Towards a Definition of an Algorithm

Noson S. Yanofsky

December 23, 2005

Abstract

We define an algorithm to be the set of programs that implement or express that algorithm. The set of all programs is partitioned into equivalence classes. Two programs are equivalent if they are "essentially" the same program. The set of all equivalence classes is the category of all algorithms. In order to explore these ideas, the set of primitive recursive functions is considered. Each primitive recursive function can be described by many labeled binary trees that show how the function is built up. Each tree is like a program that shows how to compute a function. We give relations that say when two such trees are "essentially" the same. An equivalence class of such trees will be called an algorithm. Universal properties of the category of all algorithms are given.

1 Introduction

In their excellent text *Introduction to Algorithms, Second Edition* [5], Corman, Leiserson, Rivest, and Stein begin Section 1.1 with a definition of an algorithm:

> Informally, an **algorithm** is any well-defined computational procedure that takes some value, or set of values, as **input** and produces some value, or set of values, as **output**.

Three questions spring forward:

1. "Informally"? Can such a comprehensive and highly technical book of 1180 pages not have a "formal" definition of an algorithm?

2. What is meant by "well-defined?"

3. The term "procedure" is as vague as the term "algorithm." What is a "procedure?"

Knuth [7, 8] has been a little more precise in specifying the requirements demanded for an algorithm. But he writes "Of course if I am pinned down and asked to explain more precisely what I mean by these remarks, I am forced to admit that I don't know any way to define any particular algorithm except in a programming language." ([8], page 1.)

Although algorithms are hard to define, they are nevertheless real mathematical objects. We name and talk about algorithms with phrases like "Mergesort runs in $n \lg n$ time". We quantify over all algorithms, e.g., "There does not exist an algorithm to solve the halting problem." They are as "real" as the number e or the set $\mathbb{Z}$. See [6] for an excellent philosophical overview of the subject.

Many researchers have given definitions over the years. Refer to [3] for a historical survey of some of these definitions. One must also read the important work of Yiannis Moschovakis, e.g., [10]. Many of the given definitions are of the form "An algorithm is a program in this language/system/machine." This does not really conform to the current meaning of the word "algorithm." This is more in tune with the modern usage of the word "program." They all have a feel of being a specific implementation of an algorithm on a specific system. We would like to propose another definition.

We shall define an algorithm analogously to the way that Gottlob Frege defined a natural number. Basically Frege says that that the number 42 is the equivalence class of all sets of size 42. He looks at the set of all finite sets and makes an equivalence relation. Two finite sets are equivalent if there is a one-to-one onto function from one set to the other. The set of all equivalence classes under this equivalence relation forms the set of natural numbers. For us, an algorithm is an equivalence class of programs. Two programs are part of the same equivalence class if they are "essentially" the same. Each program is an expression (or an implementation) of the algorithm, just as every set of size 42 is an expression of the number 42.

For us, an algorithm is the sum total of all the programs that express it. In other words, we look at all computer programs and partition them into different subsets. Two programs in the same subset will be two implementations of the same algorithm. These two programs are "essentially" the same.

What does it mean for two programs to be "essentially" the same? Some examples are in order:

- One program might do $Process_1$ first and then do an unrelated $Process_2$ after. The other program will do the two unrelated processes in the opposite order.

- One program might do a certain process in a loop n times and the other program will unwind the loop and do it $n-1$ times and then do the the process again outside the loop.

- One program might do two unrelated processes in one loop, and the other program might do each of these two processes in its own loops.

In all these examples, the two programs are definitely performing the same function, and everyone would agree that both programs are implementations of the same algorithm. We are taking that subset of programs to be the definition of an algorithm.

Many relations that say when two programs are "essentially" the same will be given. However, it is doubtful that we are complete. Hence the word "Towards" in the title. Whether or not two programs are essentially the same, or whether or not a program is an implementation of a particular algorithm is really a subjective decision. We give relations that most people can agree on that these two programs are "essentially" the same, but we are well aware of the fact that others can come along and give more relations.

We consider the set of all programs which we might call **Programs**. An equivalence relation $\approx$ of "essentially the sameness" is then defined on this set. The set of equivalence classes **Programs**$/\approx$ shall then be called **Algorithms**. There is a nice onto function from ϕ : **Programs** $\longrightarrow$ **Algorithms**, that takes every program P to the equivalence class $\phi(P) = [P]$. One might think of any function ψ : **Algorithms** $\longrightarrow$ **Programs** such that $\phi \circ \psi = Id_{\mathbf{Algorithms}}$ as an "implementer." ψ takes an algorithm to an implementation of that algorithm.

To continue with this line of reasoning, there are many different algorithms that perform the same function. For example, Kruskal's algorithm and Prim's algorithm are two different ways of finding a minimum spanning tree of a weighted graph. Quicksort and Mergesort are two different algorithms to sort a list. There exists an equivalence relation on the set of all algorithms. Two algorithms are equivalent $\approx'$ if they perform the same function. We obtain **Algorithms**$/\approx'$ which we might call **Comp. Functions** or computable functions. It is an undecidable problem to determine when two programs perform the same computable function. Hence we might not be able to give the relation $\approx'$, nevertheless it exists. Again there is an onto function ϕ' : **Algorithms** $\longrightarrow$ **Comp. Functions**.

We summarize our intentions with the following picture.

$$\begin{tikzcd}
\text{Programing} & \mathbf{Programs} \arrow[d, two heads] \\
\text{Computer Science} & \mathbf{Algorithms} = \mathbf{Programs}/\approx \arrow[d, two heads] \\
\text{Mathematics} & \mathbf{Comp.\ Functions} = \mathbf{Algorithms}\ /\approx'
\end{tikzcd}$$

Programs are what programers, or "software engineers" deal with. Algorithms are the domain of computer scientists. Abstract functions are of interest to pure mathematicians.

We are not trying to make any ontological statement about the existence of algorithms. We are merely giving a mathematical way of describing how one might think of an algorithm. Human beings dealt with rational numbers for millennia before mathematicians decided that rational numbers are equivalence classes of pairs of integers:

$$\mathbb{Q} \quad = \quad \{(m,n) \in \mathbb{Z} \times \mathbb{Z} | n \neq 0\}/ \approx$$

where

$$(m,n) \approx (m',n') \text{ iff } mn' = nm'.$$

Similarly, one can think of the existence of algorithms in any way that one chooses. We are simply offering a mathematical way of presenting them.

There is a fascinating analogy between thinking of a rational number as an equivalence class of pairs of integers and our definition of an algorithm as an equivalence class of programs. Just as a rational number can only be expressed by an element of the equivalence class, so too, an algorithm can only be expressed by presenting an element of the equivalence class. When we write an algorithm, we are really writing a program. Pseudo-code is used to allow for ambiguities and not show any preference for a language. But it is, nevertheless, a program.

Another applicable analogy is that just as a rational number by itself has no structure; it is simply an equivalence class of pairs of integers. So too, an algorithm has no structure. In contrast, the set of rational numbers has much structure. So too, the set (category) of algorithms has much structure. $\mathbb{Q}$ is the smallest field that contains the natural numbers. We shall see in Section 4 that the category of algorithms is the initial *free* category with a strict product that is closed under recursion (i.e., has a natural number object).

When a human being talks about a rational number, he prefers to use the pair $(3,5) = \frac{3}{5}$ as opposed to the equivalent pair $(6,10)$, or the equivalent $(3000,5000)$. One might say that the rational number $(3,5)$ is a "canonical representation" of the equivalence class to which it belongs. It would be nice if there was a "canonical representation" of an algorithm. We speculate further on this ideas in the last section of this paper.

The question arises as to which programming language should we use? Rather than choosing one programming language to the exclusion of others, we shall look at the language of primitive recursive functions. We choose this language because of its beauty, its simplicity of presentation, and the fact that most readers are familiar with this language. A primitive recursive function can be described in many different ways. A description of a primitive recursive function is basically the same thing as a program in that it tells how to calculate a function. There is a basic correlation between programming concepts and the operations in generating descriptions of primitive recursive functions. Recursion is like a loop, and composition is just doing one process after another. We are well aware that we are limiting ourselves because the set of primitive recursive functions is a proper subset of the set of all computable functions. By limiting ourselves, we are going to get a proper subset of all algorithms. Even though we are, for the present time, restricting ourselves, we feel that the results we will get by just looking at primitive recursive functions are worthy of presenting.

Section 2 will review the basics of primitive recursive functions and show how they may be described by special labeled binary trees. Section 3 will then give many of the relations that tell when two descriptions of a primitive recursive algorithm are "essentially" the same. Section 4 will discuss the set of all algorithms. We shall give a universal categorical description of the category of algorithms. This is the only Section that uses category theory in a non-trivial way. Section 5 will discuss complexity results and show how complexity theory fits into our framework. We conclude this paper with a list of possible ways this work can progress.

Acknowledgement. Alex Heller, Florian Lengyel, and Dustin Mulcahey were forced to listen to me working through this material. I am endebted to them for being total gentlemen while suffering in silence. I am grateful to Rohit Parikh, Karen Kletter, Walter Dean, and the entire Graduate Center Logic and Games Seminar gang for listening to and commenting on this material. Shira Abraham, Eva Cogan, Joel Kammet, and Matthew K. Meyer read through many versions of this paper and commented on it. This paper would have been, no doubt, much better had I listened to all their advice. This work was inspired by a talk that Yuri Gurevich gave on his definition of an algorithm.

2 Descriptions of Primitive Recursive Functions

Rather than talking of computer programs, per say, we shall talk of descriptions of primitive recursive functions. For every primitive recursive function, there are many different methods of "building-up", or constructing the function from the basic functions. Each method is similar to a program.

We remind the reader that primitive recursive functions $\mathbb{N}^n \longrightarrow \mathbb{N}$ are "basic functions":

- null function $n : \mathbb{N} \longrightarrow \mathbb{N}$ where $n(x) = 0$
- successor function $s : \mathbb{N} \longrightarrow \mathbb{N}$ where $s(x) = x + 1$
- for each $k \in \mathbb{N}$ and for each $i \leq k$, a projection function $\pi_i^k : \mathbb{N}^k \longrightarrow \mathbb{N}$ where $\pi_i^k(x_1, x_2, \ldots x_k) = x_i$

and functions constructed from basic functions through a finite number of compositions and recursions.

We shall extend this definition in two non-essential ways. An $n-$tuple of primitive recursive functions $(f_1, f_2, \ldots f_n) : \mathbb{N}^m \longrightarrow \mathbb{N}^n$, shall also be called a primitive recursive function. Also, a constant function $k : * \longrightarrow \mathbb{N}$ shall also be called a primitive recursive function because for every $k \in \mathbb{N}$, the constant map may be written as follows:

$$\begin{tikzcd}
* \arrow[rr, "k"] & & \mathbb{N} \\
& \mathbb{N} \arrow[ul, "!"] \arrow[ur, swap, "s o s o \cdots o s o n"] &
\end{tikzcd}$$

where ! is the unique map from $\mathbb{N}$ to the one object set $*$ and where there are k copies of the successor map s in the right hand map.

Let us spend a few minutes reminding ourselves of basic facts about recursion. The simplest form of recursion is for a given integer k and a function $g : \mathbb{N} \longrightarrow \mathbb{N}$. From this one constructs $h : \mathbb{N} \longrightarrow \mathbb{N}$ as follows

$h(0) = k$

$h(n+1) = g(h(n))$.

A more complicated form of recursion — and the one we shall employ — is for a given function $f : \mathbb{N}^k \longrightarrow \mathbb{N}^m$ and a given function $g : \mathbb{N}^k \times \mathbb{N}^m \longrightarrow \mathbb{N}^m$. From this one constructs $h : \mathbb{N}^k \times \mathbb{N} \longrightarrow \mathbb{N}^m$ as

$h(x, 0) = f(x)$

$h(x, n+1) = g(x, h(x, n))$

where $x \in \mathbb{N}^k$ and $n \in \mathbb{N}$.

The most general form of recursion, and the definition usually given for primitive recursive functions is for a given function $f : \mathbb{N}^k \longrightarrow \mathbb{N}^m$ and a given function $g : \mathbb{N}^k \times \mathbb{N}^m \times \mathbb{N}^m \longrightarrow \mathbb{N}$. From this, one constructs $h : \mathbb{N}^k \times \mathbb{N} \longrightarrow \mathbb{N}^m$

$h(x, 0) = f(x)$

$h(x, n+1) = g(x, h(x, n), n)$

where $x \in \mathbb{N}^k$ and $n \in \mathbb{N}$.

We shall use the middle definition of recursion because the extra input variable in g does not add anything substantial. It simply makes things unnecessarily complicated. However, we are certain that any proposition that can be said about the second type of recursion, can also be said for the third type. See [1] Section 7.5, and [2] Section 5.5.

Although primitive recursive functions are usually described as closed only under composition and recursion, there is, in fact, another implicit operation, bracket, for which the functions are closed. Given primitive recursive functions $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ and $g : \mathbb{N}^k \longrightarrow \mathbb{N}$, there is a primitive recursive function $h = \langle f, g\rangle : \mathbb{N}^k \longrightarrow \mathbb{N} \times \mathbb{N}$. h is defined as

$$h(x) = (f(x), g(x))$$

for any $x \in \mathbb{N}^k$. We shall see that having this bracket operation is almost the same as having a product operation.

In order to save the eyesight of our poor reader, rather than writing too many exponents, we shall write a power of the set $\mathbb{N}$ for some fixed but arbitrary number as $\mathbb{A}, \mathbb{B}, \mathbb{C}$ etc. With this notation, we may write the recursion operation as follows: from functions $f : \mathbb{A} \longrightarrow \mathbb{B}$ and $g : \mathbb{A} \times \mathbb{B} \longrightarrow \mathbb{B}$ one constructs $h : \mathbb{A} \times \mathbb{N} \longrightarrow \mathbb{B}$.

