A Functional Proof Pearl: Inverting the Ackermann Hierarchy

Linh Tran
National University of Singapore

Anshuman Mohan
National University of Singapore

Aquinas Hobor
National University of Singapore

Abstract

We implement in Gallina a hierarchy of functions that calculate the upper inverses to the Ackermann/hyperoperation hierarchy, and then use our inverses to compute the inverse of the diagonal Ackermann function $\mathrm{A}(n)$. We show that our computation runs in linear time, and that it is consistent with the usual definition of the inverse Ackermann function $\alpha(n)$.

2012 ACM Subject Classification Theory of computation → Computational complexity and cryptography; Theory of computation → Logic and verification; Theory of computation → Constructive mathematics

Keywords and phrases Ackermann, hyperoperations, Coq

Digital Object Identifier 10.4230/LIPIcs...

1 Overview

The inverse to the explosively-growing Ackermann function features in several key algorithmic asymptotic bounds, such as the union-find data structure [18] and computing a minimum spanning tree of a graph [5]. 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 [15] (hereafter just "the" Ackermann function; see Section 6.2) is a recursive two-variable function $\mathrm{A} : \mathbb{N}^2 \to \mathbb{N}$ defined as follows:

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

The one-variable *diagonal* Ackermann function $\mathrm{A} : \mathbb{N} \to \mathbb{N}$ is defined as $\mathrm{A}(n) \triangleq \mathrm{A}(n,n)$. The diagonal Ackermann function grows explosively: starting from $\mathrm{A}(0)$, the first four terms are $1, 3, 7, 61$. The fifth term is $2^{2^{2^{65536}}} - 3$, and the sixth dwarfs the fifth. This explosive behavior becomes problematical when we consider the inverse Ackermann function[5, 18].

▶ **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 *e.g.* computing $\alpha(100) \mapsto^* 4$ in this way requires $\mathrm{A}(4) = 2^{2^{2^{65,536}}}$ steps!

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	$\mathsf{log}^*_a b$

Table 1 Hyperoperations, Ackermann functions and inverse.

1.1 The hyperoperation/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 is the result of applying the previous A_{n-1} b times, with a *kludge*.

The desire to clean up this kludge, and to generalize the natural sequence of functions "addition," "multiplication," "exponentiation," ..., led to the development of the related idea of hyperoperations [8], written $a[n]b$.[1] Table 1 illustrates this relation and the Ackermann kludge by showing the first 5 hyperoperations (indexed by n and named), along with their relation to the A_n functions, and their inverses.

The kludge has three parts. First, the Ackermann hierarchy is related to the hyperoperation hierarchy with $a = 2$, ie $2[n]b$; second, for $n > 0$, A_n repeats the previous hyperoperation $2[n-1]b$ three extra times; lastly, A_n subtracts three.[2] It is worth studying and inverting the hyperoperations before handling the Ackermann function.

1.2 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. Similarly, defining multiplication in Gallina [1] is trivial, 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. Indeed, Coq is right to be nervous: `div` will not terminate when `a` > 0 and `b` $= 0$.

[1] Knuth arrows [12], written $a \uparrow^n b$, are also in the same vein, but we will focus on hyperoperations since they are more general. In particular, $a \uparrow^n b = a[n+2]b$.

[2] A_1 and A_2 do not break this pattern: $2 + (b+3) - 3 = 2 + b$, and $2 \cdot (b+3) - 3 = 2b + 3$.

Of course, division *can* be defined, but an elegant definition is a little subtle—certainly, we need to do more than just check that `b` > 0. Two standard techniques are to define a custom termination measure [6]; and to define a straightforward function augmented with an extra `nat` parameter that denotes a "gas" value that decreases at each recursive call [16]. Both techniques are vaguely unsatisfying and neither is ideal for our purposes: the first can be hard to generalize and the second requires a method to calculate the appropriate amount of gas. Calculating the amount of gas to compute $\alpha(100)$ the "canonical" way, *e.g.* at least $2^{2^{2^{65,536}}}$, is problematic for many reasons, not least because we cannot use the inverse Ackerman function in its own termination argument. Realizing this, the standard library employs a cleverer approach to define division, but we are not aware of any explanation of the technique used, and we also find it hard to extend that technique to other members of the hierarchy. One indication of this difficulty is that the Coq standard library does not include a log_b function[3], to say nothing of a log_b^* or the inverse Ackermann.

1.3 Contributions

We provide a complete solution to inverting each individual function in the hyperoperation/Ackermann hierarchy, as well as the diagonal Ackermann function itself. All our functions are structurally recursive, so Coq is immediately convinced of their termination. Moreover, all our functions run in linear time. Finally, our techniques are extremely succinct: the code to invert the diagonal Ackermann function fits in a single page of this paper, and our entire Coq development is about 1,000 lines. The rest of this paper is organized as follows.

- S2 We show a formal definition for hyperoperations, explain how to encode hyperoperations and the Ackermann function in Coq, and discuss our concept of *repeater*.
- S3 We define a notion of upper inverses and show how to invert our notion of repeater with *countdowns*.
- S4 We show how to use our techniques to define the inverse hyperoperations, whose notable members include division, logarithm and iterated logarithm with arbitrary base; and sketch a method to find the Inverse Ackermann function.
- S5 We detail a specialized computation for the Inverse Ackermann function, improving over the sketch's running time of $\mathrm{O}(n^2)$ to $\mathrm{O}(n)$.
- S6 We discuss the two-argument inverse Ackermann function and survey related work.

All of our techniques are mechanized in Coq and is available online [11].

2 Hyperoperations, Ackermann, and Repeater

Let us now consider hyperoperations more carefully to clarify how they relate to the Ackermann function. The first hyperoperation (level 0) is simply successor, and after that, every hyperoperation is the repeated application of the previous. 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 should be* 0. In the latter case, we multiply a

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

repeatedly to an initial value, *which should be* 1. The formal definition for hyperoperation is:

$$\textit{1. Initial level: } a[0]b \triangleq b+1 \qquad \textit{2. Initial values: } a[n+1]0 \triangleq \begin{cases} a & \text{when } n=0 \\ 0 & \text{when } n=1 \\ 1 & \text{otherwise} \end{cases} \qquad (2)$$

$$\textit{3. Recursive rule: } a[n+1](b+1) \triangleq a[n]\big(a[n+1]b\big)$$

The seemingly complicated recursive rule is in fact just *repeated application* in disguise. By fixing a and treating $a[n]b$ as a function of b, we can write

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

where $f^{(k)}(u) \triangleq (f\circ f\circ\cdots\circ f)(u)$ denotes k successive applications of a function f to an input u, with $f^{(0)}(u)=u$ (applying 0 times).

