On the semantics of mathematical functions

Daniel Nikpayuk

October 26, 2019

The intention of this essay is to demonstrate a universal property of mathematical functions that parallels the "subsetting" paradigm from Set Theory:

$$\{\, x \in A \mid P(x) \,\}$$

Here we can view the set A as a *model*, and the predicate $P(x)$ as a *filter*. The idea being presented then is that a function can be similarly decomposed into a model component followed by a filter component.

what is a function?

Before I can demonstrate the claim that *all* functions can be decomposed we should agree on terms, namely the idea of what a function even is. As there are alternative foundations to math these days there are different ways to implement this idea of a function. Regardless of the details, each approach tends to retain the following styles of notation:

$$f : A \to B \qquad\qquad f(x) \qquad\qquad f(x_1, x_2, \ldots, x_n)$$

and their respective connotations. In vague terms this notation tells us that a function is something which takes one or more specific inputs and equates it (them) with a single specific output.

What else can be said about functions in otherwise general terms?

For starters, we can categorize them as what I would call *strong* functions and *weak* functions. The idea of a strong function is its output value can be computed from its input value. It might seem obvious or apparent that we would want all functions to be strong in this sense, but there is in fact a lot of math out there where this is not the case, so we can't take the difference for granted.

Weak functions aren't computable, but for them to be meaningful would should at least be able to compute a *proof* that they satisfy the basic specification of a function[1]: Exactly one output exists for each input. As far as weak functions go this is generally all that's needed of them, but in practice many such functions also retain enough logical properties to derive truths about the function's domain and codomain respectively. A function is after all meant to represent a relationship between the two.

In this instance of function semantics, this sort of pattern shows up in math all the time that we give it a name: *functor*. As I am looking to keep things context agnostic (as much as possible) I will refer to these patterns as weak functions as well. As for a quick example of such a weak function:

$$e^x : \mathbb{R} \to \mathbb{R}_{>0}$$

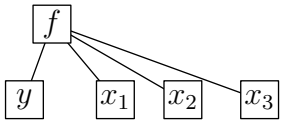
Strictly speaking there are many input values where this is not computable, though certainly they can be approximated, and otherwise such a function can at least tell us things about the relationship between $\mathbb{R}$ and $\mathbb{R}_{>0}$.

[1] Even this requirement of a proof can be weakened as mathematicians also *assume* functions axiomatically, for example the Axiom of Choice in Set Theory.

alternative notation

Before continuing with further intuitive specifications of the nature of a function, for the purposes of this essay I will here present an alternative (complementary) notation for functions than the one given above.

I present to you what I call *grammatical path* notation:

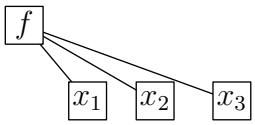


such a representation is intended to convey the same information as

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

Not too much will be said about this style of notation other than it's meant to show relationships more clearly in a visual way, the tradeoff being it takes up more space on the page. Not only can it show relationships more clearly, in fact a particular use compared with *pathless* notation is when we're wanting to reference function arguments anonymously (which tends to be done from time to time). In this case we can *path restrict* the argument with the following notation: f/n $(= x_n)$. Furthermore, the image of a function y could be then also be denoted anonymously as: $f/0$.

In practice when representing functions this way it is often more convenient to hide the $f/0$ argument altogether:



which works so long as we recall that it is still an accessible part of the grammatical structure. This convenience will largely be applied in the remainder of this essay.

This *hiding* convention does bring up one potential problem though: In the case the image is not actually hidden, how do you know whether the leftmost path represents the function image or an argument? To answer this, it is usually clear by context alone, but when it's not we can signify the difference by a change in color or a double line or double surrounding box, etc. I'm leaving the exact specification open so as to maintain a user-friendly design, but the point is it's not a difficult problem to solve.

lambda inspiration

How else would we describe the connotations of a function?

So far they equate input(s) with an output, and are strong or weak. From here we take strong functions as our muse. So what of these computable functions then? Alonzo Church introduced the lambda calculus back in the early twentieth century as a means of describing such functions. Briefly, and to translate into modern terms, his idea is that with

1. **lambda abstraction** you could construct new functions.
2. **lambda application** you could construct new *function objects*.
3. **lambda reduction** you could evaluate function objects, and as a corollary you could evaluate functions.

These insights offer us a way forward, but if we are to include weak functions we have to weaken his ideas.

The first thing to note is that function construction as well as function object construction tend to be sufficiently robust *as is* to include weak functions already, in this sense there isn't much to be said about them. On the other hand, as grammatical path notation is the means to demonstrating the univeral property alluded to at the beginning of this essay, much of the focus will be on ideas of construction, and how they can be equipped with this alternative notation.

As for function evaluation, given that weak functions aren't computable, this idea itself needs to be weakened as *translation with a halting condition*. If the function isn't computable itself, the relationship it describes between its domain and comdomain can be translated until an intended conclusion (implication) is met, which would be the halting condition. In this sense, "reduction" is no longer an appropriate word.

function construction

If our focus now is function construction, the notion of functions we've been discussing so far can be called *primitive* functions. This is in the sense of what I call the *internal state* schema, a strategy to constructively build objects in math: Start with primitive objects and use rules of construction to combine them into composite objects.

