Toward the Semantic Reconstruction of Mathematical Functions

Daniel Nikpayuk

February 13, 2020

Abstract

The purpose of this essay is to critically analyze the semantics of mathematical functions with the intention of demonstrating new and expressive ways to construct them theoretically.

Methodology

The general approach of this essay will be to show how functions can be constructed using the "natural" or intuitive strategy known as *naive composition.* Naive in the sense that we don't initially include *recursion* or *currying.*

Once we have the pattern for general functions we will then show styles of decomposition which when reversed allow for reconstructions that do not equate to our original means of combination. From there it will be demonstrated that such reconstructions also hold in the previously excluded special cases of recursion and currying thus completing this claim of universality in this *alternative* function construction.

As this is an essay no formal proofs will be given, the intention is to communicate the ideas and ways of thinking. With that said I've done my best to explain it such that it should be easy to translate into a theorem/proof style by anyone with a bit of mathematical maturity.

Philosophy

What is a function?

I suspect this is a philosophical question more than anything else, and that's worth keeping in mind. Regardless, there are foundational definitions, for example in Set Theory a function

$$f : A \to B$$

is a *relation* $f \subseteq A \times B$ such that

$$\text{for every} \quad (x, y)\ ,\ (x, y')\ \in\ f \quad \text{we have} \quad y = y'\ .$$

This is to say for every input there's at most one output, or rather for every *argument* from its *domain* there's exactly one *image* in its *codomain* (or *range*).

Such definitions aside, for the remainder of this essay I wish to keep our understanding of the nature of functions to be model *agnostic.* I'm more interested in what a function is as a specification rather than as any of its known implementations. With this in mind I make the claim that whatever a function is there are three *kinds* worth noting:

1. **Axiomatic -** For each domain argument x, a unique image y_f **is assumed** to exist. For example in Set Theory there is the *Axiom of Choice* which says we can assume a certain kind of function exists (a "choice function"). As such functions are unlikely to specify any direct output information, they are largely of use in mathematical proofs.

2. **Existential** - For each domain argument x, a unique image y_f **can be proven** to exist. For example $e^x : \mathbb{R} \to \mathbb{R}_{>0}$ can be proven to exist, even though few of its image values can be represented directly. Such functions are useful in mathematical proofs (similar to axiomatic functions), but in many instances theories of approximation offer additional content.

3. **Computational** - For each domain argument x, a unique image y_f **can be demonstrated** to exist. For example $2n : \mathbb{N} \to \mathbb{N}$ which (if given enough time and memory resources) can be directly computed for any input n.

Notation

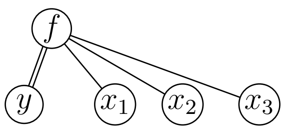
For the purposes of this essay it is ideal to have a notation other than the classical or traditional *Eulerian* form:

$$y \;=\; f(x) \qquad , \qquad t \;=\; f(s_1, s_2, \ldots, s_n) \qquad , \qquad z \;=\; h \circ g(w)$$

Although such notation offers clarity and simplicity[1] in a multitude of ways, there are many patterns it hides[2] which we will want to actively expose to assist us in our analysis.

We will in fact be introducing an alternative notation called **grammatical path** notation, and although it takes up a bit more space on the page it will be pivotal in helping us understand how to decompose general functions in more expressive ways.

To introduce this notation then, let's start with the following function:



This particular example of grammatical path notation is intended to convey the same information as the more classical:

$$y = f(x_1, x_2, x_3)$$

This notation can also be translated into a more succinct textual style by use of what's called **path restriction**:

$$f/n \quad (= x_n\,) \qquad \text{or} \qquad f_{/n} \quad (= x_n\,)$$

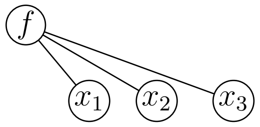
The image y_f of a function is then denoted as $f/0$ or $f_{/0}$. Notice the double line segment in the above, it is reserved strictly for function output values, the "parallel bar" aspect of it is intended to be reminiscent of the equality symbol itself as a reminder of this association.

As an aside, one advantage of this notation over *pathless* notation is when we might want to reference function arguments anonymously. We could write them in the following form:

$$/n$$

which allows us to do so (assuming the context is clear) without naming the function itself.

In practice, it is also often convenient to hide function image arguments altogether:



which works so long as we recall that they're still an accessible part of the grammatical structure. Such a convenience will largely be applied in the remainder of this essay. Finally, any ambiguity as to whether or not an image is hidden in this way is prevented by the double line segment convention.

[1] $f(x)$ style notation not only represents the image value y, it also includes in its representation the information that went into determining y in the first place: f, x.

[2] $f(x)$ style notation visually hides much of the underlying syntactic structure of its composing functions which is valuable in contexts where we don't want to take up much visual space, but because of this it also obscures patterns within the function we might otherwise want to see. Also, $f(x)$ notation doesn't clarify the difference between a function specification and a function implementation, which can become problematic in recursive definitions.

Natural Construction

We now have a basic means of expressing functions in our new notation, but it otherwise says nothing of internal structures, or how to combine them with other functions.

We take the *internal state schema* as our approach: We start with *primitive functions* (assumed to have no internal structure) and we create new functions by means of function **composition**. As this is the most common approach in math and computing literature, I refer to it as *natural construction.*

Function composition in the traditional notation is represented as follows:

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

What does this look like in our grammatical path notation? Let's take our previous section's example:

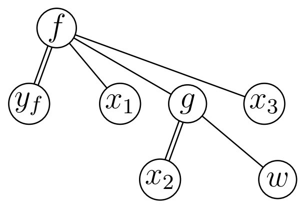
$$f(x_1, x_2, x_3)$$

and compose it with some g such that $x_2 = g(w)$:

$$f(x_1, g(w), x_3)$$

In this case, we extend our grammatical paths to express composition as follows:

$$y_f = f(x_1, g(w), x_3):$$



The idea is we view this visual as a *tree*[3] and whenever we create such a composite we replace the given *leaf* with the appropriate *root node* of the composing function, then we move the replaced leaf to be the composing function's image value. This is to say that in the above x_2 now corresponds with y_g.