This insight will help us encode both hyperoperations (2) and the Ackermann function (1) in Coq. Notice that the recursive case of hyperoperation—and indeed, the third case of the Ackermann function—is troublesome to encode in Coq due to the deep nested recursion. In the outer recursive call, the first argument (n) is shrinking but the second is expanding explosively; in the inner recursive call, the first argument is constant but the second is shrinking. The elegant solution uses double recursion [19] as follows:

```
Definition hyperop_init (a n : nat) : nat :=
  match n with 0 => a | 1 => 0 | _ => 1 end.

Fixpoint hyperop_original (a n b : nat) : nat :=
  match n with
  | 0    => 1 + b
  | S n' => let fix hyperop' (b : nat) :=
              match b with
              | 0    => hyperop_init a n'
              | S b' => hyperop_original a n' (hyperop' b')
              end
            in hyperop' b
  end.

Fixpoint ackermann_original (m n : nat) : nat :=
  match m with
  | 0    => 1 + n
  | S m' => let fix ackermann' (n : nat) : nat :=
              match n with
              | 0    => ackermann_original m' 1
              | S n' => ackermann_original m' (ackermann' n')
              end
            in ackermann' n
  end.
```

Coq is satisfied since both recursive calls are structurally smaller. Moreover, having encoded both notions in this style, the structural similarities are readily apparent. In fact, the only essential difference is the initial values (the second case of both definitions): the Ackermann function uses $\mathrm{A}(n-1,1)$, whereas hyperoperations use the initial values given in (2).

Having noticed that the deep recursion in both notions is expressing the same idea of repeated application, we arrive at another useful idea. We can express the relationship between the $(\mathrm{n}+1)^{\mathrm{th}}$- and n^{th}-levels in a *functional* way if we develop a higher-order function that transforms the latter level to the former. We call this idea a *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_a^R, is a function $\mathbb{N} \to \mathbb{N}$ such that $f_a^R(n) = f^{(n)}(a)$.

The Gallina definition is modified to structurally decrease on n:

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

The notation $f_a^R(b)$ 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:

$$a[n]b \triangleq \begin{cases} b+1 & \text{when } n = 0 \\ a[n-1]^R_{a_{n-1}}(b) & \text{otherwise} \end{cases} \qquad \text{where } a_n \triangleq \begin{cases} a & \text{when } n = 0 \\ 0 & \text{when } n = 1 \\ 1 & \text{otherwise} \end{cases}$$

Note we use $a[n-1]$ in a Curried style to denote the single-variable function $\lambda b.a[n-1]b$. The Ackermann function is likewise expressed elegantly using repeater:

$$\mathrm{A}(n,m) \triangleq \begin{cases} m+1 & \text{when } n = 0 \\ \mathrm{A}_{n-1}{}^R_{\mathrm{A}(n-1,1)}(m) & \text{otherwise} \end{cases}$$

In Coq these two definitions are written as follows:

```
Fixpoint hyperop (a n b : nat) : nat :=
  match n with
  | 0    => 1 + b
  | S n' => repeater_from (hyperop_init a n') (hyperop a n') b
  end.

Fixpoint ackermann (n m : nat) : nat :=
  match n with
  | 0    => S m
  | S n' => repeater_from (ackermann n' 1) (ackermann n') m
  end.
```

In the remainder of this paper we construct efficient inverses to these functions. Our key idea is an inverse to the higher-order repeater function; we call this inverse *countdown*.

3 Inverses and Countdown

On $\mathbb{R}$, many functions are bijections and thus have an inverse in a normal sense. Functions on $\mathbb{N}$ are often non-bijections and thus should be treated differently.

3.1 Upper Inverses and Expansions

▶ **Definition 3.1.** Define the *upper inverse* of F, written F_+^{-1} as $\min\{m : F(m) \geq n\}$. Notice that this is well-defined as long as F is unbounded, *i.e.* $\forall b.\ \exists a.\ b \leq F(a)$. However, as a notion of "inverse," it really only makes sense if F is strictly increasing, *i.e.* $\forall n, m.\ n < m \Rightarrow F(n) < F(m)$, which is in some sense the analogue of injectivity in the discrete domain.

We call this function the "upper inverse" because for strictly increasing functions like addition, multiplication, and exponentiation, the upper inverse is the ceiling of the corresponding inverse functions on $\mathbb{R}$. We can characterize inverses more meaningfully as follows:

▶ **Theorem 3.1.** *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 $\forall m.\ n \leq F(m) \iff f(n) \leq m$ implies: (1) $f(n)$ is a lower bound to $\{m : F(m) \geq n\}$ 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 $\forall m.\ n \leq F(m) \implies f(n) \leq m$. Now for all $m \geq f(n)$, $F(m) \geq F(f(n)) \geq n$ by increasing-ness, thus complete the proof. ◀

▶ **Corollary 3.2.** *If $F : \mathbb{N} \to \mathbb{N}$ is strictly increasing, then $F_+^{-1} \circ F$ is the identity function.*

Proof. By ($\Leftarrow$) of 3.1, $F(n) \leq F(n)$ implies $(F_+^{-1} \circ F)(n) \leq n$. By ($\Rightarrow$), $(F_+^{-1} \circ F)(n) \leq (F_+^{-1} \circ F)(n)$ implies $F(n) \leq F\big((F_+^{-1} \circ F)(n)\big)$; F is strictly increasing, so $n \leq (F_+^{-1} \circ F)(n)$. ◀

Our setup for inverse requires increasing functions, and our definitions for hyperoperations/Ackermann use repeater. Suppose F is a strictly increasing function. For a given a, is F_a^R strictly increasing? No! For example, the identity function id is strictly increasing, but $\mathsf{id}_a^R(n) = (\mathsf{id} \circ \ldots \circ \mathsf{id})(a) = a$ is a constant function. We need a little more.

▶ **Definition 3.2.** Given $a \in \mathbb{N}$, a function $F : \mathbb{N} \to \mathbb{N}$ is an *expansion* if $\forall n.\ F(n) \geq n$. An expansion F is *strict from a* if $\forall n \geq a.\ F(n) \geq n + 1$.

If $a \geq 1$ and F is an expansion strict from a, $\forall n.\ F_a^R(n) = F^{(n)}(a) \geq a + n \geq 1 + n$, so F_a^R is itself an expansion strict from 0. We refer to strictly increasing f as *repeatable* from $a \geq 1$ if they are also strict expansions from a, so that repeatability is preserved through F_a^R.

▶ **Definition 3.3.** We denote the set of functions repeatable from a as $\textsc{Rept}_a$.

▶ **Remark 3.3.** It is trivial to see that $\forall s, t.\ s \leq t \Rightarrow \textsc{Rept}_s \subseteq \textsc{Rept}_t$.

3.2 Contractions and the countdown operation

Suppose that $F \in \textsc{Rept}_a$ for any $a \geq 1$ and let f be F's inverse, *i.e.* F_+^{-1}. Our goal is to use f to compute an inverse to F's repeater F_a^R. Notice that this inverse must exist since $F \in \textsc{Rept}_a$ implies $F_a^R \in \textsc{Rept}_0$. For reasons that will be clear momentarily, we write this inverse as f_a^C. Now fix n and observe that for all m, $f^{(m)}(n) \leq a \iff m \geq f_a^C(n)$ since

$$\begin{aligned} f_a^C(n) \leq m &\iff n \leq F_a^R(m) = F^{(m)}(a) \iff f(n) \leq F^{(m-1)}(a) \\ &\iff f^{(2)}(n) \leq F^{(m-2)}(a) \iff \ldots \iff f^{(m)}(n) \leq a \end{aligned} \tag{3}$$

Moreover, setting $m = f_a^C(n)$, we realize that $f^{(f_a^C(n))} \leq a$. **Together these imply that $f_a^C(n)$ is the minimum number of times f needs to be compositionally applied to n before equalling or passing a.** In other words, count the length of the chain $\{n, f(n), f^{(2)}(n), \ldots\}$ that terminates as soon as we reach/pass a. For this process to work we need each chain link to be strictly less than the previous, *i.e.* f must be a *contraction*.

