Grammatical Elements of Function Induction

Daniel Nikpayuk

March 10, 2020

Abstract

In this essay I give an informal walk-through in how to build functions that build functions, and the grammar that allows for such possibilities.

This is of theoretical importance as it provides a point of direction toward general induction operators for function types. It is also of practical importance as it offers a way to create inventories of variant[1] functions without having to code each function manually.

Philosophy

In Modern Type Theories there's a special function for each *type* called the **induction** function, which is also called *dependent elimination*. As a rule, it effectively tells us what kind of functions are even possible for its given *type* definition, but we can also think of it as the function used to build other functions acting on the given type. The philosophy of this essay takes inspiration from this formal version of induction and generalizes it as a concept to include pretty much any function that defines other functions.

The question then becomes: How do we construct functions that construct functions? In a very broad sense, looking at programming languages, we use grammar to build functions, so it would make sense that our focus here should be on function building grammar.

Beyond that, the major idea pushing the plot of this essay forward, so to speak, is that an **interpretation** is the common thread between evaluating a function and constructing that same function. This is to say: If we can express how a function is evaluated, we can express how it is constructed. Informally, the only major difference between the two is that we *read* the interpretation when evaluating, while we *write* the interpretation when constructing.

This of course is philosophy, not something that can be proven here. At the very least, it is still enough to motivate the following methodology.

Methodology

Under the premise that an interpretation is the abstract intersection between function evaluation and function construction, the strategy in discovering *construction grammar* is in fact to focus on isolating function evaluation grammar.[2] Although not identical, the translation should be relatively straightforward from there. This is also a practical approach given how much literature already exists regarding interpreters, evaluators, compilers.

Generally speaking this is the main methodology, unfortunately there are a few complications: Ideally we would like to go straight to known powerful versions of our intended construction grammar, but we quickly run into what I would call **the signature problem**. Otherwise, our approach is to introduce basic versions of our construction grammar first—ones which don't have the signature problem—then use them to mitigate the signature problem. After that we can finally build the powerful versions of the grammar we seek.

[1] Sometimes *variant* just means optimized variations.

[2] Another way to look at this: We are effectively translating automatic into manual.

As for the signature problem, we start by implementing function signatures as tuples. This means we will want to look more closely at tuples themselves and how to build them. Notably, it's easier to define constructors for tuples of a fixed length, but less so for ones that allow for a few different lengths. For example, what if we want to be able to create various subtuples of a given fixed tuple? This is the core of the signature problem, and solving it allows us to extend our construction grammar to greater potential.

In detail, the approach in this essay can be outlined as follows:

1. We introduce construction primitives (apply, delay, force, transit, if), along with conditional compositions (coapply, stems, binds).

2. We introduce function signatures and implement them as tuples, further requiring us to introduce tuple primitives (cons, car, cdr, eq?).

3. We extend the conditional compositions by solving the signature problem, also showing how more complex functions can be built.

Beyond this, in terms of algorithmic analysis I will use what's called **grammatical path theory** and its notation. I have researched and written this up in [1]. With that said, there is no formal requirement for the casual reader to have this prerequisite, and for that reason I have largely kept grammatical path theory explanations at an intuitive level—providing basic explanations when needed.

nonlinear combinations

An induction function is one used to create all other functions for a given type, so if we now consider all such functions for their given type as a *space*, how would we go about constructing and representing its elements?

The traditional approaches are through **modelling**: Either as emergent combinations which we'll here call *primary* models, or submodels of those primaries, which we'll call *secondary* models. For example, regarding emergent combinations, we have linear combinations within *vector space theory*:

$$a_1 \mathbf{v}_1 \ + \ \ldots \ + \ a_n \mathbf{v}_n$$

which represent all vectors in the *span* of a basis. Notably this representation is considered to be a **closed form**. To contrast this, we can take a primary model such as $\mathbb{R}^2$ and subset it to derive the submodel known as the unit closed disk:

$$\mathcal{D} \ := \ \{ \ (x, y) \in \mathbb{R}^2 \mid x^2 + y^2 \leq 1 \ \}$$

My claim is that in theory we can represent the elements of a type's function space as either a primary or secondary model, but in practice both have their strengths and weaknesses and are better suited for different contexts.

We'll start with the secondary submodelling approach: It is better suited for function design. It's one thing to build a function, but we have to know what to build in the first place—which often means we need to design then translate that design into a construction. I should mention that using submodelling for design is possibly my own bias, but I find having a single umbrella function as a model in this context really does help clarify its design.

As for the primary emergent combination approach[3]: Due to the desire for performance it is usually a better fit when actually implementing our function construction. To emphasize this point—and in analogy to linear combinations where nothing is wasted in the construction—we will refer to such constructions here as **nonlinear combinations**. We take the name *nonlinear* because these functions are built exactly in that way—nonlinearly: They are combinations by means of function composition.