Before we head on, let's interrogate this design. There are a few things to consider:

1. Does the associative law hold for grammatical path composition?
2. How does grammatical path composition relate to the three *kinds* of functions?
3. How do we extend the textual translation of grammatical paths?

The quick answer to the first question is: Yes, the associative law holds. When we interpret functions as grammatical paths, they combine to form trees. Any proof that composition holds then reduces to associativity of tree construction using the above rule.

As for how grammatical path composites relate to function kinds? Kinds can be ordered in terms of their utility: Axiomatic have the lowest utility and are the weakest kind. Existential are in the middle, while computational have the highest utility and so are the strongest. In terms of composition, **the kind of the composite** should equate with the kind of its weakest component. If in the above f and g are computational, $f \circ_2 g$ should be computational as well. If either is axiomatic, then the composite is axiomatic, and so on.

Finally, to extend the textual translation for grammatical path notation we use *locator paths.* For example the value x_2 in the above would be denoted as:

$$f_{/2/0}$$

The choice of name "grammatical path" is meant as a hint toward this. Beyond that, such textual translations even allows us to signify the individual functions making up the composition:

$$g \ = \ f_{/2}$$

This of course is intended to be a relevant and user-friendly notational feature in the following.

[3]Such terminology I borrow from Graph Theory, while noting it is also perfectly acceptable to call such a visual a *river.*

infinite functions

A quick note on functions which take arbitrarily indexed input:

$$f(x_1, x_2, x_3, \ldots) \qquad \text{or} \qquad f(\{x_\alpha\}_{\alpha\in\mathcal{I}})$$

Grammatical path notation assumes a slightly different philosophy than common math: If you look at the above two examples of arbitrary function input, you might notice they potentially represent infinite structures, but the representations themselves are still finite. This insight informs our choice here to limit ourselves to finite pathed grammars.

More explicitly: For the purposes of this essay there is no direct need to explore such infinite possibilties—no potential in our theory is lost by restricting ourselves to grammars of finite path sizes. With that said, I should also note grammatical path notation itself does not deny the idea of infinite functions or paths, it's just that there's no immediate need to explore such semantics here.

Decomposition

We have demonstrated the general form of functions effectively as being trees with the content of *types*,[4] and their *instances*, all the while maintaining compositional coherence. With this being the case, we are now ready to look at alternative decompositions.

To start, we would like to properly *modularize* the above mentioned *patterns* as:

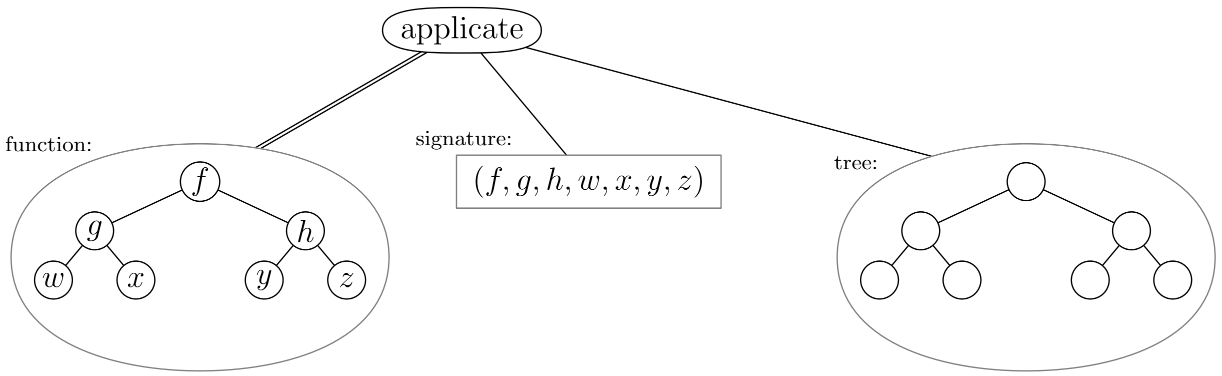
- **tree structures** which can be thought of as the grammatical syntax of our functions.
- **typed signatures** which can be thought of as the grammatical semantics of our functions.
- **compositional enumerations** which can be thought of as grammatical translations from syntax to semantics.

To keep our narrative as clean and minimal as possible we will in fact use a special function to achieve such modularizations. This function is inspired by the notion of *function application* from Church's Lambda Calculus, as such I have named it the **applicate** function. As it is not my intention to formally define this function here, I will instead give an example to build up our understanding.

Let's say then we start with primitive functions f, g, h and combine them through natural construction to form the composite:

$$y \;=\; f(\,g(w,x)\,,\,h(y,z)\,)$$

We can now use the applicate function for our first decomposition as follows:



The idea is there's a recognized analogy to *number theory* which uses multiplication to indirectly decompose a natural number into primes: We instead use applicate to indirectly decompose a general function into a signature and a tree.[5] As for modularizing out a translation pattern: It is embedded in the applicate function

[4]The suggestion here is such content can be framed in the context of Type Theory.

[5]To stretch the analogy a little, such distinct forms could be considered "primes" when it comes to the idea of patterns.

itself, as it is assumed to map the signature to the tree such that it enumerates the tree while preserving compositional coherence.

Actually, the signature in the above is technically incomplete as it requires the necessary type info:

$$(f : E \to F \to G,\ g : A \to B \to E,\ h : C \to D \to F,\ w : A,\ x : B,\ y : C,\ z : D)$$

but since this might be hard to parse visually we may also wish to redisplay it vertically at times:

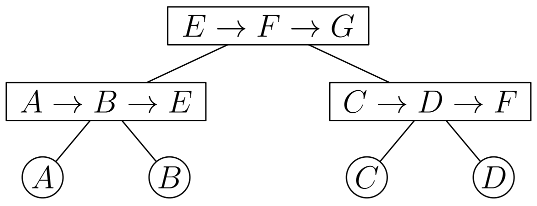
$$\left(\begin{array}{l} f : E \to F \to G\,, \\ g : A \to B \to E\,, \\ h : C \to D \to F\,, \\ w : A\,, \\ x : B\,, \\ y : C\,, \\ z : D \end{array} \right)$$

Otherwise, when the context is clear it might be better to hide this information altogether for aesthetics and clarity—which is why it was done with the above graphic, and why it will be the default policy for the remainder.

