Automated Search of Functions and Synthesis of Code

Bruno Miguel Carrajola Patrício
Instituto Superior Técnico, Lisboa, Portugal

December 2019

Abstract

The process of scientific discovery can be explained as a cycle that starts with observing facts that surround us, models those observations into theories, makes predictions from those theories and then confronts them with other observations, reinforcing or disproving those theories. Most of the times, the weak link of this chain of events is the inductive step from concrete observed facts to general theories because it is not always easy for scientists to find these correlations. In this work, we propose a solution to that problem: what if we can automate that step and allow computers to do it? This can be achieved if the empirical laws that scientists try to find are not only computable, but also structurally simple. We explore this statement by relating these laws with the set of the primitive recursive functions (and later on with a subset of it, the elementary functions), allowing us to present relatively simple automatic scientists that would be an early response to this problem and a starting point into a more serious and complete solution for the automation of the inductive inference process.
Keywords: Scientific Discovery, Empirical Laws, Automated Scientists, Primitive Recursive Functions, Code Generation.

1. Introduction

Scientific advances have been achieved over time through a relatively simple and easy to describe process. Scientists begin with analyzing what surrounds them and making measures of those observations, performing the first step of this procedure. They then try to understand and find correlations between these facts, translating them into theories expressed by mathematical expressions. Once they do this, they make predictions deriving from these theories and, if those predictions are verified in reality, the theories come out reinforced and if not they are refuted. This process is pushed to the limit so that a theory can be consistently reinforced or then disproved once and for all. The hardest part of this cyclical process, called the scientific discovery process, is the step where we generalise from particular observations to a theory, since correlations between facts are sometimes hard to find.

In order to overcome this obstacle, scientists started to try to formalize this process by describing it through a clear set of rules; in that case, scientific discovery could be algorithmically structured, allowing us to achieve scientific advances and find theories much faster while obtaining much more complex results by applying those rules. Furthermore, it could even be possible for machines to learn these processes and perform them themselves. However, this will only be possible by supposing that natural laws are computable (as defended by Kelly in [8] and Szudzik in [13] with the computable universe hypothesis) and are also simple enough to be discovered by using these kinds of processes, i.e. these laws would have to be algorithmic themselves and, consequently, the expected behaviour of the world, if not its exact behaviour, would also have to be algorithmic, even if infeasibly computable (see [1]).

By taking into account these assumptions, we propose ourselves to develop a scientist that can identify the mathematical expressions that explain the natural laws that surround us and return those mathematical expressions as computer programs in Python language. This will be made by searching through the set of the primitive recursive functions at first (we do this because we believe that the functions outside the class of the primitive recursive functions are too complex to explain natural laws and also because it is not possible to explain the whole class of recursive functions by a brute-force algorithm while it is for the primitive recursive ones, as seen in [2] and explored further ahead, simplifying the construction of our scientist) and then through one of its subsets, the elementary functions. Furthermore, there is a need to present Learning Theory concepts and results (using notions of computer science, since we will be working on the basis of the computable universe hypothesis), so that we can make the best decisions possible throughout the process of constructing the scientist. We will also study properties of the set of the primitive recursive functions (and then of the elementary functions) that will, in a first phase, allow us to enumerate them in order to perform a search procedure through this set and then to transform such functions into Python programs.

2. Learning Theory

The following concepts and definitions come from Osherson et al. [10], to whom the paper [6] had a

big influence. It is assumed that learning involves the following four concepts:

1. A learner, or scientist;
2. A subject to be learned;
3. An environment, in which the thing to be learned is exhibited to the learner;
4. The conjecture that occurs to the learner about the subject to be learned on the basis of the environment.

A learning paradigm is a specification of these four concepts. One of these paradigms is the identification of recursive functions by a scientist, i.e. the problem of understanding which recursive functions can be identified by which scientists and under which conditions that identification is made. It is through this learning paradigm that we will try to identify the empirical laws through the use of the computationalist hypothesis (see Kelly in [8] and Case in [1]), where we assume that both the empirical laws and scientific methods are recursive relations. This means that it is reasonable to accept that a law written in standard fashion (i.e. as an algebraic expression) and a computer program are interchangeable and so we can conclude that the identification of computable functions is a way to identify those empirical laws. Thus, by understanding how to identify recursive functions we will be also understanding how we can discover empirical laws. That problem will be solved by attacking the computational limits of what is learnable by a scientist and the rigidity of the learning criteria of said scientist within this paradigm.

2.1. Computability

The next computability theory notions can mainly be found in [3], [4] and [12].

Definition 2.1. Partial Recursive Function
A partial recursive function ψ is defined by the input/output pairs $(n, \psi(n))$ such that if P is a program that computes ψ then for all values of n on which P halts it returns $\psi(n)$. The set of the pairs that define ψ is called the graph of ψ. For the values in which P does not halt, we say ψ is undefined. For the set of values n such that there is a pair $(n, \psi(n))$ in the graph of ψ, i.e. P halts, we call the domain or ψ. If e is the code of the program P in question, then ψ will be denoted as ϕ_e and $\{e\}$ will denote P.

Definition 2.2. (Total) Recursive function
A function ψ is recursive if it has as domain the set of natural number, $\mathbb{N}$, i.e. is total; in other words, the graph $(n, \psi(n))$ that defines ψ has an element for each value $n \in \mathbb{N}$. If P is a program that computes ψ then for all values of n, P halts and it returns $\psi(n)$. The set of all recursive functions is denoted by $\mathcal{R}$.

