Inverting the Ackermann Hierarchy

Abstract

We build a hierarchy of functions that are an upper inverse of the usual Ackermann hierarchy, and then use this inverse hierarchy to compute the inverse of the diagonal Ackermann function $\mathrm{A}(n, n)$. We show that this computation is consistent with the usual definition of the inverse Ackermann function $\alpha(n)$. We implement this computation in Gallina, where we show that it runs in linear time.

Keywords: Inverse Ackermann, Automata, Union-Find, Division

1. Overview

The inverse to the explosively-growing Ackermann function features in several key algorithmic asymptotic bounds, such as the union-find data structure [?] and computing a minimum spanning tree of a graph [?]. Unfortunately, both the Ackermann function and its inverse can be hard to understand, and the inverse in particular can be hard to define in a computationally-efficient manner in a theorem prover. Let us consider why this is so.

Definition 1.1. The Ackermann-Péter function [?] (hereafter, just the Ackermann function) is a recursive two-variable function $\mathrm{A} : \mathbb{N}^2 \to \mathbb{N}$:

$$A(m, n) \triangleq \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m > 0, n = 0 \\ A(m - 1, A(m, n - 1)) & \text{if } m > 0, n > 0 \end{cases} \qquad (1)$$

The diagonal Ackermann function $\mathrm{A}(n)$ is then defined by $\mathrm{A}(n) \triangleq \mathrm{A}(n, n)$.

The diagonal Ackermann function grows explosively: starting from $\mathrm{A}(0)$, the first four terms are $1, 3, 7, 61$. The fifth term is $2^{2^{2^{65,536}}} - 3$, and the sixth dwarfs the fifth. This explosive behavior is problematical when we turn our attention to the canonical definition of the inverse Ackermann function[?].

Definition 1.2. The inverse Ackermann function $\alpha(n)$ is canonically defined as the minimum k for which $n \leq \mathrm{A}(k)$, *i.e.* $\alpha(n) \triangleq \min\{k \in \mathbb{N} : n \leq \mathrm{A}(k)\}$.

In a sense this definition is computational: starting with $k = 0$, calculate $\mathrm{A}(k)$, compare it to n, and then increment k until $n \leq \mathrm{A}(k)$. Unfortunately, the running time of this algorithm is $\Omega(\mathrm{A}(\alpha(n)))$, so to compute, for example, $\alpha(100) \mapsto^* 4$ in this way requires more than $\mathrm{A}(4) = 2^{2^{2^{65,536}}}$ steps!

The hyperoperation/Knuth/Ackermann hierarchy. The Ackermann function is relatively simple to define, but a little hard to understand. We think of it as a sequence of n-indexed functions $\mathrm{A}_n \triangleq \lambda b.\mathrm{A}(n, b)$, where for each $n > 0$, A_n iterates over the previous A_{n-1} b times, *with a kludge.*

The desire to clean up this kludge, as well as generalize the natural sequence of functions "addition," "multiplication," "exponentiation," ..., lead to the development of two related ideas: hyperoperations [**?**], written $a[n]b$, and Knuth arrows [**?**], written $a \uparrow^n b$. Since hyperoperations are a little more general (in particular, $a \uparrow^n b = a[n+2]b$), we will focus on them. Hereafter we will refer to this sequence of functions as "the hierarchy" when we mean the general pattern rather than *e.g.* specifically "the Ackermann hierarchy" or "the hyperoperation hierarchy". To illustrate the pattern, and demonstrate the Ackermann kludge, the following table contains the first 5 hyperations (indexed by n and named), along with how they relate to the A_n functions, and their inverses:

n	function	$a[n]b$	$\mathrm{A}_n(b)$	inverse	
0	successor	$1+b$	$1+b$	predecessor	$b-1$
1	addition	$a+b$	$2+b$	subtraction	$b-a$
2	multiplication	$a \cdot b$	$2b+3$	division	$\frac{b}{a}$
3	exponentiation	a^b	$2^{b+3}-3$	logarithm	$\mathsf{log}_a b$
4	tetration	$\underbrace{a^{\cdot^{\cdot^{a}}}}_{b}$	$\underbrace{2^{\cdot^{\cdot^{2}}}}_{b+3} - 3$	iterated logarithm	$\mathsf{log}^*_a b$

The kludge has three parts. First, the Ackermann hierarchy is related to the hyperoperation hierarchy when $a = 2$; second, for $n > 0$, A_n repeats the previous hyperoperation (**not** the previous A_{n-1}!) three extra times; lastly, A_n subtracts three[1]. Our initial goal of inverting the Ackermann function

[1]It might appear that A_1 and A_2 break this pattern, but they do not since $2+(b+3)-3 = 2+b$ and $2 \cdot (b+3) - 3 = 2b + 3$.

can thus be broken into three parts: first, inverting each individual member of the hyperoperation hierarchy; second, using these individual inverses to invert the diagonal hyperoperation $a[n]n$; and lastly adjusting for the kludge.