This then is the general approach to modularizing out the syntactic, semantic, and translation patterns within our general functions. Finding alternative constructions becomes a matter of recombining these modular patterns in novel ways—different than our natural construction—such that we're also able to recreate the same general functions we started with.

One possible schematic approach is to recognize that since we've isolated syntax from semantics we can then shift a bit of the syntax back onto the semantic side, or we can shift some of the semantics back onto the syntactic side, or a little bit of both. Basically what's being said is we can decompose a function into **fragments** which don't necessarily have clear types of their own, but which may be valuable in expressing and building general functions in new ways.

For example what if we took the type info held within the signature and added it back to the tree structure:

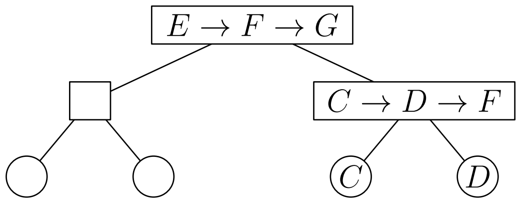


this is a fragment which has a clear compositional coherence. We now only need add specific type instances (respectively) to create a specific function instance. This fragment is such that it can be used effectively to define—also filter—an entire class of functions.

To be fair, this much alone can be represented with classical notation:

$$(A \to B \to E) \to (C \to D \to F) \to G$$

but there are other fragments which are better represented in grammatical path:



One implication of such a fragment is the interpretation that there's no contradiction if we wanted to add new nodes to the unspecified part. The meaning of such an unspecified part then is it will eventually represent a function which has *at least* two arguments of its own. This is in contrast to being required to have exactly two arguments. Classical notation can handle this style of expression as well, but having hidden the structural aspects, such lines of reasoning start becoming more awkward and less user-friendly in the long run.

Representation

The idea of function representation is to decompose our general functions even further than a basic *applicate decomposition* to recreate the more traditional Eulerian presentation: $f(x)$. We do this to show the connection between the two, and to put the Eulerian style on a more rigorous footing.

Let's continue with our previous section's example, but refine its signature:

$$(\, f, g, h, w, x, y, z\,) \quad \longrightarrow \quad (\,+,\ \times,\ \times,\ x,\ x,\ y,\ y\,)$$

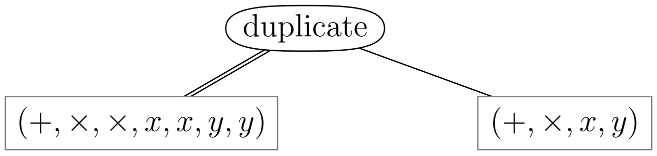
We are now working with the sum of squares function $(x^2 + y^2)$. So if that's the case, how then do we decompose this function beyond the basic signature and tree?

I'm willing to claim the tree component is as abstract as it's going to get for the purposes of this essay, so let's focus on our new and improved signature: The idea is to simplify, but instead of breaking it up into simpler parts, we're trying to minimize the signature as a representation.

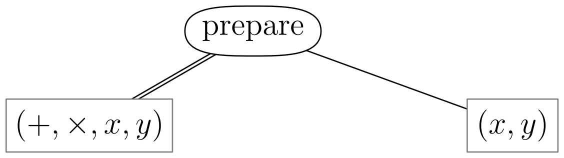
To this end, we might first observe in the above signature that there's some redundancy which we could reduce:

$$(\,+,\ \times,\ \times,\ x,\ x,\ y,\ y\,) \quad \longrightarrow \quad (\,+,\ \times,\ x,\ y\,)$$

so only distinct values remain. As with the applicate function, we actually achieve this using a function that does the reverse. In this case we would recreate the original signature by storing the reversal information in what's called a **duplicate** function:



After this can we simplify any further? Notice the values x, y are variable, but the operators $+, \times$ are fixed; static; constant. We can simplify our signature further by encoding these constants in a **prepare** function allowing us to hide them:



this way we can now represent our sum of squares function as:

$$\text{sum-of-squares}(x,y) \ := \ [\,(x,y) \ \mapsto \ \text{prepare} \ \mapsto \ \text{duplicate} \ \mapsto \ \text{applicate}\,]_{/0}$$

the convention here being the result of applicate is in grammatical path form, in which case the final output value is navigated as /0. Notice, we have now also represented the sum of squares function in what would be considered the traditional Eulerian form:

$$\text{sum-of-squares}(x,y)$$

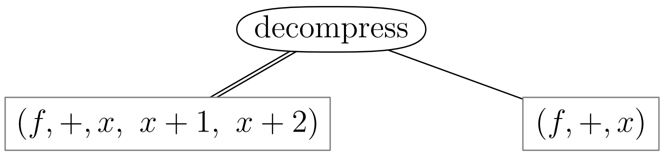
Let's try another one. What if we have a new signature, but of the following distinct values:

$$(\, f,\ +,\ x,\ x+1,\ x+2\,)$$

in this case we should be able to compress the distinct values based on the variable x:

$$(\, f,\ +,\ x\,)$$

To achieve this in reverse, we would need a **decompress** function:



From these examples and many others I've observed that if you decompose general functions in these ways to obtain their representations they tend to follow a common *construction pattern*:

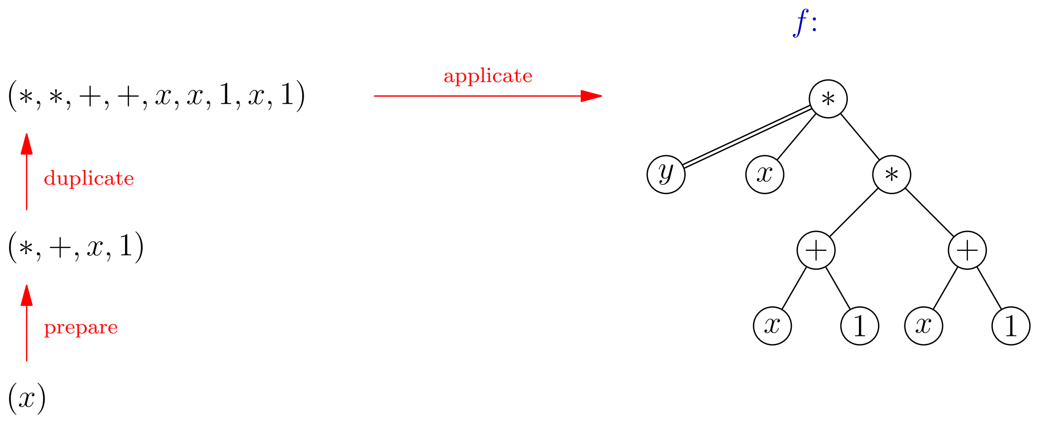
$$\text{prepare} \mapsto \text{decompress} \mapsto \text{duplicate} \mapsto \text{applicate}$$

which demonstrates a universal property when it comes to function representation.[6]

The idea is we always start with a **facade signature** which we extend with a prepare function, thus creating the next-step signature. At times we might decompress this signature, otherwise defaulting to an identity function. At other times this is followed with a duplicate function, again defaulting to an identity otherwise. Either way, beyond this we always end up with an **application signature** which we pass to our applicate function finally creating our desired grammatical path function.

So now that we're comfortable with decompositions and how representations work, let's use this construction pattern to implement a grammatical path function. In particular, the following two examples show how the same algorithmic pattern can be implemented in two different ways, the first being called the **canonical construction**:

$$y = f(x) = x(x+1)^2$$

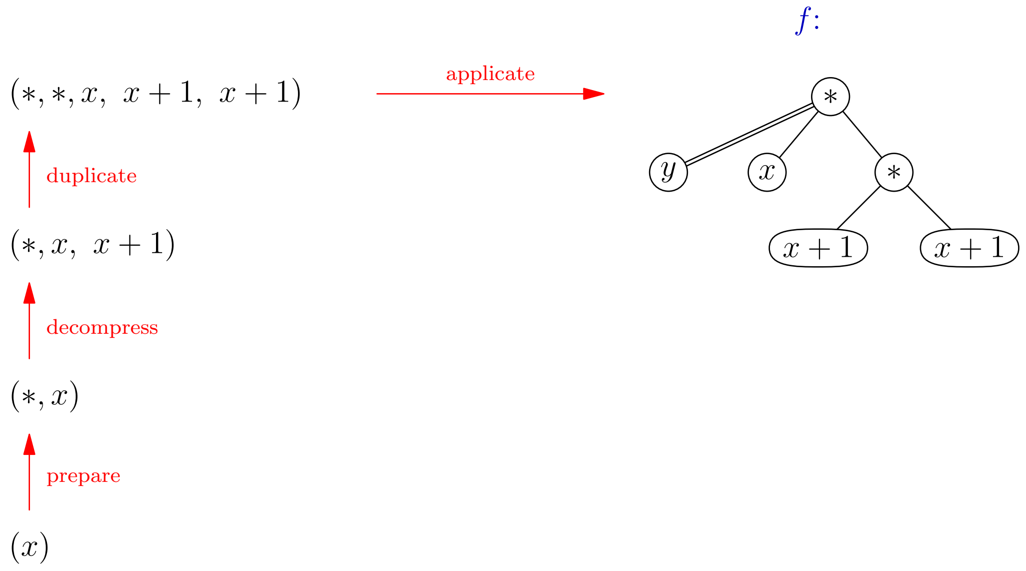


This is *canonical* in the sense that the tree component of the grammatical path notation is of maximum size respective to the (style of) implementation.

The second construction differs because we shrink the tree component: We shift some of the syntax over to the semantic signature side which allows us to compress where we previously couldn't within the canonical version. From a computational perspective this is more (processor) efficient as we make fewer redundant calculations:

[6] Actually, the matter is a bit more subtle than this. If in our theoretical designs we wished to maintain narrative purity, the signature as its own type would be implemented as a tuple, and later as tuples of tuples for the sake of currying. In this case then, we could not actually use the Eulerian form until we had first built the tuple type, or even the list type which overlaps.

$$y = f(x) = x(x+1)^2$$



if you're familiar with *functional programming* languages such as LISP or its variants, you'll notice the decompress function is effectively the same as using the *let* keyword allowing local storage of values as variables—often for the purpose of reducing redundant calculations within the function body itself.

Recursion

We've now demonstrated the overall idea of how to build general functions in ways other than natural construction. It's time to complete this narrative by returning to recursion here in this section and currying in the following, to show these can be expressed similarly.

For the most part constructions and decompositions and reconstructions work the same here, but there are two problems that arise in the recursive context which otherwise don't for naive composition: *circular definitions*, and *ill-defined evaluations*. Furthermore, both Eulerian and grammatical path notations are susceptible.

problematic definitions

Let's give the following example to start us off:

$$n! \quad := \quad \begin{cases} 1 & \text{if } n = 0, \\ n \cdot (n-1)! & \text{otherwise.} \end{cases}$$

This is a definition for a computational form of the *factorial* function, which is intended to have the natural numbers $\mathbb{N}$ as domain. We use this definition all the time, but if you think about it, it doesn't strictly make sense: Our factorial function doesn't yet exist, but we're still using it to define itself.

This is the circular definition problem. How do we solve it?

As it turns out, our understanding of general functions is currently enough to mitigate this. In particular, it's where the different *kinds* of functions previously introduced come into play—notably the interplay between *computational* and *existential* kinds will be instrumental in properly defining computational recursive functions.

Before we continue I want to say that although we have everything we need, the interpretation and line of reasoning in the following is still a bit subtle. There are in fact three subtleties we need to go over, and for this purpose a case study would be helpful.

