Recursion: Squeezing the Infinite into the Finite

Rusi P. Mody
Persistent Computing Institute, Pune
rustompmody@gmail.com

Anuradha Laxminarayan
The Magus, Pune
lanuradha@gmail.com

Jayant Kirtane
ApexPlus Technologies, Hyderabad
jayant.kirtane@gmail.com

***Abstract*—This Innovative Practice Full Paper presents an approach to the teaching of recursion as part of CS education. Recursion is a concept that is generally considered to be hard. Likewise, in the course of mathematics education, induction is problematic, but for the opposite reason: math students are often bewildered by what looks to them suspiciously like a circular definition. So where recursion is hard to comprehend, induction does not motivate. When we see that induction and recursion are two sides of the same coin, it becomes possible to address *math and programming as endpoints of one axis*, by solving the pedagogy of recursion and induction together.**

Furthermore, imperative programming (IP) and OOP lose clarity to efficiency, while functional programming (FP) is declarative at the cost of computational intuitions. This oppositional outlook results in a deleterious proliferation of paradigms. Since FP provides the denotational semantics of IP, while IP gives the implementation semantics of FP, it becomes feasible to integrate these by making language questions more reified in one's presentation. *Hence the second axis*: *FP – IP*.

We present recursion as a specific instance of our dual-axis, dual-paradigm approach to teaching programming that weaves together multiple facets of programming theory, foundations and pragmatics into a single narrative. While the entire material tends to increase considerably over the usual approach, especially as the material spans across several courses in the curriculum, it turns out considerably more satisfying to the teacher and kinder to the student this way than all other approaches we have tried.

Whereas the conventional view of programming pigeonholes recursion as one programming technique among several, we approach it as an ontology that permeates CS. From this vantage point, we have developed a teaching method based on a set of seven principles and their corresponding actionables. So for instance, teaching co-recursion before recursion is an action that emerges from the perspective that it is data that is essentially recursive, while recursive functions are just the computational mirror of such data.

We formulate these principles and the corresponding actionables interspersed with illustrative programming examples.

***Index Terms*—Computer Science Education, Curriculum Development, Programming, Functional Programming, Mathematics**

I. Introduction

Recursion pervades CS

Recursion is hard

For the ubiquity of recursion in CS see [1]. For the difficulty, there is wide literature recording the issues around the teaching of recursion.[2] and [3] have detailed surveys about different approaches. [4] describes specific difficulties in the context of OOP. [5], [6] have even tried using animation, songs and stories to motivate. [7] is an old classic that discusses different techniques at length. And the legendary SICP [8] focusses on recursion by using the functional style.

The question arises: Is recursion intrinsically hard? Is it hard to teach? Or do extraneous aspects of our languages make it difficult? One purpose of this paper is to show that teaching recursion can be made easy by combining the right tools with a matching philosophy.

After several years of teaching programming in general and recursion in particular, in different ways using different programming environments, we have reified a smooth path formulated as 7 principles. We present recursion as an ontology with these laws and their associated facts, actionables and caveats.

Section II gives the background:

- A recapitulation of the Platonic and empiric realms that help disentangle key concepts from the idiosyncrasies of mainstream languages.
- The desirable features of a tool for teaching core concepts
- A brief outline of our specially constructed tool

Section III is the centerpiece of the paper: 7 laws that significantly aid and streamline the teaching of recursion. Section IV dwells on the finite-infinite duality and how it inspires our approach to CS101 with the formula: start with equational modelling, follow by seguing into the programming mainstream. Finally Section V summarizes and wraps up.

II. Ontics

A. Platonism vs Empiricism

This is a difference in world-view going back to the divergence between Plato and Aristotle [9]:

- Platonism: Plato averred that there are certain ideas that are innate to human beings and therefore universal and hence in a very absolute sense **true**.
- Empiricism: Aristotle rejected the innate-idea idea and asserted that we start with a clean slate – *tabula rasa* – learning from experience.

For teachers this choice of outlook is piquant, and for CS teachers much more so. The problem is that math is *the* archetypal area where Plato clearly wins, in that math works with too much "unreasonable effectiveness" for it to be a mere concoction [10], [11], [12]. Insofar as CS is strongly related to math, CS must be Platonic. On the other hand, as teachers we must teach to the students' given *tabula rasa* and not to our preconceptions. Moreover, CS as an engineering/tech field has to stay grounded in empiric reality. Hence our challenge as CS teachers:

The Empiric delivery of Platonism

Let us take a closer look at what this perspective means. Loops as found in programming languages are an empiric fact. Induction in all its forms is a Platonic truth. We have a choice: Does recursion fall between these two stools? Or does it serve as the bridge that links the two worlds?

Towards affirming the latter, we begin with our tiny `pug`. `pug` is an interpreter that is the centerpiece of the larger *Pugofer*, a pedagogic ecosystem expressly constructed to bridge the empiric-platonic divide [13]. From here, we navigate to mainstream languages like Python, Java, etc. [14].

