BRICS RS-95-42 M. Goldberg: An Adequate Left-Associated Binary Numeral System

BRICS

Basic Research in Computer Science

An Adequate Left-Associated Binary Numeral System in the λ-Calculus

Mayer Goldberg

BRICS Report Series RS-95-42

ISSN 0909-0878 August 1995

See back inner page for a list of recent publications in the BRICS Report Series. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@daimi.aau.dk

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.aau.dk/BRICS/`
`ftp ftp.brics.aau.dk (cd pub/BRICS)`

An Adequate Left-Associated Binary Numeral System in the λ-Calculus *

Mayer Goldberg
Computer Science Department
Indiana University †
(mayer@cs.indiana.edu)

August 15, 1995

Abstract

This paper introduces a sequence of λ-expressions, modelling 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. These functions can be computed efficiently. Their complexity is independent of the order of evaluation.

Keywords: Programming calculi, λ-calculus, functional programming.

*This work was carried out while visiting BRICS (Basic Research in Computer Science, Centre of the Danish National Research Foundation).

†Bloomington, IN 47405, USA.

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 [1, 3] and Barendregt numerals [1], however, typically involve linear representations of numbers. In such systems, the size of the representation of a number n is proportional to n.

In this paper, we present an adequate binary numeral system for the λ-calculus, where the successor function, the predecessor function, and the test for zero are implemented efficiently. This implementation does not depend on the order of evaluation.

Both the particular representation used in this paper and the problem of showing that it is an adequate numeral system are due to H.P. Barendregt. They were given to the author as a challenge during a visit to Indiana University in 1990 [2].

1.2 Prerequisites and Notation

We assume some familiarity with the λ-calculus [1, 3]. 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 $\textbf{and} = (\lambda xy.(x\ (y\ \mathbf{T}\ \mathbf{F})\ \mathbf{F}))$. *Selectors* are given by $\mathsf{U}_k^n = \lambda x_0 \cdots x_n.x_k$ where $k \leq n$. The ordered n-tuple $\langle x_1, \ldots, x_n \rangle$ is denoted by $[x_1, \cdots, x_n] = \lambda s.(s\ x_1 \cdots x_n)$. The k-th projection of an ordered n-tuple is denoted by $\pi_k^n = \lambda x.(x\ \mathsf{U}_{k-1}^{n-1})$. 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 $\longrightarrow\!\!\!\!\rightarrow$.

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:

2.1.1 DEFINITION: (Barendregt) *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 $b_1 b_2 \cdots b_k$, $b_j \in \{z, w\}$, 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.

2.1.2 EXAMPLE:

$$\begin{array}{lcllcl}
\mathsf{bin}_0 & = & \lambda zw.z & \mathsf{bin}_4 & = & \lambda zw.(z\ z\ w) \\
\mathsf{bin}_1 & = & \lambda zw.w & \mathsf{bin}_5 & = & \lambda zw.(w\ z\ w) \\
\mathsf{bin}_2 & = & \lambda zw.(z\ w) & \mathsf{bin}_6 & = & \lambda zw.(z\ w\ w) \\
\mathsf{bin}_3 & = & \lambda zw.(w\ w) & \mathsf{bin}_7 & = & \lambda zw.(w\ w\ w)
\end{array}$$

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 Ph.D. thesis [6], we show similarly that addition, subtraction, multiplication, quotient, remainder, and the test for equality can also be computed on the bits directly. These extensions lay beyond the scope of this paper.

2.2 Uniqueness of Representation

One problem affects all n-ary numeral systems: A representation is not unique, that is, different representations exist for the same number. For example, in our system, $\lambda zw.w$, $\lambda zw.(w\ z\ z)$, and $\lambda zw.(w\ z\ z\ z\ z\ z)$ all represent the number 1. In the λ-calculus, however, it is extremely inelegant for two numbers to have different normal forms, and yet to be numerically equal.

We thus propose the following two-fold compromise:

- We define the test for zero (and ultimately, the test for equality) to ignore trailing zero bits.
- We define the predecessor function (and ultimately, addition, subtrac-

tion, multiplication, quotient, remainder, etc.) not to leave trailing zero bits.