If f and g are functions with the appropriate source and targets, then we shall write their composition as $h = f \circ g$. If they have the appropriate source and target for the bracket operations, we shall write the bracket operation as $h = \langle f, g\rangle$. We are in need of a similar notation for recursion. So if there are $f : \mathbb{A} \longrightarrow \mathbb{B}$ and $g : \mathbb{A} \times \mathbb{B} \longrightarrow \mathbb{B}$ we shall write the function that one obtains from them through recursion as $h = f \sharp g : \mathbb{A} \times \mathbb{N} \longrightarrow \mathbb{B}$

We are going to form a directed graph that contains all the descriptions of primitive recursive functions. We shall call this graph **PRdesc**. The vertices of the graph shall be powers of the natural numbers $\mathbb{N}^0 = *, \mathbb{N}, \mathbb{N}^2, \mathbb{N}^3, \ldots$. The edges of the graph shall be descriptions of primitive recursive functions. One should keep in mind the following intuitive picture.

$$\begin{tikzcd}
\cdots \arrow[r, bend left=15] \arrow[r] \arrow[r, bend right=15] & \mathbb{N}^k \arrow[l, bend left=80] \arrow[r, bend left=15] \arrow[r, bend left=5] \arrow[r, bend right=5] \arrow[r, bend right=15] \arrow[rr, bend left=10] \arrow[rr, bend right=10] & \cdots \arrow[l, bend right=80] \arrow[r, bend left=15] \arrow[r] \arrow[r, bend right=15] & \mathbb{N}^4 \arrow[r, bend left=15] \arrow[r, bend right=15] & \mathbb{N}^3 \arrow[r, bend left=15] \arrow[r] \arrow[r, bend right=15] & \mathbb{N}^2 \arrow[lll, bend right=50] \arrow[l, bend left=80] \arrow[r, bend left=15] \arrow[r] \arrow[r, bend right=15] & \mathbb{N} \arrow[l, bend right=80] \arrow[l, bend left=80] & * \arrow[l, bend left=15] \arrow[l] \arrow[l, bend right=15]
\end{tikzcd}$$

2.1 Trees

Each edge in **PRdesc** shall be a labeled binary tree whose leaves are basic functions and whose internal nodes are labeled by **C, R** or **B** for composition, recursion and bracket. Every internal node of the tree shall be derived from its left child and its right child. We shall use the following notation:

Composition.

$$\begin{tikzcd}[column sep=small]
& g \circ f : \mathbb{A} \longrightarrow \mathbb{C} \\
& \boxed{\mathbf{C}} \arrow[dl, no head] \arrow[dr, no head] \\
f : \mathbb{A} \longrightarrow \mathbb{B} & & g : \mathbb{B} \longrightarrow \mathbb{C}
\end{tikzcd}$$

Recursion.

$$\begin{tikzcd}[column sep=small]
& h = f \sharp g : \mathbb{A} \times \mathbb{N} \longrightarrow \mathbb{B} \\
& \boxed{\mathbf{R}} \arrow[dl, no head] \arrow[dr, no head] \\
f : \mathbb{A} \longrightarrow \mathbb{B} & & g : \mathbb{A} \times \mathbb{B} \longrightarrow \mathbb{B}
\end{tikzcd}$$

Bracket.

$$\begin{tikzcd}[column sep=small]
& \langle f, g \rangle : \mathbb{A} \longrightarrow \mathbb{B} \times \mathbb{C} \\
& \boxed{\mathbf{B}} \arrow[dl, no head] \arrow[dr, no head] \\
f : \mathbb{A} \longrightarrow \mathbb{B} & & g : \mathbb{A} \longrightarrow \mathbb{C}
\end{tikzcd}$$

PRdesc has more structure than a simple graph. There is a composition of edges. Given a tree $f : \mathbb{A} \longrightarrow \mathbb{B}$ and a tree $g : \mathbb{B} \longrightarrow \mathbb{C}$, there is another tree $g \circ f : \mathbb{A} \longrightarrow \mathbb{C}$. It is, however, important to realize that **PRdesc** is *not* a category. For three composable edges, the trees $h \circ (g \circ f)$ and $(h \circ g) \circ f$ exist and they perform the same operation, but they are, nevertheless, different programs and different trees. There is a composition of morphisms, but this composition is not associative.

Furthermore, for each object $\mathbb{A}$ of the graph, there is a distinguished morphism $\pi^{\mathbb{A}}_{\mathbb{A}} : \mathbb{A} \longrightarrow \mathbb{A}$. $\pi^{\mathbb{A}}_{\mathbb{A}} : \mathbb{A} \longrightarrow \mathbb{A}$ is a distinguished map, but it does not act like an identity. It is simply a function whose output is the same as its input.

2.2 Some Macros

Because the trees that we are going to construct can quickly become large and cumbersome, we will employ several programming shortcuts, called macros. We use the macros to improve readability as they can be rewritten as composition, recursion, and bracket.

Multiple Projections. There is a need to generalize the notion of a projection. The π^k_i accept k inputs and outputs one number. A multiple projection takes k inputs and outputs m outputs. Consider $\mathbb{A} = \mathbb{N}^k$ and the sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$ where each x_i is in $\{1, 2, \ldots, k\}$ and $i \neq j$ implies $x_i \neq x_j$. Let $\mathbb{B} = \mathbb{N}^m$, then for every X there exists $\pi^{\mathbb{N}^k}_{\mathbb{N}^m} = \pi^{\mathbb{A}}_{\mathbb{B}} : \mathbb{A} \longrightarrow \mathbb{B}$ as

$$\pi^{\mathbb{A}}_{\mathbb{B}} = \langle \pi^{\mathbb{A}}_{x_1}, \langle \pi^{\mathbb{A}}_{x_2}, \langle \ldots, \langle \pi^{\mathbb{A}}_{x_{m-1}}, \pi^{\mathbb{A}}_{x_m} \rangle\rangle \ldots \rangle.$$

In other words, $\pi^{\mathbb{A}}_{\mathbb{B}}$ outputs the proper numbers in the order described by X. Whenever possible, we shall be ambiguous with superscripts and subscripts.

Products. We would like a product of two maps. Given $f : \mathbb{A} \longrightarrow \mathbb{B}$ and $g : \mathbb{C} \longrightarrow \mathbb{D}$, we would like $f \times g : \mathbb{A} \times \mathbb{C} \longrightarrow \mathbb{B} \times \mathbb{D}$. The product can be defined using the bracket as

$$f \times g = \langle f \circ \pi^{\mathbb{A}\times\mathbb{C}}_{\mathbb{A}}, g \circ \pi^{\mathbb{A}\times\mathbb{C}}_{\mathbb{C}} \rangle$$

or in terms of trees

$$\begin{tikzcd}[column sep=small] & f \times g : \mathbb{A} \times \mathbb{C} \longrightarrow \mathbb{B} \times \mathbb{D} & \\ & \boxed{\mathbf{P}} \arrow[dl, dash] \arrow[dr, dash] & \\ f : \mathbb{A} \longrightarrow \mathbb{B} & & g : \mathbb{C} \longrightarrow \mathbb{D} \end{tikzcd}$$

is defined (=) as the tree

$$\begin{tikzcd}[column sep=tiny]
 & & & \begin{array}{c} f \times g = \langle f \circ \pi_{\mathbb{A}}^{\mathbb{A}\times\mathbb{C}}, g \circ \pi_{\mathbb{C}}^{\mathbb{A}\times\mathbb{C}} \rangle : \mathbb{A}\times\mathbb{C} \longrightarrow \mathbb{B}\times\mathbb{D} \\ \boxed{\mathbf{B}} \end{array} \arrow[dll, no head] \arrow[drr, no head] & & & \\
 & \begin{array}{c} f \circ \pi_{\mathbb{A}}^{\mathbb{A}\times\mathbb{C}} : \mathbb{A}\times\mathbb{C} \longrightarrow \mathbb{B} \\ \boxed{\mathbf{C}} \end{array} \arrow[dl, no head] \arrow[dr, no head] & & & & \begin{array}{c} g \circ \pi_{\mathbb{C}}^{\mathbb{A}\times\mathbb{C}} : \mathbb{A}\times\mathbb{C} \longrightarrow \mathbb{D} \\ \boxed{\mathbf{C}} \end{array} \arrow[dl, no head] \arrow[dr, no head] & \\
\pi_{\mathbb{A}}^{\mathbb{A}\times\mathbb{C}} : \mathbb{A}\times\mathbb{C} \longrightarrow \mathbb{A} & & f : \mathbb{A} \longrightarrow \mathbb{B} & & \pi_{\mathbb{C}}^{\mathbb{A}\times\mathbb{C}} : \mathbb{A}\times\mathbb{C} \longrightarrow \mathbb{C} & & g : \mathbb{C} \longrightarrow \mathbb{D}
\end{tikzcd}$$

Diagonal Map. A diagonal map will be used. A diagonal map is a map $\triangle : \mathbb{A} \longrightarrow \mathbb{A}\times\mathbb{A}$ where $x \mapsto (x, x)$. It can be defined as

$$\begin{tikzcd}[column sep=tiny]
\triangle : \mathbb{A} \longrightarrow \mathbb{A}\times\mathbb{A} & = & & \begin{array}{c} \langle \pi_{\mathbb{A}}^{\mathbb{A}}, \pi_{\mathbb{A}}^{\mathbb{A}} \rangle : \mathbb{A} \longrightarrow \mathbb{A}\times\mathbb{A}. \\ \boxed{\mathbf{B}} \end{array} \arrow[dl, no head] \arrow[dr, no head] & \\
 & & \pi_{\mathbb{A}}^{\mathbb{A}} : \mathbb{A} \longrightarrow \mathbb{A} & & \pi_{\mathbb{A}}^{\mathbb{A}} : \mathbb{A} \longrightarrow \mathbb{A}
\end{tikzcd}$$

We took the bracket operation as fundamental and from the bracket operation we derived the product operation and the diagonal map. We could have just as easily taken the product and the diagonal as fundamental and constructed the bracket as

$$\begin{tikzcd}
\mathbb{A} \arrow[rr, "\langle f,g \rangle"] \arrow[dr, "\triangle"'] & & \mathbb{B}\times\mathbb{C} \\
 & \mathbb{A}\times\mathbb{A}. \arrow[ur, "f\times g"'] &
\end{tikzcd}$$

We chose to do it this way, simply because the bracket is one operation as opposed to using both the product *and* the diagonal map.

Twist Map. We shall need to switch the order of inputs and outputs. The twist map shall be defined as

$$tw_{\mathbb{A},\mathbb{B}} = \pi_{\mathbb{B}}^{\mathbb{A}\times\mathbb{B}} \times \pi_{\mathbb{A}}^{\mathbb{A}\times\mathbb{B}} : \mathbb{A}\times\mathbb{B} \longrightarrow \mathbb{B}\times\mathbb{A}.$$

Or in terms of trees:

$$\begin{tikzcd}[column sep=small]
tw_{\mathbb{A},\mathbb{B}} : \mathbb{A}\times\mathbb{B} \longrightarrow \mathbb{B}\times\mathbb{A} \quad = & & \pi^{\mathbb{A}\times\mathbb{B}}_{\mathbb{B}} \times \pi^{\mathbb{A}\times\mathbb{B}}_{\mathbb{A}} : \mathbb{A}\times\mathbb{B} \longrightarrow \mathbb{B}\times\mathbb{A} & \\
 & & \boxed{\mathbf{P}} \arrow[dl, no head] \arrow[dr, no head] & \\
 & \pi^{\mathbb{A}\times\mathbb{B}}_{\mathbb{B}} : \mathbb{A}\times\mathbb{B} \longrightarrow \mathbb{B} & & \pi^{\mathbb{A}\times\mathbb{B}}_{\mathbb{A}} : \mathbb{A}\times\mathbb{B} \longrightarrow \mathbb{A}
\end{tikzcd}$$

Second Variable Product. Given a function $g_1 : \mathbb{A}\times\mathbb{B} \longrightarrow \mathbb{B}$ and a function $g_2 : \mathbb{A}\times\mathbb{B} \longrightarrow \mathbb{B}$, we would like to take the product of these two functions while keeping the first variable fixed. We define the operation

$$g_1 \boxtimes g_2 : \mathbb{A}\times\mathbb{B}\times\mathbb{B} \longrightarrow \mathbb{B}\times\mathbb{B}$$

on elements as follows

$$(g_1 \boxtimes g_2)(a, b_1, b_2) = (g_1(a, b_1), g_2(a, b_2)).$$

In terms of maps, $\boxtimes$ may be defined from the composition of the following maps:

$$g_1 \boxtimes g_2 = (g_1 \times g_2) \circ (\pi_{\mathbb{A}} \times tw_{\mathbb{A},\mathbb{B}} \times \pi_{\mathbb{B}}) \circ (\triangle \times \pi^{\mathbb{B}\times\mathbb{B}}_{\mathbb{B}\times\mathbb{B}}) :$$

$$\mathbb{A}\times\mathbb{B}\times\mathbb{B} \longrightarrow \mathbb{A}\times\mathbb{A}\times\mathbb{B}\times\mathbb{B} \longrightarrow \mathbb{A}\times\mathbb{B}\times\mathbb{A}\times\mathbb{B} \longrightarrow \mathbb{B}\times\mathbb{B}.$$

