# Higher-Order Recursion Abstraction: How to Make Ackermann, Knuth and Conway Look Like a Bunch of Primitives, Figuratively Speaking

Baltasar Trancón y Widemann
Ilmenau University of Technology, DE
`baltasar.trancon@tu-ilmenau.de`

September 17, 2018

**Abstract**

The Ackermann function is a famous total recursive binary function on the natural numbers. It is the archetypal example of such a function that is not primitive recursive, in the sense of classical recursion theory. However, and in seeming contradiction, there are generalized notions of total recursion, for which the Ackermann function is in fact primitive recursive, and often featured as a witness for the additional power gained by the generalization. Here, we investigate techniques for finding and analyzing the primitive form of complicated recursive functions, namely also Knuth's and Conway's arrow notations, in particular by recursion abstraction, in a framework of functional program transformation.

## 1 Introduction

The Ackermann function is a famous total binary function on the natural numbers, given by the recursive equations:

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

It is not a particularly useful function in practice; its fame (and ubiquitous appearance in undergraduate computer science curricula) stems from the fact that it is one of the simplest examples of a total, recursive (that is, recursively computable) function which is not *primitive* recursive. The class of primitive recursive functions is defined inductively, characteristically involving a recursion operator that, simply speaking, allows the value of a function at $n$ depend on its value at $n-1$, without explicit self-reference.

Intuitively (and provably), the doubly nested self-reference of the Ackermann function in the third equation refuses to be adapted to the primitive recursion scheme. But one may find seemingly contradictory statements: the notion of primitive recursion is relative to its underlying datatype $\mathbb{N}$; the Ackermann function *does* have a representation in terms of analogous recursion operators for suitable, more powerful datatypes [6, 2]. A systematic introduction to recursion operators on lists [3] gives the Ackermann function (disguised as a list function) as a high-end example, but does not discuss the generalization to related functions. Here, we start off where they concluded. The purpose of the present article is threefold:

1. Recall from [3] how to obtain a "primitive" representation of Ackermann's and related functions by means of systematic *recursion abstraction* in a higher-order functional programming language.

2. Pinpoint how exactly these representations are "cheating" with respect to the original, first-order definition of primitive recursion.

3. Generalize the approach to (apparently) even more complicated recursive functions.

We begin by summarizing formal details about primitive recursion in mathematical theory and in functional programming. The expert reader is welcome to safely skip the next, preliminary section. The remainder of the article is about three concrete examples, in increasing order of logical complexity: the Ackermann function (archetypal), Knuth's up-arrows (slightly more powerful), Conway's chained arrows (significantly more powerful). They form a well-known hierarchy of operators for the arithmetics of "insanely large" numbers.

Even the non-expert reader might enjoy a quick glance ahead at Figures 1 and 2 on page 4, where the classical and primitive definition forms are contrasted in synopsis. With a little contemplation, all elements of the latter, except the *fold* operators, can be found verbatim in the former. The remainder of this article is a thorough exploration of the formal details that constitute their respective equivalence.

The program code scattered throughout this article is a valid literate Haskell program. It makes use of standard Haskell'98 features only, with the exception of some `Rank2Types` in meta-level analyses. Note that, for better fit with the mathematical notation, the first-fit rule of Haskell pattern matching is shunned; all overlapped patterns are explicitly guarded.

## 2 Primitive Recursion

### 2.1 Primitive Recursion in Mathematics

The conventional definition of the class of primitive recursive functions is inductive. It contains the nullary *zero* function and the unary *successor* function, and the projections, is closed under composition and under a certain recursion operator:

Let $e$ and $g$ be $k$-ary and $(k+2)$-ary primitive recursive functions, respectively. Then the unique $(k+1)$-ary function $h$ given by the recursive equations

$$\begin{aligned} h(x_1, \dots, x_k; 0) &= e(x_1, \dots, x_k) \\ h(x_1, \dots, x_k; n) &= g\big(x_1, \dots, x_k; n-1, h(x_1, \dots, x_k; n-1)\big) \qquad (n > 0) \end{aligned}$$

is also primitive recursive. Then the self-referential definition of $h$ can be replaced equivalently by a "primitive" one, specifying $e$ and $g$ instead. It has been shown by Ackermann that a precursor to the function now named after him is *not* of this form. Double recursion [7] has been introduced ad-hoc for Ackermann's and related functions, but obviously a nesting depth of two is no natural complexity limit—why not have triple recursion, and so forth? Less arbitrary limits are given by type systems that allow recursive computations on other datatypes than just natural numbers; see below.

### 2.2 Primitive Recursion in Functional Programming

The algebraic approach to functional programming [5, 1] generalizes the notion of a recursion operator to arbitrary algebraic datatypes (and with some effort, beyond). In a functional programming context, one can do away with the auxiliary arguments $x_1, \dots, x_k$ to the generating functions $e$ and $g$, by having them implicitly as free variables in the defining expression. One can also do away with the $(k+1)$-th argument without losing essential power [4], thus arriving at a scheme called variously *structural recursion*, *iteration*, *fold* or *catamorphism*.

In the base case of the natural numbers (represented here by the Haskell datatype *Integer*, tacitly excluding negative numbers and $\bot$), the operator is:

$$\begin{aligned} &\mathit{foldn}\ g\ e\ 0 &&= e \\ &\mathit{foldn}\ g\ e\ n \mid n > 0 &&= g\ (\mathit{foldn}\ g\ e\ (n-1)) \end{aligned}$$

This form, as well as the constructor operations *zero* and *succ*, can be deduced from the category-theoretic representation of the natural numbers as the initial algebra of a certain functor. We do not repeat the details here, but one consequence is highly relevant: Fold operators have the nice universal property of *uniqueness*. On one hand, we can of course retrieve the preceding definition, where the generated function is named $h$:

From $h \equiv \mathit{foldn}\ g\ e$ we can conclude

$$\begin{aligned} &h\ 0 && \equiv e \\ &h\ n \mid n > 0 && \equiv g\ (h\ (n-1)) \end{aligned}$$

But, on the other hand, the converse does also hold: From

$$\begin{aligned} &h\ 0 && \equiv e \\ &h\ n \mid n > 0 && \equiv g\ (h\ (n-1)) \end{aligned}$$

we may deduce that necessarily $h \equiv \mathit{foldn}\ g\ e$.

