Inverting the Ackermann Hierarchy

Abstract

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

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

1. Overview

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

S?? We explain our core techniques of *repeaters* and *countdowns* that allow us to define each level of the Ackermann hierarchy—and their upper inverses—in a straightforward and uniform manner. We show how countdowns, in particular, can be written structurally recursively.

S?? We show how to use our techniques to define the Ackermann and upper inverse Ackermann functions themselves.

S?? We detail a few optimizations that improve the running time of our individual hierarchy inverse functions from $O(n^2)$ to $O(n)$, and then further improve the running time of our inverse Ackermann function from $O\big(n \cdot \alpha(n)\big)$ to $O(n)$.

S?? We extend our functions in several useful ways: to the two-argument inverse Ackermann, lower inverses, and binary representations.

S?? We discuss related work and give some closing thoughts.

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

2. Some crappy theorems to put in place

Lemma 2.1. *For all* $a, b \in \mathbb{N}$, $f : \mathbb{N} \to \mathbb{N}$ *we have* $Wf{\downarrow_a}(n, b) = 0 \ \forall n \leq a$.

Proof. Trivial. □

Lemma 2.2. *For all* $a, n, b, i \in \mathbb{N}$ *and* $f \in$ Contr *such that* $i < b$ *and* $a < f^{(i)}(n)$*:*

$$Wf{\downarrow_a}(n, b) = 1 + i + Wf{\downarrow_a}\big(f^{1+i}(n), b - i - 1\big) \tag{2}$$

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

Proof. We proceed by induction on i. Let

$$P(i) \triangleq \mathrm{W}f{\downarrow_a}(n,b) = 1 + i + \mathrm{W}f{\downarrow_a}\big(f^{1+i}(n), b-i-1\big) \qquad \forall b, n : b \geq i+1, f^{(i)}(n) > a$$

1. *Base case.* For $i = 0$, our goal $P(0)$ is:

$$\mathrm{W}f{\downarrow_a}(n,b) = 1 + \mathrm{W}f{\downarrow_a}(f(n), b-1)$$

   where $b \geq 1, f(n) \geq a+1$, which is trivial.

2. *Inductive case.* Suppose $P(i)$ has been proven. Then

$$P(i+1) \triangleq \mathrm{W}f{\downarrow_a}(n,b) = 2 + i + \mathrm{W}f{\downarrow_a}\big(f^{2+i}(n), b-i-2\big)$$

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

$$\mathrm{W}f{\downarrow_a}\big(f^{1+i}(n), b-i-1\big) = 1 + \mathrm{W}f{\downarrow_a}\big(f^{2+i}(n), b-i-2\big)$$

   Note that this is in fact $P(0)$ with $(b,n) \leftarrow \big(b-i-1, f^{(1+i)}(n)\big)$. Since $f^{(1+i)}(n) \geq a+1$ and $b-i-1 \geq 1$, the above holds and proof is complete.

□

Theorem 2.3. *For all $a \in \mathbb{N}$ and $f \in \textsc{Contr}_{1+a}$, we have*

$$f{\downarrow_a}(n) = \min\big\{i : f^{(i)}(n) \leq a\big\} \tag{3}$$

Or equivalently,

$$f{\downarrow_a}(n) \leq k \iff f^{(k)}(n) \leq a \quad \forall n, k \in \mathbb{N} \tag{4}$$

Proof. First, to see why (3) and (4) are equivalent, we rewrite (3) in the following way:

$$\big(f^{(f{\downarrow_a}(n))}(n) \leq a\big) \wedge \big(f^{(l)}(n) \leq a \implies f{\downarrow_a}(n) \leq l \quad \forall l\big)$$

□

3. Increasing Functions and Their Inverses

3.1. Ackermann function and its inverse

the old intro material is here

The time complexity of the

has traditionally been hard to estimate, especially when it is implemented with the heuristic rules of *path compression* and *weighted union*. Tarjan [**?**] showed that for a sequence of m FINDs intermixed with $n-1$ UNIONs such that $m \geq n$, the time required $t(m,n)$ is bounded as: $k_1 m\alpha(m,n) \leq t(m,n) \leq k_2 m\alpha(m,n)$. Here k_1 and k_2 are positive constants and $\alpha(m,n)$ is the inverse of the Ackermann function.

