Integer Sequence Continuation and Arbitrary Curve Fitting via Synthesis of Lambda Expressions

Eli Maynard

September 18, 2019

1 Goal

We aim to be able to predict the next item in a sequence of integers. For instance, the following sequences have obvious, natural extensions:

$$1, 1, 1, 1 \to ?$$

$$1, 2, 3, 4 \to ?$$

$$-1, -2, -3, -4 \to ?$$

Namely, 1, 5, and -5, respectively. We'd like to find an algorithm that will work, not only for obvious examples, and not just for classes of sequences—i.e., constant, linear, oscillating, exponential, geometric, etc.—but for all sequences.

1.1 Formalization

We first need to formalize what "predict the next item" precisely means. There are many possible interpretations, but we will use one that is both reasonable and mathematically convenient.

A "sequence" can be seen as a function—from an index to a value. Now, 'function' is a very broad term:

$$f : \mathbb{Z} \to \mathbb{Z} = x \mapsto \text{ the number of people who are } x \text{ years old}$$

is a function, but it is far out of scope. Since our goal is to find an *algorithm* for continuing sequences, we'll limit ourselves to computable functions.

So we need a model of computation. Ideally, it should be expressive, since what we care about is finding the 'natural' and 'simple' relation that describes the sequence. On that note, we choose to represent functions with the λ calculus.

Now in order to "find the next number" of a sequence we say that we want to find the minimal, or simplest, sufficient relation, or λ expression, that describes the sequence. Consider trying to extend the sequence

$$1, 5, -2, 4$$

the correct relation should *agree* with all the values so far, which is to say that it maps:

$$0 \to 1$$

$$1 \to 5$$

$$2 \to -2$$

$$3 \to 4$$

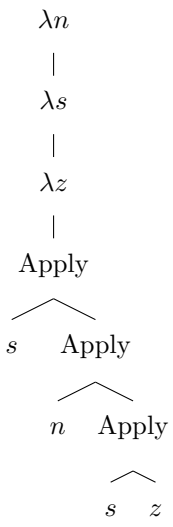
—this is sufficiency.

Defining 'minimal' is slightly tougher. We will create a metric of complexity $\mathcal{C}$ of a λ expression $\mathcal{E}$, and consider the minimal sufficient λ expression to be the sufficient expression $\mathcal{E}$ that has the minimal $\mathcal{C}(\mathcal{E})$.

We define the complexity $\mathcal{C}(\mathcal{E})$ to be the number of nodes in the tree representation of the λ-expression. For instance, consider the successor function S:

$$\begin{aligned} S &= \lambda nsz.s(nsz) \\ &= \lambda n.\lambda s.\lambda z.s(n(sz)) \end{aligned}$$

this would be represented in a tree as so:



which has 10 nodes and therefore $\mathcal{C} = 10$. Note that we don't count the parameter names to be a node unto themselves. $\mathcal{C}(\lambda n) = 1$, not 2.

Rigorizing complexity this way is one way that using λ calculus over turing machines becomes useful. Because λ calculus is expressive, λ expression node count should indeed correlate to relation complexity.

1.2 Applying λ Calculus to Integer Sequences

Our domain of discourse is the integers. We could emulate the integers in λ calculus with e.g. an extension to the Church Numerals. However, we care about expressiveness, so we instead want the fundamental structure of integers to be baked-in to the language.

Luckily for us, most of the work of figuring out the natural structure of integers is has been done for us. We start with the naturals:

$$0 \to 1 \to 2 \to 3 \to 4 \to ...$$

The naturals start with 0, and after 0 there's another number, called $S(0) = 1$, and then $S(S(0)) = 2$, and on and on... So, we include 0 and S as baked-in value to our (now 'impure') λ calculus.

In order to extend to the integers, and also because preceding doesn't seem fundamentally more complicated than succeeding, we also include $S^{-1} = P$ as a primitive value.

If we'd like to deal with any interesting functions, we'll want to allow them to be recursive. Recursive algorithms require conditional computation, so we also include a primitive equals function $E : \mathbb{Z} \to \mathbb{Z} \to a \to a \to a$ which returns a boolean in typical λ formulation.[1]

For reasons soon to be discussed, we'll be restricting our domain even further such that the algorithm will not be able to produce the Y combinator. Thus, we must also include it as a primitive value to allow for recursion.

