An investigation of compact and efficient number representations in the pure lambda calculus

Torben Æ. Mogensen

DIKU, University of Copenhagen, Denmark
Universitetsparken 1, DK-2100 Copenhagen O, Denmark
phone: (+45) 35321404 fax: (+45) 35321401 email: torbenm@diku.dk

Abstract. We argue that a compact right-associated binary number representation gives simpler operators and better efficiency than the left-associated binary number representation proposed by den Hoed and investigated by Goldberg. This representation is then generalised to higher number-bases and it is argued that bases between 3 and 5 can give higher efficiency than binary representation.

1 Introduction

The archetypal number representation in the pure lamda calculus is Church numerals, where a number n is represented as a combinator that applies a function n times to its argument, so for example 3 is represented as $\lambda f x.f\ (f\ (f\ x))$. The definitions of addition, multiplication and raising to power are extremely simple when using Church numerals. However, Church numerals are not very compact and the operations, though simple, are quite costly. den Hoed [6] suggested a compact representation of binary numbers, where a binary number $b_n \ldots b_1 b_0$ is represented as $\lambda x_0 x_1 . x_{b_0}\ x_{b_1} \ldots x_{b_n}$ or, equivalently, $\lambda x_0 x_1 . ((x_{b_0}\ x_{b_1}) \ldots x_{b_n})$. This representation is, hence, *left-associated*.

Given this as a challenge, Mayer Goldberg [2] has shown that efficient operators exist for this representation. However, we believe we can achieve better by using a right-associated representation and introducing one more variable to mark the end of a bit sequence: 0 is represented by $\lambda z x_0 x_1 . z$ and $b_n \ldots b_1 b_0$, where $b_n \neq 0$ is represented by $\lambda z x_0 x_1 . x_{b_0}\ (x_{b_1}\ (\ldots (x_{b_n}\ z))$.

We use standard lambda calculus notation: Identity function: $I \equiv \lambda x.x$, booleans: $T \equiv \lambda xy.x$, $F \equiv \lambda xy.y$, tuples: $[e_1, \ldots, e_n] \equiv \lambda x.x\ e_1\ \ldots\ e_n$ and projections: $\pi_k^n \equiv \lambda t.t(\lambda x_1 \cdots x_n . x_k)$. The identity function is $I \equiv \lambda x.x$. We use the notation $\lceil n \rceil$ to mean the representation of the number n. Examples:

$$\lceil 0 \rceil \equiv \lambda z x_0 x_1 . z$$
$$\lceil 1 \rceil \equiv \lambda z x_0 x_1 . x_1\ z$$
$$\lceil 5 \rceil \equiv \lambda z x_0 x_1 . x_1\ (x_0\ (x_1\ z))$$

We note that this representation, like most binary representations, allow leading zeroes. For example, 0 can be represented as $\lambda z x_0 x_1 . x_0\ x$ as well as $\lambda z x_0 x_1 . z$. We want to avoid introducing leading zeroes. Goldberg [2] builds this into the operators, but we define a separate operator that is applied when needed.

2 Basic operations on numbers

Shifting a binary number up by one bit is easily done in constant time:

$$\uparrow \equiv \lambda bn.\lambda zx_0x_1.b\ x_0\ x_1\ (n\ z\ x_0\ x_1)$$

Single bits are represented as $0 \equiv T$, $1 \equiv F$. For brevity and slightly better efficiency, we will often use specialised versions of $\uparrow$:

$$\uparrow_0 \equiv \lambda n.\lambda zx_0x_1.x_0\ (n\ z\ x_0\ x_1) \qquad \uparrow_1 \equiv \lambda n.\lambda zx_0x_1.x_1\ (n\ z\ x_0\ x_1)$$

We can make a function that finds the least significant bit in a number by

$$lsb \equiv \lambda n.n\ T\ (\lambda x.T)\ (\lambda x.F)$$

The idea in this operator is that we apply a number to three terms: One for handling the empty bit string, one for handling the case where the least significant bit is 0 and one for the case where the least significant bit is 1.