In this case, "composite" is exactly the right word. The way to build new functions from basic ones is to *compose* them:

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

Function composition has different meanings when it comes to strong and weak functions, but both share the property that the composition operator is associative. Otherwise the thing to note is that a strong function composed with a strong function is meant to be strong, while if just one of the functions is weak the composite is necessarily weak as well.

How does function composition relate to grammatical path notation? Take our previous grammatical path function

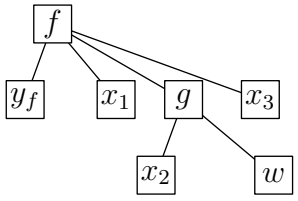
$$f(x_1, x_2, x_3)$$

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

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

As an example of how to represent function composition in this alternative notation we now extend our previous visual:

$$y_f = f(x_1, g(w), x_3)\text{:}$$



the idea is we view this visual as a *tree* and so whenever we create a composite we replace the given *leaf* with the appropriate *root node* of the composing function, and otherwise move the replaced leaf to then be the composing function's image value. As example of this, y_g in the above representation corresponds to x_2.

As a design concern, potentially there's more than one way to show a function composition visually as an extension of the previous *primitive* grammatical path notation, so we should ask: Why choose this approach? We could have swapped the labels g and x_2 in the above graphic, for example.

To figure this out we go back to the idea of strong and weak functions. On the one hand if the composite function is strong each composing function is also strong and thus their images (outputs) are computable. We would need a way to represent this. If on the other hand the composite is weak then at least one of the composing functions is weak and we'd still need a way to represent its output symbolically.

This grammatical path notation allows us to do these things, in particular allowing us to hide x_2 visually, or to represent it anonymously as $x_2\,(= f/2/0)$. Also, this approach to the notation further allows us to reserve the root position $g\,(= f/2)$ for the function name itself.

Besides, if we had these labels swapped as in the suggested alternate, it would break with the existing design.

strong function evaluation

For the most part compositionality is straightfoward, but there is a subtlety worth discussing that's not often brought up with strong function composition: *order of evaluation*. This sort of consideration is usually part of the realm of computing science and compiler[2] design, but it's still something we should discuss here as

[2]The programs that interpret source code and evaluate functions.

the underlying conventions in math that prevent it from being an issue are themselves usually only taught informally, and given that we're constructing functions philosophically it's best to be as explicit as possible here.

Continuing with the previous example, in order to evaluate f it is usually the case we need to evaluate g first, but this is not always true. This consideration shows up in computing science in competing paradigms called *applicative order evaluation* and *normal order evaluation*.

With applicative order we first compute all the arguments that are also functions (g in the example) before passing on to the next function, while in normal order the arguments are passed without first evaluating them. In many programming contexts such arguments don't even end up being used within the body of their function's definition, in which case they're not evaluated at all. It's a moot point when you're only interested in final output, but there are many contexts in which such computational savings are semantically meaningful.

In any case, how or why is such a distinction applicable in a math notation context? The issue arises due to the ambiguity of mathematical function notation: $f(x)$ is meant to represent y (as in $y = f(x)$), but it also demonstrates the relationship between y and x as conveyed by f. The problem then is we can view $f(x)$ instead as the *function object* type (f, x) which has all the information needed to be evaluated, but otherwise is just a data structure holding that information—it is not intended to be evaluated on its own.

So when we interpret and evaluate the function $f(x_1, g(w), x_3)$ this ambiguity allows us to "abuse the notation" and interpret it as $f(x_1, (g, w), x_3)$ and thus pass the object (g, w) unevaluated until it is needed. Another way to look at it if you're concerned about such lax *type casting* is we could instead substitute some symbolic u for $g(w)$ without actually calculating it, and then only calculate it if needed during evaluation of f. Either way, as mathematicians we are using one form in some contexts, and the other form in others, all the while never making these assumptions clear.

With this said, for our purposes it is better to clarify and specify our evaluation policy here as:

applicative order evaluation

As a final note on this matter: Although these evaluation paradigms can compete, the reason we adopt applicative order evaluation is a) it's a simpler starting point, b) we can implement normal order evaluation (by means of function objects) as a specialized pattern within it. By the way, these paradigms also go by the better known names: *eager* (applicative) evaluation, and *lazy* (normal) evaluation, respectively.

currying

Currying is a universal property of functions that effectively says for each function:

$$f \ : \ A \times B \to C$$

there is an equivalent function

$$f' \ : \ A \to (B \to C)$$

such that

$$f(a, b) \ = \ f'(a)(b)$$

As this is an equivalence the implication goes both ways.

To clarify this property in more casual language: A function's arguments can be applied all at once during its evaluation, or one at a time. In the case you evaluate one at a time you're returning a new function to which the remaining arguments are then applied. Given this convention, mathematicians usually drop the parenthesis in the type specification:

$$f' \ : \ A \to B \to C$$

with the understanding that the composition is *right associative* (right-to-left application of parentheses).

In terms of our grammatical path notation, currying basically says the following two functions are equivalent:

As we have already discussed the basics of function composition, this grammatical path notation helps us to understand currying as an *extension of function composition*, allowing for composition not only within the arguments of functions, but also within its image. For clarity, such a composite image might then be called a *curried image*.

decomposition

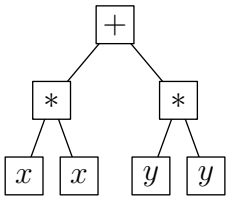
We have enough now to show that any tree structured function can be decomposed into what I term as the *model / filter* schema.