Since the second variable product is related to product which is derived from the bracket, we write it as

$$\begin{tikzcd}[column sep=small]
 & g_1 \boxtimes g_2 : \mathbb{A}\times\mathbb{B}\times\mathbb{B} \longrightarrow \mathbb{B}\times\mathbb{B} & \\
 & \boxed{\mathbf{B'}} \arrow[dl, no head] \arrow[dr, no head] & \\
g_1 : \mathbb{A}\times\mathbb{B} \longrightarrow \mathbb{B} & & g_2 : \mathbb{A}\times\mathbb{B} \longrightarrow \mathbb{B}
\end{tikzcd}$$

Second Variable Composition. Given a function $g_1 : \mathbb{A}\times\mathbb{D} \longrightarrow \mathbb{B}$ and a function $g_2 : \mathbb{A}\times\mathbb{C} \longrightarrow \mathbb{D}$, we would like to compose the output of g_2 into the second variable of g_1. We define the operation

$$g_1 \ddot{\circ} g_2 : \mathbb{A}\times\mathbb{C} \longrightarrow \mathbb{B}$$

on elements as follows

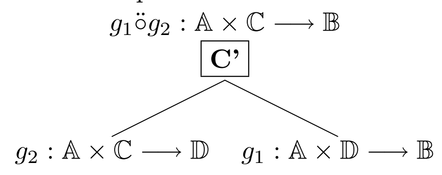
$$(g_1 \ddot{\circ} g_2)(a, c) = g_1(a, g_2(a, c)).$$

In terms of maps, $\ddot{\circ}$ may be defined as the composition of the following maps

$$g_1 \ddot{\circ} g_2 = (g_1) \circ (\pi^{\mathbb{A}}_{\mathbb{A}} \times g_2) \circ (\triangle \times \pi^{\mathbb{C}}_{\mathbb{C}}) :$$

$$\mathbb{A}\times\mathbb{C} \longrightarrow \mathbb{A}\times\mathbb{A}\times\mathbb{C} \longrightarrow \mathbb{A}\times\mathbb{D} \longrightarrow \mathbb{B}$$

We write second variable composition as


$g_1 \ddot{\circ} g_2 : \mathbb{A} \times \mathbb{C} \longrightarrow \mathbb{B}$
$\boxed{\mathbf{C'}}$
$g_2 : \mathbb{A} \times \mathbb{C} \longrightarrow \mathbb{D}$
$g_1 : \mathbb{A} \times \mathbb{D} \longrightarrow \mathbb{B}$

3 Relations

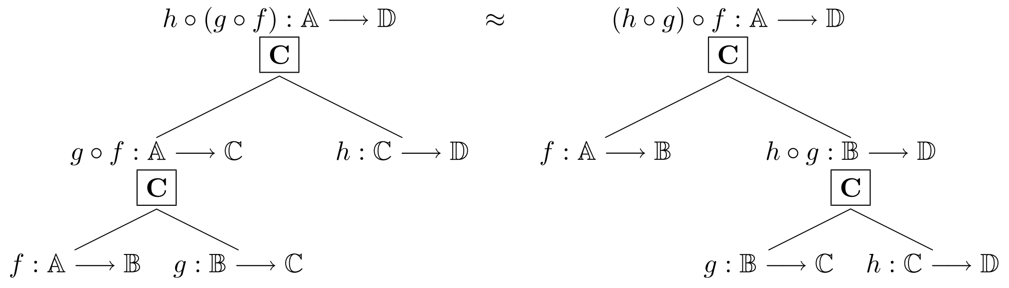
Given the operations of composition, recursion and bracket, what does it mean for us to say that two descriptions of a primitive recursive function are "essentially" the same? We shall examine these operations, and give relations to describe when two trees are essentially the same. If two trees are exactly alike except for a subtree that is equivalent to another tree, then we may replace the subtree with the equivalent tree.

3.1 Composition

Composition is Associative. That is, for any three composable maps f, g and h, we have

$$h \circ (g \circ f) \approx (h \circ g) \circ f.$$

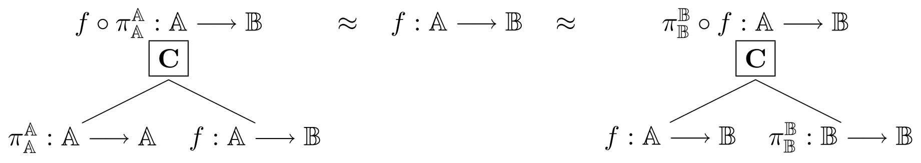
In terms of trees, we say that the following two trees are equivalent:


$h \circ (g \circ f) : \mathbb{A} \longrightarrow \mathbb{D}$
$\approx$
$(h \circ g) \circ f : \mathbb{A} \longrightarrow \mathbb{D}$
$\boxed{\mathbf{C}}$
$\boxed{\mathbf{C}}$
$g \circ f : \mathbb{A} \longrightarrow \mathbb{C}$
$h : \mathbb{C} \longrightarrow \mathbb{D}$
$f : \mathbb{A} \longrightarrow \mathbb{B}$
$h \circ g : \mathbb{B} \longrightarrow \mathbb{D}$
$\boxed{\mathbf{C}}$
$\boxed{\mathbf{C}}$
$f : \mathbb{A} \longrightarrow \mathbb{B}$
$g : \mathbb{B} \longrightarrow \mathbb{C}$
$g : \mathbb{B} \longrightarrow \mathbb{C}$
$h : \mathbb{C} \longrightarrow \mathbb{D}$

Projections as Identity of Composition. The projections $\pi^{\mathbb{A}}_{\mathbb{A}}$ and $\pi^{\mathbb{B}}_{\mathbb{B}}$ act like identity maps. That means for any $f : \mathbb{A} \longrightarrow \mathbb{B}$, we have

$$f \circ \pi^{\mathbb{A}}_{\mathbb{A}} \approx f \approx \pi^{\mathbb{B}}_{\mathbb{B}} \circ f.$$

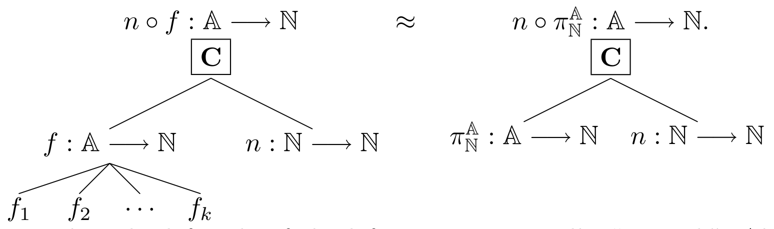
In terms of trees this amounts to


$f \circ \pi^{\mathbb{A}}_{\mathbb{A}} : \mathbb{A} \longrightarrow \mathbb{B}$
$\approx$
$f : \mathbb{A} \longrightarrow \mathbb{B}$
$\approx$
$\pi^{\mathbb{B}}_{\mathbb{B}} \circ f : \mathbb{A} \longrightarrow \mathbb{B}$
$\boxed{\mathbf{C}}$
$\boxed{\mathbf{C}}$
$\pi^{\mathbb{A}}_{\mathbb{A}} : \mathbb{A} \longrightarrow \mathbb{A}$
$f : \mathbb{A} \longrightarrow \mathbb{B}$
$f : \mathbb{A} \longrightarrow \mathbb{B}$
$\pi^{\mathbb{B}}_{\mathbb{B}} : \mathbb{B} \longrightarrow \mathbb{B}$

Composition and the Null Function. The null function always outputs a 0 no matter what the input is. So for any function $f : \mathbb{A} \longrightarrow \mathbb{N}$, if we are going to compose f with the null function, then f might as well be substituted with a projection, i.e.,

$$n \circ f \approx n \circ \pi_{\mathbb{N}}^{\mathbb{A}}.$$

In terms of trees:


$n \circ f : \mathbb{A} \longrightarrow \mathbb{N}$
$\approx$
$n \circ \pi_{\mathbb{N}}^{\mathbb{A}} : \mathbb{A} \longrightarrow \mathbb{N}.$
C
C
$f : \mathbb{A} \longrightarrow \mathbb{N}$
$n : \mathbb{N} \longrightarrow \mathbb{N}$
$\pi_{\mathbb{N}}^{\mathbb{A}} : \mathbb{A} \longrightarrow \mathbb{N}$
$n : \mathbb{N} \longrightarrow \mathbb{N}$
f_1 f_2 $\cdots$ f_k

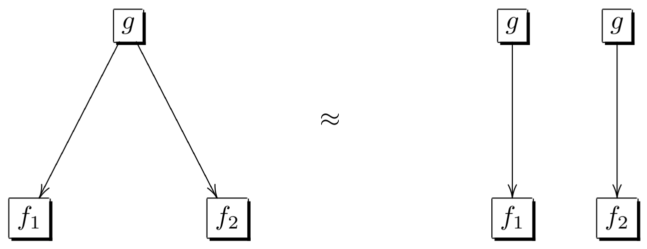
Notice that the left side of the left tree is essentially "pruned." Although there is much information on the left side of the left tree, it is not important. It can be substituted with another tree that does not have that information.

3.2 Composition and Bracket

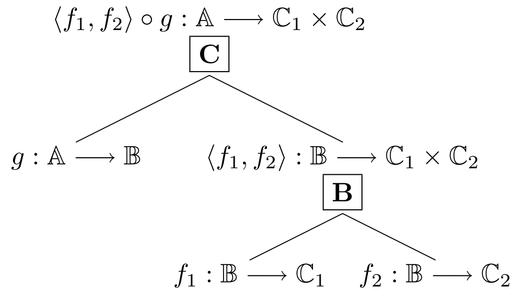
Composition Distributes Over the Bracket on the Right. For $g : \mathbb{A} \longrightarrow \mathbb{B}$, $f_1 : \mathbb{B} \longrightarrow \mathbb{C}_1$ and $f_2 : \mathbb{B} \longrightarrow \mathbb{C}_2$, we have

$$\langle f_1, f_2 \rangle \circ g \approx \langle f_1 \circ g, f_2 \circ g \rangle.$$

In terms of procedures, this says that doing g and then doing both f_1 and f_2 is the same as doing both $f_1 \circ g$ and $f_2 \circ g$, i.e., the following two flowcharts are essentially the same.



In terms of trees, this amounts to saying that this tree


$\langle f_1, f_2 \rangle \circ g : \mathbb{A} \longrightarrow \mathbb{C}_1 \times \mathbb{C}_2$
C
$g : \mathbb{A} \longrightarrow \mathbb{B}$
$\langle f_1, f_2 \rangle : \mathbb{B} \longrightarrow \mathbb{C}_1 \times \mathbb{C}_2$
B
$f_1 : \mathbb{B} \longrightarrow \mathbb{C}_1$
$f_2 : \mathbb{B} \longrightarrow \mathbb{C}_2$

is equivalent ($\approx$) to this tree

$$\begin{tikzcd}
 & & & \begin{array}{c}\langle f_1 \circ g, f_2 \circ g\rangle : \mathbb{A} \longrightarrow \mathbb{C}_1 \times \mathbb{C}_2 \\ \boxed{\mathbf{B}}\end{array} \arrow[dll, no head] \arrow[drr, no head] & & & \\
 & \begin{array}{c}f_1 \circ g : \mathbb{A} \longrightarrow \mathbb{C}_1 \\ \boxed{\mathbf{C}}\end{array} \arrow[dl, no head] \arrow[dr, no head] & & & & \begin{array}{c}f_2 \circ g : \mathbb{A} \longrightarrow \mathbb{C}_2 \\ \boxed{\mathbf{C}}\end{array} \arrow[dl, no head] \arrow[dr, no head] & \\
g : \mathbb{A} \longrightarrow \mathbb{B} & & f_1 : \mathbb{B} \longrightarrow \mathbb{C}_1 & & g : \mathbb{A} \longrightarrow \mathbb{B} & & f_2 : \mathbb{B} \longrightarrow \mathbb{C}_2
\end{tikzcd}$$

It is important to realize that it does not make sense to require composition to distribute over bracket on the left:

$$g \circ \langle f_1, f_2\rangle \not\approx \langle g \circ f_1, g \circ f_2\rangle.$$

The following two flowcharts are *not* essentially the same.

$$\begin{tikzcd}
\boxed{f_1} \arrow[dr] & & \boxed{f_2} \arrow[dl] & & \boxed{f_1} \arrow[d] & \boxed{f_2} \arrow[d] \\
 & \boxed{g} & & \not\approx & \boxed{g} & \boxed{g}
\end{tikzcd}$$

The left g requires two inputs. The right g's only require one.