Theorem 2.1. *s-1-1 theorem for binary functions*
For any fixed value $m \in \mathbb{N}$, there is a computable total function g such that $\psi(m, n) = \phi_{g(m)}(n)$. This means that for any arbitrary m, $g(m)$ is the code of $\psi(m, n)$.

Theorem 2.2. *Kleene's Theorem*
For any binary partial recursive function ψ there is a number $e \in \mathbb{N}$ such that $\{e\}(x) = \psi(e, x)$. In other words, $\phi_e(x) = \psi(e, x)$.

2.2. Scientific methods

Definition 2.3. Text for a function
A text T for a function ψ is a total function $T : \mathbb{N} \to \mathbb{N}^2$ such that for every $a, b \in \mathbb{N}$, $(a, b) \in \text{range}(T) \Leftrightarrow \psi(a) = b$.

The set of all the texts T for functions is denoted as $\mathcal{T}$. T_n denotes the pair $T(n)$, while $T[n]$ denotes the sequence of pairs $T_0 \dots T_{n-1}$. A text allows repetitions and is sensible to the order of its pairs, which means that there is an uncountable number of texts for a function ψ. A text is thus a function whose domain is important to give an order to the pairs contained in its range.

Definition 2.4. Text in canonical form
Let T be a text for a function ψ. T is said to be in the canonical form if $T(i) = (i, \psi(i))$ for any $i \in \mathbb{N}$.

Definition 2.5. $SEG = \{T[n] : T \in \mathcal{T}, n \in \mathbb{N}\}$ is called the set of prefixes of recursive functions. $INIT \subset SEG$ is the subset of prefixes of texts in canonical form.

Definition 2.6. Scientist
A scientist for functions $\mathcal{M}$ is on itself a function such that $\mathcal{M} : SEG \to \mathbb{N}$.

Definition 2.7. Convergence of scientist
A scientist for functions $\mathcal{M}$ converges to $i \in \mathbb{N}$ on text T for a function if there exists $p \in \mathbb{N}$ such that for $t > p, \mathcal{M}(T[t]) = i$.

Definition 2.8. Ex-identification of functions
A scientist for functions $\mathcal{M}$ Ex-identifies a function ψ for a text T if it converges on a conjecture $i \in \mathbb{N}$ such that $\phi_i = \psi$ when provided text T. A scientist for functions $\mathcal{M}$ Ex-identifies a function ψ if, for every text T for ψ provided, it converges for conjectures that are code for ψ, i.e. for any T for ψ the conjecture returned $i \in \mathbb{N}$, which can differ depending on T, is one such that $\phi_i = \psi$. A scientist for functions $\mathcal{M}$ Ex-identifies a set of functions Ψ if $\mathcal{M}$ Ex-identifies every function $\psi \in \Psi$. The class of all the sets of recursive functions Ex-identifiable by a scientist is denoted Ex.

Definition 2.9. Total scientist
A scientist $\mathcal{M}$ is total on a recursive function ψ if it provides conjectures for any prefix of any text

regarding ψ given to the scientist. $\mathcal{M}$ is total on a set of functions Ψ if it is total on every function $\psi \in \Psi$. $\mathcal{M}$ is total if it is total on the whole set of recursive functions $\mathcal{R}$.

Proposition 2.1. (see [10]) *For each scientist $\mathcal{M}$ for functions, there exists another scientist $\mathcal{N}$ for functions, algorithmically obtainable from $\mathcal{M}$, such that $\mathcal{N}$ is total and if $\mathcal{M}$ identifies any recursive function ψ then so does $\mathcal{N}$.*

Proposition 2.2. (see [10]) *Let $\mathcal{M}$ be a method that Ex-identifies the recursive function ψ. If $\mathcal{M}$ converges to a conjecture e on the canonical text for ψ, then there exists a scientist $\mathcal{M}'$ that converges to e on all texts for ψ.*

The last two propositions allow us to state the following: the achievements of a scientist that does not need to receive texts in canonical order are the same as the ones who do. The same is valid for total scientists: for every not total scientist there is a total scientist that identifies the same sets of functions. We can then work only regarding total scientists that receive texts in canonical order, which means that from now on, every time we write about a text T for ψ we can simplify notation and write only ψ, since the T is question is canonical and the order of the elements of a text in canonical order coincides with the order of the values of function ψ.

Definition 2.10. n-variant
A partial recursive function ξ is an n-variant of a function $\psi \in \mathcal{R}$ if it coincides with ψ in all but finitely many points never exceeding n. We write $\psi =^n \xi$.

Definition 2.11. $\star$-variant
A partial recursive function ξ is a $\star$-variant of a function $\psi \in \mathcal{R}$ if it coincides with ψ in all but finitely many points. We write $\psi =^\star \xi$.

Definition 2.12. Ex^n-identification
A scientist $\mathcal{M}$ Ex^n-identifies a function $\psi \in \mathcal{R}$ if there exists an order $p \in \mathbb{N}$ such that, for every $t \geq p$, $\mathcal{M}(\psi[t]) = e$ and we have $\phi_e =^n \psi$. A set of functions is said to be Ex^n-identifiable if it exists a scientist that Ex^n-identifies every function in that set. Ex^n is the class of sets Ex^n-identifiable.