We define, in a similar way, operators for stripping leading zeroes and dividing a binary number by 2:

$$\begin{array}{lll} strip \equiv \lambda n.\pi_1^2\ (n\ Z\ A\ B) & \text{where} & \downarrow \equiv \lambda n.\pi_2^2\ (n\ Z\ A\ B) \quad \text{where} \\ Z \equiv [\lceil 0 \rceil, T] & & Z \equiv [\lceil 0 \rceil, \lceil 0 \rceil] \\ A \equiv \lambda p.p\ (\lambda nz.[z\ \lceil 0 \rceil\ (\uparrow_0 n), z]) & & A \equiv \lambda p.p\ (\lambda nm.[\uparrow_0 n, n]) \\ B \equiv \lambda p.p\ (\lambda nz.[\uparrow_1 n, F]) & & B \equiv \lambda p.p\ (\lambda nm.[\uparrow_1 n, n] \end{array}$$

For the *strip* operator, we build a pair that contains the most compact representation of the bits that have been treated and an indication if this is the empty bit sequence. Finally, we use π_1^2 to extract the final result from the pair. For division, the pair contains the whole number and the number divided by 2.

An adequate number system needs operators for zero-testing, increment and decrement. We make these using the same techniques as above:

$$zero? \equiv \lambda n.n\ T\ I\ (\lambda x.F)$$

$$\begin{array}{lll} succ \equiv \lambda n.\pi_2^2\ (n\ Z\ A\ B) & \text{where} & pred \equiv \lambda n.\pi_2^2\ (n\ Z\ A\ B) \quad \text{where} \\ Z \equiv [\lceil 0 \rceil, \lceil 1 \rceil] & & Z \equiv [\lceil 0 \rceil, \lceil 0 \rceil] \\ A \equiv \lambda p.p\ (\lambda nm.[\uparrow_0 n, \uparrow_1 n]) & & A \equiv \lambda p.p\ (\lambda nm.[\uparrow_0 n, \uparrow_1 m]) \\ B \equiv \lambda p.p\ (\lambda nm.[\uparrow_1 n, \uparrow_0 m]) & & B \equiv \lambda p.p\ (\lambda nm.[\uparrow_1 n, \uparrow_0 n]) \end{array}$$

pred can introduce a leading zero if the number is a power of two.

3 Other number bases

It is easy to generalise the above representation to other number bases. If we have an n-digit number $d_{n-1} \ldots d_1 d_0$ (where $d_{n-1} > 0$) in base b, we can represent

this as $\lambda z x_0 x_1 \ldots x_{n-1}.x_{d_0}\ (x_{d_1}\ (\ldots (x_{d_{n-1}}\ z)\ldots))$. 0 is, of course, represented as $\lambda z x_0 x_1 \ldots x_{n-1}.z$. The operations are also easily generalised:

$$\begin{aligned}\uparrow_i &\equiv \lambda n.\lambda z x_0 \ldots x_{b-1}.x_i\ (n\ z\ x_0\ \ldots\ x_{b-1})\\ \uparrow\ &\equiv \lambda d n.\lambda z x_0 \ldots x_{b-1}.d\ x_0\ \ldots\ x_{b-1}\ (n\ z\ x_0\ \ldots\ x_{b-1})\end{aligned}$$

where a digit d for the general $\uparrow$ operator is represented as $\lambda x_0 \ldots x_{b-1}.x_d$. The *lsb* operator is replaced by an *lsd* (least significant digit) operator:

$$lsd \equiv \lambda n.n\ (\lambda x_0 \ldots x_{b-1}.x_0)\ (\lambda x.\lambda x_0 \ldots x_{b-1}.x_1)\ \ldots\ (\lambda x.\lambda x_0 \ldots x_{b-1}.x_{b-1})$$