E, S, and P are all assigned complexity $\mathcal{C} = 1$. Y has the complexity of its canonical form where $\mathcal{C} = 14$.

2 Reducing the Search Space

It is now possible to write an algorithm to continue integer sequences! It would iterate over all λ expressions in our domain in order of complexity and stop when it found a sufficient one. This would be the minimally sufficient expression[2].

[1] For those unfamiliar: booleans take the form of functions $b : a \to a \to a$. A boolean function takes two values and gives you one back—it chooses between two values. True and false are defined as $true = \lambda ab.a$ and $false = \lambda ab.b$.

[2] It is possible that there are two different expressions which are both minimally sufficient; we essentially ignore this issue and allow the algorithm to pick whichever expression it pleases.

Figure 1: Rough pseudocode for predicting the next item of a sequence.

```
1: procedure PREDICT-NEXT-ITEM(sequence)
2:     program ← FIND-MINIMALLY-SUFFICIENT-PROGRAM(sequence)
3:     f ← EVALUATE(program)
4:     i ← LENGTH(sequence)            ▷ The predicted item's index is the length of the sequence
5:     return f(i)
6: procedure FIND-MINIMALLY-SUFFICIENT-PROGRAM(sequence)
7:     for complexity in [1, ∞) do
8:         for program in PROGRAMS-OF-COMPLEXITY(complexity) do
9:             if SUFFICIENT(program, sequence) then
10:                return program
11: procedure SUFFICIENT(program, sequence)
12:    f ← EVALUATE(program)
13:    for i in [0, LENGTH(sequence)) do ▷ Check all indicies for which the sequence has an item
14:        if f(i) ≠ sequence[i] then          ▷ If the program disagrees with the sequence
15:            return false
16:    return true
```

However, this algorithm would be unbearably slow. In order to combat this, we will reduce the search space of λ expressions that `programs-of-complexity` has to sift through.

2.1 Using De Brujin Indices

If two λ expressions differ only by parameter names[3], i.e $\lambda x.x$ vs $\lambda y.y$, they are said to be α-equivalent and are functionally equivalent. We could reduce the search space by keeping track of variable names and making sure we don't generate any two α-equivalent expressions, but there is a more elegant way to avoid α-equivalency.

A reference is usually denoted with a symbol, like x or y, but in reality it's a reference to a particular abstraction. In $\lambda x.x$, the inner x is really referring to the single abstraction it's contained in. In $\lambda x.\lambda y.yx$, the inner y is referring to the closest abstraction and the inner x is referring to the

[3]Variable shadowing is relevant here, but not worth discussing explicitly.

second-closest abstraction.

We can replace references with indices representing the 'distance' to the abstraction it's referencing, starting at 1. $\lambda x.x$ thus becomes $\lambda 1$, and $\lambda x.\lambda y.yx$ becomes $\lambda\lambda 12$. Using these indices, called De Brujin indices, α-equivalence is not something that needs to be checked for.

In the code, we start De Brujin indices at 0 rather than 1. Because it's code. Not math.

2.2 Avoiding Invalid Expressions

The most glaring issue is that we will be generating just-plain-invalid expressions, like $S00$—$S0$ is not a function, so this will fail at runtime. We can avoid this by only generating expressions that span among the following types[4]:

$$\mathbb{Z}$$

$$\mathbb{Z} \to \mathbb{Z}$$

$$(\mathbb{Z} \to \mathbb{Z}) \to \mathbb{Z}$$

$$\mathbb{Z} \to (\mathbb{Z} \to \mathbb{Z})$$

$$(\mathbb{Z} \to \mathbb{Z}) \to (\mathbb{Z} \to \mathbb{Z})$$

and so forth

Further, we will require that all *sub*expressions be typeable this way[5].

Notably, this disallows us from generating the Y combinator $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ which contains untypeable subexpression xx. This is why we explicitly include the Y combinator as a primitive value. This does not cause any issues because the Y combinator itself *is* typeable this way; it has type $(a \to a) \to a$.