Definition 2.13. $Ex^\star$-identification
A scientist $\mathcal{M}$ $Ex^\star$-identifies a function $\psi \in \mathcal{R}$ if there exists an order $p \in \mathbb{N}$ such that, for every $t \geq p$, $\mathcal{M}(\psi[t]) = e$ and we have $\phi_e =^\star \psi$. A set of functions is said to be $Ex^\star$-identifiable if it exists a scientist that $Ex^\star$-identifies every function in that set. $Ex^\star$ is the class of sets $Ex^\star$-identifiable.

Definition 2.14. Bc-identification
A scientist for functions $\mathcal{M}$ Bc-identifies a recursive function $\psi \in \mathcal{R}$ if there exists an order $p \in \mathbb{N}$ such that, for any $t \geq p$ we have that $\phi_{\mathcal{M}(\psi[t])} = \psi$. A scientist for functions $\mathcal{M}$ Bc-identifies a set of functions Ψ if $\mathcal{M}$ Bc-identifies every function $\psi \in \Psi$. The class of all the sets of functions that are Bc-identifiable is denoted by Bc.

Definition 2.15. Bc^n-identification
A scientist $\mathcal{M}$ Bc^n-identifies a recursive function $\psi \in \mathcal{R}$ if there exists an order $p \in \mathbb{N}$ such that, for any $t \geq p$, we have that $\phi_{\mathcal{M}(\psi[t])} =^n \psi$, i.e. from a certain order the scientist outputs a code for an n-variant of ψ, but not necessarily the same one. The class of all the sets of functions that are Bc^n-identifiable is denoted by Bc^n.

Definition 2.16. $Bc^\star$-identification
A scientist $\mathcal{M}$ $Bc^\star$-identifies a recursive function $\psi \in \mathcal{R}$ if there exists an order $p \in \mathbb{N}$ such that, for any $t \geq p$ we have that $\phi_{\mathcal{M}(\psi[t])} =^\star \psi$, i.e. from a certain order the scientist outputs a code for a $\star$-variant of ψ, but not necessarily the same one. The class of all the sets of functions that are $Bc^\star$-identifiable is denoted by $Bc^\star$.

Proposition 2.3. $Ex \subset Ex^1 \subset Ex^2 \subset \cdots \subset Ex^n \subset Ex^{n+1} \subset \cdots \subset Ex^\star \subset Bc \subset Bc^1 \subset Bc^2 \subset \cdots \subset Bc^n \subset Bc^{n+1} \subset \cdots \subset Bc^\star$.

Proposition 2.4. $\mathcal{R} \in (Bc^\star \setminus \bigcup_{n\in\mathbb{N}} Bc^n)$.

Remark: to observe the sequence of proofs and results that lead to the statements in Propositions 2.3 and 2.4 see [2].

3. The Search Procedure

To proceed to the search procedure, we make another assumption: the empirical laws can be explained among the primitive recursive functions. We believe this because the recursive functions that are not primitive recursive (e.g. the Ackermann function in [5]) are very complex and have a behaviour not expected to be present in nature. We will thus focus our attention to this class of functions.

Definition 3.1. Primitive Recursive Functions
The primitive recursive functions are those inductively defined by the following rules:

1. The $0-$ary constant function 0 is primitive recursive.

2. The 1-ary successor function S, defined by the expression $S(x) = x + 1$, is primitive recursive.

3. For any $n \in \mathbb{N}$ and for $i \in \mathbb{N}$ such that $1 \leq i \leq n$, we have that the function $P_{n,i}$ defined by the expression $P_{n,i}(x_1, \ldots, x_n) = x_i$ is primitive recursive.

4. Given a $k-$ary primitive recursive function f and k many $m-$ary primitive recursive functions $g_1, \ldots, g_k$, the function h resulting from the composition of these functions, defined by the expression $h(\mathbf{x}) = f(g_1(\mathbf{x}), \ldots, g_k(\mathbf{x}))$, is primitive recursive, where $\mathbf{x} = (x_1, \ldots, x_k)$.

5. For a $k-$ary primitive recursive function f and a $k+2-$ary primitive recursive function g, the $k+1-$ary function h defined as
$$h(\mathbf{x}, y) = \begin{cases} f(\mathbf{x}), & y = 0 \\ g(\mathbf{x}, y-1, h(\mathbf{x}, y-1)), & \text{otherwise} \end{cases}$$
is primitive recursive, where $\mathbf{x} = (x_1, \ldots, x_k)$.

The set of the primitive recursive functions is denoted by $\mathcal{PRIM}$.

For us to search amongst the primitive recursive functions, we are going to need notation to identify these functions. We thus introduce the notion of description for primitive recursive functions.

Definition 3.2. Description for Primitive Recursive Functions
A description of a primitive recursive function is an expression inductively defined by the following rules:

1. The symbol `Z()` is a 0-ary description that describes the constant **0**.

2. The symbol `S()` is a 1-ary description that describes the successor, i.e., the 1-ary function with the expression $S(x) = x + 1$.

3. The symbol `P(n,i)`, for any n and i such that $1 \leq i \leq n$ is an n-ary description that describes the i-th projection, i.e., the n-ary function defined by the expression $P_{n,i}(x_1, \ldots, x_n) = x_i$.

4. If `G` is a k-ary description, with $k \geq 0$, that describes the function g and if `H_1`, ..., `H_k` are n-ary descriptions that describe the functions $h_1, \ldots, h_k$ respectively, with $n \geq 0$, then `C(G,[H_1, ..., H_k])` is an n-ary description that describes the n-ary function f defined by the expression $f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_n(x_1, \ldots, x_n))$. We say that f is obtained from g and $h_1, \ldots, h_k$ by composition.

