Colin Runciman
Department of Computer Science, University of York
Heslington, YORK Y01 5DD
United Kingdom

ABSTRACT

A prime concern in the design of any general purpose programming language should be the ease and safety of working with natural numbers, particularly in conjunction with discrete data structures. This theme of commitment to the naturals as the basic numeric data type is explored in the context of a lazy functional language.

Non-Title Keywords: *structural correspondence, numeric types, total functions, closed systems, functional programming, lazy evaluation.*

1. Introduction

Some thirty years into the history of machine-independent programming language design, the treatment of numbers is still problematic. Language designers *worry* about numbers. Should there be just one numeric type or many? Should there be subtypes? What about complex numbers? What about precision and implementation cost? What about translation of existing programs written in other languages? And so the list goes on.

This paper does not propose answers to any of the above questions (though they are all important). Instead it asks yet another question. What about the *natural* numbers? In a great deal of programming, the more elaborate number systems – integers, floats, complexes and the like – are rarely, if ever, used. The natural numbers, on the other hand, get used almost all the time. As a first requirement, therefore, a programming language should make it easy to work with the natural numbers. Depending on the programming application, there may or may not be further numeric requirements relating to other number systems.

It seems that this requirement for the natural numbers is habitually overlooked by language designers. There are concessions towards it – for example the introduction of distinct type names such as `unsigned` in C[10] or `cardinal` in Modula 2.[18] But these fall short of a full commitment to the support of natural number programming.

The conventional approximation of *real* numbers, using the floating point representation, as adopted in most programming languages, can lead to errors of serious proportions for some patterns of computation. Others in pursuit of numeric ideals in programming have addressed the very difficult issue of how to provide a true implementation of real numbers in the usual mathematical sense.[3, 16] In many ways their ideal is the same as ours: programming with the exact number system that is needed for the problem in hand. Whereas the reals precisely meet the need to represent continuously varying quantities found in some computer applications, it will be argued here that the naturals are precisely suited to represent characteristics of the discrete structures which inhabit the internal world of programming.

The discussion in this paper relates particularly to purely functional programming languages in the recursion equation style. However, similar arguments can also be applied to other forms of programming language. For a brief introduction to languages of the kind used here the reader is referred to papers by Turner.[15, 13] For a more extensive account, see the book by Bird & Wadler.[1]

2. Structure and properties of data types

An important theme in subsequent discussions will be that the treatment of numbers ought to match the treatment of other data types. This section therefore reviews principles about the nature of data types that have become widely accepted (particularly among functional programming language designers) since the original work of Milner on the type system of ML.[12]

2.1. Construction

It is generally agreed that values of data types should be the results of applying a small number of *constructors*. To build types with a large set of values, recursive type constructions are used. The classic example, deriving from the original LISP design,[11] is the `list` type.

```
list a = []
       | a : list a
```

This type equation describes the possible constructions of a list of items of some type `a`. There are two constructors. The constant `[]` is the constructor for the empty list. The infix operator `:` is the functional constructor for non-empty lists; its left argument is the first item in the constructed list and its right argument is the list of items thereafter. Construction by `[]` or by `:` are the only possibilities for a well-defined list, and they are mutually exclusive. Also, lists are *freely* constructed; there are no laws equating one construction with any other, so different constructions produce different lists.

Though free construction defines the unique primitive form of a list, abbreviated notations are often more convenient than explicit compound applications of constructors. For example, instead of

```
'h' : 'e' : 'l' : 'l' : 'o' : []
```

the programmer may write

```
"hello"
```

to express precisely the same thing. It must be stressed that such strings do *not* provide additional constructions; they are are simply a shorthand for free constructions using `:` and `[]`.

2.2. Recursive definition & inductive reasoning

Type definitions such as that for `list` conveniently suggest a natural form of recursive function definition over values of the relevant type. This is assisted by the use of constructor-based patterns to define argument cases. Use of such patterns is illustrated in the following definition of the `length` function which computes the number of items in a list.

```
length [] -> 0
length (x:xs) -> 1 + length xs
```

The two different argument patterns used in this definition are mutually exclusive, so there is no ambiguity about which clause to apply in any given case. For other definitions this may not be so. The convention adopted is that the defining clause relevant to any given function application is the *first* whose left-hand side matches.

For programmers who wish to use formal reasoning, and for implementors of formal support tools, free construction also forms the basis for rigorous case analysis and generic laws.