We'll use a "sum of squares" (sosq) function as our primary example:

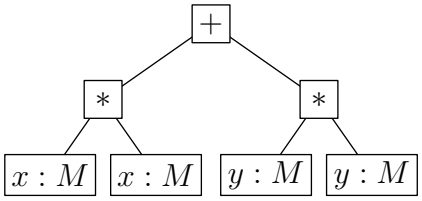
$$\text{sosq}(x, y) := x^2 + y^2$$

Note: As this is only an essay outlining major ideas, no formal proofs will be given regarding any of the following case studies.

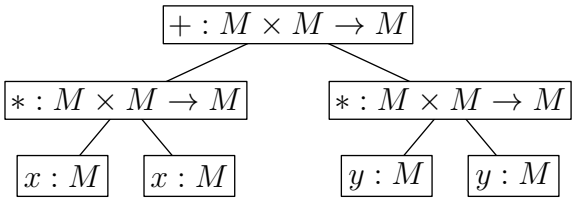
Getting started, the *body* of this function is $x^2 + y^2$, which as a grammatical visual is as follows:



The process of this decomposition starts by determining the modelling function. This is done by generalizing the existing body. In this case, we can start by acknowledging the underlying types of the arguments x, y as well as the functions $+, *$. As it is the intention to generalize, we will simply specify the arguments to have some type M:



but in cases such as these we should specify the function types as well:



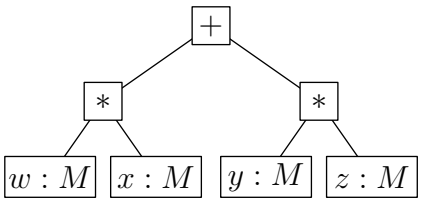
This can be somewhat overwhelming to look at, so if we've previously specified the underlying types and there's no risk of confusion we will instead now *actively* hide such type info when reasonable.

From here the most natural way to generalize toward a model is to free up the restrictions on the input arguments. In the case of sosq we have the following constraints:

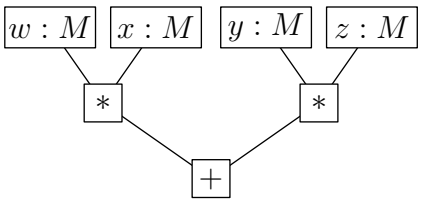
$$\text{sosq}/+^1/*^1 \;=\; \text{sosq}/+^1/*^2 \qquad \text{and} \qquad \text{sosq}/+^2/*^1 \;=\; \text{sosq}/+^2/*^2$$

which is to say $x = y$ and $y = y$. Notice that I've chosen to use the anonymous style notation here to describe these constraints. In order to help aid in reading path directions I've superscripted the given *step* ordinals above the operator names. The anonymous style is a bit more tedious if you're not use to reading it, but it portrays underlying relationships more clearly than simply saying $x = x$.

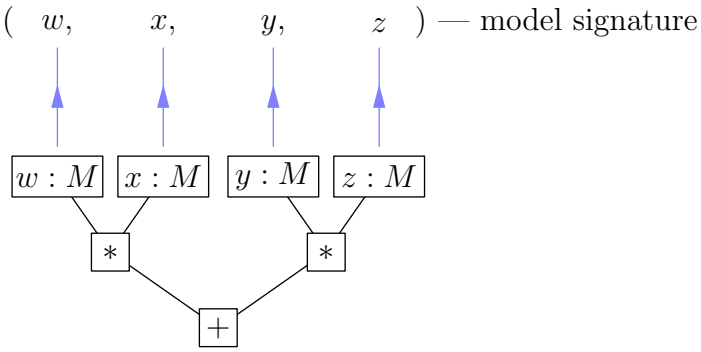
So if we free up this constraint, we can generalize accordingly:



For the time being this is sufficiently general to be our model, so we now turn our focus to determining this model's *signature*. We do this first by flipping our visual:



to which we can then *factor out* sosq's signature:



This is to say our model's signature is:

$$(w, x, y, z)$$

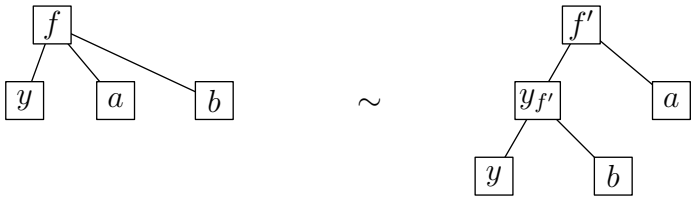
and so then our model itself is defined as follows:

$$\text{model}(w, x, y, z) \;:=\; w * x + y * z$$

By the way this factoring process is effectively equivalent to *lambda abstraction* mentioned previously regarding the lambda calculus. In more general trees it is computed by enumerating the (non-image) leaves of the given body of the function.

As a quick aside, in the case of curried functions this approach to lambda abstraction needs to be modified slightly: The enumeration over the tree used to factor out the signature needs to be done in two parts. First it would step through the non-image leaves, and only then it would go into the image leaf and continue. Greater care is required so we don't inadvertently change the signature's ordering.

For example with initial equivalence of curried functions:



if we enumerated over f' in the non-curried way, we'd end up with a $(b)(a)$ signature, which is not what we'd want.

