A NEW REPRESENTATION OF THE RATIONAL NUMBERS FOR FAST EASY ARITHMETIC

ERIC C. R. HEHNER and R. N. S. HORSPOOL

Abstract. A novel system for representing the rational numbers based on Hensel's *p*-adic arithmetic is proposed. The new scheme uses a compact variable-length encoding that may be viewed as a generalization of radix complement notation. It allows exact arithmetic, and approximate arithmetic under programmer control. It is superior to existing coding methods because the arithmetic operations take particularly simple, consistent forms. These attributes make the new number representation attractive for use in computer hardware.

Key words. number systems, number representation, rational arithmetic, *p*-adic numbers, radix complement, floating-point

1. Introduction. "It's very illuminating to think about the fact that some –at most four hundred– years ago, professors at European universities would tell the brilliant students that if they were very diligent, it was not impossible to learn how to do long division. You see, the poor guys had to do it in Roman numerals. Now, here you see in a nutshell what a difference there is in a good and bad notation." (Dijkstra 1977)

We consider that a good scheme for representing numbers, especially for computers, would have the following characteristics.

(a) All rational numbers are finitely representable. This requires that the representation be variable-length.

(b) The representation is compact. It should require less space, on average, than the fixed-length schemes commonly used in computers. Since the numbers provided by a fixed-length representation are not used equally often, compactness can be achieved by giving frequently-occurring numbers short encodings at the expense of longer encodings for less-frequent numbers.

(c) The addition, subtraction, and multiplication algorithms are those of the usual integer arithmetic. The division algorithm is as easy as multiplication, and it proceeds in the same direction as the other three algorithms. This property is important for the storage and retrieval of variable-length operands and results.

Although the desired representation is variable-length, there is no implication that operations must be performed serially by digit. Just as data can be retrieved and stored *d* digits at a time, so the arithmetic unit can be designed to perform operations *d* digits at a time. By choosing *d* to be large compared to the average length of operands, we can obtain the speed of a fixed-length design together with the ability to handle operands that are not representable in a fixed length (Wilner 1972).

2. Background. Before we present our proposal, we shall briefly review representations in common use. (For their history, see (Knuth 1969a).)

Almost universally, the sequence of digits $\cdots d_i \cdots d_3\, d_2\, d_1\, d_0$ is used to represent the nonnegative integer

$$n = \sum_i d_i b^i$$

where b (the base) is an integer greater than one (usually two or ten), and each digit d_i, represents an integer in the range $0 \le d_i < b$. As a rule, we do not write leading 0s. We must break the rule, however, to represent zero (if we follow it, there is nothing to write).

There is no direct representation of negative integers in common use. Instead we prefix a unary operator to the representation of positive integers. The combination of "sign and magnitude" is indirect because, to perform arithmetic, we may first have to apply some algebraic transformations. If asked to add two numbers, we first examine the signs to determine whether to use the addition or subtraction algorithm; if we are using the subtraction algorithm, we compare the magnitudes to determine which is to be subtrahend, and which minuend. With a direct representation, if asked to add, we simply add. The radix complement representation is direct in this sense, but it includes only a finite subset of the integers.

Rationals are commonly represented by a pair of integers: a numerator and denominator. In this form, multiplication and division are reasonably easy, but addition and subtraction are relatively hard, and normalization is difficult (Horn 1977). When addition and subtraction are wanted more often than multiplication and division, a representation that makes the former easier at the expense of the latter would be preferable. For this reason, we usually restrict our numbers to a subset of the rationals known as the "fixed-point" or "floating-point" numbers. By inserting a radix point in a sequence of digits (fixed-point), or indicating by means of an exponent where a radix point should be placed (floating-point), we represent those rationals such that, in lowest terms, the denominator divides some power of the base. In this form, addition and subtraction are, after alignment of the radix point, the same as for integers. With a variable-length representation, a major difficulty with the usual division algorithm is that it proceeds from left to right, opposite to the direction of the other three algorithms. To simplify retrieval, processing, and storage, all algorithms should examine their operands and produce their results in the same direction.

The left-to-right division algorithm gives us a way of extending the fixed/floating point representation to include all positive rationals: an infinite but eventually repeating sequence of digits can be finitely denoted by indicating the repeating portion. On paper, the repeating portion is sometimes denoted by overscoring it; for example, $611/495 = 1.2\overline{34}$. A minor annoyance is the fact that representations are not unique; for example, $0.\overline{9} = 1$ and $0.4\overline{9} = 0.5$. A major annoyance is that further arithmetic is awkward: addition normally begins with the rightmost digit, but a sequence that extends infinitely to the right has no rightmost digit.

The usual representation of nonnegative integers can be extended in various ways. The base may be negative (Songster 1963), or even imaginary (Knuth 1960). In the "balanced ternary" representation (Avizienis 1971), the base is three, and the digits represent the integers minus-one, zero, and one. Our extension, which we now present, is in quite a different direction.