So 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 "normalisation" 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 digital 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 benefit, for example, of switching to and from one of the well-known, linear numeral systems in order to define arithmetic functions in one system in terms of the other system, as Barendregt does it in Lemma 6.4.5 and Corollary 6.4.6 of his reference book on the λ-calculus [1, Page 140]. This notion of *expansion* need not be explicit, but could be implicit in a particular reduction sequence. The following definitions let us express formally just how much can a given expression "expand":

2.3.1 DEFINITION:

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 $\|x\| \leq N$.

ii. *The Width of a Term, $\mathfrak{wd}$.*[1] The *width* of a finitely wide term M, denoted by $\mathfrak{wd}(M)$ is given by

$$\mathfrak{wd} \quad = \quad \sup\{\|x\| : M \longrightarrow\!\!\!\!\!\rightarrow x\}$$

---

[1] $\mathfrak{wd}$ is *wd* in Gothic letters.

The following two points should be noted:

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

$$M \;=\; (\underline{(\lambda f.((\lambda x.(f\ (x\ x)))\ (\lambda x.(f\ (x\ x)))))}\ (\lambda xy.y)\ \mathbf{I})$$

  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:

$$M \;=\; ((\lambda x.(x\ x))\ (\lambda x.(x\ x)))$$

  It is simple to verify that $M \longrightarrow M$, and so M has a finite width, but no normal form.

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

3 Arithmetic Functions

3.1 Testing for Zero

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

i.
$$\begin{array}{rcl}(\mathbf{Zero?}_{\mathsf{bin}}\ \mathsf{bin}_0) & \twoheadrightarrow & \mathbf{T}\\ (\mathbf{Zero?}_{\mathsf{bin}}\ \mathsf{bin}_{n+1}) & \twoheadrightarrow & \mathbf{F}\end{array}$$

ii. $\mathfrak{wd}(\mathbf{Zero?}_{\mathsf{bin}}\ \mathsf{bin}_n) \;=\; O(\log n)$.

Proof:

i. To compute the zero predicate, we apply a given numeral to two λ-expressions, substituting those λ-expressions respectively for z and w, in the body of the numeral. The problem of testing for zero thus reduces to the problem of identifying whether w occurs in the body of the numeral.

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) [1, Page 149]):

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

In particular, we have:

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

We define M as follows:

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

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

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

For any $n > 0$, the binary expansion of n contains the 1-bit, and so w occurs free in the body of bin_n, thus when we substitute $D_{\mathbf{F}}$ for w in the body of bin_n, the result will be $D_{\mathbf{F}}$. To obtain the test for zero, we only need to take the second projection. We thus define the test for zero as follows:

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

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

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

ii. Let

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

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

$$\begin{aligned} \mathfrak{wd}(\mathbf{Zero?}_{\mathsf{bin}}\ \mathsf{bin}_n) &\leq C + \mathfrak{wd}(\mathsf{bin}_n) + k \cdot r \\ &= O(k) \\ &= O(\log n) \end{aligned}$$

■

3.2 The Successor Function

3.2.1 Proposition:

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

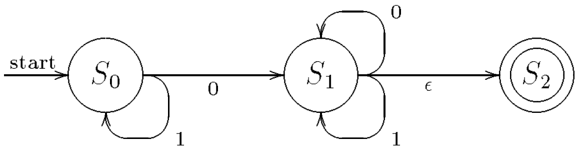
$$(\mathbf{Succ?}_{\mathsf{bin}}\ \mathsf{bin}_n) \twoheadrightarrow \mathsf{bin}_{n+1}.$$

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

Proof:

i. To compute the successor function on bin_n, we need to implement a state machine consisting of three states: 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 state machine is depicted in the following diagram:



In computing a successor of bin_n, we apply bin_n to *three* expressions: The first two substitute for the bits in the body of bin_n, and the third expression is used to mark the end of the stream of bits. Each expression needs to have access to

(i) An encoding of the current state (i.e. of either S_0 or S_1), denoted by σ.

(ii) 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.

(iii) A partial reconstruction of the body of the successive numeral, denoted by r.