3.3 Bracket

Bracket is Associative. The bracket is associative. For any three maps f, g, and h with the same domain, we have

$$\langle\langle f, g\rangle, h\rangle \approx \langle f, \langle g, h\rangle\rangle$$

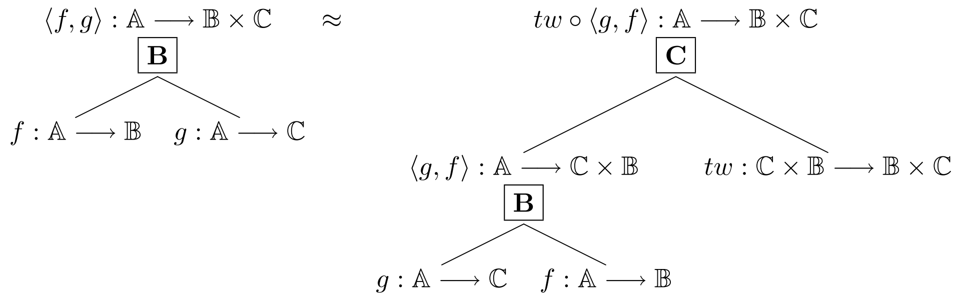
In terms of trees, this amounts to

$$\begin{tikzcd}
 & & \begin{array}{c}\langle\langle f, g\rangle h\rangle : \mathbb{A} \longrightarrow \mathbb{B} \times \mathbb{C} \times \mathbb{D} \\ \boxed{\mathbf{B}}\end{array} \arrow[dl, no head] \arrow[dr, no head] & & \approx & & \begin{array}{c}\langle f, \langle g, h\rangle\rangle : \mathbb{A} \longrightarrow \mathbb{B} \times \mathbb{C} \times \mathbb{D} \\ \boxed{\mathbf{B}}\end{array} \arrow[dl, no head] \arrow[dr, no head] & & \\
 & \begin{array}{c}\langle f, g\rangle : \mathbb{A} \longrightarrow \mathbb{B} \times \mathbb{C} \\ \boxed{\mathbf{B}}\end{array} \arrow[dl, no head] \arrow[dr, no head] & & h : \mathbb{A} \longrightarrow \mathbb{D} & & f : \mathbb{A} \longrightarrow \mathbb{B} & & \begin{array}{c}\langle g, h\rangle : \mathbb{B} \longrightarrow \mathbb{C} \times \mathbb{D} \\ \boxed{\mathbf{B}}\end{array} \arrow[dl, no head] \arrow[dr, no head] & \\
f : \mathbb{A} \longrightarrow \mathbb{B} & & g : \mathbb{A} \longrightarrow \mathbb{C} & & & & g : \mathbb{A} \longrightarrow \mathbb{C} & & h : \mathbb{A} \longrightarrow \mathbb{D}
\end{tikzcd}$$

Bracket is Almost Commutative. It is not essential what is written in the first or the second place. For any two maps f and g with the same domain,

$$\langle f, g\rangle \approx tw \circ \langle g, f\rangle.$$

In terms of trees, this amounts to


$\langle f, g\rangle : \mathbb{A} \longrightarrow \mathbb{B} \times \mathbb{C}$
$\approx$
$tw \circ \langle g, f\rangle : \mathbb{A} \longrightarrow \mathbb{B} \times \mathbb{C}$
$\boxed{\mathbf{B}}$
$\boxed{\mathbf{C}}$
$f : \mathbb{A} \longrightarrow \mathbb{B}$
$g : \mathbb{A} \longrightarrow \mathbb{C}$
$\langle g, f\rangle : \mathbb{A} \longrightarrow \mathbb{C} \times \mathbb{B}$
$tw : \mathbb{C} \times \mathbb{B} \longrightarrow \mathbb{B} \times \mathbb{C}$
$\boxed{\mathbf{B}}$
$g : \mathbb{A} \longrightarrow \mathbb{C}$
$f : \mathbb{A} \longrightarrow \mathbb{B}$

Twist is Idempotent. There are other relations that the twist map must respect. Idempotent means

$$tw_{\mathbb{A},\mathbb{B}} \circ tw_{\mathbb{A},\mathbb{B}} \approx \pi^{\mathbb{A}\times\mathbb{B}}_{\mathbb{A}\times\mathbb{B}} : \mathbb{A} \times \mathbb{B} \longrightarrow \mathbb{A} \times \mathbb{B}.$$

Twist is Coherent. We would like the twist maps of three elements to get along with themselves.

$$(tw_{\mathbb{B},\mathbb{C}} \times \pi_{\mathbb{A}}) \circ (\pi_{\mathbb{B}} \times tw_{\mathbb{A},\mathbb{C}}) \circ (tw_{\mathbb{A},\mathbb{B}} \times \pi_{\mathbb{C}}) \approx (\pi_{\mathbb{C}} \times tw_{\mathbb{A},\mathbb{B}}) \circ (tw_{\mathbb{A},\mathbb{C}} \times \pi_{\mathbb{B}}) \circ (\pi_{\mathbb{A}} \times tw_{\mathbb{B},\mathbb{C}}).$$

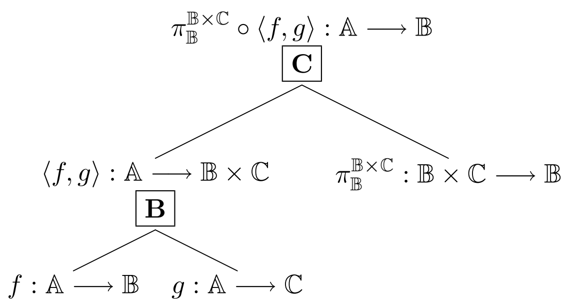
This is called the hexagon law or the third Reidermeister move. Given the idempotence and hexagon laws, it is a theorem that there is a unique twist map made of smaller twist maps between any two products of elements ([9] Section XI.4).

Bracket and Projections. A bracket followed by a projection onto the first output means the second output is ignored.

$$f \approx \pi^{\mathbb{B}\times\mathbb{C}}_{\mathbb{B}} \circ \langle f, g\rangle$$

In terms of trees, this amounts to

$f : \mathbb{A} \longrightarrow \mathbb{B} \qquad \approx$

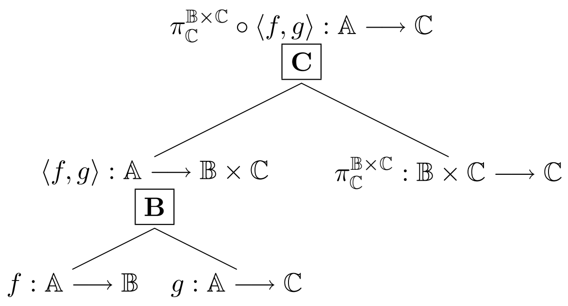

$\pi^{\mathbb{B}\times\mathbb{C}}_{\mathbb{B}} \circ \langle f, g\rangle : \mathbb{A} \longrightarrow \mathbb{B}$
$\boxed{\mathbf{C}}$
$\langle f, g\rangle : \mathbb{A} \longrightarrow \mathbb{B}\times\mathbb{C}$
$\pi^{\mathbb{B}\times\mathbb{C}}_{\mathbb{B}} : \mathbb{B}\times\mathbb{C} \longrightarrow \mathbb{B}$
$\boxed{\mathbf{B}}$
$f : \mathbb{A} \longrightarrow \mathbb{B}$
$g : \mathbb{A} \longrightarrow \mathbb{C}$

Similarly for a projection onto the second output.

$$g \approx \pi^{\mathbb{B}\times\mathbb{C}}_{\mathbb{C}} \circ \langle f, g\rangle$$

Or

$g : \mathbb{A} \longrightarrow \mathbb{C} \qquad \approx$


$\pi^{\mathbb{B}\times\mathbb{C}}_{\mathbb{C}} \circ \langle f, g\rangle : \mathbb{A} \longrightarrow \mathbb{C}$
$\boxed{\mathbf{C}}$
$\langle f, g\rangle : \mathbb{A} \longrightarrow \mathbb{B}\times\mathbb{C}$
$\pi^{\mathbb{B}\times\mathbb{C}}_{\mathbb{C}} : \mathbb{B}\times\mathbb{C} \longrightarrow \mathbb{C}$
$\boxed{\mathbf{B}}$
$f : \mathbb{A} \longrightarrow \mathbb{B}$
$g : \mathbb{A} \longrightarrow \mathbb{C}$

3.4 Bracket and Recursion

When there are two unrelated processes, we can perform both of them in one loop or we can perform each of them in its own loop.

```
$h = \langle f_1(x), f_2(x)\rangle$
For $i$ = 1 to $n$
    $h = (g_1(x, \pi_1 h), g_2(x, \pi_2 h))$
```

$\approx$

```
$h_1 = f_1(x)$
For $i$ = 1 to $n$
    $h_1 = g_1(x, h_1)$
```

;

```
$h_2 = f_2(x)$
For $i$ = 1 to $n$
    $h_2 = g_2(x, h_2)$
```

In $\sharp$ notation this amounts to saying

$$h = \langle f_1, f_2\rangle \sharp (g_1 \boxtimes g_2) \approx \langle f_1 \sharp g_1, f_2 \sharp g_2\rangle = \langle h_1, h_2\rangle.$$

In terms of trees this says that this tree:

$$\begin{tikzcd}[column sep=tiny]
 & & & \begin{array}{c} h = ((\langle f_1, f_2\rangle\sharp(g_1 \boxtimes g_2)) : \mathbb{A}\times\mathbb{N} \longrightarrow \mathbb{B}\times\mathbb{B} \\ \boxed{\mathbf{R}} \end{array} \arrow[dll, no head] \arrow[drr, no head] & & & \\
 & \begin{array}{c} \langle f_1, f_2\rangle : \mathbb{A} \longrightarrow \mathbb{B}\times\mathbb{B} \\ \boxed{\mathbf{B}} \end{array} \arrow[dl, no head] \arrow[dr, no head] & & & & \begin{array}{c} g_1 \boxtimes g_2 : \mathbb{A}\times\mathbb{B}\times\mathbb{B} \longrightarrow \mathbb{B}\times\mathbb{B} \\ \boxed{\mathbf{B'}} \end{array} \arrow[dl, no head] \arrow[dr, no head] & \\
f_1 : \mathbb{A} \longrightarrow \mathbb{B} & & f_2 : \mathbb{A} \longrightarrow \mathbb{B} & & g_1 : \mathbb{A}\times\mathbb{B} \longrightarrow \mathbb{B} & & g_2 : \mathbb{A}\times\mathbb{B} \longrightarrow \mathbb{B}
\end{tikzcd}$$

is equivalent ($\approx$) to this tree:

$$\begin{tikzcd}[column sep=tiny]
 & & & \begin{array}{c} \langle h_1, h_2\rangle = \langle f_1\sharp g_1, f_2\sharp g_2\rangle : \mathbb{A}\times\mathbb{N} \longrightarrow \mathbb{B}\times\mathbb{B} \\ \boxed{\mathbf{B}} \end{array} \arrow[dll, no head] \arrow[drr, no head] & & & \\
 & \begin{array}{c} h_1 = (f_1\sharp g_1) : \mathbb{A}\times\mathbb{N} \longrightarrow \mathbb{B} \\ \boxed{\mathbf{R}} \end{array} \arrow[dl, no head] \arrow[dr, no head] & & & & \begin{array}{c} h_2 = (f_2\sharp g_2) : \mathbb{A}\times\mathbb{N} \longrightarrow \mathbb{B} \\ \boxed{\mathbf{R}} \end{array} \arrow[dl, no head] \arrow[dr, no head] & \\
f_1 : \mathbb{A} \longrightarrow \mathbb{B} & & g_1 : \mathbb{A}\times\mathbb{B} \longrightarrow \mathbb{B} & & f_2 : \mathbb{A} \longrightarrow \mathbb{B} & & g_2 : \mathbb{A}\times\mathbb{B} \longrightarrow \mathbb{B}.
\end{tikzcd}$$

3.5 Recursion and Composition

Unwinding a Recursive Loop. Consider the following two algorithms

```
$h = f(x)$
For $i$ = 1 to $n$
    $h = g_1(x, h)$
    $h = g_2(x, h)$
```

```
$h' = g_1(x, f(x))$
For $i$ = 1 to n-1
    $h' = g_2(x, h')$
    $h' = g_1(x, h')$
$h' = g_2(x, h')$
```

This is the most general form of unwinding a loop. If g_1 is the identity process (does nothing), these become

```
$h = f(x)$
For $i$ = 1 to $n$
    $h = g_2(x, h)$
```

```
$h' = f(x)$
For $i$ = 1 to n-1
    $h' = g_2(x, h')$
$h' = g_2(x, h')$.
```

If g_2 is the identity process, these become

```
$h = f(x)$
For $i$ = 1 to $n$
$h = g_1(x, h))$
```

```
$h' = g_1(x, f(x))$
For $i$ = 1 to $n$-1
    $h' = g_1(x, h')$.
```

In terms of recursion, the most general form of unwinding a loop, the left top box coincides with
$h(x, 0) = f(x)$
$h(x, n+1) = g_2(x, g_1(x, h(x, n)))$.
The right top box coincides with:
$h'(x, 0) = g_1(x, f(x))$
$h'(x, n+1) = g_1(x, g_2(x, h'(x, n)))$.
How are these two recursions related? We claim that for all $n \in \mathbb{N}$

$$g_1(x, h(x, n)) = h'(x, n).$$

This may be proven by induction. The $n = 0$ case is trivial. Assume it is true for k, and we shall show it is true for $k + 1$.

$$g_1(x, h(x, k{+}1)) = g_1(x, g_2(x, g_1(x, h(x, k)))) = g_1(x, g_2(x, h'(x, k))) = h'(x, k{+}1).$$

The first equality is from the definition of h; the second equality is the induction hypothesis; and the third equality is from the definition of h'.