3. Constructing the representation. To construct our representation, we shall follow the approach of Hensel's *p*-adic arithmetic (Hensel 1908, 1913). We begin with the usual representation of nonnegative integers, that is, a sequence of digits $\cdots d_3\, d_2\, d_1\, d_0$. Each digit d_i represents an integer in the range $0 \le d_i < b$, where the base b is an integer greater than one. We then constructs the representation of other numbers (all rationals, some irrationals and some imaginary numbers (Knuth 1969b)) by means of arithmetic. We shall limit ourselves to rational numbers, and give a finite representation of them that is implementable in computer hardware.

3.1. Addition, subtraction and multiplication. For the addition and subtraction algorithms, we adopt the usual algorithms for positive integers, with one qualification. When subtracting a digit of the subtrahend from a smaller digit of the minuend, rather than "borrow" one from a minuend digit some distance away, we "carry" one to the immediately neighboring digit of the subtrahend. For example,

$$\begin{array}{rccccc}
 & 2 & 7 & 0 & 0 & 4 \\
- & & 3_1 & 5_1 & 2_1 & 6 \\
\hline
 & 2 & 3 & 4 & 7 & 8
\end{array}$$

beginning at the right, we subtract 6 from 4 to get 8 with a carry; then, rather than subtracting the 2 in the next position, we subtract 3 from 0 to get 7 with a carry; etc. Whether we "carry" or "borrow" is inconsequential; the important point is that we affect only the immediately neighboring digit, not a digit an arbitrary distance away. This form of subtraction is usual in circuit design; it is crucial to our number representation.

We now construct minus-one by subtracting one from zero.

$$\begin{array}{rcccc}
\cdots & 0 & 0 & 0 & 0 \\
-\cdots & 0_1 & 0_1 & 0_1 & 1 \\
\hline
\cdots & 9 & 9 & 9 & 9
\end{array}$$

Beginning at the right, the subtraction algorithm generates a sequence of 9s. Though the sequence is unending, it is repetitive, and can be specified finitely. We shall use a quote (quotation mark) to mean that the digit(s) to its left is (are) to be repeated indefinitely to the left. Twenty-five, for example, is represented by 0'25 or by 0'025 or by 00'25. The first of these is called "normalized"; in general, a representation is normalized when it is as short as possible. A table of normalized representations of integers follows; each entry specifies the usual representation together with our representation.

		0:	0'
1:	0'1	-1:	9'
2:	0'2	-2:	9'8
3:	0'3	-3:	9'7
:		:	
9:	0'9	-9:	9'1
10:	0'10	-10:	9'0
11:	0'11	-11:	9'89
:		:	

For brevity on the written page (the abbreviation is inapplicable in computer memories), we make the convention that when no quote appears in a number, 0' is assumed to be appended to its left. Thus the positive integers take their familiar form.

The reader may verify, with examples, that the addition and subtraction algorithms work consistently throughout the scheme. This is no surprise to anyone familiar with radix complement arithmetic, for the similarity is apparent. The rule for negation is the radix complement rule: complement each digit (in decimal, change 0 to 9, 1 to 8, 2 to 7, etc.) and then add one. (Negation can be performed more simply; for example, in binary starting at the right, leave trailing 0s and rightmost 1 alone, flip the rest.) The definition of radix complement is made in terms of a fixed-size space available for storing numbers in computers (the "word"); for example, if w bits are available for encoding integers, two's complement is defined by performing arithmetic modulo 2^w. We prefer to define it independent of space constraints, and to view the fixed-size scheme as a truncation of *p*-adic numbers to *w* bits. Radix complement is often preferred in computers because of its nice arithmetic properties; our finite representation of *p*-adic rationals enjoys these properties without suffering from the wasted space and overflow problems inherent in a fixed-length representation.

The usual multiplication algorithm need not be altered for use with negative integers. For example, multiplying minus-two by three, either way round, yields the desired result.

```
    9'8               3
×     3          × 9'8
    ---            ----
    9'4            2 4
                 2 7
               2 7
                :
                :
              -------
                   9'4
```

3.2 Division. Before we present the division algorithm, let us look at a simple example. Since 9' represents minus-one, we may expect division by 3 to give 3' as the representation of minus-one-third. Let us test the consistency of this representation with our arithmetic. To negate, we first complement, obtaining 6' (since 6' is double 3', it represents minus-two-thirds); then we add one, obtaining 6'7 as our representation of one-third. Subtracting 3' from 0' also produces 6'7. Multiplying 6'7 by 3 yields 1, confirming its consistency.

```
               0'-1 = 9'  (minus-one)
               9'÷3 = 3'  (minus-one-third)
complement of 3' = 6'
or             3'×2 = 6'  (minus-two-thirds)
               6'+1 = 6'7 (one-third)
or             0'-3'= 6'7
              6'7×3 = 1
```

We now present the division algorithm by means of an example. For the moment, we restrict ourselves to integer dividends and divisors, and further to divisors whose rightmost digit is nonzero and relatively prime to the base. (Two integers are relatively prime if and only if their only common divisor is 1. In decimal, the divisor's final digit must be 1, 3, 7, or 9.) We shall remove all restrictions shortly.