▶ **Definition 3.4.** A function $f : \mathbb{N} \to \mathbb{N}$ is a *contraction* if $\forall n.\ f(n) \leq n$. Given an $a \geq 1$, a contraction f is *strict above a* if $\forall n > a.\ n \geq f(n) + 1$. We denote the set of contractions by $\textsc{Contr}$ and the set of contractions strict above a by $\textsc{Contr}_a$.

▶ **Remark 3.4.** Similar to Remark 3.3, $\forall s \leq t.\ \textsc{Contr}_s \subseteq \textsc{Contr}_t$.

What kinds of functions have contractive inverses? Expansions, naturally:

▶ **Theorem 3.5.** *For all* $a \in \mathbb{N}$, $F \in \text{REPT}_a \implies F_+^{-1} \in \text{CONTR}_a$.

Proof. For all n, $F(n) \geq n \implies n \geq F_+^{-1}(n)$, so F_+^{-1} is a contraction. If $n \geq a+1$, $n-1 \geq a$, so $F(n-1) \geq n \implies n-1 \geq F_+^{-1}(n)$, so F_+^{-1} is strict above a. ◀

Theorem 3.5 shows a clear inverse relationship between expansions strict from some a and contractions strict above that same a. It ensures that the inverse of an expansion's repeater not only exists but can be built from its own inverse, in a method formalized as *countdown*.

▶ **Definition 3.5.** Let $f \in \text{CONTR}_a$, the *countdown to* a of f, denoted by $f_a^C(n)$, is the minimum number of times f needs to be applied to n to reach/pass a: $\min\{m : f^{(m)}(n) \leq a\}$.

Inspired by (3), we provide a neat, algebraically manipulable logical sentence equivalent to (3.5), which is more useful later in our paper.

▶ **Corollary 3.6.** *Let* $a \in \mathbb{N}$ *and* $f \in \text{CONTR}_a$. *Then* $\forall n, m.\ f_a^C(n) \leq m \iff f^{(m)}(n) \leq a$.

Proof. Fix a and n. The interesting direction is ($\implies$). Suppose $f_a^C(n) \leq m$, we get $f^{(m)}(n) \leq f^{(f_a^C(n))}(n)$ due to $f \in \text{CONTR}$, and $f^{(f_a^C(n))}(n) \leq a$ due to definition 3.5. ◀

Another useful result is the recursive formula for *countdown*.

▶ **Theorem 3.7.** *For all* $a \in \mathbb{N}$ *and* $f \in \text{CONTR}_a$, f_a^C *satisfies:*

$$f_a^C(n) = \begin{cases} 0 & \text{if } n \leq a \\ 1 + f_a^C(f(n)) & \text{if } n \geq a+1 \end{cases}$$

Proof. By corollary 3.6, $n \leq a \iff f^{(0)}(n) \leq a \iff f_a^C(n) \leq 0$, thus the case $n \leq a$ is resolved. Suppose $n \geq a+1$ and let $f_a^C(f(n)) = m$. We have $f_a^C(n) \leq 1+m \iff f^{(1+m)}(n) \leq a$, which is equivalent to $f^{(m)}(f(n)) \leq a$, which holds by m's definition.

Now since $n \geq a+1$, $f_a^C(n) \geq 1$ by the above. Let p be *s.t.* $f_a^C(n) = p+1$. It remains to prove $f_a^C(f(n)) \leq p$, or $f^{(p)}(f(n)) \leq a$, or $f^{(p+1)}(n) \leq a$, which holds by p's definition. ◀

3.3 A Structurally Recursive Computation for Countdown

The higher-order repeater function is well-defined for any input functions, even those not in REPT_a (although for such functions it may not be useful), and so is easy to define in Coq as shown in S2. In contrast, a *countdown* only exists for certain functions, most conveniently contractions, which proves a little more challenging to encode into Coq. Our strategy is to define a *countdown worker*, written with only structural recursion, and then prove that this worker computes the countdown when given a contraction.

▶ **Definition 3.6.** For any $a \in \mathbb{N}$ and $f : \mathbb{N} \to \mathbb{N}$, the *countdown worker* to a of f is a function $f_a^{CW} : \mathbb{N}^2 \to \mathbb{N}$ such that:

$$f_a^{CW}(n, b) = \begin{cases} 0 & \text{if } b = 0 \vee n \leq a \\ 1 + f_a^{CW}(f(n), b-1) & \text{if } b \geq 1 \wedge n > a \end{cases}$$

Essentially, *countdown worker* operates on two arguments, the *true argument* n, which we wish to count down to a, and the *budget* b, the maximum number of times we attempt to compositionally apply f on the input before giving up. If the input goes below or equal a after k applications, *i.e.* $f^{(k)}(n) \leq a$, we return the count k. If the budget is exhausted, i.e. $b = 0$, while the result is still above a, we fail by returning the original budget. This definition is workmanlike, but it can clearly be written as a Coq `Fixpoint`:

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

When f is a contraction strict from a, and given a sufficient budget, *countdown worker* will compute the correct *countdown* value. We will show that a budget of n is sufficient, using several lemmas about *countdown worker*, letting us define *countdown* in Coq:

▶ **Definition 3.7.** Redefine $f_a^C(n) \triangleq f_a^{CW}(n,n)$.

```
Definition countdown_to a f n := countdown_worker a f n n.
```

We start with a simple fact that *countdown worker* returns 0 when $n \le a$, which is intended for *countdown*. It follows trivially from definition 3.6.

▶ **Lemma 3.8.** *For all* $a, b \in \mathbb{N}$, $f : \mathbb{N} \to \mathbb{N}$ *we have* $\forall n \le a.\ f_a^{CW}(n,b) = 0$.

Next we show the internal working of *countdown worker* at the i$^{\text{th}}$ recursive step, including the accumulated result $1+i$, the current input $f^{(1+i)}(n)$, and the current budget $b-i-1$.

▶ **Lemma 3.9.** *For all* $a, n, b, i \in \mathbb{N}$ *and* $f \in$ Contr *such that* $i < b$ *and* $a < f^{(i)}(n)$*:*

$$f_a^{CW}(n,b) = 1 + i + f_a^{CW}\big(f^{1+i}(n), b-i-1\big) \tag{4}$$

Proof. Fix a. We proceed by induction on i. Define

$$P(i) \triangleq \forall b, n.\ b \ge i+1 \implies f^{(i)}(n) > a \implies f_a^{CW}(n,b) = 1+i+f_a^{CW}\big(f^{1+i}(n), b-i-1\big)$$

1. *Base case.* For $i = 0$, our goal $P(0)$ is: $f_a^{CW}(n,b) = 1 + f_a^{CW}(f(n), b-1)$ where $b \ge 1, f(n) \ge a+1$, which is trivial.
2. *Inductive step.* Suppose $P(i)$ has been proven. Then $P(i+1)$ is

$$f_a^{CW}(n,b) = 2 + i + f_a^{CW}\big(f^{2+i}(n), b-i-2\big)$$

for $b \ge i+2, f^{1+i}(n) \ge a+1$. This also implies $b \ge i+1$ and $f^{(i)}(n) \ge f^{1+i}(n) \ge a+1$ by $f \in$ Contr, thus $P(i)$ holds. It suffices to prove:

$$f_a^{CW}\big(f^{1+i}(n), b-i-1\big) = 1 + f_a^{CW}\big(f^{2+i}(n), b-i-2\big)$$

This is in fact $P(0)$ with (b,n) substituted for $\big(b-i-1, f^{(1+i)}(n)\big)$. Since $f^{(1+i)}(n) \ge a+1$ and $b-i-1 \ge 1$, the above holds and $P(i+1)$ follows. The proof is complete. ◀

Now it is time to prove the correctness of *countdown*, in other words that our computational definition 3.7 computes the value originally specified in definition 3.5.