This will become our tool of recursion abstraction in the following sections. As pointed out in [3] for the Ackermann example, the characteristic feature of "morally primitive" recursive functions is that this abstraction needs to applied more than once.

The resulting recursive functions do not look particular powerful at first glance: mathematically, $\mathit{foldn}\ g\ e\ n$ is simply $g^n(e)$. Their hidden power comes from the polymorphic nature of the fold operator, which can define functions from the natural numbers to arbitrary underlying datatypes.

$$\begin{aligned} &\textbf{type}\ \mathit{FoldN}\ a = (a \to a) \to a \to \mathit{Integer} \to a \\ &\mathit{foldn} :: \mathit{FoldN}\ a \end{aligned}$$

The fold operator for lists is arguably the most popular one; see [3]. This is not only due to the fact that lists are the workhorse of datatypes: List folding is both reasonably simple regarding type and universal properties, and gives to nontrivial, useful recursive algorithms already from very simple generator functions.

$$\begin{aligned} &\mathit{foldr}\ g\ e\ [\,] && = e \\ &\mathit{foldr}\ g\ e\ (a : \mathit{xs}) && = g\ a\ (\mathit{foldr}\ g\ e\ \mathit{xs}) \end{aligned}$$

The type scheme for list folding is

$$\begin{aligned} &\textbf{type}\ \mathit{FoldR}\ a\ b = (a \to b \to b) \to b \to [\,a\,] \to b \\ &\mathit{foldr} :: \mathit{FoldR}\ a\ b \end{aligned}$$

where we now have two type parameters; namely $a$ for the list elements to be consumed, and $b$ for the underlying datatype in which to perform the computation.

The associated universal property is that for a recursive function $h$ on lists, the equations

$$\begin{aligned} &h\ [\,] \equiv e \\ &h\ (a : \mathit{xs}) \equiv g\ a\ (h\ \mathit{xs}) \end{aligned}$$

hold if and only if $h \equiv \mathit{foldr}\ g\ e$.

The trick behind apparently primitive representations of Ackermann and friends is to instantiate the fold type schemes with an underlying datatype that is strictly more powerful than the natural numbers, that is, that cannot be Gödel-encoded and -decoded by primitive recursive functions in the original sense.

Consider a type system that has some algebraic types. Then the class of primitive recursive functions *relative* to that type system is defined inductively as containing some elementary functions, and being closed under composition and the fold operators for all algebraic datatypes, instantiated with all valid types.

The classical definition can be retrieved as the special case of the type system whose only datatype is $\mathbb{N}$, and whose function types are all of the first-order form $\mathbb{N}^k \to \mathbb{N}$. The programme of the present article is to consider functions that are not primitive recursive in that classical sense, and to study the derivations of their respective primitive forms in a more powerful type system, namely one with higher-order function types, as usual in functional programming languages.

Our chief interest lies in the program transformation techniques necessary to take a *self-referential definition*, in terms of generally recursive equations, to a *primitive definition*, that is an expression without self-references, and composed only of legal building blocks for primitive recursive functions.

$$\begin{aligned}
A(0,n) &= n+1 \\
A(m,0) &= A(m-1,1) && (m>0) \\
A(m,n) &= A(m-1,A(m,n-1)) && (m,n>0) \\[1ex]
a \uparrow^0 b &= a\,b \\
a \uparrow^n 0 &= 1 \\
a \uparrow^n b &= a \uparrow^{n-1} (a \uparrow^n (b-1)) && (n,b>0) \\[1ex]
\langle p \to q \rangle &= (p+1)^{(q+1)} \\
\langle X \to p \to 0 \rangle &= \langle X \to p \rangle \\
\langle X \to 0 \to q \rangle &= \langle X \to 0 \rangle \\
\langle X \to p \to q \rangle &= \langle X \to (\langle X \to (p-1) \to q \rangle - 1) \to (q-1) \rangle && (p,q>0)
\end{aligned}$$

Figure 1: Exercise—Ackermann, Knuth and Conway in traditional appearance

## 2.3 Elimination of Recursion

The skeptical reader might wonder what qualifies the fold operators as particularly primitive, seeing that their definitions make blatant use of self-reference. But that is an artifact of the way algebraic datatypes are commonly defined. An alternative definition technique, namely *Church encoding*, does not have this property. A Church-encoded version of the natural numbers is the following non-recursive type definition:

$$\textbf{type}\ \mathit{Nat} = \forall\, a.\,(a \to a) \to a \to a$$

It allows the construction of numbers by the usual operations, by strategic deferral of their respective interpretations $z$ and $s$:

$$\begin{aligned}
&\mathit{zero} :: \mathit{Nat} && \mathit{succ} :: \mathit{Nat} \to \mathit{Nat} \\
&\mathit{zero}\ s\ z = z && \mathit{succ}\ n\ s\ z = s\ (n\ s\ z)
\end{aligned}$$

A trivial coercion to standard Haskell numbers can be given by providing the obvious interpretations.

$$\begin{aligned}
&\mathit{toInteger} :: \mathit{Nat} \to \mathit{Integer} \\
&\mathit{toInteger}\ n = n\ (+1)\ 0
\end{aligned}$$

But, less trivially, every Church-encoded number implements the action of the fold operator on itself, by substituting $g$ for $s$ and $e$ for $z$, respectively, which can thus be defined without self-reference as

$$\begin{aligned}
&\mathit{foldn}' :: (a \to a) \to a \to \mathit{Nat} \to a \\
&\mathit{foldn}'\ g\ e\ n = n\ g\ e
\end{aligned}$$

and is easily seen to be equivalent to the original:

$$\mathit{foldn}'\ g\ e \equiv \mathit{foldn}\ g\ e \circ \mathit{toInteger}$$

For a general discussion of this construction for arbitrary algebraic datatype signatures, and its equivalence to the initial algebra approach, see [8].

