JFP **15** (1): 1–13, 2005. © 2004 Cambridge University Press
DOI: 10.1017/S0956796804005313 Printed in the United Kingdom

THEORETICAL PEARL

Church numerals, twice!

RALF HINZE

Institut für Informatik III, Universität Bonn, Römerstraße 164, 53117 Bonn, Germany
(*e-mail:* `ralf@informatik.uni-bonn.de`)

Abstract

This pearl explains Church numerals, twice. The first explanation links Church numerals to Peano numerals via the well-known encoding of data types in the polymorphic λ-calculus. This view suggests that Church numerals are folds in disguise. The second explanation, which is more elaborate, but also more insightful, derives Church numerals from first principles, that is, from an algebraic specification of addition and multiplication. Additionally, we illustrate the use of the parametricity theorem by proving exponentiation as reverse application correct.

1 Introduction

Church (1941) devised the following scheme for representing natural numbers in the untyped λ-calculus: the natural number n is encoded by a function that applies its first argument n times to its second argument. Using a compositional style the first three natural numbers are defined

$$\begin{aligned} \ulcorner 0 \urcorner &= \lambda\varphi \,.\, id \\ \ulcorner 1 \urcorner &= \lambda\varphi \,.\, \varphi \\ \ulcorner 2 \urcorner &= \lambda\varphi \,.\, \varphi \cdot \varphi. \end{aligned}$$

In general, we have $\ulcorner n \urcorner = \lambda\varphi \,.\, \varphi^n$ where φ^n is given by $\varphi^0 = id$ and $\varphi^{n+1} = \varphi \cdot \varphi^n$. Building upon this representation the successor function reads

$$succ\ n = \lambda\varphi \,.\, \varphi \cdot n\ \varphi.$$

The following definitions of addition, multiplication, and exponentiation are due to Rosser.

$$\begin{aligned} m + n &= \lambda\varphi \,.\, m\ \varphi \cdot n\ \varphi \\ m \times n &= m \cdot n \\ m \uparrow n &= n\ m \end{aligned}$$

Interestingly, multiplication is implemented by function composition and exponentiation by reverse function application. It is relatively straightforward to prove the definitions correct: $succ\ \ulcorner n \urcorner = \ulcorner n+1 \urcorner$, $\ulcorner m \urcorner + \ulcorner n \urcorner = \ulcorner m+n \urcorner$, $\ulcorner m \urcorner \times \ulcorner n \urcorner = \ulcorner mn \urcorner$, and $\ulcorner m \urcorner \uparrow \ulcorner n \urcorner = \ulcorner m^n \urcorner$ – see Barendregt (1992) for an inductive proof.

The purpose of this pearl is to provide additional background and hopefully additional insights by deriving the Church numeral system in two different ways.

Though Church numerals were devised for the untyped λ-calculus, we will work in a typed setting: we use Girard's System F (Girard, 1972), also known as the polymorphic or second-order λ-calculus (Reynolds, 1974), augmented by inductive types (Mendler, 1991; Parigot, 1992). To avoid clutter, however, we usually omit type abstractions (written explicitly as $\Lambda A.e$) and type applications (written explicitly as $e\ [T]$).

2 Church numerals, first approach

The first derivation takes as a starting point the unary representation of the natural numbers, also known as the Peano numeral system:

$$\mathbf{data}\ \mathit{Nat} \quad = \quad \mathit{Zero} \mid \mathit{Succ}\ \mathit{Nat}\,.$$

Here n is represented by $\ulcorner n \urcorner = \mathit{Succ}^n\ \mathit{Zero}$, i.e. the successor function is applied n times to the constant zero. Arithmetic operations can be conveniently expressed in terms of the fold operator for Nat.

$$\begin{array}{lcl}
\mathit{fold} & : & \forall N.(N \to N) \to N \to \mathit{Nat} \to N \\
\mathit{fold}\ \mathit{succ}\ \mathit{zero}\ \mathit{Zero} & = & \mathit{zero} \\
\mathit{fold}\ \mathit{succ}\ \mathit{zero}\ (\mathit{Succ}\ n) & = & \mathit{succ}\ (\mathit{fold}\ \mathit{succ}\ \mathit{zero}\ n)
\end{array}$$

In essence, $\mathit{fold}\ \mathit{succ}\ \mathit{zero}$ replaces Zero by zero, Succ by succ and evaluates the resulting term. The recursion scheme captured by fold is known as *structural recursion* over the natural numbers, which is an instance of a more general scheme called primitive recursion. The fold operator satisfies the following so-called *universal property*, which provides the central key for reasoning about fold (Bird & de Moor, 1997; Hutton, 1999).

$$h = \mathit{fold}\ \varphi\ a \quad \Longleftrightarrow \quad \left\{ \begin{array}{lcl} h\ \mathit{Zero} & = & a \\ h\ (\mathit{Succ}\ n) & = & \varphi\ (h\ n) \end{array} \right.$$

The universal property states that $\mathit{fold}\ \varphi\ a$ is the *unique* solution of the recursion equations on the right. A simple consequence of the property is the *reflection law* $\mathit{fold}\ \mathit{Succ}\ \mathit{Zero} = \mathit{id}$ (simply put $h = \mathit{id}$, $a = \mathit{Zero}$, and $\varphi = \mathit{Succ}$).