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

$$\mathsf{U}_k^n \;=\; \lambda x_0 \cdots x_n . x_k$$

to encode the various choices. We store this information, and a procedure m in an ordered 4-tuple. Again, observe that:

$$\begin{array}{l}
([m, b_1, r_1, \sigma_1][m, b_2, r_2, \sigma_2]) \\
\quad \longrightarrow \; ((\lambda x.(x\ m\ b_1\ r_1\ \sigma_1))\ (\lambda x.(x\ m\ b_2\ r_2\ \sigma_2))) \\
\quad \longrightarrow \; ((\lambda x.(x\ m\ b_2\ r_2\ \sigma_2))\ m\ b_1\ r_1\ \sigma_1) \\
\quad \longrightarrow \; (m\ m\ b_2\ r_2\ \sigma_2\ b_1\ r_1\ \sigma_1)
\end{array}$$

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 particular behaviour of m depends upon many variables, we use *Decision-Logic Tables* to describe this behaviour in a concise manner.

Decision logic tables (DLT) [7, 8] are a tabular description of how an n-variable function can be used to control selection logic. The format of a DLT is as follows:

list of variable names	*list of all possible combinations of values of variables*
list of actions to be taken	*selections of combinations of actions as a function of combinations of variables*

DLT's 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 DLT abbreviations, in order to preserve clarity.

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

Main DLT: *Determining State*		
Value of σ_1	U_0^1	U_1^1
Dispatch to the DTL of S_0	$\surd$	
Dispatch to the DTL of S_1		$\surd$

The DLT at S_0									
Value of b_1	U_0^2	U_0^2	U_0^2	U_1^2	U_1^2	U_1^2	U_2^2	U_2^2	U_2^2
Value of b_2	U_0^2	U_1^2	U_2^2	U_0^2	U_1^2	U_2^2	U_0^2	U_1^2	U_2^2
$[m, b_2, (r_1\ w), \mathsf{U}_1^1]$	$\surd$	$\surd$							
$(r_1\ w)$			$\surd$						
$[m, b_2, (r_1\ z), \mathsf{U}_0^1]$				$\surd$	$\surd$				
$(r_1\ z\ w)$						$\surd$			
irrelevant							$\surd$	$\surd$	$\surd$

The DLT at S_1									
Value of b_1	U_0^2	U_0^2	U_0^2	U_1^2	U_1^2	U_1^2	U_2^2	U_2^2	U_2^2
Value of b_2	U_0^2	U_1^2	U_2^2	U_0^2	U_1^2	U_2^2	U_0^2	U_1^2	U_2^2
$[m, b_2, (r_1\ z), \mathsf{U}_1^1]$	$\surd$	$\surd$							
$(r_1\ z)$			$\surd$						
$[m, b_2, (r_1\ w), \mathsf{U}_1^1]$				$\surd$	$\surd$				
$(r_1\ w)$						$\surd$			
irrelevant							$\surd$	$\surd$	$\surd$

As can be seen from the DLT's for S_0 and S_1, when $b_1 = U_2^2$, the return value is irrelevant. We could return any value whatsoever, so we arbitrarily pick the $\mathbf{I}$ combinator. All three DLT's are combined in M:

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

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

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

ii. The proof is similar to the proof of Proposition 3.1.1, albeit more tedious. It can be found in our Ph.D. thesis [6].

■

3.3 The Predecessor Function

3.3.1 PROPOSITION:

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

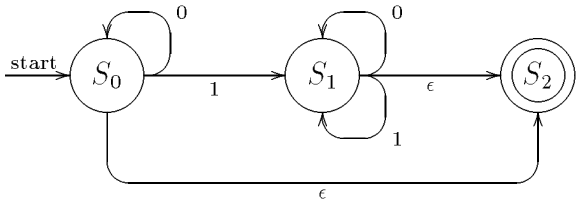
$$(\mathbf{Pred?}_{\mathsf{bin}}\ \mathsf{bin}_{n+1}) \twoheadrightarrow \mathsf{bin}_n.$$

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

Proof:

i. To compute the predecessor function on bin_n, we need to implement a state machine consisting of three states: 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 state machine is depicted in the following diagram:



In computing the predecessor of bin_n, just as in computing its successor, we apply bin_n to *three* expressions: The first two substitute for the bits in the body of bin_n, and the third expression is used to mark the end of the stream of bits. Each expression needs to have access to

(i) An encoding of the current state (i.e. of either S_0 or S_1), denoted by σ.