Digits are represented as above. The strip operator is simple to generalise, as are diving by the base (*i.e.*, removing the last digit), the zero test and the successor and predecessor operators:

$$\begin{aligned}strip &\equiv \lambda n.\pi_1^2\ (n\ Z\ A_0\ \ldots\ A_{b-1}) \qquad \text{where}\\ Z &\equiv [\lceil 0\rceil, T]\\ A_0 &\equiv \lambda p.p\ (\lambda n z.[z\ \lceil 0\rceil\ (\uparrow_0 n), z])\\ A_i &\equiv \lambda p.p\ (\lambda n z.[\uparrow_i n, F])\ \ ,\quad i>0\end{aligned}$$

$$\begin{aligned}\downarrow &\equiv \lambda n.\pi_2^2\ (n\ Z\ A_0\ \ldots\ A_{b-1}) \qquad \text{where}\\ Z &\equiv [\lceil 0\rceil, \lceil 0\rceil]\\ A_i &\equiv \lambda p.p\ (\lambda n m.[\uparrow_i n, n])\end{aligned}$$

$$zero? \equiv \lambda n.n\ T\ I\ (\lambda x.F)^{b-1}$$

$$\begin{aligned}succ &\equiv \lambda n.\pi_2^2\ (n\ Z\ A_0\ \ldots\ A_{b-1}) \qquad \text{where}\\ Z &\equiv [\lceil 0\rceil, \lceil 1\rceil]\\ A_i &\equiv \lambda p.p\ (\lambda n m.[\uparrow_i n, \uparrow_{i+1} n]) \qquad , i<b-1\\ A_{b-1} &\equiv \lambda p.p\ (\lambda n m.[\uparrow_{b-1} n, \uparrow_0 m])\end{aligned}$$

$$\begin{aligned}pred &\equiv \lambda n.\pi_2^2\ (n\ Z\ A_0\ \ldots\ A_{b-1}) \qquad \text{where}\\ Z &\equiv [\lceil 0\rceil, \lceil 0\rceil]\\ A_0 &\equiv \lambda t.t\ (\lambda n m z.[\uparrow_0 n, \uparrow_{b-1} m]\\ A_i &\equiv \lambda t.t\ (\lambda n m z.[\uparrow_i n, \uparrow_{i-1} n]) \qquad , i>0\end{aligned}$$

4 Comparing different number bases

Using he representations shown above, each digit in a base b number is represented by a variable and an application. This could lead us to believe that we can get arbitrarily compact representations by choosing higher number bases. But an actual representation of a lambda term on a machine will have to represent variables using a fixed alphabet, so the number of symbols needed to represent a variable depend on the number of different variables used. In order to be precise about this, we will use de Bruijn notation: each occurrence of a variable is replaced by a so-called de Bruijn number that counts the number of lambda

abstractions one has to pass in the syntax tree to get to the abstraction that binds the variable. Abstractions no longer name the bound variable.

We still haven't solved the problem, as we have replaced an unbounded number of variables by an unbounded number of de Bruijn numbers. We can, however, replace these by number strings. A common representation of lambda terms uses unary representation for de Bruijn numbers such that, *e.g.*, 3 is represented as $sssz$. This has the operational interpretation that z returns the value of the variable at the front of the current environment while s walks down one level in the environment. This idea is the basis of several abstract machines [1] [3].

We will explicitly bracket applications. By omitting the (redundant) opening bracket, we get a representation similar to reverse polish notation. Examples:

lambda term	de Bruijn notation	compact representation
$\lambda x.x$	$\lambda 0$	λz
$\lambda xy.y\ x$	$\lambda\lambda 0\ 1$	$\lambda\lambda zsz)$
$\lambda z x_0 x_1.x_1\ (x_0\ (x_1\ z))$	$\lambda\lambda\lambda 0\ (1\ (0\ 2))$	$\lambda\lambda\lambda zszzssz)))$

