A Functional Proof Pearl: Inverting the Ackermann Hierarchy

Anonymous Author(s)

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

Keywords: Ackermann · hyperoperations · Coq

1 Overview

The inverse to the explosively-growing Ackermann function features in several key algorithmic asymptotic bounds, such as the union-find data structure [14] and the computation of a minimum spanning tree of a graph [4]. 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 *The Ackermann-Péter function [11] (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 one-variable *diagonal* Ackermann function $\mathcal{A} : \mathbb{N} \to \mathbb{N}$ is defined as $\mathcal{A}(n) \triangleq A(n,n)$. The diagonal Ackermann function 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 [4,14].

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

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

n	function	$a[n]b$	$\mathcal{A}_n(b)$	upper inverse	$a\langle n\rangle b$	$\alpha_n(b)$
0	successor	$1+b$	$1+b$	predecessor	$b-1$	$b-1$
1	addition	$a+b$	$2+b$	subtraction	$b-a$	$b-2$
2	multiplication	$a\cdot b$	$2b+3$	division	$\lceil \frac{b}{a} \rceil$	$\lceil \frac{b-3}{2} \rceil$
3	exponentiation	a^b	$2^{b+3}-3$	logarithm	$\lceil \log_a\ b \rceil$	$\lceil \log_2\ (b+3) \rceil - 3$
4	tetration	$\underbrace{a^{\cdot^{\cdot^{a}}}}_{b}$	$\underbrace{2^{\cdot^{\cdot^{2}}}}_{b+3}-3$	iterated logarithm	$\log_a^*\ b$	$\log_2^*\ (b+3)-3$

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

1.1 The hyperoperation/Ackermann hierarchy

The Ackermann function is relatively easy to define, but hard to understand. We see it 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[1] [6], written $a[n]b$. Table 1 gives the first five hyperoperations (indexed by n and named) and the related kludgy $\mathcal{A}_n$ Ackermann functions. We also name and give their inverses, which we write as $a\langle n\rangle b$ and $\alpha_n(b)$. The kludge has three parts. First, the Ackermann hierarchy is related to the hyperoperation hierarchy with a set 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].

It is worth studying and inverting the hyperoperations before handling the Ackermann inverses. Once we have defined these inverses, we shall define an efficient inverse α to the diagonal Ackermann $\mathcal{A}$ thanks to our Theorem 25, which characterizes α without referring to $\mathcal{A}$ directly: $\forall n.\ \alpha(n) = \min\left\{k : \alpha_k(n) \le k\right\}$.

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 [16] is easy, 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.
```

Coq accepts the definition of `mult`; indeed this is how 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 Coq's termination

[1] Knuth arrows [8], 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] $\mathcal{A}_1$ and $\mathcal{A}_2$ do not break this pattern: $2+(b+3)-3 = 2+b$, and $2\cdot(b+3)-3 = 2b+3$.

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 $\texttt{a} > 0$ and $\texttt{b} = 0$.

Of course, division *can* be defined, but an elegant definition is a little subtle. We certainly need to do more than just check that $\texttt{b} > 0$. Two standard techniques are to define a custom termination measure [5]; and to define a straightforward function augmented with an extra `nat` parameter that denotes a "gas" value that decreases at each recursive call [12]. 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. For instance, 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 are not aware of any explanation of the technique used, and we also find it hard to extend that technique to other functions in 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: practical utility follows from theoretical grace. Finally, our techniques are compact: consider the 14 lines of code that invert the diagonal Ackermann function in `nat` on the 18th page of this paper (§A). Our Coq development 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 then sketch a method to find the inverse Ackermann function.
§5 We give a computation for the inverse Ackermann function in $\mathrm{O}(n)$ time.
§6 We extend to $\mathrm{O}(b)$, where b is the bitlength of the binary encoding of n.
§7 We discuss the value of linear time inverses for explosively growing functions, and extend to the two-argument inverse Ackermann function.

All our techniques are mechanized in Coq and our code is available online at github.com/inv-ack/inv-ack [7]. 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.

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

2 Hyperoperations and Ackermann via Repeater

Let us formally define hyperoperations and clarify the intuition given by Table 1 by relating them 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 an *initial value*, which should be the additive identity 0. In the latter case, we multiply a repeatedly to an initial value, which which should now be the multiplicative identity 1. The formal definition of hyperoperation is:

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

The seemingly complicated recursive rule 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; $f^{(0)}(u)=u$ (*i.e.* applying 0 times yields the identity).

This insight helps us encode hyperoperations (2) and Ackermann (1) in Coq. Notice that the recursive case of hyperoperations — and indeed, the third case of Ackermann — 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 Equation 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 $(n+1)^{\text{th}}$ and n^{th} levels in a *functional* way via a higher-order function that transforms the latter level to the former. For this we employ 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 $f_a^{\mathcal{R}}$, is a function $\mathbb{N} \to \mathbb{N}$ such that $f_a^{\mathcal{R}}(n) = f^{(n)}(a)$.*

```
Fixpoint repeater_from (f : nat -> nat) (a n : nat) : nat :=
  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 $\lambda b.a[n-1]b$.