(ii) 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.

(iii) 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.

(iv) A partial reconstruction of the body of the preceeding 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 (i) and (ii) 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 are encountered. Together, (iii) and (iv) correspond to (iii) in the construction of the successor. We store this information, as well as a procedure m, in an ordered 5-tuple. As usual by now, observe that:

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

As one can see, m is passed a copy of itself, and all the information stored in both ordered 5-tuples (again, both ordered 5-tuples have m in common). We use three DLT's to represent the behaviour of m:

Main DLT: *Determining State*		
Value of σ_1	U_0^1	U_1^1
Dispatch to the DTL of σ_0	$\surd$	
Dispatch to the DTL of σ_1		$\surd$

The DLT at σ_0									
Value of b_1	U_0^2	U_0^2	U_0^2	U_1^2	U_1^2	U_1^2	U_2^2	U_2^2	U_2^2
Value of b_2	U_0^2	U_1^2	U_2^2	U_0^2	U_1^2	U_2^2	U_0^2	U_1^2	U_2^2
$[m, b_2, (r_{12}\ w), (r_{12}\ w), \mathsf{U}_0^1]$	$\surd$	$\surd$							
$(r_{12}\ w)$			$\surd$						
$[m, b_2, r_{11}, (r_{12}\ z), \mathsf{U}_1^1]$				$\surd$	$\surd$				
r_{11}						$\surd$			
irrelevant							$\surd$	$\surd$	$\surd$

The DLT at σ_1									
Value of b_1	U^2_0	U^2_0	U^2_0	U^2_1	U^2_1	U^2_1	U^2_2	U^2_2	U^2_2
Value of b_2	U^2_0	U^2_1	U^2_2	U^2_0	U^2_1	U^2_2	U^2_0	U^2_1	U^2_2
$[m, b_2, r_{11}, (r_{12}\ z), \mathsf{U}^1_1]$	$\surd$	$\surd$							
r_{11}			$\surd$						
$[m, b_2, (r_{12}\ w), (r_{12}\ w), \mathsf{U}^1_1]$				$\surd$	$\surd$				
$(r_{12}\ w)$						$\surd$			
irrelevant							$\surd$	$\surd$	$\surd$

As can be seen from the DLT's for S_0 and S_1, when $b_1 = U^2_2$, the return value is irrelevant, just like in the derivation of the successor. We could return any value whatsoever, so we once again pick the $\mathbf{I}$ combinator. All three DLT's are combined in M:

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

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

$$
\begin{aligned}
\mathbf{Pred?}_{\mathsf{bin}} \;=\; \lambda n z w . (n\ & [M, \mathsf{U}^2_0, z, \mathbf{I}, \mathsf{U}^1_0] \\
& [M, \mathsf{U}^2_1, z, \mathbf{I}, \mathsf{U}^1_0] \\
& [M, \mathsf{U}^2_2, z, \mathbf{I}, \mathsf{U}^1_0])
\end{aligned}
$$

Recall that r_1 contains the partial reconstruction of the preceeding 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 3.1.1, albeit more tedious. It can be found in our Ph.D. thesis [6].

■

3.4 Adequacy

3.4.1 PROPOSITION: *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 [1] that bin is an adequate numeral system. ■

4 Conclusion and Assessment

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

4.1 Extensibility

The definition of bin is easily extensible to other basis, though base 10 might well be the only other useful choice. Similarly, bin can be extended to have a sign, a decimal point and an exponent, facilitating fixed-size floating point arithmetic.

There is some interest in possible representations of real numbers on computers, as streams of decimals or integer coefficients of continued fractions. It is possible that the laziness inherent in the *normal order* of evaluation could facilitate a lazy numeral system for real numbers as an extension of bin. Such a numeral system would require that certain operations, such as the test for equality and an encoding mechanism (see Section 4.3) be restricted, in order to avoid non-termination. Lazy numbers offer a potential for efficiency, while maintaining the flexibility of carrying on a computation to arbitrary precision.