Addition, multiplication, and exponentiation are given by

$$\begin{array}{lcl}
(+), (\times), (\uparrow) & : & \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat} \\
m + n & = & \mathit{fold}\ \mathit{Succ}\ n\ m \\
m \times n & = & \mathit{fold}\ (\mathit{add}\ n)\ \ulcorner 0 \urcorner\ m \\
m \uparrow n & = & \mathit{fold}\ (\mathit{mult}\ m)\ \ulcorner 1 \urcorner\ n\,.
\end{array}$$

Now, let us reinvent Church numerals using the Peano numerals as a starting point. For the sake of argument, assume that there are no **data** declarations so that we cannot introduce new constants. In this case, we can only treat Zero and Succ as variables and λ-abstract over them. Thus, $\mathit{Succ}^n\ \mathit{Zero}$ becomes

$$\lambda \mathit{succ}\,.\,\lambda \mathit{zero}\,.\,\mathit{succ}^n\ \mathit{zero}\,.$$

What is the type of this term? A possible choice is $(\mathit{Nat} \to \mathit{Nat}) \to \mathit{Nat} \to \mathit{Nat}$, but $(\mathit{Bool} \to \mathit{Bool}) \to \mathit{Bool} \to \mathit{Bool}$ works, as well. In fact, $(N \to N) \to N \to N$ is a sensible type, for all N. This motivates the following type definition.

$$\textbf{type } \mathit{Church} \quad = \quad \forall N.(N \to N) \to N \to N$$

How are the types Nat and Church related? Ideally, they should be isomorphic since both represent the same set, the set of natural numbers. And, in fact, they are. The conversion maps are given by

$$\begin{array}{lcl} \mathit{nat} & : & \mathit{Church} \to \mathit{Nat} \\ \mathit{nat}\ c & = & c\ \mathit{Succ}\ \mathit{Zero} \\ \mathit{church} & : & \mathit{Nat} \to \mathit{Church} \\ \mathit{church}\ n & = & \lambda \mathit{succ}\ \mathit{zero}.\mathit{fold}\ \mathit{succ}\ \mathit{zero}\ n. \end{array}$$

The proof of $\mathit{nat} \cdot \mathit{church} = \mathit{id}$ makes use of the universal property of fold.

$$\begin{array}{cl} & (\mathit{nat} \cdot \mathit{church})\ n \\ = & \quad \{ \text{ definition of } \mathit{nat} \text{ and } \mathit{church} \} \\ & (\lambda \mathit{succ}\ \mathit{zero}.\mathit{fold}\ \mathit{succ}\ \mathit{zero}\ n)\ \mathit{Succ}\ \mathit{Zero} \\ = & \quad \{ \beta\text{-conversion} \} \\ & \mathit{fold}\ \mathit{Succ}\ \mathit{Zero}\ n \\ = & \quad \{ \text{ reflection law } \} \\ & n \end{array}$$

The reverse direction, $\mathit{church} \cdot \mathit{nat} = \mathit{id}$, is more involving.

$$\begin{array}{cl} & (\mathit{church} \cdot \mathit{nat})\ c \\ = & \quad \{ \text{ definition of } \mathit{church} \text{ and } \mathit{nat} \} \\ & \lambda \mathit{succ}\ \mathit{zero}.\mathit{fold}\ \mathit{succ}\ \mathit{zero}\ (c\ \mathit{Succ}\ \mathit{Zero}) \end{array}$$

Now we are stuck. The universal property is not applicable since the arguments of fold, namely succ and zero, are unknowns. Instead we must apply the so-called *parametricity condition* of the type Church. Briefly, each polymorphic type gives rise to a general property that each element of the type satisfies (Wadler, 1989). For Church we obtain the following 'theorem for free'. Let $\mathit{xtimes} : \mathit{Church}$ and let A and A' be arbitrary types; then for all $\varphi : A \to A$, $\varphi' : A' \to A'$, and $h : A \to A'$

$$h \cdot \mathit{xtimes}\ [A]\ \varphi = \mathit{xtimes}\ [A']\ \varphi' \cdot h \quad \Longleftarrow \quad h \cdot \varphi = \varphi' \cdot h.$$

Intuitively, the type ensures that xtimes only composes its argument with itself: $\mathit{xtimes}\ \varphi = \varphi \cdot \ldots \cdot \varphi$. Thus, $h \cdot \varphi = \varphi' \cdot h$ implies $h \cdot (\varphi \cdot \ldots \cdot \varphi) = (\varphi' \cdot \ldots \cdot \varphi') \cdot h$. Setting $\mathit{xtimes} = c$, $\varphi = \mathit{Succ}$, $\varphi' = \mathit{succ}$, and $h = \mathit{fold}\ \mathit{succ}\ \mathit{zero}$, we have

$$\mathit{fold}\ \mathit{succ}\ \mathit{zero} \cdot c\ \mathit{Succ} = c\ \mathit{succ} \cdot \mathit{fold}\ \mathit{succ}\ \mathit{zero}, \tag{1}$$

provided $\mathit{fold}\ \mathit{succ}\ \mathit{zero} \cdot \mathit{Succ} = \mathit{succ} \cdot \mathit{fold}\ \mathit{succ}\ \mathit{zero}$. This equation, however, follows directly from the definition of fold. Using (1) we can complete the proof.

$$\begin{array}{cl} = & \quad \{ (1) \} \\ & \lambda \mathit{succ}\ \mathit{zero}.c\ \mathit{succ}\ (\mathit{fold}\ \mathit{succ}\ \mathit{zero}\ \mathit{Zero}) \end{array}$$

$$
\begin{array}{ll}
= & \{ \text{definition of } \mathit{fold} \} \\
& \lambda \mathit{succ}\ \mathit{zero} \,.\, c\ \mathit{succ}\ \mathit{zero} \\
= & \{ \eta\text{-conversion} \} \\
& c
\end{array}
$$

The isomorphism suggests that a Church numeral is a fold in disguise as each numeral can be rewritten into the form $\lambda \mathit{succ}\ \mathit{zero} \,.\, \mathit{fold}\ \mathit{succ}\ \mathit{zero}\ n$ for some n.