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

```
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 intuituve 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, written F^{-1}, is $\lambda n. \min\{m : F(m) \geq n\}$.*

This is well-defined as long as F is unbounded, *i.e.* $\forall b.\ \exists a.\ b \leq F(a)$. However, it only makes sense 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 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. The sentence $\forall m.\ f(n) \leq m \Leftrightarrow n \leq F(m)$ implies: (1) $f(n)$ is a lower bound to $\{m : n \leq F(m)\}$ and (2) $f(n)$ is itself in the set, since plugging in $m \triangleq f(n)$ 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 $\forall m.\ n \leq F(m) \Rightarrow f(n) \leq m$. Now, $\forall m \geq f(n).\ F(m) \geq F(f(n)) \geq n$ by increasing-ness, thus completing the proof.

Corollary 6 *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 $(F^{-1} \circ F)(n) \leq n$. By ($\Rightarrow$) of the same theorem, $(F^{-1} \circ F)(n) \leq (F^{-1} \circ F)(n)$ implies $F(n) \leq F\big((F^{-1} \circ F)(n)\big)$. F is strictly increasing, so $n \leq (F^{-1} \circ F)(n)$.

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, $F_a^{\mathcal{R}}$ is not necessarily strictly increasing: *e.g.* the identity function $\texttt{id}$ is strictly increasing, but $\texttt{id}_a^{\mathcal{R}}(n) = (\texttt{id} \circ \ldots \circ \texttt{id})(a) = a$ is a constant function. We need a little more.

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: $\forall n.\ F_a^{\mathcal{R}}(n) = F^{(n)}(a) \geq a + n \geq 1 + n$. That is, $F_a^{\mathcal{R}}$ is itself an expansion strict from 0.

Definition 9 *$\forall a \geq 1$, if F is a strictly increasing function that is also a strict expansion from a, then $F_a^{\mathcal{R}}$ enjoys an important property: $F_a^{\mathcal{R}}$ "preserves" the properties of F, albeit with 0 substituted for a. We thus say that such a function F is* repeatable from a*, and denote the set of functions repeatable from a as $\textsc{rept}_a$. It is straightforward 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 $F \in \text{REPT}_a$ 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 $f_a^{\mathcal{C}}$. Now 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 $m = f_a^{\mathcal{C}}(n)$ 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 as we reach/pass a. This only works if each chainlink is strictly less than the previous, *i.e.* f is a *contraction*.

Definition 10 *A function $f : \mathbb{N} \to \mathbb{N}$ is a* contraction *if $\forall n.\ f(n) \leq n$. Given an $a \geq 1$, a contraction f is* strict above a *if $\forall n > a.\ n > f(n)$. 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, $\forall s \leq t.\ \text{CONT}_s \subseteq \text{CONT}_t$.*

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

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 $\forall n, m.\ f_a^{\mathcal{C}}(n) \leq m \Leftrightarrow f^{(m)}(n) \leq a$.*

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. When $n \leq a$, use Corollary 13 as follows: $n = f^{(0)}(n) \leq a \Leftrightarrow f\,{}^{\mathcal{C}}_{a}(n) \leq 0$. When $n > a$, proceed by antisymmetry. Define $m \triangleq f\,{}^{\mathcal{C}}_{a}(f(n))$, and note that Corollary 13 gives $f\,{}^{\mathcal{C}}_{a}(n) \leq 1 + m \Leftrightarrow f^{(1+m)}(n) \leq a$. 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 $f\,{}^{\mathcal{C}}_{a}(n) \geq 1$ by the above. Define $p \triangleq f\,{}^{\mathcal{C}}_{a}(n) - 1$. It remains to show $f\,{}^{\mathcal{C}}_{a}(f(n)) \leq p \Rightarrow f^{(p)}(f(n)) \leq a \Rightarrow f^{(p+1)}(n) \leq a$, which holds by unfolding p's definition.

3.3 A structurally recursive computation of countdown

The higher-order repeater function is well-defined for all input functions, even those not in $\textsc{rept}_a$ (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, which 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 *For any $a \in \mathbb{N}$ and $f : \mathbb{N} \to \mathbb{N}$, the* countdown worker *to a of f is a function $f\,{}^{\mathcal{CW}}_{a} : \mathbb{N}^2 \to \mathbb{N}$ such that:*

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

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 admittedly inelegant, but the point is that it can clearly be written as a Coq `Fixpoint`:

```
Fixpoint cdn_wkr (a : nat) (f : nat -> nat) (n b : nat) : nat :=
  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 a method to calculate it in constant time. 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 $f\,{}^{\mathcal{C}}_{a}(n) \triangleq f\,{}^{\mathcal{CW}}_{a}(n, n)$.*

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

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

Theorem 17 *$\forall a \in \mathbb{N}.\ \forall f \in \text{CONT}_a$, we have $\forall n.\ f_a^{\mathcal{C}}(n) = \min\left\{i : f^{(i)}(n) \leq a\right\}$.*

Proof (outline). Assume the initial arguments (n, b), and let the original call $f_a^{\mathcal{CW}}(n, b)$ 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 to this rule will be the last call, which will return 0 and add nothing to the result.

Suppose $b = n$, and let $m \triangleq \min\left\{i : f^{(i)}(n) \leq a\right\}$[4]. Then $m \leq n$ since $f^{(n)}(n) \leq a$. 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 $f^{(m)}(n) \leq a$. This last call adds 0 to the accumulated result, which is m at the moment. Therefore $f_a^{\mathcal{CW}}(n, n) = m$.

We give an extended version of this proof in Appendix C.1, 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 *$\forall F \in \text{REPT}_a$. $f \triangleq F^{-1}$ satisfies $f \in \text{CONT}_a$ and $f_a^{\mathcal{C}} = \left(F_a^{\mathcal{R}}\right)^{-1}$. Furthermore, if $a \geq 1$, then $F_a^{\mathcal{R}} \in \text{REPT}_0$ and $f_a^{\mathcal{C}} \in \text{CONT}_0$.*

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

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 The inverse hyperoperations, including `div`, `log`, and `log*`

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

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

```
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.
```

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

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 $a\langle 0\rangle \in \textsc{cont}_0$. Then:

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

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

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

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

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 $a_0 = a, a_1 = 0, a_n = 1$. Then define P and Q as:

$$P(n) \triangleq \Big(a[n] \in \textsc{rept}_{a_n}\Big) \quad \text{and} \quad Q(n) \triangleq \Big((a\langle n\rangle = (a[n])^{-1}\Big).$$

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

$$P(n) \Rightarrow Q(n) \Rightarrow Q(n+1) \qquad (5) \qquad a_n \ge 1 \Rightarrow P(n) \Rightarrow P(n+1) \qquad (6)$$

Goal 1: $P(0) \Leftrightarrow \lambda b.(b+1) \in \textsc{rept}_a$ 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:

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

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.*

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 $\alpha_{i+1} \triangleq \alpha_i{}^{\mathcal{C}}_{\mathcal{A}_i(1)}$ is tempting. But this approach is flawed because it still depends on $\mathcal{A}_i$. Instead, we reexamine the inverse relationship: suppose $\alpha_i = \big(\mathcal{A}_i\big)^{-1}$ and $\alpha_{i+1} = \big(\mathcal{A}_{i+1}\big)^{-1}$. Then $\mathcal{A}_{i+1}(m) = \big(\mathcal{A}_i\big)^{(m)}\big(\mathcal{A}_i(1)\big)$. We then have:

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

Equivalently, $\alpha_{i+1}(n) = \min\{m : (\alpha_i)^{(m+1)}(n) \leq 1\}$, or $\alpha_{i+1}(n) = \alpha_i{}_1^{\mathcal{C}}(\alpha_i(n))$. From Equation 7 we can thus define the inverse Ackermann hierarchy:

Definition 24 $\alpha_i \triangleq \begin{cases} \lambda n.(n-1) & \text{if } i = 0 \\ (\alpha_{i-1}{}_1^{\mathcal{C}}) \circ \alpha_{i-1} & \text{if } i \geq 1 \end{cases}$

Abracadabra! We can express α using the hierarchy *without referring to* $\mathcal{A}$ *itself*:

Theorem 25 $\forall n.\ \alpha(n) = \min\{k : \alpha_k(n) \leq k\}$.

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

Definition 26 *The inverse Ackermann worker is a function* $\alpha^{\mathcal{W}}$*:*

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

The following theorem shows that this function computes the inverse Ackermann function correctly when passed appropriate arguments.