The Ackermann function, commonly denoted $\mathrm{A}(m,n)$, was first defined by Ackermann [**?**], but this definition was not as widely used as the following variant, given by Peter and Robinson [**?**]:

$\mathrm{A}(m,n)$ increases extremely fast on both inputs, and, consequently, so does $\mathrm{A}(n,n)$. This implies $\alpha(n)$ increases extremely slowly, although it still tends to infinity. However, this does not mean that computing $\alpha(n)$ for each n is an easy task. In fact, the naive method would iteratively check $\mathrm{A}(k,k)$ for $k = 0, 1, \ldots,$ until $n \leq \mathrm{A}(k,k)$. The computation time would be staggering. For instance, suppose $n > 1$, and $\alpha(n) = k+1$. This is equivalent to

$$\mathrm{A}(k,k) < n \leq \mathrm{A}(k+1,k+1) \tag{5}$$

The naive algorithm would need to compute $\mathrm{A}(t,t)$ for $t = 0, 1, \ldots, k, k+1$ before terminating. Although one could argue that the total time to compute $\mathrm{A}(t,t)$ for $t \leq k$ is still $O(n)$, as they are all less than n, the time to compute $\mathrm{A}(k+1,k+1)$ could be astronomically larger than n. This situation motivates the need for an alternative, more efficient approach for computing the inverse Ackermann function.

3.2. The hierarchy of Ackermann functions

If one denotes $\mathrm{A}_m(n) = \mathrm{A}(m,n)$, one can think of the Ackermann function as a hierarchy of functions, each level A_m is a recursive function built with the previous level A_{m-1}:

Definition 3.1. The Ackermann hierarchy is a sequence of functions $\mathrm{A}_0, \mathrm{A}_1, \ldots$ defined as:

1. $A_0(n) = n+1 \quad \forall n \in \mathbb{N}$.
2. $A_m(0) = A_{m-1}(1) \quad \forall m \in \mathbb{N}_{>0}$.

3. $A_m(n) = A^{(n)}_{m-1}(0) \quad \forall n, m \in \mathbb{N}_{>0}$,

where $f^{(n)}(x)$ denotes the result of applying n times the function f to the input x. This hierarchy satisfies $\mathrm{A}_m(n) = \mathrm{A}(m, n) \ \ \forall m, n \in \mathbb{N}$.

This hierarchical perspective can be reversed, as shown in the next section, to form an inverse Ackermann hierarchy of functions, upon which we can compute the inverse Ackermann function as defined in Definition 1.1.

By extend, we mean to other members of the Ackermann/Knuth/hyper-operation hierarchy. By this we mean to the sequence of function families $f^a_0, f^a_1, \ldots, f^a_n, \ldots$, where $f^a_0(x) \triangleq x+1$ and, for each $i > 0$, $f^a_i(x)$ is the result of iterating f^a_{i-1} x times on a,

$$\begin{array}{lcll|l}
f^a_0(b) & \triangleq & 1 + \underbrace{1 + \ldots + 1}_{b} & = & b+1 & b-1 \\
f^a_1(b) & \triangleq & \underbrace{(f^a_0 \circ \ldots \circ f^a_0)}_{b}(a) & = & a+b & a-b \\
f^a_2(b) & \triangleq & \underbrace{(f^a_1 \circ \ldots \circ f^a_1)}_{b}(a) & = & a*b & \frac{b}{a} \\
f^a_3(b) & \triangleq & \underbrace{(f^a_2 \circ \ldots \circ f^a_2)}_{b}(a) & = & a^b & \mathsf{log}_a(b) \\
f^a_4(b) & \triangleq & \underbrace{(f^a_3 \circ \ldots \circ f^a_3)}_{b}(a) & = & \underbrace{a^{\cdot^{\cdot^{a}}}}_{b} & \mathsf{log}^*_a(b)
\end{array}$$

[4], yielding the sequence of functions Indeed, troubles increase as one goes up (down?) the inverse Ackermann hierarchy: although the standard library provides a log_2 function, it does not provide a log_b function

, nor any flavor of the iterated logarithm log^*_b or functions further in the hierarchy. And, of course, efficiently computing the inverse Ackermann function is harder than computing the inverse of any particular level of the hierarchy.