For our example, we divide 191 by 33. The rightmost digit of the result, d_0, when multiplied by the rightmost digit of the divisor, 3, must produce a number whose rightmost digit is the rightmost digit of the dividend, 1. Our restrictions ensure that there is exactly one such digit: $d_0 \times 3$ must end with 1, therefore $d_0=7$. We subtract 7×33 from 191; ignoring the rightmost digit, which must be 0, we use the difference in place of the dividend, and repeat.

```
      1 9 1 ÷ 3 3 = 1 2'7
    - 2 3 1
      9'6
    - 6 6
    9'3
  - 3 3
  9'6
```

When the difference is one we have seen before, we can insert the quote, and stop.

Like multiplication, the division algorithm produces each digit by table look-up. Compare this with the left-to-right division algorithm: a "hand calculation" requires a guess that may need to be revised; a "machine calculation" requires repeated subtraction to produce each digit of the result.

The right-to-left division algorithm is well-defined only when the divisor's final digit is relatively prime to the base. For example, an attempt to divide 1 by 2 in decimal fails because no multiple of 2 has 1 as its final digit. Therefore, the means presented so far enable us to represent only a subset of the rationals: those rationals such that, in lowest terms, the denominator has no factors that are factors of the base. Compare this with the fixed/floating-point representation; it represents those rationals such that, in lowest terms, the denominator has only factors that are factors of the base. The two representations are, in a sense, complementary; combining them, we are able to represent all rationals. We allow a radix point, whose position is independent of the quote, or an exponent, as in scientific notation. For example,

```
 12'34÷10 = 12'3.4
12'3.4÷10 = 12!34
 12!34÷10 = 1.2'34
1.2'34÷10 = .12'34
.12'34÷10 = .21'234
```

The last example is probably best written with an exponent, and suggests that the more appropriate machine representation uses an exponent rather than a radix point. This requires a normalization rule: when normalized, the mantissa's rightmost digit is not 0. Thus 12'300E2 becomes 12'3E4, and 120'E2 becomes 012'E3. All rationals except zero have a unique normalized representation.

To divide two arbitrary integers in decimal, we must first "cast out" all 2 and 5 factors from the divisor. These factors can be determined, one at a time, by inspecting the divisor's final digit. A 2 is cast out by multiplying both dividend and divisor by 5; similarly, a 5 is cast out with a multiplication by 2. The generalization to arbitrary (rational) operands is now easy.

4. Binary is beautiful. Base two, a common choice in computers for circuit reasons, has two important advantages for us. First, it is a prime number. Since it has no factors, there is no "casting out"; division of two normalized operands can always proceed directly. Second, multiplication and division tables are trivial. This advantage for multiplication is well-known: the right-to-left division algorithm is a true analogue of multiplication, so the same advantage applies. At each step, the rightmost bit of the dividend remaining

(a) is the next bit of the result, and

(b) specifies whether or not to subtract the divisor.
For example, dividing 1 (one) by 11 (three) proceed as follows.

```
        1÷1 1 = 0 1'1
    - 1 1
    ─────
      1'
  - 1 1
  ─────
  1'0
-   0
─────
  1'
```

1. Dividend = 1. Its rightmost bit = 1. Therefore
   (a) Rightmost bit of result = 1.
   (b) Subtract divisor, and ignore rightmost 0.
2. Dividend remaining = 1'. Its rightmost bit = 1. Therefore
   (a) Next bit of result = 1.
   (b) Subtract divisor, and ignore rightmost 0.
3. Dividend remaining = 1'0. Its rightmost bit = 0. Therefore
   (a) Next bit of result = 0.
   (b) Subtract 0; ignore rightmost 0.
4. Dividend remaining = 1' as in step 2. Therefore insert quote after 2 bits. Result = 01'1.

5. Radix conversion algorithm. The algorithm for converting the representation of a positive integer from one base to another is well-known. The given integer is repeatedly divided by the base of the new representation; the sequence of remainders gives, from right to left, the digits of the new representation. For example, converting eleven from decimal to binary

```
11 = 2×5+1
 5 = 2×2+1
 2 = 2×1+0
 1 = 2×0+1
```

gives 1011.The algorithm is usually considered to terminate when the quotient is zero; if it is allowed to continue, an endless sequence of 0s is produced.

Let us apply the algorithm to a negative integer. For example, converting minus-eleven from decimal to binary

```
-11 = 2×-6 + 1
 -6 = 2×-3 + 0
 -3 = 2×-2 + 1
 -2 = 2×-1 + 0
 -1 = 2×-1 + 1
 -1 = 2×-1 + 1
```

gives · · · 1110101, which is the *p*-adic form. Note that the remainders must be digits in the new base. By recognizing a repeated state of the computation, we are able to insert a quote and arrive at our finite representation 1'0101.

