Ackermann's Function Is Not Primitive Recursive

Lawrence C. Paulson

March 24, 2022

Abstract

Ackermann's function is defined in the usual way and a number of its elementary properties are proved. Then, the primitive recursive functions are defined inductively: as a predicate on the functions that map lists of numbers to numbers. It is shown that every primitive recursive function is strictly dominated by Ackermann's function. The formalisation follows an earlier one by Nora Szasz [1].

Contents

Remark. This development was part of the Isabelle distribution from 1997 to 2022. It has been transferred to the AFP, where it may be more useful.

1 Ackermann's Function and the PR Functions

This proof has been adopted from a development by Nora Szasz [1].

theory *Primrec* **imports** *Main* **begin**

1.1 Ackermann's Function

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*)

PROPERTY A 4

lemma *less-ack2* [*iff*]: *j* < *ack i j*
⟨*proof*⟩

PROPERTY A 5-, the single-step lemma

lemma *ack-less-ack-Suc2* [*iff*]: *ack i j* < *ack i* (*Suc j*)
⟨*proof*⟩

PROPERTY A 5, monotonicity for <

lemma *ack-less-mono2*: *j* < *k* ⟹ *ack i j* < *ack i k*
⟨*proof*⟩

PROPERTY A 5', monotonicity for ≤

lemma *ack-le-mono2*: *j* ≤ *k* ⟹ *ack i j* ≤ *ack i k*
⟨*proof*⟩

PROPERTY A 6

lemma *ack2-le-ack1* [*iff*]: *ack i* (*Suc j*) ≤ *ack* (*Suc i*) *j*
⟨*proof*⟩

PROPERTY A 7-, the single-step lemma

lemma *ack-less-ack-Suc1* [*iff*]: *ack i j* < *ack* (*Suc i*) *j*
⟨*proof*⟩

PROPERTY A 4'? Extra lemma needed for *CONSTANT* case, constant functions

lemma *less-ack1* [*iff*]: *i* < *ack i j*
⟨*proof*⟩

PROPERTY A 8

lemma *ack-1* [*simp*]: *ack* (*Suc 0*) *j* = *j* + *2*
⟨*proof*⟩

PROPERTY A 9. The unary *1* and *2* in *ack* is essential for the rewriting.

lemma *ack-2* [*simp*]: *ack* (*Suc* (*Suc 0*)) *j* = *2* ∗ *j* + *3*
⟨*proof*⟩

Added in 2022 just for fun

lemma *ack-3*: *ack (Suc (Suc (Suc 0))) j = 2 ^(j+3) − 3*
⟨*proof*⟩

PROPERTY A 7, monotonicity for < [not clear why *ack-1* is now needed first!]

lemma *ack-less-mono1-aux*: *ack i k < ack (Suc (i +i′)) k*
⟨*proof*⟩

lemma *ack-less-mono1*: *i < j ⟹ ack i k < ack j k*
⟨*proof*⟩

PROPERTY A 7', monotonicity for ≤

lemma *ack-le-mono1*: *i ≤ j ⟹ ack i k ≤ ack j k*
⟨*proof*⟩

PROPERTY A 10

lemma *ack-nest-bound*: *ack i1 (ack i2 j) < ack (2 + (i1 + i2)) j*
⟨*proof*⟩

PROPERTY A 11

lemma *ack-add-bound*: *ack i1 j + ack i2 j < ack (4 + (i1 + i2)) j*
⟨*proof*⟩

PROPERTY A 12. Article uses existential quantifier but the ALF proof used *k + 4*. Quantified version must be nested ∃ *k′*. ∀ *i j*. ...

lemma *ack-add-bound2*:
assumes *i < ack k j* **shows** *i + j < ack (4 + k) j*
⟨*proof*⟩

1.2 Primitive Recursive Functions

primrec *hd0* :: *nat list* ⇒ *nat* **where**
hd0 [] = 0
| *hd0 (m # ms) = m*

Inductive definition of the set of primitive recursive functions of type *nat list* ⇒ *nat*.

definition *SC* :: *nat list* ⇒ *nat*
where *SC l = Suc (hd0 l)*

definition *CONSTANT* :: *nat* ⇒ *nat list* ⇒ *nat*
where *CONSTANT k l = k*

definition *PROJ* :: *nat* ⇒ *nat list* ⇒ *nat*
where *PROJ i l = hd0 (drop i l)*

definition *COMP* :: [*nat list* ⇒ *nat*, (*nat list* ⇒ *nat*) *list*, *nat list*] ⇒ *nat*

where *COMP g fs l* = *g* (*map* (λ*f*. *f l*) *fs*)

fun *PREC* :: [*nat list* ⇒ *nat*, *nat list* ⇒ *nat*, *nat list*] ⇒ *nat*
where
PREC f g [] = *0*
| *PREC f g* (*x* # *l*) = *rec-nat* (*f l*) (λ*y r*. *g* (*r* # *y* # *l*)) *x*
— Note that *g* is applied first to *PREC f g y* and then to *y*!