Theorem 27 $\forall n.\ \alpha^{\mathcal{W}}(\alpha_0, \alpha_0(n), 0, n) = \alpha(n)$.

Proof (outline). If 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 arguments $(\alpha_{i+1}, \alpha_{i+1}(n), i+1, b-(i+1))$ at the next recursive call. A straightforward induction on k then shows that if $k \leq \min\{b, \alpha_k(n)\}$,

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

Let $m \triangleq \min\{k : \alpha_k(n) \leq k\}$. Then $m \leq n$ since $\alpha_n(n) \leq n$. (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 more involved mathematical proof of correctness in Appendix C.2, and a mechanized proof here. We thus have a (re-)definition of inverse Ackermann that is definable via a Coq-accepted worker, *i.e.* $\alpha(n) \triangleq \alpha^{\mathcal{W}}(\alpha_0, \alpha_0(n), 0, n)$. Below, we present the inverse Ackermann function in Gallina.

```
Fixpoint inv_ack_wkr (f : nat -> nat) (n k b : nat) : nat :=
  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 : nat) : nat :=
  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.
```

5 Time Bound of Our Inverses in Unary Encoding

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 give intuition and remain accessible to readers, we present a detailed sketch via lemma statements. We put full proofs of these lemmas in Appendix E. We elide the parallel analysis for the inverse hyperoperations, which have simpler initial values.

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 28 *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 following lemma establishes the general runtime structure of the *countdown* function when its input is encoded in `nat`.

Lemma 29 $\forall a \geq 1,\ \forall n \in \texttt{nat},\ \forall f \in \text{CONT}_a$,

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

5.1 A naïve α_i on `nat` is $\Omega(n^2)$, and is $O(n)$ after optimization

We start with the following lemma about the running time of each α_i, which is a consequence of Lemma 29. Its full proof can be found in Appendix E.

Lemma 30 *When α_i is defined per Definition 24,*

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

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

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 $\Omega(n^2)$, thus making $\mathcal{T}_{\alpha_i}(n) = \Omega(n^2)$. In fact, $i = 2$ suffices.

Lemma 31 $\forall i \geq 2.\ \mathcal{T}_{\alpha_i}(n) = \Omega(n^2)$.

Remark 32 *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 $O(n)$ as follows:*

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

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

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

Theorem 33 *When α_i is defined per Definition 24 with the added hardcoding of α_1 to $\lambda n.(n-2)$, $\forall i.\ \mathcal{T}_{\alpha_i}(n) \leq 4n + \left(19 \cdot 2^{i-3} - 2i - 13\right) \log_2 n + 2i = O(n)$.*

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, and also put a minimal standalone Coq definition for it in Appendix A.

Definition 34 *Define* $\tilde{\alpha}(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ \alpha^{\mathcal{W}}\left(\alpha_1, \alpha_1(n), 1, n-1\right) & \text{if } n \geq 2 \end{cases}$

Theorem 35 *$\forall n.\ \tilde{\alpha}(n) = \alpha(n)$ and (benchmark) $\mathcal{T}_{\tilde{\alpha}}(n) = O(n)$.*

Proof. Consider the correctness of $\tilde{\alpha}(n)$. 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 26. Theorem 27 then implies $\tilde{\alpha}(n) = \alpha(n)$.

Consider the time complexity of $\tilde{\alpha}(n)$. 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, calculate $\texttt{leb}\left(\alpha_k(n), k\right)$, taking no more than $k+1$ steps (as shown in Appendix E Lemma 45). Second, calculate $\alpha_{k+1}(n) = {\alpha_k}^{\mathcal{C}}_1(x)$ given $x \triangleq \alpha_k(n)$, taking time $\mathcal{T}_{\alpha_{k+1}}(n) - \mathcal{T}_{\alpha_k}(n)$ (as shown in Appendix E Lemma 46). The computation will terminate at $k = \alpha(n)$. Thus, $\forall n \geq 1$,

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

6 Performance Improvement With Binary Encoding

Thus far we have used the Coq type `nat`, which represents a number n using n bits. The binary system represents n in $\lfloor \log_2 n \rfloor + 1$ bits. In both systems, addition/subtraction of b-bit numbers is $\Theta(b)$, while multiplication is $\Theta\left(b^2\right)$. In general, arithmetic operations are often faster when the inputs are encoded in binary[5]. 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 F.

Remark 36 *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 $O(b)$.*

[5] We put a further explanation of Coq's binary type `N` in Appendix D.