If we begin with a fraction, we must allow the quotients to be fractions. We then need an extra criterion to make the algorithm deterministic. For example, converting one-third to binary, we might begin either with

```
1/3 = 2× -1/3 + 1
```

or with

```
1/3 = 2× 1/6 + 0
```

To produce a result without a radix point, the original fraction, and each quotient fraction, in lowest terms, must have denominators that are relatively prime to the new base. The implied criterion is that the denominator must remain unchanged.

```
 1/3 = 2× -1/3 +1
-1/3 = 2× -2/3 +1
-2/3 = 2× -1/3 +0
```

When a quotient is obtained that has appeared before, a cycle is recognized. Placing the quote accordingly, we have produced the binary representation of one-third, namely 01'1. Thus the usual radix conversion algorithm naturally produces our representation.

In the preceding examples, we began with numbers in their familiar decimal representations: sign-and-magnitude, numerator/denominator. If we begin instead with our representation, the algorithm remains the same for rationals (without radix points) as for positive integers. For example, beginning with 6'7 (one-third in decimal), we see that the rightmost digit is odd, therefore the rightmost binary digit is 1.

```
(6'7-1)÷2 = 3'
 (3'-1)÷2 = 6'
 (6'-0)÷2 = 3'
```

The division by two is performed as a multiplication by five, discarding the rightmost digit of the result, which is 0. In general, when converting from base *b1* to base *b2*, a finite number of rightmost digits of each quotient is sufficient to determine the next digit of the result if all factors of *b2* are factors of *b1*.

6. Properties of the proposed number system. Excluding the radix point or exponent, the general form of our representation is

$$d_{n+m}d_{n+m-1}\cdots d_{n+1}{}'d_nd_{n-1}\cdots d_2d_1d_0$$

The number represented is

$$\sum_{i=0}^{n} d_ib^i - \sum_{i=n+1}^{n+m} d_ib^i/(b^m-1)$$

where *b* is the base of the representation. To justify this formula, we shall break the digit sequence into two parts: the digits to the left of the quote will be called the “negative part”, and the digits to its right will be called the “positive part”. The positive part was our starting point for the construction of the representation.

$$d_n \cdots d_0 = \sum_{i=0}^{n} d_i b^i$$

The negative part can be found as follows.

$$\begin{aligned} d_{n+m} d_{n+m-1} \cdots d_{n+1}' &= d_{n+m} d_{n+m-1} \cdots d_{n+1}'0 \cdots 0 + d_{n+m} d_{n+m-1} \cdots d_{n+1} \\ &= d_{n+m} d_{n+m-1} \cdots d_{n+1}' \times b^m + \sum_{i=n+1}^{n+m} d_i b^{i-n-1} \end{aligned}$$

Therefore

$$d_{n+m} \cdots d_{n+1}' = -\sum_{i=n+1}^{n+m} d_i b^{i-n-1} / (b^m - 1)$$

Putting the positive and negative parts together, we find

$$d_{n+m} \cdots d_{n+1}' d_n \cdots d_0 = (d_{n+m} \cdots d_{n+1}') \times b^{n+1} + (d_n \cdots d_0)$$

and hence we obtain the above formula.

From the formula, we see that sign determination is trivial. If both positive and negative parts are present, we merely compare their leading digits. Assuming the representation to be normalized, $d_{n+m} \neq d_n$.

If $d_{n+m} < d_n$, the number is positive.

If $d_{n+m} > d_n$, the number is negative.

If one part is absent, the sign is given by the part that is present. If both parts are absent, the number is zero.

The formula can be used to convert from our representation to numerator/denominator representation, if one so desires. For example, in decimal 12'7 = 7–120/99=191/33; in binary 01'1=1–010/11=1/11 (one-third). From this formula, we can also derive an easy method for converting to a right-repeating decimal expansion (or binary expansion). Simply subtract the negative part from the positive part with their leading digits aligned and the negative part repeated indefinitely to the right.

$$\begin{gathered} 12'345 = 345 - 121.\overline{21} = 223.\overline{78} \\ 123'45 = 45 - 12.\overline{312} = 32.\overline{687} \\ 43'21 = 21 - 43.\overline{43} = -22.\overline{43} \end{gathered}$$

The opposite conversion can be performed by reversing the steps.

$$2.\overline{34} = 2 - 34' = 56'8$$

The above paragraphs suggest two comparison algorithms. The first is to subtract the comparands, then determine the sign of the result. The second is to convert the comparands to right-repeating form, then perform the usual digit-by-digit comparison. The first has the advantage that it is a right-to-left algorithm, but the second may have an efficiency advantage.

7. Length of representation. When two *p*-adic rational numbers are added, subtracted, multiplied, or divided, the result is an infinite but eventually repeating sequence of digits. For termination of the arithmetic algorithms, and for finite representation of the result (placing the quote), one must be able to recognize when the state of the computation is one that has occurred before. In a hand calculation, recognition poses no problem: one merely scans the page. The analogous approach for machines requires the arithmetic unit to contain some associative memory. Each state of the computation is compared (associatively, in parallel) with all stored states; if a match is found, the operation is complete, otherwise the state is stored and the operation continues. The resulting digit sequence must be checked for normalization.