$$\begin{array}{lll} \mathit{ack} = & \mathit{foldn}\ (\lambda f \to \mathit{foldn}\ f\ (f\ 1)\ ) & (+1) \\ \mathit{knuth} = & \mathit{foldn}\ (\lambda f \to \mathit{foldn}\ f\ 1\ ) & \circ\,(*) \\ \mathit{cback} = & \mathit{foldr}\ (\lambda o\ k \to \mathit{foldn}\ (\lambda f \to \mathit{foldn}\ (f \circ (-\ 1))\ (k\ 0\ o))\ (\lambda p \to k\ p\ o))\ \mathit{cpow} & \\ \quad \textbf{where}\ \mathit{cpow}\ q\ p = (p+1)\ \hat{}\ (q+1) & & \end{array}$$

Figure 2: Solution—Ackermann, Knuth and Conway looking like a bunch of primitives

## 3 The Ackermann Function

### 3.1 Derivation

We start with the Haskell version of the original self-referential definition

$$\begin{array}{ll} \mathit{ack}_0\ 0\ \ n & = n+1 \\ \mathit{ack}_0\ m\ 0 \mid m > 0 & = \mathit{ack}_0\ (m-1)\ 1 \\ \mathit{ack}_0\ m\ n \mid m > 0 \wedge n > 0 & = \mathit{ack}_0\ (m-1)\ (\mathit{ack}_0\ m\ (n-1)) \end{array}$$

and proceed by successive application of elementary transformation steps. Each new version is indexed differently, such that all can coexist in a single literate Haskell source file for this whole article.

In a first transformation step, we exploit currying to eliminate the second function argument $n$ and thus unify the second and third equation in an auxiliary function *aux*.

$$\begin{array}{l} \mathit{ack}_1\ 0 \qquad\quad = (+1) \\ \mathit{ack}_1\ m \mid m > 0 = \mathit{aux} \\ \quad \textbf{where}\ \mathit{aux}\ 0 \qquad\quad = \mathit{ack}_1\ (m-1)\ 1 \\ \qquad\qquad \mathit{aux}\ n \mid n > 0 = \mathit{ack}_1\ (m-1)\ (\mathit{ack}_{1\mathrm{a}}\ m\ (n-1)) \end{array}$$

We factor out the common term *ack* $(m-1)$ by beta expansion to an argument $f$. What seems like a simple matter of abbreviation here will prove a very useful preparation step later.

$$\begin{array}{l} \mathit{ack}_{1\mathrm{a}}\ 0 \qquad\quad = (+1) \\ \mathit{ack}_{1\mathrm{a}}\ m \mid m > 0 = \mathit{aux}\ (\mathit{ack}_{1\mathrm{a}}\ (m-1)) \\ \quad \textbf{where}\ \mathit{aux}\ f\ 0 \qquad\quad = f\ 1 \\ \qquad\qquad \mathit{aux}\ f\ n \mid n > 0 = f\ (\mathit{ack}_{1\mathrm{a}}\ m\ (n-1)) \end{array}$$

The single most tricky point is to get rid of the back-reference from *aux* to *ack*. Note that *aux* is only ever applied to $f = \mathit{ack}\ (m-1)$, hence we may substitute[1]

$$\mathit{ack}\ m \equiv \mathit{aux}\ (\mathit{ack}\ (m-1)) \equiv \mathit{aux}\ f$$

and obtain:

$$\begin{array}{l} \mathit{ack}_{1\mathrm{b}}\ 0 \qquad\quad = (+1) \\ \mathit{ack}_{1\mathrm{b}}\ m \mid m > 0 = \mathit{aux}\ (\mathit{ack}_{1\mathrm{b}}\ (m-1)) \\ \quad \textbf{where}\ \mathit{aux}\ f\ 0 \qquad\quad = f\ 1 \\ \qquad\qquad \mathit{aux}\ f\ n \mid n > 0 = f\ (\mathit{aux}\ f\ (n-1)) \end{array}$$

Now we have *ack* in a form generated by the fold operator from a function *aux* whose definition is independent from *ack*. We may invoke the universal property and conclude:

$$\begin{array}{l} \mathit{ack}_2 = \mathit{foldn}\ \mathit{aux}\ (+1) \\ \quad \textbf{where}\ \mathit{aux}\ f\ 0 \qquad\quad = f\ 1 \\ \qquad\qquad \mathit{aux}\ f\ n \mid n > 0 = f\ (\mathit{aux}\ f\ (n-1)) \end{array}$$

[1]In case of doubt about the validity of this slightly simplified argument, see section 6 for details.

It remains to be established that the self-referential definition of the generator *aux* can be reduced to a primitive form. As it happens, it is already of the right shape.

$$\begin{aligned}&\mathit{ack}_3 = \mathit{foldn}\ \mathit{aux}\ (+1)\\&\quad \textbf{where}\ \mathit{aux}\ f = \mathit{foldn}\ f\ (f\ 1)\end{aligned}$$

So, in summary, we have the Ackermann function given by an entirely non-self-referential expression that invokes two nested instances of the fold operator.

$$\mathit{ack}_4 = \mathit{foldn}\ (\lambda f \to \mathit{foldn}\ f\ (f\ 1))\ (+1)$$