2.3. Laziness

It is over a decade since the advantages of *lazy construction*,[5] and more generally of *lazy evaluation*,[8] were first pointed out. A lazy evaluator only evaluates expressions so far as is absolutely necessary to yield the information required by their context. Instead of computing values for each argument in a function application before computing the relevant function body, arguments are only evaluated sufficiently to determine which clause of the function definition applies. The consequence of adopting lazy evaluation is that the semantics of the programming language become more generous, in the sense that more programs yield well-defined results. Computations can even be defined over *infinite* data structures.

To illustrate, think of double-spacing text. Assuming that a text is represented as a list of lines, and that a line is represented as a list of characters, the `doublespace` function can be defined as follows.

```
doublespace t -> interleave t blanklines

interleave (x:xs) ys -> x : interleave ys xs
interleave [] ys -> []

blanklines -> [] : blanklines
```

Here `blanklines` is an infinite list. It is lazily evaluated, as part of any `doublespace` application, to provide only as many blank lines as are actually needed for the particular text supplied as argument.

Lazy evaluation not only extends the programmer's repertoire, it also simplifies the business of reasoning about programs. This is because straightforward substitution "by definition" is valid at any point in any expression.[14]

2.4. Numbers: the exception?

The definition and use of the `list` type sets a pattern that can be followed for a rich and varied collection of other data types, from truth values to quad-trees. But when it comes to *numbers* there is some hesitation. Defining a suitable free construction for the more exotic number systems is problematic, to say the least. Therefore numeric types are typically defined, in effect, by a vast array of constant constructors – the numerals.

```
integer = ... | -2 | -1 | 0 | 1 | 2 | ...
```

This is quite unlike all other data types with a large number of values. And for the *natural* numbers it is also quite unnecessary because there is a classic free construction analogous to that for the `list` type.

```
natural = 0
        | natural +1
```

Decimal numerals are suitable short-forms for these constructions (*eg* `4` for `0+1+1+1+1`) just as in an earlier example a string was used to express a character list.

So even before any consideration of how numbers are used in programs, here is a motive for treating the natural numbers as a first class type. They can be defined by a free construction, which leads to the convenience of related recursive definition and inductive reasoning by patterned cases.

3. Natural arithmetic

With numbers comes arithmetic. Programmers' expectations from any general purpose language extend at least to the four basic arithmetic operations $+$, $-$, $\times$ and $\div$ in some form. Some programmers may expect more, such as exponentiation. The result of an arithmetic operation applied to a pair of numbers is, of course, another number. But this is a simplified view. Only a *total* operation guarantees a defined result for all possible arguments. And only a *closed* operation guarantees results that lie within the same number system as arguments.

There are clear advantages for the programmer if arithmetic operators are both total and closed, since the type discipline then provides maximum security against arithmetic failure with minimal complication. One motive for having only a single numeric type is precisely the desire for such a closed system. However, if the usual mathematical definitions of the operators are adopted, this single numeric type must also be universal or all-embracing. For example, division of integers yields rationals, and exponentiation of rationals yields reals or even complexes. So although closing operators over a universal type reduces the risk of *arithmetic* failure, it increases the risk that the ultimate results of arithmetic cannot be interpreted sensibly in their context. A universal number type may be appropriate for calculators (say), where numbers themselves constitute the domain of computation, but not for general purpose programming systems where the context is a world of discrete data structures.

3.1. Integer arithmetic is not closed enough

For this reason (among others) exact integer arithmetic, with closed operations such as division with remainder, is commonly separated in programming languages from the inexact system of "floating" arithmetic. But this attempt to close the integers does not go far enough. Operators such as exponentiation are still a problem (hence its omission from the arithmetic primitives in Pascal,[17] for example). Negative numbers are often alien to the context of integral arithmetic, and the programmer must therefore be constantly on guard against the possibility of negative values arising. Total operators remain elusive in view especially of the notorious divide-by-zero problem.