▶ **Theorem 3.10.** *For all* $a \in \mathbb{N}$ *and* $f \in$ Contr$_a$*, we have* $\forall n.\ f_a^C(n) = \min\big\{i : f^{(i)}(n) \le a\big\}$.

Proof. Since $f \in$ Contr$_a$ and $\mathbb{N}$ is well-ordered, let $m = \min\big\{i : f^{(i)}(n) \le a\big\}$ (we prove its existence in Coq's baseline intuitionistic logic in our code base), then

$$\Big(f^{(m)}(n) \le a\Big) \wedge \Big(f^{(k)}(n) \le a \implies m \le k \ \ \forall k\Big) \tag{5}$$

It then suffices to prove $m = f_a^C(n)$. Suppose firstly that $m = 0$. Then $n = f^{(0)}(n) \leq a$, thus $f_a^C(n) = f_a^{CW}(n, n) = 0 = m$ by lemma 3.8.

Now consider $m > 0$. We would like to apply lemma 3.9 to get

$$f_a^C(n) = f_a^{CW}(n, n) = m + f_a^{CW}\left(f^{(m)}(n), n - m\right),$$

then use lemma 3.8 over (5)'s first conjunct to conclude that $f_a^C(n) = m$. It then suffices to prove the premises of lemma 3.9, namely $a < f^{(m-1)}(n)$ and $m - 1 < n$.

The former follows by contradiction: if $f^{(m-1)}(n) \leq a$, (5)'s second conjunct implies $m \leq m - 1$, which is wrong for $m > 0$. The latter then easily follows by $f \in \text{CONTR}_a$:

$$n \geq 1 + f(n) \geq 2 + f(f(n)) \geq \cdots \geq m + f^{(m)}(n)$$

Therefore, $f_a^C(n) = m$ in all cases, which completes the proof. ◀

Equation (3) and theorem 3.10 thus establish the correctness of the Coq definitions for *countdown worker* and *countdown*, justifying our unifying the equivalent definitions 3.5 and 3.7. We wrap everything together with the following theorem.

▶ **Theorem 3.11.** *For all a and $F \in \text{REPT}_a$, $f \triangleq F_+^{-1}$ satisfies $f \in \text{CONTR}_a$ and $f_a^C = \left(F_a^R\right)_+^{-1}$. Furthermore, if $a \geq 1$, $F_a^R \in \text{REPT}_0$ and $f_a^C \in \text{CONTR}_0$.*

Proof. By theorem 3.5, $f \triangleq F_+^{-1} \in \text{CONTR}_a$, and $f_a^C = \left(F_a^R\right)_+^{-1}$ follows from (3) and corollary 3.6. Now if $a \geq 1$, a simple induction shows that $F^{(n)}(a) \geq a + n \geq 1 + n\ \forall n$, so $F_a^R \in \text{REPT}_0$, hence $f_a^C = \left(F_a^R\right)_+^{-1} \in \text{CONTR}_0$ by theorem 3.5. ◀

4 Inverting the Hyperoperations and the Ackermann Function

In this section, we use *countdown* to define the inverse hyperoperation hierarchy, which features elegant new definitions of division, log, and log*. We then modify the inverse hyperoperation hierarchy to arrive at the inverse Ackermann hierarchy.

4.1 The Inverse Hyperoperations, including Division, Log, and Log*

▶ **Definition 4.1.** The inverse hyperoperations, written $a\langle n\rangle b$, are defined as follows:

$$a\langle n\rangle b \triangleq \begin{cases} b - 1 & \text{if } n = 0 \\ a\langle n-1\rangle_{a_n}^C(b) & \text{if } n \geq 1 \end{cases} \quad \text{where } a_n = \begin{cases} a \text{ if } n = 1 \\ 0 \text{ if } n = 2 \\ 1 \text{ if } n \geq 3 \end{cases} \tag{6}$$

Note that we use $a\langle n-1\rangle$ in a Curried style to denote the single-variable function $\lambda b.a\langle n-1\rangle b$.

```
Fixpoint inv_hyperop a n b :=
  match n with
  | 0 => b - 1
  | S n' =>
    countdown_to (hyperop_init a n') (inv_hyperop a n') b
  end.
```

We now prove that $a\langle n\rangle$ is the inverse to $a[n]$. First, note that $\forall a.\ a\langle 0\rangle \in \text{CONTR}_0$. Then:

▶ **Lemma 4.1.** $a\langle 1\rangle b = b - a$.

Proof. Since $a\langle 0\rangle \in \textsc{Contr}_0 \subseteq \textsc{Contr}_a$, theorem 3.7 applies.

$$a\langle 1\rangle b \quad = \quad (a\langle 0\rangle)_a^C\,(b) \quad = \quad \begin{cases} 0 & \text{if } b \le a \\ 1 + a\langle 1\rangle(b-1) & \text{if } b \ge a+1 \end{cases}$$

Then, $a\langle 1\rangle b = b - a$ by induction on b. ◀

▶ **Corollary 4.2.** $\forall a \ge 1, a\langle 1\rangle \in \textsc{Contr}_1$.

Note that $a\langle n\rangle b$ is a total function, but that its behavior is not used in practice for $a = 0$ when $n \ge 2$ or for $a = 1$ when $n \ge 3$. For the values we do care about, we have our inverse:

▶ **Theorem 4.3.** *When $n \le 1$, $n \le 2 \wedge a \ge 1$, or $a \ge 2$, then $a\langle n\rangle = (a[n])_+^{-1}$.*

Proof. $\forall n \ge 2$. let $a_0 = a, a_1 = 0, a_n = 1$. Define

$$P(n) \triangleq (a[n] \in \textsc{Rept}_{a_n}) \quad \text{and} \quad Q(n) \triangleq \left(a\langle n\rangle = (a[n])_+^{-1}\right)$$

We have three goals: $\forall a.\ Q(0) \wedge Q(1)$, $\forall a \ge 1.\ Q(2)$, and $\forall a \ge 2.\ Q(n)$. Note $\forall n.\ a\langle n+1\rangle = a\langle n\rangle_{a_n}^C$ and $a[n+1] = a[n]_{a_n}^R$. By theorem 3.11,

$$P(n) \Rightarrow Q(n) \Rightarrow Q(n+1) \qquad (7) \qquad a_n \ge 1 \Rightarrow P(n) \Rightarrow P(n+1) \qquad (8)$$