6.1 Countdown and contractions with binary numbers

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 : N -> N) (a n : N) (b : nat) : 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 `nat` translation of `n`, *i.e.* `N.to_nat n`. This is untenable as it 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 37 $f \in \text{CONT}$ *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 $\forall n > a, \forall k.\ f(n) \leq \left\lfloor \frac{n-a}{2^k} \right\rfloor + a$. 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 : N -> N) (a n : N) : N :=
  bin_cdn_wkr f a n (nat_size (n - a)).
```

The following is the binary version of Lemma 29:

Lemma 38 *$\forall n \in$ `N`, if f is a binary strict contraction above a,*

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

6.2 Inverse Ackermann in $\mathbf{O}(\log_2 n)$ time, where n is magnitude

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 $\texttt{N.log2}(x+2) - 2 = \lfloor \log_2(x+2) \rfloor - 2 = \lceil \log_2(x+3) \rceil - 3$, so the above Coq code is correct.

Theorem 39 $\forall i,\ \forall n,\ \mathcal{T}_{\alpha_i}(n) \leq 2\log_2 n + \left(3 \cdot 2^i - 3i - 13\right) \log_2 \log_2 n + 3i.$

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 -> N) (n k : N) (b : nat) : N :=
  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 : N) : 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, and give a standalone Coq definition in Appendix B. As in Theorem 35, the time complexity is the sum of each component's runtime:

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

With reference to Lemmas 46 (Appendix E), 51 and 52 (Appendix F), 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 46. By Lemma 51, each $\mathcal{T}_{\texttt{N.leb}}$ 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 52. The remaining items total $\Theta(\log_2 n)$. Thus, $\forall n \geq 8$:

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

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.

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.

7.2 The two-parameter inverse Ackermann function

Some authors [4,14] prefer a two-parameter inverse Ackermann function.

Definition 40 *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 $\hat{\alpha}(n,n)$ and the single-parameter $\alpha(n)$ are neither equal nor directly related, but it is straightforward to modify our techniques to compute $\hat{\alpha}(m,n)$.

Definition 41 *The two-parameter inverse Ackermann worker is a function* $\hat{\alpha}^{\mathcal{W}}$*:*

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

Theorem 42 $\hat{\alpha}(m,n) = 1 + \hat{\alpha}^{\mathcal{W}}\left(\alpha_1, \alpha_1\left(\lceil \log_2 n \rceil\right), \left\lfloor \frac{m}{n} \right\rfloor, \lceil \log_2 n \rceil\right)$.

We mechanize the above for both unary and binary inputs in our codebase.

8 Related Work and Conclusion

The Coq standard library has linear-time definitions of division and base-2 discrete logarithm on `nat` and `N`. The Mathematical Components library [9] has a discrete logarithm on `nat`. 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 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 it 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, like the modern Hindu-Arabic decimal system, enabled *algorisms*: the place value of a glyph determined how many times it 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 $a \times 60^2 + b \times 60 + c$. 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. Including multiplication and division bought great concision: a number n was represented in $\lfloor \log_{60} n \rfloor + 1$ glyphs.

8.2 The Ackermann function and its inverse

The three-variable Ackermann function was presented by Wilhelm Ackermann as an example of a total computable function that is not primitive recursive [1]. It 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 [11], 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 [14] 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 [15] 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] verified the time complexity of union-find in Coq. They formalized a version of the Ackermann function, but not its inverse.

8.3 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 [10] and Seidel [13]. 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.

References

1. Ackermann, W.: Zum hilbertschen aufbau der reellen zahlen. Mathematische Annalen **99**(1), 118–133 (Dec 1928)
2. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Springer (2004)
3. Charguéraud, A., Pottier, F.: 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)
4. Chazelle, B.: A minimum spanning tree algorithm with Inverse-Ackermann type complexity. J. ACM **47**(6), 1028–1047 (2000)
5. Chlipala, A.: Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant. MIT Press (2013)
6. Goodstein, R.L.: Transfinite ordinals in recursive number theory. J. Symb. Log. **12**(4), 123–129 (1947)
7. Inv-Ack: inv-ack/inv-ack, https://github.com/inv-ack/inv-ack
8. Knuth, D.E.: Mathematics and computer science: Coping with finiteness. Science **194**(4271), 1235–1242 (1976)
9. Mahboubi, A., Tassi, E.: Mathematical components, https://math-comp.github.io/mcb/
10. Nivasch, G.: Inverse Ackermann without pain, http://www.gabrielnivasch.org/fun/inverse-ackermann
11. Péter, R.: Konstruktion nichtrekursiver funktionen. Mathematische Annalen **111**(1), 42–60 (Dec 1935)
12. Pierce, B.C.: An evaluation function for imp, https://softwarefoundations.cis.upenn.edu/lf-current/ImpCEvalFun.html
13. Seidel, R.: Understanding the inverse Ackermann function, http://cgi.di.uoa.gr/~ewcg06/invited/Seidel.pdf
14. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. J. ACM **22**(2), 215–225 (1975)
15. Tarjan, R.E., van Leeuwen, J.: Worst-case analysis of set union algorithms. J. ACM **31**(2), 245–281 (1984)
16. Coq development team: The Coq proof assistant, https://coq.inria.fr/

A Standalone Linear-Time Inverse Ackermann α in `nat`

```
Require Import Omega Program.Basics.

Fixpoint cdn_wkr (a : nat) (f : nat -> nat) (n b : nat) : nat :=
  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 : nat -> nat) (n k b : nat) : nat :=
  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.
```

B Standalone Linear-Time Inverse Ackermann α in N

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

Definition nat_size (n : N) : nat :=
  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 : N -> N) (a n : N) (b : nat) : 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 : N -> N) (a n : N) : N :=
  bin_cdn_wkr f a n (nat_size (n - a)).

Fixpoint bin_inv_ack_wkr (f : N -> N) (n k : N) (b : nat) : N :=
  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).
```

C Proofs of Correctness for Worker Functions

C.1 Countdown Worker

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

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 43 $\forall a, n, b, i \in \mathbb{N}.\ \forall f \in \text{CONT}.$ *such that* $i < b$ *and* $a < f^{(i)}(n)$*:*

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

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

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

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

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

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

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

  This is in fact $P(0)$ with (b,n) substituted for $\left(b-i-1, f^{(1+i)}(n)\right)$. Since $f^{(1+i)}(n) \geq a+1$ and $b-i-1 \geq 1$, 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 $m = \min\left\{i : f^{(i)}(n) \leq a\right\}$.[6] Our goal becomes $f_a^{\mathcal{C}}(n) = m$. Note that this setup gives us:

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

If $m = 0$, then $n = f^{(0)}(n) \leq a$. So $f_a^{\mathcal{C}}(n) = f_a^{\mathcal{CW}}(n,n) = 0 = m$ by Definition 15. When $m > 0$, our plan is to apply Lemma 43 to get

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

and then use Definition 15 over (13)'s first conjunct to conclude $f_a^{\mathcal{C}}(n) = m$. It suffices to prove the premises of Lemma 43: $a < f^{(m-1)}(n)$ and $m-1 < n$.

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

C.2 Inverse Ackermann Worker

This subsection provides the fully detailed proof of Theorem 27, which states that given the appropriate arguments, the function *inverse Ackermann worker* in Definition 26 correctly computes the value of the inverse Ackermann function as defined in Theorem 25.

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

Theorem 27 $\alpha^{\mathcal{W}}\big(\alpha_0, \alpha_0(n), 0, n\big) = \alpha(n)$.

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 $\alpha^{\mathcal{W}}\big(\alpha_0, \alpha_0(n), 0, b\big)$.

Lemma 44 $\forall n, b, k$ *such that* $k \leq \min\big\{b, \alpha_{k-1}(n)\big\}$,

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

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

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

- *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

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

  Definition 24 gives $\alpha_{k+1} = \big(\alpha_k \begin{smallmatrix}\mathcal{C}\\ 1\end{smallmatrix}\big) \circ \alpha_k$, 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 27. We reproduce the statement here as a convenient point of reference for readers.

Proof (of Theorem 27). Since the sequence $\big\{\alpha_k(n)\big\}_{k=1}^{\infty}$ decreases while $\{k\}_{k=1}^{\infty}$ increases to infinity, there exists $m \triangleq \min\big\{k : \alpha_k(n) \leq k\big\} = \alpha(n)$. 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 $m \geq 1$, $m - 1 < m \Rightarrow m - 1 < \alpha_{m-1}(n) \Rightarrow m \leq \alpha_{m-1}(n)$. In both cases, Lemma 44 implies:

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

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

D Support for Binary Numbers in Coq

Coq has support for binary numbers with the type `N`, which consists of constructors `N0` and `Npos`. The latter, `Npos`, unfolds directly 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 provides a separate constructor `N0`.

E Proofs of Time Bound Lemmas on `nat`

This section provides fully detailed proofs for the time complexity bounds established in §5, *i.e.* Lemma 30 for the general running time formula of α_i, Lemma 31 for the lower bound on the naïve version, and Theorem 33 for the upper bound on the improved version. These results and their supporting lemmas are reproduced here for readers' easy reference.

