J. Functional Programming **10** (6): 607–623, November 2000. Printed in the United Kingdom
© 2000 Cambridge University Press

THEORETICAL PEARLS

An adequate and efficient left-associated binary numeral system in the λ-calculus

MAYER GOLDBERG*
Department of Computer Science, Ben Gurion University, Beer Sheva 84105, Israel
(*e-mail:* `gmayer@cs.bgu.ac.il`)

Abstract

This paper introduces a sequence of λ-expressions modeling the binary expansion of integers. We derive expressions computing the test for zero, the successor function, and the predecessor function, thereby showing the sequence to be an adequate numeral system, i.e. one in which all recursive functions are lambda-definable. These functions can be computed efficiently; To this end, we introduce a notion of complexity that is independent of the order of evaluation.

1 Introduction

1.1 Numeral systems in the λ-calculus

Numbers are traditionally represented on computers with a size proportional to their logarithm. Traditional numeral systems in the λ-calculus, such as Church numerals (Barendregt, 1984; Church, 1941) and Barendregt numerals (Barendregt, 1984), however, typically involve linear representations of numbers. In such systems, the size of the representation of a number n is proportional to this number.

In this paper, we present an *adequate* binary numeral system for the λ-calculus, and define λ-terms that compute the zero predicate and the successor and predecessor functions efficiently. The notion of efficient computation in the λ-calculus is subtle because different reduction strategies result in different complexities. We avoid this problem by requiring the complexity of our computation to be independent of any order of evaluation. This requirement implies that we could not, for example, make use of fixed-point combinators to define our λ-expressions.

The particular representation used in this paper is due to den Hoed (1980). The problem of showing that efficient number-theoretic functions are definable for this

* This work was carried out while visiting BRICS (Basic Research in Computer Science, Centre of the Danish National Research Foundation, URL: `http://www.brics.dk/`.

system was given as a challenge to the author by Barendregt during a visit to Indiana University in 1990 (Barendregt, 1990).

1.2 Prerequisites and notation

We assume some familiarity with the λ-calculus (Barendregt, 1984; Church, 1941). The *identity* combinator is given by $\mathbf{I} = \lambda x.x$. The boolean values *true* and *false* are denoted by $\mathbf{T} = \lambda xy.x$ and $\mathbf{F} = \lambda xy.y$ respectively. *Conjunction* is denoted by **and** $= (\lambda xy.x(y\mathbf{TF})\mathbf{F})$. *Selectors* are given by $\mathbf{U}_k^n = \lambda x_0 \cdots x_n.x_k$ where $k \leqslant n$. The ordered n-tuple $\langle x_1, \ldots, x_n\rangle$ is denoted by $[x_1, \cdots, x_n] = \lambda s.(sx_1 \cdots x_n)$. The k-th projection of an ordered n-tuple is denoted by $\pi_k^n = \lambda x.(x\mathbf{U}_{k-1}^{n-1})$. The *length* of a λ-term M is the number of symbols it occupies, and is noted as $\|M\|$. Finally, the reflexive, transitive closure of the one-step reduction $\longrightarrow$ is given by $\twoheadrightarrow$. A numeral system is *adequate* if all recursive functions are λ-definable for it.

2 Binary numerals

2.1 Representation

Since various data structures can be implemented in the λ-calculus, we could select any one of several different binary representations for our numerals. We choose to use, however, a representation that is unique to the λ-calculus:

Definition 1 (den Hoed)
The sequence $\mathsf{bin} = \{\mathsf{bin}_n\}_{n\in\omega}$. We define bin_n as follows: let the variable z (pronounced: 'zero') represent a 0-bit, and let the variable w (pronounced: 'wan') represent a 1-bit. Let $b_1b_2\cdots b_k$, $b_j \in \{z, w\}$, be a sequence of bits corresponding to the binary expansion of n, such that b_1 and b_k are the low and the high bits, respectively. Then

$$\mathsf{bin}_n = \lambda zw.b_1 \cdots b_k$$

The sequence of bits is thus represented by a left-associated application of z's and w's.

Examples

$$\begin{array}{lll@{\qquad}lll}
\mathsf{bin}_0 & = & \lambda zw.z & \mathsf{bin}_4 & = & \lambda zw.zzw \\
\mathsf{bin}_1 & = & \lambda zw.w & \mathsf{bin}_5 & = & \lambda zw.wzw \\
\mathsf{bin}_2 & = & \lambda zw.zw & \mathsf{bin}_6 & = & \lambda zw.zww \\
\mathsf{bin}_3 & = & \lambda zw.ww & \mathsf{bin}_7 & = & \lambda zw.www
\end{array}$$

Our goal in this paper is to show that the sequence bin is an adequate numeral system, and that the successor function, the predecessor function, and the test for zero can all be computed on the bits directly, without expanding their argument into some linear representation. In our PhD thesis (Goldberg, 1996), we show similarly that addition, subtraction, multiplication, quotient, remainder, and the test for equality can also be computed on the bits directly.

2.2 Uniqueness of representation

One problem that affects all n-ary numeral systems is uniqueness: For example, in our system, $\lambda zw.w$, $\lambda zw.wzz$, and $\lambda zw.wzzzzz$ all represent the number 1. In the λ-calculus, however, it is more elegant for two numerals representing the same number to have the same normal form.