Using this representation, an n-bit number is represented by a string of length $2.5 \times n + 1$ on average for den Hoed's left-associated representation. The right-associated representation requires and $2.5 \times n + 6$ symbols on average to represent an n-bit number. Given that a base-b digit and an application takes on average $(b+3)/2$ symbols to write using our notation and that you need $n * ln(2)/ln(b)$ base-b digits to represent an n-bit number, we get the following measures of compactness for our base-b representation:

base	2	3	4	5	6	7	8	9	10
compactness	2.5	1.89	1.75	1.72	1.74	1.78	1.83	1.89	1.96

This table indicates that we will get the asymptotically most compact representation if we use the base 5 representation. Since the above compactness measures depend somewhat on the exact details of the textual representation, we must take the numbers with a grain of salt. It is our guess that, for most reasonable measures, it will be better to use a base between 3 and 6 rather than base 2. A rough estimate is that the cost of processing a digit is roughly proportional to the number of different digits, so computational cost should follow a pattern similar to the compactness measures. If the cost per digit is $kb + l$, the cost of processing an n-bit number (in base b) is $(kb+l)ln(2)/ln(b)$. Hence, the minimum is obtained when $b(ln(b) - 1) = l/k$. This gives:

l/k	0	1/4	1/2	2/3	1	3/2	2	3	4
minimum	2.72	2.95	3.18	3.32	3.59	3.97	4.32	4.97	5.57
optimal base	3	3	3	3	4	4	4	5	6

As can be seen, the minimum is always at b higher than 2. This indicates that it will always be better to use base 3 instead of base 2, and it may be better to choose an even higher base. Note that these measures are about asymptotic costs. For small numbers it will be better to use a small number base, such as binary or even unary.

5 Balanced ternary representation

An interesting number representation is balanced ternary. This variant base-3 notation has digits "-" (-1), "0" and "+" (1). This allows representation of negative numbers without a separate sign symbol. Balanced ternary has been used in some early Russian computers, but lost to the simplicity of binary electronics. It may, however, well be good for number representation in the lamda calculus.

Representing numbers in balanced ternary is very similar to ordinary ternary representation: We represent a balanced ternary digit string $t_n \ldots t_0$, where $t_n \neq 0$, as $\lambda z x_- x_0 x_+ . x_{t_0}\ (\ldots (x_{t_n}\ z))$. Examples:

$$\begin{array}{ll} \lceil 0 \rceil & \equiv \lambda z x_- x_0 x_+ . z \\ \lceil 1 \rceil & \equiv \lambda z x_- x_0 x_+ . x_+\ z \\ \lceil -1 \rceil & \equiv \lambda z x_- x_0 x_+ . x_-\ z \\ \lceil 2 \rceil & \equiv \lambda z x_- x_0 x_+ . x_-\ (x_+\ z) \\ \lceil -5 \rceil & \equiv \lambda z x_- x_0 x_+ . x_+\ (x_+\ (x_-\ z)) \end{array}$$

The operators are similar to those of normal ternary.

$$\begin{array}{ll} \uparrow_- & \equiv \lambda n . \lambda z x_- x_0 x_+ . x_-\ (n\ z\ x_-\ x_0\ x_+) \\ \uparrow_0 & \equiv \lambda n . \lambda z x_- x_0 x_+ . x_0\ (n\ z\ x_-\ x_0\ x_+) \\ \uparrow_+ & \equiv \lambda n . \lambda z x_- x_0 x_+ . x_+\ (n\ z\ x_-\ x_0\ x_+) \\ \uparrow & \equiv \lambda t n . \lambda z x_- x_0 x_+ . t\ x_-\ x_0\ x_+\ (n\ z\ x_-\ x_0\ x_+) \end{array}$$

where a ternary digit (trit) is represented as $\hat{-} \equiv \lambda x_- x_0 x_+ . x_-$, $\hat{0} \equiv \lambda x_- x_0 x_+ . x_0$ and $\hat{+} \equiv \lambda x_- x_0 x_+ . x_+$. Operators for least significant trit, zero-testing, addition and subtraction are almost the same as for "ordinary" ternary numbers:

$$\begin{array}{lll} \mathit{lst} & \equiv \lambda n . n\ \hat{0}\ (\lambda x . \hat{-})\ (\lambda x . \hat{0})\ (\lambda x . \hat{+}) & \\ & & \\ \mathit{zero?} & \equiv \lambda n . n\ T\ (\lambda x . F)\ I\ (\lambda x . F) & \\ & & \\ \mathit{succ} & \equiv \lambda n . \pi_2^2\ (n\ Z\ A_-\ A_0\ A_+) & \text{where} \\ Z & \equiv [\lceil 0 \rceil, \lceil 1 \rceil] & \\ A_- & \equiv \lambda t . t\ (\lambda n m z . [\uparrow_-\ n, \uparrow_0\ n]) & \\ A_0 & \equiv \lambda t . t\ (\lambda n m z . [\uparrow_0\ n, \uparrow_+\ n]) & \\ A_+ & \equiv \lambda t . t\ (\lambda n m z . [\uparrow_+\ n, \uparrow_-\ m]) & \\ & & \\ \mathit{pred} & \equiv \lambda n . \pi_2^2\ (n\ Z\ A_-\ A_0\ A_+) & \text{where} \\ Z & \equiv [\lceil 0 \rceil, \lceil -1 \rceil] & \\ A_- & \equiv \lambda t . t\ (\lambda n m z . [\uparrow_-\ n, \uparrow_+\ m]) & \\ A_0 & \equiv \lambda t . t\ (\lambda n m z . [\uparrow_0\ n, \uparrow_-\ n]) & \\ A_+ & \equiv \lambda t . t\ (\lambda n m z . [\uparrow_+\ n, \uparrow_0\ n]) & \end{array}$$

Note the symmetry of *succ* and *pred*. They can now both introduce a leading zero. Since we now have negative numbers, an useful operator is negation. This is just a matter of replacing + by − and *vice versa*:

$$\mathit{negate} \equiv \lambda n . \lambda z x_- x_0 x_+ . n\ z\ x_+\ x_0\ x_-$$

6 Binary operators

Some binary operators, like addition, require walking down two digit strings simultaneously. The representations we have shown so far aren't geared to this, so we introduce yet another representation, which allows us to inspect one ternary digit at a time. The representation is similar to the previously shown, except that the variables z, x_-, x_0 and x_+ are abstracted at every digit rather than globally for all digits:

$$\begin{array}{l} \lfloor 0 \rfloor \equiv \lfloor \epsilon \rfloor \equiv \lambda z x_- x_0 x_+ . z \\ \lfloor t_n \ldots t_1 t_0 \rfloor \equiv \lambda z x_- x_0 x_+ . x_{t_0} \ (\lfloor t_n \ldots t_1 \rfloor) \quad , t_n \neq 0 \end{array}$$

where ϵ represents the empty digit string. We introduce operators $\Uparrow$, which takes a number n in the "normal" balanced ternary representation ($\lceil n \rceil$) to the new representation ($\lfloor n \rfloor$) and its inverse $\Uparrow$:

$$\begin{array}{lll} \Uparrow & \equiv \lambda n.n \ \lfloor 0 \rfloor \ A_- \ A_0 \ A_+ & \text{where} \\ A_- & \equiv \lambda n.\lambda z x_- x_0 x_+ . x_- \ n & \\ A_0 & \equiv \lambda n.\lambda z x_- x_0 x_+ . x_0 \ n & \\ A_+ & \equiv \lambda n.\lambda z x_- x_0 x_+ . x_+ \ n & \end{array}$$

$$\Downarrow \ n \equiv n \ \lceil 0 \rceil \ (\lambda m.\uparrow_-(\Downarrow \ m)) \ (\lambda m.\uparrow_0(\Downarrow \ m)) \ (\lambda m.\uparrow_+(\Downarrow \ m))$$