5. If `G` is an n-ary description with $n \geq 0$ that describes the function g and `H` is an $(n+2)$-ary description that describes the function h then `R(G,H)` is an $(n+1)$-ary description that describes the $(n+1)$-ary function f recursively defined by the expressions $f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$ and $f(x_1, \ldots, x_n, y+1) = h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y))$. We say that f is obtained from g and h by primitive recursion.

Proposition 3.1. (see [11]) $\mathcal{PRIM}$ is recursively enumerable.

Proposition 3.2. $\mathcal{PRIM} \in Ex$.

Proof. To prove that the primitive recursive functions are Ex-identifiable we simply present a scientist that Ex-identifies this set of functions (Algorithm 1). Let $\rho : \mathbb{N} \to \mathcal{PRIM}$ be an enumeration for the primitive recursive functions, ρ_i denote the function obtained from $\rho(i)$ and ψ the primitive recursive function explained by a text in the canonical form with prefix σ. This algorithm proceeds in the following way: it searches for the least $i \in \mathbb{N}$ such that the output of ρ_i applied to every $k < n$ has the same value as the output of ψ applied to k (which is obtainable in σ). Since ρ is an enumeration for the set $\mathcal{PRIM}$, then this algorithm will not overlook any primitive recursive function and since σ is a prefix of a text that explains a primitive recursive function, this algorithm will eventually halt. □

Algorithm 1 Scientist that Ex-identifies $\mathcal{PRIM}$

```
function M(σ : SEG): ℕ
    i ← 0;
    k ← 0;
    n ← length of σ
    while k < n do
        if ρ_i(k) = ψ(k) then k ← k + 1
        else
            k ← 0;
            i ← i + 1;
        end if
    end while
    return i
end function
```

This means that the implementation of the search algorithm depends on implementing an enumeration for the primitive recursive functions.

3.1. Enumerations

To define an enumeration for the primitive recursive functions, we resort to the notation introduced in Definition 3.2, used to define primitive recursive functions through descriptions. The enumeration will be performed recursively and will be based upon the size of those descriptions. We define a function that receives a number and outputs the list descriptions with the size given as input in the following way:

- If the size is 0, it yields an empty list.
- If the size is 1, it yields the descriptions `Z()`, `S()` and all the ones for the projections with arity up to 3.[1]

[1] Since the arity of the projection symbol does not interfere with its size, we needed to establish an upper bound for this parameter in order to be certain that the computation

- For other values for the size, we use the rules 4 and 5 of Definition 3.2, to construct the descriptions with the intended size: it will yield the descriptions `R(F,G)` such that `F` and `G` are descriptions whose summed sizes are equal to the input value decremented by one and such that the arity of `G` is the arity of `F` plus two; it will also yield the descriptions `C(G,[H_1,...,H_k])` such that `G` is k-ary and the sum of the sizes of descriptions `G`, `H_1`, ..., `H_k` is the input value decremented by one.

We then thought about ways to ameliorate this enumeration, in order to make it more efficient. One way to do it was to take into account the arity of the functions by providing it as input to the enumeration algorithm. This is possible since the objective of our work is to try to identify a primitive recursive function through input and correspondent output data, and thus we have a way of knowing the arity of the function *a priori*. Another way was to prevent repetitions of descriptions on the lists `[H_1,...,H_k]` of the descriptions with main symbol `C`. By doing this, we will still have exhaustiveness when it comes to listing descriptions for every primitive recursive function. We will then define a second enumeration function, that given as input the size and the arity of the descriptions to be listed will return a list of descriptions constructed in the following way:

- If the size is 0, it yields an empty list.
- If the size is 1 and the arity is 0, it yields `Z()`.
- If the size is 1 and the arity is 1, it yields `S()`.
- If the size is 1, for any arity ar, it yields the descriptions `P(`ar`,i)`, for $1 \leq \mathtt{i} \leq ar$.
- For other values for the size, we use the rules 4 and 5 of Definition 3.2 to construct the descriptions with the intended size and arity: it will yield the descriptions `R(F,G)` such that `F` and `G` are descriptions whose summed sizes are equal to the input size value decremented by one and such that the arity of `F` is the arity provided minus one and the arity of `G` is the value given plus one; it will also yield the descriptions `C(G,[H_1,...,H_k])` such that `G` is k-ary, each `H_1,...,H_k` has as arity the provided value, the sum of the sizes of descriptions `G`, `H_1`, ..., `H_k` is the input value for the size decremented by one and such that it does not happen the case where any two of the descriptions `H_1,...,H_k` are the same.

of the list of descriptions halts. It was decided that an adequate bound would be 3, since it allows to define the most simple unary and binary functions, which will be the scope for the tests of our scientists, at least at first.