We thus propose the following two-fold compromise:

- We define a test for zero (and ultimately, the test for equality) that ignores trailing zero bits.
- We define the predecessor function (and ultimately, addition, subtraction, multiplication, quotient, remainder, etc.) not to leave trailing zero bits.

Thus, the functions we provide do not introduce trailing zeros in their results, and ignore them in their arguments. Another solution, which is simpler to derive and to verify, would be to define a ‘normalization’ combinator, taking a binary numeral and removing its trailing zero bits. This solution, however, is less efficient.

2.3 Size of our representation

The size of bin_n, our representation of n, is proportional to the number of bits in the binary expansion of n, i.e. to $\log n$. It is also clear that bin numerals are as concise (in the sense of having the least number of symbols) as possible for a binary numeral system in the λ-calculus.

What is not as obvious, but just as important if bin is to be practical for implementation on a computer, is whether the various arithmetic operations that we might want to carry out on this representation can be computed directly on the bits, without expanding our binary representation to a less compact one. We do not have the convenience, for example, of switching to and from one of the well-known, linear numeral systems to define arithmetic functions in one system in terms of the other system, as Barendregt does in Lemma 6.4.5 and Corollary 6.4.6 (p. 140) of his textbook on the λ-calculus (Barendregt, 1984). We want to avoid both explicit expansion, as well as expansion, that is implicit in a particular reduction sequence.

The following definitions let us express formally just how much can a given expression ‘expand’:

Definition 2

(i) *Finitely wide terms.* A λ-term M is *finitely wide* if there exists a number $N > 0$, such that if for all λ-terms x, if $M \twoheadrightarrow_R x$ then $\|x\| \leqslant N$.

(ii) *The width of a term*, Width. The *width* of a finitely wide term M, denoted by $\mathsf{Width}(M)$ is given by

$$\mathsf{Width}(M) \quad = \quad \sup\{\|x\| : M \longrightarrow\!\!\!\!\!\rightarrow x\}$$

The following points should be noted:

- Some λ-terms do not have a finite width, but have a normal form. For example, let M be defined as follows:

$$\begin{array}{rcl} M & = & ((\underline{\lambda f.((\lambda x.f(xx))} \\ & & \quad\quad \underline{(\lambda x.f(xx)))}) \\ & & (\lambda xy.y) \\ & & \mathbf{I}) \end{array}$$

  It is simple to verify that $M \twoheadrightarrow \mathbf{I}$. The underlined sub-expression, however, does not have a normal form, and expands arbitrarily. We can thus have reduction sequences that result in expressions of arbitrary width. Therefore, M does not have a finite width.
- Some λ-terms do not have a normal form, but have a finite width. For example, let M be defined as follows:

$$M \quad = \quad ((\lambda x.xx)\ (\lambda x.xx))$$

  It is simple to verify that the only possible reduction from M is to itself, and so M has a finite width, but no normal form.

The width of a λ-term is used in proving that a test for zero, the successor function, and the predecessor function can all be computed without expanding the representation of their arguments beyond $\log n$.

2.4 *Complexity*

We use the notion of *finite width* developed in section 2.3. to show that the expression we construct for computing the successor and predecessor functions on binary numerals do so (i.e. reduce to normal form) with space complexity proportional to the number of bits. A notion similar to that of finite width could be introduced in order to show that the time complexity for computing these functions is also proportional to the number of bits.

3 Mealy machines

The constructions for the expressions that compute the successor and predecessor functions on binary numerals are quite involved. To simplify their presentation, we precede each construction with a corresponding *Mealy machine* that provides an abstract description of the algorithm we use. A Mealy machine is a finite state automaton that both reads in an input symbol and writes out an output symbol at each non-ϵ transition. A Mealy machine, mapping an input stream of binary digits to an output stream of binary digits can be used to depict the algorithm by which an input stream of binary digits which denotes a binary number, gets mapped to an output stream of binary digits that denotes a binary number, and the algorithms for computing the successor and predecessor on binary strings are just simple enough to be depicted by such a state machine. The terms $\mathbf{Succ}_{\mathsf{bin}}$ and $\mathbf{Pred}_{\mathsf{bin}}$, which respectively compute the successor and predecessor functions on binary numerals, are implementations of the corresponding Mealy machines.

The following depiction of a Mealy machine denotes that there is a transition from state A to state B in which the symbol a is consumed and the symbol b is the output:

$$\begin{tikzcd} A \arrow[r, "a/b"] & B \end{tikzcd}$$

4 Decision logic tables

In this paper we use Decision Logic Tables (DLT) in deriving expressions for the successor and predecessor functions on bin.

A DLT (Kavanagh, 1960; McDaniel, 1968) is a tabular form for describing a program segment driven by an n-variable boolean function. The format of a decision logic table is as follows:

list of variable names [or boolean conditions]	list of all possible combinations of values of variables [or values of boolean conditions]
list of actions to be taken at a given combination of values	selections of combinations of actions as a function of combinations of variables

In some situations, not all relevant boolean conditions can be considered in parallel. For example, given two variables a and b, the test of whether b is equal to zero should precede the division of a by b, and therefore any test on the quotient of a and b. Such situations have traditionally been handled by *nesting* or *dispatching to* other decision logic tables as one of the actions.