Getting back to the decomposition, since we now have our model it's time to derive the filter:

$$\text{model} \quad / \quad \text{filter}$$

$$(w * x + y * z) \quad / \quad P(w, x, y, z)$$

With the process used to determine this model, our function's filter then just ends up being a predicate statement that explicitly states the information we had abstracted away:

$$\text{filter}(w, x, y, z) \quad :\Longleftrightarrow \quad (w = x \text{ and } y = z)$$

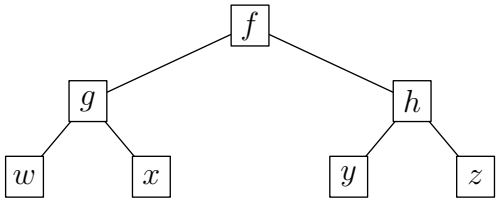
And that's it! We now have the first demonstration that a function can be decomposed into this *model/filter* schema:

body	model	/	filter
$x^2 + y^2$	$(w * x + y * z)$	/	$(w = x \text{ and } y = z)$

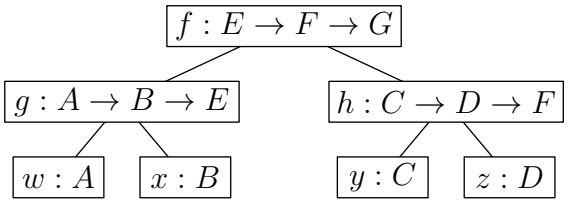
complete decomposition

With the above decomposition we were satisfied with the model component as it was, but the truth is we can still go further to achieve a complete decomposition.

So far our understanding of how to construct functions is derived from the *internal state* schema which first defines primitive functions and then uses those primitives equipped with function composition to define new function combinations. Then again, this grammatical path notation now suggests an extension, or reorientation of this schema. To return to our above example, the function primitives of our *sum of squares function* are the well known monoids $+, *$, but we can now loosen this constraint to arbitrary functions f, g, h:



While we're at it we could free up the monoid *type* we're currently using in place for the domains and codomains respectively:



in standard notation this would be:

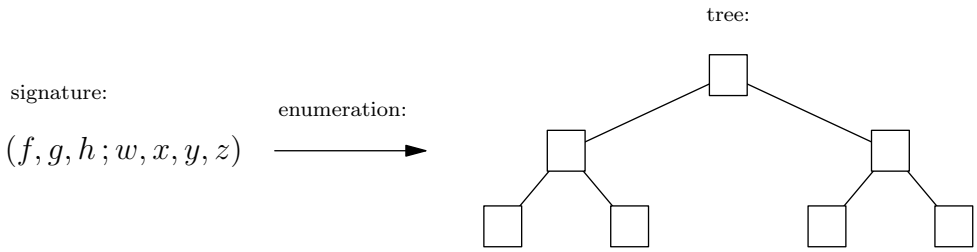
$$f(g(w, x), h(y, z))$$

So here's the thing: The functions f, g, h themselves are no longer fixed or hardwired, they are now variable. The idea then is that a complete model decomposition factors these out as well:

$$\text{model} \quad / \quad \text{filter}$$

$$f(g(w, x), h(y, z)) \quad / \quad P(f, g, h; w, x, y, z)$$

The consequence here is that the model component of this decomposition can itself be decomposed, allowing us to modularize the construction of a function into the following parts:



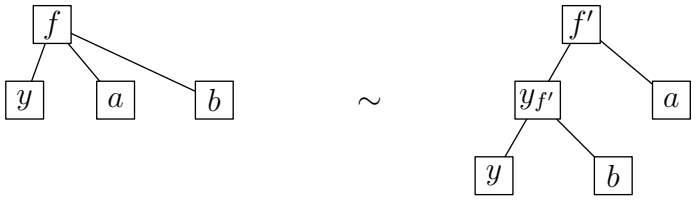
Which is to say a function model can not only be constructed from the *internal state* schema, it can also be constructed as a combination of a signature, a tree, and an enumeration that admits to certain type-semantic properties.

In this extended schema we still need primitive functions, but our means of defining combinations becomes modular, and much more expressive because of it—so much so we can define a type theoretic *induction* operator to categorize all functions possessing a given signature if we so choose:

$$\text{induct}(\mathcal{S}, \mathcal{T}, e, f)$$

where $\mathcal{S}$ is the desired signature, $\mathcal{T}$ the composing tree, $e : \mathcal{S} \to \mathcal{T}$ the required enumeration relating the two, and f the model filter.

Overall induction is straightforward, but due to *currying* we need to take a refined approach with the ideas of *signatures* and *enumerations*. Take the basic currying equivalance for example:



The representative example of a signature is effectively a *tuple*, but this currying equivalence shows when we lambda abstract:

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

we end up with signatures that are also families of tuples, and moreover any given tuple in fact has many equivalances. If we want this induction principle to be expressive, we need to allow for these alternative forms as well.

As for enumerations: Taken as true function primitives (or even meta-functions), they embed the elements of a signature into the nodes and leaves of their given tree. This embedding ends up excluding the leaves that (after-the-fact) become the image leaves of their respective composing functions. Naturally functions are mapped to nodes, while both functions and non-functions are mapped to the remaining leaves, all in such a way that type theoretic composition is satisfied.