A method of recognizing the repeated state that requires storing only one state, at the expense of possibly delaying recognition by a few digits, is the following (Brent 1978). If the states are S_1, S_2, S_3,· · ·, then test (S_1 vs. S_2), (S_2, vs. S_3, S_4), (S_4 vs. S_5,· · ·,S_8), etc., until a match is found. Again, the resulting digit sequence must be normalized.

7.1. Length Bounds. An alternative method of implementing the arithmetic algorithms is as follows:

(a) calculate a bound for the length (number of digits) of the result;
(b) calculate a correct, though not necessarily minimal, length for the negative (repeating) part;
(c) produce a sufficient number of digits to ensure that a repetition of the length calculated in (b) has occurred;
(d) normalize the result.

This approach obviates the need to recognize a repeated state. However, the complexity of our formulae for the length bounds makes the approach unattractive.

The bounds are summarized in Table 1. The notation used in the table is the following.
b: the base of the representation
p_i: the length of the positive part (i.e. the number of digits to the right of the quote) in the base *p* representation of x_i
n_i: the length of the negative part (i.e. the number of digits to the left of the quote) in the base *p* representation of x_i
∅: Euler's ∅ function; ∅(*m*) is defined, for positive integer *m*, as the number of positive integers not exceeding *m* that are relatively prime to *m*.

Table 1
Length Bounds

$x_3=x_1+x_2$	$p_3\le \mathrm{MAX}(p_1,p_2)+n_3+2$
$x_3=x_1-x_2$	n_3 is a divisor of $\mathrm{LCM}(n_1,n_2)$
$x_3=x_1\times x_2$	$p_3\le p_1+p_2+n_3+1$
	n_3 is a divisor of $\mathrm{LCM}(n_1,n_2)$
$x_3=x_1\div x_2$	$p_3\le p_1+p_2+n_3+1$ if $p_2\le n_2$
	$p_3\le p_1-p_2+n_3+1$ if $n_2<p_2\le p_1$
	$p_3\le n_3+1$ if $p_1,n_2<p_2$
	n_3 is a divisor of $\mathrm{LCM}(n_1,\emptyset(x_2(b^{n_2}-1)))$

7.2. Approximation. Our representation provides a service that is unavailable to users of fixed-length floating-point hardware: exact results. As arithmetic operators are applied repeatedly during a computation, this extra service may begin to cost extra: the length of representation of the results may tend to grow. As the lengths grow, storage costs increase; as the lengths become greater than the number of digits that a data path or arithmetic unit can accommodate at one time, processing time increases. Many users do not require perfect accuracy, and are unwilling to pay extra for it.

Users of fixed-length floating-point hardware have their accuracy and expense chosen for them by the computer designer. The designer's impossible task is to choose one accuracy and expense (amount of storage per number) to satisfy all users at all times. Some designers give their users a choice of two accuracies (single and double precision); it seems preferable to allow each user to choose any accuracy, from none to complete.

For reasons of mathematical cleanliness, we prefer not to provide approximate versions of the addition, subtraction, multiplication, and division operators, but to provide a separate approximation operator. Two possible (and ideal) forms of this operator are:

(a) Given a number n, and a number of digits d, $n@d$ is a number that is closest to n and whose representation has no more than d digits.
(b) Given a number n, and a tolerance t, $n@t$ is a number in the range $n \pm nt$ whose representation is shortest.

In either form, the problem of finding a “closest” or “shortest” result is a hard one; in practice, a reasonably close or short result is easy to obtain, and acceptable. One method is to convert the number to right-repeating form (using the algorithm in §6) and then truncate the result. For example, 12'34.567 is first converted to $22.445\overline{78}$ and this may be approximated to 0'22.4458, to 0'22.436, or to 0'22.45, etc. In general, if the final truncated result has k digits to the right of the radix point, then it is easy to show that the error introduced by the approximation is less than b^{-k} where b is the base of the number system. This form of approximation corresponds to the first style of approximation operator given above.

The role of the numerical analyst has traditionally been to analyze the accuracy (error) provided by the manufacturer, and to design algorithms which make the best use of this accuracy (i.e., whose final result is most accurate). With our proposal, the numerical analyst's role will be to choose approximations that make a computation as cheap as possible, while achieving the desired accuracy. Correctness proofs will be facilitated by making the chosen approximations explicit.

7.3. Compactness. We would like to compare the computing expense of solving a sample of numerical problems using a fixed-length floating-point representation with the expense using our representation. To make the comparison, it would be unfair to use existing programs, since they are designed to run on existing floating-point machines. We should proceed as follows.

1. Ascertain the accuracy achieved by each of the existing programs for floating-point.
2. Write programs for our representation that achieve the same accuracy as cheaply as possible. This requires numerical analysis that has not yet been developed.
3. Compare the cost of running the programs on their respective machines.