Functions on Nat are programmed using Zero, Succ, and fold. What are the corresponding operations on Church? For fold we calculate

$$
\begin{array}{ll}
& \mathit{fold}\ \varphi\ a\ n \\
= & \{ \beta\text{-conversion} \} \\
& (\lambda s\ z \,.\, \mathit{fold}\ s\ z\ n)\ \varphi\ a \\
= & \{ \text{definition of } \mathit{church} \} \\
& \mathit{church}\ n\ \varphi\ a.
\end{array}
$$

The encodings of the constructor functions Zero and Succ can be specified as follows.

$$
\begin{array}{rcl}
\mathit{zero} & = & \mathit{church}\ \mathit{Zero} \\
\mathit{succ}\ (\mathit{church}\ m) & = & \mathit{church}\ (\mathit{Succ}\ m)
\end{array}
$$

Given this specification it is straightforward to derive $\mathit{zero} = \lambda s\ z \,.\, z$ and $\mathit{succ}\ c = \lambda s\ z \,.\, s\ (c\ s\ z)$. To summarize, define the relation '$\sim$' by $n \sim c \iff \mathit{church}\ n = c \iff n = \mathit{nat}\ c$, then

$$
\left.
\begin{array}{r}
\mathit{Zero} \sim \mathit{zero} = \ulcorner 0 \urcorner \\
\mathit{Succ}\ n \sim \mathit{succ}\ c \\
\mathit{fold}\ \varphi\ a\ n = c\ \varphi\ a
\end{array}
\right\} \quad \Longleftarrow \quad n \sim c.
$$

Using this correspondence we can mechanically transform functions on Nat into operations on Church. For instance, the structurally recursive definitions of addition, multiplication, and exponentiation give rise to the following operations on Church.

$$
\begin{array}{lcl}
(+), (\times), (\uparrow) & : & \mathit{Church} \to \mathit{Church} \to \mathit{Church} \\
m + n & = & m\ \mathit{succ}\ n \\
m \times n & = & m\ (n+)\ \ulcorner 0 \urcorner \\
m \uparrow n & = & n\ (m\times)\ \ulcorner 1 \urcorner
\end{array}
$$

Comparing these definitions to the ones given in section 1 we see that we have found alternative implementations of '$+$', '$\times$', and '$\uparrow$'. Or, to put it negatively, the correspondence of Church to Nat does not explain Rosser's implementation of the arithmetic operations.

3 Church numerals, second approach

Ready for a second go? This time we start from an algebraic specification of addition and multiplication, where a specification consists of a signature and properties that the operations of the signature are required to satisfy. We consider the following five constants and operations.

$$\begin{array}{lcl} 0, 1 & : & \mathbb{N} \\ (+), (\times) & : & \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ nat & : & \mathbb{N} \to Nat \end{array}$$

Two points are worth noting. First, exponentiation is deliberately omitted from the signature – this design decision is clearly unmotivated and will be justified only later (see remark 1). Second, we include a so-called *observer function*, which maps elements of the new type $\mathbb{N}$ to elements of Nat. Observer functions allow us to distinguish elements of the new type. If there were none, then the equational specification below could be trivially satisfied by setting $\mathbb{N} = ()$; see Hughes (1995) for a more comprehensive discussion.

The set of natural numbers with addition and multiplication forms a commutative semiring, that is, $(\mathbb{N}; +; 0)$ and $(\mathbb{N}; \times; 1)$ are commutative monoids, 0 is the zero of '$\times$', and '$\times$' distributes over '$+$'. Interestingly, we will not require all of the laws. The following subset is sufficient for our purposes.

$$0 + x = x = x + 0$$
$$(x + y) + z = x + (y + z)$$
$$0 \times x = 0$$
$$1 \times x = x = x \times 1$$
$$(x \times y) \times z = x \times (y \times z)$$
$$(x + y) \times z = (x \times z) + (y \times z)$$

In addition, we must determine nat.

$$nat\ 0 = Zero \qquad (2)$$
$$nat\ (1 + x) = Succ\ (nat\ x) \qquad (3)$$

The most straightforward way to represent values of type $\mathbb{N}$ is by terms of the algebra. The data type $Expr$ implements the term algebra of $\mathbb{N}$.

$$\mathbf{data}\ Expr = Null \mid One \mid Expr :\!+ Expr \mid Expr :\!\times Expr$$

Each of the operations 0, 1, '$+$', and '$\times$' is simply implemented by the corresponding constructor: $0 = Null$, $1 = One$, $(+) = (:\!+)$, and $(\times) = (:\!\times)$. In other words, the operations do nothing. All the work is performed by the observer function nat, which can be seen as an interpreter for the arithmetic language. Let us derive its definition using the laws above. The first two cases are straightforward.

$$\begin{array}{lcl} nat\ Null & = & Zero \\ nat\ One & = & Succ\ Zero \end{array}$$

It is tempting to set $nat\ (m :\!+ n) = nat\ m + nat\ n$ with $(+) = fold\ Succ$, but this equation does not follow immediately from the laws. Instead, we proceed by making

https://doi.org/10.1017/S0956796804005313 Published online by Cambridge University Press

a further case distinction on m.

$$\begin{array}{lcl} nat\ (Null :\!+ a_1) & = & nat\ a_1 \\ nat\ (One :\!+ a_1) & = & Succ\ (nat\ a_1) \\ nat\ ((m :\!+ n) :\!+ a_1) & = & nat\ (m :\!+ (n :\!+ a_1)) \end{array}$$

Now we are stuck. There is no obvious way to simplify $nat\ ((m :\!\times n) :\!+ a_1)$. Again, we help ourselves by making a further case distinction on m.