Since we're iterating over programs in order of complexity, we may generate them with a goal complexity in mind. Thus we want an algorithm that given a type τ and complexity $\mathcal{C}$, will return all valid programs of type τ and complexity $\mathcal{C}$. We will express this algorithm as a series of replacement rules, to be followed nondeterministically, where $[\tau]^{\mathcal{C}}B$ represents a request to generate a program of type τ and complexity $\mathcal{C}$ with bound variables B:

[4] i.e., polymorphism is okay

[5] This is subtly different than only requiring the *top-level* generated function to have one of these types. Consider a naughty invalid type T—then we may have $(T \to \mathbb{Z})\mathbb{Z}$ which reduces to $\mathbb{Z}$, so it seems that we would be able to generate it, when in fact we cannot.

Figure 2: Replacement rules to generate a program of some type and complexity. Variables a, b, $\mathcal{C}$, n, k, ϕ, and ψ are free.

$$[\mathbb{Z}]^1 \to 0 \tag{1}$$

$$[\mathbb{Z} \to \mathbb{Z}]^1 \to S \tag{2}$$

$$[\mathbb{Z} \to \mathbb{Z}]^1 \to P \tag{3}$$

$$[\mathbb{Z} \to \mathbb{Z} \to (a \to a \to a)]^1 \to E \tag{4}$$

$$[(a \to a) \to a]^{14} \to Y \tag{5}$$

$$[a \to b]^{\mathcal{C}} \to \lambda\phi.[b]^{\mathcal{C}-1} \qquad \text{where } \phi : a \text{ is hereafter bound} \tag{6}$$

$$[\tau]^{\mathcal{C}} \to [a \to \tau]^n [a]^k \qquad \text{where } n + k + 1 = \mathcal{C} \tag{7}$$

$$[\tau]^1 \to \psi \qquad \text{where } \psi : \tau \text{ was previously bound} \tag{8}$$

For (4), remember that booleans are of type $a \to a \to a$ so E is of type $\mathbb{Z} \to \mathbb{Z} \to$ Boolean. Also, in (7), the restriction is $n + k + 1 = \mathcal{C}$ and not $n + k = \mathcal{C}$ because it's an application and so the application adds 1 to the complexity.

Now all programs generated in this manner will run successfully. The only issue is that they may not halt, due to the inclusion of the Y combinator.

2.2.1 So Many Types

(Section under construction)

In this formulation, we have hidden something slightly subtle. Take another look at rule (8):

$$\forall \tau, \mathcal{C}, a, n, k \quad \tau^{\mathcal{C}} \to [a \to \tau]^n a^k \text{ s.t } n + k + 1 = \mathcal{C}$$

—a can be anything! a can be literally any (valid) type and the type of the generated expression will be t. This is a huge deal, because there are *infinite* valid types—so we can't just iterate over all of them in the code.

There is a saving grace. We may not be bounded by types, but we are bounded by complexity—we know that the $a \to t$ term should be of complexity n, and the a term should be of complexity

k. We need to figure out how to iterate over all values of a in such a way that we know when we can stop without missing out on any generated expressions.

We iterate over types by depth:

$\mathbb{Z}$	Depth 1
$\mathbb{Z} \to \mathbb{Z}$	Depth 2
$(\mathbb{Z} \to \mathbb{Z}) \to \mathbb{Z}$	Depth 3
$\mathbb{Z} \to (\mathbb{Z} \to \mathbb{Z})$	Depth 3
$(\mathbb{Z} \to \mathbb{Z}) \to (\mathbb{Z} \to \mathbb{Z})$	Depth 3
And so on	

And assert, in essence, that if a type T is not producing any novel λ-expressions (i.e., that haven't been produced by a type of a lower depth) of complexity $< \mathcal{C}$, then no type of depth *higher* than T will, either, and we can stop iterating over types after that depth.

I should prove that this is true, but I haven't yet.

2.3 Contrivance Checks

It is possible that, as we're iterating over λ expressions, we will reach one which is functionally equivalent to an expression we have already tested. We may, for instance, generate and test $\lambda x.S(S(x))$ for sufficiency and then later generate $(\lambda yx.S(S(x)))0$. Since these are functionally equivalent[6] and we have already tested the former, it would be redundant to also test the latter. And since we care about finding the minimal expression, and we iterate in order of increasing complexity, it would be best to just skip generating the altter expression at all!