- To the math eye: `pug` is a Haskell-like PL, an empiric artifact, that models computation as functions over immutable values. Unlike ghc, it is tiny, focusing on what is essential to the beginner and eliding advanced material.
- To the programming eye: `pug` is a mathematical, Platonic universe of many shapes that not only serves as the underlying semantics for programming languages but even helps clarify mathematics itself. For the place of `pug` in the larger *Pugofer* ecosystem, see [15]. For the benefits of `pug` over classic Haskell see [16]

B. The centrality of ontologies

Teachers know that whatever the vehicle, example, or excuse may be, they ultimately need to **teach concepts.** We like to go a step further: Some concepts are "truer" than others; we call them *ontologies.* On the other hand, teaching can only ever proceed with specific examples. Hence the teacher needs to be ever cognizant of the ontologies behind the carrier examples until they become an integral part of the students' mentation. Effective teaching happens when this *ontological weight,* i.e., the number of carrier ontologies required for a new concept is minimized. Our emphasis on beginning with an equational style follows from the fact that reduced ontological weight makes teaching recursion easy. To illustrate with a classic C example for 'list-of-T':

```
struct listnode
{ T val;
  listptr next;
}
```

```
typedef struct
    listnode *listptr;
```

Note the **mutual** recursion: `listnode` contains a `listptr` and `listptr` points to `listnode`. So where between these two types does the actual list type lie? Its *ontology is lost* amid C's engineering considerations [17]. Inter alia, not just C, Java and C++ but even Haskell is burdened with minute distinctions, e.g., half a dozen integer types which buy engineering nuance at the cost of ontic confusion: mainstream languages throw a mostly irrelevant *paradox of choice* [18] at beginners. This is elaborated in the next section.

C. Recursion-Paradigm Misfit

To teach a programming concept or modality like recursion, a vehicle PL is of course a prerequisite. As we show below, almost every standard PL or class of PLs unfortunately satisfies Dijkstra's aphorism: *Our programming languages belong to the problem set, not the solution set.*

1) IP

Recursion through IP has a greatly increased ontological weight due to: (a) substitution vs sequence model [8], (b) pointers double the cognitive load of recursion, (c) `return` adds to it. (Is `return` imperative or functional?)

2) OOP

Not only do all the problems of IP remain in OOP, they get worse [19], [20].

3) Haskell

Viewed locally, Haskell and the like are good for recursion. However, there is much that is arcane in modern Haskell for a beginner.

- Just as one cannot use a `scanf` in C without pointers, one cannot do Haskell IO without monads, a very high price to pay.
- Type errors in the face of the "open world assumption" are intimidating for a beginner [21]. `pug` improves on Helium by initially dropping even type-classes completely.
- That recursion is natural and easy in FP does not imply it is necessary to use it, still less overuse it. [22] [Note: This is not a Haskell problem but FP culture at large.]

4) Scheme

Scheme is more kind to the beginner than the above. Yet, not having type inference, pattern matching and equation-syntax is a significant ontological deficit [23]. Also Scheme is strictly not considered an FPL since it allows assignment and mutation. This can have surprising effects on student expectation [24].

> *The most important decisions in language design concern what is to be left out* – Niklaus Wirth

D. Pedagogics

Necessary Thinking-Tools for Recursion

- A minimal notation (follows from Wirth above)
- Beginner friendly errors [25]
- Equationally rather than procedurally specified [26]
- Data structure mirrored by pattern-matching [27] in code
- Type discipline: (a) strong (b) static (c) simple (d) type inference – in short Hindley-Damas-Milner [28], [29], [30].
- Language levels tailored to the beginner's natural learning curve [31], [32].
- Interpreted rather than compiled language.

One of the big leaps of Haskell over its predecessors is type-classes [33]. While `pug` does have a Haskell-like standard prelude, beginners start with a simple prelude with **no type classes**. `pug` was intentionally constructed with all the above in mind. Many of its features are Haskell-like but they constitute a tiny subset of Haskell, taking hardly a couple of seconds to compile the entire C-sources. So, its lineage is conventional FP but its focus is pedagogical.

Two-Pass Algorithm

ACM CS-2013 curricular guidelines for teaching programming [34] clarify: (a) focus on the key concepts initially, (b) identify the paradigm that best highlights those concepts and elides irrelevant details, and (c) choose the language that is the best vehicle optimizing the above and makes the ramp-up easy. This leads to our two-pass algorithm:

- Pass 1 (Platonic): Choose the most suitable notation for the relevant concept.
- Pass 2 (Empiric): Show how to convert to a mainstream language.

While having two passes looks more work than one, 2 passes turn out to be not just effective but even time-efficient for teaching. (See "Rob Hagan principle" [35].) `pug` in Pass 1 and IP/OO in Pass 2 naturally satisfies the constraints of teaching-learning not just recursion but also CS in general.

E. Notes on Notation

The Dijkstra Dot: Function-call notation

Dijkstra and his school use `f.x` where functional programmers use `f x` and traditional mathematicians use $f(x)$. This seemingly idiosyncratic syntax has many advantages as explained by Dijkstra [36], [37]. In our context there are additional advantages [38], [16], [39] but these are outside the scope of this recursion-centered discussion. We adopt the notation for function definition from Dijkstra as: `f.x =x +1` and once this is available in a file, say `file.gs`, it can be loaded into `pug` and we can evaluate `f` just like other expressions at the interpreter prompt as below:

```
? 1+2
3 : Int
? f.3
4 : Int
?f
f : Int → Int
```

ADTs are concrete!

The acronym ADT in the OOP context means *abstract* data type, whereas in the FP context it means *algebraic* datatype [40] whose hallmark feature is that **values are concrete.**

The classic `data` definition in Haskell creates new concrete types but is confusing whereas the newer format called GADT [41] is considered advanced.

It is central to our methodology that the GADT style is the only one used. Type definitions are heralded with the keyword `ctype` (concrete type).

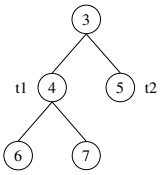
As an example, we define a binary tree type containing leaf and branch nodes with constructors `Lf` and `Br` as in Fig 1. It becomes clear from such a definition that the final target type `BinTree` is the type being constructed and recursion means there are other instances of the same type in the type signature of at least one of its constructors.