$$\begin{array}{lcl} nat\ ((Null :\!\times a_2) :\!+ a_1) & = & nat\ a_1 \\ nat\ ((One :\!\times a_2) :\!+ a_1) & = & nat\ (a_2 :\!+ a_1) \\ nat\ (((m :\!+ n) :\!\times a_2) :\!+ a_1) & = & nat\ ((m :\!\times a_2) :\!+ ((n :\!\times a_2) :\!+ a_1)) \\ nat\ (((m :\!\times n) :\!\times a_2) :\!+ a_1) & = & nat\ ((m :\!\times (n :\!\times a_2)) :\!+ a_1) \end{array}$$

The last case, $nat\ (m :\!\times n)$, is an instance of the previous one (with $a_1 = Null$).

$$\begin{array}{lcl} nat\ (Null :\!\times a_2) & = & Zero \\ nat\ (One :\!\times a_2) & = & nat\ a_2 \\ nat\ ((m :\!+ n) :\!\times a_2) & = & nat\ ((m :\!\times a_2) :\!+ (n :\!\times a_2)) \\ nat\ ((m :\!\times n) :\!\times a_2) & = & nat\ (m :\!\times (n :\!\times a_2))\ . \end{array}$$

At this point the reader may wonder whether *nat* is really well-defined. Now, the case analysis is clearly exhaustive; termination can be established using a so-called *polynomial interpretation* of operations (Dershowitz & Jouannaud, 1990).

$$\begin{array}{rcl@{\qquad}rcl} Null_\tau & = & 2 & m :\!+_\tau n & = & 2m + n \\ One_\tau & = & 2 & m :\!\times_\tau n & = & m^2 n \end{array}$$

A multivariate polynomial op_τ of n variables is associated with each n-ary operation op. For each equation $nat\ l = \ldots\ nat\ r\ \ldots$ we must then show that $\tau\ l > \tau\ r$ for all variables (ranging over positive integers) where τ is given by $\tau(op\ e_1 \ldots e_n) = op_\tau(\tau\ e_1) \ldots (\tau\ e_n)$.

Furthermore, it is worth noting that the implementation does *not* satisfy the specification. The laws only hold under observation, that is, $0 + x = x$, for instance, is weakened to $nat\ (0 + x) = nat\ x$. As a consequence, *Nat* and *Expr* are not isomorphic. This is, however, typical of abstract types.

Remark 1

Why didn't we include exponentiation in the specification? The answer is simply that in this case the derivation no longer works: there is no way to simplify the call $nat\ ((((a_3 :\!\uparrow (n :\!\uparrow m)) :\!\times a_2) :\!+ a_1))$. Exponentiation lacks the property of associativity, which we used for rewriting nested additions and multiplications. □

Let us now try to improve the efficiency of *nat*. For a start, we can avoid the construction and deconstruction of many terms if we specialize *nat* for $e :\!+ a_1$ and $(e :\!\times a_2) :\!+ a_1$. We specify

$$\begin{array}{rcl} nat_1\ e\ a_1 & = & nat\ (e :\!+ a_1) \\ nat_2\ e\ a_2\ a_1 & = & nat\ ((e :\!\times a_2) :\!+ a_1). \end{array}$$

Given this specification we can easily derive the following implementation.

$$
\begin{array}{lcl}
nat_1\ Null & = & \lambda a_1 \,.\, nat\ a_1 \\
nat_1\ One & = & \lambda a_1 \,.\, Succ\ (nat\ a_1) \\
nat_1\ (m \mathbin{:+} n) & = & \lambda a_1 \,.\, nat_1\ m\ (n \mathbin{:+} a_1) \\
nat_1\ (m \mathbin{:\times} n) & = & \lambda a_1 \,.\, nat_2\ m\ n\ a_1 \\
nat_2\ Null & = & \lambda a_2\ a_1 \,.\, nat\ a_1 \\
nat_2\ One & = & \lambda a_2\ a_1 \,.\, nat_1\ a_2\ a_1 \\
nat_2\ (m \mathbin{:+} n) & = & \lambda a_2\ a_1 \,.\, nat_2\ m\ a_2\ ((n \mathbin{:\times} a_2) \mathbin{:+} a_1) \\
nat_2\ (m \mathbin{:\times} n) & = & \lambda a_2\ a_1 \,.\, nat_2\ m\ (n \mathbin{:\times} a_2)\ a_1
\end{array}
$$

The rewriting opens up further opportunities for improvement. Note that the parameter a_1 is eventually passed to nat in each case. Likewise, a_2 is eventually passed to nat_1. These observations suggest that we could try to advance the function calls and pass $nat\ a_1$ instead of a_1 and similarly $nat_1\ a_2$ instead of a_2. The idea can be formalized as follows (the new observer functions are called $\underline{nat}_1$ and $\underline{nat}_2$).

$$
\underline{nat}_1\ e\ \underline{a}_1 = nat\ (e \mathbin{:+} a_1) \quad \Longleftarrow \quad \underline{a}_1 = nat\ a_1 \tag{4}
$$

$$
\underline{nat}_2\ e\ \underline{a}_2\ \underline{a}_1 = nat\ (e \mathbin{:\times} a_2 \mathbin{:+} a_1) \quad \Longleftarrow \quad \underline{a}_1 = nat\ a_1 \ \wedge\ \underline{a}_2 = \underline{nat}_1\ a_2 \tag{5}
$$

Note that the parameter $\underline{a}_2$ equals $\underline{nat}_1\ a_2$ rather than $nat_1\ a_2$ since we want to avoid dependencies on the 'old' code. Given this specification it is straightforward to derive the following implementation of $\underline{nat}_1$.