The following example is used to illustrate the use of a decision logic table. Consider the following highly simplified process of evaluating a paper for publication. A paper can be either accepted or rejected, and the author can be requested to make revisions to the paper before it can appear in print. Deciding what to do with the paper depends upon the answers to the following three questions: (a) Is the material in the paper correct? (b) Is the main result of the paper of interest? (c) Is the paper written clearly? The following decision logic table associates combinations of answers to these questions with combinations of actions to be taken:

Note that when the results in the paper are both correct and interesting, but not clearly written, a combination of two actions takes place: The paper is accepted for publication, and the author is asked to revise the paper.

The Highly Simplified Process of Evaluating a Paper for Publication

Is the paper correct?	Yes	Yes	Yes	Yes	No	No	No	No
Is the result interesting?	Yes	Yes	No	No	Yes	Yes	No	No
Is the paper clear?	Yes	No	Yes	No	Yes	No	Yes	No
Reject the paper			√	√	√	√	√	√
Accept the paper	√	√						
Ask the author to revise		√						

Decision logic tables can be formally manipulated and simplified, as well as automatically compiled into computer programs. Since they are not in common use today, we shall avoid the traditional decision logic table abbreviations, in order to preserve clarity.

5 Arithmetic functions

5.1 *Testing for zero*

Proposition 1
There exists a combinator $\mathbf{Zero?}_{\mathsf{bin}}$ such that for all $n \in \mathbb{N}$ we have

(i) $(\mathbf{Zero?}_{\mathsf{bin}}\ \mathsf{bin}_0) \twoheadrightarrow \mathbf{T}$
$(\mathbf{Zero?}_{\mathsf{bin}}\ \mathsf{bin}_{n+1}) \twoheadrightarrow \mathbf{F}$
(ii) $\mathsf{Width}(\mathbf{Zero?}_{\mathsf{bin}}\ \mathsf{bin}_n) = O(\log n)$.

Proof

(i) Binary numerals are of the form $\lambda zw.b_1 \cdots b_n$, where $b_k \in \{z, w\}$. Computing the zero predicate amounts to deciding whether $w \notin \{b_1, \ldots, b_n\}$. To this end, we apply the binary numerals to two λ-terms $D_{\mathrm{T}}, D_{\mathrm{F}}$ (indexed by the boolean values *true* and *false*), thus substituting them for z, w in the application $(b_1 \cdots b_n)$. We embed the computable applicative structure (Mitchell, 1996) $\langle \mathsf{Bool}, \wedge \rangle$ in the term model of $\{b_1, b_2, \ldots, b_n, (b_1 b_2), (b_1 b_2 b_3), \ldots, (b_1 \cdots b_n)\}$, such that

$$(D_a D_b) = D_{a \wedge b}$$

where $\wedge$ denotes Boolean conjunction. Given this substitution, the application $(b_1 \cdots b_n)$ reduces to either D_{T} or D_{F}. If the resulting expression is D_{T}, then all the b_k's are z's and the binary numeral is zero.
We make use of the following property of the application of two ordered pairs (compare with Barendregt's hint in his Problem 6.8.15(ii) Page 149):

$$\begin{array}{rcl}([a_1,b_1]\ [a_2,b_2]) & \longrightarrow & ((\lambda x.xa_1b_1)\ (\lambda x.xa_2b_2)) \\ & \longrightarrow & ((\lambda x.xa_2b_2)a_1b_1) \\ & \longrightarrow & (a_1a_2b_2b_1)\end{array}$$

In particular, we have:

$$([M,b_1]\ [M,b_2]) \quad = \quad (MMb_2b_1)$$

We define M as follows:

$$M \quad = \quad \lambda mb_2b_1.[m,(\mathbf{and}\ b_1\ b_2)]$$

By pairing M with $\mathbf{F}$ and $\mathbf{T}$ we obtain D_{F} and D_{T} respectively:

$$\begin{array}{rcl}D_{\mathrm{F}} & = & [M,\mathbf{F}] \\ D_{\mathrm{T}} & = & [M,\mathbf{T}]\end{array}$$

We now have

$$\begin{array}{rclcrcl}(D_{\mathrm{F}}\ D_{\mathrm{F}}) & \twoheadrightarrow & D_{\mathrm{F}} & \qquad & (D_{\mathrm{T}}\ D_{\mathrm{F}}) & \twoheadrightarrow & D_{\mathrm{F}} \\ (D_{\mathrm{F}}\ D_{\mathrm{T}}) & \twoheadrightarrow & D_{\mathrm{F}} & & (D_{\mathrm{T}}\ D_{\mathrm{T}}) & \twoheadrightarrow & D_{\mathrm{T}}\end{array}$$

To obtain the result of the test for zero, we only need to take the second projection. We thus define the test for zero as follows:

$$\mathbf{Zero?}_{\mathsf{bin}} \quad = \quad \lambda n.(\pi_2^2\ (n\ D_{\mathrm{T}}\ D_{\mathrm{F}}))$$

