A Functional Proof Pearl: Inverting the Ackermann Hierarchy

Linh Tran
Department of Mathematics
Yale University
USA
School of Computing
National University of Singapore
Singapore

Anshuman Mohan
Yale-NUS College and
School of Computing
National University of Singapore
Singapore

Aquinas Hobor
Yale-NUS College and
School of Computing
National University of Singapore
Singapore

Abstract

We implement in Gallina a hierarchy of functions that calculate the upper inverses to the hyperoperation/Ackermann hierarchy. Our functions run in $\Theta(b)$ for inputs expressed in unary, and in $O(b^2)$ for inputs expressed in binary (where b = bitlength). We use our inverses to define linear-time functions—$\Theta(b)$ for both unary-represented and binary-represented inputs—that compute the upper inverse of the diagonal Ackermann function $\mathcal{A}(n)$. We show that these functions are consistent with the usual definition of the inverse Ackermann function $\alpha(n)$.

CCS Concepts • **Theory of computation → Constructive mathematics**; *Design and analysis of algorithms*; • **Mathematics of computing** → *Discrete mathematics.*

Keywords Ackermann, hyperoperations, inverses, Coq

ACM Reference Format:
Linh Tran, Anshuman Mohan, and Aquinas Hobor. 2020. A Functional Proof Pearl: Inverting the Ackermann Hierarchy. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '20), January 20–21, 2020, New Orleans, LA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3372885.3373837

1 Overview

The inverse to the explosively-growing Ackermann function features in several algorithmic asymptotic bounds, such as the union-find data structure [19] and the computation of a graph's minimum spanning tree [4]. Unfortunately, Ackermann and its inverse are both hard to understand, and the inverse is hard to define in a computationally-efficient manner in a theorem prover. Let us consider why this is so.

Definition 1. *The Ackermann-Péter function [16] (hereafter just "the" Ackermann function; see §8) is written* $A : \mathbb{N}^2 \to \mathbb{N}$ *and 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 single-variable *diagonal* Ackermann function written $\mathcal{A} : \mathbb{N} \to \mathbb{N}$ is defined as $\mathcal{A}(n) \triangleq A(n,n)$. $\mathcal{A}$ grows explosively: starting from $\mathcal{A}(0)$, 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 [4, 19].

Definition 2. *The inverse Ackermann function, denoted by* $\alpha : \mathbb{N} \to \mathbb{N}$*, is defined such that* $\alpha(n)$ *is the smallest* k *for which* $n \le \mathcal{A}(k)$, i.e. $\alpha(n) \triangleq \min\{k \in \mathbb{N} : n \le \mathcal{A}(k)\}$.

This definition is in some sense computational: start at $k = 0$, calculate $\mathcal{A}(k)$, compare to n, and increment k until $n \le \mathcal{A}(k)$. Unfortunately, its runtime is $\Omega(\mathcal{A}(\alpha(n)))$, so *e.g.* computing $\alpha(100) \mapsto^* 4$ takes $\mathcal{A}(4) = 2^{2^{2^{65536}}} - 3$ steps!

1.1 The Hyperoperation/Ackermann Hierarchy

The Ackermann function is relatively easy to define, but hard to understand. It can be seen as a sequence of n-indexed functions $\mathcal{A}_n \triangleq \lambda b.A(n,b)$, where for each $n > 0$, $\mathcal{A}_n$ is the result of applying the previous $\mathcal{A}_{n-1}$ b times, with a *kludge*.

The desire to clean up this kludge, and to generalize the natural sequence of functions (addition, multiplication, exponentiation, *etc.*), led to the development of hyperoperations [8], written $a[n]b$[1]. In Table 1 we show the first five hyperoperations (indexed by n) and the related kludgy $\mathcal{A}_n(b)$

[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, $a \uparrow^n b = a[n+2]b$.

CPP '20, January 20–21, 2020, New Orleans, LA, USA

ACM ISBN 978-1-4503-7097-4/20/01…$15.00
https://doi.org/10.1145/3372885.3373837

Table 1. Hyperoperations, the Ackermann hierarchy, and inverses thereof.

n	$a[n]b$	$\mathcal{A}_n(b)$	$a\langle n\rangle b$	$\alpha_n(b)$
0	$1+b$	$1+b$	$b-1$	$b-1$
1	$a+b$	$2+b$	$b-a$	$b-2$
2	$a\cdot b$	$2b+3$	$\lceil \frac{b}{a} \rceil$	$\lceil \frac{b-3}{2} \rceil$
3	a^b	$2^{b+3}-3$	$\lceil \log_a\ b \rceil$	$\lceil \log_2 (b+3) \rceil - 3$
4	$\underbrace{a^{\cdot^{\cdot^{a}}}}_{b}$	$\underbrace{2^{\cdot^{\cdot^{2}}}}_{b+3} - 3$	$\log_a^*\ b$	$\log_2^* (b+3) - 3$

Ackermann functions. We also show their inverses, which we write as $a\langle n\rangle b$ and $\alpha_n(b)$ respectively. The kludge has three parts. First, $\mathcal{A}_n$ sets a to 2, *i.e.* $2[n]b$; second, for $n > 0$, $\mathcal{A}_n$ repeats the previous hyperoperation $2[n-1]b$ three extra times; lastly, $\mathcal{A}_n$ subtracts three.[2] We show here in our codebase that the kludge is stated correctly.

It is worth studying and inverting the hyperoperations before handling the Ackermann inverses. With computationally efficient definitions of the individual Ackermann inverses α_n in hand, we shall redefine α—the inverse of the diagonal Ackermann $\mathcal{A}$—in a computationally effficient way. The key is our Theorem 26 (§4), which characterizes α without referring to $\mathcal{A}$ directly: $\forall n.\ \alpha(n) = \min\{k : \alpha_k(n) \le k\}$.

1.2 Increasing Functions and Their Inverses

Defining increasing functions is often easier than defining their inverses. For instance, addition, multiplication, and exponentiation on Church numerals are simpler than subtraction, division, and logarithm. Similarly, defining multiplication in Gallina [6] is easy, but defining division along the same lines leads to some suffering:

```
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.
```

Coq accepts the definition of `mult`; indeed this is how multiplication is defined in the standard library. The function `div` is meant to calculate multiplication's upper inverse, *i.e.* `div` $x\ y \mapsto^* \lceil \frac{x}{y} \rceil$, but the definition is rejected by Coq's 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` $= 0$.

[2] $\mathcal{A}_1$ and $\mathcal{A}_2$ do not break this pattern: $2 + (b+3) - 3 = 2 + b$, and $2\cdot(b+3) - 3 = 2b + 3$.

```
Require Import Omega Program.Basics.

Fixpoint cdn_wkr a f n b :=
  match b with 0 => 0 | S b' =>
    if (n <=? a) then 0
      else S (cdn_wkr f a (f n) k')
  end.

Definition countdown_to a f n := cdn_wkr a f n n.

Fixpoint inv_ack_wkr f n k b :=
  match b with 0 => 0 | S b' =>
    if (n <=? k) then k
      else let g := (countdown_to f 1) in
        inv_ack_wkr (compose g f) (g n) (S k) b
  end.

Definition inv_ack_linear n :=
  match n with 0 | 1 => 0 | _ =>
    let f := (fun x => x - 2) in
      inv_ack_wkr f (f n) 1 (n - 1)
  end.
```

Figure 1. A linear-time Coq computation of the inverse Ackermann function for inputs in unary, *i.e.* `nat`.

Of course, division *can* be defined, but an elegant definition is a little subtle. We certainly need to do more than just check that `b` > 0. Two standard techniques are to define a custom termination measure [5]; and to augment a straightforward function with an extra "`gas:nat`" parameter whose value decreases at each recursive call [17]. 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 $2^{2^{2^{65536}}}$, 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 find it difficult to extend that technique—essentially, defining an automaton—to other functions in the hierarchy in a computationally efficient way. One indication of this difficulty is that the Coq standard library does not include a $\log_b$ function[3], to say nothing of a $\log_b^*$ function or the inverse Ackermann function.

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, our functions run

[3] Coq's standard library includes a $\log_2$ function. However, just as the halving function div_2 is easier to define than general division, $\log_2$ is simpler to define than logarithm with arbitrary base. Note that change-of-base does not work on discrete logarithms: $\left\lfloor \frac{\lfloor \log_2 100 \rfloor}{\lfloor \log_2 7 \rfloor} \right\rfloor = 3 \neq 2 = \lfloor \log_7 100 \rfloor$.

in linear time: practical utility follows from theoretical grace. Finally, our techniques are compact. The key inverse functions themselves are remarkably succinct: in Figure 1 we show 14 lines of code that calculates the inverse diagonal Ackermann function α in linear time for unary-represented inputs. The entire Coq development, which proves the correctness of our definitions, is about 2,500 lines, split between parallel efforts in unary-encoded and binary-encoded inputs.

§2 We show a formal definition for hyperoperations, present Coq encodings for hyperoperations and the Ackermann function, and discuss *repeater*.
§3 We discuss upper inverses and show how to invert repeater with *countdown*.
§4 We use our techniques to define the inverse hyperoperations, whose notable members include division, logarithm and iterated logarithm with arbitrary base. We sketch a route to the inverse Ackermann function.
§5 We give a Coq computation for the inverse Ackermann function in $O(n)$ time for a unary-encoded input n.
§6 We extend our work to calculate inverse Ackermann for binary inputs. We show that this takes time $O(b)$, where b is the bitlength of the input n.
§7 We discuss the value of linear time inverses for explosively growing functions, benchmark our results experimentally, extend our work to calculate the two-argument inverse Ackermann function, and implement our techniques in Isabelle/HOL.
§8 We present historical notes and survey related work.