$$
\begin{array}{lcl}
\underline{nat}_1\ Null & = & \lambda \underline{a}_1 \,.\, \underline{a}_1 \\
\underline{nat}_1\ One & = & \lambda \underline{a}_1 \,.\, Succ\ \underline{a}_1 \\
\underline{nat}_1\ (m \mathbin{:+} n) & = & \lambda \underline{a}_1 \,.\, \underline{nat}_1\ m\ (\underline{nat}_1\ n\ \underline{a}_1) \\
\underline{nat}_1\ (m \mathbin{:\times} n) & = & \lambda \underline{a}_1 \,.\, \underline{nat}_2\ m\ (\underline{nat}_1\ n)\ \underline{a}_1
\end{array}
$$

Let us calculate the definition of $\underline{nat}_2$. We assume $\underline{a}_1 = nat\ a_1$ and $\underline{a}_2 = \underline{nat}_1\ a_2$ and consider each of the four cases. Cases $e = Null$ and $e = One$:

$$
\begin{array}{cl}
 & \underline{nat}_2\ Null\ \underline{a}_2\ \underline{a}_1 \\
= & \quad \{\ \text{assumptions and (5)}\ \} \\
 & nat\ (Null \mathbin{:\times} a_2 \mathbin{:+} a_1) \\
= & \quad \{\ 0 \times x = 0 \text{ and } 0 + x = x\ \} \\
 & nat\ a_1 \\
= & \quad \{\ \underline{a}_1 = nat\ a_1\ \} \\
 & \underline{a}_1
\end{array}
\qquad
\begin{array}{cl}
 & \underline{nat}_2\ One\ \underline{a}_2\ \underline{a}_1 \\
= & \quad \{\ \text{assumptions and (5)}\ \} \\
 & nat\ (One \mathbin{:\times} a_2 \mathbin{:+} a_1) \\
= & \quad \{\ 1 \times x = x\ \} \\
 & nat\ (a_2 \mathbin{:+} a_1) \\
= & \quad \{\ \underline{a}_1 = nat\ a_1 \text{ and (4)}\ \} \\
 & \underline{nat}_1\ a_2\ \underline{a}_1 \\
= & \quad \{\ \underline{a}_2 = \underline{nat}_1\ a_2\ \} \\
 & \underline{a}_2\ \underline{a}_1.
\end{array}
$$

https://doi.org/10.1017/S0956796804005313 Published online by Cambridge University Press

Cases $e = m \dotplus n$ and $e = m \mathbin{\dot{\times}} n$:

$$
\begin{array}{cl}
 & \underline{nat}_2\ (m \dotplus n)\ \underline{a}_2\ \underline{a}_1 \\
= & \quad \{\ \text{assumptions and (5)}\ \} \\
 & nat\ ((m \dotplus n) \mathbin{\dot{\times}} a_2 \dotplus a_1) \\
= & \quad \{\ (x + y) \times z = (x \times z) + (y \times z) \\
 & \qquad \text{and } (x + y) + z = x + (y + z)\ \} \\
 & nat\ ((m \mathbin{\dot{\times}} a_2) \dotplus ((n \mathbin{\dot{\times}} a_2) \dotplus a_1)) \\
= & \quad \{\ \underline{a}_2 = \underline{nat}_1\ a_2 \text{ and (5)}\ \} \\
 & \underline{nat}_2\ m\ \underline{a}_2\ (nat\ ((n \mathbin{\dot{\times}} a_2) \dotplus a_1)) \\
= & \quad \{\ \text{assumptions and (5)}\ \} \\
 & \underline{nat}_2\ m\ \underline{a}_2\ (\underline{nat}_2\ n\ \underline{a}_2\ \underline{a}_1)
\end{array}
$$

$$
\begin{array}{cl}
 & \underline{nat}_2\ (m \mathbin{\dot{\times}} n)\ \underline{a}_2\ \underline{a}_1 \\
= & \quad \{\ \text{assumptions and (5)}\ \} \\
 & nat\ ((m \mathbin{\dot{\times}} n) \mathbin{\dot{\times}} a_2 \dotplus a_1) \\
= & \quad \{\ (x \times y) \times z = x \times (y \times z)\ \} \\
 & nat\ (m \mathbin{\dot{\times}} (n \mathbin{\dot{\times}} a_2) \dotplus a_1) \\
= & \quad \{\ \underline{a}_1 = nat\ a_1 \text{ and (5)}\ \} \\
 & \underline{nat}_2\ m\ (\underline{nat}_1\ (n \mathbin{\dot{\times}} a_2))\ \underline{a}_1 \\
= & \quad \{\ \text{definition of } \underline{nat}_1\ \} \\
 & \underline{nat}_2\ m\ (\underline{nat}_2\ n\ (\underline{nat}_1\ a_2))\ \underline{a}_1 \\
= & \quad \{\ \underline{a}_2 = \underline{nat}_1\ a_2\ \} \\
 & \underline{nat}_2\ m\ (\underline{nat}_2\ n\ \underline{a}_2)\ \underline{a}_1.
\end{array}
$$

A final generalization step[1] yields:

$$
\begin{array}{lcl}
\underline{nat}_2\ Null & = & \lambda \underline{a}_2\ \underline{a}_1 . \underline{a}_1 \\
\underline{nat}_2\ One & = & \lambda \underline{a}_2\ \underline{a}_1 . \underline{a}_2\ \underline{a}_1 \\
\underline{nat}_2\ (m \dotplus n) & = & \lambda \underline{a}_2\ \underline{a}_1 . \underline{nat}_2\ m\ \underline{a}_2\ (\underline{nat}_2\ n\ \underline{a}_2\ \underline{a}_1) \\
\underline{nat}_2\ (m \mathbin{\dot{\times}} n) & = & \lambda \underline{a}_2\ \underline{a}_1 . \underline{nat}_2\ m\ (\underline{nat}_2\ n\ \underline{a}_2)\ \underline{a}_1.
\end{array}
$$