inductive *PRIMREC* :: (*nat list* ⇒ *nat*) ⇒ *bool* **where**
SC: *PRIMREC SC*
| *CONSTANT*: *PRIMREC* (*CONSTANT k*)
| *PROJ*: *PRIMREC* (*PROJ i*)
| *COMP*: *PRIMREC g* ⟹ ∀*f* ∈ *set fs*. *PRIMREC f* ⟹ *PRIMREC* (*COMP g fs*)
| *PREC*: *PRIMREC f* ⟹ *PRIMREC g* ⟹ *PRIMREC* (*PREC f g*)

Useful special cases of evaluation

lemma *SC* [*simp*]: *SC* (*x* # *l*) = *Suc x*
⟨*proof*⟩

lemma *PROJ-0* [*simp*]: *PROJ 0* (*x* # *l*) = *x*
⟨*proof*⟩

lemma *COMP-1* [*simp*]: *COMP g* [*f*] *l* = *g* [*f l*]
⟨*proof*⟩

lemma *PREC-0*: *PREC f g* (*0* # *l*) = *f l*
⟨*proof*⟩

lemma *PREC-Suc* [*simp*]: *PREC f g* (*Suc x* # *l*) = *g* (*PREC f g* (*x* # *l*) # *x* # *l*)
⟨*proof*⟩

1.3 Main Result: Ackermann's Function is not Primitive Recursive

lemma *SC-case*: *SC l* < *ack 1* (*sum-list l*)
⟨*proof*⟩

lemma *CONSTANT-case*: *CONSTANT k l* < *ack k* (*sum-list l*)
⟨*proof*⟩

lemma *PROJ-case*: *PROJ i l* < *ack 0* (*sum-list l*)
⟨*proof*⟩

COMP case

lemma *COMP-map-aux*: ∀*f* ∈ *set fs*. *PRIMREC f* ∧ (∃*kf*. ∀*l*. *f l* < *ack kf* (*sum-list l*))
⟹ ∃*k*. ∀*l*. *sum-list* (*map* (λ*f*. *f l*) *fs*) < *ack k* (*sum-list l*)

⟨*proof*⟩

lemma *COMP-case*:
assumes *1*: $\forall\, l.\ g\ l < ack\ kg\ (sum\text{-}list\ l)$
and *2*: $\forall\, f \in set\ fs.\ PRIMREC\ f \wedge (\exists\, kf.\ \forall\, l.\ f\ l < ack\ kf\ (sum\text{-}list\ l))$
shows $\exists\, k.\ \forall\, l.\ COMP\ g\ fs\ \ l < ack\ k\ (sum\text{-}list\ l)$
⟨*proof*⟩

PREC case

lemma *PREC-case-aux*:
assumes *f*: $\bigwedge l.\ f\ l + sum\text{-}list\ l < ack\ kf\ (sum\text{-}list\ l)$
and *g*: $\bigwedge l.\ g\ l + sum\text{-}list\ l < ack\ kg\ (sum\text{-}list\ l)$
shows $PREC\ f\ g\ l + sum\text{-}list\ l < ack\ (Suc\ (kf + kg))\ (sum\text{-}list\ l)$
⟨*proof*⟩

proposition *PREC-case*:
$[\![\bigwedge l.\ f\ l < ack\ kf\ (sum\text{-}list\ l);\ \bigwedge l.\ g\ l < ack\ kg\ (sum\text{-}list\ l)]\!]$
$\Longrightarrow \exists\, k.\ \forall\, l.\ PREC\ f\ g\ l < ack\ k\ (sum\text{-}list\ l)$
⟨*proof*⟩

lemma *ack-bounds-PRIMREC*: $PRIMREC\ f \Longrightarrow \exists\, k.\ \forall\, l.\ f\ l < ack\ k\ (sum\text{-}list\ l)$
⟨*proof*⟩

theorem *ack-not-PRIMREC*:
$\neg\ PRIMREC\ (\lambda l.\ case\ l\ of\ [] \Rightarrow 0 \mid x\ \#\ l' \Rightarrow ack\ x\ x)$
⟨*proof*⟩

end

References

[1] N. Szasz. A machine checked proof that Ackermann's function is not primitive recursive. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 317–338. Cambridge University Press, 1993.