On this note, we call an expression 'contrived' if and only if it is functionally equivalent to another expression of lesser complexity. We strive to avoid generating contrived expressions. Note that it is not possible to determine if an expression is contrived *in general*, since equivalence of λ expressions is undecidable. We may, however, deduce contrivance or non-contrivance in particular situations.

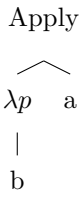
[6] and, in fact, β-equivalent

2.3.1 Single-Reduction

If an expression is β-equivalent to another expression, then they are also functionally equivalent. Thus, if a *single* β-reduction decreases an expression's complexity, then the original expressions was contrived.

Without loss of generality[7], we'll only consider expressions of the form $(\lambda p.b)a$—the expressions upon which β-reduction works. p for parameter, b for body, and a for argument.

Figure 3: Expression of $(\lambda p.b)a$ as a tree



What we care about is is the complexity of $(\lambda p.b)a$ *after* β-reduction. β-reduction of $(\lambda p.b)a$ will result in $b[p/a]$, i.e., the body b in which every instance of p is replaced with a. So we have that:

$$\mathcal{C}((\lambda p.b)a) > \mathcal{C}(b[p/a]) \implies (\lambda p.b)a \text{ is contrived} \tag{1}$$

And note that the complexity of $b[p/a]$ is the complexity of b, minus the number of references to p in b—since each of these references will be removed—, plus [the number of references to p in b] times [the complexity of a]—since each reference will then be replaced by a.

If we call the number of references to p in b $\Sigma_p(b)$, then we have:

$$\mathcal{C}(b[p/a]) = \mathcal{C}(b) - \Sigma_p(b) + \Sigma_p(b) \cdot \mathcal{C}(a) \tag{2}$$

Also note that, by Figure 3,

$$\mathcal{C}((\lambda b.p)a) = 2 + \mathcal{C}(b) + \mathcal{C}(a) \tag{3}$$

[7]Note that if $\mathcal{E}$ contains a contrived subexpression then $\mathcal{E}$ is contrived

Plugging (3) and (2) into (1), we get that

$$
\begin{aligned}
& 2 + \mathcal{C}(b) + \mathcal{C}(a) > \mathcal{C}(b) - \Sigma_p(b) + \Sigma_p(b) \cdot \mathcal{C}(a) \implies (\lambda p.b)a \text{ is contrived} \\
\Leftarrow \quad & 2 + \mathcal{C}(a) > -\Sigma_p(b) + \Sigma_p(b) \cdot \mathcal{C}(a) \\
\Leftarrow \quad & 2 + \mathcal{C}(a) > (\mathcal{C}(a) - 1) \cdot \Sigma_p(b) \\
\Leftarrow \quad & 3 + (\mathcal{C}(a) - 1) > (\mathcal{C}(a) - 1) \cdot \Sigma_p(b) \\
\Leftarrow \quad & 3 > (\mathcal{C}(a) - 1) \cdot (\Sigma_p(b) - 1)
\end{aligned}
$$

thus, we arrive at the following theorem:

Theorem 2.1 (Single-Reduction) *If a λ expression $\mathcal{E}$ is of the form $(\lambda p.b)a$ and $(\mathcal{C}(a) - 1) \cdot (\Sigma_p(b) - 1) < 3$ then $\mathcal{E}$ is contrived.*

The form of this theorem is very pretty, but in the code we actually use the simple corollary that contrivance is implied by:

$$\Sigma_p(b) > \frac{3}{\mathcal{C}(a) - 1} + 1$$

which is equivalent to

$$\Sigma_p(b) \geq lgi\left(\frac{3}{\mathcal{C}(a) - 1} + 1\right)$$

where lgi calculates the least greater integer:

$$lgi(x) = \begin{cases} x + 1 & x \in \mathbb{Z} \\ \lceil x \rceil & x \in \mathbb{R} \setminus \mathbb{Z} \end{cases}$$

2.3.2 Multiple-Reduction

(have yet to implement)

The power of Single-Reduction is that its condition is very, very quick to compute. It allows us to reduce the search space by an appreciable amount with a trivial amount of runtime dedicated to the computation of the condition. The more generalized form of Single-Reduction would do a full β-reduction and, if at any point, the complexity decreased beyond the complexity of the original form, then we would know the original expression was contrived. The issue here is that we'd have to *do* a full β-reduction, which is expensive. Hold that thought.