## 3.2 Verification

Since the final version of the Ackermann function admittedly looks very different from the original one, it is instructive to expand it back and demonstrate the inductive equivalence to the original recursive equations. As the base case for $n$ we have

$$\begin{aligned}\mathit{ack}_4\ 0\ n &\equiv \mathit{foldn}\ (\lambda f \to \mathit{foldn}\ f\ (f\ 1))\ (+1)\ 0\ n\\&\equiv (+1)\ n\\&\equiv n+1\end{aligned}$$

and as the inductive case we have

$$\begin{aligned}\mathit{ack}_4\ m \mid m>0 &\equiv \mathit{foldn}\ (\lambda f \to \mathit{foldn}\ f\ (f\ 1))\ (+1)\ \ m\\&\equiv (\lambda f \to \mathit{foldn}\ f\ (f\ 1))\ (\mathit{foldn}\ (\lambda f \to \mathit{foldn}\ f\ (f\ 1))\ (+1)\ (m-1))\\&\equiv (\lambda f \to \mathit{foldn}\ f\ (f\ 1))\ (\mathit{ack}_4\ (m-1))\\&\equiv \mathit{foldn}\ (\mathit{ack}_4\ (m-1))\ (\mathit{ack}_4\ (m-1)\ 1)\end{aligned}$$

which cannot be unfolded yet. Proceeding to the second argument, we have the base case

$$\begin{aligned}\mathit{ack}_4\ m\ 0 &\equiv \mathit{foldn}\ (\mathit{ack}_4\ (m-1))\ (\mathit{ack}_4\ (m-1)\ 1)\ 0\\&\equiv \mathit{ack}_4\ (m-1)\ 1\end{aligned}$$

and the inductive case

$$\begin{aligned}\mathit{ack}_4\ m\ n \mid n>0 &\equiv \mathit{foldn}\ (\mathit{ack}_4\ (m-1))\ (\mathit{ack}_4\ (m-1)\ 1)\ \ n\\&\equiv \mathit{ack}_4\ (m-1)\ (\mathit{foldn}\ (\mathit{ack}_4\ (m-1))\ (\mathit{ack}_4\ (m-1)\ 1)\ (n-1))\\&\equiv \mathit{ack}_4\ (m-1)\ (\mathit{ack}_4\ m\ (n-1))\end{aligned}$$

by back-substitution. $\square$

## 3.3 Analysis

So, in a sense, the Ackermann function is primitive after all! As discussed above, the polymorphic fold operator glosses over the fact that underlying datatypes of different, incommensurable power are used here. To make things more explicit, consider monomorphic instances of *foldn*:

$$\begin{aligned}&\mathit{foldn}_1 :: \mathit{FoldN}\ \mathit{Integer}\\&\mathit{foldn}_1 = \mathit{foldn}\\&\mathit{foldn}_2 :: \mathit{FoldN}\ (\mathit{Integer} \to \mathit{Integer})\\&\mathit{foldn}_2 = \mathit{foldn}\end{aligned}$$

Using these, we can make the different roles explicit:

$$\mathit{ack}_5 = \mathit{foldn}_2\ (\lambda f \to \mathit{foldn}_1\ f\ (f\ 1))\ (+1)$$

Contradiction is avoided by the fact that the higher-order datatype underlying the outer instance cannot be converted to and from the traditional first-order datatype underlying the inner one by means of primitive recursive conversion functions.

## 4 Knuth's Up-Arrows

A natural next candidate function to investigate is Knuth's up-arrow, which is known to have quite similar but mildly more flexible behavior than the Ackermann function. The usual recursive definition is, for $a, b \geqslant 0; n > 0$:

$$\begin{aligned} a \uparrow^1 b &= a^b \\ a \uparrow^n 0 &= 1 \\ a \uparrow^n b &= a \uparrow^{n-1} \big(a \uparrow^n (b-1)\big) \qquad (n > 1; b > 0) \end{aligned}$$

This can be canonically extended to $n = 0$:

$$\begin{aligned} a \uparrow^0 b &= a\, b \\ a \uparrow^n 0 &= 1 && (n > 0) \\ a \uparrow^n b &= a \uparrow^{n-1} \big(a \uparrow^n (b-1)\big) && (n, b > 0) \end{aligned}$$

which is also a neat case of consistent abuse of notation, seeing that

$$a \uparrow^n b = a \underbrace{\uparrow \cdots \uparrow}_{n} b$$

### 4.1 Derivation

This definition translates straightforwardly to Haskell:

$$\begin{aligned} &\mathit{knuth}_0\ a\ 0\ \ b && = a * b \\ &\mathit{knuth}_0\ a\ n\ 0 \mid n > 0 && = 1 \\ &\mathit{knuth}_0\ a\ n\ b \mid n > 0 \wedge b > 0 && = \mathit{knuth}_0\ a\ (n-1)\ (\mathit{knuth}_0\ a\ n\ (b-1)) \end{aligned}$$

In analogy to the Ackermann example, we curry argument $b$.

```
$knuth_1\ a\ 0 \qquad\quad = (a*)$
$knuth_1\ a\ n \mid n > 0 = aux$
  where $aux\ 0 \qquad\ = 1$
        $aux\ b \mid b > 0 = knuth_1\ a\ (n-1)\ (knuth_1\ a\ n\ (b-1))$
```

If we had tried this example first, we would probably have gotten stuck at this point! But following the procedure for the Ackermann function, we beta-expand the recursive subterm *knuth* $a\ (n-1)$, even if there is only one occurrence, and the step does not simplify anything at first glance.

```
$knuth_{1a}\ \ a\ 0 \qquad\quad = (a*)$
$knuth_{1a}\ \ a\ n \mid n > 0 = aux\ (knuth_{1a}\ a\ (n-1))$
  where $aux\ f\ 0 \qquad\ = 1$
        $aux\ f\ b \mid b > 0 = f\ (knuth_{1a}\ a\ n\ (b-1))$
```

We find that, in the context of the last equation, we can again substitute

$$\mathit{knuth}\ a\ n \equiv \mathit{aux}\ (\mathit{knuth}\ a\ (n-1)) \equiv \mathit{aux}\ f$$

and arrive at:

```
$knuth_{1b}\ \ a\ 0 \qquad\quad = (a*)$
$knuth_{1b}\ \ a\ n \mid n > 0 = aux\ (knuth_{1b}\ a\ (n-1))$
  where $aux\ f\ 0 \qquad\ = 1$
        $aux\ f\ b \mid b > 0 = f\ (aux\ f\ (b-1))$
```

