Recursion: Conceptual Necessity or Convenience? Cause, Effect, or By-product?*

Cem Bozşahin

Department of Linguistics
Boğaziçi University
Bebek İstanbul 34342 Türkiye
cem.bozsahin@bogazici.edu.tr

Abstract

We show that the often-discussed question raised by recursion about whether recursive definitions are indispensable is provably semantic in nature. Recursive description with constant branching and depth is not a necessity or sufficiency but a syntactic convenience. Computation without recursive description is a by-product (i.e. an effect) of finding fixpoints of functionals, whether or not functions and functionals are recursively defined. This is a semantic problem independent of recursion because it only concerns values and their equality. Therefore, the often discursive and analogical discussions of recursion in linguistics, cognitive science and computer science, about whether recursive description is syntactically necessary or computationally unavoidable, can, perhaps must, be avoided if its mathematical properties are sufficiently clear. We show that they are.

1 Introduction

[HCF02] start the discussion of recursion in linguistics and cognitive science as follows:

> We argue that an understanding of the faculty of language requires substantial interdisciplinary cooperation. We suggest how current developments in linguistics can be profitably wedded to work in evolutionary biology, anthropology, psychology, and neuroscience. We submit that a distinction should be made between the faculty of language in the broad sense (FLB) and in the narrow sense (FLN). FLB includes a sensory-motor system, a conceptual-intentional system, and the computational mechanisms for recursion, providing the capacity to generate an infinite range of expressions from a finite set of elements. We hypothesize that FLN only includes recursion and is the only uniquely human component of the faculty of language. [HCF02]:1569

There is a reference here to a computational system and some mathematical properties, for example "infinite range of expressions from a finite set of elements." The properties of recursion in the so-called narrow language faculty (FLN) are consequently described discursively:

> We assume, putting aside the precise mechanisms, that a key component of FLN is a computational system (narrow syntax) that generates internal representations and maps them into the sensory-motor interface by the phonological system, and into the conceptual-intentional interface by the (formal) semantic system; adopting alternatives that have been proposed would not materially modify the ensuing discussion. All approaches agree that a core property of FLN is recursion, attributed to narrow syntax. [HCF02]:1571

It is one of Chomsky's now familiar "virtually conceptually necessary" arguments, this time for recursion, which continue to this day [RWC23]; see [Pos03] for extended discussion.

[Lob14] pointed out and I corroborated [Boz16]—along with many others—that this discursive language has done more damage than good in understanding the role of recursion as

*I am grateful to Başak Aray and PLS15 reviewers for comments and discussion; to Yasin Aydın who prompted me to rethink about recursion; to Halit Oğuztüzün and Taylan Özgür Yıldırım for making our meeting possible.

an abstraction, causing confusion because of unnecessary ontological commitments concerning mechanisms and substrate. Lobina leaves the criticism at the level of recursive *computation* so I start with his assessment of conflations in linguistics before moving further:

> a) recursion was first employed in the 1950s and 60s in the same manner as it was used in mathematical logic, a field that exerted a great influence on linguists at the time; and b) its application has not translated much over the years, at least as far as Chomsky's individual writings are concerned. Building upon that, I [..] demonstrate that the confusion surrounding this concept is not solely a matter of imprecision, [..] but a story of conflations: between recursive mechanisms and recursive structures; namely, between the self-reference so typical of recursive functions and self- embedded sentences; between the recursive applications of specific rules of Post production systems and self-embedding operations; and, lastly, between what an operation does and how it applies. As a result, I will conclude, most of the discussion in the literature as to the centrality and uniqueness of recursion in natural language has centred on issues (such as whether all languages exhibit self-embedded sentences) that have little to do with the introduction of recursive tools into linguistics, let alone the reasons for introducing such techniques in the first place. As such, then, some of the strongest claims to be found in the literature are either fallacious or quite simply misplaced. [Lob14]:57