Now, although we could continue with the factorial function as our assisting example, for this subsection let's instead explore the well known *Fibonacci* numbers:

$$\mathrm{Fib}_n \;=\; \mathrm{Fib}_{n-1} \;+\; \mathrm{Fib}_{n-2} \qquad , \quad \mathrm{Fib}_0 \;=\; \mathrm{Fib}_1 \;=\; 1$$

With that in mind, the first subtlety of interpretation in defining recursive functions is to recognize that mathematicians do it by describing *equality relationships* between values of a recursive function's arguments. On the upside this approach allows for much flexibility in the identity manipulations that mathematicians prefer, but this is also problematic: Such definitions are not *constructive*, they only describe a relationship that *exists*. They are specifications, not implementations, yet many such functions are actually intended to be computational.

The second subtlety of interpretation—in the context of our Fibonacci example—is to realize that the recursive function

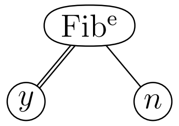
$$\mathrm{Fib} : \mathbb{N} \to \mathbb{N}$$

can be thought of as both existential and computational. To distinguish the two, I will write the existential form as:

$$\mathrm{Fib}^{\mathrm{e}} : \mathbb{N} \to \mathbb{N}$$

leaving the unscripted form as defaulting to computational. Either way, this recursive function *Fib* is readily proven to be existential based on *mathematical induction* or any other similar foundational style of proof.[7]

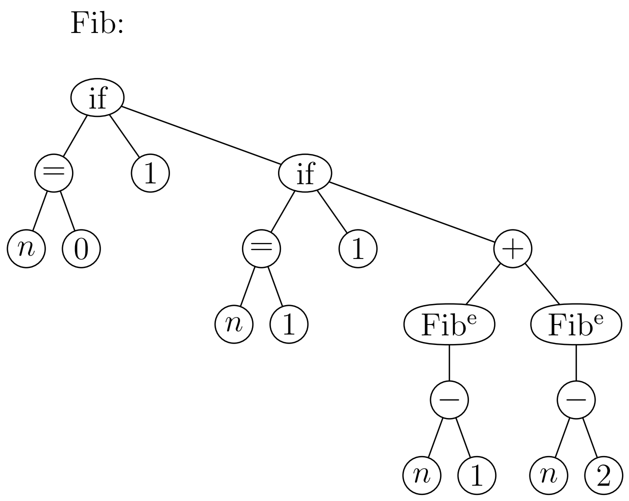
Accepting this existential form (or verifying it yourself), we can now introduce it in our grammatical path notation:



This is necessary to move us away from the traditional notation, unfortunately it is also nothing more than a symbolic function at this point—it has no internal structure or details to help us demonstrate that it is in fact computational as well, or that such a computational form can be expressed with grammatical path notation.

This brings us to the third subtlety of our recursive interpretation: To avoid circular definitions, we will need to build a valid computational function while making no assumption it is equal to $\mathrm{Fib}^{\mathrm{e}}$. The idea is we avoid a circular definition by implementing the computational form without recursively calling the computational form. Unfortunately because of this there's no reason to assume any computational form we come up with is in fact the same function as the existential form, so we'll need to prove their equality after the fact.

With these subtleties in the clear, let's continue by defining our computational Fib **candidate** in grammatical path form:



[7]There are several models I am aware of depending on the use of modern Type Theory or even Category Theory.

From here we can factor out the application signature as:

$$(\text{if}, =, \text{if}, =, +, \text{Fib}^\text{e}, -, \text{Fib}^\text{e}, -, n, 0, 1, n, 1, 1, n, 1, n, 2)$$

If you find such a signature overwhelming don't fear, so do I. We can simplify this and alleviate our fears a little with a *deduplication*:

$$(\text{if}, =, +, \text{Fib}^\text{e}, -, n, 0, 1, 2)$$

By getting rid of the redundancy this is much more reasonable to work with. In any case, if you'll notice, there's only one variable—all other values being constant, so we can save another step and reduce straight to our facade signature:

$$(n)$$

So far so good as this lines up with—rather does not contradict—the facade signature of the existential form:

$$\text{Fib}^\text{e}(n)$$

To summarize so far: We have just shown that our computational candidate is a grammatical path function which decomposes the same way as other functions, and that its facade signature matches the existential form. From here, all that's left is to show using mathematical induction or some other means that Fib^e and our candidate are in fact equal. This would then leave us to conclude that recursive functions also follow these alternative constructions.

As an aside, what does this mean in terms of actually computing recursive functions this way? Our interpretation says that we start with the defined computational form $\text{Fib}(n)$, expand its signature accordingly, and use the grammatical path form to determine the output. Because of the recursive definition, at some point we will be required to call the existential form as part of the calculation, and that's okay because we know it exists! In which case we would then substitute that form again for the computational form, and continue on, and so on, and so on, until halting conditions are reached and the computation is complete.

problematic evaluations

Let's now return to our factorial function:

$$n! \quad := \quad \begin{cases} 1 & \text{if } n = 0, \\ n \cdot (n-1)! & \text{otherwise.} \end{cases}$$

and look at how the evaluation of such a recursive function can be problematic.

We could run through the evaluation using a few values to get a feel for the situation. In doing so, we would eventually discover the problematic nature comes when we try and use the above definition to evaluate *zero factorial.* In this case, we would start by substituting 0 for each occurrence of n:

$$0! \quad := \quad \begin{cases} 1 & \text{if } 0 = 0, \\ 0 \cdot (0-1)! & \text{otherwise.} \end{cases}$$

From here we would want to reduce, but as you can see there ends up being a $(-1)!$ expression which otherwise isn't defined. As we now have an unforeseen problem, we may first try to solve it by extending our domain to include all integers $\mathbb{Z}$, but then we run into an infinite loop—which I don't think is the behaviour we sought to define in the first place.