Now we may invoke the universal property of the fold operator to abstract from the outer recursion first, as in the previous example.

$$\begin{aligned}
&\mathit{knuth}_{2\mathrm{a}}\ a = \mathit{foldn}\ \mathit{aux}\ (a*)\\
&\quad \textbf{where}\ \mathit{aux}\ f\ 0 = 1\\
&\quad\phantom{\textbf{where}}\ \mathit{aux}\ f\ b \mid b > 0 = f\ (\mathit{aux}\ f\ (b-1))
\end{aligned}$$

Or, alternatively, we may abstract from the inner recursion first.

$$\begin{aligned}
&\mathit{knuth}_{2\mathrm{b}}\ a\ 0 = (a*)\\
&\mathit{knuth}_{2\mathrm{b}}\ a\ n \mid n > 0 = (\lambda f \rightarrow \mathit{foldn}\ f\ 1)\ (\mathit{knuth}_{2\mathrm{b}}\ a\ (n-1))
\end{aligned}$$

Either way, double recursion abstraction leads to:

$$\mathit{knuth}_3\ a = \mathit{foldn}\ (\lambda f \rightarrow \mathit{foldn}\ f\ 1)\ (a*)$$

For the friends of point-free humor this is:

$$\mathit{knuth}_4 = \mathit{foldn}\ (\lambda f \rightarrow \mathit{foldn}\ f\ 1) \circ (*)$$

Visual type inference tells us that the nesting pattern is the same as for the Ackermann function, a first-order instance of fold nested within a second-order instance.

$$\mathit{knuth}_5 = \mathit{foldn}_2\ (\lambda f \rightarrow \mathit{foldn}_1\ f\ 1) \circ (*)$$

## 4.2 Verification

Although the mode of derivation has paralleled the, already verified, Ackermann function quite closely, it might be assuring to repeat the verification process for Knuth's arrows. As the base case for $n$ we have

$$\begin{aligned}
\mathit{knuth}_3\ a\ 0\ b &\equiv \mathit{foldn}\ (\lambda f \rightarrow \mathit{foldn}\ f\ 1)\ (a*)\ 0\ b\\
&\equiv (a*)\ b\\
&\equiv a * b
\end{aligned}$$

and as the inductive case we have:

$$\begin{aligned}
\mathit{knuth}_3\ a\ n \mid n > 0 &\equiv \mathit{foldn}\ (\lambda f \rightarrow \mathit{foldn}\ f\ 1)\ (a*)\ \ n\\
&\equiv (\lambda f \rightarrow \mathit{foldn}\ f\ 1)\ (\mathit{foldn}\ (\lambda f \rightarrow \mathit{foldn}\ f\ 1)\ (a*)\ (n-1))\\
&\equiv (\lambda f \rightarrow \mathit{foldn}\ f\ 1)\ (\mathit{knuth}_3\ a\ (n-1))\\
&\equiv \mathit{foldn}\ (\mathit{knuth}_3\ a\ (n-1))\ 1
\end{aligned}$$

As the base case for $b$ we have

$$\begin{aligned}
\mathit{knuth}_3\ a\ n\ 0 &\equiv \mathit{foldn}\ (\mathit{knuth}_3\ a\ (n-1))\ 1\ 0\\
&\equiv 1
\end{aligned}$$

and as the inductive case we have:

$$\begin{aligned}
\mathit{knuth}_3\ a\ n\ b \mid b > 0 &\equiv \mathit{foldn}\ (\mathit{knuth}_3\ a\ (n-1))\ 1\ \ b\\
&\equiv \mathit{knuth}_3\ a\ (n-1)\ (\mathit{foldn}\ (\mathit{knuth}_3\ a\ (n-1))\ 1\ (b-1))\\
&\equiv \mathit{knuth}_3\ a\ (n-1)\ (\mathit{knuth}_3\ a\ n\ (b-1))
\end{aligned}$$

$\square$

## 5 Conway's Chained Arrows

Knuth's up-arrow notation was conceived as an iterative generalization of the step from addition to multiplication and from there to exponentiation, remaining in the realm of binary operators. Conway's chained arrow notation releases the latter restriction to encode even huger numbers. It defines a single operator on a list of arbitrarily many numbers, traditionally called "chains" and written with interspersed arrows (hence the name). A typical self-referential definition is the following system of equations, where $p, q$ are numbers and $X$ is a nonempty subsequence:

$$\begin{aligned}
\langle p\rangle &= p \\
\langle p \to q\rangle &= p^q \\
\langle X \to p \to 1\rangle &= \langle X \to p\rangle \\
\langle X \to 1 \to q\rangle &= \langle X \to 1\rangle \\
\langle X \to p \to q\rangle &= \big\langle X \to \langle X \to (p-1) \to q\rangle \to (q-1)\big\rangle \qquad (p, q > 1) \\
&\text{with the canonical extension} \\
\langle\,\rangle &= 1
\end{aligned}$$

We have added angled brackets around chains to emphasize the relevance of bracketing: since a chain denotes a number, chains may be nested (as in the last equation); but unlike for many other arithmetic operators, a chain of more than two elements is not *merely* an abbreviation for the nested iteration of a binary operator $\to$. The full situation is complicated: even though generally $\langle o \to p \to q\rangle$ is different from both $\langle\langle o \to p\rangle \to q\rangle$ and $\langle o \to \langle p \to q\rangle\rangle$, this does not imply that the overall operation *cannot* be expressed by a fold operator acting binarily on the list of numbers—that will be exactly our next move.