Interactions

As we have already seen, users interact with the system by typing an expression at the **?** prompt. The system responds with a result (as any interpreter does), and also its type. Pedagogically, this is significant: types serve like a beginner cyclist's guide wheels. The user almost never gives a type. The system invariably responds with a type as if to say: Your answer is this, and its type is this. Fig 2 has a bigger example of interactions for the `BinTree` type described in Fig 1. In both this paper and our classes, the notation becomes clearer through use. Our classes proceed with students experimenting with the interpreter and validating our assertions. Through such animate sessions one can teach programm**ing** instead of presenting prepared programs.

```
ctype BinTree where
  Lf: Int → BinTree
  Br: Int → BinTree → BinTree → BinTree
```

Fig. 1. A concrete type for a binary tree




```
? t2
Lf.5: BinTree
? t1
Br.4.(Lf.6).(Lf.7): BinTree
? Br.3.t1.t2
Br.3.(Br.4.(Lf.6).(Lf.7)).(Lf.5): BinTree
```

Fig. 2. Concrete Bintree values in pug

III. Ontological Principles and Imperatives

In keeping with the above, CS teachers need to spiral between the Platonic ideal world and the empiric real world. So we talk of:

Category	Realm	Symbol
Principles	Platonic	⊙
Facts	Empiric	⊡
Actions	Bridge between Principles and Facts	☞
Caveats	Challenges	⚠

These together serve both for an efficient transmission of core concepts, thinking habits and an inculcation of wholesome programming practices.

⊙ **Law 1.** ***Data before code***

⊙ **Law 2.** ***Types precede values***

⊙ **Law 3.** ***It is data that is recursive, code only follows***

⊙ **Law 4.** ***The infinite precedes the finite***

⊙ **Law 5.** ***"Recursion" after Recursion***

⊙ **Law 6.** ***Math before programming***

⊙ **Law 7.** ***One example, many concepts***

In this section, we discuss these principles, their contextual facts, and imperatives that arise from this combination.

⊙ **Law 1.** ***Data before code***

⊡ **Fact 1.** Students have already seen sequences as part of college education; only, these are not recursive but flat structures.

So we begin where the students are. An important pedagogical step is to reduce unlearning of previously formed concepts until the right time. Sequences as flat lists in the form of list comprehensions hit that sweet middle spot where even the notation is the familiar *set-builder* which they have seen before.

The takeaway from this fact is:

☞**Action.** Teach list comprehensions before recursion

Understanding list laws through list comprehensions (LCs) minimizes the conceptual addition to the recursive formulation later on.

So we scale up our data and play with lists with as much ease as with numbers, and demonstrate list algebras with `pug` as a *calculator*.

```
? [x*x | x ∈ [1...5]]
[1, 4, 9, 16, 25] : [Int]
```

We learn to read this as: *gather x for x drawn from* `[1...5]` Note the subtle movement from the set notion $x \in \{1...5\}$, so that the importance of order is implicitly understood. A few examples:

```
--Listifying every element of a given list
? [[x] | x ∈ [1...5]]
[[1], [2], [3], [4], [5]] : [[Int]]

--For a list defined as s = [[1],[2,3,4],[5,6]]
--Flattening s (concat in Haskell)
? [y | x ∈ s, y ∈ x]
[1, 2, 3, 4, 5, 6] : [Int]

--We can repeat each i, i times as:
? [[x | y ∈ [1...x]] | x ∈ [1...4]]
[[1], [2,2], [3,3,3], [4,4,4,4]] : [[Int]]
```

Notice how much functionality has been created without a single function definition.

After experimenting with examples in `pug`, the more verbose comprehension syntax in Python is easy to understand.

```
>>> [[x for y in range(1, x+1)] for x in range(1,5)]
 [[1],  [2,2],  [3,3,3],  [4,4,4,4]]
```

We encourage entry into Python only after a little maturity with typeful thinking that *Pugofer* inculcates. Doing it in this order sets the ontology right.
Note: Python supports LCs but the Python tutorial [42] reduces LCs from a math concept to a `for` loop.

⊙ **Law 2.** ***Types precede values***

The classic Platonic form of the above is: *The Platonic world of ideals is real; the empiric world that we call real is only its shadow.*

Nowadays the Platonic world is called the *World of Forms*. The mistranslation of *eideos* couldn't be more stark:
eideos (Greek): essence (closer to soul) → ideal → idea → concept → (meaningless) form. A more secular rendering is:

⊡ **Fact.** Types in Programming ≃ Well-formed-ness in Math

In the context of bad physics, the physicist Pauli famously said: *It's not even wrong.* This 3-valued Pauli logic – right, wrong, and not even wrong – is an important cornerstone that takes a student from being a non-mathematician to becoming an able mathematician. Typeful thinking is the reification of the *not even wrong.* If we would like our students to stay aligned to this original mathematical truth, then

☞**Action.** CS101 should use a programming vehicle that aligns more with *types for thinking* than *types for efficient code-generation* (C/C++) or *types as runtime tags* (Python)

Type clarity is model clarity, and therefore it follows that: A type error means the programming model is not proper. *It is not a programming error, it is a thinking error.* Type inferencing support from the language allows the model to be thought with, checked, composed and verified.

In the case of recursive data types, these errors can be quite hard to debug. So checking the model in `pug` and then translating it into Python often shortens the coding time compared to straight Python.

⊡ **Fact 2.** Python-like languages with a single reflexive universe for types and values are powerful in the hands of experts but messy for the beginner student – their *looseness* encourages sloppy thinking.

⊡ **Fact 3.** For beginner programmers, we need to provide a prop to organise the world of values in their thinking. Types – we often call them *shapes* – are just these props.

☞**Action.** Emphasize a crisp separation between the world of types and the world of values.

For example, notice how in `f :Int → Int` the value `f` is on the left of the `:` and its type is on the right. We emphasise that the 2 sides of the `:` are the two worlds that programmers relate, and that each time they **conceive** a value they must be sure of its type. This is the ideal. In practice, obedient `pug` aids and guides greatly.

⊙ **Law 3.** ***It is data that is recursive, code only follows.***

⊡ **Fact.** Induction and recursion are two sides of the same coin.