The code looks familiar. We are pretty close to Rosser's implementation of addition and multiplication. As a last step we simply remove the interpretative layer. Specifying 0, 1, '+', and '×' by

$$
\begin{array}{lcl}
0 & = & \underline{nat}_2\ Null \\
1 & = & \underline{nat}_2\ One \\
\underline{nat}_2\ m + \underline{nat}_2\ n & = & \underline{nat}_2\ (m \dotplus n) \\
\underline{nat}_2\ m \times \underline{nat}_2\ n & = & \underline{nat}_2\ (m \mathbin{\dot{\times}} n),
\end{array}
$$

we obtain the definitions given in section 1. We have even derived the type of Church numerals: $\underline{nat}_2$ has type $\forall N . Expr \to (N \to N) \to N \to N$ $(= Expr \to Church)$. Interestingly, the type of $\underline{nat}_2$ is more general than one would expect. By contrast, $\underline{nat}_1$ has type $Expr \to Nat \to Nat$ because of the occurrence of $Succ$ in the equation for $\underline{nat}_1\ One$.

If we look at the derivation of $\underline{nat}_2$, we notice that we have only used the algebraic properties of 0, 1, '+', and '×' but not the specification of nat. This observation motivates the following generalization of (4) and (5): Let A be an arbitrary type and let $h : Church \to A$ be an arbitrary function; then $c : Church$ satisfies $R_2(c\ [A], c)$, where

$$
\begin{array}{lcl}
R_0(\underline{e}, e) & \iff & \underline{e} = h\ e \\
R_1(\underline{e}, e) & \iff & \underline{e}\ \underline{a}_1 = h\ (e + a_1) \;\Longleftarrow\; R_0(\underline{a}_1, a_1) \\
R_2(\underline{e}, e) & \iff & \underline{e}\ \underline{a}_2\ \underline{a}_1 = h\ (e \times a_2 + a_1) \;\Longleftarrow\; R_0(\underline{a}_1, a_1) \ \wedge\ R_1(\underline{a}_2, a_2).
\end{array}
$$

[1] We have derived $\underline{nat}_2\ Null\ (\underline{nat}_1\ a_2)\ (nat\ a_1) = nat\ a_1$ etc. Now, we generalize $\underline{nat}_1\ a_2$ and $nat\ a_1$ to fresh variables, say, $\underline{a}_2$ and $\underline{a}_1$.

https://doi.org/10.1017/S0956796804005313 Published online by Cambridge University Press

This can be seen as *the* specification of Church numerals, from which we can derive the definitions of 0, 1, '+', and '×'. An important special case is obtained for $a_2 = 1$ and $a_1 = 0$:

$$h\ c = c\ [A]\ \underline{a}_2\ \underline{a}_1 \iff \begin{cases} h\ 0 & = \ \underline{a}_1 \\ h\ (1 + a_1') & = \ \underline{a}_2\ (h\ a_1'). \end{cases} \tag{6}$$

Note that the implication has been strengthened to an equivalence. Furthermore, note that (6) corresponds to the universal property of *fold*! Thus, using (6) we can derive the alternative definitions of '+', '×', and '↑' given in section 2. Additionally, from the specification of *nat*, equations (2) and (3), we can immediately conclude that $nat\ c = c\ Succ\ Zero$.

4 Exponentiation as reverse application

Rosser's definition of exponentiation seems to be peculiar. One property that sets it apart from the other operations is that it makes non-trivial use of polymorphism. Compare the definitions of '+', '×', and '↑' (in this section we will be explicit about type abstractions and type applications—with the exception of *id* and '·'). Let $\bar{T} = T \to T$; then

$$\begin{array}{lcl} m + n & = & \Lambda N\ .\ \lambda\varphi : \bar{N}\ .\ m\ [N]\ \varphi \cdot n\ [N]\ \varphi \\ m \times n & = & \Lambda N\ .\ m\ [N] \cdot n\ [N] \\ m \uparrow n & = & \Lambda N\ .\ (n\ [\bar{N}])\ (m\ [N]). \end{array}$$

Exponentiation is the only operation whose arguments are instantiated to two different types. This observation suggests that we cannot reasonably expect to derive exponentiation in an algebraic manner. Hence, we make do with proving its correctness. Now, it is straightforward to show that (omitting type arguments)

$$\ulcorner m \urcorner \uparrow \ulcorner n \urcorner = \ulcorner n \urcorner\ \ulcorner m \urcorner = \underbrace{\ulcorner m \urcorner \cdot \ldots \cdot \ulcorner m \urcorner}_{n \text{ times}} = \underbrace{\ulcorner m \urcorner \times \cdots \times \ulcorner m \urcorner}_{n \text{ times}} = \ulcorner m^n \urcorner.$$

But, can we also verify the correctness of '↑' without making assumptions about the arguments? The answer is in the affirmative. In the sequel we show that the two definitions of exponentiation are equal using type-theoretic arguments only. The proof of $(n\ [\bar{N}])\ (m\ [N]) = n\ [Church]\ (m\times)\ \ulcorner 1 \urcorner\ [N]$ proceeds in three major steps, each of which appeals to parametricity. Thus, the following can be seen as an instructive exercise in the use of the parametricity theorem.

$$\begin{array}{cl} & n\ [Church]\ (m\times)\ \ulcorner 1 \urcorner\ [N]\ f \\ = & \quad \{\ \text{Lemma 1}\ \} \\ & n\ [\bar{\bar{N}}]\ (m\ [N]\ \cdot)\ id\ f \\ = & \quad \{\ \text{define } const\ a\ b = a\ \} \\ & n\ [\bar{\bar{N}}]\ (m\ [N]\ \cdot)\ id\ (const\ f\ g) \\ = & \quad \{\ \text{Lemma 2}\ \} \\ & n\ [\bar{\bar{N}}]\ (m\ [N]\ \cdot)\ (const\ f)\ g \end{array}$$

https://doi.org/10.1017/S0956796804005313 Published online by Cambridge University Press

$$\begin{aligned} &= \quad \{ \text{Lemma 3} \} \\ &\quad n\ [\bar{N}]\ (m\ [N])\ f \end{aligned}$$

Lemma 1 shows that applying a Church numeral to polymorphic arguments and instantiating the result is the same as first instantiating the arguments and then applying the numeral.