For the first goal, $P(0) \iff \lambda b.(b+1) \in \textsc{Rept}_a$ and $Q(0) \iff a\langle 0\rangle = (a[0])_+^{-1} \iff (b - 1 \le c \iff b \le c + 1)$. These are both straightforward. Then, $Q(1)$ holds by (7). In the second goal, we have $a \ge 1$. $P(1)$ holds by $P(0)$ and (8). Then, $Q(2)$ holds by $Q(1)$ and (7). In the third goal, we have $a \ge 2$, and, using (7) and $Q(0)$, the goal reduces to $P(n)$. Using (8) and the fact that $\forall n \ne 1.\ a_n \ge 1$, the goal reduces to $P(2)$. This is equivalent to:

$$a[2] \in \textsc{Rept}_0 \iff \forall b < c.\ ab < ac \quad \wedge \quad \forall b \ge 1.\ ab \ge b + 1,$$

which is trivial for $a \ge 2$. Induction on n gives us the third goal and thus the proof. ◀

▶ **Remark 4.4.** Three early hyperoperations are $a[2]b = ab$, $a[3]b = a^b$ and $a[4]b = {}^b a$, so by theorem 4.3, we can define their inverses $\lceil b/a \rceil$, $\lceil \log_a b \rceil$, and $\log_a^* b$ as $a\langle 2\rangle b$, $a\langle 3\rangle b$, and $a\langle 4\rangle b$. Note that the functions $\log_a b$, and $\log_a^* b$ are not in the Coq Standard Library.

4.2 The Inverse Ackermann hierarchy

We can use *countdown* to build the inverse Ackermann hierarchy, where each level α_i is the inverse to the level A_i. Since we know that $\mathrm{A}_{i+1} = \mathrm{A}_i{}_{\mathrm{A}_i(1)}^R$, the recursive rule $\alpha_{i+1} \triangleq \alpha_i{}_{\mathrm{A}_i(1)}^C$ is tempting. However, that approach is flawed because it still depends on A_i. Instead, we reexamine the inverse relationship: suppose $\alpha_i = \left(\mathrm{A}_i\right)_+^{-1}$ and $\alpha_{i+1} = \left(\mathrm{A}_{i+1}\right)_+^{-1}$. Then $\forall m.\ \mathrm{A}_{i+1}(m) = \left(\mathrm{A}_i\right)^{(m)}\left(\mathrm{A}_i(1)\right)$. We then have:

$$\alpha_{i+1}(n) \le m \iff n \le \left(A_i\right)^{(m+1)}(1) \iff \left(\alpha_i\right)^{(m+1)}(n) \le 1 \qquad (9)$$

This is equivalent to $\alpha_{i+1}(n) = \min\left\{m : \left(\alpha_i\right)^{(m+1)}(n) \le 1\right\}$, or $\alpha_{i+1}(n) = \alpha_i{}_1^C\left(\alpha_i(n)\right)$. From equation (9) we can thus define the following inverse Ackermann hierarchy:

▶ **Definition 4.2.** Define the inverse Ackermann hierarchy by $\alpha_i \triangleq \begin{cases} \lambda n.(n-1) & \text{if } i = 0 \\ \left(\alpha_{i-1}{}_1^C\right) \circ \alpha_{i-1} & \text{if } i \ge 1 \end{cases}$

We can redefine the inverse Ackermann function *without directly referring to* $A(i)$ using α_i:

▶ **Definition 4.3.** $\alpha(n) \triangleq \min\{k : \alpha_k(n) \leq k\}$.

All that remains is to provide a structurally-recursive function that can compute α effectively.

▶ **Definition 4.4.** The inverse Ackermann worker is a function $\mathrm{W}\alpha$:

$$\mathrm{W}\alpha(f, n, k, b) = \begin{cases} k & \text{if } b = 0 \vee n \leq k \\ \mathrm{W}\alpha(f_1^C \circ f, f_1^C(n), k+1, b-1) & \text{if } b \geq 1 \wedge n \geq k+1 \end{cases} \tag{10}$$

When given the arguments $(\alpha_i, \alpha_i(n), i, b)$ such that $\alpha_i(n) > i$ and $b > i$, $\mathrm{W}\alpha$ will take on arguments $(\alpha_{i+1}, \alpha_{i+1}(n), i+1, b-1)$ at the next recursive call. Thus if $\mathrm{W}\alpha$ is given a sufficient budget b, the initial arguments $(\alpha_0, \alpha_0(n), 0, b)$ will recursively move through the chain $\{\alpha_i(n)\}$ and terminate at a point k where $\alpha_k(n) \leq k$. We want to show that $\mathrm{W}\alpha$ can correctly compute $\alpha(n)$ given a budget of n. We start with the following lemma.

▶ **Lemma 4.5.** *For all n, b, k such that $k+1 \leq \min\{b, \alpha_k(n)\}$,*

$$W\alpha(\alpha_0, \alpha_0(n), 0, b) = W\alpha(\alpha_{k+1}, \alpha_{k+1}(n), k+1, b-k-1)$$

Proof. We prove $\forall k.\ P(k)$ by induction, where

$$P(k) \triangleq (\alpha_k(n) \geq k+1) \wedge (b \geq k+1) \implies \mathrm{W}\alpha(\alpha_0, \alpha_0(n), 0, b) = \mathrm{W}\alpha(\alpha_{k+1}, \alpha_{k+1}(n), k+1, b-k-1)$$

1. *Base case.* $\mathrm{W}\alpha(\alpha_0, \alpha_0(n), 0, b) = \mathrm{W}\alpha(({\alpha_0}_1^C) \circ \alpha_0, {\alpha_0}_1^C(\alpha_0(n)), 1, b-1)$ by (10) with $b \geq 1$ and $\alpha_0(n) \leq 1$. Since $\alpha_1 = (\alpha_0{}_1^C) \circ \alpha_0$ per definition 4.2, $P(0)$ holds.
2. *Inductive step.* Suppose $\alpha_{k+1}(n) \geq k+2$ and $b \geq k+2$. Then $\alpha_k(n) \geq k+1$ and $b \geq k+1$, so $P(k)$ applies. It suffices to show

$$\mathrm{W}\alpha(\alpha_{k+1}, \alpha_{k+1}(n), k+1, b-k-1) = \mathrm{W}\alpha(\alpha_{k+2}, \alpha_{k+2}(n), k+2, b-k-2)$$

Per definition 4.2, $\alpha_{k+2} = (\alpha_{k+1}{}_1^C) \circ \alpha_{k+1}$, so this is just (10) with $b-k-1 \geq 1$ and $\alpha_{k+1}(n) \geq k+2$, so $P(k+1)$ holds. The proof is complete by induction on k. ◀

We are at last ready for a strategy for computing the inverse Ackermann function.

▶ **Theorem 4.6.** $W\alpha(\alpha_0, \alpha_0(n), 0, n) = \alpha(n)$.

Proof. Since the sequence $\{\alpha_k(n)\}_{k=1}^{\infty}$ decreases while $\{k\}_{k=1}^{\infty}$ increases to infinity, there exists $m \triangleq \min\{k : \alpha_k(n) \leq k\} = \alpha(n)$. Note that $m \leq n$ since $\alpha_n(n) \leq n$. If $m = 0$, $\alpha_0(n) \leq 0 \implies \alpha_0(n) = 0$. Thus $\mathrm{W}\alpha(\alpha_0, \alpha_0(n), 0, n) = 0 = m$. If $m \geq 1$, lemma 4.5 implies

$$\mathrm{W}\alpha(\alpha_0, \alpha_0(n), 0, n) = \mathrm{W}\alpha(\alpha_m, \alpha_m(n), m, n-m) = m$$

where the last equality follows from the recursive rule. ◀

The code to compute the inverse Ackermann function in Gallina follows.

```
Fixpoint inv_ack_worker (f : nat -> nat) (n k bud : nat) : nat :=
  match bud with
  | 0 => 0
  | S bud' =>
    match (n - k) with
    | 0 => k
    | S _ =>
      let g := (countdown_to 1 f) in
      inv_ack_worker (compose g f) (g n) (S k) bud'
    end
  end.

Definition inv_ack n :=
  match (alpha 0) with f => inv_ack_worker f (f n) 0 n end.
```

5 Time Bound of Our Inverses

In this section, we provide a time analysis for the inverse Ackermann function. First, we show that the running time of the version defined in definition 4.3 is $O(n^2)$. We then provide a simple improvement that cuts the running time to $O(n)$.