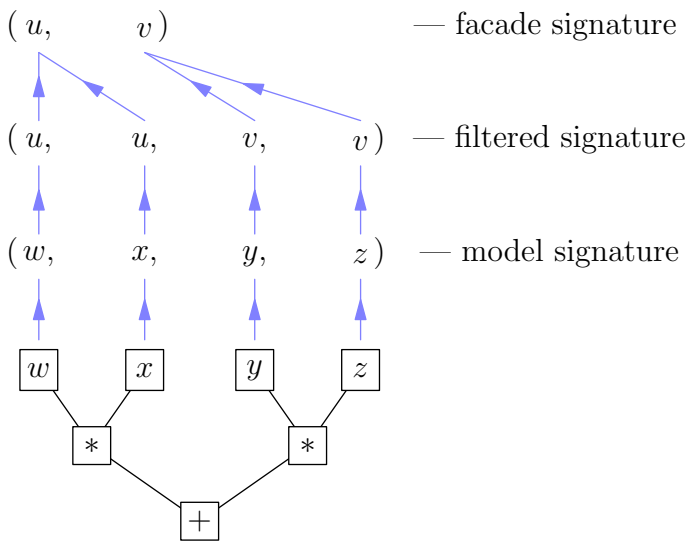
In practice a *left-to-right* parsing algorithm is a reasonable default, which would then cycle twice in the case of curried functions.

model/facade paradigm

There is a special case of the model/filter schema worth mentioning here.

Even though the filter and model of a function have the same signature the filter itself isn't a function, it is a logical statement. In practice however the filter does often provide enough information to turn it into a function, and when one exists it is called a *facade*.

From a programming perspective we would effectively be taking the model's signature, filtering it, then refactoring it into its respective facade signature. Looking back at our *sum of squares* (sosq) function this would be as follows:



The facade then ends up decompressing its own signature back into its (filtered) model signature and applies this to the model function itself. Another way to say this: When such facades exist, we can decompose the main function into its facade followed by its model. With our sum of squares this is as follows:

$$\text{sosq}(u,v) \quad = \quad \text{facade}(u,v) \;\mapsto\; \text{model}(w,x,y,z)$$

recursion

Before finishing this section, one last top level class of functions needs to be demonstrated to successfully claim universality of this grammatical path notation, and its model/filter schema. As the subsection title suggests, I am speaking of *recursive functions*.

As in previous sections we will explore a case study here. In particular let's look at the well known Fibonacci numbers:

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

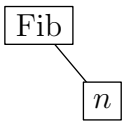
The thing to note about the current approach in defining recursion is that mathematicians do it by describing *equality relationships* between values of a recursive function's arguments (the above equality as example). On the upside this approach allows for much flexibility in the identity manipulations that mathematicians prefer, but this is also problematic: Such definitions are not *constructive*, they only describe a relationship that *exists*. This is to say the details are missing when it comes to actually computing the number values of such functions.

I intend here to give an outline of actual constructive approaches. With that said, the underlying interpretation is actually quite narrow and subtle, so we will walk through this line of thinking carefully. We start by asking: How do we translate this equality relationship that *specifies* a recursive function into a grammatical path representation that *implements* such a function?

The first subtlety of interpretation is to realize that the recursive function

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

exists as a weak function, based on mathematical induction, using the above equality relationship in its proof. Accepting this assumption (or verifying it yourself), we can now introduce it *by name* in the following grammatical path notation:



but this is little more than a symbolic function, it has no internal structure or details to help us calculate it. Intuitively we know we can do better by demonstrating a strong (computable) function as well.

The second subtlety comes from the need to show the grammatical path representation of a recursive function can always be decomposed into a model and facade. This is subtle because we can't actually show a decomposition without knowing the implementation details of the (strongly defined) function to begin with. For this reason the technical approach is to define the

$$\text{model}\ ,\ \text{facade}$$

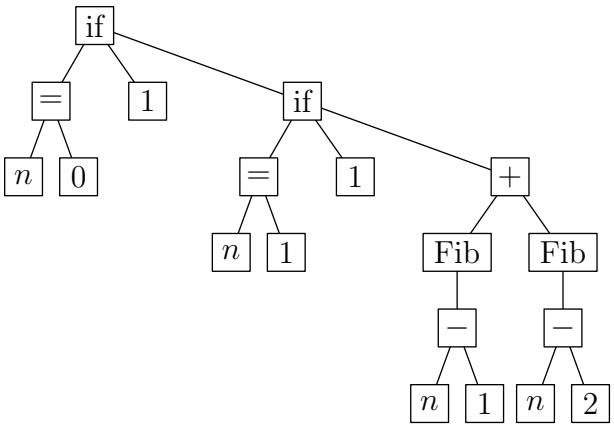
functions independently of Fibonacci, then show their composition to be equivalent to it:

$$\text{Fib} \quad \sim \quad (\,\text{facade} \ \mapsto\ \text{model}\,)$$

If we can do this it would follow naturally that these *model* and *facade* functions are the model and facade components of Fib itself—thus proving strong recursive functions also follow the model/filter schema.[3]

Let's start by defining what will be the *model* component of Fib. The thing to realize about recursion is that it is actually just a specialized form of function composition, and since we know $\text{Fib} : \mathbb{N} \to \mathbb{N}$ exists we can define our model as:

model:



from here we can factor out the complete signature as:

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

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

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

By getting rid of the redundancy this is much more reasonable. In any case, we can do better still: Notice the only non-constant element of this signature is the variable n, in this case then we can (holding all other input constant) refactor our facade signature to a minimal version as:

$$\text{facade}(n)$$

This lines up with our desired $\text{Fib}(n)$. From here it is relatively easy to show with mathematical induction that Fib and this model/facade combo are equal. As such, the interpretation for computing this function is to start with $\text{Fib}(n)$, substitute our respective implementation

$$\text{Fib}_{\text{fac}}(n) \ \mapsto\ \text{Fib}_{\text{mod}}(\ldots)$$

and evaluate until you arrive at the terms:

$$\text{Fib}(n-1)\ ,\ \text{Fib}(n-2)$$

in which case you recursively apply again, and again, and $[\ldots]$, until halting conditions are reached.

By the way, in the above the ellipsis of $\text{Fib}_{\text{mod}}(\ldots)$ is there to hide the clutter of the signature we otherwise already know.

[3] Weak recursive functions follow almost identically to this approach.

applications

filter swapping

One of the reasons we modularize a function into its model/filter form is because it allows us to swap things out cleanly and otherwise reuse parts.

Returning to one of our above examples: Looking at the sosq model as being independent of sosq itself, what other functions can we resolve from it? For starters we could swap out the filter to narrow the function arguments:

body	model	/	filter
$wx + 1$	$(w * x + y * z)$	/	$(y : \{1\}$ and $z : \{1\})$

The difference with this filter and the original sosq one is that with $wx + 1$ we are narrowing the model's arguments independently of each other, while with $x^2 + y^2$ we're relating the arguments to each other. In practice any such filter will be some combination of both.

Looking back at the Fibonacci model, many of its arguments were constants. I didn't make it explicitly clear at the time, but we were taking this narrowing approach there as well.

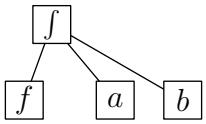
Riemann integration

As an example of how more general decompositions can be practical, let's look at how we'd represent the following intended function:

$$\int_a^b f(x)\,dx$$

Here the following would be sufficient as our model:

$$\text{model}(; f, a, b):$$



where

$$f : \mathbb{R} \to \mathbb{R} \qquad a : \mathbb{R} \qquad b : \mathbb{R}$$

keeping in mind we could encode this more directly as:

$$\text{model} \;:\; (\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{R} \to \mathbb{R}$$

Our filter follows as:

$$\text{filter}(; f, a, b) \quad := \quad f \text{ is Riemann Integrable over the interval } [a, b]$$

Although we can claim this is a filter in a general sense, we shouldn't take for granted that such a logical property alone actually encodes a valid function every time. So far I'm being casual about the filter property "Riemann Integrable", but given that it is a well researched concept within formal calculus / real analysis I'm comfortable saying we can use the specifics of its definition to show that this model/filter combination in fact satisfies a weak function.

Finally, in the general case it is not a strong function given that it is real valued, but there is still enough detail in its definition regarding *Riemann sums* that we can at least define approximation functions which are themselves fully computable.

decomposition lattices

So far this modelling property of functions might not seem especially novel, but it shows its some of its strength when we look closer at functions with facades. Notably, any given facade (which is a function itself) might be decomposed into its own facade and its own model:

$$\text{facade } r \quad = \quad \text{subfacade } r \ \mapsto \ \text{submodel } s$$

here r, s are the respective signatures of these functions. The reason the submodel has a different signature than the (sub)facade is to reinforce the idea that such signatures are implementation specific and may in fact differ. With decomposable facades we can in theory now split the main function into three:

$$\text{function } r \quad = \quad \text{subfacade } r \ \mapsto \ \text{submodel } s \ \mapsto \ \text{model } t$$

In practice some decompositions end up being trivial, but many do not. Functions that allow for non-trivial decompositions increase function expressivity and even offer options when creating function inventories: Which functions occur most often and are best suited as primitives or near primitives? Alternative options allow for greater flexibility in design, and even the potential to compress existing dependency narratives.

The reason this works is because we now have variability of the input signature when it comes to determining the model component:

$$\text{function } r \quad = \quad \text{subfacade } r \ \mapsto \ \underbrace{\text{submodel } s \ \mapsto \ \text{model } t}_{\substack{\text{function expressivity happens here in} \\ \text{terms of alternative representations}}}$$

The signature t for the most general model is fixed, but with facade decomposition there ends up being a lattice of signatures s to respecify the model input. As such, when choosing a model we may now also be more selective of its signature.

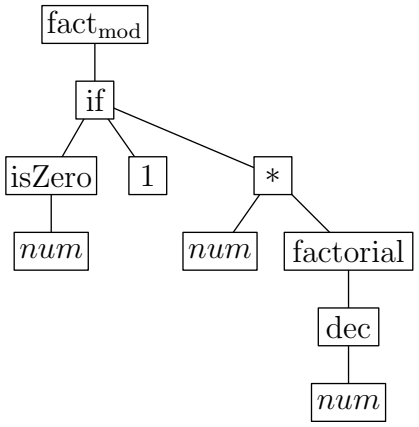
recursive evaluation

We're pretty much done at this point, especially regarding *constructive* aspects of functions, but I think it's important and instructive to briefly look at an *evaluative* aspect of functions that is often neglected, and to which this grammatical path notation is well suited to help explain.