The other advantage to taking this approach when implementing function definitions is that such combinations create a coded combinatorial inventory: We know what exactly should be in this space, allowing us to make room for its functions when we're organizing our code library. It also allows us to take a **lazy policy** during implementation, as we are only required to build the functions we need when we need them. Later, down the road if we need more we already have a location for where they should be within our designs.

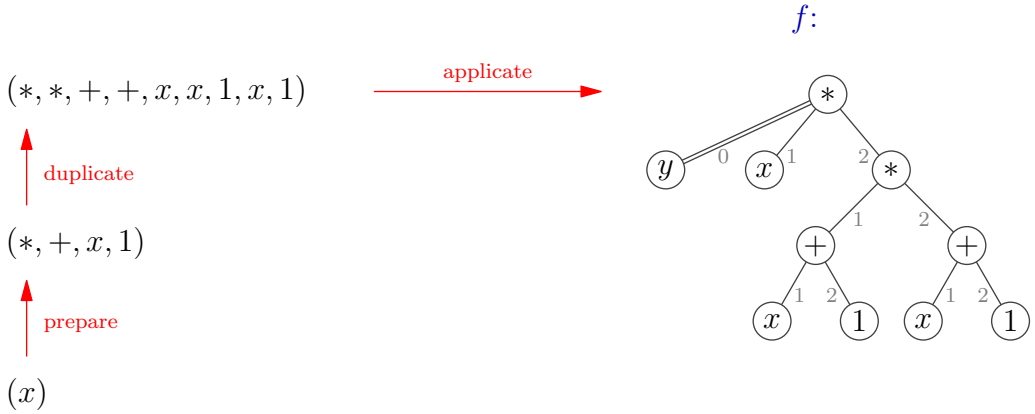
[3]Note, this in fact coincides with the idea of *induction* itself.

divide and compress

Nonlinear combinations allow us to define functions by breaking them up into smaller more manageable functions, but in doing so we actually create a new problem for ourselves: **Compression**, which plays a more active role in function definitions than we might realize.

In fact we compress for two clear reasons: The first is just to make our human user notations easier to read and work with. The following function has a sufficiently complicated grammatical path representation such that we would want to refactor its information and hide any of the details we'd want to take for granted:

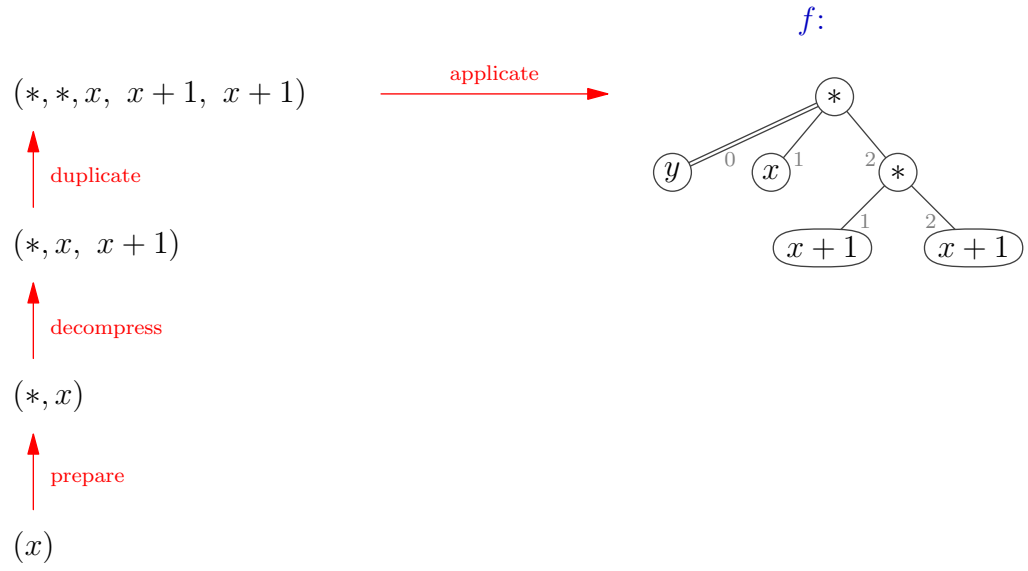
$$y = f(x) = x(x+1)^2$$



This diagram is described as *canonical*, in the sense that the tree component of this grammatical path notation is of maximum size. In anycase, if we go in the reverse direction of the mapping arrows, we're able to compress the function signature, first reducing redundancy, then hiding those values that are otherwise held constant, allowing us to simplify all the way down to (x).