3.2. From description to code

Our last implementation step remains on defining a way to write the code of a program for a certain function. Before doing so, there is a result we need to present, regarding the development of programs. A program is a sequence of instructions that can be simple assignments (for example $x := 0, x := y, x := y + 1$), conditionals (`if` {guard} `then` ... `else` ...), `for`-loops (`for` $i = 1, \ldots, y$ `do` ... where i never resets) or `while` cycles (`while` {guard} `do` ...). We call a program *loop-program* if it can be built using a sequence of assignments, conditionals and `for`-loops. With these in mind, we present the following statement:

Theorem 3.1. (see [9] and [7]) *The primitive recursive functions are exactly those computed by loop-programs, i.e. the programs that can be written without* `while` *cycles.*

This means that for every primitive recursive function it is possible to write a program only with a sequence of assignments, if-clauses and sequential and nested `for`-loops. Thus, our next step is to obtain said program in Python language for the found description. We developed a program that will do such thing. This program will use an auxiliary list of variables so that it is possible for us to perform the attributions needed for the program to function correctly. Plus, by construction, these varibles will always be of one of two types: tuples or integers. This will be important to have in mind later when we define the way of making the attributions and the operations regarding the variables. Getting back to the program, it begins by writing in a text file the commands that define a function in Python: "`def function(x):`" and then in the next line with the correct indentation we perform the attribution of the input of the function to the first element of the list of variables: "`a0 = x`". This list of variables will be updated throughout the execution of this program so that the correct variable is used every time. Plus, the changes of line and the indentations will also be employed in agreement with the correct ones. Then, the program's execution depends on what symbol of the description it reads:

- if the symbol is `Z()`, then it will attribute to the correct variable the value 0.
- if the symbol is `S()`, then it will be attributed to a new variable the value of the successor of the adequate variable. However, before doing this operation we need to make sure that the variable is an integer and not a tuple; if it is a tuple then it is one of only one element (due to the arities of the operations in question) and then we say that the new variable is the successor of the first element of the tuple.

- if the symbol is `P(n,i)`, then the value of a new variable will be the $i-$th element of the appropriate variable. Reversely to what happens in the previous case, we need to make sure that the variable in question is a tuple; if it is not, then it is an integer and so, before performing the projection, we need to transform the integer variable into a tuple variable and only then realize the projection.

- if the symbol is `C(G,[H_1, ..., H_k])` then the program will write the correct code for computing the output of the functions qith descriptions `H_1,...,H_k`, attribute those values to the correct variables, create a tuple variable composed of the different outputs of the previous k functions and then attribute to the correct value the one computed by applying function with description `G` to that tuple of values.

- if the symbol is `R(G,H)`, then this program will first make sure that the input variable is a tuple, then attributes to a new variable the tuple resulting of deleting the last element of the input tuple. It will then write the code corresponding to computing function with description `G`. Finally, it will start a `for`-loop that will be executed the same number of times as the value of the last element of the input tuple and then it will write the code of function with description `H` with the correct input: the variable resulting of deleting the last element of the input tuple with the number of the iteration of the `for`-loop and another variable appended (this variable starts by being the output of function with description `G` and then it will be updated as the result of executing this loop one time).

The last two possibilities are, of course, recursive, in the sense that when writing the lines of code for the functions that are arguments of the symbols in question it will call the main function. In the end, this program will write a line of code that allows the correct variable to be returned.

4. A restriction to $\mathscr{E}$

We will now explore another way of attacking the problem of the search of expressions that describe natural laws. We have been focusing on the primitive recursive functions to the learning environment of our scientist. However it may not be necessary to have an environment of this type since we suspect that every natural law can be express in a much simpler way: as an elementary function.

Definition 4.1. (see [5]) Elementary Functions
The set $\mathscr{E}$ of elementary functions is the smallest class such that:

1. the functions $x+1$, $P_{n,i}$ $(1 \leq i \leq n)$, $x \dot{-} y$ [2], $x+y$ and xy are in $\mathscr{E}$;
2. $\mathscr{E}$ is closed under composition;
3. $\mathscr{E}$ is closed under the operations of forming bounded sums and bounded products (i.e. if $f(\mathbf{x}, y)$ is in $\mathscr{E}$ then so are the functions $\sum_{z<y} f(\mathbf{x}, z)$ and $\prod_{x<y} f(\mathbf{x}, z)$).

We present some elementary functions and their definitions following the rules of Definition 4.1:

- $exp(x,y) = x^y = \prod_{z<y} x$
- $pred(x) = x \dot{-} 1 = x \dot{-} (x + 1 \dot{-} x)$
- $sg(x) = \begin{cases} 0, & x = 0 \\ 1, & x \neq 0 \end{cases} = x \dot{-} (x \dot{-} 1)$
- $\bar{sg}(x) = 1 \dot{-} sg(x)$
- $dist(x,y) = |x-y| = (x \dot{-} y) + (y \dot{-} x)$
- $fact(x) = x! = \prod_{z<x}(z+1)$
- $min(x,y) = x \dot{-} (x \dot{-} y)$
- $max(x,y) = (x \dot{-} y) + y$
- $qt(x,y) = \lfloor \frac{y}{x} \rfloor = \sum_{v<y+1} \prod_{u<v+1} sg(x \times \bar{sg}(x(z+1) \dot{-} y))$

We will proceed the same way we did regarding the set of the primitive recursive functions, starting with the introduction of a form of notation for the elementary functions by defining descriptions for this set of functions. However, this time we will not just define symbols for the operations in the definition of this set but we will also add other operations that we think useful to a more efficient definition.

Definition 4.2. Description for elementary functions
A description of an elementary function is an expression that is inductively defined as follows:

1. The symbol `EZ()` is a 0-ary description that describes the constant **0**.

2. The symbol `ES()` is a unary description that describes the successor, i.e., the 1-ary function with the expression $S(x) = x+1$.

3. The symbol `EP(n,i)`, for any n and i such that $1 \leq i \leq n$ is an n-ary description that describes the projection, i.e., the n-ary function defined by the expression $P_{n,i}(x_1, \ldots, x_n) = x_i$.