His conclusion reads as yet another lesson in science that making a concept the cornerstone of a bold claim without clarifying it does not lead to a better understanding:

> there is a bit of a disconnect between what Chomsky is talking about when talking about recursion and what other linguists are talking about when talking about recursion. As a result, rather incompatible claims are being put forward in the literature; indeed, to defend that what is universal in linguistics is the recursively defined operations of a computational system (clearly Chomsky's take) is very different from the claim that all languages must exhibit self-embedded sentences if recursion is to be accepted as such a central concept. [Lob14]:69

I show that the recursively defined operations of a computational system are not the core properties underlying the functional dependencies we seek to understand in recursion. It is not only computationally realizable (i.e. decidable) *primitive recursion* that Lobina and colleagues bring to focus in showing inadequacies, but *general recursion* has the same property of being a consequence of solving something else: fixpoints. Therefore we must seek the answer to the recursion puzzle in the mathematical invariant rather than the computational variant, Poincaré-style [Poi05]. Recursion is a representational device; fixpoint (equality) is the semantic invariant.

2 Recursion Sui Generis

Recursion is often informally described as a form of self-application, with the understanding that it typically requires a name to be able to do that. The combinator **U** illustrates pure self-application and serves as a useful starting point for understanding general recursion:

$$\mathbf{U} = \lambda f.f(f) \tag{1}$$

Self-application by name is evident in the following lambda term; $\mathcal{F}$ is referred inside itself:

$$\mathcal{F} = \lambda n.\,\texttt{if}\; n = 0\,\texttt{then}\,1\,\texttt{else}\,n \times \mathcal{F}(n-1) \tag{2}$$

Self-application is obtained by first abstracting over the name in (2), as in (3), to obtain a nameless functional, then applying it to the named abstraction to obtain $\mathcal{H}\,\mathcal{F} = \mathcal{F}$, in (4) and (5). $\mathcal{F}$ is the fixpoint of $\mathcal{H}$ in (3).

$$\mathcal{H} = \lambda f.\lambda n.\,\texttt{if}\; n = 0\,\texttt{then}\,1\,\texttt{else}\,n \times f(n-1) \tag{3}$$

$$\mathcal{H}\,\mathcal{F} = [\lambda f.\lambda n.\,\texttt{if}\; n = 0\,\texttt{then}\,1\,\texttt{else}\,n \times f(n-1)]\,\mathcal{F} = \tag{4}$$

$$\lambda n.\,\texttt{if}\; n = 0\,\texttt{then}\,1\,\texttt{else}\,n \times \mathcal{F}(n-1) = \mathcal{F} \tag{5}$$

This is not what nameless **U** does: $\mathbf{U}\mathcal{F} = \mathcal{F}(\mathcal{F}) \neq \mathcal{F}$. Thus simple self-application alone does not yield recursion. We define the following term, which is nameless (x and f are placeholders):

$$\mathbf{\Phi} = \lambda x.\lambda f.f(x\,x\,f)$$

Then $\mathbf{\Phi\Phi}$ (due to [Tur39]) gives us the term in (6), which, when supplied with a function, say h, yields (7).

$$\mathbf{\Phi\Phi} = (\lambda x.\lambda f.f(x\,x\,f))\mathbf{\Phi} = \lambda f.f(\mathbf{\Phi\Phi}f) \tag{6}$$

$$\lambda f.f(\mathbf{\Phi\Phi}f)h = h(\mathbf{\Phi\Phi}h) \tag{7}$$

Therefore $\mathbf{\Phi\Phi}h = h(\mathbf{\Phi\Phi}h)$. This is a characteristic equation, not of recursion but fixpoints. The difference between $\mathbf{\Phi\Phi}$ and $\mathcal{H}$ is that although $\mathcal{H}$ can only find the fixpoint (2), $\mathbf{\Phi\Phi}$ can produce the fixpoint of any functional (like $\mathcal{H}$) if one exists. Moreover, in producing fixpoints $\mathbf{\Phi\Phi}$ handles recursion without names. For example, for (2)–(5), it operates on the nameless $\mathcal{H}$:

$$\begin{aligned}\mathcal{F}=\mathcal{H}\mathcal{F}=\mathbf{\Phi\Phi}\mathcal{H}=\mathcal{H}(\mathbf{\Phi\Phi}\mathcal{H})=[\lambda f.\lambda n.\,\mathtt{if}\;n=0\;\mathtt{then}\;1\;\mathtt{else}\;n\times f(n-1)](\mathbf{\Phi\Phi}\mathcal{H})=\\ \lambda n.\,\mathtt{if}\;n=0\;\mathtt{then}\;1\;\mathtt{else}\;n\times(\mathbf{\Phi\Phi}\mathcal{H})(n-1)=\\ \lambda n.\,\mathtt{if}\;n=0\;\mathtt{then}\;1\;\mathtt{else}\;n\times\mathcal{F}(n-1)=\mathcal{F}\end{aligned}$$

Recall that an operator to find the fixpoint F is a functional X. It obtains the form of the function with no "change," i.e. $XF = F$. Therefore F must be a function and X must be a functional taking a function. These properties are clear in (7): h must be a function, therefore $\mathbf{\Phi\Phi}$ is functional.

My reason for providing details in (1)–(7) is to highlight these functional properties, which will prove to be very relevant to understanding what fixpoint operators do while they handle recursion in the process, and that without names. Altogether they show that what we need in recursion is having a function and a functional to fix it; that is, having values, not names.

3 Recursion and Fixpoints

The best-known fixpoint combinator is **Y** (going back to [Cur29]), with the following definition. It is the same as $\mathbf{\Phi\Phi}$ because $\mathbf{Y}h = \mathbf{\Phi\Phi}h = h(\mathbf{\Phi\Phi}h) = h(\mathbf{Y}h)$.

$$\mathbf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

We can see from the previous section that **Y** must be a functional because $\mathbf{Y}{=}\mathbf{\Phi\Phi}$, and therefore maps functions to functions. Such functions do not have to be recursive at all.

Notice that $\mathbf{Y}{=}\mathbf{\Phi\Phi}{=}\mathbf{U\Phi}$. It means that **U** of (1) itself would not be enough to find fixpoints, or to implement recursion namelessly. What is "missing" in **U** is self-reference to *another* value of the recursing function.

It is common understanding of recursion in linguistics, cognitive science and computer science, that what we refer back to while computing recursively is another *instance* of the *same kind*, like sentences within sentences. These internally nested sentences bear different *values*, for example *I think she thinks he thought I like candies*; compare *I think I think I think I like candies*, which is what we would get if not only kind but values have to be identical.

I now show that fixpoint combinators handle single, multiple and no recursion the same way, therefore what they really care about is fixpointing the function rather than recursion handling. In doing so, however, they turn recursive description into nameless nesting, just the kind that is required in empirical fields which claim that recursion is different *values* of the same kind.

The characteristic equation of the fixpoint problem is the following:

$$\mathbf{Y}h = h(\mathbf{Y}h) = h(h(\mathbf{Y}h)) = h(h(h(\mathbf{Y}h))) = \cdots$$

I use $\mathbf{Y}$ rather than $\mathbf{\Phi\Phi}$ to simplify description. The first thing to note is that these *instances* of h are *functions*, not self-references to a name, therefore different values are to be expected. The process of reaching the fixpoint can be seen in its stages; for example:

$$\mathbf{Y}h = [\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))]h = (\lambda x.h(xx))(\lambda x.h(xx)) =$$
$$h[(\lambda x.h(xx))(\lambda x.h(xx))] = h[[\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))]h] = h(\mathbf{Y}h)$$