The second reason we compress is for computational performance. If you'll notice in the above diagram there is some redundancy with the $(x+1)$ expressions, which if we calculated separately would be wasteful. In this case, if we were to again refactor the tree into a signature, we could take advantage of this repetition and simplify:

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



So far so good, but how then does compression become an issue?

Compression works best with **complete information**. Generally, we would look at the bigger picture all at once to find those repetitive patterns we'd want to compress. This becomes an issue when we *divide and express* our above mentioned functions through nonlinear combination: By modularizing out simpler functions we create information silos—we isolate our ability to recognize redundant patterns across functions. We can still compress each component function, but our ability to compress overall, along with our compression ratios, will on average lessen. Can we mitigate this issue?

Yes, fortunately. We mitigate this issue by realizing that the **applicate** function above—which bijectively maps the *raw* (uncompressed) signature content to the tree nodes—can be reinterpreted to handle **just in time** compression: We equip our applicate function with the idea of "lazy evaluation" creating a kind of "delayed applicate". It has the same mapping pattern as before, but now it only maps as needed.

The implication in this change is we can now map parts of the signature to the whole tree piece by piece—parts corresponding to component functions used to define the function whole. In this sense we have what I would call a **scope signature**, a component of local memory that spans across each module function. In practice what this means is we can *after the fact* push our compressions—otherwise redundant computations—onto this scope memory to be used across the range of function components.

Conditional Composition

It's time to finally get started. In this section we seek to isolate grammar from function evaluators for reuse within our desired construction grammar.

We begin with the most basic of evaluator primitives:

$$\text{apply}(f, x) \quad := \quad f(x)$$

which takes a function f, and some value x and *applies* them to get their output value. The first thing to realize about this evaluator in terms of how it can be translated into a constructor is to realize that if the value x is the output of another function g then the **apply** operator effectively becomes composition:

$$\text{apply}(f, g(x)) \quad = \quad f(g(x)) \quad = \quad f \circ g\,(x)$$

Although interesting, this exact result isn't quite useful on its own. For one, we may wish to compose two functions without evaluating at a given value, but also, we want more flexibility in our ability to express our compositions. Even still, this apply operator can point us in the right direction. To that end, and to create more effective composition operators, let's go back to apply and decompose it:

$$\begin{aligned} \text{delay}(f, x) &:= (f, x) \\ \text{force}((f, x)) &:= f(x) \\ \\ \text{apply} &= \text{force} \circ \text{delay} \end{aligned}$$

The terminology for **delay** and **force** come from [2], and are otherwise key grammar when implementing **normal order evaluation**—also known as *lazy evaluation*—a style of programming that only evaluates the parts of a function that are actually used.

As we've now decomposed *apply*, all we're missing is the **if** conditional

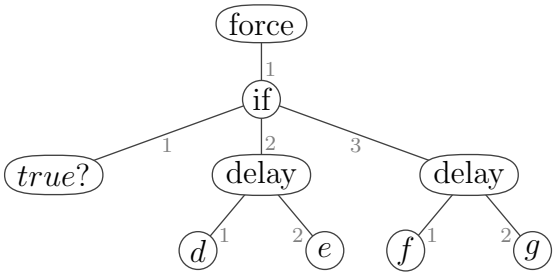
$$\text{if}\,(\,\textit{boolean},\ \textit{antecedent},\ \textit{consequent}\,)$$

to achieve our first point of constructive grammar. Here *boolean* is a truth value, when *true* this function returns the *antecedent*, when $false$ it returns the *consequent*.

We are now ready to introduce our first conditional composition operator.

The coapply function:

$$\text{coapply}(true?, d, e, f, g):$$



If we were to code this in the LISP programming language style, it would be along these lines:

```
(define (coapply true? d e f g)
  (force
    (if true?
     (delay d e)
     (delay f g))
))
```

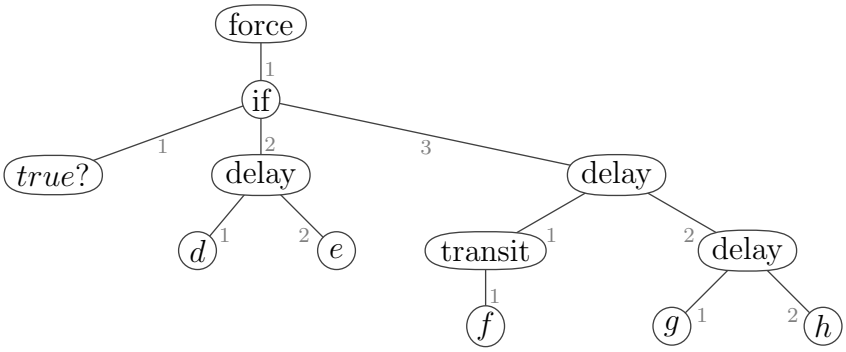
This is a good start, but to evaluate more general functions we will need grammar that lets us not only compose two functions, but several. In the case of such *chain compositions*, we mostly already have what we need, but we do still need to introduce one more function—the **transit** operator:

$$\text{transit}(g) \qquad := \qquad g \circ \text{force}$$

With this we can extend the coapply function to allow for *continuation passing* within the first serious contender for our induction grammar.