When recursion was introduced previously we went over the subtlety of interpreting the construction of recursive functions, in particular the need to define their existence before we could define actual implementations. As it turns out, when it comes to the evaluation of recursive functions there are similarly subtleties in how to correctly interpret them, necessary to prevent things like infinite loops. These are considerations to which mathematicians seldom discuss (if at all), so I thought I would end this essay on such a note.

Getting into it now, I had previously mentioned that function recursion is actually just a special form of function composition, so let's elaborate on that. Our example this time will be the *factorial* function:

factorial(num):

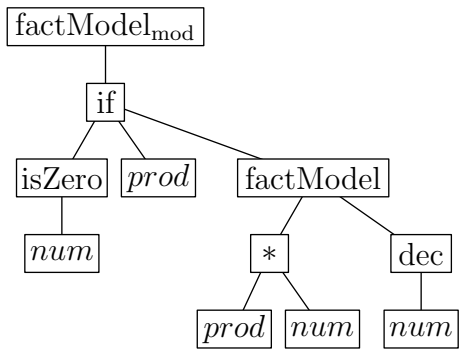


Here unlike in previous examples I have condensed the functions

$$(n = 0) \quad \text{to} \quad \text{isZero(n)} \qquad \text{and} \qquad (n - 1) \quad \text{to} \quad \text{dec(n)}$$

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

$$\text{factModel}(prod, num)\text{:}$$



where we implement factorial itself as:

$$\text{factorial}(num) \;:=\; \text{factModel}(1, num)$$

Either way, we can construct a recursive function by means of function composition, and at first glance it appears we can evaluate such constructions as is—we certainly did previously—but this isn't the whole truth: The use of function composition here is *cyclical*, and as a consequence any practical interpretation requires us to also mitigate the possibility of the semantics breaking down as we attempt our evaluation—the aforementioned infinite loop problem.

This problem arises due to the *applicative order* policy we previously assumed when evaluating composite functions. As a reminder, when evaluating a composite function:

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

the policy states that we first evaluate the input terms $x_1, g(w), x_3$ before passing them to f in its evaluation. As an addendum: Given that x_1, x_3 aren't functions the policy further states that any such non-function would by default "evaluate" to itself.

The problem with the applicative order approach can be demonstrated with our factorial function example. Let's look at what actually happens when our num input equals zero: Within the body of our function we would need to evaluate the following expression:

$$\begin{array}{l}\text{if (} \\ \quad \text{isZero}(num), \\ \quad prod, \\ \quad \text{factModel}(num * prod\,, \text{dec}(num)) \\ \text{)}\end{array}$$

but before we could evaluate the main function *if* we would need to evaluate its input. In doing so we would need to evaluate

$$\text{factModel}(num * prod\,, \text{dec}(num))$$

which given $num = 0$ would be

$$\text{factModel}(0\,, -1)$$

Not only are the semantics of our intended function wrong ($prod = 0$), but on the next iteration our conditional test becomes isZero(-1) which will now always return *false* as we keep decrementing ad infinitum, thus our infinite loop.

So 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 normal order evaluation just a little bit. Lazy evaluation as it is also called is achieved by decomposing the apply operator as follows:

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

If you're unfamiliar with these functions, the apply operator takes a function, and some input, and then evaluates:

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

whereas the delay operator takes the same input and converts it to a *function object*:

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

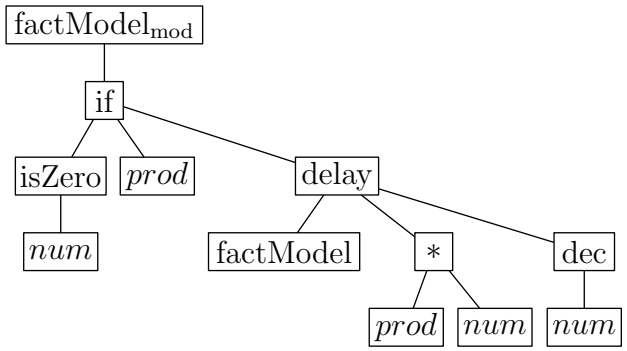
and finally the force operator takes function object input and evaluates:

$$\text{force}((+, 1, 2)) \;=\; +(1, 2) \qquad \text{which in common notation is } (1 + 2) \;=\; 3$$

Again, I'm being casual here, if we were to be rigorous we'd have to consider these operators to be polymorphic (dependent function types), and within a type theoretic framework we'd tediously have to determine the types required to make the semantics of these functions valid. If we want to remain casual but infer a basic level of rigor, we can instead conceptualize these operators as entire families of functions.

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

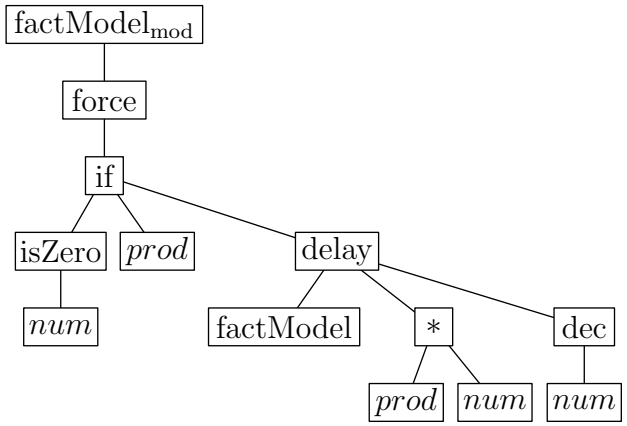
factModel($prod, num$):



In this first step the *delay* function converts its input into a function object, 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 function object which is a non-function, so we "evaluate" it to itself.