Lemma 1

Let $xtimes : Church$ be a Church numeral, let T be a type, and let $\varphi : Church \to Church$, $\varphi' : \bar{\bar{T}} \to \bar{\bar{T}}$. Then

$$xtimes\ [Church]\ \varphi\ a\ [T] = xtimes\ [\bar{\bar{T}}]\ \varphi'\ (a\ [T]) \quad \Longleftarrow \quad \varphi\ b\ [T] = \varphi'\ (b\ [T]).$$

Proof

The proposition is implied by the free theorem for $Church$ with $A = Church$ and $A' = \bar{\bar{T}}$. The types A and A' suggest that $h : Church \to \bar{\bar{T}}$ is type instantiation: $h = \lambda c\,.\,c\ [T]$. The premise of the free theorem is easily checked:

$$\begin{aligned} &h \cdot \varphi = \varphi' \cdot h \\ \Longleftrightarrow &\quad \{ \text{definition of } h \} \\ &\varphi\ b\ [T] = \varphi'\ (b\ [T]) \quad \Box \end{aligned}$$

Lemma 2 expresses that postcomposition commutes with precomposition.

Lemma 2

Let $xtimes : Church$ be a Church numeral, let T be a type, and let $\varphi, f, g : \bar{T}$. Then

$$xtimes\ [\bar{T}]\ (\varphi\ \cdot)\ f \cdot g \quad = \quad xtimes\ [\bar{T}]\ (\varphi\ \cdot)\ (f \cdot g).$$

Proof

Again, the proposition follows from the free theorem for $Church$ with $A = A' = \bar{T}$ and $h = (\cdot\ g)$. The premise of the free theorem holds unconditionally.

$$\begin{aligned} &h \cdot (\varphi\ \cdot) = (\varphi\ \cdot) \cdot h \\ \Longleftrightarrow &\quad \{ \text{definition of } h \} \\ &(\varphi \cdot f) \cdot g = \varphi \cdot (f \cdot g) \\ \Longleftrightarrow &\quad \{ \text{associativity of `$\cdot$'} \} \\ &true \quad \Box \end{aligned}$$

Setting $f = id$, we obtain as a simple consequence $xtimes\ [\bar{T}]\ (\varphi\ \cdot)\ id\ (g\ a) = xtimes\ [\bar{T}]\ (\varphi\ \cdot)\ g\ a$.

Lemma 3 relates function composition and composition of postcompositions.

Lemma 3

Let $xtimes : Church$ be a Church numeral, let T be a type, and let $\varphi : \bar{T}$. Then

$$const\ (xtimes\ [T]\ \varphi\ a) \quad = \quad xtimes\ [\bar{T}]\ (\varphi\ \cdot)\ (const\ a).$$

https://doi.org/10.1017/S0956796804005313 Published online by Cambridge University Press

Proof
We apply the free theorem for *Church* with $A = T$ and $A' = \bar{T}$. The types more or less dictate that $h : T \to \bar{T}$ is *const*. It remains to verify the premise:

$$
\begin{aligned}
& \mathit{const} \cdot \varphi = (\varphi\, \cdot) \cdot \mathit{const} \\
\iff & \quad \{ \text{ operator sections: } (a \times)\, b = a \times b \ \} \\
& \mathit{const}\ (\varphi\ a)\ b = (\varphi \cdot \mathit{const}\ a)\ b \\
\iff & \quad \{ \text{ definition of `$\cdot$' } \} \\
& \mathit{const}\ (\varphi\ a)\ b = \varphi\ (\mathit{const}\ a\ b) \\
\iff & \quad \{ \text{ definition of } \mathit{const} \ \} \\
& \varphi\ a = \varphi\ a \qquad \square
\end{aligned}
$$

Using parametricity, we can also show that the two definitions of addition (and the two definitions of multiplication) are equivalent. The proofs are left as instructive exercises to the reader.

5 Final remarks

Church numerals are not just an intellectual curiosity. They gain practical importance through their relationship to lists, the functional programmer's favourite data structure. It is well-known that representations of the natural numbers serve admirably as templates for list implementations (Okasaki, 1998). The vanilla list type, for instance, is based on the unary representation of the natural numbers.

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Nat} && = \ \mathit{Zero} \mid \mathit{Succ}\ \mathit{Nat} \\
&\mathbf{data}\ \mathit{List}\ a && = \ \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)
\end{aligned}
$$

The encoding of *Nat* using a polymorphic type is an instance of a general scheme for representing data types in System F discovered independently by Leivant (1983) and Böhm & Berarducci (1985). If we apply the encoding to *List*, we obtain the continuation- or context-passing implementation of lists also known as the *backtracking monad* (Hughes, 1995; Hinze, 2001).

$$
\begin{aligned}
&\mathbf{type}\ \mathit{Church} && = \ \forall X . (X \to X) \to X \to X \\
&\mathbf{type}\ \mathit{Backtr}\ A && = \ \forall X . (A \to X \to X) \to X \to X
\end{aligned}
$$