Even then, the comparison will be biased in favor of fixed-length floating-point, since its users are not given the option of more or less accuracy. Needless to say, we have not made the desired comparison.

We have, however, tested our representation on integer data (Hehner 1976). Our sample was a large, well-known compiler (XCOM, the compiler for XPL (McKeeman 1970)). The integer constants in the program require, on average, 5.7 bits each. A complete trace of the values of all integer variables and integer-valued array elements during an execution of the program (compiling itself) revealed that they require, on average, 6.4 bits each. There are some essential overhead costs associated with the use of variable-length data items. To be easily accessible, the data items may need to be addressed indirectly. Furthermore, memory compactions may be necessary with data items dynamically changing in size. Even when these costs are taken into account, there is a 31% saving compared to a 32-bit fixed-length encoding. (The cost is measured as a space-time product, since space is being traded for time when using a variable-length encoding.) Therefore, for this data, our representation is significantly more economical than a well-known standard.

8. Conclusion. The representation of the rational numbers presented in this paper has several appealing properties. Given the usual representation of the positive integers, it is, in a genuine sense, the natural extension to the rationals. The algorithms for addition, subtraction, and multiplication are those of the usual integer arithmetic; the division algorithm is truly the analogue of multiplication.

The complexity of rational arithmetic in numerator/denominator form has not been reduced by our representation, but it has been redistributed and its character has changed. For humans, the problems of detecting a repeated state and normalization have the character of a pattern-match. If they can be solved economically for computers, then we believe rational arithmetic with programmer-controlled accuracy will become an attractive proposition.

The notation introduced in this paper may be generalized in several directions. For integer base $b > 1$, the digit set can represent any b integers that are all different modulo b (balanced ternary is a special case). The base need not be a positive integer. And the representation can be used for rational power series. Krishnamurthy has considered the use of a fixed-length truncation of p-adic numbers with the result that arithmetic is exact within a limited range (Krishnamurthy 1977).

Acknowledgments. We thank Art Sedgwick and Bill McKeeman for their interest in and comments on this work. We especially thank Don Knuth for informing us of previous related work.

REFERENCES

A. AVIZIENIS (1971), *Digital Computer Arithmetic: A Unified Algorithmic Specification*, Proc. Symposium on Computers and Automata. Polytechnic Press. Polytechnic Institute. Brooklyn, NY.

R. BRENT (1978), communicated via D. E. Knuth.

EDSGER W. DIJKSTRA (1977), *An interview with prof. dr. Edsger W. Dijkstra*, Datamation, 23. no. 5, p. 164.

E. C. R. HEHNER (1976). *Computer design to minimize memory requirements*, Computer, 9. no. 8. pp. 65-70.

K. HENSEL (1908). *Theorie der algebraischen Zahlen*, Leipzig-Berlin.

— (1913), *Zahlentheorie*, Berlin-Leipzig.

B. K. P. HORN (1977), *Rational Arithmetic for Minicomputers*, MIT Press, Cambridge, MA.

D. E. KNUTH (1960), *An Imaginary Number System*, Comm. ACM. 3, pp. 245-247.

— (1969a). *Seminumerical algorithms*, The Art of Computer Programming, vol. 2. Addison-Wesley, Reading, MA.

— (1969b). Ibid., § 4.1. ex. 31, p. 179.

E. V. KRISHNAMURTHY (1977), *Matrix processors using p-adic arithmetic for exact linear computations*, IEEE Trans. Computers, C-26. pp. 633-639.

W. M. MCKEEMAN, J. J. HORNING AND D. B. WORTMAN (1970), *A Compiler Generator*, Prentice-Hall, Englewood Cliffs. NJ.

G. F. SONGSTER (1963), *Negative base number representation systems*, IEEE Trans. Computers. EC-l 2, pp. 274-577.

W. T. WILNER (1975). *Design of the B1700*, Proceedings of AFIPS 1972, FJCC vol. 41, AFIPS Press, Montvale, NJ, pp. 489-497.

This paper is published in *SIAM J. Comput.* 8 no.2 pp.124-134, May 1979.

Reference added later

Another paper on the same topic: Number Representation

Appendix added later

Comparison with Other Representations

There are two representations of the rational numbers in common use. One uses a sign (+ or –), followed by a nonnegative integer (numerator), followed by a division symbol, followed by a positive integer (denominator). For example, –58/2975 . (If no sign is written, the sign is + .) The other is a sign followed by a sequence of digits, with a radix point (called a decimal point in base ten) somewhere in the sequence, and an overscore over one or more of the rightmost digits. For example, $-0.02\overline{34}$. (There are alternative notations to the overscore.) The overscore can be thought of as saying that the digits beneath it are repeated forever to the right. In the example, that's –0.023434343434... . Quote notation does not need a sign; it has a sequence of digits with a radix point somewhere in the sequence, and a quote somewhere in the sequence. For example, 4.3'2 . The quote can be thought of as saying that the digits to its left are repeated forever to the left. In the example, that's ...43434343434.32 . All three examples in this paragraph represent the same rational number.

$$-58/2975 = -0.02\overline{34} = 4.3'2$$

The three representations can be compared in two ways: space required for storage, and time required for arithmetic operations. We begin with space. Quote notation and overscore notation require essentially the same space. But quote notation and numerator-denominator notation can differ greatly. The worst case occurs for some prime denominators (see Fermat's little theorem). For example, +1/7 = 285714'3 (in binary it is 011'1). To represent +1/947 in binary as a sign and numerator and denominator requires 12 bits, and as quote notation requires 947 bits. (Extra bits are required to delimit two variable-length numbers, but these are the same for all three representations, so ignoring them does not affect the comparison.)