The latter is given by a recursive equation. Solving this by using a recursion operator will make termination dependent on evaluation order. Hence, we delay the recursion until the right-hand side has been reduced. A similar trick was used in [4]:

$$\begin{array}{l} \Downarrow \equiv U \ U \quad \text{where} \\ U \equiv \lambda u n.n \ (\lambda u.\lceil 0 \rceil) \ (\lambda u m.\uparrow_-(u \ u \ m)) \ (\lambda u m.\uparrow_0(u \ u \ m)) \ (\lambda u m.\uparrow_+(u \ u \ m)) \ u \end{array}$$

The idea used in the following is that one of the arguments to a binary operator is converted to the new representation by $\Uparrow$ while the other is processed directly in the original representation. Hence, we define an operator eq_0 that takes a number in the original representation and a number in the new representation and compares these for equality. We then define an equality operator $eq \equiv \lambda x y.eq_0 \ x \ (\Uparrow y)$ that takes both arguments in the original representation.

$$\begin{array}{lll} eq_0 & \equiv \lambda n.n \ Z \ A_- \ A_0 \ A_+ & \text{where} \\ Z & \equiv \lambda n.zero? \ (\Downarrow \ n) & \\ A_- & \equiv \lambda e n.n \ F \ (\lambda m.e \ m) \ (\lambda m.F) \ (\lambda m.F) & \\ A_0 & \equiv \lambda e n.n \ (e \ n) \ (\lambda m.F) \ (\lambda m.e \ m) \ (\lambda m.F) & \\ A_+ & \equiv \lambda e n.n \ F \ (\lambda m.F) \ (\lambda m.F) \ (\lambda m.e \ m) & \end{array}$$

We, similarly, first define an addition operator add_0 that takes its second argument in the new representation and use this to define $add \equiv \lambda x y.add_0 \ x \ (\Uparrow y)$. At each step in the addition process, we have two trits and a carry (which is also a trit, represented as $\hat{-}$, $\hat{0}$ or $\hat{+}$). This can give results from -3 to 3, which are

output as a trit and a carry, which is used for the next step. We find the trits in one number by applying suitable Z, A_-, A_0 and A_+ to the number. The trits of the other number are obtained by a 4-way branch in each A-term, as in eq_0. The value of the carry is found by a 3-way branch (in each B-term below).

$$
\begin{array}{lll}
add_0 & \equiv \lambda n.C\ (n\ Z\ A_-\ A_0\ A_+) & \text{where} \\
Z & \equiv \lambda nc.c\ (pred\ (\Downarrow\ n))\ (\Downarrow\ n)\ (succ\ (\Downarrow\ n)) & \\
A_- & \equiv \lambda anc.n\ (B_-\ n)\ B_{-2}\ B_-\ B_0 & \\
A_0 & \equiv \lambda anc.n\ (B_0\ n)\ B_-\ B_0\ B_+ & \\
A_+ & \equiv \lambda anc.n\ (B_+\ n)\ B_0\ B_+\ B_{+2} & \\
B_{-2} & \equiv \lambda m.c\ (\uparrow_0\ (a\ m\ \hat{-}))\ (\uparrow_+\ (a\ m\ \hat{-}))\ (\uparrow_-\ (a\ m\ \hat{0})) & \\
B_- & \equiv \lambda m.c\ (\uparrow_+\ (a\ m\ \hat{-}))\ (\uparrow_-\ (a\ m\ \hat{0}))\ (\uparrow_0\ (a\ m\ \hat{0})) & \\
B_0 & \equiv \lambda m.\uparrow\ c\ (a\ m\ \hat{0}) & \\
B_+ & \equiv \lambda m.c\ (\uparrow_0\ (a\ m\ \hat{0}))\ (\uparrow_+\ (a\ m\ \hat{0}))\ (\uparrow_-\ (a\ m\ \hat{+})) & \\
B_{+2} & \equiv \lambda m.c\ (\uparrow_+\ (a\ m\ \hat{0}))\ (\uparrow_-\ (a\ m\ \hat{+}))\ (\uparrow_0\ (a\ m\ \hat{+})) & \\
C & \equiv \lambda an.a\ n\ \hat{0} &
\end{array}
$$