All our techniques are mechanized in Coq and our code is available online at github.com/inv-ack/inv-ack [11]. Further, definitions and theorems presented in this paper are linked directly to appropriate points in our codebase, represented using hyperlinks and the symbol 🖺 where appropriate.

2 Hyperoperations and Ackermann via Repeater

Let us formally define hyperoperations and clarify the intuition given by Table 1 by relating hyperoperations to the Ackermann function. The first hyperoperation (level 0) is simply successor, and every hyperoperation that follows is the repeated application of the previous. Level 1 is thus addition, and b repetitions of addition give level 2, multiplication. Next, b repetitions of multiplication give level 3, exponentiation. There is a subtlety here: in the former case, we add a repeatedly to the additive identity 0, but in the latter case, we multiply a repeatedly to the multiplicative identity 1. The formal definition of hyperoperation is:

$$\begin{array}{lrcl} \textit{1. } 0^{th} \textit{ level:} & a[0]b & \triangleq & b+1 \\ \textit{2. Initial values:} & a[n+1]0 & \triangleq & \begin{cases} a & \text{when } n=0 \\ 0 & \text{when } n=1 \\ 1 & \text{otherwise} \end{cases} \\ \textit{3. Recursive rule:} & a[n+1](b+1) & \triangleq & a[n]\big(a[n+1]b\big) \end{array} \tag{2}$$

The recursive rule looks complicated, but is actually 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 compositional applications of a function f to an input u. Here $f^{(0)}(u) = u$, *i.e.* applying f zero times yields the identity.

This insight helps us encode hyperoperations (2) and Ackermann (1) in Coq. Notice that the recursive case of hyperoperations and the third case of Ackermann both feature deep nested recursion, which makes our task tricky. In the outer recursive call, the first argument 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 [2] 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 on structurally smaller arguments. Moreover, our encoding makes the structural similarities readily apparent. In fact, the only essential difference is the initial values (*i.e.* the second case of both definitions): the Ackermann function uses $\mathcal{A}(n-1,1)$, whereas hyperoperations use the initial values given in (2).

We notice that the deep recursion in both cases is expressing the same notion of repeated application, and this leads us to another useful idea. We can elegantly express the relationship between the $(n+1)^{\text{th}}$ and n^{th} levels via a higher-order function that transforms the latter level to the former using a version of the well-known function iterator `iter` [2]:

Definition 3. *$\forall a \in \mathbb{N}, f : \mathbb{N} \to \mathbb{N}$, the repeater from a of f, denoted by $f^{\mathcal{R}}_a$, is a function $\mathbb{N} \to \mathbb{N}$ such that $f^{\mathcal{R}}_a(n) = f^{(n)}(a)$.*

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

This allows simple and function-oriented definitions of hyperoperations and the Ackermann function that we give below. Note that the Curried $a[n-1]$ denotes the single-variable function $\lambda b.a[n-1]b$.

$$a[n]b \triangleq \begin{cases} b+1 & \text{when } n = 0 \\ a[n-1]^{\mathcal{R}}_{a_{n-1}}(b) & \text{otherwise} \end{cases}$$

$$\text{where } a_n \triangleq \begin{cases} a & \text{when } n = 0 \\ 0 & \text{when } n = 1 \\ 1 & \text{otherwise} \end{cases}$$

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

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

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

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

3 Inverses via Countdown

Many functions on $\mathbb{R}$ are bijections and thus have an intuitive inverse. Functions on $\mathbb{N}$ are often non-bijections and so their inverses do not come as naturally.

3.1 Upper Inverses, Expansions, and Repeatability

Definition 4. *The upper inverse of F, denoted by F^{-1}, is $\lambda n.\, \min\{m : F(m) \geq n\}$.*

This is well-defined if F is unbounded, *i.e.* $\forall b.\ \exists a.\ b \leq F(a)$. However, it only really serves as an "inverse" if F is strictly increasing, *i.e.* $\forall n, m.\ n < m \Rightarrow F(n) < F(m)$, which is a rough analogue of injectivity in the discrete domain.

We call this 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 $\mathbb{R}$. The following clarifies this further:

Theorem 5 (a Galois connection). *If $F : \mathbb{N} \to \mathbb{N}$ is increasing, then f is the upper inverse of F if and only if $\forall n, m.\ f(n) \leq m \Leftrightarrow n \leq F(m)$.*

Proof. Fix n. Then $\forall m.\ f(n) \leq m \Leftrightarrow n \leq F(m)$ implies:

1. $f(n)$ is a lower bound to $\{m : n \leq F(m)\}$
2. $f(n)$ is itself in the set, since plugging in $f(n)$ for m yields $n \leq F(f(n))$, which makes f the upper inverse of F.

Conversely, if f is the upper inverse of F, we know that $\forall m.\ n \leq F(m) \Rightarrow f(n) \leq m$. Finally, by increasing-ness, $\forall m \geq f(n).\ F(m) \geq F(f(n)) \geq n$. □

Corollary 6 (property of Galois connections). *If F is strictly increasing, then $F^{-1} \circ F$ is the identity function.*

Proof. Proceed by antisymmetry. By ($\Leftarrow$) of Theorem 5, $F(n) \leq F(n)$ implies $(F^{-1} \circ F)(n) \leq n$. Next, by ($\Rightarrow$) of the same theorem, $(F^{-1} \circ F)(n) \leq (F^{-1} \circ F)(n)$ implies that $F(n) \leq F\big((F^{-1} \circ F)(n)\big)$. We know F is strictly increasing, so we get $n \leq (F^{-1} \circ F)(n)$. □

Remark 7. *Definition 4, Theorem 5, and Corollary 6 encompass the primary reasoning framework we provide to clients of our inverse functions.*

Our inverses require increasing functions, and our definitions of hyperoperations/Ackermann use repeater. But if F is strictly increasing, $F^{\mathcal{R}}_a$ is not necessarily strictly increasing. For example, the identity function `id` is strictly increasing, but $\texttt{id}^{\mathcal{R}}_a(n) = (\texttt{id} \circ \ldots \circ \texttt{id})(a) = a$ is a constant function. We need something stronger.

Definition 8. *A function $F : \mathbb{N} \to \mathbb{N}$ is an expansion if $\forall n.\ F(n) \geq n$. Further, given $a \in \mathbb{N}$, an expansion F is strict from a if $\forall n \geq a.\ F(n) > n$.*

If $a \geq 1$ and F is an expansion *strict from a*, we quickly get: $\forall n.\ F^{\mathcal{R}}_a(n) = F^{(n)}(a) \geq a + n \geq 1 + n$. That is, $F^{\mathcal{R}}_a$ is itself an expansion strict from 0.

Definition 9. *Suppose $a \geq 1$, and F has two properties: is a strictly increasing function and it is also a strict expansion from a. Then $F_a^{\mathcal{R}}$ "preserves" the two properties of F. As noted above, the only difference is that $F_a^{\mathcal{R}}$ is a strict expansion from 0 instead of from a. We thus say that a function F with these two properties is* repeatable from a. *We denote the set of functions repeatable from a as REPT_a. It is straightforward to see that $\forall s, t.\ s \leq t \Rightarrow \text{REPT}_s \subseteq \text{REPT}_t$.*

3.2 Contractions and the Countdown Operation

Suppose $F \in \text{REPT}_a$ for some $a \geq 1$, and let $f \triangleq F^{-1}$, *i.e.* the inverse of F. Our goal is to use f to compute an inverse to $F_a^{\mathcal{R}}$. From the preceeding discussion we know that this inverse must exist, since $F \in \text{REPT}_a$ implies $F_a^{\mathcal{R}} \in \text{REPT}_0$. For reasons that will be clear momentarily, we write this inverse as $f_a^{\mathcal{C}}$. Fix n and observe that $\forall m.\ f^{(m)}(n) \leq a \Leftrightarrow m \geq f_a^{\mathcal{C}}(n)$ since

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

Setting m as $f_a^{\mathcal{C}}(n)$ gives us $f^{(f_a^{\mathcal{C}}(n))}(n) \leq a$. Together these say that $f_a^{\mathcal{C}}(n)$ is the smallest number of times f needs to be compositionally applied to n before the result equals or passes below a. In other words, count the length of the chain $\{n, f(n), f^{(2)}(n), \ldots\}$ that terminates as soon a member of the chain equals or passes below a. This strategy only really works if each chainlink is strictly smaller than the previous, *i.e.* f is a *contraction*:

Definition 10. *A function $f : \mathbb{N} \to \mathbb{N}$ is a* contraction *if $\forall n.\ f(n) \leq n$. Next, if given some $a \geq 1$, a contraction f is* strict above *a if $\forall n > a.\ f(n) < n$. We denote the set of contractions by CONT and the set of contractions strict above a by CONT_a. Analogously to our observation in Definition 9, $\forall s \leq t.\ \text{CONT}_s \subseteq \text{CONT}_t$.*

Next, observe that all expansions have contractive inverses:

Theorem 11. *$\forall a \in \mathbb{N}.\ F \in \text{REPT}_a \Rightarrow F^{-1} \in \text{CONT}_a$.*

Proof. $\forall n.\ F(n) \geq n \Rightarrow n \geq F^{-1}(n)$, so $F^{-1} \in \text{CONT}$. Note, $n > a \Leftrightarrow n - 1 \geq a$. Since $F \in \text{REPT}_a$, $F(n-1) \geq n$ holds. Finally, $n - 1 \geq F^{-1}(n) \Rightarrow n > F^{-1}(n)$. □

This clarifies the inverse relationship between expansions strict from some a and contractions strict above that same a. The inverse of an expansion's repeater exists, and can be built from the expansion's inverse. To this end we introduce *countdown*, a new iterative technique in the spirit of *repeater*.