Most students have already studied math before entering a CS curriculum. Induction is already encountered in math, so it is just a matter of beginning from what is known. This method has been followed in many places including [43] and surveyed in [3]. We go one step further:

Consider a structurally inductive type which we have seen before in Fig 1. Let us define a function that swaps the left and right subtrees of a tree `swaptree :BinTree →BinTree`. Since `Lf` and `Br` are the only 2 constructors, every `Bintree` value must match one of the 2 pattern-forms in its type definition. And when the function mirrors just these 2 type patterns, it describes a **total** function over this type:

```
swaptree.(Lf.x)   =  Lf.x
swaptree.(Br.l.r)  =  Br.(swaptree.r).(swaptree.l)
```

⊙ **Corollary.** *Code structure only mirrors data structure*

That is to say: **Once data structure is known, code structure can be deduced.** We then work with several inductive proofs over this type. For a detailed coverage see [43]. The ease with which students prove and write code from this advice tells us that it matches an interesting corollary from [44]:

⊙ **Corollary.** *Trees before lists and lists before natural numbers*

This order, with the right level of generality, begins to put into perspective lists as skewed trees and numbers as a structurally similar type, even if with different constructors. Students have seen sequences in some form, but not in recursive form. So starting with lists *has an unlearning element.* But trees are new. There is also a visual appeal to its recursive structure.

We have often received comments from students that while they have solved many problems in mathematical induction (with $\mathbb{N}$) earlier, they understood induction only after seeing structural induction (with trees and lists).

This is usually achieved after the drill with simple recursive functions on lists and trees, and proving properties about them.

☞**Action.** In order to motivate induction, teach structural induction as the norm, with natural number induction (on $\mathbb{N}$) as the special case.

There is a wide variety of examples of this in [43] and elsewhere so we do not reconstruct them here. Our point of departure is *the intentional pedagogical sequencing.*

⚠ *Caveat.* Of course if one chooses a language like Python then it gets messy.

In our finding, computational aspects, like the relation between iteration and tail-recursion [45], while a necessary part of the whole picture, ought to be delayed until structural aspects of data are well assimilated. Implementation notions like stack add a technological detail that is heavyweight for a newcomer and not essential to the idea of recursion [46]. We discuss these as part of imperative programming and language implementations later in the curriculum. For more on the movement from recursion to tail recursion which we would typically cover in a later programming course, we suggest [8] as a reference. We, in fact, take it forward to continuation passing style (CPS) in a programming paradigms course, but that is not for this paper. So, the path of least resistance in terms of reducing ontological weight is:

☞**Action.** Teach recursion in data structure before recursion as control structure

⊙ **Law 4.** ***The infinite precedes the finite***

The insight that *Recursion is squeezing the infinite into the finite* makes it evident why infinity before finitude, or more specifically, co-recursion before recursion, is a sound idea. This may seem counter-intuitive in our time because we **equate reality with empiric reality**. However if we start from math, the scene looks quite different. Even a seemingly finite entity like the number 3 belongs to the infinite set $\mathbb{N}$ which itself is the start of a tower of transfinite cardinalities that rises dizzyingly, endlessly. In short, math is always about the infinite, whether obviously or subtly. There are other versions of the law:

⊙ **Corollary.** *The infinite is simpler than the finite*

This becomes evident if we start with types rather than values [Law 2]. Infinite list values in `pug` can be described with the following type:

```
ctype List.a where    -- infinite
  Cons : a → List.a → List.a
```

While a finite type needs an extra constructor, and is defined as:

```
ctype List.a where    -- finite
  Cons  : a → List.a → List.a
  Empty : List.a
```

And from here we get the action

☞**Action.** Teach corecursion before recursion

When teaching recursion, the conventional wisdom is that it is very important to get the base case right. With infinite data there are no base cases. The base case is needed to limit the infinite data but adds complexity to understanding.

⊙ **Corollary.** *Corecursion is simpler than recursion because the base case can be broken off as a separable concern; unlike recursion where a bad base case guarantees a broken program.*

Two simple example definitions can illustrate this:

- The infinite list containing 1 defined as:
  `ones =1::ones`
- The list of natural numbers defined as:
  `nats =1::[x+1 | x <- nats]`

These can be easily verified in a dialogue with `pug`

```
? ones
[1,1,1,1,1,1...
? nats
[1,2,3,4,5...
```

▷ *Note.* The more advanced and subtle distinction in the Theory of Computing (ToC) between decidable and semi-decidable hinges on the fact that infinite set enumeration may be meaningful even when it does not terminate. In fact,

⊙ **Corollary.** *Enumeration in ToC is corecursion in programming.*

⚠ **Caveats**

1) Corecursion is meaningless in the context of imperative languages. In particular:

   ⊙ **Corollary.** *Co-recursion and assignment simply cannot coexist*

   The school child learning algebra learns to "simplify" $x = \sqrt{2 + \sqrt{2 + \sqrt{2 + ...}}}$ into $x = \sqrt{2 + x}$ which trivially solves to $x = 2$. This cannot be acceptable if the infinite $\sqrt{}$ is read as a literal process – a view that the imperative programmer is committed to. Likewise the imperative programmer seeing Pugofer/Haskell's infinite list of ones defined as `ones =1 ::ones` has no choice but to be dazed by being committed to having the rhs completely well-cooked and in the pocket before the assignment to lhs happens. In short:

⊙ **Corollary.** *Co-recursion is not intrinsically hard; it is hard in the context of assignment/mutation, i.e., IP.*

2) Lazy Evaluation is the traditional way of looking at things like infinite lists … **by imperative programmers**. It assumes the framing that the norm is imperative, and lazy evaluation is the odd case for managing infinite data structures. In our view the older term "normal order" [47] would correct the biased frame.