We have here been a bit sloppy, as the B terms have free variables a and c which correspond to different bound variables depending on where the B is substituted. Subtraction is easily obtained by negating one argument before addition, so we just define the subtraction operator as $sub \equiv \lambda xy.add\ x\ (negate\ y)$. Multiplication is simple to define using addition and subtraction:

$$
\begin{array}{lll}
mul & \equiv \lambda nm.n\ \lceil 0 \rceil\ A_-\ A_0\ A_+ & \text{where} \\
A_- & \equiv \lambda n.sub\ (\uparrow_0\ n)\ m & \\
A_0 & \equiv \lambda n.(\uparrow_0\ n) & \\
A_+ & \equiv \lambda n.add\ (\uparrow_0\ n)\ m &
\end{array}
$$

7 Benchmarks

Our claims of efficiency above have, with the exception of the measure of compactness, been rather weakly argued on grounds of simpler operators. To remedy this, we have timed some calculations using different representations. To execute the calculations, we need a normaliser for the lambda calculus. We have elected to use a normaliser based on normalisation by evaluation and implemented in scheme [5]. This uses a call-by-value reduction strategy.

The, admittedly simplistic, test we use is counting from 0 to 50000 using the *succ* operator. While this may not be representative of "real" calculations, we have in the majority of representations only implemented unary operators, which limits the scope of tests we can make on these.

Representation	time to count to 50000
Right-associated binary	6270 ms
Base 3	4380 ms
Base 4	4000 ms
Base 5	3900 ms
Base 6	4470 ms
Balanced ternary	4660 ms

The time used to execute the benchmark drops by more than 30% from binary to base 3, but the advantage of further going to base 4 or 5 is less (around 10%).

This supports the conjecture that the optimal base is higher than 2 and likely to be around 4 or 5. Balanced ternary is slightly slower than ordinary ternary, but that should be no surprise since the benchmark doesn't use negative numbers.

8 Conclusion

We have investigated a number of different compact number representations for the lambda calculus, starting with the left-associated binary number system suggested by den Hoed. We argued that we get better calculation efficiency by choosing a right-associated representation and adding an explicit end symbol. We then found that number bases in the range 3-6 increase compactness and calculation efficiency over binary representation.

While execution efficiency seems optimal at base 4 or 5, the operators become much bigger in these bases than in ternary: The size of the *succ* operator is approximately quadratic in the number base, and the size of the addition operator is approximately cubic in the number base. This may make base 3 the overall best choice. If ease of conversion to/from binary notation is important, a base-4 representation might be preferable.

The ease of handling negative numbers leads us to suggest using a balanced ternary number representation, for which we present some binary operators in addition to the unary operators we presented for the other systems.

While we, arguably, gain efficiency over Goldbergs operators for den Hoed's representation, this may not be an entirely fair comparison: After all, Goldbergs work was an answer to a challenge if he could make decent operations for a specific number system that was not designed for that purpose. Hence, he didn't *a priori* have the freedom we have exploited of changing the number system to gain better efficiency.

References

1. Guy Cousineau, Pierre-Louis Curien, Michel Mauny, and Ascander Suárez. Combinateurs catégorieques et impleméntation des languages fonctionnels. Technical Report 86-3, LIENS, 1986.
2. Mayer Goldberg. An adequate and efficient left-associated binary numreal system in the λ-calculus. *Journal of Functional Programming*, 10(6):607–623, November 2000.
3. Jean-Louis Krivine. Un interpréteur du λ-calcul. 1985.
4. T. Æ. Mogensen. Self-applicable online partial evaluation of the pure lambda calculus. In William L. Scherlis, editor, *Proceedings of PEPM '95*, pages 39–44. ACM, ACM Press, 1995.
5. T. Æ. Mogensen. Gödelisation in the untyped lambda calculu. In O. Danvy, editor, *Proceedings of PEPM'99*. BRICS Notes Series, 1999.
6. W. L. van den Poel, C. E. Schaap, and G. van der Mey. New arithmetical operators in the theory of combinators. *Indagationes Mathematicae*, (42):271–325, 1980.