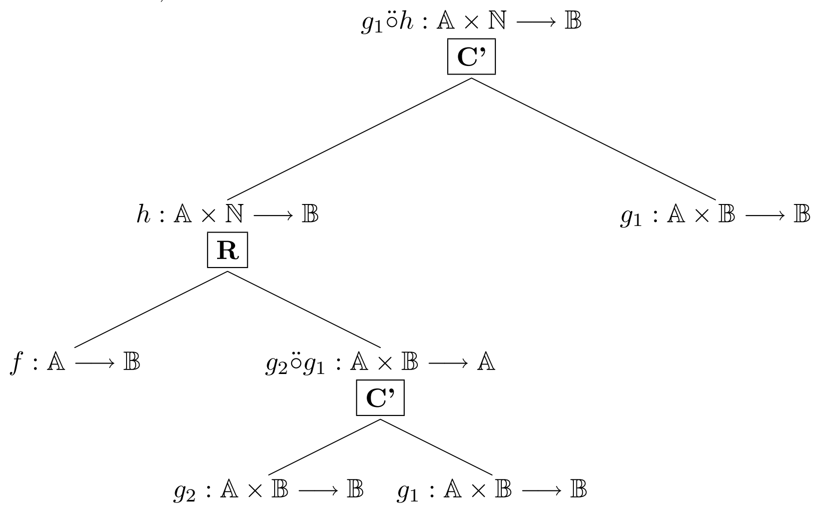
Although $g_1 \ddot{o} h$ and g_2 are constructed differently, they are essentially the same program and hence the same algorithm and for any input, they output the same numbers. So we shall set them equivalent to each other:

$$g_1 \ddot{o} h \approx h'$$

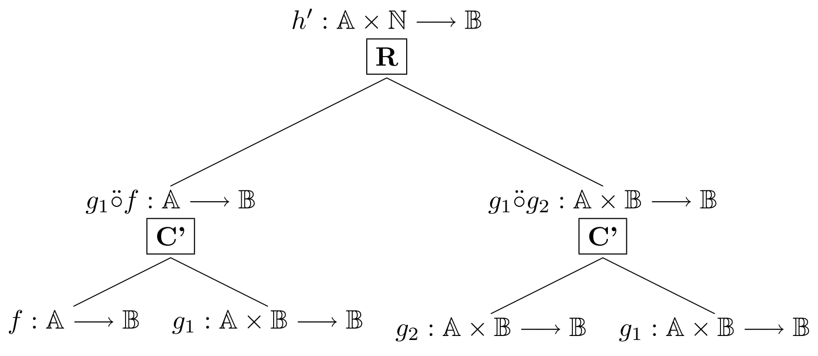
If one leaves out the h and h' and uses the $\sharp$ notation, this becomes

$$g_1 \ddot{o}(f \sharp (g_2 \ddot{o} g_1)) \approx (g_1 \ddot{o} f) \sharp (g_1 \ddot{o} g_2).$$

In terms of trees, this means that


$g_1 \ddot{o} h : \mathbb{A} \times \mathbb{N} \longrightarrow \mathbb{B}$
C'
$h : \mathbb{A} \times \mathbb{N} \longrightarrow \mathbb{B}$
R
$g_1 : \mathbb{A} \times \mathbb{B} \longrightarrow \mathbb{B}$
$f : \mathbb{A} \longrightarrow \mathbb{B}$
$g_2 \ddot{o} g_1 : \mathbb{A} \times \mathbb{B} \longrightarrow \mathbb{A}$
C'
$g_2 : \mathbb{A} \times \mathbb{B} \longrightarrow \mathbb{B}$
$g_1 : \mathbb{A} \times \mathbb{B} \longrightarrow \mathbb{B}$

is equivalent ($\approx$) to


$h' : \mathbb{A} \times \mathbb{N} \longrightarrow \mathbb{B}$
$\boxed{\mathbf{R}}$
$g_1 \ddot{\circ} f : \mathbb{A} \longrightarrow \mathbb{B}$
$\boxed{\mathbf{C'}}$
$g_1 \ddot{\circ} g_2 : \mathbb{A} \times \mathbb{B} \longrightarrow \mathbb{B}$
$\boxed{\mathbf{C'}}$
$f : \mathbb{A} \longrightarrow \mathbb{B}$
$g_1 : \mathbb{A} \times \mathbb{B} \longrightarrow \mathbb{B}$
$g_2 : \mathbb{A} \times \mathbb{B} \longrightarrow \mathbb{B}$
$g_1 : \mathbb{A} \times \mathbb{B} \longrightarrow \mathbb{B}$

Recursion and Null. If h is defined by recursion from f and g, i.e. $h = f \sharp g$, then by definition of recursion

$$h(x, 0) = f(x)$$

or

$$h(x, n(y)) = f(x)$$

where n is the null function and $y \in \mathbb{N}$. Or $h \ddot{\circ} n = f$. We shall set these equivalent

$$h \ddot{\circ} n \approx f$$

Using the $\sharp$ notation, this amounts to:

$$(f \sharp g) \ddot{\circ} n \approx f.$$

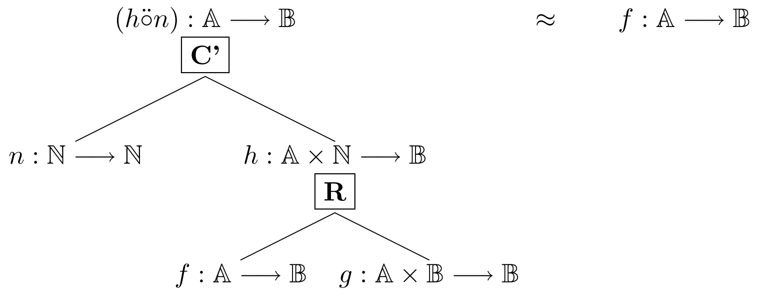
In terms of algorithms, this amounts to saying that the following two algorithms are equivalent:

```
h = f(x)
For i= 0 to 0
   h = g(x,h)
```

$\approx$

```
h = f(x)
```

In terms of trees, this is


$(h \ddot{\circ} n) : \mathbb{A} \longrightarrow \mathbb{B}$
$\approx$
$f : \mathbb{A} \longrightarrow \mathbb{B}$
$\boxed{\mathbf{C'}}$
$n : \mathbb{N} \longrightarrow \mathbb{N}$
$h : \mathbb{A} \times \mathbb{N} \longrightarrow \mathbb{B}$
$\boxed{\mathbf{R}}$
$f : \mathbb{A} \longrightarrow \mathbb{B}$
$g : \mathbb{A} \times \mathbb{B} \longrightarrow \mathbb{B}$

Notice that the g on the left tree is not on the right tree.

Recursion and Successor. Let h be defined by recursion from f and g, i.e. $h = f\sharp g$. Then by definition of recursion

$$h(x, k+1) = g(x, h(x, k))$$

or

$$h(x, s(k)) = g(x, h(x, k))$$

where s is the successor function and $k \in \mathbb{N}$. Or $h\ddot{\circ}s = g\ddot{\circ}h$. We shall set these equivalent

$$h\ddot{\circ}s \approx g\ddot{\circ}h$$

Using the $\sharp$ notation, this becomes.

$$(f\sharp g)\ddot{\circ}s \approx g\ddot{\circ}(f\sharp g).$$

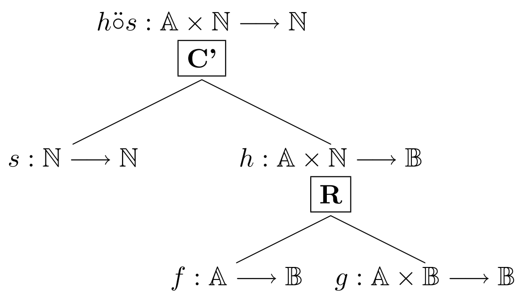
In terms of algorithms, this says that the following two algorithms are equivalent

```
$h = f(x)$
For $i$ = 1 to k+1
    $h = g(x, h)$
```

$\approx$

```
$h = f(x)$
For $i$ = 1 to k
    $h = g(x, h)$
$h = g(x, h)$
```

In terms of trees, this says that the tree


$h\ddot{\circ}s : \mathbb{A} \times \mathbb{N} \longrightarrow \mathbb{N}$
C'
$s : \mathbb{N} \longrightarrow \mathbb{N}$
$h : \mathbb{A} \times \mathbb{N} \longrightarrow \mathbb{B}$
R
$f : \mathbb{A} \longrightarrow \mathbb{B}$
$g : \mathbb{A} \times \mathbb{B} \longrightarrow \mathbb{B}$

is set equivalent $\approx$ to the tree

$$\begin{tikzcd}
 & & g\ddot{o}h : \mathbb{A}\times\mathbb{N} \longrightarrow \mathbb{N} & \\
 & & \boxed{\textbf{C'}} \arrow[dl, no head] \arrow[dr, no head] & \\
 & h : \mathbb{A}\times\mathbb{N} \longrightarrow \mathbb{B} & & g : \mathbb{A}\times\mathbb{B} \longrightarrow \mathbb{B} \\
 & \boxed{\textbf{R}} \arrow[dl, no head] \arrow[dr, no head] & & \\
f : \mathbb{A} \longrightarrow \mathbb{B} & & g : \mathbb{A}\times\mathbb{B} \longrightarrow \mathbb{B} &
\end{tikzcd}$$

3.6 Products

The product is associative. That is for any three maps $f : \mathbb{A} \longrightarrow \mathbb{A}'$, $g : \mathbb{B} \longrightarrow \mathbb{B}'$ and $h : \mathbb{C} \longrightarrow \mathbb{C}'$ the two products are equivalent:

$$f \times (g \times h) \approx (f \times g) \times h : \mathbb{A} \times \mathbb{B} \times \mathbb{C} \longrightarrow \mathbb{A}' \times \mathbb{B}' \times \mathbb{C}'.$$

This follows immediately from the associativity of bracket.

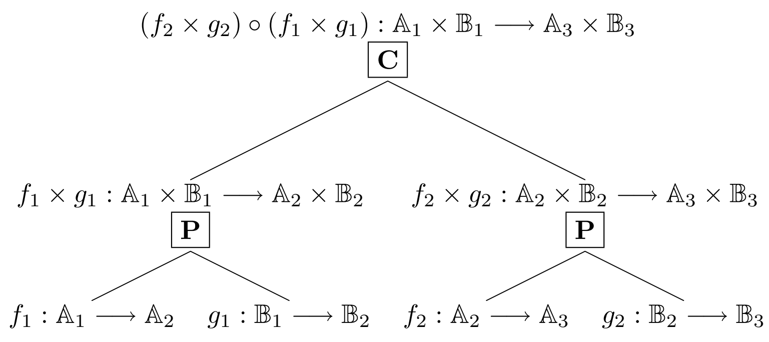
Interchange Rule. We must show that the product and the composition respect each other. In terms of maps, this corresponds to the following situation:

$$\begin{tikzcd}
\mathbb{A}_1 \arrow[d, "f_1"'] \arrow[dd, bend right=60, "f_2\circ f_1"'] & \mathbb{A}_1 \times B_1 \arrow[l, "\pi"'] \arrow[r, "\pi"] \arrow[d, "f_1\times g_1"'] & \mathbb{B}_1 \arrow[d, "g_1"'] \arrow[dd, bend left=60, "g_2\circ g_1"] \\
\mathbb{A}_2 \arrow[d, "f_2"'] & \mathbb{A}_2 \times B_2 \arrow[l, "\pi"'] \arrow[r, "\pi"] \arrow[d, "f_2\times g_2"'] & \mathbb{B}_2 \arrow[d, "g_2"'] \\
\mathbb{A}_3 & \mathbb{A}_3 \times B_3 \arrow[l, "\pi"'] \arrow[r, "\pi"] & \mathbb{B}_3
\end{tikzcd}$$

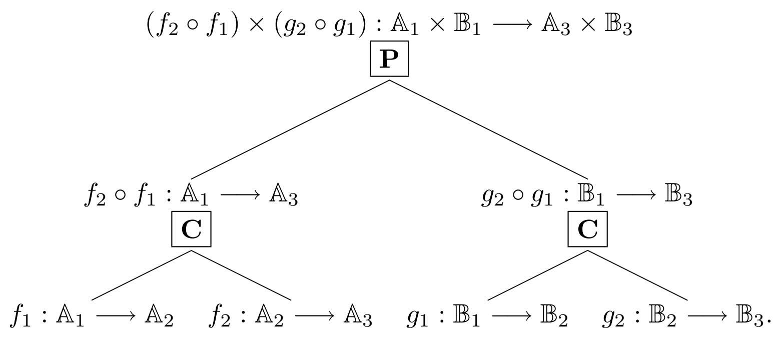
$(f_2 \times g_2) \circ (f_1 \times g_1)$ and $(f_2 \circ f_1) \times (g_2 \circ g_1)$ are two ways of getting from $\mathbb{A}_1 \times \mathbb{B}_1$ to $\mathbb{A}_3 \times \mathbb{B}_3$. We shall declare these two methods equivalent:

$$(f_2 \times g_2) \circ (f_1 \times g_1) \approx (f_2 \circ f_1) \times (g_2 \circ g_1).$$

In terms of trees, this tree:


$(f_2 \times g_2) \circ (f_1 \times g_1) : \mathbb{A}_1 \times \mathbb{B}_1 \longrightarrow \mathbb{A}_3 \times \mathbb{B}_3$
$\boxed{\mathbf{C}}$
$f_1 \times g_1 : \mathbb{A}_1 \times \mathbb{B}_1 \longrightarrow \mathbb{A}_2 \times \mathbb{B}_2$
$\boxed{\mathbf{P}}$
$f_2 \times g_2 : \mathbb{A}_2 \times \mathbb{B}_2 \longrightarrow \mathbb{A}_3 \times \mathbb{B}_3$
$\boxed{\mathbf{P}}$
$f_1 : \mathbb{A}_1 \longrightarrow \mathbb{A}_2$
$g_1 : \mathbb{B}_1 \longrightarrow \mathbb{B}_2$
$f_2 : \mathbb{A}_2 \longrightarrow \mathbb{A}_3$
$g_2 : \mathbb{B}_2 \longrightarrow \mathbb{B}_3$

is equivalent ($\approx$) to this tree:


$(f_2 \circ f_1) \times (g_2 \circ g_1) : \mathbb{A}_1 \times \mathbb{B}_1 \longrightarrow \mathbb{A}_3 \times \mathbb{B}_3$
$\boxed{\mathbf{P}}$
$f_2 \circ f_1 : \mathbb{A}_1 \longrightarrow \mathbb{A}_3$
$\boxed{\mathbf{C}}$
$g_2 \circ g_1 : \mathbb{B}_1 \longrightarrow \mathbb{B}_3$
$\boxed{\mathbf{C}}$
$f_1 : \mathbb{A}_1 \longrightarrow \mathbb{A}_2$
$f_2 : \mathbb{A}_2 \longrightarrow \mathbb{A}_3$
$g_1 : \mathbb{B}_1 \longrightarrow \mathbb{B}_2$
$g_2 : \mathbb{B}_2 \longrightarrow \mathbb{B}_3.$

One should realize that this equivalence is not anything new added to our list of equivalences. It is actually a consequence of the definition of product and the equivalences that we assume about bracket. In detail

$$(f_2 \times g_2) \circ (f_1 \times g_1) = \langle f_2\pi, g_2\pi \rangle \circ \langle f_1\pi, g_1\pi \rangle \approx \langle f_2\pi\langle f_1\pi, g_1\pi \rangle\rangle, g_2\pi\langle f_1\pi, g_1\pi \rangle\rangle$$
$$\approx \langle f_2 \circ f_1\pi, g_2 \circ g_1\pi \rangle = (f_2 \circ f_1) \times (g_2 \circ g_1).$$

The first and the last equality are from the definition of product. The first equivalence comes from the fact that composition distributes over bracket. The second equivalence is a consequence of the relationship between the projection maps and the bracket.