The type *Backtr* has been reinvented quite a few times. It appears, for instance, in a paper about *deforestation* (Gill *et al.*, 1993). The central theorem of the paper, *foldr-build fusion*, states that

$$
\mathit{foldr}\ \mathit{cons}\ \mathit{nil}\ (\mathit{build}\ g) \ = \ g\ \mathit{cons}\ \mathit{nil}, \tag{7}
$$

where *foldr* is the fold operator for lists and *build* is given by

$$
\begin{aligned}
&\mathit{build} && : \ (\forall X . (A \to X \to X) \to X \to X) \to \mathit{List}\ A \\
&\mathit{build}\ g && = \ g\ \mathit{Cons}\ \mathit{Nil}\ .
\end{aligned}
$$

Setting $\mathit{backtr}\ x = \lambda \mathit{cons}\ \mathit{nil} . \mathit{foldr}\ \mathit{cons}\ \mathit{nil}\ x$ we can rewrite (7) as $\mathit{backtr} \cdot \mathit{build} = \mathit{id}$. In other words, the fusion theorem is a direct consequence of the fact that *List A* and *Backtr A* are isomorphic. Unsurprisingly, the fusion theorem can be generalized to arbitrary data types, as well (Takano & Meijer, 1995).

The second derivation of the Church numerals started from an algebraic specification of the natural numbers. Does this transfer to lists, as well? The answer is an emphatic "Yes!". The algebraic structure of the list type is that of a *monad* with zero and plus (Moggi, 1991; Wadler, 1990). Using the monad laws as a starting point the derivation goes through equally well, see Hughes (1986) and Hinze (2000). Interestingly, if we confine ourselves to the additive fragment (0 and '+'), then we obtain Hughes's efficient sequence type (Hughes, 1986) – compare Hughes's implementation to the definition of $\underline{nat}_1$ in section 3. As an aside, exponentiation, in particular, Rosser's definition of '$\uparrow$' has no counterpart in the world of lists (the reverse application of two lists is not even typeable).

Apropos efficiency. Though inductive types and their encodings are isomorphic, they are not equivalent in terms of efficiency. Rosser's addition and multiplication, for instance, are constant time operations while the implementations based on folds take time linear in the size of their first argument (the same holds for the list operations). Conversely, projection functions such as predecessor (or *head* and *tail* in the case of lists) are constant time operations for inductive types while they take linear time for the polymorphic encodings.

Acknowledgements

I am grateful to Patricia Johann and to the anonymous referees for pointing out several typos and for valuable suggestions regarding presentation.

References

Barendregt, H. (1992) Lambda calculi with types. In: Abramsky, S., Gabbay, D. M. and Maibaum, T. (eds.), *Handbook of Logic in Computer Science, Volume 2, Background: Computational Structures*, pp. 118–309. Clarendon Press, Oxford.

Bird, R. and de Moor, O. (1997) *Algebra of Programming*. Prentice Hall Europe.

Böhm, C. and Berarducci, A. (1985) Automatic synthesis of typed λ-programs on term algebras. *Theor. Comput. Sci.* **39**(2–3), 135–154.

Church, A. (1941) *The Calculi of Lambda-Conversion*. Annals of Mathematics Studies No. 6, Princeton University Press.

Dershowitz, N. and Jouannaud, J.-P. (1990) Rewrite systems. In: van Leeuwen, J. (ed.), *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pp. 243–320. Elsevier.

Gill, A., Launchbury, J. and Peyton Jones, S. L. (1993) A short cut to deforestation. *FPCA '93: The Sixth International Conference on Functional Programming Languages and Computer Architecture*, pp. 223–232. ACM Press.

Girard, J.-Y. (1972) *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII.

Hinze, R. (2000) Deriving backtracking monad transformers. In: Wadler, P. (ed.), *Proceedings 2000 International Conference on Functional Programming*, pp. 186–197. Montreal, Canada.

Hinze, R. (2001) Prolog's control constructs in a functional setting — Axioms and implementation. *Int. J. Foundations of Comput. Sci.* **12**(2), 125–170.

Hughes, J. (1995) The design of a pretty-printing library. In: Jeuring, J. and Meijer, E. (eds.), *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques: Lecture Notes in Computer Science 925*, pp. 53–96. Båstad, Sweden. Springer-Verlag.

Hughes, R. J. M. (1986) A novel representation of lists and its application to the function "reverse". *Infor. Process. Lett.* **22**(3), 141–144.

Hutton, G. (1999) A tutorial on the universality and expressiveness of fold. *J. Funct. Program.* **9**(4), 355–372.

Leivant, D. (1983) Reasoning about functional programs and complexity classes associated with type disciplines. *Proceedings 24th Annual IEEE Symposium on Foundations of Computer Science, FOCS'83*, pp. 460–469. Tucson, AZ. IEEE Press.

Mendler, N. P. (1991) Inductive types and type constraints in the second-order lambda calculus. *Ann. Pure & Appl. Logic*, **51**(1–2), 159–172.

Moggi, E. (1991) Notions of computation and monads. *Infor. & Computation* **93**(1), 55–92.

Okasaki, C. (1998) *Purely Functional Data Structures.* Cambridge University Press.

Parigot, M. (1992) Recursive programming with proofs. *Theor. Comput. Sci.* **94**(2), 335–356.

Reynolds, J. (1974) Towards a theory of type structure. *Proceedings, Colloque sur la Programmation: Lecture Notes in Computer Science 19*, pp. 408–425. Springer-Verlag.

Takano, A. and Meijer, E. (1995) Shortcut deforestation in calculational form. *Proceedings Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pp. 306–313. La Jolla, San Diego, CA. ACM Press.

Wadler, P. (1989) Theorems for free! *Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*, pp. 347–359. London, UK. Addison-Wesley.

Wadler, P. (1990) Comprehending monads. *Proceedings 1990 ACM Conference on LISP and Functional Programming*, pp. 61–78. Nice, France. ACM Press.