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 $\mathrm{A} : \mathbb{N}^2 \to \mathbb{N}$:

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

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.

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. Increasing Functions and Their Inverses

3. The Inverse Ackermann Hierarchy

4. A linear time computation in Gallina

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