The stem function:

$$\text{stem}(true?, d, e, f, g, h):$$



In this case we branch as before, but if the boolean value is false and we return the *consequent* instead of the *antecedent*, we not only force the function and value pair g, h, we also pass them as input to the function f.

In LISP style, and based off our above coapply definition, stem would be coded as:

```
(define (stem true? d e f g h)
  (coapply true? d e (transit f) (delay g h))
)
```

The stem function is interesting in many ways in its own right,[4] but for our purposes here its greatest value is in its ability to chain compose with itself. For this reason, and to simplify notation a little, it is worth introducing **bind** operators.

stem's bind operators:

$$
\begin{aligned}
(\,true?,\ break,\ w,\ next\,) \quad &\gg\!=_{\text{pass}} \quad cont \quad := \quad \text{stem}(\,true?,\ break,\ w,\ cont,\ next,\ -\,)\\
(\,true?,\ break,\ w,\ x\,) \quad &\gg\!=_{\text{post}} \quad cont \quad := \quad \text{stem}(\,true?,\ break,\ w,\ cont,\ -,\ x\,)\\
(\,true?,\ next\,) \quad &\gg\!=_{\text{fipass}} \quad cont \quad := \quad \text{stem}(\,true?,\ cont,\ -,\ cont,\ next,\ -\,)\\
(\,true?,\ x\,) \quad &\gg\!=_{\text{fipost}} \quad cont \quad := \quad \text{stem}(\,true?,\ cont,\ -,\ cont,\ -,\ x\,)
\end{aligned}
$$

The operators here are variations that otherwise let us chain compose stem with alternate instances of itself.

In particular, the **pass** bind operator takes a tuple as left input and a single function *cont* as right input, and simply composes them such that—within the context of stem's meaning—when the boolean value *true*? breaks[5] we have specified stem's output w along with its continuation, aptly named *break*. When on the other hand *true*? doesn't break then *cont* will act as the continuation of some specific function *next* applied to the variable input denoted $-$.[6]

One thing we haven't discussed about general composition

$$f \circ g$$

is that we can hold either functions f, g as constant, and we end up with two specializations: *pre-composition* and *post-composition*. This is where **post** bind comes in: It is identical in meaning to the pass bind, but unlike pass which *passes* the variable, post bind applies a fixed value x to the variable. This of course implies that the variable $-$ signifies a function.

Finally, the **fipass** and **cofipass** are specializations which we will discuss when we introduce the *distem* function, but for the now let's continue with our construction grammar. The biggest value of the bind operators is they allow us to create very clean grammar for conditional chain compositions:[7]

$$
\begin{aligned}
&(\,true?_0,\ break_0,\ w_0,\ f_0\,) \quad \gg\!=_{\text{pass}}\\
&(\,true?_1,\ break_1,\ w_1,\ f_1\,) \quad \gg\!=_{\text{pass}}\\
&(\,true?_2,\ break_2,\ w_2,\ f_2\,) \quad \gg\!=_{\text{pass}}\\
&\qquad\qquad\vdots\\
&(\,true?_n,\ break_n,\ w_n,\ f_n\,) \quad \gg\!=_{\text{pass}}\\
&\ \text{id}\ \ x
\end{aligned}
$$

Hidden within this sequence is a chain composition because when all *break*s fail we end up with:

$$f_n \circ f_{n-1} \circ \ldots \circ f_1 \circ f_0\,(x)$$

Let's figure this notation out a little though as it might not be entirely straightforward. In terms of theory the meaning of this chain can be rewritten as follows:

$$
\begin{aligned}
& & & \text{stem}(\,true?_0,\ break_0,\ w_0,\ cont_0,\ f_0,\ x\,)\\
cont_0 & = & & \text{stem}(\,true?_1,\ break_1,\ w_1,\ cont_1,\ f_1,\ -\,)\\
cont_1 & = & & \text{stem}(\,true?_2,\ break_2,\ w_2,\ cont_2,\ f_2,\ -\,)\\
& & \vdots & \\
cont_{n-1} & = & & \text{stem}(\,true?_n,\ break_n,\ w_n,\ cont_n,\ f_n,\ -\,)\\
cont_n & = & & \text{id}
\end{aligned}
$$

[4] I call it the stem function in analogy to *stem cells* in biology. As a function model it can specialize to several important compositional patterns—such as recursion—known in existing computational theories.

