# Ackermann's Function in *Iterative* Form

A Subtle Termination Proof with Isabelle/HOL

Lawrence C Paulson FRS, Computer Laboratory, University of Cambridge
Isabelle Workshop 2020

# I. A Brief History of Ackermann’s Function

# Wilhelm Ackermann's "generalised exponential" (1928)

Wie man daraus erkennt, ist $\varphi(a, b, 1)$ mit $a \cdot b$ identisch. $\varphi(a, b, 2)$ stimmt mit $a^b$ überein. $\varphi(a, b, 3)$ ist die $b$-malige Iteration von $a^b$, genommen für $a$, usw. Unsere gesuchte Funktion erhalten wir nun, wenn wir in $\varphi(a, b, c)$ alle drei Argumente gleichnehmen. Wir behaupten also, $\varphi(a, a, a)$ kann nicht ohne Benutzung des zweiten Typs definiert werden[3]).

Die Ausschließung von simultanen Rekursionen ist für unsere Behauptung wesentlich. Es gelten nämlich die folgenden Formeln:

$$\varphi(a, b, 0) = a + b,$$

$$\varphi(a, 0, n+1) = \boxed{\alpha(a, n)},$$

$$\varphi(a, b+1, n+1) = \varphi(a, \varphi(a, b, n+1), n).$$

# Rózsa Péter's 2-argument function (1935)

Die Funktion $\varphi_0(n) = 2n$, welche der Ackermannschen Ausgangsfunktion $\psi_0(n, a) = n + a$ entspricht, genügt für $n = 0$ der Bedingung a) nicht. Statt ihrer nehme ich die ebenfalls lineare und den Bedingungen a), b) genügende Funktion $2n + 1$ als Ausgangsfunktion.

Also definiere ich die Funktionen $\varphi_m(n)$ wie folgt:

$$\varphi_0(n) = \boxed{2n + 1}$$

und für alle $m$

$$\begin{cases} \varphi_{m+1}(0) = \varphi_m(1) \\ \varphi_{m+1}(n+1) = \varphi_m(\varphi_{m+1}(n)). \end{cases}$$

# Raphael Robinson's refinement (1948)

2. **The majorizing function.** Let the function $G_n x$ be defined by the double recursion

$$G_0 x = Sx, \qquad G_{Sn} 0 = G_n 1, \qquad G_{Sn} Sx = G_n G_{Sn} x.$$

---

[3] W. Ackermann, *Zum Hilbertschen Aufbau der reelen Zahlen*, Math. Ann. vol. 99 (1928) pp. 118–133.

[4] R. Péter, *Über die mehrfache Rekursion*, Math. Ann. vol. 113 (1936) pp. 489–527.

# Basic facts about Ackermann's function, $\phi_m(n)$

- Its purpose was always to exhibit a computable function wasn't "recursive".
  - what we now call *primitive recursive* (PR)
  - if *f* is PR, then there exists *m* where $\phi_m$ is a *strict* upper bound for *f*
- It generates huge numbers: $\phi_4(3) = 2^{2^{65536}} - 3$
- Expressing it in most formal models of computation is difficult.

# II. Ackermann’s Function using a Stack

# Ackermann’s function in Isabelle

```
fun ack :: "[nat,nat] ⇒ nat" where
   "ack 0 n             = Suc n"
 | "ack (Suc m) 0       = ack m 1"
 | "ack (Suc m) (Suc n) = ack m (ack (Suc m) n)"
```

the <u>recursive</u> version that we all know and love

# A stack-oriented version as a *term rewriting system*

$$\Box \# n \# 0 \# L \longrightarrow \Box \# \mathrm{Suc}\, n \# L$$
$$\Box \# 0 \# \mathrm{Suc}\, m \# L \longrightarrow \Box \# 1 \# m \# L$$
$$\Box \# \mathrm{Suc}\, n \# \mathrm{Suc}\, m \# L \longrightarrow \Box \# n \# \mathrm{Suc}\, m \# m \# L$$

- The box constrains rewriting to the head of the list
- A stack represents a nest of calls: $\mathrm{ack}(k_n, \mathrm{ack}(k_n - 1, \ldots, k_1))$
- Does it terminate? No term rewriting termination checker knows!