On average, the space requirements for numerator-denominator and quote notations are not very different. The 180,000 shortest numerator-denominator representations require 15.65 bits on average, and those same numbers in quote notation require 39.48 bits on average. Taking the shortest numerator-denominator numbers, and then translating those numbers to quote notation, results in a biased comparison in favor of numerator-denominator. If we take all binary quote representations up to and including 14 bits (all quote positions and all radix point positions), then discard those that are not normalized, we have 1,551,567 numbers requiring 13.26 bits on average. If we translate them to numerator-denominator notation, then normalize the result by removing common factors, they require 26.48 bits on average. This comparison is biased in favor of quote notation. An unbiased comparison would seem to favor numerator-denominator, but not by much.

We now look at the time comparison. To add two numbers in numerator-denominator notation, for example $(+a/b) + (–c/d)$, requires the following steps.
• sign comparison to determine if we will be adding or subtracting; in our example, the signs differ so we will be subtracting
• then 3 multiplications; in our example, $a×d$, $b×c$, $b×d$
• then, if we are subtracting, a comparison of $a×d$ to $b×c$ to determine which is subtrahend and which is minuend, and what is the sign of the result; let's say $a×d < b×c$ so the sign will be –
• then the addition or subtraction; $b×c – a×d$ and we have $–(b×c – a×d)/(b×d)$
• finding the greatest common divisor of the new numerator and denominator
• dividing numerator and denominator by their greatest common divisor to obtain a normalized result
Normalizing the result is not necessary for correctness, but without it, the space requirements quickly grow during a sequence of operations. Subtraction is almost identical to addition.

Adding two numbers in overscore notation is problematic because there is no right end to start at. The easiest way to do the addition is to translate the numbers to quote notation, then add, then translate back. Likewise for subtraction.

To add two numbers in quote notation, just add them the same way you add two positive integers. The repetition is recognized when the repeating parts of the two operands return to their starting digits. Then the result may be roll-normalized by checking whether the first digit equals the first digit after the quote. Likewise for subtraction. For both addition and subtraction, quote notation is superior to the other two notations.

Multiplication in numerator-denominator notation is two integer multiplications, finding a greatest common divisor, and then two divisions. Multiplication in overscore notation is problematic for the same reason that addition is. Multiplication in quote notation proceeds exactly like positive integer multiplication, comparing each new sum to previous sums in order to detect the repetition. For multiplication, quote notation is superior to overscore notation, and may be slightly better than numerator-denominator notation.

Division in numerator-denominator notation has the same complexity as multiplication in numerator-denominator notation. Division in overscore notation is problematic because it requires a sequence of subtractions, which are problematic in overscore notation. Division in quote notation proceeds just like multiplication in quote notation, producing the answer digits from right to left, each one determined by the rightmost digit of the current difference and divisor (trivial in binary). For division, quote notation is superior to both overscore and numerator-denominator notations.

It should not be surprising that arithmetic is easier in quote notation; that was the criterion of its design.

Appendix added later

Here is an example multiplication in detail, with line numbers for reference in the explanation below: `1'01 × 1'0 = 110` .

```
0                       0'
1                     + 0'
2                       0'
3               + 1'0 1
4                 1'0 1
5             + 1'0 1
6             1'0 1 1
7           + 1'0 1
8           1'0 1 0
9         + 1'0 1
10        1'0 1 0
```

The multiplicand is `1'01` . The multiplier is `1'0` and we will be examining its bits from right to left. On line 0 start with `0'` because you are accumulating a sum. The rightmost bit of the multiplier is `0` so add `0'` on line 1 to obtain the sum `0'` on line 2. The next bit of the multiplier is `1` so add the multiplicand shifted left on line 3 to obtain the sum on line 4 but don't bother to write the rightmost bit of the sum. The next bit of the multiplier is `1` so add the multiplicand shifted left on line 5 to obtain the sum on line 6 but don't bother to write the rightmost bit of the sum. The next bit of the multiplier is `1` so add the multiplicand shifted left on line 7 to obtain the sum on line 8 but don't bother to write the rightmost bit of the sum. The next bit of the multiplier is `1` so add the multiplicand shifted left on line 9 to obtain the sum on line 10 but don't bother to write the rightmost bit of the sum. Each time you obtain a sum, check to see if you have obtained that sum before. The sum on line 10 is the same as the sum on line 8, so you're done. The answer, from right to left, is the rightmost bits of the sums. Line 2 gives `0` , line 4 gives `1` , line 6 gives `1` , and line 8 gives `0` . So that's `0110` . Line 10 was a repetition of the immediately preceding sum, so the quote mark is one digit from the left. That's `0'110` . When a number starts with `0'` , we usually don't bother to write it, so the answer is `110` .