   ⊙ **Corollary.** *"Lazy evaluation" is the imperative (programmer's) mis-framing of normal (order) evaluation*

3) Corecursion manifests as generators in (say) Python. But to posit Python's

```
def ones():
  while True: yield 1
```

   as **the primary form** of infinity and its `pug` form `ones =1 ::ones` as an exotic aside is in our view the ontological error of making the computational process primary and the mathematical account of it an exotic curiosity.

4) This ontological weight then ramifies as pedagogical burdens: Imperative programmers already need to deal with the ambiguous status of `return` [Section II-C]. Now they further need to think of distinguishing `return` from `yield` as well!

⊙ Law 5. *"Recursion" after Recursion*

Undergraduate entrants to CS have studied arithmetic and algebraic expressions. That *expressions contain subexpressions* is well-understood. In other words, students who have already seen nesting operationally in school algebra, now reify that as "recursion" of the nested data structure `expression`

From here nested types like we see in Fig. 1 come naturally. Take another example: counting the number of combinations. Students are already familiar with ${}^nC_r$ They calculate with the Pascal formula: ${}^{n+1}C_r = {}^nC_r + {}^nC_{r-1}$. As a mathematical construction, we need to have well defined cases over the type $\mathbb{N}$ and so we define c as:

$$
\begin{array}{lll}
c: \mathbb{N} \to \mathbb{N} \to \mathbb{N} & & \\
0 & \text{‹c›}\ (r{+}1) & = 0 \\
n & \text{‹c›}\ 0 & = 1 \\
(n{+}1) & \text{‹c›}\ r & = n\ \text{‹c›}\ r + n\ \text{‹c›}\ (r{-}1)
\end{array}
$$

What does it take to run it in `pug`? A slight notational movement, ascii-ing and voila! we have a pug-gram... er program!

```
c : Int → Int → Int
0     `c` (r+1)  =  0
n     `c` 0      =  1
(n+1) `c` r      =  n`c`r + n`c`(r-1)
```

Next, we make one small but significant change: we keep the definition structure and change the data structure. Thus we enumerate the elements we are counting by scaling up from numbers to sets, while keeping the known recursive form invariant.

⊡ **Fact.** Constructions and enumerations can provide these intuitions into math formulae.

So we *calculate* recursive data structures like sets and lists, by assuming we know recursion. From this it follows:

☞**Action.** Keep the familiar recursive form; scale up the data structure.

Note that `c` is just the classic Pascal identity. We now define `cE`, the corresponding enumerator that in fact computes the combination values for a given ${}^nC_r$. In fact, here we notice the need to move from set comprehensions to deterministic list comprehensions, an opportunity to discuss a more general theory of collections [48].

```
cE : [t] → Int → [[t]]
[]     `cE` (r+1) =  []
xs     `cE` 0     =  [[]]
(x::xs)`cE` r     =  xs `cE` r ++ [x::y|y ∈ xs `cE` (r-1)]
```

We can verify this with `pug` as our dialoguer:

```
? 4 `c` 2
6 : Int
? "abcd" `cE` 2
["cd", "bd", "bc", "ad", "ac", "ab"] : [[Char]]
```

Philosophical Slant:
The above is a foray into basic combinatorics. It has been studied before, and its recursive form is familiar. Math is the primary focus when we start and that it runs on a computer is secondary. You could call it executable math, but it helps to introduce these functions as calculators evaluating sophisticated data structures.

Playing calculators with recursive data without worrying about its mechanics, rides on previous knowledge of the form of recursion encountered in high school.

⚠ *Caveat.* This is only possible with equational reasoning (as in FP)

⊙ Law 6. *Math before programming*

The Persian *al-Khwārizmi* is the man whose name today gives us "algorithm". And his book, *al-Kitāb al-Mukhtasar fi Hisāb* ***al-Jabr*** *wal-Muqābalah* is where we get "algebra" [49]. So historically, *al-Khwārizmi* (algorithm) precedes *al-Jabr* (algebra). But when we switch from fact to principle, the order inverts: we give our attention to *algorithm* only because we consider algebra significant. (The intertwined Indo-Greek-Persian-Arabic history does not concern us here [50], [51])

A similar flip happens in the world of programming: The imperative paradigm (IP) is historically the first to happen. And then we go to the divisions today between IP, OOP, FP, etc. When we revert to *al-Khwārizmi* where algorithm and algebra originate, we see that *expressions* denoting *values* naturally precede *statements* embodying *actions*. Seen in the grammar of any imperative language, the simplest statement – assignment – cannot be written without the right hand side, an expression. In more detail [52]. So we could also say:

⊙ **Corollary.** *Algebraic before algorithmic*

Further, putting together Dijkstra's fulminations against "operational thinking" with Backus' exhortation to FP, we get:

⊡ **Fact.** Plato precedes von Neumann

... not just historically but pedagogically as well. Therefore the teaching suggestions whose general direction is emerging in this writing:

☞**Action.** Teach interpreted languages before compiled ones

Compiled languages result in a mixup of notions of evaluation and execution [53] and create a distance between syntax and meaning, burdening beginners with motley machine and toolchain details. Instead, an interpreter serves as a calculator over complex data structures. To go from the dominant CS-school trend where compiled languages are employed de rigeur in preference to interpreted ones, we need to remember:

⊡ **Fact.** The Turing machine is an *interpreter* as is the von Neumann machine.

☞**Action.** Evaluation – rather than execution – should be the beginner's programming model

⊙ **Law 7.** ***One example, many concepts***

Many ideas threaded through one example instead of diverse ideas in diverse examples is more effective than one example per idea. This is of course a teaching law and follows from teaching *one new thing at a time.* Below we take a case study: the classic problem of fibonacci through many different recursion schemes.