4.2 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. This complexity is independent of the order of evaluation. The use of selectors rather than arbitrary tokens in the dispatching mechanism results in considerable gains in efficiency, and the resulting λ-expressions are both more concise and simpler to verify.

4.3 Implementation

The numeral system bin is extremely suitable for implementation in functional programming languages which 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 [4]. Our implementation can be combined with the Gödeliser developed as a part of our Ph.D. thesis [5, 6], so that such numerals, as well as possible extensions to the bin numeral system, can be displayed.

4.4 Decision-Logic Tables

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

Acknowledgements

I am grateful to BRICS[2] for hosting me this summer and for providing a stimulating environment. Thanks are also due to Olivier Danvy, Daniel P. Friedman, and Larry Moss for their comments and encouragement.

The diagrams were drawn with Kristoffer Rose's XY-pic package.

[2] Basic Research in Computer Science,
Centre of the Danish National Research Foundation.

References

[1] Hendrik P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* North-Holland, 1984.

[2] Hendrik P. Barendregt. Personal Communication, Bloomington, Indiana, 1990.

[3] Alonzo Church. *The Calculi of Lambda-Conversion.* Princeton University Press, 1941.

[4] William Clinger and Jonathan Rees (editors). Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[5] Mayer Goldberg. Gödelisation in the λ-calculus. BRICS Research Series RS-95-38, DAIMI, Aarhus University, Denmark, July 1995.

[6] Mayer Goldberg. *Recursive Application Survival in the λ-Calculus.* PhD thesis, Department of Computer Science, Indiana University, December 1995. Forthcoming.

[7] T.F. Kavanagh. Tabsol – a fundamental concept for system-oriented language. In *Proceedings of the Eastern Joint Computer Conference*, pages 117–127, New York, December 1960.

[8] Herman McDaniel. *An Introduction to Decision Logic Tables.* John Wiley & Sons, 1968.

Recent Publications in the BRICS Report Series

RS-95-42 **Mayer Goldberg.** ***An Adequate Left-Associated Binary Numeral System in the λ-Calculus.*** **August 1995. 16 pp.**

RS-95-41 **Olivier Danvy, Karoline Malmkjær, and Jens Palsberg.** ***Eta-Expansion Does The Trick.*** **August 1995. 23 pp.**

RS-95-40 **Anna Ingólfsdóttir and Andrea Schalk.** ***A Fully Abstract Denotational Model for Observational Congruence.*** **August 1995. 29 pp.**

RS-95-39 **Allan Cheng.** ***Petri Nets, Traces, and Local Model Checking.*** **July 1995. 32 pp. Full version of paper appearing in Proceedings of AMAST '95, LNCS 936, 1995.**

RS-95-38 **Mayer Goldberg.** ***Gödelisation in the λ-Calculus.*** **July 1995. 7 pp.**

RS-95-37 **Sten Agerholm and Mike Gordon.** ***Experiments with ZF Set Theory in HOL and Isabelle.*** **July 1995. 14 pp. To appear in** ***Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and its Applications*****, LNCS, 1995.**

RS-95-36 **Sten Agerholm.** ***Non-primitive Recursive Function Definitions.*** **July 1995. 15 pp. To appear in** ***Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and its Applications*****, LNCS, 1995.**

RS-95-35 **Mayer Goldberg.** ***Constructing Fixed-Point Combinators Using Application Survival.*** **June 1995. 14 pp.**

RS-95-34 **Jens Palsberg.** ***Type Inference with Selftype.*** **June 1995. 22 pp.**

RS-95-33 **Jens Palsberg, Mitchell Wand, and Patrick O'Keefe.** ***Type Inference with Non-structural Subtyping.*** **June 1995. 22 pp.**

RS-95-32 **Jens Palsberg.** ***Efficient Inference of Object Types.*** **June 1995. 32 pp. To appear in** ***Information and Computation.*** **Preliminary version appears in** ***Ninth Annual IEEE Symposium on Logic in Computer Science*****, LICS '94 Proceedings, pages 186–195.**