:

- If $i \in I_p$ then $\mathsf{simul}^p(i, t, x_1, \ldots, x_p)$ is the precisely $\mathsf{Run}_{R_i}\text{-}\mathsf{at}(t, x_1, \ldots, x_n)) = \mathsf{Code}(\underline{s}_t, \mathtt{L}_{j_t})$, where $(\underline{s}_t, \mathtt{L}_{j_t})$ is the t-th configuration of R_i on input $(x_1, \ldots, x_t)$.

- $\mathsf{simul}^p(i, t, x_1, \ldots, x_t p) = 0$ otherwise.

Then, we have:

THEOREM 3.1.2. *For all $p \in \mathbb{N}$, the function simul^p is primitive recursive.*

PROOF (SKETCH). We need only care about the case $i \in I_p$. This is easier said than done formally, and it's close to a recapitulation of arguments we've already thought about.

If $t = 0$, we define:

$$\mathsf{simul}^p(i, t, x_1, \ldots, x_p) = \mathsf{pair}(i, \mathsf{Code}((x_1, \ldots, x_p), \mathtt{L}_1)),$$

where $\mathtt{L}_1$ is the initial instruction of R^i (and which can be primitively recursively retrieved from i.

Otherwise, we define:

$$\mathsf{simul}^p(i, t, x_1, \ldots, x_p) = \mathsf{pair}(i, \mathsf{next}_{R^i}(\mathsf{unpair}_2(\mathsf{simul}^p(i, t-1, x_1, \ldots, x_p)))),$$

where next_{R^i} is defined as in Lemma 2.6.2. □

Now, let's fix notation for what will end up being our god machine. We will write ϕ^p for the machine which on input $(i, x_1, \ldots, x_p)$ does the following:

- If $i \notin I_p$, then $\phi^p(i, x_1, \ldots, x_p)$ is undefined, i.e. it never halts, to punish us for trying to get it to compute an invalid program.
- If $i \in I_p$, then our god machine runs the computation of the machine with index i on input $(x_1, \ldots, x_p)$. More explicitly:
  - $\phi^p(i, x_1, \ldots, x_p)$ never halts if the machine with index i never halts on input $(x_1, \ldots, x_p)$.
  - $\phi^p(i, x_1, \ldots, x_p)$ terminates with the output of the machine with index i otherwise.

To define our god machine ϕ^p we will play a game similar to the one we played in the previous section. First of all, we can define the following function:

$$\mathsf{term\text{-}time}(i, x_1, \ldots, x_p) := \mu t.(\mathsf{unpair}_2(\mathsf{simul}^p(i, t, x_1, \ldots, x_p) = 0),$$

which is our *universal termination time function*. Let us also define the following set:

$$\mathsf{halted}^p = \{(i, t, x_1, \ldots, x_p) : (\mathsf{unpair}_2(\mathsf{simul}^p(i, t, x_1, \ldots, x_p) = 0\},$$

which contains a tuple $(i, t, x_1, \ldots, x_p)$ if and only if $i \in I_p$ and R^i on input $(x_1, \ldots, x_p)$ has reached the halting state at time t. This is clearly primitive recursive. To make life a bit easier, let's also fix notation for the "fibres" of this set:

$$\mathsf{halted}^p(i) = \{(t, x_1, \ldots, x_p) : (i, t, x_1, \ldots, x_p) \in \mathsf{halted}^p\},$$

We should also figure out what to do with the results of computations, here's a way:

$$\mathsf{computed}^p = \{(i, y, t, x_1, \ldots, x_p) : (i, t, x_1, \ldots, x_p) \in \mathsf{halted}^p \text{ and at instant } t \text{ the register } R_1 \text{ of } R^i \text{ contains } y\},$$

so $\mathsf{computed}^p$ contains a tuple $(i, y, t, x_1, \ldots, x_p)$ if and only if $i \in I_p$, the machine R^i on input $(x_1, \ldots, x_p)$ has halted at time t and has produced output y. We can also fix notation for the fibres of this set, as before:

$$\mathsf{computed}^p(i) = \{(y, x_1, \ldots, x_p) : (i, y, t, x_1, \ldots, x_p) \in \mathsf{computed}^p\},$$

Porting these sets into our μ operator, we have that:

$$\phi^p(i, x_1, \ldots, x_p) = \mu y. \left[(y, \mathsf{term\text{-}time}^p(i, x_1, \ldots, x_p), x_1, \ldots, x_p) \in \mathsf{computed}^p(i)\right].$$

It is obvious that ϕ^p is recursive, and hence there is a register machine which computes ϕ^p. This is a universal machine! To summarise, we have shown the following:

THEOREM 3.1.3 (The enumeration theorem). *For every integer p, the function ϕ^p is a partial recursive function. Moreover, for every partial recursive function $f \in \mathcal{R} \cap \mathcal{F}_p^\star$ there is some $i \in \mathbb{N}$ such that:*

$$f = \lambda x_1, \ldots, x_p. \phi^p(i, x_1, \ldots, x_p).$$

Let us fix some notation, to make using ϕ^p easier. For each integer $i \in \mathbb{N}$ let:

$$\phi_i^p = \lambda x_1 x_2 \ldots x_p. \phi^p(i, x_1, \ldots, x_p).$$

Definition 3.1.4. Let $f \in \mathcal{F}_p^\star$ be a partial recursive function. We say that $i \in \mathbb{N}$ is an *index* of f if $f = \phi_i^p$.

Some pedantic remarks:

(1) If i is the index of a register machine that computes $f \in \mathcal{F}_p^\star$, then i is also the index of f.

(2) Every $i \in \mathbb{N} \setminus I_p$ is an index of the partial function in $\mathcal{F}_p^\star$ whose domain is empty.

(3) It could happen that j is the index of a machine R (with p input registers) but that machine does not compute a partial function in $\mathcal{F}_p^\star$ because when it reaches the halting state it hasn't cleaned up its scratch registers.