We start by examining the running time of two operations used in the Coq definition of *countdown worker* on `nat`, namely `leb` and `compose`.

Lemma 45 *The standard Coq library function* $\texttt{leb}(x, y)$ *returns* `true` *if* $x \leq y$ *and* `false` *otherwise.* $\mathcal{T}_{\texttt{leb}}(x, y) = \min\{x, y\} + 1$ *when* n *and* a *are of type* `nat`.

Proof. Per its Coq definition, $\texttt{leb}(x, y)$ returns `true` if $x = 0$. When $x > 0$, $\texttt{leb}(x, y)$ returns `false` if $y = 0$, or $\texttt{leb}(x - 1, y - 1)$ if $y > 0$. Therefore,

$$\mathcal{T}_{\texttt{leb}}(x, y) = \begin{cases} 1 & \text{when } x = 0 \text{ or } y = 0 \\ \mathcal{T}_{\texttt{leb}}(x - 1, y - 1) + 1 & \text{when } x > 0, y > 0 \end{cases}$$

Let $P(x) \triangleq \forall y.\ \mathcal{T}_{\texttt{leb}}(x, y) = \min\{x, y\} + 1$. $P(0)$ holds since both sides are 1. Assume $P(x - 1)$ holds for $x > 0$, and consider any y

- If $y = 0$, $\mathcal{T}_{\texttt{leb}}(x, 0) = 1 = \min\{x - 1, 0\} + 1$.
- If $y > 0$, $\mathcal{T}_{\texttt{leb}}(x, y) = \mathcal{T}_{\texttt{leb}}(x - 1, y - 1) + 1 = \min\{x - 1, y - 1\} + 2 = \min\{(x - 1) + 1, (y - 1) + 1\} + 1 = \min\{x, y\} + 1$. The second equality, $\mathcal{T}_{\texttt{leb}}(x - 1, y - 1) + 1 = \min\{x - 1, y - 1\} + 2$, comes thanks to $P(x - 1)$.

In both cases, $P(x)$ holds. The proof is complete by induction on x.

Lemma 46 $\mathcal{T}_{f \circ g}(n) = \mathcal{T}_f(g(n)) + \mathcal{T}_g(n)$.

Proof. Per the Coq definition `compose`, computing $(f \circ g)(n)$, *i.e.* `(compose f g)` (n), requires first computing $m \triangleq g(n)$, and then $f(m)$.

Lemma 47 $\forall a \in \mathbb{N}, f \in \text{CONT}_a$, *where* $\texttt{succ}(x) \triangleq x + 1$,

$$\mathcal{T}_{f_a^{\mathcal{C}}}(n) = \sum_{i=0}^{f_a^{\mathcal{C}}(n)-1} \mathcal{T}_f\left(f^{(i)}(n)\right) + \sum_{i=0}^{f_a^{\mathcal{C}}(n)} \mathcal{T}_{\texttt{leb}}\left(f^{(i)}(n), a\right) + \sum_{i=0}^{f_a^{\mathcal{C}}(n)-1} \mathcal{T}_{\texttt{succ}}(i) \quad (14)$$

Proof. Since $f \in \text{CONT}_a$, $f\,{}_a^{\mathcal{C}}$ is the minimum k such that $f^{(k)}(n) \leq a$. The execution of $f\,{}_a^{\mathcal{CW}}(n,n)$ then takes $k+1$ recursive calls, where the i^{th} call for $0 \leq i \leq k$ takes the arguments i, a and $n_i \triangleq f^{(i)}(n)$ from the previous call (or the initial argument when $i = 0$), and performs the following computations:

1. Compute $\texttt{leb}\,(n_i, a)$ for $\mathcal{T}_{\texttt{leb}}\left(f^{(i)}(n), a\right)$ steps.
2. If $\texttt{leb}\,(n_i, a) = \texttt{true}$, return 0. Else proceed to the next step.
3. Compute $n_{i+1} \triangleq f(n_i) = f^{(i+1)}(n)$ for $\mathcal{T}_f\left(f^{(i)}(n)\right)$ steps.
4. Pass $n_{i+1}, i+1, a$ to the next recursive call and wait for it to return $k-i-1$.
5. Add 1 to the result for $\mathcal{T}_{\texttt{succ}}(k-i-1)$ steps and return $k-i$.

Summing up the time of each call gives the desired expression for $\mathcal{T}_{f\,{}_a^{\mathcal{C}}}(n)$.

Next, we restate and prove Lemma 29, which gives a formula for *countdown*'s running time.

Lemma 29 *$\forall a \geq 1, \forall n \in \texttt{nat}, \forall f \in \text{CONT}_a$ we have:*

$$\mathcal{T}_{f\,{}_a^{\mathcal{C}}}(n) = \sum_{i=0}^{f\,{}_a^{\mathcal{C}}(n)-1} \mathcal{T}_f\left(f^{(i)}(n)\right) + (a+2) f\,{}_a^{\mathcal{C}}(n) + f^{\left(f\,{}_a^{\mathcal{C}}(n)\right)}(n) + 1 \tag{15}$$

Proof. Per Definition 16 of countdown, $\mathcal{T}_{\texttt{leb}}\left(f^{(i)}(n), a\right) = a+1$ if $i < f\,{}_a^{\mathcal{C}}(n)$ and $f^{(i)}(n)+1$ otherwise (by Lemma 45). Thus, the second summand in (14) is equal to $(a+1) f\,{}_a^{\mathcal{C}}(n) + f^{\left(f\,{}_a^{\mathcal{C}}(n)\right)}(n) + 1$. Since $\mathcal{T}_{\texttt{succ}}(i) = 1$ on $\texttt{nat}$, the third summand is equal to $f\,{}_a^{\mathcal{C}}(n)$, completing the desired formula.

The above lemma is the last step for us to build a recursive formula for the running time of the inverse Ackermann hierarchy, with each level's runtime involving that of its previous one. We restate and prove Lemma 30 from §5.1.