[5] Branches from the intended main continuation.

[6] The meaning of $-$ is that it in fact indicates a new function which is created because even though a stem function is returned from pass, it was only partially applied so we are still left with a function of one variable, the aforementioned $-$.

[7] If you're comfortable with Category Theory, this works because there is a *monad* buried within the continuation passing of stem.

The main idea is we actually start at the end with the identity function id, and compose our chain of binds that way—binds being right associative. We then properly define each

$$cont_n,\ cont_{n-1},\ \ldots,\ cont_1,\ cont_0$$

in sequence, which again in theory should leave us with a single function which in this case takes the value x.

In practice, the strength of this notation is not only that it's theoretically sound, but that it can also be readily optimized: We don't need to start at the end of this chain and compose backwards, the nature of the stem function is such that we can evaluate the conditional at the beginning and branch to the appropriate output. In the case that the *break* fails, the output just continues with the next tuple in line. In fact, given the regularity of the interpretation here we could simplify this whole sequence into the following formal grammatical pattern:

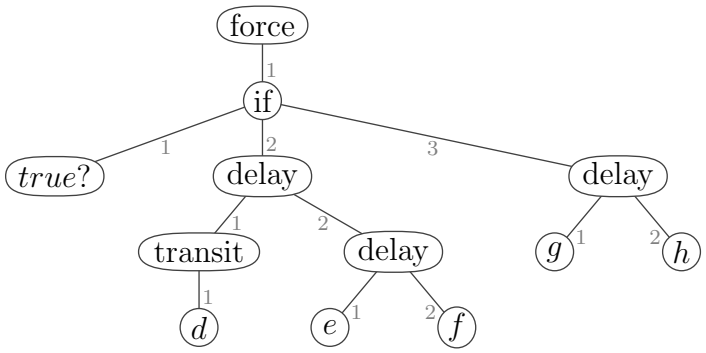
$$\begin{array}{llll} \textbf{pass}\ \ x & & & \\ true?_0 & break_0 & w_0 & f_0 \\ true?_1 & break_1 & w_1 & f_1 \\ true?_2 & break_2 & w_2 & f_2 \\ & \vdots & & \\ true?_n & break_n & w_n & f_n \end{array}$$

which can be readily implemented within several existing programming languages. That is the stem function, and its binds.

Next, and for convenience as well as performance, it is also worth introducing the **costem** function.

The costem function:

$$\text{costem}(true?, d, e, f, g, h):$$



which in LISP would be:

```
(define (costem true? d e f g h)
  (coapply true? (transit d) (delay e f) g h)
)
```

Note that costem could be defined using stem simply by negating the *true*? value, but again, for performance it's worth just defining them independently. Costem also has its own bind operators of course.

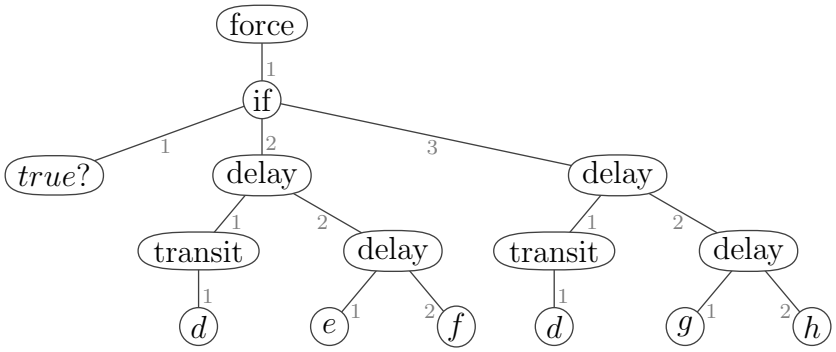
costem's bind operators:

$$\begin{array}{rlll} (\,true?,\ next,\ break,\ w\,) & \gg=_{\text{copass}} & cont & := \quad \text{costem}(\,true?,\ cont,\ next,\ -,\ break,\ w\,) \\ (\,true?,\ x,\ break,\ w\,) & \gg=_{\text{copost}} & cont & := \quad \text{costem}(\,true?,\ cont,\ -,\ x,\ break,\ w\,) \\ (\,true?,\ next\,) & \gg=_{\text{ficopass}} & cont & := \quad \text{costem}(\,true?,\ cont,\ next,\ -,\ cont,\ -\,) \\ (\,true?,\ x\,) & \gg=_{\text{ficopost}} & cont & := \quad \text{costem}(\,true?,\ cont,\ -,\ x,\ cont,\ -\,) \end{array}$$

Finally, we end this section with the **distem** function.