4 Algorithms

We have given relations telling when two programs/trees/descriptions are similar. We would like to look at the equivalence classes that these relations generate. The relations split up into two disjoint sets: those for which there is a loss of information and those for which there is no loss of information. Let us call the former set of relations **(I)** and the latter set **(II)**. The following relations are in group **(I)**.

1. Null Function and Composition: $n \circ f \approx n \circ \pi^{\mathbb{A}}_{\mathbb{N}}$
2. Bracket and First Projection: $f \approx \pi^{\mathbb{B}\times\mathbb{C}}_{\mathbb{B}} \langle f, g \rangle$
3. Bracket and Second Projection: $g \approx \pi^{\mathbb{B}\times\mathbb{C}}_{\mathbb{C}} \langle f, g \rangle$
4. Recursion and Null Function: $(f \sharp g) \ddot{\circ} n \approx f$

After setting these trees equivalent, there exists the following quotient graph and graph morphism.

$$\begin{tikzcd}\mathbf{PRdesc} \arrow[r, two heads] & \mathbf{PRdesc}/(\mathbf{I})\end{tikzcd}$$

In detail, **PRdesc**/(**I**) has the same vertices as **PRdesc**, namely powers of natural numbers. The edges are equivalence classes of edges of **PRdesc**.

Descriptions of primitive recursive functions which are equivalent to "pruned" descriptions by relations of type **(I)** we shall call "stupid programs". They are descriptions that are wasteful in the sense that part of their tree is dedicated to describing a certain function and that function is not needed. The part of the tree that describes the unneeded function can be lopped off. One might call **PRdesc**/(**I**) the graph of "intelligent programs" since within this graph every "stupid program" is equivalent to another program without the wastefulness.

We can further quotient **PRdesc**/(**I**) by relations of type **(II)**:

1. Composition Is Associative: $f \circ (g \circ h) \approx (f \circ g) \circ h$.
2. Projections Are Identities: $f \circ \pi^{\mathbb{A}}_{\mathbb{A}} \approx f \approx \pi^{\mathbb{B}}_{\mathbb{B}} \circ f$.
3. Composition Distributes Over Bracket: $\langle f_1, f_2 \rangle \circ g \approx \langle f_1 \circ g, f_2 \circ g \rangle$.
4. Bracket Is Associative: $\langle f, \langle g, h \rangle\rangle \approx \langle\langle f, g \rangle, h \rangle$.
5. Bracket Is Almost Commutative: $\langle f, g \rangle \approx tw \circ \langle g, f \rangle$.
6. Twist Is Idempotent: $tw \circ tw = \pi$.
7. Reidermeister III:

   $$(tw_{\mathbb{B},\mathbb{C}} \times \pi_{\mathbb{A}}) \circ (\pi_{\mathbb{B}} \times tw_{\mathbb{A},\mathbb{C}}) \circ (tw_{\mathbb{A},\mathbb{B}} \times \pi_{\mathbb{C}}) \approx (\pi_{\mathbb{C}} \times tw_{\mathbb{A},\mathbb{B}}) \circ (tw_{\mathbb{A},\mathbb{C}} \times \pi_{\mathbb{B}}) \circ (\pi_{\mathbb{A}} \times tw_{\mathbb{B},\mathbb{C}}).$$

8. Recursion and Bracket: $\langle f_1, f_2 \rangle \sharp (g_1 \boxtimes g_2) \approx \langle f_1 \sharp g_1, f_2 \sharp g_2 \rangle$.
9. Recursion and Composition: $g_1 \ddot{\circ} (f \sharp (g_2 \ddot{\circ} g_1)) \approx (g_1 \ddot{\circ} f) \sharp (g_1 \ddot{\circ} g_2)$.
10. Recursion and Successor Function: $(f \sharp g) \ddot{\circ} s \approx g \ddot{\circ} (f \sharp g)$.

There is a further projection onto the quotient graph:

$$\mathbf{PRdesc} \twoheadrightarrow \mathbf{PRdesc}/(\mathbf{I}) \twoheadrightarrow \mathbf{PRalg} = (\mathbf{PRdesc}/\mathbf{I})/\mathbf{II} = \mathbf{PRdesc}/((\mathbf{I}) \bigcup (\mathbf{II})).$$

PRalg, or primitive recursive algorithms, are the main object of interest in this Section.

What does **PRalg** look like? Again the objects are the same as **PRdesc**, namely powers of natural numbers. The edges are equivalence classes of edges of **PRdesc**.

What type of structure does it have? In **PRalg**, for any three composable arrows, we have

$$f \circ (g \circ h) = (f \circ g) \circ h$$

and for any arrow $f : \mathbb{A} \longrightarrow \mathbb{B}$ we have

$$f \circ \pi_{\mathbb{A}}^{\mathbb{A}} = f = \pi_{\mathbb{B}}^{\mathbb{B}} \circ f.$$

That means that composition is associative and that the π's act as identities. Whereas **PRdesc** was only a graph with a composition, **PRalg** is a genuine category.

PRalg has a strictly associative product. On objects, the product structure is obvious:

$$\mathbb{N}^m \times \mathbb{N}^n = \mathbb{N}^{m+n}.$$

On morphisms, the product $\times$ was defined using the bracket above. The π are the projections of the product. In **PRalg** the twist map is idempotent and coherent. The fact that the product respects the composition is expressed with the interchange rule.

The category **PRalg** is closed under recursion. In other words, for any $f : \mathbb{A} \longrightarrow \mathbb{B}$ and any $g : \mathbb{A} \times \mathbb{B} \longrightarrow \mathbb{B}$, there exists a unique $h : \mathbb{A} \times \mathbb{N} \longrightarrow \mathbb{B}$ defined by recursion. The categorical way of saying that a category is closed under recursion, is to say that the category contains a natural number object. The simplest definition of a natural number object in a category is a diagram

$$\begin{tikzcd} * \arrow[r, "0"] & \mathbb{N} \arrow[r, "s"] & \mathbb{N} \end{tikzcd}$$

such that for any $k \in \mathbb{N}$ and $g : \mathbb{N} \longrightarrow \mathbb{N}$, there exists a unique $h : \mathbb{N} \longrightarrow \mathbb{N}$ such that the following diagram commutes:

$$\begin{tikzcd} * \arrow[r, "0"] \arrow[dr, "k"'] & \mathbb{N} \arrow[r, "s"] \arrow[d, "h"'] & \mathbb{N} \arrow[d, "h"] \\ & \mathbb{N} \arrow[r, "g"'] & \mathbb{N}. \end{tikzcd}$$

(see e.g. [1, 2, 9]). Saying that the above diagram commutes is the same as saying that h is defined by the simplest recursion scheme.

For our more general version of recursion, we require for every $f : \mathbb{A} \longrightarrow \mathbb{B}$ and $g : \mathbb{A}\times\mathbb{B} \longrightarrow \mathbb{B}$ there exists a unique $h : \mathbb{A}\times\mathbb{N} \longrightarrow \mathbb{B}$ such that the following two squares commute:

$$\begin{tikzcd}
\mathbb{A}\times * \arrow[r, "\pi\times 0"] \arrow[d, "\wr"'] & \mathbb{A}\times\mathbb{N} \arrow[d, "h"] & & \mathbb{A}\times\mathbb{N} \arrow[r, "\pi\times s"] \arrow[d, "\langle\pi_{\mathbb{A}}^{\mathbb{A}\times\mathbb{N}}{,}h\rangle"'] & \mathbb{A}\times\mathbb{N} \arrow[d, "h"] \\
\mathbb{A} \arrow[r, "f"'] & \mathbb{B} & & \mathbb{A}\times\mathbb{B} \arrow[r, "g"'] & \mathbb{B}.
\end{tikzcd}$$

This is sometimes called a natural number object with parameters.

From the fact that in *PRalg*, we have an object $\mathbb{N}$ and the morphisms $0 : * \longrightarrow \mathbb{N}$ and $s : \mathbb{N} \longrightarrow \mathbb{N}$ and these maps satisfy

$$h\text{ö}n = (f\sharp g)\text{ö}n = f$$

and

$$h\text{ö}s = (f\sharp g)\text{ö}s = g\text{ö}(f\sharp g) = g\text{ö}h$$

we see that **PRalg** has a natural number object.

We must show that in **PRalg**, the natural number object respects the bracket operation. This fundamentally says that the central square in the following two diagrams commute.

$$\begin{tikzcd}
& \mathbb{A}\times * \arrow[rrr, "\pi\times 0"] \arrow[dr, "\wr"] \arrow[dd, "\wr"'] \arrow[dddl, "\wr"'] & & & \mathbb{A}\times\mathbb{N} \arrow[dl, "h_1"'] \arrow[dd, "\langle h_1{,}h_2\rangle"] \arrow[dddr, "h_2"] & \\
& & \mathbb{A} \arrow[r, "f_1"'] & \mathbb{B} & & \\
& \mathbb{A} \arrow[ur, Rightarrow, no head] \arrow[dl, Rightarrow, no head] \arrow[rrr, "\langle f_1{,}f_2\rangle"'] & & & \mathbb{B}\times\mathbb{B} \arrow[ul, "\pi"] \arrow[dr, "\pi"'] & \\
\mathbb{A} \arrow[rrrrr, "f_2"'] & & & & & \mathbb{B}
\end{tikzcd}$$

The left hand triangles commute from the fact that $*$ is a terminal object. The right hand triangles commute because the equivalence relation forced the projections to respect the bracket. The inner and outer quadrilateral are assumed to commute. We conclude that the central square commutes.

$$\begin{tikzcd}
 & \mathbb{A}\times\mathbb{N} \arrow[rrrr, "\pi\times s"] \arrow[dd, "\wr"] \arrow[dr, "\wr"] \arrow[dddl, "\wr"] & & & & \mathbb{A}\times\mathbb{N} \arrow[dl, "h_1"] \arrow[dd, "\langle h_1,h_2\rangle"] \arrow[dddr, "h_2"] & \\
 & & \mathbb{A}\times\mathbb{B} \arrow[rr, "g_1"'] & & \mathbb{B} & & \\
 & \mathbb{A}\times\mathbb{B}\times\mathbb{B} \arrow[ur, "\pi"] \arrow[rrrr, "g_1\boxtimes g_2"'] \arrow[dl, "\pi"] & & & & \mathbb{B}\times\mathbb{B} \arrow[ul, "\pi"] \arrow[dr, "\pi"] & \\
\mathbb{A}\times\mathbb{B} \arrow[rrrrrr, "g_2"'] & & & & & & \mathbb{B}
\end{tikzcd}$$

Similarly, the left and the right triangles commute because the projections act as they are supposed to. The inner and outer quadrilateral commute out of assumption. We conclude that central square commutes.

We also must show that the natural number object respects the composition of morphisms. In $\sharp$ notation this amounts to

$$g_1\ddot{o}(f\sharp(g_2\ddot{o}g_1)) = (g_1\ddot{o}f)\sharp(g_1\ddot{o}g_2).$$

For the simpler form of recursion, this reduces to

$$g_1\circ(k\sharp(g_2\circ g_1)) = (g_1\circ k)\sharp(g_1\circ g_2).$$

Setting $h = k\sharp(g_2\circ g_1)$ and $h' = (g_1\circ k)\sharp(g_1\circ g_2)$, we get the following natural number object diagram

$$\begin{tikzcd}
* \arrow[rr, "0"] \arrow[dr, "k"] & & \mathbb{N} \arrow[rrr, "s"] \arrow[dl, "h"'] \arrow[dr, "h'"] & & & \mathbb{N} \arrow[dl, "h"'] \arrow[dr, "h'"] & \\
 & \mathbb{B} \arrow[rr, "g_1"'] & & \mathbb{B} \arrow[r, "g_2"'] & \mathbb{B} \arrow[rr, "g_1"'] & & \mathbb{B}.
\end{tikzcd}$$

From the uniqueness of h and h' we get that the triangles commute.