5.1 Basic Time Analysis: $O(n^2)$

To formalize the notion of running time, let us identify each function on $\mathbb{N}$ with its *computation*, *i.e.* the program that computes it in Coq under a call-by-value strategy. Then briefly:

▶ **Definition 5.1.** Given a function $f : \mathbb{N} \to \mathbb{N}$ in Coq, the *running time* of f on input $n \in \mathbb{N}$, denoted by $\mathcal{T}(f, n)$ is the total number of computational steps it takes to compute $f(n)$.

▶ **Lemma 5.1.** $\forall a, n.\ \mathcal{T}(\lambda n.(n-a), n) = \Theta(\min\{a, n\})$ *per the Coq definition of subtraction.*

▶ **Lemma 5.2.** $\forall f, g : \mathbb{N} \to \mathbb{N}.\ \mathcal{T}(f \circ g, n) = \mathcal{T}(f, g(n)) + \mathcal{T}(g, n)$ *per the Coq definition of functional composition.*

▶ **Lemma 5.3.** *Per definition 3.7,* $\forall a \geq 1, \forall n \leq 1, \forall f \in \textsc{Contr}_a.\ \mathcal{T}\big(f_a^C, n\big) = 1$, *and*

$$\forall n \geq 2.\ \mathcal{T}\big(f_a^C, n\big) = \sum_{i=0}^{f_a^C(n)-1} \mathcal{T}\left(f, f^{(i)}(n)\right) + \Theta\big(a \cdot f_a^C(n)\big)$$

Proof. Per definition 3.6, the computation makes $f\,{}_a^C\,(n)$ recursive calls to $\mathrm{W} f\,{}_a^C$ before terminating. At the $(i+1)^{\text{th}}$ call, two computations must take place: $n_i - a$, which takes $\Theta(a)$ time, and $f(n_i) = n_{i+1}$, where $n_i \triangleq f^{(i)}(n)$ has been already computed by the ith call, and is greater than a. The total time is then

$$\mathcal{T}\big(f_a^C, n\big) = \sum_{i=0}^{f_a^C(n)-1} \left[\mathcal{T}\left(f, f^{(i)}(n)\right) + \Theta(a)\right] = \sum_{i=0}^{f_a^C(n)-1} \mathcal{T}\left(f, f^{(i)}(n)\right) + \Theta\big(a \cdot f_a^C(n)\big)$$

◀

From lemma 5.2 and lemma 5.3, the following lemma follows easily.

▶ **Lemma 5.4.** *Per definition 4.2 and definition 3.7,*

$$\forall i.\ \mathcal{T}\big(\alpha_{i+1}, n\big) = \sum_{k=0}^{\alpha_{i+1}(n)} \mathcal{T}\left(\alpha_i, \alpha_i^{(k)}(n)\right) + \Theta\big(\alpha_{i+1}(n)\big)$$

This lemma implies $\mathcal{T}\big(\alpha_{i+1}, n\big) \geq \mathcal{T}\big(\alpha_i, n\big)$. If there is some i such that $\mathcal{T}\big(\alpha_i, n\big) = \Theta\big(n^2\big)$, each function in the hierarchy will take at least $\Omega\big(n^2\big)$ to compute from α_i, thus making $\mathcal{T}(\alpha, n) = \Omega\big(n^2\big)$ per definition 4.4. The next lemma shows that $i = 2$ suffices.

▶ **Lemma 5.5.** *Per definition 4.2,* $\mathcal{T}\big(\alpha_2, n\big) = \Theta\big(n^2\big)$

Proof. $\alpha_1 = \big(\lambda m.(m-1)_1^C\big) \circ \big(\lambda m.(m-1)\big) \equiv \lambda m.(m-2)$. By lemma 5.4,

$$\mathcal{T}\big(\alpha_1, n\big) = \sum_{i=0}^{n-1} \mathcal{T}\big(\lambda m.(m-1), n-i\big) + \Theta\big((n-2)\big) = \Theta(n),$$

since $\forall k.\ \mathcal{T}\big(\lambda m.(m-1), k\big) = 1$. Because $\alpha_2 = \big(\alpha_1{}_0^C\big) \circ \alpha_1$, again by lemma 5.4,

$$\mathcal{T}\big(\alpha_2, n\big) = \sum_{i=0}^{\lceil (n-3)/2 \rceil} \mathcal{T}\big(\alpha_1, n-2i\big) + \Theta\left(\frac{n}{2}\right) = \Theta\left(\sum_{i=0}^{\lceil (n-3)/2 \rceil} (n-2i)\right) = \Theta\big(n^2\big)$$

◀

▶ **Corollary 5.6.** $\mathcal{T}(\alpha, n) = \Omega\big(n^2\big)$ *per definition 4.4.*

5.2 Hard-coding the second level: $O(n)$

Intuitively, it is clear that the function $\alpha_2 \equiv \lambda n.(n-2)$ slows down the entire hierarchy's performance due to one silly weakness: it does not know that it will always output $n-2$ before beginning its computation, and so it tediously subtracts 1 until it goes below 2.

We can hard-code α_1 as $\lambda n.(n-2)$ to reduce its running time from $\Theta(n)$ to $\Theta(1)$.

```
Definition sub_2 (n : nat) : nat :=
  match n with | 0 => 0 | 1 => 1 | S (S n') => n' end.
```

Without loss of generality, let us assume from now on that the constant factors in $\mathcal{T}(\alpha_1, n)$ and lemma 5.4 are both 1. The running time for α_2 is then

$$\mathcal{T}\big(\alpha_2, n\big) = \sum_{i=0}^{\lceil (n-3)/2 \rceil} \mathcal{T}\big(\alpha_1, n-2i\big) + \left\lceil \frac{n-3}{2} \right\rceil = 2\left\lceil \frac{n-3}{2} \right\rceil < n$$

In fact, this fix allows every function in the hierarchy to be computed in linear time:

▶ **Theorem 5.7.** $\forall i, n.\ \mathcal{T}\big(\alpha_i, n\big) \leq 2n + (4^{i+1} - 4)\lceil \log_2 n \rceil + 5$.

We need two crucial technical lemmas to prove this theorem.

▶ **Lemma 5.8.** $\forall n.\ \mathcal{T}\big(\alpha_3, n\big) \leq 2n + 4$.