The basic math definition

```
fib.0      =  0
fib.1      =  1
fib.(n+2)  =  fib.(n+1) + fib.n
```

The following Python code snippet defines a trivial transcription of the above math fibonacci with exponential (actually fibonacci!) behavior. Note that the advantage of this is (a) student sees clearly the step from math to programming, and that (b) naively translating math to programming can have terrible performance!

```
def fib(n):
  if n == 0: return 0
  elif n == 1: return 1
  else: return fib(n-1) + fib(n-2)
```

From here we have two paths possible:

- Improve the behavior towards linear (even logarithmic)
- Or towards corecursion. It is a fork in the pedagogic tree not a sequence. [In this paper we start with the first, though the second – corecursion first – has some advantages]. And so we have the following definitions of `fib` in `pug` and Python respectively:

Input recursion

```
fib : (Int,Int)  →  Int  →  Int
fib.(a,b) .0     =    a
fib.(a,b) .(n+1) =    fib.(b,(a+b)) .n
```

```
def fib(a,b,n):
  if (n==0): return a
else:
  return fib(b,a+b,n-1)
```

This version is good from the point of view of tail recursion and therefore loop-ability. But it is not so good for further exploration. So lets try differently:

Output recursion

Note how the fibpairs $\mathbb{N} \times \mathbb{N}$ in the input recursion flip over to output recursion in the following code:

```
fib : Int  →  (Int, Int)
fib.0      =  (0,1)
fib.(n+1)  =  (b,a+b)
    where (a,b)  =  fib.n
```

```
def fib(n):
    if n==0: return (0,1)
    (a,b) = fib(n-1)
    return (b,a+b)
```

Towards co-recursion

From here to co-recursion is easy. But we'll back off and start with the functional Bird & Wadler classic [43] along with the same "compressed" with zipWith:

```
fibs  =  1::1::[x+y | (x,y) ∈ zip.fibs.(tail.fibs)]
fibs  =  1::1::zipWith.(+) .fibs.(tail.fibs)
```

The key ideas in co-recursion are that (a) data can be essentially recursive; one does not need to go via recursive functions, and (b) the data can be infinite. In our running example, notice how the question of our search has become: *How to define the infinite list of fibonacci numbers in terms of itself?*

Pythonizing this turns out clunky. There is a need to copy (`tee`) band-aiding over the imperation.

```
from itertools import tee  #aka copy
def fibs():
    yield 1
    yield 1
    (f,g) = tee(fibs())
    next(g)
    yield from (x+y for (x,y) in zip(f,g))
```

Zip-Tailing

Insight: [43] group the `zip` and + into `zipWith`. What if we instead grouped `zip` and `tail` into a zigzag pattern? Let's call this function `zipzag`.

Specification: `zipzag.l =zip.l.(tail.l)` refines to:

```
zipzag.(x::y::xs)  =  (x,y)::zipzag.(y::xs)
fibs  =  1::1::[x+y | (x,y) ∈ zipzag.fibs]
```

Notes:

- Here *no base case is needed.* Co-recursion does not need a base case. In other words *Infinite is easier than finite.*
- `zipzag` does ...x,y... → ... (x,y)... while `fib` gets `(x,y)` from the `zipzag` and flattens out to ... `x+y` ... In other words the data structure keeps alternating
- Clearly the pair (x,y) is relevant to the problem; why not bring it into the model explicitly?

Imperating

See the implicit imperation: l=tail(l) so we can python-ize as:

```
def zipzag(l):
    a = next(l) # implicit imperation!
    for b in l:
        yield (a, b)
        a = b
```

```
def fibs ():
    yield 1; yield 1
    for (x,y) in zipzag( fibs ()):
        yield x + y
```

From 2nd order to 1st order recurrence

Dijkstra's intuition regarding fibonacci: *The programmer converts a 2nd order linear recurrence on $\mathbb{N}$ into a first order recurrence on $\mathbb{N} \times \mathbb{N}$* i.e. $(a,b)_n \to (b, a+b)_{n+1}$

Carrying over this intuition into pug we get `fibpairs` defined as:

```
fibpairs = iterate.(λ(a,b) → (b,a+b)).(1,1)
```

A straight forward pythonizing of pug doesn't work!

```
def iterate (f,x):
    yield x
    yield from iterate (f,f(x))
def fibs ():
    yield from iter((lambda a,b: (b, a+b)), (1,1))
>>> for x in fibs (): print(x)
TypeError: <lambda>() requires 2 arguments: 'a' and 'b'
```

This is because (a,b) in Python can either be a tuple or 2 function arguments – the pedagogical cost of multi-ontologies! The more literal pug → Python is clunky and confusing:

```
def fibpair ():  # ab is the (a,b) pair
    yield from iterate ((lambda ab:(ab[1],ab[0]+ab [1])), (1,1))
```

Finally, we idiomise it into the more Pythonic version:

```
def stepp(p): # step on pairs
    a,b = p
    return (b,a+b)

def fibpair3 ():  # 3rd attempt!
    yield from iterate (stepp, (1,1))
```

IV. Finite-infinite: Wall or Bridge?

For the mathematician, this paper presents nothing new: a small set of equations can have infinite solutions. It is radical for us programmers because we walk the steps from intended-executing-process to program-that-evokes the process. This is backwards from the mathematician's equations-to-solution direction. The programmer's job after all is to construct a finite artifact – the program – that embodies an infinite intention – the process. Programming inverts math because in math the equations are presented externally and the mathematician finds the solution, whereas in programming the programmer formulates the equations and the machine 'solves' them – usually called *execution.* So finite→infinite is math; infinite→finite is programming.

From loops to tail calls to corecursion to general recursion – recursion in all its myriad forms [1] – is the bridge that embodies this finite-infinite conundrum. So understanding recursion can hardly be a cakewalk! And yet an early digestion of recursion is key to the creation of a programmer.