One source of the conflations mentioned in the introduction may be this indefinite expansion. That is how recursion manifests itself. However, nonrecursive functions can have fixpoints too. They are found by $\mathbf{Y}$ as well. It is equivalent to saying that $\mathbf{Y}$ is oblivious to whether the function is recursively described, it simply operates on functionals. For example, (8) is not recursively defined. Its fixpoint functional can be (9), defined as such because c must always determine the value.

$$c = \lambda n.1 \times \cdots n \tag{8}$$
$$\mathcal{C} = \lambda g.c \tag{9}$$

$\mathbf{Y}$ can find the fixpoint of this nonrecursive functional:

$$\mathbf{Y}\mathcal{C} = [\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))]\mathcal{C} = [\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))](\lambda g.c) = \tag{10}$$
$$(\lambda x.(\lambda g.c)(xx))(\lambda x.(\lambda g.c)(xx)) = \tag{11}$$
$$(\lambda g.c)([\lambda x.(\lambda g.c)(xx)][\lambda x.(\lambda g.c)(xx)]) = c \tag{12}$$

Notice that (8) is the nonrecursive version of the factorial in (2). That is why it needs a different functional, cf. (9) and (3), but functional nevertheless. Therefore all that recursion needs is functions and functionals, not a name. Even constants can be made functions, for example 1 as $\lambda x.1$, then, to fix it, $\lambda g.\lambda x.1$. We get $\mathbf{Y}(\lambda g.\lambda x.1) = \lambda x.1$.

We have seen that being a constant or having no recursion or single recursion make no difference to the fixpoint combinator because it does not solve the recursion problem; it simply finds a fixpoint if one exists, and in the process of doing so, it eliminates names as a by-product, so that its input is properly set as a functional without names.

Now consider multiple recursion, where there is more than one self-reference inside. Take for example the following named-recursive definition of the Fibonacci sequence:

$$\mathcal{F}_b = \lambda n.\,\texttt{if}\ n \leq 2\ \texttt{then}\ 1\ \texttt{else}\ \mathcal{F}_b(n-1) + \mathcal{F}_b(n-2) \qquad n \in \mathbb{N}^+ \tag{13}$$

We can set its specific fixpoint operator without any extra measures over and above what we have seen so far, an argument which extends to arbitrary degrees of recursion in a single term:[1]

$$\text{Let } \mathcal{H}_b = \lambda f.\lambda n.\,\texttt{if}\ n \leq 2\ \texttt{then}\ 1\ \texttt{else}\ f(n-1) + f(n-2).$$
$$\text{Then } \mathcal{H}_b\mathcal{F}_b = [\lambda f.\lambda n.\,\texttt{if}\ n \leq 2\ \texttt{then}\ 1\ \texttt{else}\ f(n-1) + f(n-2)]\mathcal{F}_b =$$
$$\lambda n.\,\texttt{if}\ n \leq 2\ \texttt{then}\ 1\ \texttt{else}\ \mathcal{F}_b(n-1) + \mathcal{F}_b(n-2) = \mathcal{F}_b.$$

$\mathbf{Y}$ can find the fixpoint of this definition of $\mathcal{H}_b$ like any other, precisely because neither of them are functionals with internal name management that would make the process nontransparent:

$$\mathcal{F}_b = \mathbf{Y}\mathcal{H}_b = \mathcal{H}_b(\mathbf{Y}\mathcal{H}_b) =$$
$$(\lambda f.\lambda n.\,\texttt{if}\ n \leq 2\ \texttt{then}\ 1\ \texttt{else}\ f(n-1) + f(n-2))(\mathbf{Y}\mathcal{H}_b) =$$
$$\lambda n.\,\texttt{if}\ n \leq 2\ \texttt{then}\ 1\ \texttt{else}\ (\mathbf{Y}\mathcal{H}_b)(n-1) + (\mathbf{Y}\mathcal{H}_b)(n-2) =$$
$$\lambda n.\,\texttt{if}\ n \leq 2\ \texttt{then}\ 1\ \texttt{else}\ \mathcal{F}_b(n-1) + \mathcal{F}_b(n-2) = \mathcal{F}_b$$

[1] For example the nonprimitively recursive variably embedding Ackermann function. $\mathbf{Y}$ finds its fixpoint too.