4. The symbol `EA()` is a 2-ary description that describes the addition operation, that is the function $add(x,y) = x+y$.

5. The symbol `EM()` is a 2-ary description that describes the subtraction operation for the natural numbers, that is the function $natminus(x,y) = x \dot{-} y = \max\{x-y, 0\}$.

[2] The operator $\dot{-}$ corresponds to the subtraction for the natural numbers, i.e. we have that $x \dot{-} y = \max\{x-y, 0\}$

6. The symbol `ET()` is a 2-ary description that describes the product operation, that is the function $prod(x, y) = xy$.

7. The symbol `ED()` is a 2-ary description that describes the integer division operation, that is the function $qt(x, y) = \begin{cases} \lfloor \frac{y}{x} \rfloor, & x \neq 0 \\ 0, & x = 0 \end{cases}$.

8. If `G` is a k-ary description, with $k > 0$, that describes the function g and if `H_1`, ..., `H_k` are n-ary descriptions that describe the functions $h_1, \ldots, h_k$ respectively, with $n \geq 0$, then `EC(G,[H_1, ..., H_k])` is an n-ary description that describes the n-ary function f defined by the expression $f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_n(x_1, \ldots, x_n))$. We say that f is obtained from g and $h_1, \ldots, h_k$ by composition.

9. If `G` is an n-ary description, with $k > 0$, that describes a function g then `EBS(G)` is an n-ary description that describes functions f defined by the expression $f(x_1, \ldots, x_n) = \sum_{z<x_n} g(x_1, \ldots, x_{n-1}, z)$. We say that f is a bounded sum obtained from g.

10. If `G` is an n-ary description, with $k > 0$, that describes a function g then `EBP(G)` is an n-ary description that describes functions f defined by the expression $f(x_1, \ldots, x_n) = \prod_{z<x_n} g(x_1, \ldots, x_{n-1}, z)$. We say that f is a bounded product obtained from g.

Proposition 4.1. (see [11]) $\mathscr{E}$ is recursively enumerable.

Proposition 4.2. $\mathscr{E} \in Ex$.

Proof. To perform this proof we only need to present a scientist that Ex-identifies this set of functions (Algorithm 2, adapted from Algorithm 1). Let $\pi : \mathbb{N} \to \mathscr{E}$ be an enumeration for the primitive recursive functions, π_i denote the function obtained from $\pi(i)$ and ψ the elementary function explained by a text in the canonical form with prefix σ. This algorithm searches for the least $i \in \mathbb{N}$ such that the output of π_i applied to every $k < n$ has the same value obtainable from σ for $\psi(k)$. Since π is an enumeration for the set $\mathscr{E}$, then this algorithm will be exhaustive in the set of elementary functions and since σ is a prefix of a text that explains an elementary function, this algorithm will eventually halt. □

4.1. Enumeration

We will now resort to the descriptions for elementary functions to define an enumeration for this set of functions. We will take into account the size and the arity of the descriptions to develop a listing function for these functions:

Algorithm 2 Scientist that Ex-identifies $\mathscr{E}$

```
function M(σ : SEG): ℕ
    i ← 0;
    k ← 0;
    n ← length of σ
    while k < n do
        if π_i(k) = ψ(k) then k ← k + 1
        else
            k ← 0;
            i ← i + 1;
        end if
    end while
    return i
end function
```

- If the size is 0, it yields an empty list.
- If the size is 1 and the arity is 0, it yields `EZ()`.
- If the size is 1 and the arity is 1, it yields `ES()`.
- If the size is 1 and the arity is 2, it yields the descriptions `EA()`, `EM()`, `ET()` and `ED()`.
- If the size is 1, for any arity ar, it yields the descriptions `EP(`ar`,i)`, for $1 \leq$ `i` $\leq ar$.
- For other values for the size, we use the rules 8, 9 and 10 of Definition 4.2 to construct the descriptions with the intended size and arity: it will yield the descriptions `EC(G,[H_1,...,H_k])` such that `G` is k-ary, each `H_1,...,H_k` has as arity the provided value and the sum of the sizes of descriptions `G`, `H_1`, ..., `H_k` is the input value for the size decremented by one. It also yeilds the descriptions `EBS(F)` and `EBP(G)` where `F` and `G` are descriptions with the provided arity and size one unit less than the given one.

4.2. From description to code

Our last step will once again implement a procedure to write the code of a program that computes a function through its description. Since the set $\mathscr{E}$ is contained in the set $\mathcal{PRIM}$ (in [5]) and we know that for all the primitive recursive functions there exists a program written only with a sequence of assignments, if-clauses and sequential and nested for-loops (Theorem 3.1 and in [9]), then it will also be possible to obtain a program with these characteristics for the elementary functions (once again, these programs will be written in Python language), and so we will develop a program that performs this operation. Just like with the primitive recursive functions, we use an auxiliary list of variables in order to make possible the attributions needed for the program to function correctly. These variables will once again always be tuples or integers. Like before,

we will write in a text file, beginning by writing in the first two lines "`def function(x):`" and "`a0 = x`", with the correct indentation. Throughout the program, the auxiliary list of variables will be updated, to allow the use of the correct variable every time, just like the changes of line and the indentations. We know arrive to the differences between the two programs: the execution depending on the symbol it reads. The rules will be the following:

- if the symbol is `EZ()`, then it will attribute to the correct variable the value 0.
- if the symbol is `ES()`, then it will be attributed to a new variable the value of the successor of the adequate variable after making sure that the variable is an integer and not a tuple; if it is not an integer then it is a tuple of only one element (due to the arities of the operations in question) and then we say that the new variable is the successor of the first element of the tuple.
- if the symbol is `EP(n,i)`, then the value of a new variable will be the $i-$th element of the appropriate variable after making sure that that variable is a tuple; if it is not, then it is an integer and so, before performing the projection, we need to transform the integer variable into a tuple variable and only then perform the projection.
- if the symbol is `EA()`, then the value of the new variable will be the addition of the two elements of the pair that compose the previous variable. In this case we don't have to worry about the type of the variable since the description will have arity 2 and so the variable in question will obligatory be a tuple (a pair, to be precise).
- if the symbol is `ET()`, then the value of the new variable will be the product of the two elements of the pair that compose the previous variable. Once again and for the same reasons as for `EA()` we don't have to worry about the type of the variable.
- if the symbol is `EM()`, then the value of the new variable will be the subtraction of the two elements of the pair that compose the previous variable. Once again, there is no need to worry about the type of the variable.
- if the symbol is `ED()`, then the value of the new variable will be the quotient of the second element of the pair over the first, if the first is not 0; if it is, then it will attribute to the new variable the value 0 (this is done in order for it to be a total function).
- if the symbol is `EC(G,[H_1, ..., H_k])` then the program will write the correct code for computing the output of the `h_1,...,h_k` functions, attribute those values to the correct variables, create a tuple variable composed of the different outputs of the previous k functions and then attribute to the correct value the one computed by applying function `G` to that tuple of values.
- if the symbol is `EBS(G)` the program will first make sure that the input variable is a tuple (if it is not, it will turn it into one), attributes to a new variable a tuple with the values of the input variable except the last one and then it will give to a new variable the value 0 that will be updated throughout the `for`-loop that will be written after (meaning that when the last element of the input pair is 0 and the `for`-loop does not execute, the returned value will be 0). This `for`-loop will be executed the same number of times as the value of the last element of the input variable and it will execute the following commands: for the i-the execution, a new tuple variable will be created that will have as values the same as the input variable but with the last one swapped by i; then it will execute the code of respective to description `G` and update the variable declared before the `for`-loop began (that started with the value 0 assigned) by adding to its current value the result of the application of function defined by `G`; in the end, it returns this updated variable's value.
- if the symbol is `EBP(G)` the program will act very similarly to when the symbol is `EBS(G)`: the only changes are that the variable that has initially value 0 in the previous case is initialized with the value 1 (which will make the result 1 when the `for`-loop is not performed, i.e. when the last element of the input pair is 0) and that this value is updated by multiplying its value with the value obtained after the application of the code relative to description `G` instead of being added.

In the end, this program will write a line of code that allows the correct variable to be returned.

5. Results

To test our search algorithms, it was used a computer with operating system Windows 10 and processor Intel(R) Core(TM) i5-4210U CPU @ 1.70 GHz 2.40 GHz, with the Python version 3.7.0 installed. Before analyzing the results obtained, we need to present one definition that will aid us to understand and explain them: the notion of locking sequence.

Definition 5.1. (see [10] and [4]) Locking Sequence Let $\psi \in \mathcal{R}$, $\mathcal{M}$ a scientist and $\sigma \in SEG$. We say that σ is a locking sequence for $\mathcal{M}$ on ψ if (a) $content(\sigma) \subset \psi$, (b) $\phi_{\mathcal{M}(\sigma)} = \psi$ and (c) for all $\tau \in SEG$ such that if $content(\tau) \subset \psi$ then $\mathcal{M}(\sigma \diamond \tau) = \mathcal{M}(\sigma)$.

This concept is important because from the moment a scientist founds a locking sequence σ, it converges immediately in all texts for ψ having σ as prefix, and thus if a small locking sequence is found, then the scientist will converge very fast and will do so for many different inputs.

Generally speaking, we obtained very satisfying results, founding in the grand majority of times small locking sequences in very fast computations. Moreover, the time efficiency of the algorithms improved with the upgrades we performed, be them the changes made in the listing of the primitive recursive functions or the change in paradigm from searching among the primitive recursive functions to restricting that search to the set of elementary functions. Furthermore, these modifications also allowed each scientist to find a greater number of more complex functions than the previous ones. However, this came with some setbacks since sometimes the search could not go forward. This happened because some expressions describe hard to compute functions, making the search algorithm stuck in the computation of the application of those functions to some input values, not allowing the scientist to compare the actual result with the expected one, which can be explained because the concerned functions are computed with nested `for`-loops that had to be executed a tremendous number of times, for example.

We also understood that the size of the locking sequences found depends not only on the concerned function but on the values given as input: if the function that explained the relation between the input and output values could be described by an expression present in the early stages of the descriptions' listing, then a very small locking sequence was obtained; if a function could only be described with an expression that appears at a more posterior position in the listing of descriptions, a small locking sequence could only be achieved if the values provided were not also explained by a description that appeared before in the enumeration but also if these values were not big enough to cause the scientist to be stuck at the verification of some hard to compute descriptions. These values sometimes were not that big: for some functions the pairs $(1, 3)$ and $(2, 3)$ as input was enough. Another pivotal point to understand the efficiency of the scientists is to observe that the order of the input elements also had influence on the outcome of the search; if the tuples that are big enough to cause the scientist to get stuck in the verification of a description are preceded by tuples that are easy to compute and whose result is different than the respective ones in the out list, then the scientist will overcome the verification of that description and move on through the search. Trying to force this input tuples to be provided first can be done only with trial and error, since predicting the outcome of a situation like this is practically speculation and close to impossible due to the fact that it heavily depends on the function we are trying to find. On the other hand, the number of values provided is not that important to the efficiency of the scientists as one could assume at the beginning, at least compared to the other factors we commented previously. More values provided did in fact slow down the search but mostly because the values provided were big enough to slow down the computation of the verification of some descriptions. We see that when the provided values' outputs are easily computed by the several possible descriptions listed or if the initial tuples fail the verification right ahead, preventing the same computation and verification of the following values, the increase in the time of computation, although existing, is residual.