Note that as a byproduct of our construction, this definition of $\mathbf{Zero?}_{\mathsf{bin}}$ ignores trailing zeros, for example:

$$\begin{array}{rcl}(\mathbf{Zero?}_{\mathsf{bin}}\ (\lambda zw.zwzzz)) & \twoheadrightarrow & \mathbf{F} \\ (\mathbf{Zero?}_{\mathsf{bin}}\ (\lambda zw.zzzzz)) & \twoheadrightarrow & \mathbf{T}\end{array}$$

(ii) Let

$$\begin{array}{rcl}C & = & \mathsf{Width}(\mathbf{Zero?}_{\mathsf{bin}}) + \max\{\mathsf{Width}(\pi_2^2([M,b])) : b \in \{\mathbf{F},\mathbf{T}\}\} \\ r & = & \max\{\mathsf{Width}([M,b_1]\ [M,b_2]) : b_1,b_2 \in \{\mathbf{F},\mathbf{T}\}\}\end{array}$$

For any $n \in \mathbb{N}, \mathsf{bin}_n = \lambda zw.b_1 \cdots b_k$, we have:

$$\begin{array}{rcl}\mathsf{Width}(\mathbf{Zero?}_{\mathsf{bin}}\ \mathsf{bin}_n) & \leqslant & C + \mathsf{Width}(\mathsf{bin}_n) + k \cdot r \\ & = & O(k) \\ & = & O(\log n)\end{array}$$

□

5.2 *The successor function*

Proposition 2

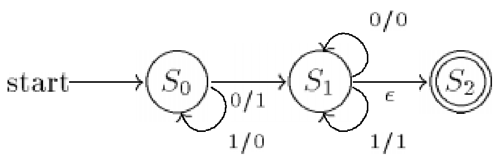
(i) There exists a combinator $\mathbf{Succ}_{\mathsf{bin}}$, such that for all $n \in \mathbb{N}$ we have

$$(\mathbf{Succ}_{\mathsf{bin}}\ \mathsf{bin}_n) \twoheadrightarrow \mathsf{bin}_{n+1}.$$

(ii) $\mathsf{Width}(\mathbf{Succ}_{\mathsf{bin}}\ \mathsf{bin}_n) \ = \ O(\log n)$.

Proof

(i) We describe the successor function on bin_n using a three-state Mealy machine: The first state, S_0, propagates the carry; The second state, S_1, goes through the remaining bits after the carry operation has been performed; The third state, S_2, is the final state.
The machine is depicted in the following diagram:



Let $\mathsf{bin}_n = \lambda zw.b_1 \cdots b_k$, where $b_j \in \{z, w\}$. We apply bin_n to *three* terms $M_0, M_1, M_{\mathsf{End}}$, substituting M_0, M_1, respectively, for z, w in the body of the binary numeral, giving the application $(M_{\delta_1} \cdots M_{\delta_k} M_{\mathsf{End}})$, where $\delta_k \in \{0, 1\}$, for $j \leqslant k$, and where M_{End} is a marker for the end of the number. Let $\mathsf{T} = \{M_{\delta_j}\}_{j \leqslant k}$. We embed the computable applicative structure $\langle \mathsf{Bool}, f \rangle$, where f is given by

$$\begin{aligned}
f(S_0, 0r, w) &= f(S_1, r, w1)\\
f(S_0, 1r, w) &= f(S_0, r, w0)\\
f(S_0, \mathsf{End}, w) &= w\\
f(S_1, 0r, w) &= f(S_1, r, w0)\\
f(S_1, 1r, w) &= f(S_1, r, w1)\\
f(S_1, \mathsf{End}, w) &= w
\end{aligned}$$

in the term model of $T \cup \{MN : M, N \in \mathsf{T}\}$. Given this substitution, we can implement the Mealy machine described above, thus computing bin_{n+1}. Note that M_γ is not used in the applicative structure; It's rôle, as will be explained later, is to stop the state machine return the body of bin_{n+1} (essentially implementing the state S_2). The remainder of the proof describes the implementation details.
In moving from state to state, the reconstruction of the partial body of bin_{n+1} is carried along and maintained together with some additional information. Each expression $M_\delta \in \mathsf{T}$ needs to have access to

- An encoding σ of the current state (i.e. of either S_0 or S_1).
- An encoding of whether the given expression is substituted for a 0-bit or a 1-bit, or is a mark for the end of the stream of bits (noted by ϵ in the diagram). This is denoted by b.
- A partial reconstruction of the body of the successive numeral. This is denoted by r.