Increasing functions and their inverses. Defining increasing functions is often significantly simpler than defining their inverses. The Church numeral encodings of addition, multiplication, and even exponentiation are each simpler than their corresponding inverses of subtraction, division, and logarithm. We see the same pattern in mechanized contexts: defining multiplication in Gallina is child's play, but defining division is unexpectedly painful:

```
Fixpoint mult a b :=
 match a with
 | 0 => 0
 | S a' => b + mult a' b
 end.
```

```
Fixpoint div a b :=
 match a with
 | 0 => 0
 | _ => 1 + div (a - b) b
 end.
```

The definition of `mult` is of course accepted immediately by Coq; indeed it is the precise way multiplication is defined in the standard library. The function `div` should calculate multiplication's upper inverse, *i.e.* `div` $x\ y \mapsto^* \lceil \frac{x}{y} \rceil$, but the definition is rejected by the termination checker. Coq worries that `a - b` might not be structurally smaller than `a`, since subtraction is "just another function," and is thus treated opaquely. And indeed Coq is right to be nervous, since `div` will not in fact terminate when `a` > 0 and `b` $= 0$.

Of course, division *can* be defined, but an elegant definition is a little subtle—certainly, something beyond merely checking that `b` > 0 is needed. One method is to define a custom termination measure [**?**], but this is both vaguely unsatisfying and not always easy to generalize. The standard library employs a cleverer approach to define division, but we are not aware of any explanation of the technique used, nor is it trivial to extend to other members of the hierarchy. One indication that this is so is that the Coq standard library does not include a log_b function[2].

Contributions. We provide a complete solution to inverting each individual function in the Ackermann hierarchy, as well as the Ackermann function itself. All of our functions are structurally recursive, so Coq is immediately convinced of their termination. Moreover, all of our functions run in linear

[2]Coq's standard library **does** include a log_2 function, but change-of-base does not work on nat: $\left\lfloor \frac{\lfloor \mathsf{log}_2 100 \rfloor}{\lfloor \mathsf{log}_2 7 \rfloor} \right\rfloor = 3 \neq 2 = \lfloor \mathsf{log}_7 100 \rfloor$. efficiency of standard library version? SSReflect?

time (in the size of a number's representation in unary or binary), so they are as asymptotically efficient as they can be, and thus suitable for extraction[3]. Finally, our techniques are succinct: the code to invert the Ackermann function fits onto a single page of this paper, and our entire Coq development is only X lines. The rest of this paper is organized as follows.

- S?? We explain our core techniques of *repeaters* and *countdowns*. that allow us to define each level of the Ackermann hierarchy—and their upper inverses—in a straightforward and uniform manner. We show how countdowns, in particular, can be written structurally recursively.
- S?? We show how to use our techniques to define the Ackermann and upper inverse Ackermann functions themselves. Add inverse hyperop?
- S?? We detail a few optimizations that improve the running time of our individual hierarchy inverse functions from $O(n^2)$ to $O(n)$, and then further improve the running time of our inverse Ackermann function from $O\big(n \cdot \alpha(n)\big)$ to $O(n)$.
- S?? We extend our functions in several useful ways: to the two-argument inverse Ackermann, lower inverses, and binary representations.
- S?? We discuss related work and give some closing thoughts.

All of our techniques are mechanized in Coq [**?**], and for online readers, each theorem and definition is hyperlinked to our Coq development for browsing.

2. Repeater and Countdown

2.1. The repeater operation

First, we develop a simple way to move from one level of the hyperoperation hierarchy to the next. Addition is level 1, and b repetitions of addition give multiplication, which is level 2. Next, b repetitions of multiplication give exponentiation, which is level 3. However, there is a subtlety here: in the former case, we add a repeatedly to an *initial value*, which must clearly be 0. In the latter case, we multiply a repeatedly to an initial value, which must now be 1. In general, the formal definition for hyperoperation is

[3] A criterion more useful in practice for log_b and perhaps log^*_b than the other members of the hierarchy, to be sure.