Real and imaginary data structures

It has already been noted that the context of much arithmetic in programming is computatation over discrete data structures. Correspondences between operations on numbers and operations on data structures are common and important. Often numbers are used to represent information about data structures and their access such as indexes or sizes. Typically, the numbers corresponding to real data structures can only be naturals. A negative number could only correspond to some kind of imaginary structure and in practice represents a mistake leading to undefined or wrong results. So making these structure-linked numbers integers is clearly inappropriate. The correct representation is a natural number. To avoid the latent introduction of negative values, arithmetic must be closed over the naturals. This is a very practical motive for looking at natural arithmetic. The aim is a total closed system of arithmetic with results that can be safely interpreted in the context of the discrete structures in general programming.

Exceptional negatives

Another consideration is that many uses – maybe even *most* uses – of negative integers in programs are really *ab*uses. They represent exceptional cases such as when a number is unknown or beyond some limit. An obvious danger in this kind of numeric encoding is that exception values might accidentally get sucked into arithmetic as though they were meaningful in the same sense as a natural number.

For example, consider the data structures forming the state of an editor which includes line marker registers named `a`, `b`, `c` and so on. A well-defined line marker has bound to it some valid line number. Suppose the programmer implementing the editor decides that the value held in a currently *undefined* register will be `-1`. Then if register `r` is undefined but the user refers to the line `r+3` it would be an all too easy programming slip to accept this as referring to line `2`.

Providing the natural numbers, with full constructive and pattern matching rights, removes the temptation to adopt such a representation, with its attendant pitfalls.

3.2. Natural subtraction

Consider now the four basic arithmetic operations over the naturals. Two of these operations, addition and multiplication, present no difficulties since, defined in the usual way, they are both total and closed. However, the same cannot be said of the other two operations, subtraction and division, each of which poses its own problems. Subtraction is dealt with in this section and division in the next.

The problem of defining total, closed subtraction over the naturals has a standard solution. If the second argument is greater than the first (so that under integer subtraction the result would be negative) the result is simply zero. This natural subtraction operation, which may be defined by the following clauses

```
n ·–· 0 -> n
0 ·–· n -> 0
(n+1) ·–· (m+1) -> n ·–· m
```

occurs in most textbooks dealing with computability. See, for example, Chapter 9 of Harel's book.[7] But since there is no trace of this form of subtraction in current widely-used programming languages, one might be forgiven for concluding that its scope of usefulness is limited to a theoretical context. This is not so. It is exactly what is required for practical programming with the naturals.

The appropriateness of `·–·` because of correspondence with data structure operations can be illustrated by a specific example, namely the `drop` function (commonly defined in preludes or libraries of functional programming languages). The result of `drop n xs` is what's left of list `xs` after the first `n` items have been removed. If the list has fewer than `n` items, the result is `[]`. (It is hardly possible to remove any more than this, barring the introduction of imaginary "antilists".)

```
drop 0 xs -> xs
drop n [] -> []
drop (n+1) (x:xs) -> drop n xs
```

There is an exact analogy between `·–·` and `drop`. The relationship between then can be expressed in the following law.

```
length (drop n xs) = length xs ·–· n
```

Such laws are essential for programming to be a reasoned and accurate process. Even if they are not expressed and appealed to *explicitly* in a formal program derivation; they are certainly relied on and appealed to *implicitly* in the programmer's head.

3.3. Natural Division

As for subtraction, so for division. We wish to find a closed total variant, that coincides with all natural results of integer arithmetic. However, integer division is already closed over the naturals in all cases for which it is defined. The sole problem is the lack of a defined result for divisions by zero.

A double veneer

One approach is to argue along the following lines. Since an undefined result arises from `0`-constructed second arguments, let's conceal integer division beneath a new primitive defined in terms of all and only those divisions with `+1`-constructed second arguments. Calling this new primitive `//`, the preceding specification demands exactly the following definition.

```
x // y -> x / (y+1)
```

Intuitively, `x // y` may be viewed independently of `/` by thinking of sectioning `x` as evenly as possible, cutting at `y` places. The single numeric result of such a division is the length of the smallest section. So `x // 0` yields `x`, since no cuts are made.

Although `//` is a form of division with a well-defined result when the second argument is zero, its other results clearly do not agree with those of `/`. But this agreement can be established by defining natural division `·/·` in terms of `//` and natural subtraction.

```
x ·/· y -> x // (y·–·1)
```

A likeness between `·–·` and this version of `·/·` can be seen if one looks at how their results change as their second arguments change. As the second argument of `·–·` *increases* the corresponding result *decreases* until it reaches zero – at which all subsequent results are fixed since integer subtraction does not yield a natural number. Similarly, as the second argument of `·/·` *decreases* the corresponding result *increases* until it reaches the first argument value.