The values of σ and b determine the value of the given expression. Any finite set of λ-expressions, for which we have a test of equality could therefore be used for encoding (1) and (2). Furthermore, since the encodings in (1) and (2) serve only as tags upon which to dispatch, we can eliminate the test altogether by using *selectors*, i.e. expressions of the form

$$\mathbf{U}_k^n \quad = \quad \lambda x_0 \cdots x_n . x_k$$

to encode the various choices. The function we represent by our applicative structure is thus a function of arity 3. The arguments to the function parameterize the terms in T, and are referred to as the information content of M. We store this information, and a procedure m in an ordered 4-tuple. Again, observe that:

$$\begin{aligned}
&([m, b_1, r_1, \sigma_1]\,[m, b_2, r_2, \sigma_2]) \\
&\quad \longrightarrow \ ((\lambda x.xmb_1r_1\sigma_1)\ (\lambda x.xmb_2r_2\sigma_2)) \\
&\quad \longrightarrow \ ((\lambda x.xmb_2r_2\sigma_2)\ mb_1r_1\sigma_1) \\
&\quad \longrightarrow \ (mmb_2r_2\sigma_2b_1r_1\sigma_1)
\end{aligned}$$

As one can see, m is passed a copy of itself, as well as all the information stored in both ordered 4-tuples (both 4-tuples have m is common). On the basis of the information it is passed, m can return the body of the successive numeral or it can construct a new ordered 4-tuple, in which case the computation continues.

Since the function represented by our applicative structure takes three arguments, and is quite complicated, we use three *Decision Logic Tables* (cf. section 4) to describe this behavior in a concise manner.

The main decision logic table in our proof distinguishes between the different states in the automaton. A separate decision logic table is provided for each state, with the exception of the *final* state (which does nothing). The three decision logic tables are given below:

Main DLT: *Determining State*

Value of σ_1	$\mathbf{U}_0^1$	$\mathbf{U}_1^1$
Dispatch to the DLT of S_0	$\surd$	
Dispatch to the DLT of S_1		$\surd$

The DLT at S_0

Value of b_1	$\mathbf{U}^2_0$	$\mathbf{U}^2_0$	$\mathbf{U}^2_0$	$\mathbf{U}^2_1$	$\mathbf{U}^2_1$	$\mathbf{U}^2_1$	$\mathbf{U}^2_2$	$\mathbf{U}^2_2$	$\mathbf{U}^2_2$
Value of b_2	$\mathbf{U}^2_0$	$\mathbf{U}^2_1$	$\mathbf{U}^2_2$	$\mathbf{U}^2_0$	$\mathbf{U}^2_1$	$\mathbf{U}^2_2$	$\mathbf{U}^2_0$	$\mathbf{U}^2_1$	$\mathbf{U}^2_2$
$[m, b_2, (r_1 w), \mathbf{U}^1_1]$	√	√							
return with $(r_1 w)$			√						
$[m, b_2, (r_1 z), \mathbf{U}^1_0]$				√	√				
return with $(r_1 z w)$						√			
irrelevant							√	√	√

The DLT at S_1

Value of b_1	$\mathbf{U}^2_0$	$\mathbf{U}^2_0$	$\mathbf{U}^2_0$	$\mathbf{U}^2_1$	$\mathbf{U}^2_1$	$\mathbf{U}^2_1$	$\mathbf{U}^2_2$	$\mathbf{U}^2_2$	$\mathbf{U}^2_2$
Value of b_2	$\mathbf{U}^2_0$	$\mathbf{U}^2_1$	$\mathbf{U}^2_2$	$\mathbf{U}^2_0$	$\mathbf{U}^2_1$	$\mathbf{U}^2_2$	$\mathbf{U}^2_0$	$\mathbf{U}^2_1$	$\mathbf{U}^2_2$
$[m, b_2, (r_1 z), \mathbf{U}^1_1]$	√	√							
return with $(r_1 z)$			√						
$[m, b_2, (r_1 w), \mathbf{U}^1_1]$				√	√				
return with $(r_1 w)$						√			
irrelevant							√	√	√

The actions to be taken at each state are a function of b_1 and b_2. In both states, the computation of the body of bin_{n+1} terminates when $b_2 = \mathbf{U}^2_2$. Also, the situation where $b_1 = \mathbf{U}^2_2$ cannot occur (since for all n, bin_n abstracts over at least one bit), and so the return value in such a situation is irrelevant; We could return any value whatsoever, so we arbitrarily pick the **I** combinator.

The decision logic tables for the states S_1 and S_2 specify different actions to be taken upon different possible values of b_1 and b_2. In general, we would require a selection mechanism of the form:

$$\begin{array}{l} \mathsf{Case}\ b_i \\ \quad \mathsf{Tag}_1 \implies \mathsf{Action}_1 \\ \quad \mathsf{Tag}_2 \implies \mathsf{Action}_2 \\ \quad \mathsf{Tag}_3 \implies \mathsf{Action}_3 \\ \mathsf{Esac} \end{array}$$

However, since we are using *selectors* for tags, i.e. expressions of the form $\mathbf{U}^k_r$, for $0 \leqslant r \leqslant k \leqslant 2$, we can use the following for our selection mechanism:

$$(b_i\ \mathsf{Action}_1\ \mathsf{Action}_2\ \mathsf{Action}_3)$$

All three decision logic tables are combined in M:

$$\begin{array}{rcl} M & = & \lambda m b_2 r_2 \sigma_2 b_1 r_1 \sigma_1.(\sigma_1(b_1(b_2[m, b_2, (r_1 w), \mathbf{U}^1_1] \\ & & \qquad\qquad\qquad\qquad [m, b_2, (r_1 w), \mathbf{U}^1_1] \\ & & \qquad\qquad\qquad\qquad (r_1 w)) \\ & & \qquad\qquad\qquad (b_2[m, b_2, (r_1 z), \mathbf{U}^1_0] \\ & & \qquad\qquad\qquad\qquad [m, b_2, (r_1 z), \mathbf{U}^1_0] \\ & & \qquad\qquad\qquad\qquad (r_1 z w)) \\ & & \qquad\qquad\qquad \mathbf{I}) \\ & & \qquad\qquad (b_1(b_2[m, b_2, (r_1 z), \mathbf{U}^1_1] \\ & & \qquad\qquad\qquad\qquad [m, b_2, (r_1 z), \mathbf{U}^1_1] \\ & & \qquad\qquad\qquad\qquad (r_1 z)) \\ & & \qquad\qquad\qquad (b_2[m, b_2, (r_1 w), \mathbf{U}^1_1] \\ & & \qquad\qquad\qquad\qquad [m, b_2, (r_1 w), \mathbf{U}^1_1] \\ & & \qquad\qquad\qquad\qquad (r_1 w)) \\ & & \qquad\qquad\qquad \mathbf{I})) \end{array}$$

We now define the successor function in terms of M as follows:

$$\begin{array}{rcl} \mathbf{Succ}_{\mathsf{bin}} & = & \lambda n z w.(n[M, \mathbf{U}^2_0, \mathbf{I}, \mathbf{U}^1_0] \\ & & \qquad\quad [M, \mathbf{U}^2_1, \mathbf{I}, \mathbf{U}^1_0] \\ & & \qquad\quad [M, \mathbf{U}^2_2, \mathbf{I}, \mathbf{U}^1_0]) \end{array}$$

(ii) The proof is similar to the proof of Proposition 1, albeit more tedious. It can be found in our PhD thesis (Goldberg, 1996).

□

5.3 *The predecessor function*

Proposition 3

(i) There exists a combinator $\mathbf{Pred}_{\mathsf{bin}}$ such that for all $n \in \mathbb{N}$ we have

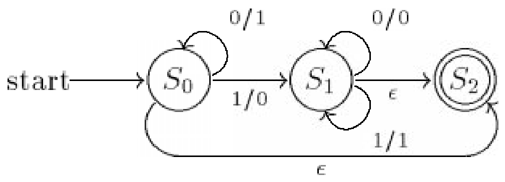
$$(\mathbf{Pred}_{\mathsf{bin}}\ \mathsf{bin}_{n+1}) \twoheadrightarrow \mathsf{bin}_n.$$

(ii) $\mathsf{Width}(\mathbf{Pred}_{\mathsf{bin}}\ \mathsf{bin}_n) = O(\log n)$.

Proof

(i) We describe the predecessor function on bin_n using a three-state Mealy machine: The first state, S_0, propagates the carry; the second state, S_1, goes through the remaining bits after the carry operation has been performed; the third state, S_2, is the final state.

The machine is depicted in the following diagram:



Let $\mathsf{bin}_{n+1} = \lambda zw.b_1 \cdots b_k$, where $b_j \in \{z, w\}$. We apply bin_{n+1} to *three* terms $M_0, M_1, M_{\mathsf{End}}$, substituting M_0, M_1 for z, w, respectively, in the body of the binary numeral, giving the application $(M_{\delta_1} \cdots M_{\delta_k} M_{\mathsf{End}})$, where $\delta_k \in \{0, 1\}$, for $j \leqslant k$, and where M_{End} is a marker for the end of the number. Let $\mathsf{T} = \{M_{\delta_j}\}_{j \leqslant k}$. We embed the computable applicative structure $\langle \mathsf{Bool}, f \rangle$, where f is given by

$$\begin{array}{rcl}
f(S_0, 0r, w) & = & f(S_0, r, w1) \\
f(S_0, 1r, w) & = & f(S_1, r, w0) \\
f(S_0, \mathsf{End}, w) & = & w \\
f(S_1, 0r, w) & = & f(S_1, r, w0) \\
f(S_1, 1r, w) & = & f(S_1, r, w1) \\
f(S_1, \mathsf{End}, w) & = & w
\end{array}$$

in the term model of $T \cup \{MN : M, N \in \mathsf{T}\}$. Given this substitution, we can implement the Mealy machine described above, thus computing bin_n. As in the derivation of $\mathbf{Succ}_{\mathsf{bin}}$, here too M_{End} is not used in the applicative structure itself, and is really a device for stopping the state machine and returning the body of bin_n (essentially implementing the state S_2). The remainder of the proof describes the implementation details.

In moving from state to state, the reconstruction of the partial body of bin_n (the preceding numeral) will need to be carried along and maintained together with some additional information. Therefore, each expression needs to have access to

- An encoding σ of the current state (i.e. of either S_0 or S_1).
- An encoding of whether the given expression is substituted for a 0-bit, a 1-bit, or is a mark for the end of the stream of bits. This is denoted by b.
- A partial reconstruction of the body of the preceding numeral, under the assumption that additional z's in the number are trailing, and should be ignored. This reconstruction is denoted by r_1.

- A partial reconstruction of the body of the preceding numeral, under the assumption that additional z's in the number are *not* trailing, and should not be dropped. This reconstruction is denoted by r_2.