Definition 12. *Let $f \in \text{CONT}_a$. The countdown to a of f, written $f_a^{\mathcal{C}}(n)$, is the smallest number of times f needs to be applied to n for the answer to equal or go below a.* i.e., *$f_a^{\mathcal{C}}(n) \triangleq \min\{m : f^{(m)}(n) \leq a\}$.*

Inspired by Equation 3, we provide a neat, algebraically manipulable logical sentence equivalent to Definition 12, which is more useful later in our paper:

Corollary 13. *If $a \in \mathbb{N}$ and $f \in \text{CONT}_a$, then we have $\forall n, m.\ f_a^{\mathcal{C}}(n) \leq m \Leftrightarrow f^{(m)}(n) \leq a$.*

Proof. Fix a and n. The interesting direction is ($\Rightarrow$). Suppose $f_a^{\mathcal{C}}(n) \leq m$, we get $f^{(m)}(n) \leq f^{(f_a^{\mathcal{C}}(n))}(n)$ due to $f \in \text{CONT}$, and $f^{(f_a^{\mathcal{C}}(n))}(n) \leq a$ due to Definition 12. □

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

Theorem 14. *$\forall a \in \mathbb{N}$ and $f \in \text{CONT}_a$, $f_a^{\mathcal{C}}$ satisfies:*

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

Proof. In the case when $n \leq a$, use Corollary 13 as follows: $n = f^{(0)}(n) \leq a \Leftrightarrow f_a^{\mathcal{C}}(n) \leq 0$. When $n > a$, proceed by antisymmetry. Define $m \triangleq f_a^{\mathcal{C}}(f(n))$, and note that Corollary 13 gives $f_a^{\mathcal{C}}(n) \leq 1 + m \Leftrightarrow f^{(1+m)}(n) \leq a$. Next, a simple expansion of the last clause gives $f^{(m)}(f(n)) \leq a$, and unfolding the definition of m shows that this last clause is true. Now since $n > a$, we have $f_a^{\mathcal{C}}(n) \geq 1$ by the above. Define $p \triangleq f_a^{\mathcal{C}}(n) - 1$. It remains to show $f_a^{\mathcal{C}}(f(n)) \leq p \Rightarrow f^{(p)}(f(n)) \leq a \Rightarrow f^{(p+1)}(n) \leq a$. This holds by unfolding the definition of p. □

3.3 A Coq Computation of Countdown

The higher-order repeater function is well-defined for all 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 §2. In contrast, a *countdown* only exists for certain functions, most conveniently contractions. This restriction makes it a little harder to encode into Coq. Our strategy is to first define a *countdown worker* which is structurally recursive and is thus more palatable to Coq, and to then prove that this worker correctly computes the countdown when it is passed a contraction.

Definition 15. *$\forall a \in \mathbb{N}$ and $f : \mathbb{N} \to \mathbb{N}$, the* countdown worker *to a of f is a function $f_a^{\mathcal{CW}} : \mathbb{N}^2 \to \mathbb{N}$ such that:*

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

The worker operates on two arguments: the *true argument* n, for which we want to simulate *countdown to a*, and the *budget* b, the maximum number of times we shall attempt to compositionally apply f on the input before giving up. If the input goes below or equals 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 the point is that it can clearly be written in Coq:

```
Fixpoint cdn_wkr a f n b :=
  match b with 0 => 0 | S b' =>
    if (n <=? a) then 0
                 else S (cdn_wkr f a (f n) k')
  end.
```

Given an f that is a contraction strict from a, and given a sufficient budget, `cdn_wkr` will compute the correct *countdown* value. Careful readers may have noticed that *budget* is similar to *gas*, which we discussed in §1.2 and dismissed for potentially being too computationally expensive to calculate. Budget is actually a refinement of gas because we always accompany its use with an efficient calculation for it. In fact, we will soon show that a budget of n is sufficient in this case. This lets us define *countdown* in Coq for the first time:

Definition 16. *Redefine* $f_a^C(n) \triangleq f_a^{C\mathcal{W}}(n,n)$.

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

The computational $f_a^C(n)$ finds the same value as the theoretical Definition 12:

Theorem 17. *We can show that* $\forall a \in \mathbb{N}.\ \forall f \in \mathit{CONT}_a$, *we can show that* $\forall n.\ f_a^C(n) = \min\left\{i : f^{(i)}(n) \le a\right\}$.

Proof outline. Assume the initial arguments (n, b), and let the original call $f_a^{C\mathcal{W}}(n,b)$ be the 0^{th} call. A straightforward induction on i shows that in the i^{th} call, the arguments will be $\left(f^{(i)}(n), b-i\right)$, while an accumulated amount i will have been added to the final result. The only exception is the last call, which will return 0 and add nothing to the result.

Suppose $b = n$, and let $m \triangleq \min\left\{i : f^{(i)}(n) \le a\right\}$[4]. Then $m \le n$ since $f^{(n)}(n) \le a$. Thus, before the budget is exhausted, the function reaches the m^{th} call. This is actually the last recursive call since $\left(f^{(m)}(n), n-m\right)$ satisfies the terminating condition $f^{(m)}(n) \le a$. This last call adds 0 to the accumulated result, which is m at the moment. Therefore $f_a^{C\mathcal{W}}(n,n) = m$. □

We give an extended version of this proof in Appendix A, and mechanize it here. Theorem 17 and (3) establish the correctness of the Coq definitions of *countdown worker* and *countdown*, thereby justifying our budget of n and our unification of Definitions 12 and 16. We wrap everything together with the following theorem:

Theorem 18. $\forall F \in \mathit{REPT}_a.\ f \triangleq F^{-1}$ *satisfies* $f \in \mathit{CONT}_a$ *and* $f_a^C = \left(F_a^{\mathcal{R}}\right)^{-1}$. *Furthermore, if* $a \ge 1$, *then* $F_a^{\mathcal{R}} \in \mathit{REPT}_0$ *and* $f_a^C \in \mathit{CONT}_0$.

Proof. By Theorem 11, $f \triangleq F^{-1} \in \text{CONT}_a$. Corollary 13 and (3) then show that $f_a^C = \left(F_a^{\mathcal{R}}\right)^{-1}$. If $a \ge 1$, a simple induction shows that $F^{(n)}(a) \ge a + n \ge 1 + n$, so $F_a^{\mathcal{R}} \in \text{REPT}_0$. Hence $f_a^C = \left(F_a^{\mathcal{R}}\right)^{-1} \in \text{CONT}_0$ by Theorem 11. □

[4] We prove the existence of the min in Coq's intuitionistic logic here.

4 Inverting Hyperoperations and Ackermann

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

4.1 Inverse Hyperoperations, Including `div`, `log`, and `log*`

Definition 19. *The inverse hyperoperations, written* $a\langle n\rangle b$, *are defined as:*

$$a\langle n\rangle b \triangleq \begin{cases} b-1 & \text{if } n = 0 \\ a\langle n-1\rangle_{a_n}^C(b) & \text{if } n \ge 1 \end{cases} \quad \text{where } a_n = \begin{cases} a & \text{if } n = 1 \\ 0 & \text{if } n = 2 \\ 1 & \text{if } n \ge 3 \end{cases} \tag{4}$$

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

where the Curried $a\langle n-1\rangle$ denotes the single-variable function $\lambda b.a\langle n-1\rangle b$. We now show that $a\langle n\rangle$ is the inverse to $a[n]$. First note that $a\langle 0\rangle \in \text{CONT}_0$. Then:

Lemma 20. $\forall a.\ \forall b.\ a\langle 1\rangle b = b - a$.

Proof. Theorem 14 applies because $a\langle 0\rangle \in \text{CONT}_0 \subseteq \text{CONT}_a$, giving the intermediate step shown below. Thereafter we have $a\langle 1\rangle b = b - a$ by induction on b.

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

□

Corollary 21. $\forall a \ge 1, a\langle 1\rangle \in \mathit{CONT}_1$.