An infinite result

However, there is a more radical approach to natural division. The principle has already been established that numbers should share advantageous characteristics of other data types. Since there may be infinite data structures, why not infinite numbers? There is exactly one infinite well-formed construction of type `natural`; this we may call `infinity` and define as follows.

```
infinity -> infinity +1
```

If `infinity` is available as a natural value, it becomes an obvious candidate for the result of division by zero, as in the following (re-)definition of `·/·`.

```
x ·/· 0 -> infinity
x ·/· y -> x / y
```

Defined like this, the zero divisor case appears to be an exception to the normal rule of division. To see that this is not really so, let's make an explicit definition of division as counted subtraction.

```
x ·/· y -> if x < y then 0 else ((x - y) ·/· y) +1
```

If a division operation according to this definition is lazily evaluated for a zero second argument then the result obtained is `infinity`.

This alternative solution to division by zero, introducing the possibility of an infinite result, depends on having *lazy* numbers. Potential advantages of lazy/infinite numbers beyond the particular role of totalising division will be considered shortly.

3.4. Natural exponentiation

Although exponentiation is not closed over the integers (because the exponent may be negative), it *is* closed over the naturals. Once again, the computability textbook specifies the function we need in the following primitive recursive definition.

```
pow n 0 -> 1
pow n (p+1) -> n * pow n p
```

4. Lazy numbers

Recall the introduction of `infinity` as the result of zero-divide. Are there any more general attractions and applications for lazy numbers, and for `infinity` in particular? Indeed there are, and the following paragraphs give some examples.

4.1. Infinity in abstract algorithms

In the formulation of many algorithms, an abstract presentation (removed from any particular programming language) often finds a use for infinity. For example, in an algorithm to determine shortest paths in a network, there might be associated with each node the length of the shortest path so far discovered to that node: and these values might be set initially to infinity. Rendering such an abstract algorithm into a specific programming language may be problematic. (In one compiler known to the author the declaration

`infinity=857` appears!) As we have seen in the case of zero-divide, lazy evaluation makes `infinity` a perfectly good value. It may be combined with other numbers to form expressions with useful results: for example, the result of the comparison `n < infinity` is `True` for all finite `n`.

4.2. Extended and simplified validity of laws

In the absence of a lazy construction for numbers, many useful laws have to be hedged about with conditions. For example, seeing a defining clause of the form

```
f (n+1) -> e
```

one might reasonably expect the equivalence law

```
f (n+1) = e
```

to hold. But this equivalence does *not* hold for *all* `n` unless the successor construction `+1` is lazy. Computing the left-hand side using an eager `+1`, the result will be undefined if, say, `n` is an expression whose evaluation does not terminate; whereas `e` may yield a well-defined value.

An earlier section drew attention to the importance of laws of correspondence involving both data structures and numbers. Some such laws only make sense for lazy/infinite data structures if there are also lazy/infinite numbers. So the provision of lazy numbers reduces the need to draw distinctions, when applying laws, between finite and infinite structures.

4.3. Extended use of auxiliary components

The same sort of argument can be applied to programmed components. Various general-purpose auxiliaries (including higher-order functions – the *glue*[9] with which functional components are assembled into programs), have numeric arguments or results. Their usefulness is increased by the introduction of lazy/infinite numbers. This includes (but is not limited to) working with infinite structures in ways that would otherwise be impossible. For example, a function `range lo hi` might be used to generate either finite or infinite series; similarly a function `repeat n f` might be used for indefinite as well as definite iteration.

4.4. More generous semantics

It is an invariant rule, already noted, that lazy evaluation corresponds to a more generous semantics than conventional eager evaluation. Specifically, numeric programs executed under the rules of lazy arithmetic may produce useful results, even though under conventional programming arithmetic they would only go into an infinite loop or stop with a system-level error message. The reverse is never the case.

4.5. Unnecessary computation avoided

As its name implies, *lazy* evaluation is related to the avoidance of work that is not absolutely necessary. For example, consider a natural number computation that reduces to the following form.

complicated `+1 > 0`

One can see that in principle the *complicated* (and therefore expensive) numeric sub-expression need not be evaluated to determine the overall result. Under conventional programmed arithmetic, however, this principle does not actually apply. Under lazy evaluation, it does.