1. *Initial level:* $a[0]b = b + 1\ \forall b$.
2. *Initial values:* $a[n+1]0 = a$ if $n = 0$, 0 if $n = 1$, and 1 otherwise.
3. *Recursive rule:* $a[n+1]b = a[n]\,(a[n+1](b-1))\ \ \forall n\ \forall b \geq 1$

The recursive formula may look complicated at first, but in fact is just *repeated application* in disguise. By fixing a and treating $a[n]b$ as a function of b, we can write

$$\begin{aligned} a[n+1]b &= a[n]\,(a[n+1](b-1)) = a[n]\,(a[n](a[n+1](b-2))) \\ &= \underbrace{(a[n] \circ a[n] \circ \cdots \circ a[n])}_{b \text{ times}}\,(a[n+1]0) = (a[n])^{(b)}\,(a[n+1]0) \end{aligned}$$

where $f^{(k)}(u) \triangleq (f \circ f \circ \cdots \circ f)(u)$ denotes applying a function f to an input u k times. In order to view the relationship between the (n+1)$^{\text{th}}$- and (n)$^{\text{th}}$-levels in a *functional* way, we need an operation that transforms the latter to the former, which leads us to the notion of *repeater*.

Definition 2.1. For all $a \in \mathbb{N}$ and $f : \mathbb{N} \to \mathbb{N}$, the *repeater from* a of f, denoted by $f{\uparrow_a}$, is a function $\mathbb{N} \to \mathbb{N}$ such that $f{\uparrow_a}(n) = f^{(n)}(a)\ \forall n$.

The Gallina definition is modified to structurally decrease on n:

```
Fixpoint repeater_from (a:nat) (f:nat->nat) (n:nat)
:= match n with
   | 0 => a
   | S n' => f (repeater_from a f n')
end.
```

The notation $f{\uparrow_a}(b)$ does much better at separating the function, i.e. the repeater of f, and the variable n than $f^{(n)}(a)$, while making clear that a is a parameter of *repeater* itself. It allows a simple and function-oriented definition of hyperoperations:

$$a[n]b = \begin{cases} b+1 & \text{if } n = 0 \\ a[n-1]{\uparrow_{a_n}}(b) & \text{if } n \geq 1 \end{cases} \quad \text{where} \quad a_n = \begin{cases} a & \text{if } n = 1 \\ 0 & \text{if } n = 2 \\ 1 & \text{if } n \geq 3 \end{cases} \tag{2}$$

2.2. Increasing expansions and their inverse

Our goal is to find an inverse hierarchy to the hyperoperations, which should include division, logarithm and $\log^*$ as inverses to multiplication, exponentiation and tetration. The advent of repeater motivates an "inverse repeater" operation that takes us to the next level in the supposed "inverse hyperoperation" hierarchy. Before going into details, let us clarify the notion of "inverse".

Definition 2.2. For all $F : \mathbb{N} \to \mathbb{N}$, $f : \mathbb{N} \to \mathbb{N}$ is the *upper inverse* of F if $f(n) = \min\{m : F(m) \geq n\}\ \forall n$, and is the *lower inverse* of F if $f(n) = \max\{m : F(m) \leq n\}\ \forall n$.

To construct a useful notion of inverse for our purpose, observe that for $a \geq 2$, each $a[n]$ is strictly increasing and grows to infinity with its input, while not always reaches 0, which ensures the existence of the upper but not the lower inverse, since $\{m : a[n]m \leq 0\} = \varnothing$ for $n \geq 3$. Thus we decide to go on with the upper inverse notion, and leave the other for discussion in section 4. Furthermore, we can restrict ourselves to only considering strictly increasing F, which in fact gives more sense to the above definition since inverse preserves increasing-ness, and allows us to describe upper inverse in a simple logical sentence:

Theorem 2.1. *If $F : \mathbb{N} \to \mathbb{N}$ is increasing, then f is the upper inverse of F if and only if $\forall n, m : f(n) \leq m \iff n \leq F(m)$.*

Proof. Fix n, the sentence $n \leq F(m) \iff f(n) \leq m\ \forall m$ implies: (1) $f(n)$ is lower bound to the set $\{m : F(m) \geq n\}$ and (2) $f(n)$ is in the set itself since plugging in $m := f(n)$ will yield $n \leq F(f(n))$, which makes f the upper inverse of F. Conversely, if f is the upper inverse of F, we immediately have $n \leq F(m) \implies f(n) \leq m\ \forall m$. Now for all $m \geq f(n)$, $F(m) \geq F(f(n)) \geq n$ by increasing-ness, thus complete the proof. □

The last property we would like F to have is one that ensures strict increasing-ness can be preserved through *repeater*. Suppose for some a and strictly increasing F, the function $F{\uparrow}_a$ is strictly increasing, then the chain a, $F(a)$, $F^{(2)}(a)$, … has to be strictly increasing, which leads to the notion of *expansions*.