4. The Inverse Ackermann Hierarchy

In this section we build the inverse Ackermann hierarchy, defining and proving the inverse relationship between this heirarchy and the Ackermann hierarchy.

[4]The initial value of $f_i(0)$ is a little idiosyncratic: for $i \in \{1, 2\} itis 0; for i \not> 2$ it is 1.

4.1. The countdown operation

Firstly, to define some sort of "inverse" for the repeat application operation found in Definition 3.1, we define the following *countdown* operation:

Definition 4.1. Given a function $f : \mathbb{N} \to \mathbb{N}$. The *countdown* of f, denoted by f^*, is given by:

$$f^*(n) = \begin{cases} 0 & \text{if } n \leq \max\{0, 1, f(n)\} \\ 1 + f^*(f(n)) & \text{if } n > \max\{0, 1, f(n)\} \end{cases} \tag{6}$$

The important observation is that, if the sequence $\{n, f(n), f(f(n)), \ldots\}$ strictly decreases to 0, then f^* counts the minimum index where it reaches 1 or below. We give a formal definition for functions with such decreasing sequences:

Definition 4.2. A function $f : \mathbb{N} \to \mathbb{N}$ is a *contraction* if $f(0) = 0$ and $f(n) \leq n - 1\ \forall n > 0$.

Theorem 4.1. *If $f : \mathbb{N} \to \mathbb{N}$ is a contraction, then*

$$A = \left\{k : f^{(k)}(n) \leq 1\right\} \neq \varnothing \quad \textit{and} \quad f^*(n) = \min A$$

In other words,

$$\forall n, k \in \mathbb{N},\ f^*(n) \leq k \iff f^{(k)}(n) \leq 1$$

Proof sketch. By (6) and definition 4.2, we have

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

Then by a strong induction on n, the theorem holds. □

4.2. The inverse Ackermann hierarchy

With the above countdown operation in hand, we can build the inverse Ackermann hierarchy.

Definition 4.3. The inverse Ackermann hierarchy is a sequence of functions $\alpha_0, \alpha_1, \ldots$ recursively defined as:

1. $\alpha_0(n) = \max\{n - 2, 0\} \quad \forall n \in \mathbb{N}$.

2. $\alpha_m = \alpha^*_{m-1} \quad \forall m \in \mathbb{N}_{>0}$.

We will prove that each function α_m is a contraction, thus the $*$ operations are truly counting down to 1.

Theorem 4.2. *For all $m \in \mathbb{N}$, α_m is a contraction and*

$$\alpha_{m+1}(n) = \min\left\{k : \alpha_m^{(k)}(n) \leq 1\right\} \tag{7}$$

Proof. TODO TODO TODO □

Having sufficiently established the inverse Ackermann hierarchy, we now link it to the Ackermann hierarchy in definition 3.1. However, the relationship is not trivially clear, as the Peter-Ackermann function is in fact not exactly their inverses. We first define a *canonical* variant of the Ackermann hierarchy, then use it as an intermediate to link first two hierarchies.

4.3. The canonical Ackermann hierarchy

The canonical Ackermann hierarchy is a somewhat "cleaner" variant of the Ackermann hierarchy, with simpler initial values, but still built on the repeated application operation.

Definition 4.4. For any function $F : \mathbb{N} \to \mathbb{N}$, the *repeater* of F is another function $F^R : \mathbb{N} \to \mathbb{N}$ defined by:

$$F^R(n) = \begin{cases} 1 & \text{if } n = 0 \\ F\left(F^R(n-1)\right) & \text{if } n \geq 1 \end{cases} \tag{8}$$

In other words, $\forall n, F^R(n) = F^{(n)}\left(F^R(0)\right) = F^{(n)}(1)$.

Formally defining the repeated application operation helps us define the canonical Ackermann hierarchy in a neat way below.

Definition 4.5. The canonical Ackermann hierarchy is a sequence of functions $\mathrm{C}_0, \mathrm{C}_1, \ldots$ defined as:

1. $\mathrm{C}_0(n) = n + 2 \quad \forall n \in \mathbb{N}$.
2. $\mathrm{C}_m = \mathrm{C}^R_{m-1} \quad \forall m \in \mathbb{N}_{>0}$.