The values of (1) and (2) are the same as the corresponding ones in the construction of the successor. Since the predecessor of a bin numeral may have one less bit, we generate two reconstructions of the numeral, in parallel, and commit to one of the two when either a 1-bit or the terminal mark is encountered. Together, (3) and (4) correspond to (3) in the construction of the successor. We store this information, as well as a procedure m, in an ordered 5-tuple. The function we represent by our applicative structure is thus a function of arity 4. The arguments to the function parameterize the terms in T, and are referred to as the information content of M. We store this information, and a procedure m in an ordered 5-tuple. As usual by now, observe that:

$$\begin{aligned}
&([m, b_1, r_{11}, r_{12}, \sigma_1]\ [m, b_2, r_{21}, r_{22}, \sigma_2])\\
&\quad\longrightarrow ((\lambda x.xmb_1r_{11}r_{12}\sigma_1)\\
&\qquad\quad (\lambda x.xmb_2r_{21}r_{22}\sigma_2))\\
&\quad\longrightarrow ((\lambda x.xmb_2r_{21}r_{22}\sigma_2)\\
&\qquad\quad mb_1r_{11}r_{12}\sigma_1)\\
&\quad\longrightarrow (mmb_2r_{21}r_{22}\sigma_2b_1r_{11}r_{12}\sigma_1)
\end{aligned}$$

As one can see, m is passed a copy of itself, and all the information that is stored in both ordered 5-tuples (again, both ordered 5-tuples have m in common).

Since the function represented by our applicative structure takes four arguments, and is quite complicated, we use three decision logic tables to represent the behavior of m:

Main DLT: *Determining State*

Value of σ_1	$\mathbf{U}_0^1$	$\mathbf{U}_1^1$
Dispatch to the DLT of σ_0	√	
Dispatch to the DLT of σ_1		√

As was the case with the derivation of the successor function, the actions to be taken at each state are a function of b_1 and b_2. In both cases, the computation of the body of bin_n terminates when $b_2 = \mathbf{U}_2^2$. Also, just as with the successor function, the situation where $b_1 = \mathbf{U}_2^2$ cannot occur, and so the return value in this case is irrelevant, and once again, we arbitrarily pick the $\mathbf{I}$ combinator.

The DLT at σ_0

Value of b_1	$\mathbf{U}_0^2$	$\mathbf{U}_0^2$	$\mathbf{U}_0^2$	$\mathbf{U}_1^2$	$\mathbf{U}_1^2$	$\mathbf{U}_1^2$	$\mathbf{U}_2^2$	$\mathbf{U}_2^2$	$\mathbf{U}_2^2$
Value of b_2	$\mathbf{U}_0^2$	$\mathbf{U}_1^2$	$\mathbf{U}_2^2$	$\mathbf{U}_0^2$	$\mathbf{U}_1^2$	$\mathbf{U}_2^2$	$\mathbf{U}_0^2$	$\mathbf{U}_1^2$	$\mathbf{U}_2^2$
$[m, b_2, (r_{12}w), (r_{12}w), \mathbf{U}_0^1]$	$\surd$	$\surd$							
return with $(r_{12}w)$			$\surd$						
$[m, b_2, r_{11}, (r_{12}z), \mathbf{U}_1^1]$				$\surd$	$\surd$				
return with r_{11}						$\surd$			
irrelevant							$\surd$	$\surd$	$\surd$

The DLT at σ_1

Value of b_1	$\mathbf{U}_0^2$	$\mathbf{U}_0^2$	$\mathbf{U}_0^2$	$\mathbf{U}_1^2$	$\mathbf{U}_1^2$	$\mathbf{U}_1^2$	$\mathbf{U}_2^2$	$\mathbf{U}_2^2$	$\mathbf{U}_2^2$
Value of b_2	$\mathbf{U}_0^2$	$\mathbf{U}_1^2$	$\mathbf{U}_2^2$	$\mathbf{U}_0^2$	$\mathbf{U}_1^2$	$\mathbf{U}_2^2$	$\mathbf{U}_0^2$	$\mathbf{U}_1^2$	$\mathbf{U}_2^2$
$[m, b_2, r_{11}, (r_{12}z), \mathbf{U}_1^1]$	$\surd$	$\surd$							
return with r_{11}			$\surd$						
$[m, b_2, (r_{12}w), (r_{12}w), \mathbf{U}_1^1]$				$\surd$	$\surd$				
return with $(r_{12}w)$						$\surd$			
irrelevant							$\surd$	$\surd$	$\surd$