What prevents us programmers integrating this knowledge into our praxis are the following dualities:

- Essential vs Extraneous: Much of practised programming pedagogy complicates teaching by throwing in motley historically accreted details of mainstream PLs as if they were a necessary part of the programming domain.
- Imperative vs Functional: While imperative programming is imperative(!) for the well-rounded CS-ist, presenting it *as fundamental* makes all other forms seem *'advanced paradigms'*. Following Kuhn's 2012 recanting [54] of his seminal 1962 work that introduced 'paradigm' [55], we suggest some suspicion toward this rampant 'paradigm-itis'.

If the essential math dichotomy is finite Vs. infinite, the engineer's is theory/math Vs. practice/engineering. This dichotomy reifies into the constructive sequence: IP–machines **refined from** FP–executable specifications [56]. In short our suggestion is to focus on engineering *after* math – the two passes – rather than engineering *over* math. In summary:

☞**Action.** Teach mainstream IP/OOP **after** CS-as-math

follows naturally from the meta-circular law

⊙ **Law.** *Actions follow Laws*

V. Conclusion

This paper has demonstrated the general problem of programming pedagogy with the specific exemplar of recursion. We believe and have attempted to show that focussing on *some technology* of an extant programming language before introducing the equational essence is at issue.

Taking a step back from the conventional approach, the problem can be factored into (a) the Platonic essence of the programming problem: to see a program as a solver of equations followed by (b) the details of 'coding it up' into a mainstream language. **Formulating problems as equations is our primary business. Code merely codifies this.**

Lamport [57], [58] makes a similar point to ours of prioritizing mathematical modelling and reasoning over language technology except he uses TLA where we use *Pugofer*.

The method has been found to work well: student reviews are recorded in somewhat more detail here [59]. Our repeated feedback is best summed in just one comment of a delighted student: **Recursion is magic**!

Acknowledgment

Prof. Hari V. Sahasrabuddhe (hvs), switching from Pascal to Scheme for CS101 in 1988, started this journey; his dragging the unwilling first author to teach CS101 with Scheme + C in '91 kept the ship on course and prepared for the platonic-empiric balance – he'd probably say math-engineering balance. Mark Jones made these dreams realizable in '93 (on IBM PC-XTs) with his delightful creation – gofer. 20 years later, Dr. Anand Deshpande continued what hvs started by giving us a free hand for 8 years in summer/winter programming bootcamps; Prof. Aamod Sane, a recent adopter of *Pugofer,* gave valuable feedback. Finally, thanks to over two thousand students across 30 years for suffering us!

References

[1] R. P. Mody. (2012) Recursion pervasive in CS. [Online]. Available: http://blog.languager.org/2012/05/recursion-pervasive-in-cs.html

[2] R. McCauley, S. Grissom, S. Fitzgerald, and L. Murphy, "Teaching and learning recursive programming: a review of the research literature," *Computer Science Education*, vol. 25, no. 1, pp. 37–66, 2015.

[3] C. Rinderknecht, "A survey on teaching and learning recursive programming," *Informatics Educ.*, vol. 13, pp. 87–119, 2014.

[4] T. Verhoeff, *A Master Class on Recursion*. Springer, 2018, pp. 610–633. [Online]. Available: https://doi.org/10.1007/978-3-319-98355-4_35

[5] L. Butgereit, "Teaching recursion through games, songs, stories, directories and hacking," in *2016 Int. Conference on Advances in Computing and Communication Engineering*. IEEE, 2016, pp. 401–407.

[6] Greg Michaelson, "Teaching recursion with counting songs," *ACM SIGCHI, June*, June 2015.

[7] W. H. Burge, *Recursive Programming Techniques*. Addison-Wesley Publishing Company, 1975.

[8] H. Abelson, G. J. Sussman, and with Julie Sussman, *Structure and Interpretation of Computer Programs*. Cambridge: MIT Press/McGraw-Hill, 1996.

[9] J. Samet, "The historical controversies surrounding innateness," in *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed. Metaphysics Research Lab, Stanford University, 2019. [Online]. Available: https://plato.stanford.edu/archives/sum2019/entries/innateness-history/

[10] E. Wigner. (1960) The unreasonable effectiveness of mathematics. [Online]. Available: https://www.maths.ed.ac.uk/~v1ranick/papers/wigner.pdf

[11] R. W. Hamming, "The unreasonable effectiveness of mathematics," *American Math. Monthly*, vol. 87, no. 2, Feb. 1980.

[12] A. Rocke. (2021, June) Revisiting the unreasonable effectiveness of mathematics. [Online]. Available: https://mathoverflow.net/questions/394934/revisiting-the-unreasonable-effectiveness-of-mathematics

[13] R. P. Mody. (2014) Pugofer to pug. [Online]. Available: http://blog.languager.org/2014/09/pugofer.html

[14] (2014) Pugofer: A platonic universe that is good for equational reasoning. [Online]. Available: http://blog.languager.org/2014/09/pugofer.html

[15] R. P. Mody. (2022) Pugofer to pug. [Online]. Available: http://blog.languager.org/2022/03/pugofer-pug.html

[16] ——. (1995) Haskell to pug: Motivations and syntax. [Online]. Available: http://github.com/rusimody/pugofer/tree/master/techreports/pug-a-teachers-haskell.pdf

[17] R. Mody, "C in education and software engineering," *SIGCSE Bull.*, vol. 23, no. 3, p. 45–56, sep 1991. [Online]. Available: https://doi.org/10.1145/126459.126471

[18] B. Schwartz, *The Paradox of Choice: Why More Is Less*. Harper Perennial, 2005.