The distem function:

$$\text{distem}(true?, d, e, f, g, h):$$



in LISP:

```
(define (distem true? d e f g h)
  (coapply true? (transit d) (delay e f) (transit d) (delay g h))
)
```

The importance of the distem operator is that it extends *continuation passing* to both sides of the conditional: Instead of the possibility of *break*ing as with stem and costem, it continues no matter what. This allows for *chain branching*, thus increasing our expressivity to build functions greatly. Either way, distem has four bind operators of its own.

distem's bind operators:

$$(true?,\ next_1,\ next_2) \gg=_{\text{dipass}} cont := \text{distem}(true?,\ cont,\ next_1,\ -,\ next_2,\ -)$$

$$(true?,\ x,\ next) \gg=_{\text{lepost}} cont := \text{distem}(true?,\ cont,\ -,\ x,\ next,\ -)$$

$$(true?,\ next,\ w) \gg=_{\text{ripost}} cont := \text{distem}(true?,\ cont,\ next,\ -,\ -,\ w)$$

$$(true?,\ x,\ w) \gg=_{\text{dipost}} cont := \text{distem}(true?,\ cont,\ -,\ x,\ -,\ w)$$

Here *lepost* is short for "left post", while *ripost* stands for "right post". This also means it's not unreasonable to alias lepost as *ripass*, or to alias ripost as *lepass*.[8]

This by the way is the reason **fipass, fipost, ficopass, ficopost** exist—they are technically distem binds, where one of the "next" functions is set to the identity function id. We could implement these cases using distem of course, but for performance—given that id doesn't actually do anything even though we're still required to waste resources calling it—it's better to implement them as special cases of stem and costem instead.

Function Signatures

We've now introduced our basic function construction grammar. How then do we extend it? The most natural way would be to expand upon the input that *true*? accepts. At the moment it's just a boolean truth value, but in practice we might want it to be a function instead:

$$match?(\ldots)$$

[8]Truth be told I'm not fully satisfied with this nomenclature, but as any experienced programmer knows, naming is one of the harder problems in computing.

This is where the signature problem comes in: How do we define the signature[9] of such a function? Truth be told, the signature could be anything we want, and it sounds easy enough to build: We just compose our respective stem function of interest with the *match*? we want. Overall this works, except for when we want the signature to contain stem's other variables:

$$(d,\ e,\ f,\ g,\ h)$$

This is to say: What if we wanted the ability to test the other input of our respective stem function? There's a few ways to solve this. We could encapsulate the whole stem function within another function and set the variable values accordingly, but this is somewhat wasteful. Another option is we could have simply defined our stem function to take the full signature as its input from the beginning, but what if we only want some of the variables:

$$(e,\ h)$$

not all of them. In this case, when implementing *match*? we could still define it with the full signature and simply choose which variables to use and which to ignore, but this is also wasteful, you know?

In some contexts these approaches are perfectly valid, but there are also *mission critical* contexts where every extra computation counts. The point is, regardless of these alternatives, we should at least have the option to be able to further optimize if we choose. Fortunately, the major point of this essay is to show how we can actually create such variations of a function. To that end, let's figure out how to solve this *signature problem* with the tools we've build so far. Before this though, we still need to look more closely at signatures themselves.

In practical theory signatures are usually implemented as **tuples**, where one can consider a tuple of a given length to be the recursive nesting of pairs:

$$(a, b, c, \ldots, z) = (a, (b, (c, (\ldots, (z,\ \text{null}\,)\ldots))))$$

where the final value is always the null tuple, or 0-tuple.

In anycase, we should start by figuring out what tuple primitives are available to us. In fact, in addition to previously introduced primitives, there are four that will be shown as sufficient:

$$\{\,\text{cons, car, cdr, eq?}\,\}$$

three of which are defined as follows:[10]

$$\begin{array}{lcl} \text{cons}(x,y) & := & (x,y) \\ \text{car}((x,y)) & := & x \\ \text{cdr}((x,y)) & := & y \end{array}$$

The **cons**, **car**, **cdr** terminology I borrow from the LISP programming language. In particular, cons is the pair or 1-tuple constructor, car is the first element projection function, and cdr the second element projection function.[11] In coding literature the second element of a tuple is frequently referred to as the *rest* of the tuple following the first element, which is a convention we will use in the following example.