The decision logic tables for the states S_1 and S_2 specify different actions to be taken upon different possible values of b_1 and b_2. Just as with the successor function, we rely on the fact that b_1 and b_2 are *selectors* in order to simplify the selection mechanism. All three decision logic tables are combined in M:

$$
\begin{array}{rcl}
M & = & \lambda m b_2 r_{21} r_{22} \sigma_2 b_1 r_{11} r_{12} \sigma_1.(\sigma_1(b_1(b_2[m, b_2, (r_{12}w), (r_{12}w), \mathbf{U}_0^1] \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\quad [m, b_2, (r_{12}w), (r_{12}w), \mathbf{U}_0^1] \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\quad (r_{12}\ w)) \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad (b_2[m, b_2, r_{11}, (r_{12}z), \mathbf{U}_1^1] \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\quad [m, b_2, r_{11}, (r_{12}z), \mathbf{U}_1^1] \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\quad r_{11}) \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \mathbf{I}) \\
 & & \qquad\qquad\qquad\qquad\qquad (b_1(b_2[m, b_2, r_{11}, (r_{11}z), \mathbf{U}_1^1] \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\quad [m, b_2, r_{11}, (r_{11}z), \mathbf{U}_1^1] \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\quad r_{11}) \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad (b_2[m, b_2, (r_{12}w), (r_{12}w), \mathbf{U}_1^1] \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\quad [m, b_2, (r_{12}w), (r_{12}w), \mathbf{U}_1^1] \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\quad (r_{12}w)) \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \mathbf{I}))
\end{array}
$$

We now define the predecessor function in terms of M as follows:

$$
\begin{array}{rcl}
\mathbf{Pred}_{\mathsf{bin}} & = & \lambda nzw.(n[M, \mathbf{U}_0^2, z, \mathbf{I}, \mathbf{U}_0^1] \\
 & & \qquad\quad [M, \mathbf{U}_1^2, z, \mathbf{I}, \mathbf{U}_0^1] \\
 & & \qquad\quad [M, \mathbf{U}_2^2, z, \mathbf{I}, \mathbf{U}_0^1])
\end{array}
$$

Recall that r_1 contains the partial reconstruction of the preceding numeral under the assumption that any additional zero bits are trailing, and can therefore be ignored. The initial value of r_1 must therefore be z, rather than $\mathbf{I}$.

(ii) The proof is similar to the proof of Proposition 1, albeit more tedious. It can be found in our PhD thesis (Goldberg, 1996).

□

5.4 Adequacy

Proposition 4

The numeral system bin is adequate.

Proof

Having defined $\mathbf{Zero?}_{\mathsf{bin}}$, $\mathbf{Succ}_{\mathsf{bin}}$, and $\mathbf{Pred}_{\mathsf{bin}}$, it follows from Proposition 6.4.3 in Barendregt's book (Barendregt, 1984) that bin is an adequate numeral system. □

6 Conclusion and assessment

This paper introduces the sequence bin, and shows that it is an adequate numeral system. This section analyzes several aspects of bin.

6.1 Extensibility

The definition of bin is easily extensible to other bases. Similarly, bin can be extended to have a sign, a decimal point and an exponent, facilitating fixed-size floating point arithmetic.

6.2 Efficiency

Numerals in bin are represented as concisely as possible. The number-theoretic functions can be computed on bin with the same complexity as they are computed on the standard binary representation used on modern computers. The complexity of this computation is independent of the order of evaluation. The use of selectors rather than arbitrary tags in the dispatching mechanism results in considerable gains in efficiency, and the resulting λ-expressions are both more concise and simpler to verify.

6.3 Implementation

The numeral system bin is suitable for implementation in functional programming languages that model the pure, untyped λ-calculus. We have implemented both the numeral system bin, and the basic number-theoretic functions defined on it in the Scheme programming language (Clinger and Rees, 1991). Our implementation can be combined with the Gödelizer developed as a part of our PhD thesis (Goldberg, 1995; Goldberg, 1996), so that such numerals, as well as possible extensions to the bin numeral system, can be displayed.

6.4 Decision logic tables

Although decision logic tables are elaborate and verbose, they are relatively straightforward to construct, and help insure correctness. Decision logic tables have traditionally been compiled into various programming languages (Humby, 1973), and so it seems reasonable to expect that λ-expressions for computing more elaborate functions could be generated automatically from a given set of decision logic tables.

Acknowledgements

This work was supported by the Danish Research Academy. I am grateful to BRICS for hosting me and for providing a stimulating environment. Thanks are also due to Olivier Danvy, Daniel P. Friedman, Julia Lawall, and Larry Moss for their comments and encouragement.

References

Barendregt, H. P. (1984) *The Lambda Calculus, Its Syntax and Semantics.* North-Holland.

Barendregt, H. P. (1990) Personal Communication, Bloomington, Indiana.

Church, A. (1941) *The Calculi of Lambda-Conversion.* Princeton University Press.

Clinger, W. and Rees, J. (editors) (1991) Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, **IV**(3), 1–55.

Goldberg, M. (1995) Gödelisation in the λ-calculus. BRICS Research Series RS-95-38, Department of Computer Science, University of Aarhus, Denmark.

Goldberg, M. (1996) *Recursive Application Survival in the λ-Calculus.* PhD thesis, Department of Computer Science, Indiana University.

Humby, E. (1973) *Programs from Decision Tables.* Macdonald/Elsevier.

Kavanagh, T. F. (1960) Tabsol – a fundamental concept for system-oriented language. *Proc. Eastern Joint Computer Conference*, pp. 117–127.

McDaniel, H. (1968) *An Introduction to Decision Logic Tables.* Wiley.

Mitchell, J. C. (1996) *Foundations of Programming Languages.* MIT Press.

van der Poel, W. L., Schaap, C. E. and van der Mey, G. (1980) New arithmetical operators in the theory of combinators. *Indagationes Mathematicae*, **42**, 271–325. Parts I-III.