5. Practicality for the programmer

To illustrate some practical consequences of treating natural numbers in the way described above, let's now look at three small programming examples. The first two examples were selected by looking for uses of subtraction and negative values in a small library of function definitions written in a language having integers as the basic numeric type. Most definitions in this library that make use of the minus operator do so in in a recursive defining clause of the following form.

`f .. n .. ->` *something* `f .. (n-1) ..` *something*

In each such case, rewriting the definition in the form

`f .. (n+1) .. ->` *something* `f .. n ..` *something*

avoids the use of subtraction, and no further discussion seems necessary. But two rather more interesting definitions using subtraction are those of `position` and `sublist`.

5.1. Naturalised position function

The `position` function computes the first index position at which a given item appears in a given list. In the integer-based system, it is defined as follows.

```
position a xs ->
    ( pos xs 0
    where pos (x:xs) n -> if a = x then n
                          else pos xs (n+1)
    and   pos [] n -> -1
    )
```

The treatment of the `[]` case clearly belongs to the category of *exceptional negatives.* Under a natural number system the obvious possibility is to omit the second clause for `pos` so the result is undefined instead of -1 in the exceptional case. Better, why not define the `positions` (plural) function that yields the list of *all* positions at which a given item occurs?

```
positions a xs ->
    ( pos xs 0
    where pos (x:xs) n -> if a = x then n : pos xs (n+1)
                          else pos xs (n+1)
    and pos [] n -> []
    )
```

Now the result for a non-occurring item is an unproblematic `[]`.
□

5.2. Naturalised sublist function

The `sublist` function extracts, as a list, those items in a given list at index positions `m` to `n` inclusive for given `m` and `n`. Its integer-based definition follows. (The definition is at the level of functional values, and uses `o` as an infix operator denoting functional composition.)

```
sublist m n -> take (n-m+1) o drop m
```

Under normal circumstances, when both `m` and `n` are valid positions of items in the given list and $m \leq n$, the informal specification given above is satisfied. But what if these conditions do not hold? There are two cases.

$n \geq m-1$ — A well-defined result is still obtained even if one or both of `m` and `n` correspond to positions at or beyond the end of `xs`. The result is the section of `xs` for which item positions lie in the range `m..n` even if this section is empty.

$n < m-1$ — The above empty section rule no longer holds. The result is undefined because `take` is passed a negative argument, when it is only well-defined for naturals.

The problem with the integer-based definition is that it sometimes depends on an imaginary intermediate data structure. Under a natural number system, a small change to the definition suffices to ensure the desired more generous behaviour.

```
sublist m n -> take (n+1·−·m) o drop m
```

However, the following definition might be preferred as a matter of taste.

```
sublist 0 n -> take n
sublist (m+1) n -> take (n·−·m) o drop (m+1)
```

□

Turning to division, most instances of integer division in the above-mentioned library have a constant divisor. This is never negative (nor, of course, zero). Non-constant divisors are typically drawn from an inherently safe and positive source: for example, primes in factorisation or greatest common divisors in simplifying quotients.

A more interesting application of division, also involving further subtraction, concerns the justification of text on a line.

5.3. Naturalised text justification

Let a function `justify` be required to accept two arguments: `n` a number of character spaces available on a line, and `ws` a list of words from which a line is to be formed (where each word is a list of characters). The result should be a list representing the printed line. It should be like `ws` but with *blanks* (lists of spaces) between the words, so that the total number of characters is `n` and the longest and shortest blanks differ by no more than one space.

Since the result contains words and blanks alternately it seems appropriate to consider computing it by the `interleave` function introduced in §2.3. This requires a list of blanks, call it `bs`, to be specified as `interleave`'s second argument. Let `bs` be computed by an auxiliary function `blanks`. What arguments does `blanks` need? It suffices to provide the total number of spaces that must be introduced in the line and the number of inter-word gaps between which those spaces must be divided. Hence the following definition.

```
justify n ws ->
    ( interleave ws bs
    where bs = blanks (n ·−· sumlengths ws) (length ws ·−· 1)
    )
```

This provides two further illustrations of natural subtraction. If there are *no* words (`length ws = 0`) there are no gaps either; whereas under integer subtraction there would be `−1` gap implying an imaginary data structure for the blanks list `bs`. Similarly, if the total length of the words already exceeds the available number of characters, no more characters in the form of spaces are requested. This time the negative result under integer arithmetic might seem preferable, indicating how many *less* characters there should be. However, to put that information to any productive use would require the whole structure of the solution to be abandoned in favour of something more complex. Under the existing simple structure, which does not seem unreasonable to hope for, a negative value here also would demand an imaginary structure as the result of `blanks`.