Lemma 30 *When α_i is defined per Definition 24,*

$$\mathcal{T}_{\alpha_{i+1}}(n) = \sum_{k=0}^{\alpha_{i+1}(n)} \mathcal{T}_{\alpha_i}\left(\alpha_i^{(k)}(n)\right) + 3\alpha_{i+1}(n) + C_i(n) \tag{16}$$

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

Proof. From Definition 24, $\alpha_{i+1} = \left(\alpha_i\,{}_1^{\mathcal{C}}\right) \circ \alpha_i$. Since $\alpha_i \in \text{CONT}_1$, applying Lemma 29 gives us:

$$\begin{aligned}
\mathcal{T}_{\alpha_i\,{}_1^{\mathcal{C}}}\left(\alpha_i(n)\right) &= \sum_{k=0}^{\alpha_i\,{}_1^{\mathcal{C}}(\alpha_i(n))-1} \mathcal{T}_{\alpha_i}\left(\alpha_i^{(k)}\left(\alpha_i(n)\right)\right) + 3\alpha_i\,{}_1^{\mathcal{C}}\left(\alpha_i(n)\right) + \alpha_i^{\left(\alpha_i\,{}_1^{\mathcal{C}}(\alpha_i(n))\right)}\left(\alpha_i(n)\right) + 1 \\
&= \sum_{k=0}^{\alpha_{i+1}(n)-1} \mathcal{T}_{\alpha_i}\left(\alpha_i^{(k+1)}(n)\right) + 3\alpha_{i+1}(n) + \alpha_i^{\left(\alpha_{i+1}(n)\right)}\left(\alpha_i(n)\right) + 1 \\
&= \sum_{k=1}^{\alpha_{i+1}(n)} \mathcal{T}_{\alpha_i}\left(\alpha_i^{(k)}(n)\right) + 3\alpha_{i+1}(n) + \underbrace{\alpha_i^{\alpha_{i+1}(n)+1}(n) + 1}_{C_i(n)}
\end{aligned}$$

By Lemma 46, $\mathcal{T}_{\alpha_{i+1}}(n) = \mathcal{T}_{\alpha_i{}_1^{\mathcal{C}}(\alpha_i(n))} + \mathcal{T}_{\alpha_i}(n)$, which completes (16). By Definition 12 of countdown, $\alpha_{i+1}(n) = \alpha_i{}_1^{\mathcal{C}}\big(\alpha_i(n)\big) = \min\big\{l : \alpha_i^{(l+1)}(n) \leq 1\big\}$. Thus $\alpha_i^{\alpha_{i+1}(n)+1}(n) \in \{0,1\}$, or $C_i(n) \in \{1,2\}$. The proof is complete.

We are finally ready to prove the time complexity bounds of α_i. We start with proving the quadratic lower bound on the running time of the naïve version without any hardcoding, *i.e.* Lemma 31.

Lemma 31 *When α_i is defined per Definitions 24, $\forall i \geq 2$. $\mathcal{T}_{\alpha_i}(n) = \Omega\big(n^2\big)$.*

Proof. $\alpha_0 = \lambda m.(m-1)$ so $\alpha_1 = \big(\alpha_0\big)_1^{\mathcal{C}} \circ \alpha_0 = \lambda m.(m-2)$. By Lemma 30,

$$\mathcal{T}_{\alpha_1}(n) \geq \sum_{i=0}^{n-1} \mathcal{T}\big(\lambda m.(m-1), n-i\big) + 3(n-2) + 1 = 4n-5,$$

since $\mathcal{T}_{\alpha_0}(k) = 1$. Because $\alpha_2 = \big(\alpha_1\big)_1^{\mathcal{C}} \circ \alpha_1 = \lambda m.\left\lceil \frac{m-3}{2} \right\rceil$, again by Lemma 30,

$$\mathcal{T}_{\alpha_2}(n) \geq \sum_{i=0}^{\left\lceil \frac{n-3}{2} \right\rceil} \big(4(n-2i)-5\big) + 3\left\lceil \frac{n-3}{2} \right\rceil + 1 = \Omega\big(n^2\big)$$

By Lemma 30, $\forall i. \mathcal{T}_{\alpha_{i+1}}(n) \geq \mathcal{T}_{\alpha_i}(n)$, hence $\forall i \geq 2. \mathcal{T}_{\alpha_i}(n) = \Omega\big(n^2\big)$.

With the hardcoding $\alpha_1 \triangleq \lambda n.(n-2)$, we can improve the execution time of the whole hierarchy as noted in Remark 32. We now show that this small improvement makes the inverse Ackermann hierarchy run in linear time at any level, (*i.e.* we prove Theorem 33), via a few supporting lemmas.

Lemma 48 *If $n \geq 1$, $\mathcal{T}_{\alpha_3}(n) = 2\ \mathcal{T}_{\alpha_3}(n) \leq 4n+4$.*

Proof. It is easy to show that $\alpha_2^{(k)}(n) = \left\lfloor \frac{n+2}{2^k} \right\rfloor - 2$. Thus

$$\begin{aligned}
\mathcal{T}_{\alpha_3}(n) \leq\ & \sum_{k=0}^{\alpha_3(n)} \mathcal{T}_{\alpha_2}\left(\left\lfloor \frac{n+2}{2^k} \right\rfloor - 2\right) + 3\alpha_3(n) + 2 \\
\leq\ & 2\sum_{k=0}^{\alpha_3(n)} \left(\frac{n+2}{2^k} - 3\right) + 3\alpha_3(n) + 2 \\
\leq\ & 4(n+2) - 6(\alpha_3(n)+1) + 3\alpha_3(n) + 2 \leq 4n+4.
\end{aligned}$$

The following two technical lemmas that bound a sum of repeated applications of $\log_2$ and α_i are also needed in time bound proofs for the binary version of the inverse Ackermann hierarchy in Appendix F, and so have made as general as possible to allow reusability.

Lemma 49 $\forall n, \sum_{k=0}^{\log_2^*(n)-1} \log_2^{(k)} n \leq 2n$, *where* $\log_a^*(n) \triangleq \min\{k : \log_a^{(k)} n \leq 1\}$. *Note that the sum is* 0 *when* $n \leq 1$.

Proof. Define $S : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ where $S(x) \triangleq \sum_{k=0}^{\log_2^*(x)-1} \log_2^{(k)} x$. Clearly S is strictly increasing and $\forall x > 1, S(x) = n + S(\log_2 x)$. Using the fact $\log_2^*(x) \in \mathbb{N}$, we prove by induction on k the statement $P(k) \triangleq \forall x : \log_2^*(x) = k,\ S(x) \leq 2x$.

- *Base case.* $P(0)$, $P(1)$ hold trivially and $P(2) = \forall x : 2 < x \leq 4,\ x + \log_2 x \leq 2x$, which is equivalent to $\log_2 x \leq x$, which holds for all $x > 2$.
- *Inductive case.* Assume $P(k-1)$ where $k \geq 3$. Fix any x such that $\log_2^*(x) = k$, then $x > 4$. The function $\lambda x. \frac{x}{\log_2 x}$ is increasing on $[4, +\infty)$, so $\frac{4}{\log_2 4} \leq \frac{x}{\log_2 x}$ or $2 \log_2 x \leq x$. Since $\log_2^*(x) = k$, we have $\log_2^*(\log_2 x) = k - 1$. By $P(k-1)$, $S(x) = x + S(\log_2 x) \leq x + 2 \log_2 x \leq 2x$, which completes the proof.