Before heading on we implement[12] a basic "length" function in LISP style grammar with C style comments in order to demonstrate how these primitives can be used:

```
(define (length′ count tuple)          //  define length′
  (if (eq? tuple null)                 //  if tuple == null
   count                               //  return count
   (length′ (+ count 1) (cdr tuple))   //  else (recursively) call
  )                                    //   (length′ (count + 1) rest)
)
```

[9] From the perspective of grammatical path theory, if *match*? accepts signature input, it implies it is defined as a composite function—the expectation is that it takes the signature and expands it before mapping to its given syntax tree. This by the way is a reasonable assumption as most context based boolean functions are built as composite to test their respective contexual information.

[10] The definition of eq? is slightly trickier as it requires recursion, but informally at least we can define it as follows: $\text{eq?}(t_1, t_2) \ :=\ (\text{car}(t_1) = \text{car}(t_2)) \ \wedge\ \text{eq?}(\,\text{cdr}(t_1),\ \text{cdr}(t_2)\,)$

[11] Truth be told, for the purposes of this essay we only actually need **cons** and **null**, but given the importance of tuples here I thought it worth introducing the core basics for those who are unfamiliar either with the theory, or with LISP style nomenclature.

[12] In fact, the eq? function can be implemented this way as well.

By the way, if this version of the tuple *length* function is unfamiliar to you, the more intuitive interpretation is achieved as:

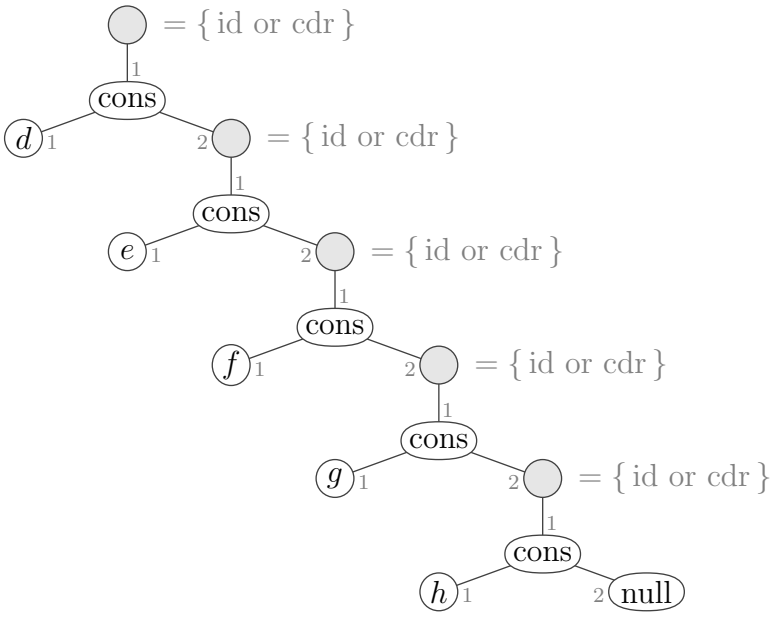
$$\text{length}(tuple) \;:=\; \text{length}'(0, tuple)$$

Getting back to the signature problem, how then do we go about creating the variations of the following tuple:

$$(d,\, e,\, f,\, g,\, h)$$

The variations—thirty-two to be exact—come from whether or not we include particular variables. We need a function where we input whether to *keep* or *drop* particular variables, and it creates the respective tuple for us. Now, if you recall from the methodology section, I discussed the idea that there are primary and secondary models, and that secondary or derived models are better suited for function design. Here is an example of that approach to modelling, applied to this particular tuple function:

signature ($keep_d?,\ keep_e?,\ keep_f?,\ keep_g?,\ keep_h?$):



Here, each *keep*? variable is meant to be set to either *id* or *cdr*, where *id* corresponds to *keep* and *cdr* corresponds to *drop*. You'll notice the inefficiencies in this function definition, there's a lot of unnecessary calculations to achieve the desired effects, but it does offer a starting point.

In fact the above model points out that we're not trying to build the tuples themselves, we're trying to build functions that build these tuples. When we look at it this way, it becomes readily translateable into our conditional composition grammars from before:

define $\text{induct}_{\text{signature}}(\text{keep}_d?,\ \text{keep}_e?,\ \text{keep}_f?,\ \text{keep}_g?,\ \text{keep}_h?)$:

ficopost id

$\text{keep}_d?$	(cons d)
$\text{keep}_e?$	(cons e)
$\text{keep}_f?$	(cons f)
$\text{keep}_g?$	(cons g)
$\text{keep}_h?$	(cons h)
null	

Here the various keep variables are boolean valued, allowing us to directly express our policy for each of the original tuple variables d, e, f, g, h. Otherwise the interpretation is that we test each policy, and if we choose

to keep its given variable we cons it and pass this *curried* (partially applied) function as the next input for the function we're currently building, in effect currying it as well. Notice that each step before the last is building the final function in the same way it would be evaluated if it already existed. This works due to the way *ficopost* is defined.