That's the first step. The second step comes from realizing that when we evaluate this modified factorial, we are now returning a function object some of the time, which it is no longer the desired output. In this case we need to now force this object after it is returned:

factModel($prod, num$):

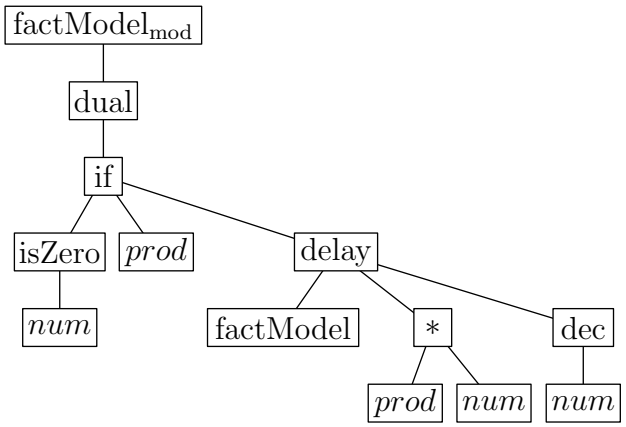


This works until you realize our conditional might instead return a number value, and since this is not a function object—there is nothing to force—our call to *force* would now have incorrect input. Solving this is the final step in this case.

There's more than one approach actually. The first is to replace the number output *prod* with its own function object $(\mathrm{id}, prod)$ 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, where if it's a function object (matching a predefined type) it dispatches to the appropriate force function, and if it's not it dispatches to the required identity:

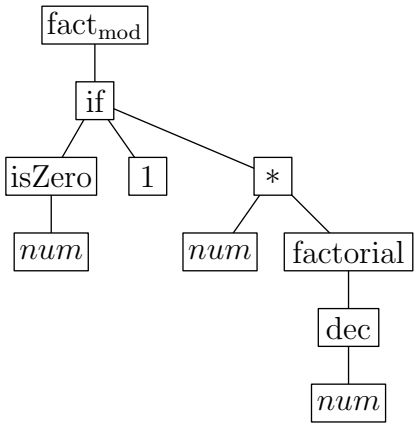
factModel$(prod, num)$:



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 $(\mathrm{id}, value)$ terms can be tedious, especially within nested conditionals.

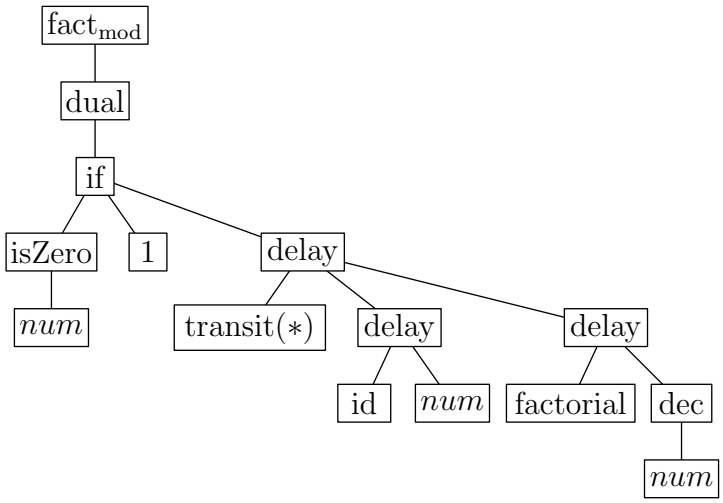
Finally, does this approach work for the first implementation of our factorial function?

factorial(num):



The easy answer is *yes*, but it takes a bit more care than the second form did:

factorial(num):



The extra care required comes from the need to maintain proper composition semantics as we now are delaying function calls and turning them into function objects.

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

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

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

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

We're getting there.

Now that we are using 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 then fix this by delaying one more time, to which this *cascading* delay effect finally halts.

To end here, I will add that this delay/transit/dual approach applies to general recursive functions.

summary

I presented a lot of ideas in this essay, so I thought it fitting to summarize them.

To start, we have asked what a function is, and what its major connotations are in the traditional literature. Beyond this, a function has input, output, can be strong or weak, can be evaluated as eager or as lazy, can be constructed as primitives and combinations of primitives, can be curried, can be constructed into a function object, can be evaluated as a function object, can be recursive. It has been shown that a grammatical path representation can do all this by means of its model/filter schema, to which its model can be decomposed into a signature, tree, evaluation, and its filter can sometimes be refactored into its own function called a facade.

As mentioned earlier this essay is more about exposition than proof, but with that said I claim the demonstrations here did offer outlines to their respective formal proofs. As for the claim that this decomposition is a universal property of functions, it holds under the assumption this grammatical path notation is able to represent *all possible functions* which is in fact a meta-mathematical claim. The best we can do is show it is at least as potent as existing models of math: Set Theory, Category Theory, Homotopy Type Theory, etc., which I think is reasonable.