4 Conclusion

Recursive computation is not the underlying cause of anything because no value is conditioned on its presence. Recursive description is a convenience which can be avoided easily *if* recurrence has constant factors of depth and branching, as in Fibonacci without recursive definition:

$$\mathcal{F}_b = \lambda n.(\phi^n - \psi^n)/\sqrt{5} \qquad \phi = (1+\sqrt{5})/2, \quad \psi = (1-\sqrt{5})/2$$

Recursion, starting with the kind that uses constant embedding and branching, although this can continue indefinitely, reaching nontrivial recursive descriptions like Ackermann, is a semantic problem, not necessitating syntactic names. Recursion is not the primitive concept; fixpoint is. Fixpoint computation is not concerned with names but with values. It is solved by an equational theory of functional values, not a logical or computational theory. Equality of values is a semantic problem. We should be talking about recursive values when it comes to intended coverage of structural descriptions. They are semantic objects.

It might be argued that $\mathbf{Y}$ itself is recursive so we would be back to square one. But, it is not recursively defined; it solves an equational problem namelessly. It is set up to *also* handle functionals on recursively defined descriptions in doing so, not just them, for example (8–12). It might also be argued that functions such as Ackermann's show that we cannot always avoid recursive description *because* it seems to have no closed form description. There is a nonrecursive description of it: Define a nameless functional on it as we did before, say $\mathcal{A}$, such that $\mathcal{A}A = A$, where A is the recursive Ackermann. $\mathbf{Y}\mathcal{A}$ is the nonrecursively described Ackermann function.

More importantly, for empirical concerns, Ackermann's structural growth rate in values is unheard of in nature, and we must keep in mind that even as such its fixpoint can be found by any fixpoint combinator without name handling, that is, namelessly *in its computation*. Constant recursion, i.e. constant branching and depth in recursive values, happens to be the case for all manifestations of recursion in linguistics and cognitive science. Therefore recursively specified empirical variants are neither descriptively nor computationally necessary. Consequently, they are not conceptually necessary in the invariant.

Recursion is convenient because it provides a compressed representation of an indefinitely nested structure, which is a concern for the variant (e.g. fixpoint computation), not the invariant. This may be yet another source of conflations mentioned in the introduction.

It is fixpoint's abstraction, not its computation, which points out that all that we need in putting recursion in its right place is the distinction between functions and functionals, which helps us understand the structure and content in nested values of the same kind.

References

[Boz16] Cem Bozşahin. Natural recursion doesn't work that way: Automata in planning and syntax. In Vincent Müller, editor, *Fundamental issues of AI*. Springer, 2016.

[CF58] Haskell B. Curry and Robert Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.

[Cur29] Haskell Curry. An analysis of logical substitution. *American Journal of Mathematics*, 51:363–384, 1929.

[HCF02] Marc Hauser, Noam Chomsky, and Tecumseh Fitch. The faculty of language: What is it, who has it, and how did it evolve? *Science*, 298:1569–1579, 2002.

[Lob14] David J Lobina. What linguists are talking about when talking about... *Language Sciences*, 45(10):56–70, 2014.

[Poi05] Henri Poincaré. *Science and Hypothesis*. Walter Scott Publishing, London, 1905.

[Pos03] Paul Postal. (Virtually) conceptually necessary. *Journal of Linguistics*, 39:599–620, 2003.

[RWC23] Ian G Roberts, Jeffrey Watumull, and Noam Chomsky. Universal grammar. In Douglas A. Vakoch and Jeffrey Punske, editors, *Xenolinguistics: Towards a Science of Extraterrestrial Language*, pages 165–181. Routledge, 2023.

[Tur39] Alan Mathison Turing. Systems of logic based on ordinals. *Proceedings of the London Mathematical Society*, 2(1):161–228, 1939.