N.B. $a\langle n\rangle b$ is a total function, but it is never actually used 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 22. *When* $n \le 1$, *or* $n \le 2 \wedge a \ge 1$, *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 and Q as:

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

We have three goals:

1. $\forall a.\ Q(0) \wedge Q(1)$
2. $\forall a \ge 1.\ Q(2)$
3. $\forall a \ge 2.\ Q(n)$

Note that $\forall n.\ a\langle n+1\rangle = a\langle n\rangle_{a_n}^C$ and $a[n+1] = a[n]_{a_n}^{\mathcal{R}}$. By Theorem 18,

$$P(n) \;\Rightarrow\; Q(n) \;\Rightarrow\; Q(n+1) \tag{5}$$

$$a_n \ge 1 \;\Rightarrow\; P(n) \;\Rightarrow\; P(n+1) \tag{6}$$

Goal 1: $P(0) \Leftrightarrow \lambda b.(b+1) \in \text{REPT}_a$ and $Q(0) \Leftrightarrow a\langle 0\rangle = (a[0])^{-1}$ $\Leftrightarrow (b - 1 \le c \Leftrightarrow b \le c + 1)$. These are both straightforward, and $Q(1)$ holds by (5). Goal 2: we have $a \ge 1$, so $P(1)$ holds by $P(0)$ and (6), and $Q(2)$ holds by $Q(1)$ and (5). Goal 3: we have $a \ge 2$, and using (5) and $Q(0)$ reduces the goal to $P(n)$. Using (6) and the fact that $\forall n \ne 1.\ a_n \ge 1$, the goal reduces to $P(2)$. This unfolds to:

$$a[2] \in \text{REPT}_0 \Leftrightarrow \forall b < c.\ ab < ac \quad \wedge \quad \forall b \ge 1.\ ab \ge b + 1,$$

which is straightforward for $a \ge 2$. Induction on n gives us the third goal. □

Remark 23. *Three early hyperoperations are $a[2]b = ab$, $a[3]b = a^b$ and $a[4]b = {}^b a$, so, by Theorem 22, 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 but are one-liners for us:*

```
Definition divc a b := inv_hyperop a 2 b.
Definition logc a b := inv_hyperop a 3 b.
Definition logstar a b := inv_hyperop a 4 b.
```

4.2 The Inverse Ackermann Hierarchy

Next, we want to use *countdown* to build the *inverse Ackermann hierarchy*, where each level α_i inverts the level $\mathcal{A}_i$. We know $\mathcal{A}_{i+1} = {\mathcal{A}_i}^{\mathcal{R}}_{\mathcal{A}_i(1)}$, so the recursive rule $\alpha_{i+1} \triangleq {\alpha_i}^{C}_{\mathcal{A}_i(1)}$ is tempting. But this approach is flawed because it still depends on $\mathcal{A}_i$. Instead, we reexamine the inverse relationship: suppose $\alpha_i = (\mathcal{A}_i)^{-1}$ and $\alpha_{i+1} = (\mathcal{A}_{i+1})^{-1}$. Then $\mathcal{A}_{i+1}(m) = (\mathcal{A}_i)^{(m)}(\mathcal{A}_i(1))$. We then have:

$$\alpha_{i+1}(n) \le m \Leftrightarrow n \le (\mathcal{A}_i)^{(m+1)}(1) \Leftrightarrow (\alpha_i)^{(m+1)}(n) \le 1 \quad (7)$$

Equivalently, $\alpha_{i+1}(n) = \min\{m : (\alpha_i)^{(m+1)}(n) \le 1\}$, or $\alpha_{i+1}(n) = {\alpha_i}^{C}_{1}(\alpha_i(n))$. From (7) we can thus define the inverse Ackermann hierarchy:

Definition 24. $\alpha_i \triangleq \begin{cases} \lambda n.(n-1) & \text{if } i = 0 \\ ({\alpha_{i-1}}^{C}_{1}) \circ \alpha_{i-1} & \text{if } i \ge 1 \end{cases}$

Corollary 25. *Applying our Galois connection (Theorem 5) to the above gives us an important inverse relation for each level of the Ackermann hierarchy:*

$$n \le \mathcal{A}_j(m) \Leftrightarrow \alpha_j(n) \le m$$

Note that we are now inverting the j^{th} level of the hierarchy, and not the diagonal Ackermann function itself.

To invert the diagonal, recall Definition 2: minimize k such that $n \le \mathcal{A}(k)$, *i.e.* $\alpha(n) \triangleq \min\{k : n \le \mathcal{A}(k)\}$. Unfolding the definition of $\mathcal{A}$ yields $\min\{k : n \le \mathcal{A}_k(k)\}$, and then by applying Corollary 25 we can—Abracadabra!—express the inverse of the diagonal Ackermann α using the hierarchy *without referring to $\mathcal{A}$ itself*:

Theorem 26. $\forall n.\ \alpha(n) = \min\{k : \alpha_k(n) \le k\}$.

Table 2. Intuition for $\alpha(n)$ defined without $\mathcal{A}(n)$.

α_k \ n	0	1	2	3	4	5	6	7	8	9
α_0	0	0	1	2	3	4	5	6	7	8
α_1	0	0	0	1	2	3	4	5	6	7
α_2	0	0	0	0	1	1	2	2	3	3
α_3	0	0	0	0	0	0	1	1	1	1

Table 2 illustrates how this new definition points to an efficient algorithm. The values come from Table 1: $\alpha_0(n)$ is $n-1$, $\alpha_1(n)$ is $n-2$, $\alpha_2(n)$ is $\lceil \frac{n-3}{2} \rceil$, and $\alpha_3(n)$ is $\lceil \log_2(b+3) \rceil - 3$. For any n, search down its k-indexed column, looking for the smallest k such that $\alpha_k(n) \le k$. Here we show show unsuccessful searches in red and successful searches in green. *e.g.* consider $n = 8$. $\alpha_0(8) = 7$. Because $7 \not\le 0$, α_0 is rejected. Similarly $\alpha_1(8) = 6 \not\le 1$ and $\alpha_2(8) = 3 \not\le 2$ are rejected. Finally $\alpha_3(8) = 1$ is accepted because $1 \le 3$. Indeed, $\alpha(8) = 3$.

Now all that remains is to provide a structurally-recursive function that computes α.

Definition 27. *The inverse Ackermann worker, written $\alpha^{\mathcal{W}}$, is a function from $\mathbb{N}^4$ to $\mathbb{N}$ defined as:*

$$\alpha^{\mathcal{W}}(f, n, k, b) \triangleq \begin{cases} k & \text{if } b = 0 \vee n \le k \\ \alpha^{\mathcal{W}}(f_1^C \circ f, f_1^C(n), k+1, b-1) & \text{if } b \ge 1 \wedge n \ge k+1 \end{cases} \quad (8)$$

Next, we show that this function computes the inverse Ackermann function when passed appropriate arguments.

Theorem 28. $\forall n.\ \alpha^{\mathcal{W}}(\alpha_0, \alpha_0(n), 0, n) = \alpha(n)$.

Proof outline. When given the arguments $(\alpha_i, \alpha_i(n), i, b-i)$ such that $\alpha_i(n) > i$ and $b > i$, $\alpha^{\mathcal{W}}$ takes on the arguments $(\alpha_{i+1}, \alpha_{i+1}(n), i+1, b-(i+1))$ at the next recursive call. A simple induction on k then shows that if $k \le \min\{b, \alpha_k(n)\}$,

$$\alpha^{\mathcal{W}}(\alpha_0, \alpha_0(n), 0, b) = \alpha^{\mathcal{W}}(\alpha_k, \alpha_k(n), k, b-k) \quad (9)$$

Let $m \triangleq \min\{k : \alpha_k(n) \le k\}$. Then $m \le n$ since $\alpha_n(n) \le n$. (9) then implies:

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

□

We put a mathematical proof of correctness in Appendix B, and a mechanized proof here. We thus have a (re-)definition of inverse Ackermann that is definable via a Coq-accepted worker, *i.e.* $\alpha(n) \triangleq \alpha^{\mathcal{W}}(\alpha_0, \alpha_0(n), 0, n)$. We present the inverse Ackermann function in Gallina:

```
Fixpoint inv_ack_wkr f n k b :=
  match b with 0 => 0 | S b' =>
    if (n <=? k) then k
      else let g := (countdown_to f 1) in
        inv_ack_wkr (compose g f) (g n) (S k) b
  end.
```

```
Fixpoint alpha m x :=
  match m with 0 => x - 1 | S m' =>
    countdown_to 1 (alpha m') (alpha m' x)
  end.

Definition inv_ack :=
  inv_ack_wkr (alpha 0) (alpha 0 n) 0 n.
```

Note that this is not the linear-time computation we presented in Figure 1. We will arrive at that code via an improvement discussed in the next section.

5 Time Bound of Our Inverses

We show that the inverse Ackermann from Definition 24 runs in $\Omega(n^2)$ under a call-by-value strategy, and in $O(n)$ after a simple optimization. To remain accessible to readers, we give intuition by presenting a detailed sketch via lemma statements. We put full proofs of these lemmas in Appendix C of the extended version of this paper [22]. We elide the parallel analysis for the inverse hyperoperations, which is easier thanks to simpler initial values but are otherwise similar.

N.B. For this section, all of our functions take inputs in `nat`, *i.e.* in unary encoding. In §6 we will move to functions operating on `N`, *i.e.* in binary encoding.

Definition 29. *For a function f on k variables, the* runtime *of f, denoted by $\mathcal{T}_f(n_1, n_2, \ldots, n_k)$, counts the computational steps to compute $f(n_1, n_2, \ldots, n_k)$.*

The next lemma establishes the general runtime structure of *countdown* when its input is encoded in `nat`.

Lemma 30. $\forall a \geq 1,\ \forall n \in \texttt{nat},\ \forall f \in \mathit{CONT}_a,$

$$\mathcal{T}_{f_a^C}(n) = \sum_{i=0}^{f_a^C(n)-1} \mathcal{T}_f\left(f^{(i)}(n)\right) + (a+2) f_a^C(n) + f^{\left(f_a^C(n)\right)}(n) + 1$$

5.1 Naïve α_i on `nat` is $\Omega(n^2)$; Optimized α_i is $O(n)$

We start with the following lemma about the running time of each α_i, which is a consequence of Lemma 30. Its full proof can be found in Appendix C.