Once we have **PRalg**, we might ask when do two algorithms perform the same operation. We make an equivalence relation and say two algorithms are equivalent ($\approx'$) iff they perform the same operation. By taking a further quotient of **PRalg** we get **PRfunc**. What does **PRfunc** look like. The objects are again powers of natural numbers and the morphisms are primitive recursive functions.

In summary, we have the following diagram.

$$\begin{tikzcd}
\mathbf{PRdesc} \arrow[d, two heads] \\
\mathbf{PRdesc}/(\mathbf{I}) \arrow[d, two heads] \\
\mathbf{PRalg} = \mathbf{PRdesc}/((\mathbf{I})\bigcup(\mathbf{II})) \arrow[d, two heads] \\
\mathbf{PRfunc} = \mathbf{PRalg}/\approx' .
\end{tikzcd}$$

Let us spend a few moments discussing some category theory. There is the category **Cat** of all (small) categories and functors between them. Consider also the category **CatXN**. The objects are triples, $(\mathbb{C}, \times, N)$ where $\mathbb{C}$ is a (small) category, $\times$ is a strict product on $\mathbb{C}$ and N is a natural number object in $\mathbb{C}$. The morphisms of **CatXN** are functors $F : (\mathbb{C}, \times, N) \longrightarrow (\mathbb{C}', \times', N')$ that respect the product and natural number object. For $F : \mathbb{C} \longrightarrow \mathbb{C}'$ to respect the product, we mean that

$$\text{For all } f, g \in \mathbb{C} \quad F(f \times g) = F(f) \times' F(g).$$

To say that F respects the natural number object means that if

$$\begin{tikzcd}
* \arrow[r, "0"] & N \arrow[r, "s"] & N
\end{tikzcd}$$

is a natural number object in $\mathbb{C}$ and

$$\begin{tikzcd}
*' \arrow[r, "0'"] & N' \arrow[r, "s'"] & N'
\end{tikzcd}$$

is a natural number object in $\mathbb{C}'$ then $F(N) = N', F(*) = *', F(0) = 0'$ and $F(s) = s'$. For a given natural number object in a category, there is an implied function $\sharp$ that takes two morphisms f and g of the appropriate arity and outputs the unique $h = f \sharp g$ of the appropriate arity. Our definition of a morphism between two objects in **CatXN** implies that

$$\text{For all appropriate } f, g \in \mathbb{C} \quad F(f \sharp g) = F(f) \sharp' F(g).$$

There is an obvious forgetful functor $U : \mathbf{CatXN} \longrightarrow \mathbf{Cat}$ that takes $(\mathbb{C}, \times, N)$ to $\mathbb{C}$. There exists a left adjoint to this forgetful functor:

$$\begin{tikzcd} \mathbf{Cat} \arrow[r, bend left, "L"] \arrow[r, phantom, "\perp"] & \mathbf{CatXN} \arrow[l, bend left, "U"] \end{tikzcd}.$$

This adjunction means that for all small categories $\mathbb{C} \in \mathbf{Cat}$ and $\mathbb{D} \in \mathbf{CatXN}$ there is an isomorphism

$$\mathbf{CatXN}(L(\mathbb{C}), \mathbb{D}) \simeq \mathbf{Cat}(\mathbb{C}, U(\mathbb{D})).$$

Taking $\mathbb{C}$ to be the empty category $\emptyset$ we have

$$\mathbf{CatXN}(L(\emptyset), \mathbb{D}) \simeq \mathbf{Cat}(\emptyset, U(\mathbb{D})).$$

Since $\emptyset$ is the initial object in **Cat**, the right set has only one object. In other words $L(\emptyset)$ is a free category with product and a natural number object and it is the initial object in the category **CatXN**.

We claim that $L(\emptyset)$ is none other then our category **PRalg**.

Theorem 1 **PRalg** *is a free initial object in the category of categories with a strict product and a natural number object.*

We have already shown that **PRalg** is a category with a strict product and a natural number object. It remains to be shown that for any object $(\mathbb{D}, \times, N') \in \mathbf{CatXN}$ there is a unique functor $F_{\mathbb{D}} : \mathbf{PRalg} \longrightarrow \mathbb{D}$. Our task is already done by recalling that the objects and morphisms in **PRalg** are all generated by the natural number object and that functors in **CatXN** must preserve this structure. In detail, $F_{\mathbb{D}}(\mathbb{N}) = N'$ and since $F_{\mathbb{D}}$ must preserve products $F_{\mathbb{D}}(\mathbb{N}^i) = (N')^i$. And similarly for the morphisms of **PRalg**. The morphisms are generated by the πs, the n and s in the natural number object of **PRalg**. They are generated by composition, product and recursion. $F_{\mathbb{D}}$ is a functor and so it preserves composition. We furthermore assume it preserves product and recursion. $(\mathbb{D}, \times, N') \in \mathbf{CatXN}$ might have many more objects and morphisms but that is not our concern here. **PRalg** has very few morphisms.

The point of this theorem is that **PRalg** is not simply a nice category where all algorithms live. Rather it is a category with much structure. The structure tells us how algorithms are built out of each other. **PRalg** by itself is not very interesting. It is only its extra structure that demonstrates the importance of this theorem. **PRalg** is not simply the category made of algorithms, rather, it is the category that makes up algorithms.

PRfunc is the smallest category with a strict product and a natural number object. However, it is important to realize that **PRfunc** is not free. One function can be constructed in two totally different ways. The result of these two different constructions will be the same function. This is in contrast to **PRalg**, where two different constructions yield two different algorithms.

Before we go on to other topics, it might be helpful to, literally, step away from the trees and look at the entire forest. What did we do here? The graph **PRdesc** has operations. Given edges of the appropriate arity, we can compose them, bracket them or do recursion on them. But these operations do not have much structure. **PRdesc** is not even a category. By placing equivalence relations on **PRdesc**, which are basically coherence relations, we are giving the quotient category better and more amenable structure. So coherence theory, sometimes called higher-dimensional algebra, tells us when two programs are essentially the same.

5 Complexity Results

An algorithm is not *one* arrow in the category **PRalg**. An algorithm is a scheme of arrows, one for every input size. We need a way of choosing each of these arrows.

There are many different species of algorithms. There are algorithms that accept n numbers and output one number. A scheme for such an algorithm might look like this:

$$\begin{tikzcd}
 & & \mathbb{N}^1 \arrow[dd, "c_1"] & & \\
 & & & \mathbb{N}^2 \arrow[dl, "c_2"] & \\
\vdots \arrow[rr, "c"] & & \mathbb{N} & & \mathbb{N}^3 \arrow[ll, "c_3"'] \\
 & \mathbb{N}^k \arrow[ur, "c_k"] & & \mathbb{N}^4 \arrow[ul, "c_4"'] & \\
 & & \cdots \arrow[uu, "c"] & &
\end{tikzcd}$$

We shall call such a graph a *star graph* and denote it $\bigstar$.

However there are other species of algorithms. There are algorithms that accept n numbers and output n numbers (like sorting or reversing a list, etc.)

Such a scheme looks like

$$\begin{tikzcd}
\mathbb{N}^1 \arrow[r, "c_1"] & \mathbb{N}^1 \\
\mathbb{N}^2 \arrow[r, "c_2"] & \mathbb{N}^2 \\
\mathbb{N}^3 \arrow[r, "c_3"] & \mathbb{N}^3 \\
\vdots & \\
\mathbb{N}^k \arrow[r, "c_k"] & \mathbb{N}^k \\
\vdots &
\end{tikzcd}$$

We shall also call such a graph a star graph.

One can think of many other possibilities. For example, algorithms that accept n numbers and outputs their max, average and minimum (or mean, median and mode) outputs three numbers. We shall not be particular as to what what type of star graph we will be working with.

Given any star graph $\bigstar$, a scheme that chooses one primitive recursive description for each input is a graph homomorphism $Sch : \bigstar \longrightarrow \mathbf{PRdesc}$ that is the identity on vertices. That is $Sch(\mathbb{N}^i) = \mathbb{N}^i$ for all $i \in \mathbb{N}$.

Composing $Sch : \bigstar \longrightarrow \mathbf{PRdesc}$ with the projection onto the equivalence classes $\mathbf{PRdesc} \longrightarrow \mathbf{PRdesc}/(\mathbf{I})$ gives a graph homomorphism $\bigstar \longrightarrow \mathbf{PRdesc}/(\mathbf{I})$. In order not to have too many names flying around, we shall also call this graph homomorphism Sch. Continuing to compose with the projections, we get the following commutative diagram.

$$\begin{tikzcd}
& \mathbf{PRdesc} \arrow[d, two heads] \\
& \mathbf{PRdesc}/(\mathbf{I}) \arrow[dd, two heads] \\
\bigstar \arrow[uur, "Sch"] \arrow[ur, "Sch"] \arrow[dr, "Sch"] \arrow[ddr, "Sch"] & \\
& \mathbf{PRalg} \arrow[d, two heads] \\
& \mathbf{PRfunc}.
\end{tikzcd}$$

We are not interested in only one graph homomorphism $\bigstar \longrightarrow$ **PRdesc**. Rather we are interested in the set of all graph homomorphisms. We shall call this set $\mathbf{PRdesc}^{\bigstar}$. Similarly, we shall look at the set of all graph homomorphisms from $\bigstar$ to **PRdesc**/(**I**), which we shall denote $(\mathbf{PRdesc}/(\mathbf{I}))^{\bigstar}$. There is also $\mathbf{PRalg}^{\bigstar}$ and $\mathbf{PRfunc}^{\bigstar}$. There are also obvious projections:

$$\mathbf{PRdesc}^{\bigstar} \twoheadrightarrow (\mathbf{PRdesc}/(\mathbf{I}))^{\bigstar} \twoheadrightarrow \mathbf{PRalg}^{\bigstar} \twoheadrightarrow \mathbf{PRfunc}^{\bigstar}$$

Perhaps it is time to get down from the abstract highland and give two examples. We shall present mergesort and insertion sort as primitive recursive algorithms. They are two different members of $\mathbf{PRalg}^{\bigstar}$. These two different algorithms perform the same function in $\mathbf{PRfunc}^{\bigstar}$.

Example: Mergesort depends on an algorithm that merges two sorted lists into one sorted list. We define an algorithm $Merge$ that accepts m numbers of the first list and n numbers of the second list. $Merge$ inputs and outputs $m+n$ numbers.

$$Merge_{0,1}(x_1) = Merge_{1,0}(x_1) = \pi_1^1(x_1) = x_1$$

$$Merge_{m,n}(x_1, x_2, \ldots, x_m, x_{m+1}, \ldots, x_{m+n}) =$$

$$\begin{cases} (Merge_{m,n-1}(x_1, x_2, \ldots, x_m, x_{m+1}, \ldots, x_{m+n-1}), x_n) & : \quad x_m \le x_n \\ (Merge_{m-1,n}(x_1, x_2, \ldots, x_{m-1}, x_{m+1}, \ldots, x_{m+n}), x_m) & : \quad x_m > x_n \end{cases}$$

With $Merge$ defined, we go on to define $MergeSort$. $MergeSort$ recursively splits the list into two parts, sorts each part and then merges them.

$$MergeSort_1(x) = \pi^{\mathbb{N}}_{\mathbb{N}}(x) = x$$

$$MergeSort_k(x_1, x_2, \ldots, x_k) =$$

$$Merge_{\llcorner k/2 \lrcorner, \ulcorner k/2 \urcorner}(MergeSort_{\llcorner k/2 \lrcorner}(x_1, x_2, \ldots, x_{\llcorner k/2 \lrcorner}), MergeSort_{\ulcorner k/2 \urcorner}(x_{\llcorner k/2 \lrcorner+1}, x_{\llcorner k/2 \lrcorner+2}, \ldots, x_k)$$

We might write this in short as

$$MergeSort = Merge \circ \langle MergeSort, MergeSort \rangle$$

□

Example: Insertion sort uses an algorithm $Insert : \mathbb{N}^k \times \mathbb{N} \longrightarrow \mathbb{N}^{k+1}$ which takes an ordered list of k numbers adds a $k + 1$th number to that list in its correct position. In detail,

$$Insert_0(x) = \pi^1_1(x) = x$$

$$Insert_k(x_1, x_2, \ldots, x_k, x) =$$

$$\begin{cases} (x_1, x_2, \ldots, x_k, x) & : \quad x_k \leq x \\ (Insert_{k-1}(x_1, x_2, \ldots, x_{k-1}, x), x_k) & : \quad x_k > x \end{cases}$$

The top case is the function $\pi^k_k \times \pi^1_1$ and the bottom case is the function $(Insert_{k-1} \times \pi) \circ (\pi^{k-1}_{k-1} \times tw_{\mathbb{N},\mathbb{N}})$. With $Insert$ defined, we go on to define $InsertionSort$.

$$InsertionSort_1(x) = \pi^{\mathbb{N}}_{\mathbb{N}}(x) = x$$

$$InsertionSort_k(x_1, x_2, \ldots, x_k) = Insert_{k-1}(InsertionSort_{k-1}(x_1, x_2, \ldots, x_{k-1}), x_k)$$

We might write this in short as

$$InsertionSort = Insert(InsertionSort \times \pi)$$

□

The point of the these examples, is to show that although these two algorithms perform the same function, they are clearly very different algorithms. Therefore one can not say that they are "essentially" the same.

Now that we have placed the objects of study in order, let us classify them via complexity theory. The only operations in our trees that are of any complexity is the recursions. Furthermore, the recursions are only interesting if they are nested within each other. So for a given tree that represents a description of a primitive recursive function, we might ask what is the largest number of nested recursions in this tree. In other words, we are interested in the largest number of "R" labels on a path from the root to a leaf of the tree. Let us call this the *Rdepth* of the tree.