### 5.1 Derivation

The above equations, read left-to-right, specify a total evaluation function. It can be translated to a Haskell function on lists of strictly positive numbers

```
conway₀ []                        = 1
conway₀ [p]                       = p
conway₀ [q, p]                    = p ^ q
conway₀ (1 : p : xs)              = conway₀ (p : xs)
conway₀ (q : 1 : xs)              = conway₀ (1 : xs)
conway₀ (q : p : xs) | p > 1 ∧ q > 1 = conway₀ ((q − 1) : p'       : xs)
  where p'                        = conway₀ (q       : (p − 1) : xs)
```

where the order is reversed, because Conway's notation matches from the right, but Haskell lists decompose from the left. In order to apply recursion abstraction successfully to this messy function, its arguments need to be regularized and reordered in a particular way. Firstly, we note that the first two cases, for lists of length less than two, are special, in the sense that they are never reached by recursion; the third case, for lists of length two exactly, is the actual recursive base case.

Secondly, we notice that the remaining cases all involve two or more numbers, and the nested recursion occurs quite inconveniently in second position in the list. Hence we separate a non-recursive front-end function for the first two cases

```
cfront₁ _ []           = 1
cfront₁ _ [p]          = p
cfront₁ b (q : p : xs) = b xs q p
```

and a recursive back-end function, with the first two list elements expanded and reordered, for the other cases

```
cback₁ []       q p          = p ^ q
cback₁ (o : ys) 1 p          = cback₁ ys p      o
cback₁ (o : ys) q 1          = cback₁ ys 1      o
```

$$\begin{array}{lll}
\mathit{cback}_1\ \mathit{xs}\quad q\ p \mid p > 1 \wedge q > 1 & = & \mathit{cback}_1\ \mathit{xs}\ (q-1)\ p' \\
\quad \textbf{where}\ p' & = & \mathit{cback}_1\ \mathit{xs}\ q\qquad (p-1)
\end{array}$$

such that

$$\mathit{conway} \equiv \mathit{cfront}\ \mathit{cback}$$

Note that the reordering of arguments, however unintuitive at first glance, moves the nested recursion $p'$ from the awkward middle to a more manageable final position.

Conway's arrows are traditionally defined starting from one rather than zero. But unlike for Knuth's arrows, there is no simple extension to zero arguments. In order to make the recursion pattern more similar to the previous ones, we consider a variant where all argument numbers are reduced by one in the front-end.[2]

$$\begin{array}{lll}
\mathit{cfront}_2\ \_\ [\,] & = & 1 \\
\mathit{cfront}_2\ \_\ [p] & = & p \\
\mathit{cfront}_2\ b\ \mathit{xs} \mid \mathit{length}\ \mathit{xs} > 1 & = & \mathit{cfront}_1\ b\ (\mathit{map}\ (-\ 1)\ \mathit{xs})
\end{array}$$

This gives us a slightly modified back-end:

$$\begin{array}{lll}
\mathit{cback}_2\ [\,]\qquad q\ p & = & (p+1)\ \hat{}\ (q+1) \\
\mathit{cback}_2\ (o : \mathit{ys})\ 0\ p & = & \mathit{cback}_2\ \mathit{ys}\ p\qquad o \\
\mathit{cback}_2\ (o : \mathit{ys})\ q\ 0 & = & \mathit{cback}_2\ \mathit{ys}\ 0\qquad o \\
\mathit{cback}_2\ \mathit{xs}\qquad q\ p \mid p > 0 \wedge q > 0 & = & \mathit{cback}_2\ \mathit{xs}\ (q-1)\ (p'-1) \\
\quad \textbf{where}\ p' & = & \mathit{cback}_2\ \mathit{xs}\ q\qquad (p\ -1)
\end{array}$$

Note the two corrections (+1) where arguments flow to results, and the single correction (− 1) where a recursive result flows back to an argument.

The first argument is the technical novelty of this exercise, because it is of list type. We tackle it by double currying:

$$\begin{array}{lll}
\mathit{cback}_3\ [\,] & = & \mathit{cpow} \\
\mathit{cback}_3\ \mathit{xs}@(o : \mathit{ys}) & = & \mathit{aux} \\
\quad \textbf{where}\ \mathit{aux}\ 0\ p & = & \mathit{cback}_3\ \mathit{ys}\ p\qquad o \\
\qquad\qquad \mathit{aux}\ q\ 0 & = & \mathit{cback}_3\ \mathit{ys}\ 0\qquad o \\
\qquad\qquad \mathit{aux}\ q\ p \mid p > 0 \wedge q > 0 & = & \mathit{cback}_3\ \mathit{xs}\ (q-1)\ (p'-1) \\
\qquad\qquad\quad \textbf{where}\ p' & = & \mathit{cback}_3\ \mathit{xs}\ q\qquad (p\ -1) \\
\mathit{cpow}\ q\ p = (p+1)\ \hat{}\ (q+1) & &
\end{array}$$

Note the auxiliary function *cpow*, and the use of *en passant* pattern matching to unify the different heads of the three recursive cases. The fold operator for lists takes generators with two arguments, a non-recursive one for the head and a recursive one for the tail. We shape the auxiliary function accordingly by beta expansion,

$$\begin{array}{lll}
\mathit{cback}_{3a}\ [\,] & = & \mathit{cpow} \\
\mathit{cback}_{3a}\ \mathit{xs}@(o : \mathit{ys}) & = & \mathit{aux}\ o\ (\mathit{cback}_{3a}\ \mathit{ys}) \\
\quad \textbf{where}\ \mathit{aux}\ o\ k\ 0\ p & = & k\qquad p\qquad o \\
\qquad\qquad \mathit{aux}\ o\ k\ q\ 0 & = & k\qquad 0\qquad o \\
\qquad\qquad \mathit{aux}\ o\ k\ q\ p \mid p > 0 \wedge q > 0 & = & \mathit{cback}_3\ \mathit{xs}\ (q-1)\ (p'-1) \\
\qquad\qquad\quad \textbf{where}\ p' & = & \mathit{cback}_3\ \mathit{xs}\ q\qquad (p\ -1)
\end{array}$$

make the, already familiar, context-sensitive substitution

$$\mathit{cback}\ \mathit{xs}@(o : \mathit{ys}) \equiv \mathit{aux}\ o\ (\mathit{cback}\ \mathit{ys}) \equiv \mathit{aux}\ o\ k$$

and eliminate the now unused variable *xs* to obtain:

[2]The notation (− 1) is actually (*subtract* 1) in Haskell.

```
cback_3b []       = cpow
cback_3b (o : ys) = aux o (cback_3b ys)
  where aux o k  0 p                 = k         p        o
        aux o k  q 0                 = k         0        o
        aux o k  q p | p > 0 ∧ q > 0 = aux o k (q − 1) (p′ − 1)
          where p′                   = aux o k q        (p  − 1)
```

We readily read off the universal property of list folding and deduce:

```
cback_4 = foldr aux cpow
  where aux o k 0 p                 = k         p        o
        aux o k q 0                 = k         0        o
        aux o k q p | p > 0 ∧ q > 0 = aux o k (q − 1) (p′ − 1)
          where p′                  = aux o k q        (p  − 1)
```

The remaining steps are business as usual. Eliminate $p$ from *aux* by currying,

```
cback_5 = foldr aux cpow
  where
    aux o k 0                = λp → k p o
    aux o k q | q > 0        = aux2
      where aux2 0           = k         0        o
            aux2 p | p > 0   = aux o k (q − 1) (p′ − 1)
              where p′       = aux o k q        (p  − 1)
```

abstract from the “productive” self-reference by beta expansion,

```
cback_5a = foldr aux cpow
  where
    aux o k 0                  = λp → k p o
    aux o k q | q > 0          = aux2 (aux o k (q − 1))
      where aux2 f 0           = k         0 o
            aux2 f p | p > 0   = f           (p′ − 1)
              where p′         = aux o k q (p  − 1)
```

and substitute

$$aux\ o\ k\ q \equiv aux2\ (aux\ o\ k\ (q-1)) \equiv aux2\ f$$

to obtain:

```
cback_5b = foldr aux cpow
  where
    aux o k      0             = λp → k p o
    aux o k      q | q > 0     = aux2 (aux o k (q − 1))
      where aux2 f 0           = k 0      o
            aux2 f p | p > 0   = f        (p′ − 1)
              where p′         = aux2 f (p  − 1)
```

Perform recursion abstraction in $q$,

```
cback_6 = foldr aux cpow
  where
    aux o k = foldn aux2 (λp → k p o)
      where aux2 f 0           = k 0      o
            aux2 f p | p > 0   = f        (p′ − 1)
              where p′         = aux2 f (p  − 1)
```

beta-expand the auxiliary expression $p'$,

$$\begin{array}{l}
\mathit{cback}_7 = \mathit{foldr}\ \mathit{aux}\ \mathit{cpow} \\
\quad \textbf{where} \\
\qquad \mathit{aux}\ o\ k = \mathit{foldn}\ \mathit{aux2}\ (\lambda p \to k\ p\ o) \\
\qquad\quad \textbf{where} \\
\qquad\qquad \mathit{aux2}\ f\ 0 \qquad\qquad = k\ 0\ o \\
\qquad\qquad \mathit{aux2}\ f\ p \mid p > 0 = (\lambda p' \to f\ (p' - 1))\ (\mathit{aux2}\ f\ (p - 1))
\end{array}$$

and perform recursion abstraction in $p$:

$$\begin{array}{l}
\mathit{cback}_8 = \mathit{foldr}\ \mathit{aux}\ \mathit{cpow} \\
\quad \textbf{where} \\
\qquad \mathit{aux}\ o\ k = \mathit{foldn}\ \mathit{aux2}\ (\lambda p \to k\ p\ o) \\
\qquad\quad \textbf{where}\ \mathit{aux2}\ f = \mathit{foldn}\ (\lambda p' \to f\ (p' - 1))\ (k\ 0\ o)
\end{array}$$

Final cosmetic translation for improved point-freeness:

$$\begin{array}{l}
\mathit{cback}_9 = \mathit{foldr}\ \mathit{aux}\ \mathit{cpow} \\
\quad \textbf{where} \\
\qquad \mathit{aux}\ o\ k = \mathit{foldn}\ \mathit{aux2}\ (\mathit{flip}\ k\ o) \\
\qquad\quad \textbf{where}\ \mathit{aux2}\ f = \mathit{foldn}\ (f \circ (-\ 1))\ (k\ 0\ o)
\end{array}$$

## 5.2 Verification

Verification is a more complex issue here because of the sheer number of cases in indirections. Since no significant new insights are to be gained, we give just one case, corresponding to the fourth equation of $\mathit{conway}_0$, for illustration purposes:

$$\begin{array}{rl}
\mathit{cfront}_2\ \mathit{cback}_8\ (1 : p : x) & \equiv \mathit{cfront}_1\ \mathit{cback}_8\ (\mathit{map}\ (-\ 1)\ (1 : p : x)) \\
& \equiv \mathit{cfront}_1\ \mathit{cback}_8\ (0 : p - 1 : x') \\
& \qquad \textbf{where}\ x' = \mathit{map}\ (-\ 1)\ x \\
& \equiv \mathit{cback}_8\ x'\ 0\ (p - 1) \\
& \equiv \qquad\quad \mathit{foldr}\ \mathit{aux}\ \mathit{cpow}\ x'\ 0\ (p - 1) \\
& \equiv \mathit{aux}\ o\ (\mathit{foldr}\ \mathit{aux}\ \mathit{cpow}\ y)\ 0\ (p - 1) \\
& \qquad \textbf{where}\ (o : y) = x' \\
& \equiv \mathit{foldn}\ \mathit{aux2}\ (\lambda p \to \mathit{foldr}\ \mathit{aux}\ \mathit{cpow}\ y\ p\ o)\ 0\ (p - 1) \\
& \equiv \qquad\qquad\quad (\lambda p \to \mathit{foldr}\ \mathit{aux}\ \mathit{cpow}\ y\ p\ o) \quad (p - 1) \\
& \equiv \mathit{foldr}\ \mathit{aux}\ \mathit{cpow}\ y\ (p - 1)\ o \\
& \equiv \mathit{cback}_8\ y\ (p - 1)\ o \\
& \equiv \mathit{cfront}_1\ \mathit{cback}_8\ (p - 1 : o : y) \\
& \equiv \mathit{cfront}_1\ \mathit{cback}_8\ (p - 1 : x') \\
& \equiv \mathit{cfront}_1\ \mathit{cback}_8\ (\mathit{map}\ (-\ 1)\ (p : x)) \\
& \equiv \mathit{cfront}_2\ \mathit{cback}_8\ (p : x)
\end{array}$$