Now refer back to figure 1 (the pseudocode). During sufficiency checking, we evaluate the program. And during evaluation, we do a full β-reduction *anyway*, so we may as well do this contrivance checking as we go. We can thus revise the `evaluate` function to throw an exception if contrivance detected. We will rely on a function `beta-reduce-once` which will β-reduce an expression once, unless the expression cannot be β-reduced, in which case it will return `nil`[8].

Figure 4: An evaluation algorithm that short-circuits if contrivance is detected

```
1: procedure EVALUATE(expression)
2:     loop
3:         reducted ← BETA-REDUCE-ONCE(expression)
4:         if reduced = nil then
5:             return expression
6:         if C(reduced) < C(expression) then
7:             throw contrived
8:         expression ← reduced
```

The calling function, `sufficient`, is expected to handle the exception should it be thrown. The code for that will not be discussed.

2.3.3 Primitive Redundancy

Any expression of the form $P(S(x))$ or $S(P(x))$ is functionally equivalent to just x and is therefore contrived. So, we want to avoid generating expressions of this form.

When implementing this, however, we found that, though it reduced search space, it consistently had a net increase on runtime. Thus, this 'optimization' has been forgone.

2.4 Structural Minimums

Take a λ expression $\mathcal{E}$. Call the total number of references to all parameters $\Sigma(\mathcal{E})$. We know that all references must be 'contained', so to speak, in $\mathcal{E}$'s structure. And the minimal structure to do this

[8]The author does not condone the use of `nil`; a `Maybe` or `Option` type would be more appropriate. However, using `nil` is a simpler way to explain the algorithm

is one that combines all the reference nodes with application nodes—the only binary node—which is a full binary tree.

Figure 5: A plausible form of the minimal-complexity structure to contain some references

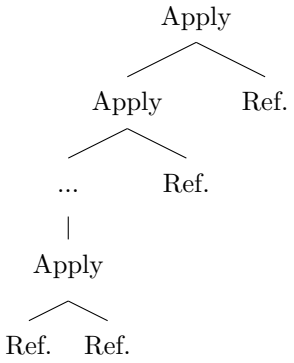


Figure 5 shows a plausible form of this minimal structure, but in general we care about the number of nodes in a full binary tree with $\Sigma(\mathcal{E})$ leaves, which is $2\Sigma(\mathcal{E}) - 1$ [1]; thus:

Theorem 2.2 (Structural Minimums) *A λ expression $\mathcal{E}$'s complexity is bounded by* $\mathcal{C}(\mathcal{E}) \geq 2\Sigma(\mathcal{E}) - 1$.

2.5 Better Structural Minimums

3 Extension to Rationals

- Generalize from functions $f : \mathbb{Z}^n \to \mathbb{Z}$ to $g : \mathbb{R} \to \mathbb{R}$; this would be incredibly useful in the world of curve-fitting

- Add addition, subtraction, multiplication and multiplicative inverse

- Add addition and subtraction atomically b/c multiplication of two inverses cannot be expressed otherwise

- Also these operations just kind of *are* the fundamental operations of the rationals. When you look at it like a closed group or whatever.

- Also add exponentiation. This would technically extend to the reals, I think, but floating-point numbers are effectively both the reals and the rationals.

- do not just extend to the rational b.c want to be able to construct e.g. π in the "correct" manner

- Constant parameters could be generated as well which would be optimized to fit the curve

4 Future Research

- Reduce search space

- Reduce search space by improving Structural Minimums. You have the types of all the references and in principle should be able to give a better minimum with that information.

References

[1] William McQuain. Binary tree theorems. http://courses.cs.vt.edu/ cs3114/Fall09/wmcquain/Notes/T03a.BinaryTree Accessed: 2019-02-10.