Formally, *Rdepth* is given recursively on the set of our labeled binary trees. The *Rdepth* of a one element tree is 0. The *Rdepth* of an arbitrary tree T is

$$Rdepth(T) = Max\,\{Rdepth(left(T)), Rdepth(right(T))\} + (label(T) == \boxed{\mathbf{R}})$$

where $(label(T) == \boxed{\mathbf{R}}) = 1$ if the label of the root of T is $\boxed{\mathbf{R}}$, otherwise it is 0.

It is known that a primitive recursive function that can be expressed by a tree with *Rdepth* of n or less is an element of Grzegorczyk's hierarchy class $\mathcal{E}^{n+1}$. (See [4], Theorem 3.31 for sources.)

Complexity theory deals with the partial order of all functions $\{f | f : \mathbb{N} \longrightarrow \mathbb{R}^+\}$ where

$$f \leq g \text{ iff } Lim_{n\to\infty}\frac{f(n)}{g(n)} < \infty.$$

For every algorithm we can associate a function that describes the *Rdepth* of the trees used in that algorithm. Formally, for a given algorithm, $A : \bigstar \longrightarrow$ **PRdesc**, we can associate a function $f_A : \mathbb{N} \longrightarrow \mathbb{R}^+$ where

$$f_A(n) = Rdepth(A(c_n))$$

when c_n is an edge in $\bigstar$. The function $\mathbf{PRdesc}^{\bigstar} \longrightarrow \{f | f : \mathbb{N} \longrightarrow \mathbb{R}^+\}$ where $A \mapsto f_A$ shall be called $Rdepth_0$.

We may extend $Rdepth_0$ to

$$Rdepth_1 : (\mathbf{PRdesc}/(\mathbf{I}))^{\bigstar} \longrightarrow \{f | f : \mathbb{N} \longrightarrow \mathbb{R}^+\}.$$

For a scheme of algorithms $[A] : \bigstar \longrightarrow (\mathbf{PRdesc}/(\mathbf{I}))$ we define

$$f_{[A]}(n) = Min_{A'}\{Rdepth(A'(c_n))\}$$

where the minimization is over all descriptions A' in the equivalence class $[A]$. (For the categorical cognoscenti, $Rdepth_1$ is a right Kan extension of $Rdepth_0$ along the projection $\mathbf{PRdesc}^{\bigstar} \longrightarrow (\mathbf{PRdesc}/(\mathbf{I}))^{\bigstar}$.

$Rdepth_1$ can easily be extended to

$$Rdepth_2 : \mathbf{PRalg}^{\bigstar} \longrightarrow \{f | f : \mathbb{N} \longrightarrow \mathbb{R}^+\}.$$

The following theorem will show us that we do not have to take a minimum over an entire equivalence class.

Theorem 2 *Equivalence relations of type* **(II)** *respect Rdepth.*

Proof. Examine all the trees that express these relations throughout this paper. Notice that if two trees are equivalent, then their *Rdepth*s are equal. □

$Rdepth_2$ can be extended to

$$Rdepth_3 : \mathbf{PRfunc}^\bigstar \longrightarrow \{f|f : \mathbb{N} \longrightarrow \mathbb{R}^+\}.$$

We do this again with a minimization over the entire equivalence class (i.e. a Kan extension.)

And so we have the following (not necessarily commutative) diagram.

$$\begin{tikzcd}
\mathbf{PRdesc}^\bigstar \arrow[d, two heads] \arrow[ddr, "Rdepth_0"] & \\
(\mathbf{PRdesc}/(\mathbf{I}))^\bigstar \arrow[dd, two heads] \arrow[dr, "Rdepth_1"] & \\
& \{f|f : \mathbb{N} \longrightarrow \mathbb{R}^+\} \\
\mathbf{PRalg}^\bigstar \arrow[d, two heads] \arrow[ur, "Rdepth_2"] & \\
\mathbf{PRfunc}^\bigstar \arrow[uur, "Rdepth_3"'] &
\end{tikzcd}$$

Corollary 1 *The center triangle of the above diagram commutes.*

This is in contrast to the other two triangles which do not commute.

In order to see why the bottom triangle does not commute, consider an inefficient sorting algorithm. $Rdepth_2$ will take this inefficient algorithm to a large function $\mathbb{N} \longrightarrow \mathbb{R}^+$. However, there are efficient sorting algorithms and $Rdepth_3$ will associate a smaller function to the primitive recursive function of sorting.

There are many subclasses of $\{f|f : \mathbb{N} \longrightarrow \mathbb{R}^+\}$ like polynomials or exponential functions. Complexity theory studies the preimage of these subclasses under the function $Rdepth_3$. The partial order in $\{f|f : \mathbb{N} \longrightarrow \mathbb{R}^+\}$ induces a partial order of subclasses of **PRfunc**.

6 Future Directions

We are in no way finished with this work and there are many directions that it can be extended.

Extend to all Computable Functions. The most obvious project that we can pursue is extend this work from primitive recursive functions to all computable functions. In order to do this we must add the minimization operation. For a given $g : \mathbb{A} \times \mathbb{N} \longrightarrow \mathbb{N}$, there is an $h : \mathbb{A} \longrightarrow \mathbb{N}$ such that

$$h(x) = Min_n \{g(x, n) = 1\}$$

Categorically, this amounts to looking at the total order of $\mathbb{N}$. This induces an order on the set of all functions from $\mathbb{A}$ to $\mathbb{N}$. We then look at all functions h' that make this square commute.

$$\begin{tikzcd}
\mathbb{A} \arrow[r, "!"] \arrow[d, "\langle \pi^{\mathbb{A}}_{\mathbb{A}}, h' \rangle"'] & * \arrow[d, "1"] \\
\mathbb{A} \times \mathbb{N} \arrow[r, "g"] & \mathbb{N}
\end{tikzcd}$$

i.e.,

$$g(x, h'(x)) = 1.$$

Let $h : \mathbb{A} \longrightarrow \mathbb{N}$ be the minimum such function.

We might want to generalize this operation. Let $f : \mathbb{A} \longrightarrow \mathbb{B}$ and $g : \mathbb{A} \times \mathbb{N} \longrightarrow \mathbb{B}$, then we define $h : \mathbb{A} \longrightarrow \mathbb{N}$ to be the function

$$h(x) = Min_n \{g(x, n) = f(x)\}.$$

Categorically, this amounts to looking at all functions h' that make the triangle commute:

$$\begin{tikzcd}
& \mathbb{A} \arrow[dl, "\langle \pi^{\mathbb{A}}_{\mathbb{A}}, h' \rangle"'] \arrow[dr, "f"] & \\
\mathbb{A} \times \mathbb{N} \arrow[rr, "g"'] & & \mathbb{B}
\end{tikzcd}$$

i.e.,

$$g(x, h'(x)) = f(x).$$

Let $h : \mathbb{A} \longrightarrow \mathbb{N}$ be the minimum such function.

Hence minimization is a fourth fundamental operation:

$$\begin{tikzcd}[column sep=small]
& h : \mathbb{A} \longrightarrow \mathbb{N} & \\
& \boxed{\mathbf{M}} \arrow[dl, dash] \arrow[dr, dash] & \\
f : \mathbb{A} \longrightarrow \mathbb{B} & & g : \mathbb{A} \times \mathbb{N} \longrightarrow \mathbb{B}
\end{tikzcd}$$

There are several problems that are hard to deal with. First, we leave the domain of total functions and go into the troublesome area of partial functions.

All the relational axioms have to be reevaluated from this point of view. Second, what should we substitute for *Rdepth* as a complexity measure?

Other Types of Algorithms We have dealt with classical deterministic algorithms. Can we do the same things for other types of algorithms. For example, it would be nice to have universal properties of categories of non-deterministic algorithms, probabilistic algorithms, parallel algorithms, quantum algorithms, etc. In some sense, with the use of our bracket operation, we have already dealt with parallel algorithms.

More Relational Axioms. It would be interesting to look at other relations that tell when two programs are essentially the same. With each new relation, we will get different categories of algorithms and a projection from the old category of algorithms to the new one. With each new relation, one must find the universal properties of the category of algorithms.

Canonical Presentations of Algorithms. Looking at the equivalent trees, one might ask whether there a canonical presentation of an algorithm. Perhaps we can push up the recursions to the top of the tree, or perhaps push the brackets to the bottom. This would be most useful for program correctness and other areas of computer science.

In a sense, Kleene's Theorem on partial recursive functions is an example of a canonical presentation of an algorithm. It says that for every computable function, there exists at least one tree-like description of the function such that the root of the tree is the only minimization in the entire tree.

When are Two Programs Really Different Algorithms. Is there a way to tell when two programs are really different algorithms? There is a subbranch of homotopy theory called obstruction theory. Obstruction theory asks when are two topological spaces in different homotopy classes of spaces. Is there an obstruction theory of algorithms?

Other Universal Objects in CatXN. We only looked at one element of **CatXN** namely **PRalg**. But there are many other elements that are worthy of study. Given an arbitrary function $f : \mathbb{N} \longrightarrow \mathbb{N}$, consider the category $\mathbb{C}_f$

with $\mathbb{N}$ as its only object and f as its only non-trivial morphism. The free **CatXN** category over $\mathbb{C}_f$ is, we believe, the category of primitive recursive functions with oracle computations from f. It would be nice to frame relative computation theory and complexity theory from this perspective.

Proof Theory. There are many similarities between our work and the proof theory. Many times, one sees two proofs that are essentially the same proof. It would be nice to do the same thing for proofs. Gentzen type proofs are already set up like trees. The cut rule in proof theory is very similar to composition in a category. What are the other operations of proofs? We would be very interested in looking at the universal properties of the category of proofs. What is the relationship between the category of algorithms and the category of proofs?

A Language Independent Definition of Algorithms. Our definition of algorithm is dependent on the language of primitive recursive functions. We could have, no doubt, done the same thing for other languages. The intuitive notion of an algorithm is language independent. Can we find a definition of an algorithm that does not depend on any language?

Permit me to get a little "spacey" for a few lines. Consider the set of *all* programs in *all* programming languages. Call this set **Programs**. Partition this set by the different programming languages that make the programs. So there will be a subset of **Programs** called **Java**, a subset called **C++**, and a subset **PL/1** etc. There is also a subset called **Primitive Recursive** which will contain all the trees that we discussed in Section 3. There will be functions between these different subsets. We might call these functions (non-optimizing) *compilers.* They take as input a program from one programming language and output a program in another programming language. In some sense **Primitive Recursive** is initial for all the these sets. By initial we mean that there are compilers going out of it. There are few compilers going into it. The reason for this is that in C++ one can program the Ackerman function. One can not do this in **Primitive Recursive**. (There are, of course, weaker programming languages than primitive recursive functions, but we ignore them here.)

For each subset of programs, e.g. **Progs1**, there is a an equivalence relation $\approx_{\mathbf{Progs1}}$ or $\approx_1$ that tells when two programs in the subset are essentially the same. If C is a compiler from **Progs1** to **Progs2** then we demand that if two programs in **Progs1** are essentially the same, then the compiled versions of each of these programs will also be essentially the same, i.e., for any two programs P and P' in **Progs1**,

$$P \approx_1 P' \qquad \Rightarrow \qquad C(P) \approx_2 C(P').$$

We also demand that if there are two compilers, then the two compiled programs will be essentially the same,

$$\text{For all programs } P, \qquad C(P) \approx_2 C'(P).$$

Now place the following equivalence relation $\equiv$ on the set **Programs** of *all* programs. Two programs are equivalent if they are the in the same programming language and they are essentially the same, i.e.,

$$P \equiv P' \text{ if there exists a relation } \approx_i \text{ such that } P \approx_i P'$$

and two programs are equivalent if they are in different programming languages but there exists a compiler that takes one to the other,

$$P \equiv P' \text{ if there exists a compiler } C \text{ and } C(P) = P'.$$

We have now placed an equivalence relation on the set of all programs that tells when two programs are essentially the same. The equivalence classes of **Programs/**$\equiv$ are algorithms. This definition does not depend on any preferred programming languages. There is much work to do in order to formulate these ideas correctly. It would also be nice to list the properties of **Algorithms** $=$ **Programs/**$\equiv$.

References

[1] M. Barr and C. Wells. *Toposes, triples and theories.* Grundlehren der Mathematischen Wissenschaften, 278. Springer-Verlag, New York, (1985).

[2] M. Barr and C. Wells. *Category Theory for Computing Science.* Prentice Hall (1990).

[3] A. Blass, Y. Gurevich. “Algorithms: A Quest for Absolute Definitions.” Available on the web.

[4] P. Clote. Computational Models and Function Algebras. Handbook of Computability Theory.

[5] T.H. Corman, C.E. Leiserson, R.L. Rivest, C. Stein; *Introduction to Algorithms, Second Edition.* McGraw-Hill (2002).

[6] W. Dean. *What algorithms could not be.* 2006 Thesis in Department of Philosophy. Rutgers University.

[7] D.E. Knuth. *The Art of Computer Programing: Volume 1 / Fundamental Algorithms.* Third Edition. Addison-Wesley. 1997.

[8] D.E. Knuth. *Selected Papers on Computer Science.* Cambridge University Press. 1996.

[9] Saunders Mac Lane. *Categories for the Working Mathematician,* Second Edition. Springer, 1998.

[10] Y.N. Moschovakis. “What Is an Algorithm?” Available on his web page.

Department of Computer and Information Science
Brooklyn College, CUNY
Brooklyn, N.Y. 11210

Computer Science Department
The Graduate Center, CUNY
New York, N.Y. 10016

e-mail: noson@sci.brooklyn.cuny.edu