Lemma 31. *When α_i is defined per Definition 24,*

$$\mathcal{T}_{\alpha_{i+1}}(n) = \sum_{k=0}^{\alpha_{i+1}(n)} \mathcal{T}_{\alpha_i}\left(\alpha_i^{(k)}(n)\right) + 3\alpha_{i+1}(n) + C_i(n),$$

where $\forall i, n.\ C_i(n) \triangleq \alpha_i^{(\alpha_{i+1}(n)+1)}(n) + 1 \in \{1, 2\}$

Crucially, this lemma implies $\mathcal{T}_{\alpha_{i+1}}(n) \geq \mathcal{T}_{\alpha_i}(n)$. Given some index i such that $\mathcal{T}_{\alpha_i}(n) = \Omega(n^2)$, each function after α_i in the hierarchy will take at least $\Omega(n^2)$ time, thus making $\mathcal{T}_{\alpha_i}(n) = \Omega(n^2)$. In fact, $i = 2$ suffices.

Lemma 32. $\forall i \geq 2.\ \mathcal{T}_{\alpha_i}(n) = \Omega(n^2)$.

Remark 33. *Although $\alpha_1(n)$ always returns $n-2$, it gets to this answer via $\Theta(n)$ steps due to the nature of* countdown. *This hurts the performance of the entire hierarchy. We can hardcode α_1 as $\lambda n.(n-2)$ to reduce its runtime from $\Theta(n)$ to $\Theta(1)$. The runtime for α_2 can thus be bounded to $O(n)$:*

$$\begin{aligned}\mathcal{T}_{\alpha_2}(n) &\leq \textstyle\sum_{i=0}^{\lceil \frac{n-3}{2} \rceil} \mathcal{T}_{\alpha_1}(n-2i) + 3\left\lceil \tfrac{n-3}{2} \right\rceil + 2 = 4\left\lceil \tfrac{n-3}{2} \right\rceil + 2 \\ &\leq 2n - 2 \;=\; O(n)\end{aligned}$$

Note that the above bound only applies for $n \geq 2$. When $n \leq 1$, the left hand side is 1.

This simple optimization cascades through the entire hierarchy: we can now prove a bound of $O(n)$ for every level. Hardcoding α_1 as $\lambda n.(n-2)$ gives the ith level of the inverse Ackermann hierarchy in $O\left(n + 2^i \log n + i\right)$ time.

Theorem 34. *When α_i is defined per Definition 24 with the added hardcoding of α_1 to $\lambda n.(n-2)$, we have:*

$$\forall i.\ \mathcal{T}_{\alpha_i}(n) \leq 4n + \left(19 \cdot 2^{i-3} - 2i - 13\right) \log_2 n + 2i = O(n).$$

5.2 Running Time of α on `nat`

A linear-time calculation of the inverse Ackermann hierarchy leads to a similarly efficient calculation of the inverse Ackermann function itself. We present $\tilde{\alpha}(n)$ below, along with a Coq encoding. The function `inv_ack_linear` is precisely the function we presented as a sneak peek in Figure 1.

Definition 35. $\tilde{\alpha}(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ \alpha^{\mathcal{W}}\left(\alpha_1, \alpha_1(n), 1, n-1\right) & \text{if } n \geq 2 \end{cases}$

```
Definition inv_ack_linear n :=
  match n with 0 | 1 => 0 | _ =>
    let f := (fun x => x - 2) in
      inv_ack_wkr f (f n) 1 (n - 1)
  end.
```

Theorem 36. *$\forall n.\ \tilde{\alpha}(n) = \alpha(n)$ and (benchmark) $\mathcal{T}_{\tilde{\alpha}}(n) = O(n)$.*

Proof. Correctness: the cases $n = 0, 1$ are trivial. For $n > 1 = \mathcal{A}(0)$, $\alpha_1(n) > 1$ and $n - 1 > 0$, so $\alpha^{\mathcal{W}}\left(\alpha_0, \alpha_0(n), 0, n\right) = \alpha^{\mathcal{W}}\left(\alpha_1, \alpha_1(n), 1, n-1\right)$ by Definition 27. Theorem 28 then implies $\tilde{\alpha}(n) = \alpha(n)$.

Complexity: at each recursive step, the transition from $\alpha^{\mathcal{W}}\left(\alpha_k, \alpha_k(n), k, n-k\right)$ to $\alpha^{\mathcal{W}}\left(\alpha_{k+1}, \alpha_{k+1}(n), k+1, n-k-1\right)$ consists of two steps: First, calculation of $\texttt{leb}\left(\alpha_k(n), k\right)$, which takes no more than $k+1$ steps (Appendix C). Second, calculation of $\alpha_{k+1}(n) = {\alpha_k}_1^C(x)$ given $x \triangleq \alpha_k(n)$, which takes time $\mathcal{T}_{\alpha_{k+1}}(n) - \mathcal{T}_{\alpha_k}(n)$ (Appendix C). The computation will terminate at $k = \alpha(n)$. Thus, $\forall n \geq 1$,

$$\begin{aligned}\mathcal{T}_{\tilde{\alpha}}(n) &\leq \mathcal{T}_{\alpha_1}(n) + \textstyle\sum_{k=1}^{\alpha(n)-1} \left[\mathcal{T}_{\alpha_{k+1}}(n) - \mathcal{T}_{\alpha_k}(n)\right] + \sum_{k=1}^{\alpha(n)} (k+1) \\ &\leq \mathcal{T}_{\alpha_{\alpha(n)}}(n) + \alpha(n)^2 \\ &= O\left(n + 2^{\alpha(n)} \log_2 n + \alpha(n)^2\right) = O(n)\end{aligned}$$

□

N.B. The hyperlink "𝔸 *(benchmark)*" above leads to experimental timebound tests of the code, which we will discuss further in §7. We provide this benchmark and the theoretical argument above, but do not prove the asymptotic bounds of the code in Coq.

6 Inputs Encoded in Binary

Thus far we have used the Coq type `nat`, which represents a number n using n bits. In contrast, the binary system represents n in $\lfloor \log_2 n \rfloor + 1$ bits. Coq comes with a built-in binary type `N`, which consists of constructors `N0` and `Npos`. The latter, `Npos`, unfolds to `positive`:

```
Inductive positive : Set :=
  | xI : positive -> positive
  | xO : positive -> positive
  | xH : positive.
```

Constructor `xH` represents 1, and constructors `xO` and `xI` represent appending 0 and 1 respectively. By always starting with 1, `positive` bypasses the issue of disambiguating *e.g.* the strings `011` and `00011`, which represent the same number but pose a minor technical challenge. To represent 0, the type `N` simply uses the separate constructor `N0`.

In both `nat` and `N`, addition/subtraction of b-bit numbers is $\Theta(b)$, while multiplication is $\Theta(b^2)$. In general, arithmetic operations are often faster when the inputs are encoded in binary. In this section we show that this advantage also extends to our techniques.

Our codebase has binary versions of hyperoperations, inverse hyperoperations, Ackermann, and inverse Ackermann. Here we show how to compute inverse Ackermann for binary inputs in $\Theta(b)$ time, where b is the bitlength, *i.e.* logarithmic time in the input magnitude. As before, we present an intuitive sketch here and put full proofs in Appendix D of the extended version of this paper [22].

Remark 37. *Although we do not prove it here, our general binary inverse hyperoperations are $O(b^2)$ time, since Ackermann and base-2 hyperoperations benefit from $\Theta(1)$ division via bitshifts, whereas general division is $O(b^2)$.*

6.1 Countdown and Contractions in Binary

Although the theoretical *countdown* is independent of the encoding of its inputs, its Coq definition needs to be adjusted to allow for inputs in `N`. The first step is to translate the arguments of `countdown_worker` from `nat` to `N`. Budget `b` must remain in `nat` so it can serve as Coq's termination argument, but all other `nat` arguments should be changed to `N`, and functions on `nat` to functions on `N`.

```
Fixpoint bin_cdn_wkr f a n b : N :=
  match b with O => 0 | S b' =>
    if (n <=? a) then 0
      else 1 + bin_cdn_wkr f a (f n) b'
  end.
```

Determining the budget for `bin_countdown_to` is tricky. A naïve approach is to use the built-in `nat` translation of `n`, *i.e.* `N.to_nat n`. This is untenable as the translation alone takes exponential time *viz* the length of n's representation. We need a linear-time budget calculation for countdowns of oft-used functions like $\lambda n.(n-2)$.

The key is to focus on functions that can bring their arguments below a threshold via repeated application in *logarithmic* time, thus allowing a log-sized budget for `bin_cdn_wkr`. Simply shrinking by 1 is no longer good enough; we need to halve the argument on every application as shown below:

Definition 38. *$f \in$ CONT is* binary strict above *$a \in \mathbb{N}$ if* $\forall n > a, f(n) \leq \lfloor \frac{n+a}{2} \rfloor$.