Lemma 50 $\forall l,\ \forall i \geq 3,\ \forall s \leq \alpha_{i+1}(n).\ \displaystyle\sum_{k=s}^{\alpha_{i+1}(n)} \log_2^{(l)} \alpha_i^{(k)}(n) \leq 2 \log_2^{(l+s)} n.$

Proof. $\forall i \geq 3, \forall n, \alpha_i(n) \leq \log_2 n$ and $\alpha_{i+1}(n) \leq \log_2^*(n) - 1$, therefore

$$\sum_{k=s}^{\alpha_{i+1}(n)} \log_2^{(l)} \alpha_i^{(k)}(n) \ \leq \ \sum_{k=s}^{\log_2^*(n)-1} \log_2^{(l+k)}(n) \ \leq \ 2 \log_2^{(l+s)} n$$

With these lemmas in hand, we are ready to tackle Theorem 33.

Theorem 33 *When α_i is defined per Definition 24 with the added hardcoding of α_1 to $\lambda n.(n-2)$, $\forall i$. $\mathcal{T}_{\alpha_i}(n) \leq 4n + \big(19 \cdot 2^{i-3} - 2i - 13\big) \log_2 n + 2i = O(n)$.*

Proof. We have proved the result for $i = 0, 1, 2$. Let us proceed with induction on $i \geq 3$. The case $i = 3$ has been covered by Lemma 48. Let $M_i \triangleq 2^{i-3}19 - 2i - 13$ for each i and suppose the result holds for $i \geq 3$. We have

$$\begin{aligned}
\mathcal{T}_{\alpha_{i+1}}(n) &\leq \sum_{k=0}^{\alpha_{i+1}(n)} \mathcal{T}_{\alpha_i}\big(\alpha_i^{(k)}(n)\big) + 3\alpha_{i+1}(n) + 2 \\
&\leq \sum_{k=0}^{\alpha_{i+1}(n)} \big(4\alpha_i^{(k)}(n) + M_i \log_2 \alpha_i^{(k)}(n) + 2i\big) + 3\alpha_{i+1}(n) + 2 \\
&= 4n + 2(i+1) + (2i+3) \underbrace{\alpha_{i+1}(n)}_{\leq \log_2 n} + 4 \underbrace{\sum_{k=1}^{\alpha_{i+1}(n)} \alpha_i^{(k)}(n)}_{\leq 2 \log_2 n} + M_i \underbrace{\sum_{k=0}^{\alpha_{i+1}(n)} \log_2 \alpha_i^{(k)}(n)}_{\leq 2 \log_2 n} \\
&\leq 4n + 2(i+1) + (2M_i + 2i + 3 + 8) \log_2 n = 4n + M_{i+1} \log_2 n + 2(i+1)
\end{aligned}$$

F Proofs of Time Bound Lemmas on N

This section provides fully detailed proofs for the time complexity bounds established in §6, *i.e.* Lemma 38 for the upper bound on the time complexity of α_i on N (`bin_alpha`), and Theorem 39 for the upper bound on the time complexity

of the α on `N` (`bin_inv_ack`). These results and their supporting lemmas are reproduced here for readers' easy reference.

We start by inspecting the execution time of crucial computations in each recursive call of `bin_countdown`, namely `N.leb` and `N.succ`. While `leb` is easy, `succ` requires more care as it is worst-case logarithmic time but amortized constant time.

Lemma 51 *In binary encoding,* $\mathcal{T}_{\texttt{N.leb}}(x, y) \leq \lfloor \log_2 \min\{x, y\} \rfloor + 1$.

Proof. `N.leb`'s Coq definition indirectly involves the function `Pos.compare_cont`, which maps from `Pos*Pos` to the set $\{Gt, Lt, Eq\}$ (respectively stand for "Greater Than", "Less Than" and "Equal"). We henceforth use `Pc` as short form for `Pos.compare_cont`. Precisely we have:

$$\texttt{N.leb}(x, y) = \begin{cases} \texttt{true} & \text{when } x = 0 \\ \texttt{false} & \text{when } x > 0, y = 0 \\ \neg\big(\texttt{Pc}(Eq, x, y) == Gt\big) & \text{when } x > 0, y > 0 \end{cases}$$

where in the last case, x and y are treated as type `Pos`. The value of $\texttt{Pc}(x, y)$ is given in the following table, where $x{\sim}0$ and $x{\sim}1$ respectively representing appending 0 and 1 to the end of x's representation:

$x \backslash y$	1	$y'{\sim}0$	$y'{\sim}1$
1	R	Lt	Lt
$x'{\sim}0$	Gt	$\texttt{Pc}(R, x', y')$	$\texttt{Pc}(Lt, x', y')$
$x'{\sim}1$	Gt	$\texttt{Pc}(Gt, x', y')$	$\texttt{Pc}(R, x', y')$

It suffices to show $\mathcal{T}_{\texttt{Pc}}(x, y) \leq \lfloor \log_2 \min\{x, y\} \rfloor + 1$. The table above gives:

$$\mathcal{T}_{\texttt{Pc}}(x, y) = \begin{cases} 1 & \text{if } x = 1 \vee y = 1 \\ \mathcal{T}_{\texttt{Pc}}\left(\left\lfloor \frac{x}{2} \right\rfloor, \left\lfloor \frac{x}{2} \right\rfloor\right) + 1 & \text{if } x > 1 \wedge y > 1 \end{cases}$$

A simple binary induction on $\min\{x, y\}$ shows $\mathcal{T}_{\texttt{Pc}}(x, y) = \lfloor \log_2 \min\{x, y\} \rfloor + 1$. The proof is complete.

Lemma 52 $\forall n \geq 1.\ S(n) = \sum_{i=0}^{n-1} \mathcal{T}_{\texttt{N.succ}}(i) \leq 2n + \log_2 n$.

Proof. Per `N.succ`'s Coq definition, $\texttt{N.succ}(u) = 1$ if $u = 0$ and $\texttt{Pos.succ}(u)$ if $u > 0$, where when treating u as `Pos` we have $\texttt{Pos.succ}(u)$ returns 2 when $u = 1$, $u'{\sim}1$ when $u = u'{\sim}0$ and $(\texttt{Pos.succ}(u')){\sim}0$ when $u = u'{\sim}1$. A binary induction on u confirms that for all u,

$$\mathcal{T}_{\texttt{N.succ}}(u) = \begin{cases} \mathcal{T}_{\texttt{N.succ}}\left(\frac{u-1}{2}\right) + 1 & \text{when } u \text{ is odd,} \\ 1 & \text{when } u \text{ is even.} \end{cases}$$

We prove the lemma by induction on n. If $n = 1$, $S(n) = \mathcal{T}_{\texttt{N.succ}}(0) = 1$, our goal is $1 \leq 2$, which is trivial. For $n > 1$, observe that . Thus for all n, $S(2n+1) = S(2n) + 1$ and

$$\begin{aligned} S(2n) &= \sum_{i=0}^{n} \mathcal{T}_{\texttt{N.succ}}(2i) + \sum_{i=0}^{n-1} \mathcal{T}_{\texttt{N.succ}}(2i+1) = \sum_{i=0}^{n} 1 + \sum_{i=0}^{n-1} \big(\mathcal{T}_{\texttt{N.succ}}(i) + 1\big) \\ &= n + 1 + S(n) + n = S(n) + 2n + 1 \end{aligned}$$

From the above, it is easy to complete the proof via induction on n.