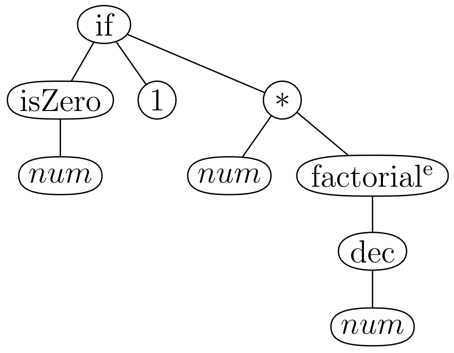
In hindsight we can solve this problem (next time around) by recognizing and preventing it before it happens: We can—using the sophistication of denial—ignore the false representation $(-1)!$ and work around it. As humans we are clever *computers*,[8] unfortunately the other non-human computers we often rely on may not be so. In such cases, for the sake of our non-human computers, it may be preferred that we come up with a new definition for our recursive function, one which avoids these evaluative issues.

[8]Humans were the first "computers" and mathematicians still are human computers—evaluating functions and other mathematical expressions on our chalkboards, on paper, etc.—and so at times we take our nuanced understandings of *computation* for granted. Electronic computers on the other hand require a higher level of rigour and clarity in the policies set for them when they evaluate their functions. In particular, one subtle issue that arises for computers is the *order of evaluation* of a function's arguments.

I should mention before continuing—now that we're aware of this problem—that the previous section was actually a bit of a cheat: We had gotten around circular definitions by separating existential and computational forms, but if you'll look back at the Fibonacci function and its computational form, it actually suffers the same ill-defined recursive call when evaluating at zero. I neglected to mention this at the time so that we could focus on the subtleties of circular definitions without introducing another subtlety still. Once we've dealt with this evaluative issue, you should be able to go back and redefine the *Fib* function accordingly.

Continuing, let's turn our *factorial* function into the case study of this section, and figure out how to properly implement it. Here is its initial computational form:

$$\text{factorial}(num):$$

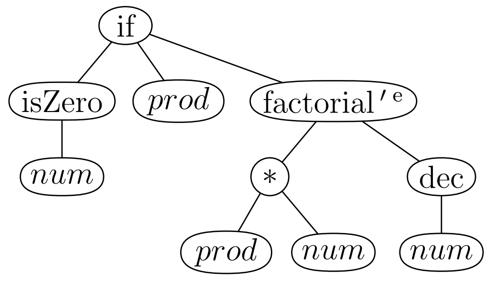


Unlike in the *Fib* example I have condensed some of the component functions:

$(n = 0)$	as	isZero(n)
$(n - 1)$	as	dec(n)

So, the above implementation is perfectly fine, but to be honest I myself would prefer to work with the following alternative:

$$\text{factorial}'(prod, num):$$



the relationship between the two being as follows:

$$\text{factorial}(num) = \text{factorial}'(1, num)$$

Either way, we still run into the same evaluative problems as seen with the classical notation. Between both notations, we have a common issue. What is it?

If we really go into the computational philosophy this problem arises due to the **applicative order evaluation** policy we implicitly assumed when evaluating composite functions. If you're unfamiliar with this idea, think about how you might evaluate one of our previously introduced functions if handed to you:

$$f(x_1, g(w), x_3)$$

The applicative order policy says that we would first evaluate the input terms $x_1, g(w), x_3$ before passing them to f in our evaluation of the composite.[9]

Let's go over in detail what specifically happens with the applicative order approach in our factorial$'$ function example when evaluated at zero. The body of our function would first require us to evaluate the following expression:

$$\begin{aligned}&\text{if} \ (\\ &\qquad \text{isZero}(num), \\ &\qquad prod, \\ &\qquad \text{factorial}'^{\,\text{e}}(prod * num\,, \text{dec}(num)) \\ &\ \ )\end{aligned}$$

and since the policy says we need to evaluate the input before the main function *if*, we would along the way need to evaluate:

$$\text{factorial}'^{\,\text{e}}(prod * num\,, \text{dec}(num))$$

As such, $num = 0$ would reduce to us evaluating:

$$\text{factorial}'^{\,\text{e}}(0\,, -1)$$

To reiterate, not only do the semantics of our intended function become wrong ($prod = 0$), but on the next iteration our conditional test becomes isZero(-1) which will evaluate to *false* and only ever continue to return *false* as we keep decrementing ad infinitum, thus our infinite loop.

What can we do about this? To solve this problem we have to time the evaluation of particular values at particular grammatical locations just right, and to do this we need to delve into the idea of **normal order evaluation** just a little bit.

To discuss this policy, we first need to introduce the following three functions:

apply, **delay**, **force**

The apply function takes another function along with associated input and simply applies the function to said input:

$$\text{apply}(+, 1, 2) \ = \ +(1, 2) \ = \ (1 + 2) \ = \ 3$$

The delay operator on the otherhand takes the same input and leaves it as an inactive *suspended signature*:

$$\text{delay}(+, 1, 2) \ = \ (+, 1, 2)$$

Finally the force operator takes a suspended signature as input and evaluates:

$$\text{force}((+, 1, 2)) \ = \ +(1, 2) \ = \ 3$$

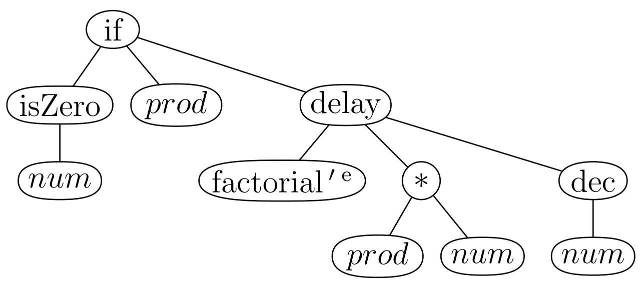
As there is a natural relationship between these functions, it is hopefully no surprise that normal order evaluation or *lazy evaluation* as it is also called is achieved by decomposing the apply operator as follows:

$$\text{apply} \ = \ \text{delay} \ \mapsto \ \text{force}$$

Assuming these operators, we can now reimplement our recursive factorial function in steps:

[9]Given that x_1, x_3 aren't functions the policy further states that any such non-functions would by default "evaluate" to themselves.