Before heading on, I should mention if you're wondering whether or not the *id* at the beginning of this chain is wasteful,[13] technically yes, but it's also only one additional computation, and it can't be helped due to the nature of continuation passing. Keep in mind that in practice we wouldn't just be building a signature though, we'd be attaching it to some function, in which case we'd replace *id* at the beginning with that function—very possibly a function that lets us store the respective output in memory as a new variable, assuming the language supports such a feature.

Refined Conditional Composition

We now have everything we need to extend our inductive grammar.

Let's say we wanted to redefine our **stem** function to test one of its input values h:

```
(define (stem match? d e f g h)
  (force
    (if (match? h)
     (delay d e)
     (delay (transit f) (delay g h))
)))
```

We would first implement *stem*'s induction operator:

define $\text{induct}_{\text{stem}}(\text{keep}_d?,\ \text{keep}_e?,\ \text{keep}_f?,\ \text{keep}_g?,\ \text{keep}_h?)$:

ficopost (force if *match*?)

$\text{keep}_d?$	(cons d)
$\text{keep}_e?$	(cons e)
$\text{keep}_f?$	(cons f)
$\text{keep}_g?$	(cons g)
$\text{keep}_h?$	(cons h)
null	

(delay d e)
(delay (transit f) (delay g h))

At which point, we'd simply define our variant *stem* as:[14]

```
(define (stem_h match? d e f g h)
  (induct_stem  drop_d  drop_e  drop_f  drop_g  keep_h)
)
```

One issue with the above I should bring up as we've nearly reached the end of this essay is the **ambiguity** of curried composition. With this *ficopost* construct, we have curried

$$\text{if } match?\ -$$

which is not only one partially applied function, it's two. Let's say we were then to compose some other function f similarly:

$$\text{if } match?\ f$$

Does this mean we're composing f with *if* or with the *match*? function? In our case, as we're sticking to a left-to-right tree enumeration pattern in terms of our curried construction, and our *match*? function only takes one (signature) input, the ambiguity can be resolved.

[13] Since that's a point of design I've privileged throughout this essay.

[14] Mind you, I'm intermixing a few notational styles here, I hope this much at this point can be forgiven.

Afterthoughts

We've reached the end of this essay.

I titled this final section as *afterthoughts* instead of the standard "summary" or "conclusion" one might expect because for me the ideas presented here are still very much the subject of open research. With that said, I wanted to leave off in such a discussion.

First of all, I feel I should note that in the methodology section I had introduced the idea that *compression* is an issue in need of mitigation when constructing functions, and that scope signatures—made possible through delayed application—offers a solution.

Yet, none of the inductive constructions actually made use of scope signatures or delayed application in the above examples. In fact none of them required the need for compression, but this should not be construed as being a poor choice in introducing the issue. For the most part, the reason I did not pursue this line of research further in this text is because I have not settled on "best practice" grammar to *push* variable/value bindings to the scope signature memory structure.[15]

Secondly, the other line of research worth bringing up is the **ambiguity** issue raised in the previous section. Grammatical ambiguity is already a known issue and is explored within other theories such as pushdown automata and context free grammars, so it's not an unexpected issue. In our case we avoided the problem by assuming our grammatical construct *ficopost* knew how many variables *match*? took and so could deduce when it became closed, or fully applied (instead of just partially applied). This is one possible way to solve this problem, but there are others, and as of yet I don't know which design is a best fit keeping in line with the larger narrative I'm trying to build here.

Finally, it's also possible—and not unreasonable—that we might generally wish to construct other functions in more expressive ways that allow us to compose functions while not being restricted to this left-to-right compositional ordering constraint. In fact, there's no reason not to expand our function induction grammar to include such expressive possibilities, only that this too is an open problem. Again, I do not *yet* know the best way forward.

Beyond this, and out of respect to you the reader, I would like to say that all the grammar presented in this essay is new, even to me, and in terms of open research it's the sort of thing where I would like to construct many and various functions first, to become fluent in this paradigm, before I commit to further grammatical designs.

Thank you.

References

[1] D. Nikpayuk. Toward the Semantic Reconstruction of Mathematical Functions (2020). https://github.com/Daniel-Nikpayuk/Mathematics/blob/master/Essays/Function%20Semantics/Version-Two/semantics.pdf

[2] H. Abelson, G.J. Sussman. Structure and Interpretation of Computer Programs (second edition). The Massachusetts Institute of Technology Press (1996).

[15] Currently the line of research which seems most promising is the realization that a memory system is just a function implemented in memoization style. In which case, grammar that lets us build functions that build such memoized functions already falls in line with the larger designs presented here.