# A stack-oriented computation of ack(2,3)

3 2
2 2 1 = ack(1,ack(2,2))
1 2 1 1
0 2 1 1 1
1 1 1 1 1
0 1 0 1 1 1
1 0 0 1 1 1
2 0 1 1 1
3 1 1 1
2 1 0 1 1
1 1 0 0 1 1
0 1 0 0 0 1 1
1 0 0 0 0 1 1
2 0 0 0 1 1
3 0 0 1 1
4 0 1 1

ack(2,2) = 7

ack(1,ack(1,5))

5 1 1
4 1 0 1
3 1 0 0 1
2 1 0 0 0 1
1 1 0 0 0 0 1
0 1 0 0 0 0 0 1
1 0 0 0 0 0 0 1
2 0 0 0 0 0 1
3 0 0 0 0 1
4 0 0 0 1
5 0 0 1
6 0 1

ack(1,7)

7 1
6 1 0
5 1 0 0
4 1 0 0 0
3 1 0 0 0 0
2 1 0 0 0 0 0
1 1 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0
2 0 0 0 0 0 0 0
3 0 0 0 0 0 0
4 0 0 0 0 0
5 0 0 0 0
6 0 0 0
7 0 0
8 0
9

what is the ordering here??

# Defining a recursive function **without** a proof of termination

```
function (domintros) ackloop :: "nat list ⇒ nat" where
  "ackloop (n # 0 # L)         = ackloop (Suc n # L)"
| "ackloop (0 # Suc m # L)     = ackloop (1 # m # L)"
| "ackloop (Suc n # Suc m # L) = ackloop (n # Suc m # m # L)"
| "ackloop [m] = m"
| "ackloop [] =  0"
```

- All recursion calls hold **conditionally**: only if the *domain predicate* holds
- Our task is to prove that the domain predicate is *always* true

# III. Verifying Ackermann’s Function in Isabelle/HOL

# Built-in properties of the domain predicate

```
ackloop_dom (Suc n # L) ⟹ ackloop_dom (n # 0 # L)
ackloop_dom (1 # m # L) ⟹ ackloop_dom (0 # Suc m # L)
ackloop_dom (n # Suc m # m # L) ⟹ ackloop_dom (Suc n # Suc m # L)
ackloop_dom [m]
ackloop_dom []
```

- It terminates for empty and single-element lists.
- It terminates for some longer lists.
- ***Does it terminate for all lists?***

# Proving termination in all cases: by induction on `ack m n`

$$\textit{ackloop\_dom}\ (\textit{ack}\ m\ n\ \#\ L) \Longrightarrow \textit{ackloop\_dom}\ (n\ \#\ m\ \#\ L)$$

*this implies termination for a longer list beginning with* `n` *and* `m`

The base case is `ack 0 n # L`

which reduces to `Suc n # L`, and we have (by definition)

$$\textit{ackloop\_dom}\ (\textit{Suc}\ n\ \#\ L) \Longrightarrow \textit{ackloop\_dom}\ (n\ \#\ 0\ \#\ L)$$

# Continuing the induction on `ack m n`

The case `ack (Suc m) 0 # L` reduces to `ack m 1 # L`

We have the *induction hypothesis*

$$\texttt{ackloop\_dom (ack m 1 \# L)} \Longrightarrow \texttt{ackloop\_dom (1 \# m \# L)}$$

then (by definition) `ackloop_dom (0 # Suc m # L)`

The case `ack (Suc m) (Suc n) # L` is similar, but needs 2 induction hyps

# The entire inductive proof is a one-liner!

```
lemma ackloop_dom_longer:
  "ackloop_dom (ack m n # L) ⟹ ackloop_dom (n # m # L)"
  by (induction m n arbitrary: L rule: ack.induct) auto
```

It's fully automatic, using the special Ackermann induction rule

# An auxiliary function to complete the proof

```
fun acklist :: "nat list ⇒ nat" where
  "acklist (n#m#L) = acklist (ack m n # L)"
| "acklist [m] = m"
| "acklist [] =  0"
```

- This formalises how the list $k_1, \ldots, k_n$ represents $\mathrm{ack}(k_n, \mathrm{ack}(k_n - 1, \ldots, k_1))$
- … and its induction rule is just right, case-splitting on whether $n < 2$.

# Terminating the termination argument

```
lemma ackloop_dom: "ackloop_dom L"
  by (induction L rule: acklist.induct) (auto simp: ackloop_dom_longer)
```

Another one-liner using a special induction and our lemma

```
termination ackloop
  by (simp add: ackloop_dom)
```

Finally, Isabelle recognises our function as total!

# Concluding the proof: Ackermann can be computed iteratively

```
lemma ackloop_acklist: "ackloop L = acklist L"
  by (induction L rule: ackloop.induct) auto
```

Equivalence between the term rewriting system
and direct calls to Ackermann's function

```
theorem ack: "ack m n = ackloop [n,m]"
  by (simp add: ackloop_acklist)
```

# Concluding remarks

- The verification of the iterative Ackermann function is easy in Isabelle/HOL
- … yet the termination of the term rewriting system is an **open question!**
- Implementations of Ackermann's function in > 200 different languages are available online:

  https://rosettacode.org/wiki/Ackermann_function

Funded by ERC Advanced Grant ALEXANDRIA (Project GA 742178).

René Thiemann investigated the rewrite systems.