Ackermann's Function in Iterative Form: A Subtle Termination Proof with Isabelle/HOL

Lawrence C. Paulson

Computer Laboratory, University of Cambridge, England
lp15@cam.ac.uk

Abstract. An iterative version of Ackermann's function is proved equivalent to the familiar recursive definition. The proof is extremely short but subtle, proceeding by introducing a function as partial and afterwards proving it total. It's a nice demonstration of Isabelle/HOL's function definition package.

1 Ackermann's Function

In 1928, Wilhelm Ackermann exhibited a function that was obviously computable and total, yet could be proved not to belong to the class of primitive recursive functions [1, p. 272]. Simplified by Rózsa Péter and Raphael Robinson, it comes down to us in the following well-known form:

```
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)"
```

It is easy to see that the recursion is well-defined and terminating. In every recursive call, either the first or the second argument decreases by one, suggesting a termination ordering: the lexicographic combination of $<$ (on the natural numbers) for the two arguments.

Nevertheless, it's not straightforward to prove that *ack* belongs to the class of computable functions as defined by Turing machines, register machines or general recursive functions. Heavyweight results such as the recursion theorem seem to be necessary. This raises the question of whether Ackermann's function has some alternative definition that is easier to reason about, and in fact, iterative definitions exist. But then we must prove that the recursive and iterative definitions are equivalent.

2 An Iterative Version

We can express an iterative definition in terms of the following recursion on lists (where # denotes list "cons"):

$$
\begin{aligned}
n \# 0 \# L &\longrightarrow \mathrm{Suc}\, n \# L \\
0 \# \mathrm{Suc}\, m \# L &\longrightarrow 1 \# m \# L \\
\mathrm{Suc}\, n \# \mathrm{Suc}\, m \# L &\longrightarrow n \# \mathrm{Suc}\, m \# m \# L
\end{aligned}
$$

the idea being to replace the recursive calls by a stack. We hope to obtain

$$[n, m] \longrightarrow^* [\mathrm{ack}(m, n)].$$

An execution trace for $\mathrm{ack}(2, 3)$ looks like this:

```
3 2
2 2 1
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
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
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
```

We can regard these three reductions as constituting a term rewriting system, subject to the proviso that they can only rewrite starting with the head of the list. Equivalently, each rewrite rule can be imagined as beginning with an anchor symbol, say $\Box$:

$$\begin{aligned}
\Box \# n \# 0 \# L &\longrightarrow \Box \# \mathrm{Suc}\, n \# L \\
\Box \# 0 \# \mathrm{Suc}\, m \# L &\longrightarrow \Box \# 1 \# m \# L \\
\Box \# \mathrm{Suc}\, n \# \mathrm{Suc}\, m \# L &\longrightarrow \Box \# n \# \mathrm{Suc}\, m \# m \# L
\end{aligned}$$

Termination isn't obvious. In the first rewrite, the head of the list gets bigger while the list gets shorter, suggesting that the length of the list should be the primary termination criterion. But in the third rewrite, the list gets longer. One might imagine a more sophisticated approach to termination based on multisets or ordinals; these however could lead nowhere for the second rewrite when $m = 0$: then $0 \# 1 \# L \longrightarrow 1 \# 0 \# L$ and often these approaches ignore the order of the list elements.

Although some termination ordering surely exists,[1] this system is an excellent way to demonstrate another approach to proving termination: by explicit reasoning about the domain of definition. It is easy, using Isabelle/HOL's function definition package [2].

3 The Iterative Version in Isabelle/HOL

We transform the rewrite system into a tail-recursive function definition. The keyword *domintros* indicates that we wish to defer the termination proof and instead define a predicate for the domain of definition. The recursion equations will then be conditional on arguments that satisfy this predicate. Our goal is to show that the predicate is always satisfied.

```
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"
```

The domain predicate, which is called *ackloop_dom*, is automatically defined according to the recursive calls. It satisfies the following properties:[2]

```
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 []
```

[1] René Thiemann has kindly run some tests using termination checkers. Without the anchors, the rewrite system is non-terminating because rewrite rules can be applied within a list. With the anchors, no termination checker delivers a conclusion.