Proof. It is easy to show that for all k, $\alpha_2^{(k)}(n) = \lceil \frac{n+3}{2^k} \rceil - 3$. Thus $\mathcal{T}\big(\alpha_3, n\big) =$

$$\begin{aligned} &\textstyle\sum_{k=0}^{\alpha_3(n)} \mathcal{T}\left(\alpha_2, \lceil \frac{n+3}{2^k} \rceil - 3\right) + \alpha_3(n) \quad \leq \quad \sum_{k=0}^{\alpha_3(n)} \left(\frac{n+3}{2^k} + 1\right) - 3\big(\alpha_3(n) + 1\big) + \alpha_3(n) \\ &\leq \textstyle\sum_{k=0}^{\alpha_3(n)} \frac{n+3}{2^k} - \alpha_3(n) - 2 \quad \leq \quad 2(n+3) - 2 \quad = \quad 2n + 4 \end{aligned}$$

◀

▶ **Lemma 5.9.** $\forall i \geq 3.\ \displaystyle\sum_{k=1}^{\alpha_{i+1}(n)} \alpha_i^{(k)}(n) \leq 3\lceil \log_2 n \rceil$.

Proof. Let the LHS be $S_i(n)$. Firstly, consider $i = 3$. Note that for $n \leq 13$, $S_3(n) = 0$ and for $n \geq 14$, i.e. $\alpha_3(n) \geq 2$, $S_3(n) = \alpha_3(n) + S_3\big(\alpha_3(n)\big)$. The result thus holds for $n \leq 13$. Suppose it holds for all $m < n$, where $n \geq 14$. Then

$$S_3(n) \quad \leq \quad \alpha_3(n) + 3\lceil \log_2(\alpha_3(n)) \rceil \quad \leq \quad \lceil \log_2 n \rceil + 3\lceil \log_2 \log_2 n \rceil$$

It is easy to prove $2\lceil \log_2 \log_2 n \rceil \leq \lceil \log_2 n \rceil$ by induction on $\lceil \log_2 n \rceil$. Thus $S_3(n) \leq 3\lceil \log_2 n \rceil$, as desired. Now for $i \geq 4$,

$$S_i(n) \quad = \quad \sum_{k=1}^{\alpha_{i+1}(n)} \alpha_i^{(k)}(n) \quad \leq \quad \sum_{k=1}^{\alpha_{i+1}(n)} \alpha_3^{(k)}(n) \quad \leq \quad \sum_{k=1}^{\alpha_4(n)} \alpha_3^{(k)}(n) \quad \leq \quad 3\lceil \log_2 n \rceil$$

◀

Proof of Theorem 5.7. We have proved the result for $i = 0, 1, 2$. Let us proceed with $i \geq 3$ by induction. The base case $i = 3$ has been covered by lemma 5.8. Now suppose the result holds for $i \geq 3$, let $M_i \triangleq 4^{i+1} - 4$ for each i, we have $\mathcal{T}\big(\alpha_{i+1}, n\big) =$

$$\begin{aligned}
&\textstyle\sum_{k=0}^{\alpha_{i+1}(n)} \mathcal{T}\left(\alpha_i, \alpha_i^{(k)}(n)\right) + \alpha_{i+1}(n) \\
&\leq \textstyle\sum_{k=0}^{\alpha_{i+1}(n)} \left(2\alpha_i^{(k)}(n) + M_i \left\lceil \log_2 \left(\alpha_i^{(k)}(n)\right)\right\rceil + 5\right) + \alpha_{i+1}(n) \\
&\leq 2n + M_i \lceil \log_2 n \rceil + 5 + (M_i + 2) \textstyle\sum_{k=1}^{\alpha_{i+1}(n)} \alpha_i^{(k)}(n) + 6\alpha_{i+1}(n) \\
&\leq 2n + M_i \lceil \log_2 n \rceil + 5 + 3(M_i + 2) \lceil \log_2 n \rceil + 6 \lceil \log_2 n \rceil \quad = \quad 2n + (4M_i + 12) \lceil \log_2 n \rceil + 5 \\
&= 2n + M_{i+1} \lceil \log_2 n \rceil + 5, \text{ since } 4M_i + 12 = 4^{i+2} - 16 + 12 = M_{i+1}.
\end{aligned}$$

◀

By hard-coding α_1 as $\lambda m.(m-2)$, the ith level of the inverse Ackermann hierarchy can be computed in $\Theta\big(n + 4^i \log n\big)$ time, *i.e.* linear time $\Theta(n)$ for fixed i.

This also allows us to improve running time of $\alpha(n)$ per Definition 4.2 by hard-coding the output when $n \leq 1 = \mathrm{A}(0, 0)$, and starting $\mathrm{W}\alpha$ with $f = \alpha_1$ and $n = \alpha_1(n)$.

$$\tilde{\alpha}(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ \mathrm{W}\alpha\big(\alpha_1, \alpha_1(n), 1, n-1\big) & \text{if } n \geq 2 \end{cases}$$

For $n > 1$, we have $1 \leq \min\big\{n-1, \alpha_1(n)\big\}$, so $\mathrm{W}\alpha\big(\alpha_0, \alpha_0(n), 0, n\big) = \mathrm{W}\alpha\big(\alpha_1, \alpha_1(n), 1, n-1\big)$. Thus $\tilde{\alpha}(n) = \alpha(n)$. Now at each recursive step, the transition from $\mathrm{W}\alpha\big(\alpha_k, \alpha_k(n), k, n-k\big)$ to $\mathrm{W}\alpha\big(\alpha_{k+1}, \alpha_{k+1}(n), k+1, n-k-1\big)$ consists of the following computations:

(1) $\alpha_k^C(x)$ given $x \triangleq \alpha_k(n)$, which takes time $\mathcal{T}\big(\alpha_{k+1}, n\big) - \mathcal{T}\big(\alpha_k, n\big)$ by lemma 5.2.
(2) $\alpha_k(n) - k$, which takes time $\Theta(k)$.

The computation will terminate at $k = \alpha(n)$. Thus $\forall n \geq 1$,

$$\begin{aligned}
\mathcal{T}\big(\tilde{\alpha}, n\big) &= \mathcal{T}\big(\alpha_1, n\big) + \sum_{k=1}^{\alpha(n)-1} \big[\mathcal{T}\big(\alpha_{k+1}, n\big) - \mathcal{T}\big(\alpha_k, n\big)\big] + \sum_{k=1}^{\alpha(n)} \Theta(k) \\
&= \mathcal{T}\left(\alpha_{\alpha(n)}, n\right) + \Theta\left(\alpha(n)^2\right) = \Theta\left(2n + 4^{\alpha(n)} \log_2 n + \alpha(n)^2\right) = \Theta(n)
\end{aligned} \tag{11}$$

Therefore, $\tilde{\alpha}$ is able to compute α in linear time.

6 Further Discussion

6.1 Two-parameter inverse Ackermann function

Some authors [18, 5] prefer a two-parameter inverse Ackermann function.

▶ **Definition 6.1.** The *two-parameter inverse Ackermann function* of m and n is

$$\alpha(m, n) \triangleq \min\left\{i \geq 1 : \mathrm{A}\left(i, \left\lfloor \frac{m}{n} \right\rfloor\right) \geq \log_2 n\right\} \tag{12}$$

Note that $\alpha(m, n)$ and $\alpha(n)$ are neither equal nor directly related, but it is straightforward to modify our techniques to compute this function.