The key advantage of binary strict contractions is that if a contraction f is binary strict above some a, then we know that $\forall n > a, \forall k.\ f(n) \leq \left\lfloor \frac{n-a}{2^k} \right\rfloor + a$. Therefore, within $\lfloor \log_2(n-a) \rfloor + 1$ applications of f, the result will become equal to or less than a. We can choose this number as the budget for `bin_cdn_wkr` to successfully reach the countdown value before terminating. Note that this budget is simply the length of the binary representation of `n - a`, which we calculate using our function `nat_size`. The Coq definition of countdown on `N` is:

```
Definition bin_countdown_to f a n :=
  bin_cdn_wkr f a n (nat_size (n - a)).
```

The following is the binary version of Lemma 30:

Lemma 39. *$\forall n \in$ `N` $\forall a \in$ `N`, if f is a binary strict contraction above a,*

$$\mathcal{T}_{f_a^C}(n) \leq \sum_{i=0}^{f_a^C(n)-1} \mathcal{T}_f\left(f^{(i)}(n)\right) + (\log_2 a + 3)\left(f_a^C(n) + 1\right) + 2\log_2 n + \log_2 f_a^C(n)$$

6.2 Inverse Ackermann in O $(\log_2 n)$

Our new Coq definition computes countdown only for strict binary contractions. Fortunately, starting from $n = 2$, the inverse hyperoperations $a\langle n \rangle b$ when $a \geq 2$ and the inverse Ackermann hierarchy α_n are all strict binary contractions. We can construct these hierarchies by hardcoding their first three levels and recursively building higher levels with `bin_countdown_to`. Furthermore, analagously to the optimization for `nat` discussed in §5.1, we hardcode an additional level.

```
Fixpoint bin_alpha (m : nat) (x : N) : N :=
  match m with
  | 0%nat => x - 1
  | 1%nat => x - 2
  | 2%nat => N.div2 (x - 2)
  | 3%nat => N.log2 (x + 2) - 2
  | S m'  => bin_countdown_to
               (bin_alpha m') 1 (bin_alpha m' x)
  end.
```

Note that for all x, $\texttt{N.div2}(x-2) = \left\lfloor \frac{x-2}{2} \right\rfloor = \left\lceil \frac{x-3}{2} \right\rceil$ and $\texttt{N.log2}(x+2)-2 = \left\lfloor \log_2(x+2) \right\rfloor - 2 = \left\lceil \log_2(x+3) \right\rceil - 3$, so the above Coq definition is correct.

Theorem 40.

$$\forall i, n.\ \mathcal{T}_{\alpha_i}(n) \le 2\log_2 n + \left(3 \cdot 2^i - 3i - 13\right)\log_2\log_2 n + 3i.$$

For any level of the Ackermann hierarchy, this theorem demonstrates a linear computation time up to the size of the representation of the input, *i.e.* logarithmic time up to its magnitude n: $\mathcal{T}_{\alpha_i}(n) = \mathrm{O}\left(\log_2 n + 2^i \log_2\log_2 n\right)$.

Moving on to inverse Ackermann itself, we follow a style nearly identical to that in §4.2. For the worker, we simply translate to `N`, keeping the budget in `nat` as described earlier. The inverse Ackermann has an extra hardcoded level.

```
Fixpoint bin_inv_ack_wkr f n k b :=
  match b with 0%nat => k | S b' =>
    if n <=? k then k else
      let g := (bin_countdown_to f 1) in
        bin_inv_ack_wkr
          (compose g f) (g n) (N.succ k) b'
  end.

Definition bin_inv_ack n :=
  if (n <=? 1) then 0 else if (n <=? 3) then 1
    else if (n <=? 7) then 2 else
      let f := (fun x => N.log2 (x + 2) - 2) in
        bin_inv_ack_wkr f (f n) 3 (nat_size n).
```

Note that, for $n > 7$, $n < \mathcal{A}\left(\lfloor \log_2 n \rfloor + 1\right) = \mathcal{A}\left(\texttt{nat\_size}(n)\right)$, so a budget of $\texttt{nat\_size}(n)$ suffices. We show the correctness and benchmark of `bin_inv_ack` in our codebase. Figure 2 shows a standalone binary-specific computation of the inverse Ackermann function that takes logarithmic time up to the input's magnitude. This is simply an assimilation of the code snippets we have already discussed in this section, and serves as a binary translation of Figure 1 shown earlier.

As in Theorem 36, the time complexity $\mathcal{T}_\alpha(n)$ is the sum of each component's runtime:

$$\begin{aligned}\mathcal{T}_{\alpha_3}(n) \ &+ \sum_{k=3}^{\alpha(n)-1} \mathcal{T}_{\alpha_{k_1}^C}\left(\alpha_k(n)\right) + \sum_{k=3}^{\alpha(n)} \mathcal{T}_{\mathtt{N.leb}}\left(\alpha_k(n), k\right) \\ &+ \sum_{k=3}^{\alpha(n)-1} \mathcal{T}_{\mathtt{N.succ}}(k) \ + \ \mathcal{T}_{\mathtt{nat\_size}}(n) \\ &+ \mathcal{T}_{\mathtt{N.leb}}(n,1) \ + \ \mathcal{T}_{\mathtt{N.leb}}(n,3) \ + \ \mathcal{T}_{\mathtt{N.leb}}(n,7)\end{aligned}$$

With reference to Lemmas 48 (Appendix C), 53 and 54 (Appendix D), we have: In the second summand, $\mathcal{T}_{\alpha_{k_1}^C}\left(\alpha_k(n)\right) = \mathcal{T}_{\alpha_{k+1}}(n) - \mathcal{T}_{\alpha_k}(n)$ for each k by Lemma 48. By Lemma 53, each $\mathcal{T}_{\mathtt{N.leb}}$ in the third summand is $\Theta\left(\log_2 k\right)$, totalling $\mathrm{O}\left(\alpha(n)\log_2\alpha(n)\right) = \mathrm{o}\left(\log_2 n\right)$. The fourth summand is $\Theta(\alpha(n)) = \mathrm{o}\left(\log_2 n\right)$ by Lemma 54. The remaining

```
Require Import Omega Program.Basics.
Open Scope N_scope.

Definition nat_size n :=
  match n with
  | 0 => 0%nat
  | Npos p =>
      let fix nat_pos_size (x : positive) : nat :=
        match x with xH => 1%nat
        | xI y | xO y => S (nat_pos_size y) end
      in nat_pos_size p
  end.

Fixpoint bin_cdn_wkr f a n b : N :=
  match b with O => 0 | S b' =>
    if (n <=? a) then 0
      else 1 + bin_cdn_wkr f a (f n) b'
  end.

Definition bin_countdown_to f a n :=
bin_cdn_wkr f a n (nat_size (n - a)).

Fixpoint bin_inv_ack_wkr f n k b :=
  match b with 0%nat => k | S b' =>
    if n <=? k then k else
      let g := (bin_countdown_to f 1) in
        bin_inv_ack_wkr
          (compose g f) (g n) (N.succ k) b'
  end.

Definition bin_inv_ack n :=
  if (n <=? 1) then 0 else if (n <=? 3) then 1
    else if (n <=? 7) then 2 else
      let f := (fun x => N.log2 (x + 2) - 2) in
        bin_inv_ack_wkr f (f n) 3 (nat_size n).
```

Figure 2. A log-time Coq computation of the inverse Ackermann function for inputs represented in binary, *i.e.* `N`.

items total $\Theta(\log_2 n)$. Thus, $\forall n \ge 8$:

$$\begin{aligned}\mathcal{T}_\alpha(n) &= \mathcal{T}_{\alpha_3}(n) + \sum_{k=3}^{\alpha(n)-1} \left(\mathcal{T}_{\alpha_{k+1}}(n) - \mathcal{T}_{\alpha_k}(n)\right) + \Theta\left(\log_2 n\right) \\ &= \mathcal{T}_{\alpha_{\alpha(n)}}(n) + \Theta\left(\log_2 n\right) \\ &= \mathrm{O}\left(\log_2 n + 2^{\alpha(n)}\log_2\log_2 n\right) + \Theta\left(\log_2 n\right) \\ &= \Theta\left(\log_2 n\right)\end{aligned}$$

7 Further Discussion

7.1 The Value of a Linear-Time Solution to the Hierarchy

Our functions' linear runtimes can be understood in two distinct but complementary ways. A runtime less than the bitlength is impossible without prior knowledge of the size of the input. Accordingly, in an information-theory or pure-mathematical sense, our definitions are optimal up to constant factors. And of course in practice, linear-time solutions are highly usable in real computations.

Table 3. Benchmarking our results (all times in seconds).

	n	time in OCaml
n:nat	100	0.000020
	1000	0.000926
	10000	0.001450
	100000	0.016406
n:N	2^{100}	0.000012
	2^{1000}	0.000082
	2^{10000}	0.000959
	2^{100000}	0.014332

Sublinear solutions are possible with *a priori* knowledge about the function and bounds on the inputs one will receive. An extreme case is $\alpha(n)$, which has value 4 for all practical inputs greater than 61. Accordingly, this function can be inverted in $O(1)$ in practice. That said, such solutions require external knowledge of the problems and lookup tables within the algorithm to store precomputed values, and thus fall more into the realm of engineering than mathematics.

Remark 41. *Rather surprisingly, for binary-encoded numbers, it appears we can invert the Ackermann function asymptotically faster than we can calculate the base-3 logarithm.*

7.2 Benchmarking our Inverses

As shown in Table 3, our functions are very rapid in practice. Extracting our code to OCaml is straightforward: we import Coq's built-in `Extraction` module and then execute `Recursive Extraction inv_ack_linear.` We then benchmark our results using a machine running i5 2.4GHz with 8GB RAM. The displayed times are the average of three runs, and we suspect that garbage collection is the largest cause of variations in the wall-clock times. Our functions run quickly in Gallina as well, but exhibit unexpectedly rapid times when the inputs are large because Coq optimizes large `nat`s under the hood.

7.3 The Two-Parameter Inverse Ackermann Function

Some authors [4, 19] prefer a two-parameter inverse Ackermann function.