Now let's do the same example the other way round: `1'0 × 1'01 = 110` .

```
0                       0'
1                   + 1'0
2                     1'0
3                   + 0'
4                     1'
5               + 1'0
6               1'0 1
7             + 1'0
8             1'0 0
9           + 1'0
10          1'0 0
```

The multiplicand is `1'0` . The multiplier is `1'01` and we will be examining its bits from right to left. On line 0 start with `0'` . The rightmost bit of the multiplier `1'01` is `1` so add the multiplicand `1'0` on line 1 to obtain the sum `1'0` on line 2. The next bit of the multiplier is `0` so add `0'` shifted left on line 3 to obtain the sum on line 4 but don't bother to write the rightmost bit of the sum. The next bit of the multiplier is `1` so add the multiplicand shifted left on line 5 to obtain the sum on line 6 but don't bother to write the rightmost bit of the sum. The next bit of the multiplier is `1` so add the multiplicand shifted left on line 7 to obtain the sum on line 8 but don't bother to write the rightmost bit of the sum. The next bit of the multiplier is `1` so add the multiplicand shifted left on line 9 to obtain the sum on line 10 but don't bother to write the rightmost bit of the sum. Each time you obtain a sum, check to see if you have obtained that sum before. The sum on line 10 is the same as the sum on line 8, so you're done. The answer, from right to left, is the rightmost bits of the sums. Line 2 gives `0` , line 4 gives `1` , line 6 gives `1` , and line 8 gives `0` . So that's `0110` . Line 10 was a repetition of the immediately preceding sum, so the quote mark is one digit from the left. That's `0'110` , or `110` .

Appendix added later

Here is a worst-case calculation, in which the result is maximal length for the lengths of the operands: 1 / 59 in binary is 59 bits long.

1/111011 = 0111110111010100100111000011010000010001010110110001111001'1

The bottom line is identical, except for the final 0, to the very first difference. That tells us where to put the quote. The entire calculation is 30 subtractions, and that's how many 1s are in the answer.

```
                                                                                                                             1
                                                                                                                   1 1 1 0 1 1
                                                                                                                 1’0 0 0 1 1
                                                                                                                 1 1 1 0 1 1
                                                                                                             1’0 1 0 1 0 0
                                                                                                           1 1 1 0 1 1
                                                                                                       1’0 1 1 1 0 1
                                                                                                         1 1 1 0 1 1
                                                                                                     1’0 1 0 0 0 1
                                                                                                       1 1 1 0 1 1
                                                                                                   1’0 0 1 0 1 1
                                                                                                     1 1 1 0 1 1
                                                                                                 1’0 0 1 0 0 0
                                                                                             1 1 1 0 1 1
                                                                                         1’0 1 1 1 1 1
                                                                                           1 1 1 0 1 1
                                                                                       1’0 1 0 0 1 0
                                                                                       1 1 1 0 1 1
                                                                                   1’0 1 0 1 1 1
                                                                                       1 1 1 0 1
                                                                                   1’0 0 1 1 1 0
                                                                                   1 1 1 0 1 1
                                                                               1’0 1 0 1 1 0
                                                                               1 1 1 0 1 1
                                                                           1’0 1 1 0 0 0
                                                                         1 1 1 0 1
                                                                     1’0 0 0 0 0
                                                            1 1 1 0 1 1
                                                           1’0 0 0 1 0
                                                        1 1 1 0 1 1
                                                    1’0 1 1 0 1 1
                                                      1 1 1 0 1 1
                                                  1’0 1 0 0 0 0
                                            1 1 1 0 1 1
                                          1’0 0 0 0 1
                                          1 1 1 0 1 1
                                      1’0 1 0 0 1 1
                                        1 1 1 0 1 1
                                    1’0 0 1 1 0 0
                                  1 1 1 0 1 1
                              1’0 1 1 1 0 0
                            1 1 1 0 1 1
                        1’0 1 1 1 1 0
                        1 1 1 0 1 1
                    1’0 1 1 0 1 0
                    1 1 1 0 1 1
                1’0 1 1 0 0 1
                  1 1 1 0 1 1
              1’0 0 1 1 1 1
                1 1 1 0 1 1
            1’0 0 1 0 1 0
            1 1 1 0 1 1
        1’0 1 0 1 0 1
          1 1 1 0 1 1
      1’0 0 1 1 0 1
        1 1 1 0 1 1
    1’0 0 1 0 0 1
      1 1 1 0 1 1
  1’0 0 0 1 1 1
    1 1 1 0 1 1
1’0 0 0 1 1 0
```