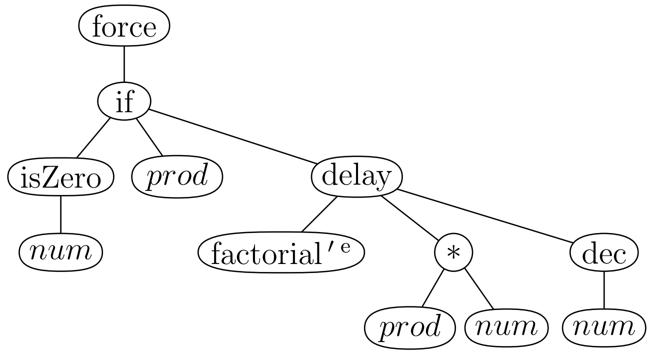
factorial$'(prod, num)$:



In this first step the *delay* function converts its input into a suspended signature, so now if we call this function with $num = 0$ our previous infinite loop problem disappears: When we go to evaluate the third argument of the main *if* function we run into a suspended signature which is a non-function, so we “evaluate” it to itself. That’s the first step.

The second step comes from realizing that in solving one problem we’ve created another: When we evaluate this modified factorial, we are now returning a suspended signature some of the time, which is no longer the desired output. In this case we need to now force this object after it is returned:

factorial$'(prod, num)$:

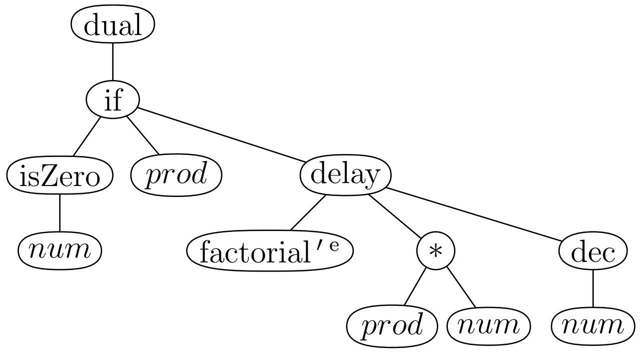


This works until we realize our conditional *if* will return a number value on its halting condition, and since this is not a suspended signature—there is nothing to force—our call to *force* will sometimes be incorrect. Solving this is the final step.

To this end, there’s more than one approach actually. The first is to replace the numerical output *prod* with its own suspended signature $(\text{id}, prod)$ where *id* is the identity function, in which case it is now the correct input for force and when applied it just returns the number *prod* which is what we want.

The second approach is to define a convenience function I call **dual** which accepts alternate (coproduct type) input: If it’s a suspended signature (matching a predefined type) it dispatches to the appropriate force function, and if it’s not it dispatches to the required identity:

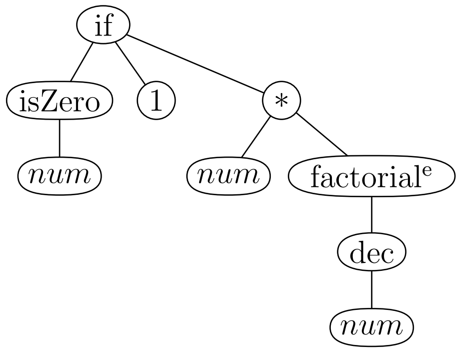
factorial′(*prod*, *num*):



Basically these approaches do the same thing in their behaviour, but I find the *dual* approach allows for more intuitive implementations during function construction. Having multiple (id, *value*) terms can be tedious, especially within nested conditionals.

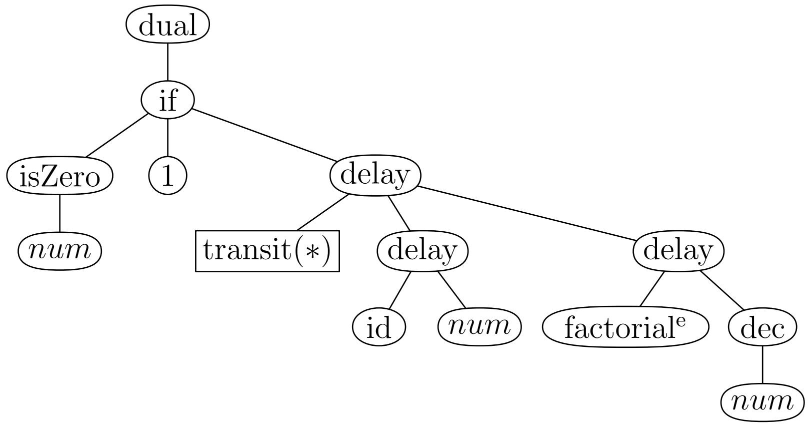
That's pretty much it as far as mitigating this particular evaluation problem goes, but if this approach is to carry any weight we should also be able to use it to reconstruct our initial factorial implementation:

factorial(*num*):



This certainly is the case, but it takes a bit more care than the other form:

factorial(*num*):



The extra care required comes from the need to maintain compositional coherence as we now are delaying a few function calls, turning them into suspended signatures.

The logic here is that we would start by delaying the recursive call

$$\text{factorial}^{\text{e}}(\text{dec}(num)) \qquad\qquad \text{a.k.a. } \text{factorial}^{\text{e}}(num-1)$$

but as it is composed with multiplication ($*$) this composition no longer makes sense. Multiplication doesn't work with suspended signature input. In this case we can convert our '$*$' operator to transit($*$) which is a version of multiplication that only takes suspended signature input. The transit operator itself converts any given function into a modified form that takes only suspended signature input. Upon evaluation such transited functions force the suspended signatures and then pass their returns to be the input of the initiating function:

$$\text{transit}(*)(a,b) \;=\; *(\,\text{force}(a)\,,\;\text{force}(b)\,)$$

Next, with transit multiplication our *num* input (the first argument) no longer makes sense so we need to change its form to (id, num). Finally, if you think about it our transit multiplication actually ruins the whole thing because it automatically evaluates the delayed objects defeating the point to begin with. We fix this by delaying one more time, to which this *cascading* delay effect finally halts.

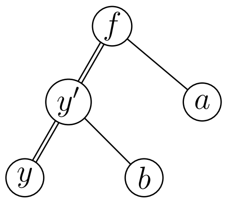
To end this line of reasoning, I will claim it's not much of a stretch then to extend this

delay / transit / dual

approach to general recursive functions.