[19] T. Verhoeff, "Look Ma, backtracking without recursion," *Olympiads in Informatics*, vol. 15, pp. 119–132, Jun. 2021. [Online]. Available: https://ioi2021.sg/ioi-conference/

[20] Classes without OO. [Online]. Available: https://wiki.c2.com/?ClassesWithoutOo

[21] B. Heeren, D. Leijen, and A. van IJzendoorn, "Helium, for learning haskell," in *Proc. of the SIGPLAN Workshop on Haskell*. ACM, 2003.

[22] F. Ruehr. The evolution of a haskell programmer. [Online]. Available: http://www.willamette.edu/~fruehr/haskell/evolution.html

[23] P. Wadler, "A critique of Abelson and Sussman or why calculating is better scheming," *ACM SIGPLAN Notices*, vol. 22, no. 3, pp. 83–94, 1987.

[24] Scott Rowe. The two doctrines truth. [Online]. Available: https://cseducators.meta.stackexchange.com/questions/436/the-two-doctrines-truth

[25] R. P. Mody. (2014, Sep.) Are typeclasses essential. [Online]. Available: https://stackoverflow.com/a/26071288/3700414

[26] T. J. Myers, *Equations, Models, and Programs: A Mathematical Introduction to Computer Science*. Prentice Hall, 1988.

[27] R. Milner, R. Harper, D. MacQueen, and M. Tofte, *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[28] R. J. Hindley, "Principal type scheme of object in combinatory logic," *Transactions of AMS*, vol. 146, pp. 29–60, 1969.

[29] L. Damas, "Type assignment in programming languages," Ph.D. dissertation, University of Edinburgh, 1984.

[30] R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and System Sciences*, vol. 16, no. 3, pp. 348–374, 1978.

[31] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi, *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, 2014.

[32] R. B. Findler and PLT. How to design programs teaching languages. [Online]. Available: https://docs.racket-lang.org/drracket/htdp-langs.html

[33] The Haskell Report. [Online]. Available: https://www.haskell.org/onlinereport/haskell2010/

[34] M. Sahami, S. Roach, E. Cuadros-Vargas, and R. LeBlanc, *Computer Science Curricula 2013*. New York: Association for Computing Machinery, 2013. [Online]. Available: https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf

[35] Richard O'Keefe. [Online]. Available: https://erlang.org/pipermail/erlang-questions/2013-January/071898.html

[36] E. W. Dijkstra and C. S. Scholten, *Predicate Calculus and Program Semantics*. Springer, 1990.

[37] E. W. Dijkstra. The notational conventions I adopted. [Online]. Available: https://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1300.PDF

[38] R. P. Mody and A. Laxminarayan. (2012) Doting on the dot. [Online]. Available: http://github.com/rusimody/pugofer/tree/master/techreports/doting-on-the-dot.pdf

[39] R. P. Mody. (2004) A thought dialogue with Edsger Dijkstra. [Online]. Available: http://github.com/rusimody/pugofer/tree/master/techreports/ewd-dot-dialogue.pdf

[40] Connor "pigworker" McBride. [Online]. Available: https://stackoverflow.com/questions/27856974/functor-instance-for-generic-polymorphic-adts-in-haskell#comment44118975_27856974

[41] Generalized Algebraic Datatypes (GADT). [Online]. Available: https://en.wikibooks.org/wiki/Haskell/GADT

[42] The Python Tutorial. [Online]. Available: https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions

[43] R. Bird and P. Wadler, *Introduction to Functional Programming*. Prentice Hall International series in Computing Science, 1988.

[44] Musa al Hassy. "Graphs:Categories :: Lists:Monoids". Numbers are just the lengths of lists which are just the flattenings of trees. [Online]. Available: https://alhassy.github.io/PathCat

[45] C. M. Kessler and J. R. Anderson, *Learning flow of control: recursive and iterative procedures*. Lawrence Erlbaum associates, 1986, vol. 2.

[46] Bob Harper. It is what it is (and nothing else). [Online]. Available: https://existentialtype.wordpress.com/2016/02/22/it-is-what-it-is-and-nothing-else/

[47] H. P. Barendregt, *The Lambda Calculus – its syntax and semantics*, ser. Studies in logic and the foundations of mathematics. North-Holland, 1985, vol. 103.

[48] D. Gries and F. B. Schneider, *A Logical Approach to Discrete Math*. Springer, 1993.

[49] Editors of Encyc. Britannica. (2022, June) al-Khwārizmi. [Online]. Available: https://www.britannica.com/biography/al-Khwarizmi

[50] ——. (2017, Sept) Hindu-arabic numerals. [Online]. Available: https://www.britannica.com/topic/Hindu-Arabic-numerals

[51] E. Lamb. The father of algebra: Al-khwarizmi or Diophantus? [Online]. Available: https://3010tangents.wordpress.com/2014/09/17/the-father-of-algebra-al-khwarizmi-or-diophantus/

[52] R. P. Mody. (2016, Jan.) The law of primacy. [Online]. Available: http://blog.languager.org/2016/01/primacy.html

[53] Abhijat Vichare, Private communication, c. 2004.

[54] J. Horgan, "What Thomas Kuhn really thought about scientific Truth," *Scientific American*, May 2012.

[55] T. S. Kuhn, *The Structure of Scientific Revolutions*. Chicago: University of Chicago Press, 1962.

[56] D. Turner, "Functional programs as executable specifications," *Phil. Trans. of the Royal Society of London*, vol. 312, 1984.

[57] S. Han interviewing Leslie Lamport, "How to write software with mathematical perfection," *Quanta Magazine*, May 2022.

[58] L. Lamport. (2022). [Online]. Available: https://lamport.azurewebsites.net/tla/tla.html

[59] A. Sane, R. Mody, A. Laxminarayan, and V. Jayaraman, "Lifecycle in CS1: Requirements, domain analysis, and implementation," in *Proceedings of the ITiCSE 22*. ACM, 2022, p. 269–275.