Definition 42. *The two-parameter inverse Ackermann function is defined as:*

$$\hat{\alpha}(m,n) \triangleq \min\left\{i \geq 1 : \mathcal{A}\left(i, \left\lfloor \frac{m}{n} \right\rfloor\right) \geq \log_2 n\right\} \tag{10}$$

Note that $\hat{\alpha}(n,n) \neq \alpha(n)$. However, it is straightforward to modify our techniques to compute $\hat{\alpha}(m,n)$.

```
theory inv_ack_standalone
  imports "HOL-Library.Log_Nat" HOL.Divides
begin

primrec cdn_wkr :: "(nat => nat) => nat =>
                    nat => nat => nat" where
  "cdn_wkr f a n 0 = 0" |
  "cdn_wkr f a n (Suc k) =
    (if n <= a then 0
               else Suc (cdn_wkr f a (f n) k))"

fun countdown_to :: "(nat => nat) => nat =>
                     nat => nat" where
  "countdown_to f a n = cdn_wkr f a n n"

primrec inv_ack_wkr :: "(nat => nat) => nat =>
                        nat => nat => nat" where
  "inv_ack_wkr f n k 0 = k" |
  "inv_ack_wkr f n k (Suc b) =
    (if n <= k then k
    else let g = (countdown_to f 1) in
    inv_ack_wkr (g o f) (g n) (Suc k) b)"

fun inv_ack_linear :: "nat => nat" where
  "inv_ack_linear 0 = 0" |
  "inv_ack_linear (Suc 0) = 0" |
  "inv_ack_linear (Suc (Suc n)) =
    inv_ack_wkr (λ x. (x - 2)) n 1 (Suc n)"

end
```

Figure 3. A linear-time Isabelle computation of the inverse Ackermann function (inputs in unary, *i.e.* `nat`).

Definition 43. *The two-parameter inverse Ackermann worker, written $\hat{\alpha}^{\mathcal{W}}$, is a function $\mathbb{N}^4 \to \mathbb{N}$, defined by:*

$$\hat{\alpha}^{\mathcal{W}}(f,n,k,b) \triangleq \begin{cases} 0 & \text{if } b = 0 \vee n \leq k \\ 1 + \hat{\alpha}^{\mathcal{W}}\left(f_1^{\mathcal{C}} \circ f, f_1^{\mathcal{C}}(n), k, b-1\right) & \text{if } b \geq 1 \wedge n \geq k+1 \end{cases} \tag{11}$$

Theorem 44. *We can calculate $\hat{\alpha}(m,n)$ from $\hat{\alpha}^{\mathcal{W}}$:*

$$\forall m, n.\ \hat{\alpha}(m,n) = 1 + \hat{\alpha}^{\mathcal{W}}\left(\alpha_1, \alpha_1\left(\lceil \log_2 n \rceil\right), \left\lfloor \frac{m}{n} \right\rfloor, \lceil \log_2 n \rceil\right).$$

We mechanize the above for both unary and binary inputs.

7.4 Implementing our Functions in Isabelle/HOL

To show that our functions are independent of any Coq idiosyncrasies, we implemented our functions in Isabelle/HOL. Despite having no previous experience with Isabelle/HOL, it took us only 2 hours to define our inverse Ackermann function. Figure 3 is an Isabelle translation of Figure 1.

Furthermore, our techniques are as *applicable* to Isabelle as they are to Coq: Isabelle offers `floorlog`, a discrete logarithm

```
definition ln :: "real => real" where
  "ln x = (THE u. exp u = x)"

definition log :: "[real,real] => real" where
  "log a x = ln x / ln a"

definition floorlog :: "nat => nat => nat" where
  "floorlog b a = (if a > 0 ∧ b > 1
                   then nat ⌊log b a⌋ + 1 else 0)"

lemma compute_floorlog[code]:
  "floorlog b x = (if x > 0 ∧ b > 1 then
              floorlog b (x div b) + 1 else 0)"
 by (* proof elided *)
```

Figure 4. The standard Isabelle technique for extractable discrete logarithm.

with arbitrary base, but `floorlog` is defined using `ln`, the continous logarithm on $\mathbb{R}$. This does not yield a computable function, and so the relevant Isabelle standard library uses a "hack" (in the form of a `[code]` annotation) to generate a computable equivalent for extraction. In Figure 4 we present relevant extracts from the Isabelle standard libraries [10, 21].

This hack requires a pre-developed computational strategy for the `[code]` extraction-substitution lemma: there ain't no such thing as a free lunch [9]. Thus, `[code]`-extracting the inverse Ackermann would require the Isabelle/HOL functions presented in Figure 3 or some equivalent thereof. Further, we found no definitions of $\log^*$ or the inverse Ackermann function in the Isabelle documentation. Our technique gives directly extractable solutions to the entire hierarchy.

8 Related Work

The Coq standard library has linear-time definitions of division and base-2 discrete logarithm on `nat` and `N`. The Mathematical Components library [13] has a discrete logarithm with arbitrary base, with inputs encoded in `nat`. The Isabelle/HOL Archive of Formal Proofs [10] provides a definition of discrete logarithm with arbitrary base along with a separTate computation strategy (`[code]`). None of these libraries provides a definition of iterated logarithm or further members of the hierarchy. Indeed, to our knowledge, we are the first to generalize this problem in a proof assistant, extend it both upwards and downwards in the natural hierarchy of functions, and provide linear-time computations.

8.1 Historical Notes

The operations of successor, predecessor, addition, and subtraction have been integral to counting forever. The ancient Egyptian number system used glyphs denoting 1, 10, 100, *etc.*, and expressed numbers using additive combinations of these. The Roman system is similar, but combines glyphs using both addition and subtraction. This buys brevity and readability, since *e.g.* 9_{roman} is two characters, "one less than ten", and not a series of nine 1s. The ancient Babylonian system was the first to introduce *algorisms*: the place value of a glyph succinctly expressed *how many times* it counted towards the number being represented, along with associated techniques for addition, multiplication, *etc.* The Babylonians operated in base 60, and so *e.g.* a three-gylph number $abc_{\text{babylonian}}$ could be parsed as $a \times 60^2 + b \times 60 + c$. Sadly they lacked a radix point, and so $a \times 60^3 + b \times 60^2 + c \times 60$, $a \times 60 + b + c \div 60$, *etc.* were also reasonable interpretations, and the correct number had to be inferred from context. Using multiplication and division in numerical representations bought great concision: a number n was represented in $\lfloor \log_{60} n \rfloor + 1$ glyphs. The modern Indo-Arabic decimal system is also an algorism, but operates in base 10 and (thankfully) has a radix point.

A form of exponentiation was discussed by Euclid (~300 BCE), and our modern understanding of it, including the superscript notation, came in 1637 thanks to René Descartes [7]. Similarly, logarithms were mentioned by Archimedes (~250 BCE) and our modern understanding came in 1614 thanks to John Napier [14].

The three-variable Ackermann function was presented by Wilhelm Ackermann [1] in 1928 as an example of a total computable function that is not primitive recursive. In 1935, Rózsa Péter [16] developed a two-argument variant of the Ackermann function, and it is her variant, often called the Ackermann-Péter function, that computer scientists—the authors included—commonly care about. In 1947, Reuben Goodstein [8] showed that a variant of the Ackermann function can be used to place the natural sequence of functions (addition, multiplication, exponentiation) in a systematic hierarchy of hyperoperations. This brought tetration, the fourth member of the sequence, into use. In the 1980s, computer scientists started using the iterated logarithm $\log^*$ in algorithmic analysis.

8.2 Inverse Ackermann in Computer Science

The inverse of the Ackermann function features in the time bound analyses of several algorithms. Tarjan [19] 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. Tarjan and van Leeuwen [20] later refined this to $O(m \cdot \alpha(n))$. Chazelle [4] showed that the minimum spanning tree of a connected graph with n vertices and m edges can be found in time $O(m \cdot \alpha(m, n))$.

Charguéraud and Pottier [3] verify the time complexity of union-find in Coq. To this end they invert another variant of the Ackermann function (*i.e.* not the Ackermann-Péter function we have studied). However, their inversion strategy relies on Hilbert's non-constructive ε operator to choose the necessary minimum (see Definition 2). This collapses Coq's distinction between `Prop` and `Type`, meaning that Charguéraud and Pottier's inverse cannot be extracted

to executable code. Moreover, Coq tactics such as `compute` cannot reduce applications of their inverse to concrete values. In contrast, with our technique, proving a goal such as `inv_ack_linear 100 = 4` is as simple as using `reflexivity`, and the computation is essentially instantaneous.

9 Conclusion

We have implemented a hierarchy of functions that calculate the upper inverses to the hyperoperation/Ackermann hierarchy and used these inverses to compute the inverse of the diagonal Ackermann function $\mathcal{A}(n)$. Our functions run in $\Theta(b)$ time on unary-represented inputs. On binary-represented inputs, our base-2 hyperoperations and inverse Ackermann run in $\Theta(b)$ time as well, where b is bitlength; our general binary hyperoperations run in $O(b^2)$. Our functions are structurally recursive, and thus easily satisfy Coq's termination checker.

Every pearl starts with a grain of sand. We had the benefit of two: Nivasch [15] and Seidel [18]. They proposed a definition of the inverse Ackermann essentially in terms of the inverse hyperoperations. However, their definition is approximate rather than exact (*i.e.* it has the same asymptotic rate of growth as the canonical definition but is off by a bounded-but-varying constant factor) and they provide only a sketch of an algorithm. Our technique is exact; our algorithm concrete, structurally recursive and asymptotically optimal; and our theory is verified in Coq.