The `sumlengths` function must sum the lengths of all the words in `ws`. It will not be discussed further as it requires no controversial arithmetic.

Consider rather the auxiliary function `blanks`. This is where division comes in, under the obvious scheme for determining successive blank lengths by dividing the remaining number of spaces to be allotted by the remaining number of gaps. Assuming the veneered form of totalisation for `·/·`, the following definition might be made.

```
blanks 0 0 -> []
blanks n g ->
    ( space s : blanks (n ·−· s) (g ·−· 1)
    where s -> n ·/· g
    )
```

To show that both arguments will reach zero, and that a satisfactory list of blanks will be computed as a result, consider first the recursive application of `blanks`. The first argument will decrease from `n` unless `n = 0` or `s = 0`, and the second argument will decrease from `g` unless `g = 0`. Since at least one of `n`, `g` must be non-zero for the recursive clause to apply the only potential problem is when `n > 0` and `s = g = 0`. But this case cannot arise: totalised division ensures that when `g = 0`, `s = n`. So one or both arguments must decrease. Since negative values cannot occur, it follows that both arguments will indeed reach zero.

The number of blanks in the result of `blanks n g` must equal the number of gaps `g`, and the total number of spaces in these blanks must be `n`. A `[]` result for the base case `n = g = 0` meets this requirement. Each recursive application forms one new blank and *truly* decrements the `g` value accordingly: `g = 1` implies `s = n`, so no subtraction is ever performed for `g = 0`, as the base clause applies instead. Similarly, the first argument is *truly* decremented by the appropriate number of spaces since `n ≥ n ·/· g`.

□

6. Practicality for the Implementor

The discussion in preceding sections has been concerned far more with the *formulation* of programs than with their *execution*. It has considered the needs of programmers, but not those of implementors. To redress the balance a little, let's now briefly consider one or two implementation issues – it is beyond the scope of the present paper to deal with implementation concerns in any depth.

The proposal for *lazy* numbers raises the most obvious questions of efficiency. Modern computing machinery provides fast arithmetic primitives at (or near) the hardware level. If a programming system uses a different form of arithmetic that does not exploit the machine provisions, one might expect to incur a serious performance penalty. For example, an important analogy has been observed between the construction of lists using a lazy operator for item addition, and the construction of numbers using a lazy successor function. But if this analogy were employed to determine the *implementation* of numbers, in conjunction with primitive recursive definitions for $+$, $\times$ *etc*, then arithmetic computations would be very slow indeed!

So we require an implementation scheme that allows machine arithmetic to be exploited where possible. One possibility would be to represent natural numbers as:

(i) a machine number; *or*

(ii) an unevaluated computation; *or*

(iii) a pair `(m,n)` where `m` is a positive machine number, `n` is the representation of another natural, and the pair represents their sum.

Literal numbers in programs would of course be represented using (i) and the implementor would aim to maintain this representation for as much arithmetic as possible. The use of forms (ii) and (iii) would be necessary in some cases to provide correct lazy semantics, but could be kept to a minimum by employing *abstract interpretation*.[6] This rather general technique of semantic analysis is increasingly being used by compiler-writers to determine properties such as *strictness*,[4] indicating contexts in which an expression *must necessarily* be evaluated. In such contexts, representation (ii) can be avoided. Abstract interpretation can also be used to establish the absence of *shared references*, and there are other forms of non-standard semantics[2] that provide yet more powerful tools for the analysis of sharing. Knowledge about unique references would allow the size of composite pairs (iii) to be curtailed. Alternatively, it might be better to restrict the component `n` in (iii) to be an unevaluated computation, collapsing chains of machine numbers into their sum, at the expense of copying partial sums into separate representations.

Using a scheme such as this, lazy numeric computation would be tolerably efficient, and numeric computations revealed by compiler analysis *not* to depend on laziness would be performed by conventional machine arithmetic.