[2] For clarity, *Suc 0* has been replaced by *1*.

The predicate obviously holds for all lists of length less than two. The properties allow us to prove instances for longer lists (establishing termination of *ackloop* for those lists), but the necessary argument isn't obvious. At closer examination, remembering that *ackloop* embodies the recursion of Ackermann's function, we might come up with the following lemma:

```
ackloop_dom (ack m n # L) ⟹ ackloop_dom (n # m # L)
```

This could be the solution, since it implies that *ackloop* terminates on the list $n \# m \# L$ provided it terminates on $\mathrm{ack}(m, n) \# L$, which is shorter. And indeed it can easily be proved by mathematical induction on m followed by a further induction on n. If $m = 0$ then it simplifies to the first *ackloop_dom* property:

```
ackloop_dom (Suc n # L) ⟹ ackloop_dom (n # 0 # L)
```

In the $\mathrm{Suc}\, m$ case, after the induction on n, the $n = 0$ case simplifies to

```
ackloop_dom (ack m 1 # L) ⟹ ackloop_dom (0 # Suc m # L)
```

but from *ackloop_dom (ack m 1 # L)* the induction hypothesis yields *ackloop_dom (1 # m # L)*, from which we obtain *ackloop_dom (0 # Suc m # L)* by the second *ackloop_dom* property. The $\mathrm{Suc}\, n$ case is also straightforward:

```
ackloop_dom (ack (Suc m) (Suc n) # L) ⟹ ackloop_dom (Suc n # Suc m # L)
```

It needs the third *ackloop_dom* property and both induction hypotheses.

In Isabelle, the proof sketched above is a one-liner thanks to a special induction rule, *ack.induct*. Function definitions in Isabelle automatically yield an induction rule customised to the recursive calls. For *ack*, it simply has the effect of two nested mathematical inductions. The proof above reduces to a single induction followed by automation:

```
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
```

4 Completing the Proof

Given the lemma above, it's straightforward to prove that every list L satisfies *ackloop_dom* by induction on the length of L. If its length is shorter than two then the result is immediate, and otherwise it has the form $n \# m \# L$, which the lemma reduces to $\mathrm{ack}(m, n) \# L$ and we are finished by the induction hypothesis.

A shorter proof turns out to be possible. Consider what *ackloop* is actually supposed to do: to replace the first two list elements by an Ackermann's function application. The following function codifies this point.

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

As mentioned above, recursive function definitions automatically provide us with a customised induction rule. In the case of `acklist`, it performs exactly the case analysis sketched at the top of this section. So this proof is also a single induction followed by automation.

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

Now we can use the termination result just proved to make the recursion equations for `ackloop` unconditional. It is now accepted as a total function.

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

The equivalence between `ackloop` and `acklist` is another one-liner. The special induction rule for `ackloop` considers the five cases of that function's definition, which are all proved automatically.

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

The equivalence between the iterative and recursive definitions of Ackermann's function is now immediate.

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

5 Related Work and Conclusions

Nora Szasz [3] proved that Ackermann's function is not primitive recursive using an early type theory-based proof assistant (ALF).

Implementations of Ackermann's function in more than 200 different programming languages, including IBM 360 assembler and Algol 68, are available online at `https://rosettacode.org/wiki/Ackermann_function`. Many of these are iterative.

Proving the termination of the iterative version of Ackermann's function is by no means obvious, yet an extremely short machine formalisation can be carried out.

Acknowledgements. This work was supported by the ERC Advanced Grant ALEXANDRIA (Project GA 742178). René Thiemann investigated the termination of the corresponding rewrite systems.

References

1. S. C. Kleene. *Introduction to Metamathematics.* North-Holland, 1952.
2. A. Krauss. Partial and nested recursive function definitions in higher-order logic. *Journal of Automated Reasoning*, 44(4):303–336, 2010.
3. 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.