A Proof of Correctness of Countdown Worker

This appendix provides the fully detailed proof of Theorem 17, which states that the function in Definition 16 correctly computes the countdown value as defined in Definition 12.

Theorem 17. $\forall n.\ f_a^C(n) = \min\left\{i : f^{(i)}(n) \le a\right\}$.

We begin with a lemma demonstrating the internal working of *countdown worker* at the i^{th} recursive step, including the accumulated result $1+i$, the current input $f^{(1+i)}(n)$, and the current budget $b-i-1$.

Lemma 45. *$\forall a, n, b, i \in \mathbb{N}$. $\forall f \in$ CONT. such that $i < b$ and $a < f^{(i)}(n)$:*

$$f_a^{C\mathcal{W}}(n,b) = 1 + i + f_a^{C\mathcal{W}}\left(f^{(1+i)}(n), b-i-1\right) \tag{12}$$

Proof. Fix a. We proceed by induction on i. Define $P(i)$ as

$$P(i) \triangleq \forall b, \forall n. b \ge i+1 \Rightarrow f^{(i)}(n) > a \Rightarrow f_a^{C\mathcal{W}}(n,b) = 1 + i + f_a^{C\mathcal{W}}\left(f^{(1+i)}(n), b-i-1\right)$$

- *Base case.* For $i = 0$, our goal $P(0)$ is: $f_a^{C\mathcal{W}}(n,b) = 1 + f_a^{C\mathcal{W}}(f(n), b-1)$ where $b \ge 1$ and $f(n) \ge a+1$. This is straightforward.
- *Inductive step.* Assume $P(i)$ has been proved, $P(i+1)$ is

$$f_a^{C\mathcal{W}}(n,b) = 2 + i + f_a^{C\mathcal{W}}\left(f^{(2+i)}(n), b-i-2\right)$$

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

$$f_a^{C\mathcal{W}}\left(f^{(1+i)}(n), b-i-1\right) = 1 + f_a^{C\mathcal{W}}\left(f^{(2+i)}(n), b-i-2\right)$$

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

□

Now we are ready to prove the correctness of the *countdown* computation.

Proof of Theorem 17. Since $f \in \text{CONT}_a$ and $\mathbb{N}$ is well-ordered, let $m = \min\left\{i : f^{(i)}(n) \le a\right\}$.[5] Our goal becomes $f_a^C(n) = m$. Note that this setup gives us:

$$\left(f^{(m)}(n) \le a\right) \wedge \left(\forall k.\ f^{(k)}(n) \le a \Rightarrow m \le k\right) \tag{13}$$

If $m = 0$, then $n = f^{(0)}(n) \le a$. So $f_a^C(n) = f_a^{C\mathcal{W}}(n,n) = 0 = m$ by Definition 15. When $m > 0$, our plan is to apply Lemma 45 to get

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

and then use Definition 15 over (13)'s first conjunct to conclude $f_a^C(n) = m$. It suffices to prove the premises of Lemma 45: $a < f^{(m-1)}(n)$ and $m-1 < n$.

The former follows by contradiction: if $f^{(m-1)}(n) \le a$, Equation 13's second conjunct implies $m \le m-1$, which is impossible for $m > 0$. The latter then easily follows by $f \in \text{CONT}_a$, since $n \ge 1 + f(n) \ge 2 + f(f(n)) \ge \cdots \ge m + f^{(m)}(n)$. Therefore, $f_a^C(n) = m$ in all cases, which completes the proof. □

B Proof of Correctness of Inverse Ackermann Worker

This appendix provides the fully detailed proof of Theorem 28, which states that given the appropriate arguments, the function *inverse Ackermann worker* in Definition 27 correctly computes the value of the inverse Ackermann function as defined in Theorem 26.

Theorem 28. $\alpha^{\mathcal{W}}(\alpha_0, \alpha_0(n), 0, n) = \alpha(n)$.

First we state and prove the following lemma, which illustrates the working of *inverse Ackermann worker* by examining each recursive call made during the execution of $\alpha^{\mathcal{W}}(\alpha_0, \alpha_0(n), 0, b)$.

[5] We prove the existence of the min in Coq's intuitionistic logic here in our codebase.

Lemma 46. *$\forall n, b, k$ such that $k \leq \min\{b, \alpha_{k-1}(n)\}$,*

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

Proof. Fix n and b. We prove $P(k)$ by induction, where

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

- *Base case.* This case, where $k = 0$, is trivial since both sides of the desired equality are identical.
- *Inductive step.* Assume $P(k)$. We need to prove $P(k+1)$. Assume $P(k+1)$'s premise, *i.e.* $(\alpha_k(n) \geq k+1) \wedge (b \geq k+1)$. Then $\alpha_k(n) > k \Rightarrow n > A(k,k) \Rightarrow n > A(k-1,k-1) \Rightarrow \alpha_{k-1}(n) > k-1 \Rightarrow \alpha_{k-1}(n) \geq k$ and $b \geq k+1 \Rightarrow b \geq k$, so $P(k)$'s premise applies. Therefore we have $P(k)$'s conclusion. It thus suffices to show

$$\alpha^{\mathcal{W}}(\alpha_{k+1}, \alpha_{k+1}(n), k+1, b-k-1) = \alpha^{\mathcal{W}}(\alpha_k, \alpha_k(n), k, b-k)$$

  Definition 24 gives $\alpha_{k+1} = (\alpha_k{}_1^C) \circ \alpha_k$, so this is (8) when $b-k \geq 1$ and $\alpha_k(n) \geq k+1$. Thus $P(k+1)$ holds.

□

With this lemma, we can proceed to establish the correctness of *inverse Ackermann worker*, *i.e.* Theorem 28. We reproduce the statement here as a convenient point of reference for readers.

Proof of Theorem 28. 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 $m = 0$, $m-1$ is also 0, so $m = 0 \leq \alpha_0(n) \leq \alpha_{m-1}(n)$. If $m \geq 1$, so $m-1 < m$, so $m-1 < \alpha_{m-1}(n) \Rightarrow m \leq \alpha_{m-1}(n)$. In both cases, Lemma 46 implies:

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

where the last equality follows from (8) when $\alpha_m(n) \leq m$.

□

Acknowledgments

We thank Olivier Danvy for a helpful early-stage discussion about the difficulty of defining `div` in Coq. This work was funded in part by Yale-NUS College grant R-607-265-322-121 and the sponsors of the Crystal Center at National University of Singapore. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Yale-NUS College or the Crystal Centre.

References

[1] Wilhelm Ackermann. 1928. Zum Hilbertschen Aufbau der reellen Zahlen. *Math. Ann.* 99, 1 (01 Dec 1928), 118–133.
[2] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions.* Springer.
[3] Arthur Charguéraud and François Pottier. 2019. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *J. Autom. Reasoning* 62, 3 (2019), 331–365.
[4] Bernard Chazelle. 2000. A minimum spanning tree algorithm with Inverse-Ackermann type complexity. *J. ACM* 47, 6 (2000), 1028–1047.
[5] Adam Chlipala. 2013. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant.* MIT Press.
[6] Coq development team. 2019. The Coq proof assistant. https://coq.inria.fr/
[7] René Descartes. 1637. Discourse de la Méthode.
[8] R. L. Goodstein. 1947. Transfinite Ordinals in Recursive Number Theory. *J. Symb. Log.* 12, 4 (1947), 123–129.
[9] Robert A. Heinlein. 1966. *The Moon Is a Harsh Mistress.* G. P. Putnam's Sons.
[10] Johannes Hölzl and Fabian Immler. 2012. Theory Log_Nat. https://www.isa-afp.org/browser_info/current/AFP/IEEE_Floating_Point/Log_Nat.html
[11] Inv-Ack. 2019. inv-ack/inv-ack. https://github.com/inv-ack/inv-ack
[12] D. E. Knuth. 1976. Mathematics and Computer Science: Coping with Finiteness. *Science* 194, 4271 (1976), 1235–1242.
[13] Assia Mahboubi and Enrico Tassi. 2018. Mathematical Components. https://math-comp.github.io/mcb/
[14] John Napier. 1614. *Mirifici Logarithmorum Canonis Descriptio.*
[15] Gabriel Nivasch. 2009. Inverse Ackermann without pain. http://www.gabrielnivasch.org/fun/inverse-ackermann
[16] Rózsa Péter. 1935. Konstruktion nichtrekursiver Funktionen. *Math. Ann.* 111, 1 (01 Dec 1935), 42–60.
[17] Benjamin C Pierce. 2019. An Evaluation Function for Imp. https://softwarefoundations.cis.upenn.edu/lf-current/ImpCEvalFun.html
[18] Raimund Seidel. 2006. Understanding the Inverse Ackermann Function. http://cgi.di.uoa.gr/~ewcg06/invited/Seidel.pdf
[19] Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* 22, 2 (1975), 215–225.
[20] Robert Endre Tarjan and Jan van Leeuwen. 1984. Worst-case Analysis of Set Union Algorithms. *J. ACM* 31, 2 (1984), 245–281.
[21] Isabelle Development Team. 2013. The Isabelle2013 Library. https://isabelle.in.tum.de/website-Isabelle2013/dist/library/index.html
[22] Linh Tran, Anshuman Mohan, and Aquinas Hobor. 2019. A Functional Proof Pearl: Inverting the Ackermann Hierarchy (Extended Version). (2019). https://www.comp.nus.edu.sg/~hobor/Publications/2020/inverse_ack_extended_cpp2020.pdf