Currying

Recursion was the special case of function composition where we replaced argument values with other functions. Currying then is the special case of composition where we replace the image value instead:



What are the implications here? In the above case, you may notice the root function f returns a function y',[10] in the classical notation this would be:

$$y' \;=\; f(a)$$

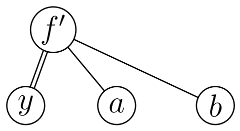
This returned function could in fact be the image value we seek, but we are also left with an argument b which can be evaluated if we so choose:

$$y \;=\; y'(b) \;=\; f(a)(b)$$

In doing so we would obtain the final image value y. To reiterate what we've done here: We first evaluate f at a, returning a function y', which we then evaluate at b. What this means is because there's still only one final output, we could view these otherwise segmented input as a single unit:

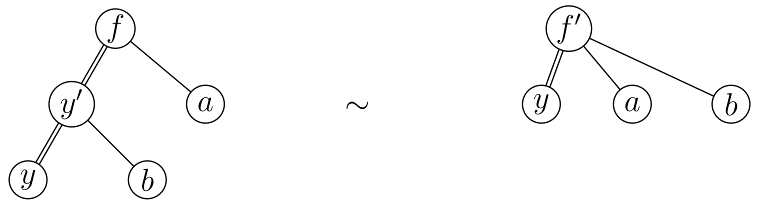
$$(a)(b) \;\sim\; (a,b)$$

So if we were to then streamline these arguments into a single function we could represent it as:



[10]Some readers might protest this notation could be confused with the *derivative* function from calculus, but using primes to indicate slight variations are a common convention in math literature, and since this essay isn't about calculus I see no confusion arising.

This brings us to the main idea of currying, that for each *curried* f there exists an uncurried f' which is equivalent:



This of course lines up with the traditional notion:

$$f : A \to (B \to C) \qquad \sim \qquad f' : A \times B \to C$$

With that said, there are two subtleties to consider in the acceptance of currying as a universal property of grammatical path functions:

- Our **applicate** function becomes a bit less straightfoward with currying. Originally, it would map a signature to a tree—enumerating and preserving compositional coherence along the way. This remains true, except the idea of enumeration is now problematic: For example a standard *left-to-right* tree parsing algorithm works for uncurried functions, but for curried ones the same enumeration algorithm reverses the order of the arguments: $(b)(a)$ instead of $(a)(b)$, which in general is not what we'd want.
- The necessity of relating the signature (a, b) with $(a)(b)$ as being equivalent expands the idea of what a signature is: Up until now we could consider a signature as a tuple, but now it becomes more like a tuple of tuples. This is in fact why I've (mostly) avoided associating a signature with the idea of a tuple throughout this essay—it is meant to be conceptually distinct, being a container of function semantic info.

Before we finish, there is one more pattern in the context of grammatical path notation worth mentioning which could be considered a form of currying.

In previous sections, we've taken for granted that we could shift the weight of what a path points to, for example:

$$2/1/1 \;=\; 2/(1/1)$$

This is what allowed us to shift syntax to the semantic side of the decompositions in previously discussed functions. This property of grammatical paths is called **right associativity**.

Conclusion

We have now reached the end of this essay.

Although I feel the ideas presented here are theoretically relevant and stand on their own, I would like to leave off by mentioning the major motivation behind them.

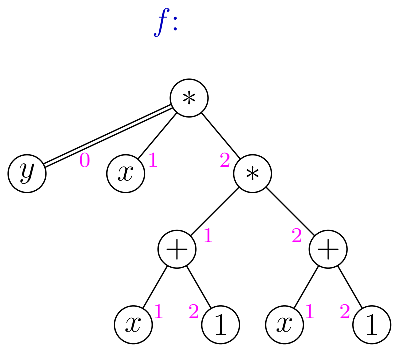
My intention beyond this essay is to build my own programming language which fits what I would consider a new genre: Languages which allow for much more expressive and nuanced constructions of functions, among other types (and their fragments). It should be noted that such a genre of languages would take as its foundation a theory other than Set Theory, Category Theory, or Type Theory for its universe. Given this, it was necessary to analyze in detail the semantics of what functions even are, hence this essay.

I call this alternative foundation to computing (or possibly even math) **concept theory**. The idea is that everything is a *concept*. Certainly in this theory there are many connections and a great amount of overlap with the other foundations—so many people have done so much to build them, and with so many amazing ideas and contributions how could I not build on such legacies? The difference in concept theory then is the philosophy of design and narrative construction of its universe.

To end on this note, let me translate one of the previously defined functions as a basic demonstration:

$$y = f(x) = x(x+1)^2$$

$$\begin{tikzcd}
(*, *, +, +, x, x, 1, x, 1) \arrow[r, "\text{applicate}"] & {} \\
(*, +, x, 1) \arrow[u, "\text{duplicate}"] & \\
(x) \arrow[u, "\text{prepare}"] &
\end{tikzcd}$$



In a concept theoretic implementation, this would be:

$$\text{concept } f \;=\; \left\{ \begin{array}{lcll}
\text{path} & : & 0 & \\
\text{(syntax)} & & 1 & \\
 & & 2 & \\
 & & 2/1 & \\
 & & 2/1/1 & \\
 & & 2/1/2 & \\
 & & 2/2 & \\
 & & 2/2/1 & \\
 & & 2/2/2 & \\
 & & & \\
\text{target} & : & 1 & \text{as} \quad x_1 \\
\text{(translation)} & & 2/1/1 & \text{as} \quad x_2 \\
 & & 2/2/1 & \text{as} \quad x_3 \\
 & & & \\
\text{sifter} & : & x_1 \;=\; x_2 \;=\; x_3 & \\
\text{(semantics)} & & &
\end{array} \right.$$

Keep in mind this is not a full description as it only shows the non-trivial details. A more complete description would include *target* associations describing the fixed portions of the definition such as:

$$\text{target:} \quad 2/1 \qquad\qquad \text{as} \quad +$$

Otherwise, the strength of concepts are their ability to modularize syntax from semantics from translation.
Thank you.