Definition 2.3. Given $a \in \mathbb{N}$, a function $F : \mathbb{N} \to \mathbb{N}$ is an *expansion from a* if $F(n) \geq n + 1\ \forall n \geq a$.

Conveniently, if $a \geq 1$ and F is an expansion from a, $F{\uparrow_a}(n) = F^{(n)}(a) \geq a + n \geq 1 + n \; \forall n$, so $F{\uparrow_a}$ is itself an expansion from 0. Combined with the preservation of strict increasing-ness through repeater, we can collectively define strictly increasing expansions on $\mathbb{N}$ to be *repeatable* from any $a \geq 1$, with repeatability being preserved through *repeater* from a. In the next section, we show how repeatable functions' upper inverses can be constructed in a systematic way that allow us to build inverses for the hyperoperation hierarchy.

2.3. Contractions and the countdown operation

This is the final step leading to the inverse hypeoperations, where we develop a way to climb from the one inverse level to the next. Suppose f is the upper inverse of a repeatable (i.e. strictly increasing expansion) F, then for $a \geq 1$, $F{\uparrow_a}$ is repeatable and has an upper inverse f^*. Now $\forall n, m$,

$$\begin{aligned} f^*(n) \leq m \iff n \leq F{\uparrow_a}(m) = F^{(m)}(a) \iff f(n) \leq F^{(m-1)}(a) \\ \iff f^{(2)}(n) \leq F^{(m-2)}(a) \iff \ldots \iff f^{(m)}(n) \leq a \end{aligned}$$

This implies $f^*(n)$ is the minimum number of times f needs to be repeated over n to yield a result below a. Before we can define $f^*(n)$ to be the minimum of $\{m : f^{(m)} \leq a\}$ and claim success, we need to make sure: (1) that set is non-empty, so that it has a minimum, and (2) we can reliably find that minimum within finitely many steps. The most ideal situation would be that the chain n, $f(n)$, $f^{(2)}(n)$, $\ldots$ decreases strictly until reaching a, which leads to the notion of *contractions*.

Definition 2.4. A function $f : \mathbb{N} \to \mathbb{N}$ is a *contraction* if $f(n) \leq n \; \forall n \in \mathbb{N}$. Given an $a \geq 1$, a contraction f is *strict from a* if $n \geq f(n) + 1 \; \forall n \geq a$. We further denote by $\textsc{Contr}$ the set of contractions and $\textsc{Contr}_a$ the set of contractions strict from a.

That leaves the question: what type of functions F whose inverses are contractions? Wonderfully, we already have an answer:

Theorem 2.2. *If F is a strictly increasing expansion, the upper inverse f of F is a*

Definition 2.5. Given a contraction that is strict from some a, the *countdown_to* operation on f, written $f{\downarrow_a}(n)$, is the number of times that f can be compositionally applied to n without the result going below a.

$$f{\downarrow_a}(n) \triangleq (\textit{blah, but much simplified}) \tag{3}$$

Repeater can be written in Coq without much trouble, but Countdown proves a little more challenging. The definition above is a somewhat simplified version of the version in our development, which we shall discuss briefly here.

(listing of final code of countdown_to_worker)

Essentially, we introduce the notion of a *budget*, which is the maximum number of times we will try to compositionally apply f on the input. If a certain number of compositional applications of f makes the input go below a, *i.e.* $f^k(n) < a$, we return k, the number of applications it took. If we exhaust the budget and have still not gone below a, we fail by returning the budget itself.

This defnition is somewhat clunky, but it can clearly be written structurally recursively in Coq, with the budget as the decreasing argument. Further, it should be clear to see that if we provide the worker with a budget of n and also ensure that the function f is a contraction that is strict from a, we will never run out of budget, and the return value will truly be the number of times f was applied. In our proofs, we always use *countdown_to_worker* with these two conditions satisfied, *i.e.*

(code of countdown_to and countdownable_to)

3. The Inverse Ackermann Function

4. Further Discussion