This makes the efficiency of each scientist mainly depending on three factors: the position of the adequate description on the enumeration lists, the magnitude of the values provided and the order in which these elements are given, whilst the size of the lists (i.e. the number of points given) proved to be less relevant to this question: as long as the values are small enough to allow the computation to proceed, the time needed to find the description is not going to be significantly bigger than for smaller but proper lists.

Regarding the code of the generated programs, we observed that sometimes they were constructed with the use of several nested `for`-loops, sometimes a considerable number of them. This does not mean that this is the simplest existing program that computes a function, it just means that it is the program constructed using the first description found that explains the function in question. It can be the case of bigger descriptions that explain the same function resulting in programs with less nested `for`-loops in its sequence of instructions.

6. Conclusions

The major achievement of this work was the computational development of scientists that were able to identify simple primitive recursive functions and/or elementary functions in a short amount of time. These scientists not only identified these functions but they did so with very little information provided, resulting in the discovery of very small locking sequences. A possible explanation for this phe-

nomenon is that the primitive recursive functions have an underlying structure that, even for a small set of input tuples, cause the respective outputs for different primitive recursive functions to be very distinct among themselves, which makes it easier to find the correct functions.

Regarding the relation between these results and the search for empirical laws, there are some assumptions we do before drawing any conclusions. First, we assume that the observations measured are natural values, which is not true. However, this allows us to focus on the identification of these laws through their form, i.e. their algebraic expressions, translated into descriptions of recursive functions of the type $\mathbb{N} \to \mathbb{N}$. This comes with a cost, since it implies ignoring the existence of experimental errors. We also assume that the laws are explained with expressions that have a certain underlying homogeneity, i.e. they are mainly explained, for example, by continuous not piecewise functions. It is this belief in the nature of the empirical laws that allow us to assume that we are able to explain every natural law not only with primitive recursive functions but even more narrowly only by functions in the set of elementary functions. If this assumption is true, then the developed scientists can be considered to be "embryos" of scientists that are actually able to identify relations of natural phenomena on their one, facilitating the scientific discovery process that many times does not evolve because those relations are not found. In order for our "embryo" scientists to evolve to ones that can actually discover expressions that explain the empirical laws, there are some improvements that can be performed, like the ones that follow:

- Improve the interface of the scientist, turning it more complex and user friendly.
- Execute these experiences on a computer with a greater processing capacity.
- Reduce even more the redundancies in the enumeration of the descriptions.
- Define more functions to be in the basis of the rules that inductively construct the descriptions (like it was done with the natural quotient function for the definition of descriptions for elementary functions, defined with the symbol `ED()`). A great improvement in this matter would be the addition of constants to this basis; this way, we could write natural numbers with size one descriptions instead of needing to write those constants with nested compositions of the successor operation applied to the zero constant, thus having expressions with smaller size that would describe functions with natural numbers in their expressions.
- Change the verification step to take into account the existence of experimental errors.

Lastly, we would like to point out that in order to fully understand the achievements and the conclusions obtained with this work, the reading of [11] is absolutely essential and critical.

References

[1] J. Case. Algorithmic scientific inference. *IJUC*, 8(3), 2012.

[2] J. F. Costa. Unity of science as seen through the universal computer. *IJUC*, 13(1):59–81, 2017.

[3] J. F. Costa. On Discovering Scientific Laws. *IJUC*, 14(3–4):285–318, 2019.

[4] J. F. Costa and P. Gouveia. Computabilidade, Inferência Indutiva, Complexidade. Draft of a book to be submitted.

[5] N. Cutland. *Computability: An introduction to recursive function theory*. Cambridge university press, 1980.

[6] E. M. Gold. Language identification in the limit. *Information and control*, 10(5):447–474, 1967.

[7] W. G. Handley and S. S. Wainer. Complexity of primitive recursion. In U. Berger and H. Schwichtenberg, editors, *Computational Logic*, pages 273–300, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[8] K. T. Kelly. *The Logic of Reliable Inquiry*. OUP USA, 1996.

[9] A. R. Meyer and D. M. Ritchie. The complexity of loop programs. In *Proceedings of the 1967 22nd national conference*, pages 465–469. ACM, 1967.

[10] D. N. Osherson, M. Stob, and S. Weinstein. *Systems That Learn: An Introduction to Learning Theory for Cognitive and Computer Scientists*. The MIT Press, 2nd edition, 1986.

[11] B. Patrício. Automated search of functions and synthesis of code. Master's thesis, Instituto Superior Técnico, 2019.

[12] A. Sernadas, M. C. S. Sernadas, and J. Ramos. *Computability and Complexity: A Mathematical Primer*. College Publications, 2018.

[13] M. P. Szudzik. The computable universe hypothesis. In *A Computable Universe: Understanding and Exploring Nature as Computation*, pages 479–523. World Scientific, 2013.