Firstly, we explore the relationship between C_m and α_m. We use the following theorem:

Theorem 4.3. *For all $m, n \in \mathbb{N}$, we have:*

$$\forall N \in \mathbb{N} : \alpha_m(N) \leq n \iff N \leq C_m(n) \tag{9}$$

In other words

$$C_m(n) = \max\{N : \alpha_m(N) \leq n\} \tag{10}$$

Before proving this theorem, we will need to prove each function in the inverse and canonical Ackermann hierarchy is increasing (non-strictly). Then (9) is sufficient to explain the "inverse" relationship between them.

Lemma 4.4. *For any function $f, F : \mathbb{N} \to \mathbb{N}$, if f is a contraction and $\forall n, N \in \mathbb{N}$, $f(N) \leq n \iff N \leq F(n)$, then*

$$\forall n, N \in \mathbb{N}, N \leq f^*(n) \iff N \leq F^R(n)$$

Proof. By Theorem 4.1 and Definition 4.4, we have: $f^*(N) \leq n \iff f^{(n)}(N) \leq 1 \iff N \leq F^{(n)}(1) = F^R(n)$ □

Now we can proceed with the proof for Theorem 4.3

Proof of Theorem 4.3. We prove by induction on m. The base case is the following statement:

$$\forall n, N \in \mathbb{N},\ N - 2 \leq n \iff N \leq n + 2$$

, which is trivial. The inductive step follows directly from Lemma 4.4. □

Now that we have established our canonical Ackermann hierarchy and its relation with the inverse Ackermann hierarchy, let us connect it to the original Ackermann hierarchy to complete the link.

Theorem 4.5. *For all $m, n \in \mathbb{N}$, we have $C_m(n + 2) = A_{m+1}(n) + 2$.*

Proof. TODO TODO TODO. □

4.4. Linking it all together

To conclude this section, we link everything together by stating and proving the main theorem that connects the inverse Ackermann hierarchy and the Ackermann hierarchy. We then use this theorem to state and prove a relation between the inverse Ackermann hierarchy and the inverse Ackermann function.

Theorem 4.6. *For all $m, n, k \in \mathbb{N}$, we have:*

$$n \leq A(m,k) \iff \begin{cases} n \leq k+1 & \text{if } m = 0 \\ \alpha_{m-1}(n+2) \leq k+2 & \text{if } m > 0 \end{cases} \tag{11}$$

Proof. TODO TODO TODO □

The next theorem is a corollary of the above, which lays the theoretical groundwork for us to compute the inverse Ackermann function in linear time.

Theorem 4.7. *For all $n\mathbb{N}$, we have:*

$$\alpha(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ 1 + \min\{m : \alpha_m(n+2) \leq m+3\} & \text{if } n \geq 2 \end{cases} \tag{12}$$

Proof. TODO TODO TODO □

In the next section, we will devise an algorithm to compute each of the functions in the inverse Ackermann hierarchy in linear time, and an algorithm to compute the inverse Ackermann function in linear time.

5. A linear time computation in Gallina

The main idea to compute the inverse Ackermann function using what we have established so far is to use Theorem 4.6 to iteratively compute each level in the hierarchy starting from input $n+2$, then stop when the condition $\alpha_m(n+2) \leq m+3$ is met.

In order for this computation to run in linear time, we need to first make sure *each* level $\alpha_m(n+2)$ is computed in linear time. We will then use a trick to achieve the linear time bound in the total computation using the relation

$$\alpha_m(n+2) = 1 + \alpha_m(\alpha_{m-1}(n+2)) \quad \forall m \in \mathbb{N}_{>0} \tag{13}$$

5.1. Inverse Ackermann hierarchy in linear time

First note that, in the Gallina specification, all natural numbers are represented with a string of S, the successor notation. All recursive functions in Gallina are required to decrease on one of its inputs, one or a few successors per recursive step. Thus all recursive functions, or Fixpoints in Gallina, must run in at least linear time over one of their inputs.

To compute the inverse Ackermann hierarchy for some input n with a Gallina-complied function, we look at its generalization: Given a contraction f over $\mathbb{N}$ and a Gallina function $\widetilde{f}$ computing f, we find a Gallina function $\widetilde{f^*}$ to compute its countdown, f^*.