□

## 5.3 Analysis

The type analysis of the recursion scheme of Conway's chained arrows is analogous to the preceding examples. The third, outermost layer of folding has yet a more complex underlying datatype.

$$\begin{array}{l}
\mathit{foldr}_3 :: \mathit{FoldR}\ \mathit{Integer}\ (\mathit{Integer} \to \mathit{Integer} \to \mathit{Integer}) \\
\mathit{foldr}_3 = \mathit{foldr}
\end{array}$$

$$\begin{array}{l}
\mathit{cback}_{10} = \mathit{foldr}_3\ \mathit{aux}\ \mathit{cpow} \\
\quad \textbf{where} \\
\qquad \mathit{aux}\ o\ k = \mathit{foldn}_2\ \mathit{aux2}\ (\mathit{flip}\ k\ o) \\
\qquad\quad \textbf{where}\ \mathit{aux2}\ f = \mathit{foldn}_1\ (f \circ (-\ 1))\ (k\ 0\ o)
\end{array}$$

## 6 Conclusion

Recursion abstraction is a sophisticated program transformation technique. Its straightforward applications are at the heart of the algebra-of-programs approach to functional programming. It is a matter of taste whether definitions in terms of recursion operators are more intuitive and readable than definitions in terms of self-reference. But without doubt, the primitive forms lends themselves more easily to formal, in particular equational, reasoning.

Iterated recursion abstraction is notably more tricky than just a single step. Self-references need to be eliminated by clever context-sensitive beta expansion. The Ackermann function is, as for many other questions, just the right introductory example to teach the technique. In particular, its argument order leads the way naturally. Conway's chained arrow notation, on the other hand, is a significantly more difficult nut to crack, and is rather at the high end of demonstrations for the potential of the technique.

The central trick has now been used four times in three exercises, so the general pattern should have become apparent. In summary, a multiply nested self-referential cycle in a definition is broken by bringing the function into the form

$$\begin{aligned}&h\;0 &&= e\\ &h\;n \mid n > 0 &&= g\;(h\;(n-1))\\ &\quad\textbf{where}\; g\;f = i\;(h\;n)\end{aligned}$$

where the auxiliary higher-order function $i$ typically arises "virtually" by beta abstraction from a given expression in which $h\;n$ occurs. Then one proceeds to

$$\begin{aligned}&h\;0 &&= e\\ &h\;n \mid n > 0 &&= g'\;(h\;(n-1))\\ &\quad\textbf{where}\; g'\;f = i\;(g'\;f)\end{aligned}$$

Globally, $g$ and $g'$ are quite different functions, but we find for $n > 0$:

$$\begin{aligned}&g\;\;(h\;(n-1)) \equiv i\;(h\;n) &&\equiv i\;(g\;\;(h\;(n-1)))\\ &g'\;(h\;(n-1)) &&\equiv i\;(g'\;(h\;(n-1)))\end{aligned}$$

That is, both $g\;f$ and $g'\;f$, where $f = h\;(n-1)$, are fixed points of $i$. Since we are in a functional programming language with unique fixed point semantics as the very foundation of self-referential definitions, we may substitute one for the other. Then the outer recursion can be reduced to primitive form

$$\begin{aligned}&h = \mathit{foldn}\;e\;g'\\ &\quad\textbf{where}\; g'\;f = i\;(g'\;f)\end{aligned}$$

and the inner self-reference of $g'$ can be processed further by recursion abstraction, using either the same or more basic techniques.

It seems plausible that this technique will work for a large class of multiply nested recursive functions. The auxiliary tactics that we have employed, namely moving nested recursion to final argument position and refactoring deep pattern matches, are possibly also more general heuristics, and merit further investigation.

### 6.1 Outlook

We close by posing several open problems as challenges to the reader:

1. Assess whether the typical, relative ease of inductive reasoning about recursive functions in primitive form carries over to well-known, albeit non-trivial identities concerning our example functions, such as:

   $$\mathit{ack}\;(m+2)\;n \equiv \mathit{knuth}\;2\;m\;(n+3) - 3$$

2. As an important special case of equational reasoning, demonstrate program simplifications, such as *fusion* rules, for expressions involving our example functions.

3. Use the higher-order primitive recursion framework as a toolkit for synthesizing novel functions with interesting, recursive equational definitions.

## Acknowledgments

Anonymous referees of a previous manuscript have made valuable contributions to the evolution of this article.

## References

[1] R. Bird and O. de Moor. *Algebra of Programming.* Prentice Hall, 1997.

[2] R. Cockett and T. Fukushima. About Charity. Technical report, University of Calgary, 1992.

[3] G. Hutton. A tutorial on the universality and expressiveness of fold. *J. Functional Programming*, 9(4):355–372, 1999.

[4] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–425, 1992.

[5] E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proc. International Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer, 1991.

[6] J. C. Reynolds. Three approaches to type structure. In *Proc. International Joint Conference on Theory and Practice of Software Development*, volume 185 of *Lecture Notes in Computer Science.* Springer, 1985.

[7] R. M. Robinson. Recursion and double recursion. *Bull. AMS*, 54(10):987–993, 1948.

[8] P. Wadler. Recursive types for free! Online manuscript, 1990.