▶ **Definition 6.2.** The *two-parameter inverse Ackermann worker* is a function $\mathrm{W}\alpha_2 : (\mathbb{N} \to \mathbb{N}) \times \mathbb{N}^3 \to \mathbb{N}$ such that for all $n, k, b \in \mathbb{N}$ and $f : \mathbb{N} \to \mathbb{N}$:

$$\mathrm{W}\alpha_2(f, n, k, b) = \begin{cases} 0 & \text{if } b = 0 \vee n \leq k \\ 1 + \mathrm{W}\alpha_2\big(f_1^C \circ f, f_1^C(n), k, b-1\big) & \text{if } b \geq 1 \wedge n \geq k+1 \end{cases} \tag{13}$$

▶ **Theorem 6.1.** *For all* m, n, $\alpha(m, n) = 1 + W\alpha_2\left(\alpha_1, \alpha_1\big(\lceil \log_2 n \rceil\big), \left\lfloor \frac{m}{n} \right\rfloor, \lceil \log_2 n \rceil\right)$.

6.2 Related Work and Conclusion

Historical notes

The operations successor, predecessor, addition, and subtraction have been integral to counting forever. The ancient Egyptian number system used glyphs denoting 1, 10, 100, *etc*, and then expressed numbers using additive combinations of these. The Roman system is similar, but it combines glyphs using both addition and subtraction. This buys brevity, since *e.g.* 9_{roman} is two characters, "one less than ten", whereas 9_{egyptian} is nine characters, a series of nine 1s. The ancient Babylonian system was similar to the modern Hindu-Arabic decimal system, an *algorism*: the place value of a glyph determined how many times it would be counted towards the number being represented. The Babylonians operated in base 60, and so *e.g.* a three-gylph number $abc_{\text{babylonian}}$ could be parsed as $a \times 3600 + b \times 60 + c$. Sadly they lacked a radix point, and so $a \times 216000 + b \times 3600 + c \times 60$ and $a \times 60 + b + c \div 60$ were also reasonable interpretations. In return for incorporating the operations multiplication and division, they enjoyed great brevity: a number n could be represented in $\lfloor \log_{60} n \rfloor + 1$ glyphs.

The Ackermann function and its inverse

The original three-variable Ackermann function was discovered by Wilhelm Ackermann as an example of a total computable function that is not primitive recursive [2]. It grows tremendously fast, but does not have the higher-order relation to repeated application and hyperoperation that we have been studying in this paper. Those properties emerged thanks to refinements by Rózsa Péter [15], and it is her variant, usually called the Ackermann-Péter function, that computer scientists commonly care about.

The inverse Ackermann features in the time bound analyses of several algorithms. Tarjan [18] showed that the union-find data structure takes time $O(m \cdot \alpha(m, n))$ for a sequence of m operations involving no more than n elements. Chazelle [5] showed that the minimum spanning tree of a connected graph with n vertices and m edges can be found in time $O(m \cdot \alpha(m, n))$.

Moving on to mechanized verifications, Charguéraud and Pottier, [4] later joined by Guéneau [9], extended Separation Logic with the notion of "time credits", formalized the O notation in Coq, and gave a proof in Coq that simultaneously verifies the correctness and the time complexity of the union-find data structure. Several others [3, 7, 10, 13] have explored the idea of checking bounds on the resources used by programs formally in proof assistants such as Coq, Isabelle/HOL, and Why3.

Every pearl starts with a grain of sand. We had the benefit of two: a website and slide deck discussing the inverse Ackermann function by Nivasch [14] and Seidel [17], respectively. They proposed a definition of the inverse Ackermann essentially in terms of the inverse hyperoperations. Unfortunately, their technique is unsound, since it diverges from the true Ackermann inverse when the inputs grow sufficiently large. Our technique is verified in Coq.

Conclusion

We have implemented a hierarchy of functions that calculate the upper inverses to the Ackermann/hyperoperation hierarchy and used these inverses to compute the inverse of the diagonal Ackermann function $\mathrm{A}(n)$. Our functions are structurally recursive, and thus immediately accepted by Coq, and we have shown that they run in linear time.

References

1. The coq proof assistant. URL: https://coq.inria.fr/.
2. Wilhelm Ackermann. *Zum Hilbertschen Aufbau der reellen Zahlen*. 1928.
3. Ales Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. Iron: managing obligations in higher-order concurrent separation logic. *PACMPL*, 3(POPL):65:1–65:30, 2019. URL: https://dl.acm.org/citation.cfm?id=3290378.
4. Arthur Charguéraud and François Pottier. Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. *J. Autom. Reasoning*, 62(3):331–365, 2019. URL: https://doi.org/10.1007/s10817-017-9431-7, doi:10.1007/s10817-017-9431-7.
5. Bernard Chazelle. A minimum spanning tree algorithm with Inverse-Ackermann type complexity. *J. ACM*, 47(6):1028–1047, 2000. URL: https://doi.org/10.1145/355541.355562, doi:10.1145/355541.355562.
6. Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013. URL: http://mitpress.mit.edu/books/certified-programming-dependent-types.
7. Jean-Christophe Filliâtre, Léon Gondelman, Andrei Paskevich, Mário Pereira, and Simão Melo De Sousa. A Toolchain to Produce Correct-by-Construction OCaml Programs. working paper or preprint, May 2018. URL: https://hal.inria.fr/hal-01783851.
8. R. L. Goodstein. Transfinite ordinals in recursive number theory. *J. Symb. Log.*, 12(4):123–129, 1947. URL: https://doi.org/10.2307/2266486, doi:10.2307/2266486.
9. Armaël Guéneau, Arthur Charguéraud, and François Pottier. A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, pages 533–560, 2018. URL: https://doi.org/10.1007/978-3-319-89884-1_19, doi:10.1007/978-3-319-89884-1_19.
10. Maximilian P. L. Haslbeck and Tobias Nipkow. Hoare logics for time bounds - A study in meta theory. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*, pages 155–171, 2018. URL: https://doi.org/10.1007/978-3-319-89960-2_9, doi:10.1007/978-3-319-89960-2_9.
11. Inv-Ack. inv-ack/inv-ack. URL: https://github.com/inv-ack/inv-ack.
12. D. E. Knuth. Mathematics and computer science: Coping with finiteness. *Science*, 194(4271):1235–1242, 1976. doi:10.1126/science.194.4271.1235.
13. Tobias Nipkow and Hauke Brinkop. Amortized complexity verified. *J. Autom. Reasoning*, 62(3):367–391, 2019. URL: https://doi.org/10.1007/s10817-018-9459-3, doi:10.1007/s10817-018-9459-3.
14. Gabriel Nivasch. Inverse ackermann without pain. URL: http://www.gabrielnivasch.org/fun/inverse-ackermann.
15. Rózsa Péter. Konstruktion nichtrekursiver funktionen. *Mathematische Annalen*, 111(1):42–60, Dec 1935. URL: https://doi.org/10.1007/BF01472200, doi:10.1007/BF01472200.
16. Benjamin C Pierce. Software foundations. URL: https://softwarefoundations.cis.upenn.edu/lf-current/ImpCEvalFun.html.
17. Raimund Seidel. Understanding the inverse ackermann function. URL: http://cgi.di.uoa.gr/~ewcg06/invited/Seidel.pdf.
18. Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975. URL: https://doi.org/10.1145/321879.321884, doi:10.1145/321879.321884.
19. wires. Error in defining Ackermann in Coq, Apr 2012. URL: https://stackoverflow.com/a/10303475.