Definition 5.1. Let $\mathcal{F}_k$ be the set of all Gallina functions $g : \mathbb{N}^k \to \mathbb{N}$. The *countdown recursor helper* is an operator $\mathrm{CRH} : \mathcal{F}_1 \to \mathcal{F}_3$ such that for all $g \in \mathcal{F}_1$ and $n_0, n_1, c \in \mathbb{N}$:

$$\mathrm{CRH}\,(g)\,(n_0, n_1, c) =$$
$$\begin{cases} 0 & \text{if } n_0 \leq 1. \\ 1 & \text{if } n_0 \geq 2, n_1 \leq 1. \\ 1 + \mathrm{CRH}\,(g)\,(n_0 - 1, g(n_1), n_1 - g(n_1) - 1) & \text{if } n_0 \geq 2, n_1 \geq 2, c = 0. \\ \mathrm{CRH}\,(g)\,(n_0 - 1, n_1, c - 1) & \text{if } n_0 \geq 2, n_1 \geq 2, c \geq 1. \end{cases} \tag{14}$$

It is trivial to see that for all $g \in \mathcal{F}$, $\mathrm{CRH}(g)$ is a Gallina Fixpoint (a Gallina-complied recursive function) in $\mathcal{F}_3$ since its first input n_0 decreases by 1 at every recursive step.

Definition 5.2. The *countdown recursor* is an operator $\mathrm{CR} : \mathcal{F}_1 \to \mathcal{F}_1$ such that for all $g \in \mathcal{F}_1$ and $n \in \mathbb{N}$:

$$\mathrm{CR}(g)(n) = \mathrm{CRH}(g)(n, g(n), n - g(n) - 1) \tag{15}$$

Since it is built by a composition of a Gallina Fixpoint CRH and a Gallina function g, $\mathrm{CR}(g)$ is indeed a Gallina function in $\mathcal{F}_1$. We prove that CR is the equivalence of the countdown operation in Gallina for contractions:

Lemma 5.1. *Let $f : \mathbb{N} \to \mathbb{N}$ be a contraction. Suppose a function $\widetilde{f} \in \mathcal{F}_1$ computes f, then $CR\left(\widetilde{f}\right)$ is a function in $\mathcal{F}_1$ computing f^*.*

Proof Sketch. Let $g := \widetilde{f}$. Firstly, from Definition 5.1, we can prove that for all $n_0, n1, c, k \in \mathbb{N}$ such that $n_0 \geq 2$ and $k \leq \min\{n_0, c\}$:

$$\mathrm{CRH}(g)\,(n_0, n_1, c) = \mathrm{CRH}(g)\,(n_0 - k, n_1, c - k)$$

This implies that if $n_0 \geq c + 2$, then

$$\mathrm{CRH}(g)\,(n_0, n_1, c) = 1 + \mathrm{CRH}(g)\,(n_0 - c - 1, g(n_1), n_1 - g(n_1) - 1)$$

If $n - 1 \geq g(n) \geq 1$, let $n_0 := n$, $n_1 := g(n)$, $c := n - g(n) - 1$ gives:

$$\begin{aligned}&\mathrm{CRH}(g)\,(n, g(n), n - g(n) - 1)\\ &= 1 + \mathrm{CRH}(g)\,(g(n), g(g(n)), g(n) - g(g(n)) - 1)\end{aligned}$$

Or

$$\mathrm{CR}(g)(n) = 1 + \mathrm{CR}(g)(g(n))$$

Together with the initial values of $\mathrm{CRH}(g)$, we conclude that $\mathrm{CRH}(g)$, or $\mathrm{CRH}\left(\widetilde{f}\right)$ indeed computes f^*.

□

With this lemma, we can define the equivalence of the inverse Ackermann hierarchy in Gallina:

Definition 5.3. The Gallina inverse Ackermann hierarchy is a sequence of functions $\widetilde{\alpha}_0, \widetilde{\alpha}_1, \ldots$ such that for all $m, n \in \mathbb{N}$:

$$\widetilde{\alpha}_m(n) = \begin{cases} n - 2 & \text{if } m = 0 \\ \mathrm{CR}\left(\widetilde{\alpha}_{m-1}\right)(n) & \text{if } m \geq 1 \end{cases} \tag{16}$$

Note that $x - y$ in Gallina is equivalent to $\max\{x - y, 0\}$ in practice.