7. Conclusion

A case has been made for the natural numbers as a distinct, freely and recursively constructed type. This leads to the introduction of natural arithmetic, and in particular to forms of total subtraction and division closed over the naturals. This form of arithmetic appears to be quite convenient. There are several benefits of raising the status of natural numbers in this way. For example: type-checkers can do a better job; formal reasoning about programs is facilitated; programs using numbers in computations over data structures become simpler and safer.

The possibility of lazy numbers has also been raised, and a number of points advanced in favour. The benefits of lazy evaluation generally are now widely recognised (though still regarded as controversial by some). However, numeric computing is a rather special province, with elaborate traditional mechanisms tied closely to machine hardware. One therefore expects considerable resistance to any suggestion for lazy naturals to form the basic numeric type for programming. There will be all the questions noted in the introduction. What about complex numbers? What about precision and implementation cost? What about translation of existing programs written in other languages? And so the list goes on.

Acknowledgements

I should like to thank members of the functional programming group at York, including Nigel Jagger in particular, for their comments on an earlier technical memorandum on the same subject. Criticisms and suggestions of the referees were genuinely appreciated, and also influenced the final form of the paper.

References

1. Richard Bird and Philip Wadler, *Introduction to Functional Programming*, Prentice Hall (1988).
2. A. Bloss and P. Hudak, ‘‘Path Semantics’’, in *Lecture Notes in Computer Science, volume 298: proceedings of 3rd Workshop on Mathematical Foundations of Programming Language Semantics„* Springer-Verlag (1988).
3. Hans-J. Boehm, Robert Cartwright, Mark Riggle and Michael J. O’Donnell, ‘‘Exact Real Arithmetic: A Case Study in Higher Order Programming’’, pp. 162-173 in *ACM Conference on LISP and Functional Programming*, MIT (1986).
4. C. Clack and S. L. Peyton Jones, ‘‘Strictness Analysis – a Practical Approach’’, pp. 35-49 in *Functional Programming Languages and Computer Architecture*, ed. Jean-Pierre Jouannaud, Springer-Verlag Lecture Notes in Computer Science 201 (September 1985).
5. D. P. Friedman and D. S. Wise, ‘‘CONS Should not Evaluate its Arguments’’, pp. 257-284 in *Proc. 3rd Int. Colloq. on Automata, Languages and Programming*, ed. S. Michaelson and R. Milner, Edinburgh U. Press, Edinburgh (1976).
6. S. Abramsky & C. Hankin (Eds.), *Abstract Interpretation of Declarative Languages*, Ellis Horwood (1987).
7. D. Harel, *Algorithmics*, Addison-Wesley (1987).
8. P. Henderson and J. H. Morris jr., ‘‘A Lazy Evaluator’’, pp. 95-103 in *Third ACM Symposium on Principles of Programming Languages*, Atlanta, GA, USA (Jan 1976).
9. R. J. M. Hughes, ‘‘Why Functional Programming Matters’’, Report PMG-40, Chalmers Institute, Goteborg, Sweden (1984).
10. B. Kernighan and D. Ritchie, *The C Programming Language (2nd edition)*, Prentice-Hall (1988).
11. John McCarthy, ‘‘Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I’’, *Communications of the ACM* **3**(4), pp. 184- (April 1960).
12. R. Milner, ‘‘A proposal for Standard ML’’, pp. 184-197 in *ACM Symposium on LISP and Functional Programming*, Austin, Texas (1984).
13. D. A. Turner, ‘‘The Semantic Elegance of Applicative Languages’’, pp. 85-98 in *ACM Conference on Functional Programming Languages and Computer Architecture*, New Hampshire, USA (October 1981).
14. D. A. Turner, ‘‘Functional Programming and Proofs of Program Correctness’’, in *Tools and Notions for Program Construction*, ed. D. Néel, Cambridge University Press (1982).
15. D. A. Turner, ‘‘Recursion Equations as a Programming Language’’, in *Functional Programming and its Applications*, ed. J. Darlington, P. Henderson, and D. A. Turner, Cambridge University Press (1982).
16. J. Vuillemin, ‘‘Exact Real Computer Arithmetic with Continued Fractions’’, pp. 14-27 in *ACM Conference on LISP and Functional Programming*, Snowbird, Utah (1988).
17. N. Wirth, *Pascal User Manual and Report (2nd edition)*, Springer-Verlag (1975).
18. N. Wirth, *Programming in Modula-2 (2nd edition)*, Springer-Verlag (1983).