Now we are ready to prove Lemma 38, reproduced here for readers' reference.

Lemma 38 *$\forall n \in \texttt{N}$, if f is a binary strict contraction above a,*

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

Proof. On strict binary contractions, `bin_countdown` does exactly the same computations as the `countdown` on `nat`, with the added step of computing the budget of `nat_size (n-a)`, thus its execution time structure is almost the same. The only difference is the running time of each component in the sum.

$$\mathcal{T}_{f_a^{\mathcal{C}}}(n) = \begin{array}{l} \displaystyle\sum_{i=0}^{f_a^{\mathcal{C}}(n)-1} \mathcal{T}_f\left(f^{(i)}(n)\right) \quad + \sum_{i=0}^{f_a^{\mathcal{C}}(n)} \mathcal{T}_{\texttt{N.leb}}\left(f^{(i)}(n), a\right) \\ \quad + \displaystyle\sum_{i=0}^{f_a^{\mathcal{C}}(n)-1} \mathcal{T}_{\texttt{N.succ}}(i) \;+ \mathcal{T}_{\texttt{sub}}(n,a) \;+\; \mathcal{T}_{\texttt{nat\_size}}(n-a) \end{array}, \quad (17)$$

By Lemma 51, $\forall i, \mathcal{T}_{\texttt{N.leb}}\left(f^{(i)}(n), a\right) \leq \log_2 a + 1$, thus the second summand in (17) is no more than $(\log_2 a + 1)\big(f_a^{\mathcal{C}}(n) + 1\big)$. By Lemma 52, the third summand is no more than $2f_a^{\mathcal{C}}(n) + \log_2 f_a^{\mathcal{C}}(n)$. Lastly, the last two summands are each no more than $\log_2 n + 1$. Thus

$$\begin{aligned} \mathcal{T}_{f_a^{\mathcal{C}}(n)}(n) \;&\leq\; \left(\begin{array}{l} \displaystyle\sum_{i=0}^{f_a^{\mathcal{C}}(n)-1} \mathcal{T}_f\big(f^{(i)}(n)\big) \;+\; (\log_2 a + 1)\big(f^{(i)}(n) + 1\big) \\ \;+\; 2f_a^{\mathcal{C}}(n) \;+\; \log_2 f_a^{\mathcal{C}}(n) \;+\; 2\big(\log_2 n + 1\big) \end{array} \right) \\ &\leq \sum_{i=0}^{f_a^{\mathcal{C}}(n)-1} \mathcal{T}_f\big(f^{(i)}(n)\big) \;+\; (\log_2 a + 3)\,\big(f_a^{\mathcal{C}}(n) + 1\big) \;+\; 2\log_2 n \;+\; \log_2 f_a^{\mathcal{C}}(n) \end{aligned}$$

With all critical lemmas established and proven, including Lemma 50 in Appendix E, we proceed to prove the time complexity bound of each level in `bin_alpha` in the form of Theorem 39.

Theorem 39 *When α_i on* `N` *is defined via* `bin_alpha`*, for all i and n we have* $\mathcal{T}_{\alpha_i}(n) \leq 2\log_2 n + \left(3 \cdot 2^i - 3i - 13\right)\log_2\log_2 n + 3i$.

Proof. Firstly, we have:

- Clearly $\mathcal{T}_{\alpha_i}(n) \leq \log_2 n + 1$ for $i = 0, 1$.
- Computing $\alpha_2(n)$ consists of 2 steps: subtracting by 2 and dividing by 2 (shifting right by 1 bit), both of which takes no more than time $\log_2 n + 1$ time, thus $\mathcal{T}_{\alpha_2}(n) \leq 2\log_2 n$.
- Computing $\alpha_3(n)$ consists of 3 steps: adding 2, then take base 2 logarithm and subtract 2 from the result, with the first two taking no longer than time $\log_2 n + 1$ each and the last taking time at most $\log_2\log_2(n+2) + 1$ $\leq \log_2\log_2 n + 2$. Thus $\mathcal{T}_{\alpha_3}(n) \leq 2\log_2 n + \log_2\log_2 n + 3$.

Let $M_i = 3\cdot 2^i - 3i - 13$, then $\mathcal{T}_{\alpha_i}(n) \leq 2\log_2 n + M_i \log_2^{(2)} n + 3i$ for $i = 0, 1, 2, 3$. Suppose the same inequality holds for $i \geq 3$, by Lemmas 46 and 38, note that Lemma 46 works for all encoding systems, we have

$$\begin{aligned}
\mathcal{T}_{\alpha_{i+1}}(n) \leq & \sum_{k=0}^{\alpha_{i+1}(n)} \mathcal{T}_{\alpha_i}\left(\alpha_i^{(k)}(n)\right) + 3\underbrace{\alpha_{i+1}(n)}_{\leq \log_2\log_2 n} + 2\underbrace{\log_2 \alpha_i(n)}_{\leq \log_2\log_2 n} + \underbrace{\log_2 \alpha_{i+1}(n)}_{\leq \log_2\log_2 n} + 3 \\
\leq & 2\log_2 n + 2\underbrace{\sum_{k=1}^{\alpha_{i+1}(n)} \log_2 \alpha_i^{(k)}(n)}_{\leq 2\log_2\log_2 n} + M_i \underbrace{\sum_{k=0}^{\alpha_{i+1}(n)} \log_2^{(2)} \alpha_i^{(k)}(n)}_{\leq 2\log_2\log_2 n} + \\
& 3i\Big(\underbrace{\alpha_{i+1}(n)}_{\leq \log_2\log_2 n} + 1\Big) + 6\log_2^{(2)} n + 3 \\
\leq & 2\log_2 n + (2M_i + 3i + 10)\log_2^{(2)} n + 3(i+1) = 2\log_2 + M_{i+1}\log_2^{(2)} n + 3(i+1)
\end{aligned}$$

where $\forall i \geq 3, \alpha_i(n) \leq \log_2 n$ and $\alpha_{i+1}(n) \leq \log_2^{(2)} n$, while the $2\log_2^{(2)} n$ upper bounds for the sums come from Lemma 50. By induction on i, proof is complete.