Lemma 5.1 and Theorem 4.2 trivially implies that $\widetilde{\alpha}_m$ is a Gallina computation of α_m for all $m \in \mathbb{N}$.

The important thing to come up with Gallina computations for the hierarchy is we want to compute them in linear time. We assert the hierarchy $\{\widetilde{\alpha}_m\}$ succeeds in doing so:

Theorem 5.2. *For each $m \in \mathbb{N}$, computing $\widetilde{\alpha}_m(n)$ takes at most $(m + o(1))n$ steps, where n tends to infinity.*

Proof. TODO TODO TODO □

5.2. Inverse Ackermann in linear time

As mentioned, the task is to find the minimum x for which $\alpha_x(n) \leq x+3$ for $n \geq 4$. It is tempting to go for a naive approach, after all the efficiencies we have developed so far: Starting from $x = 0$, we iteratively compute $\alpha_x(n)$ and compare it with $x + 3$. However, by our earlier analysis in Theorem 5.2, the total amount of time needed is

$$T(n) = \sum_{m=0}^{\alpha(n)-1} (m + o(1))n = O\left(n\alpha(n)^2\right)$$

which is pretty efficient, given how slow-growing $\alpha(n)$ is. However, our ultimate goal is $O(n)$ time, thus we came up with a better approach. Interestingly, it is also based on Theorem 5.2. Specifically, using (6), Theorem 5.2 implies we can actually compute $\alpha_m(n)$ in $O(\alpha_{m-1}(n))$ time, *if* $\alpha_{m-1}(n)$ is given. It is thus beneficial if we retain the value of $\alpha_{m-1}(n)$ when computing $\alpha_m(n)$. The definition below is our Gallina recursor for this.

Definition 5.4. The inverse Ackermann recursor helper is a function $\mathrm{IARH} : \mathcal{F}_1 \times \mathbb{N}^3 \to \mathbb{N}$ such that for all $g \in \mathcal{F}_1, n, c_0, c \in \mathbb{N}$, we have:

$$\mathrm{IARH}(g, n, c_0, c) = \begin{cases} c & \text{if } c = 1 \\ 1 + \mathrm{IARH}\left(\mathrm{CR}(g), n, n - \mathrm{CR}(g)(n) - 1, c - 1\right) & \text{if } c \geq 1, c_0 = 0 \\ \mathrm{IARH}\left(g, n - 1, c_0 - 1, c - 1\right) & \text{if } c \geq 1, c_0 \geq 1 \end{cases} \tag{17}$$

Note that $n - 1$ in Gallina means $\max\{n - 1, 0\}$ here.

Definition 5.5. The inverse Ackermann recursor is a function $\mathrm{IAR} \in \mathcal{F}_1$ such that for all $n \in \mathbb{N}$, we have

$$\mathrm{IAR}(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ \mathrm{IARH}\left(\alpha_0, n + 2, 2, n\right) & \text{if } n \geq 2 \end{cases} \tag{18}$$

The following theorem is a central result in this paper, which asserts the correctness of IAR.

Theorem 5.3. *IAR is a Gallina function computing* $\alpha(n)$.

It is trivial to see that both IARH and IAR are Gallina functions. It suffices to show that $\mathrm{IAR}(n) = \alpha(n)$ for $n \geq 2$, since their values already match for $n \leq 1$. Furthermore, if we trace the first two recursive steps in IARH, we obtain

$$\begin{aligned}&\mathrm{IARH}(n+2, \alpha_0, 2, n)\\ &= 1 + \mathrm{IARH}(\alpha_1, n-2, n-\alpha_1(n)-1, n-3) \quad \forall n \geq 2\end{aligned}$$

It then suffices to show the following:

Lemma 5.4. *For all $n \geq 2$,*

$$\mathit{IARH}(\alpha_1, n, n-\alpha_1(n)-1, n-3) = \min\{m : \alpha_m(n+2) \leq m+3\}$$

.

Before proving Lemma 5.4, we need an intermediate lemma.

Lemma 5.5. *For all $n \geq 2$*