The Implementation and Use of Lazy Naturals

Elliott Hughes

3rd Year Project
Department of Computer Science
THE UNIVERSITY of York

March 1997

Abstract

Programmers, despite being Peano's successors, have neglected the natural numbers. A natural adjunct to functional programming's algebraic types and inductive proofs, natural numbers are even here passed over in favour of the machine word as the basic numeric type. This neglect has tainted functional programs with unnecessary eagerness and complexity of definition.
This paper presents implementations of lazy natural numbers in Haskell and presents the case for the adoption of such a type.

Contents

Chapter 1

Introduction

In a world where so much effort goes in to being lazy – from the lazy context-switches of our post-BSD operating systems to our lazy functional programming languages – there lurks an unexpected source of eagerness. The lazy functional languages secretly imbue our programs with eagerness, often accidentally ensuring that all our effort comes to naught.

Taunted by the C programmers, the designers of functional languages sacrificed a degree of laziness at the altar of efficiency. They defined arithmetic to be something special, something built-in, something primitive. What isn't immediately obvious from the languages' preludes is that arithmetic is eventually performed on machine registers using machine arithmetic instructions. Even those systems that allow arbitrary precision are guilty of using eager arithmetic.

And that's not all. The pernicious two's-complement register cuts deeper. We live by the natural numbers. They describe our data structures, they guide our proofs. But here too, we emulated our foe. We took on board the integer (as embodied by the machine register) without stopping to consider the consequences. Without stopping to question whether the integer is what we really really want.

Why distinguish between the integer and the machine register? Many programmers prefer not to. Many programmers write incorrect programs because they don't know – or don't care – about underflow and overflow. The machine register embodiment of an integer is rather like arranging numbers on a disc where the largest and the smallest representable numbers are adjacent to one another. Addition is performed by rotating the disc in one sense, subtraction by a rotation in the opposite sense. Subtract past the smallest number and you reappear amongst the largest numbers. Add past the largest number and you reappear amongst the smallest numbers. Although most programmers are aware of this fact, and most architectures provide a signal of the event when it occurs, only machine languages allow access to the information. C programmers, for example, must explicitly test their arguments before they attempt to add or subtract. C programmers, needless to say, tend not to do this. In order that functional programmers needn't miss out on all the fun, modern functional languages such as Haskell offer the possibility of querying the implementation's limits of precision.

1.1 Motivation

There are two distinct issues here: our languages don't afford naturals the respect they deserve, and we suffer a loss of laziness.

Writing correct definitions

A particularly insidious problem is the effect that integers have on the correctness of definitions. Continually worrying about the exceptional negative cases distracts us from the real problem, and it can often be quite hard to see that what is defined is what was intended. There are also philosophical objections to the use of $(\mathsf{n}+\mathsf{k})$ patterns because unlike $(\mathsf{x} : \mathsf{xs})$ patterns, they don't reflect the underlying structure of the data.

The fixed size of an integer has its effect on correctness too. An apparently correct equation may not work because of the order in which it applies arithmetic operators: one always has to be careful that a result doesn't become needlessly large, lest its register overflows.

Lazy evaluation

With a lazy functional language there is no need to constrain oneself to finite data structures (though they must of course be describable finitely). Lazy evaluation ensures that results are not computed until required, and that they are made available as they are computed. It's not an all-or-nothing situation, as it would be in an eager language.

A simple demonstration of the problem uses primes, the infinite list of all prime numbers (translated to Haskell from the example in [3]):

```
primes :: [Int]
primes                =  map (head) (iterate sieve [2 . .])
where sieve (p : xs)  =  [x | x ← xs, (x'rem'p) ≠ 0]
```

The authors go on to demonstrate that infinite lists are perfectly usable with a lazy functional language, provided that only some finite portion of the list need be evaluated. (An eager functional language would insist on evaluating the entirety of the infinite list immediately it was encountered.) For example, the first 100 primes can be generated by evaluating:

$$\mathsf{take}\ 100\ \mathsf{primes}$$

On the face of it, it would seem that the following would evaluate to True:

$$\mathsf{length\ primes} > 4$$

Indeed, one would expect to be able to answer True as soon as the fifth item of the list is encountered.

In actual fact, the definition of length and eager built-in arithmetic conspire to ensure that computation never terminates (barring abortive termination caused by lack of memory). Primitive addition requires both of its arguments to be fully evaluated. In this case, it needs to build the infinite list $1 + 1 + \cdots + 1$ before it can begin to tell us anything about the length of the list.

Strictness

To ensure that every well-formed expression has a value, functional languages extend each type with a value $\perp$ (pronounced "bottom") which represents the undefined value for that type. A function whose value is $\perp$ whenever a particular argument is $\perp$ is said to be *strict* in that parameter. A function whose value is other than $\perp$ despite a particular argument being $\perp$ is said to be *non-strict* in that parameter.

Non-strictness is only possible with lazy evaluation. Because results are only computed on demand, if a function never makes use of a parameter, it is irrelevant whether or not the argument corresponding to that parameter has a defined value. As an example, consider the function:

$$\mathsf{k}_0\ \mathsf{x} \quad = \quad 0$$

The variable x does not occur on the right-hand side of the definition, so its value is not required. This means that k_0 is non-strict. That's to say, $\mathsf{k}_0\ \perp = 0$.

It's also perfectly possible for a structure to be *partially undefined*, as with the tuple $(3, \perp)$. A function whose value is $\perp$ if any part of a particular argument is $\perp$ is said to be *hyper-strict* in that parameter.

Arithmetic is of necessity partially strict (i.e. at least one argument must have a value other than $\perp$), but it need not be strict in all its parameters and it need not be hyper-strict. An example is $4 + \perp$. Although this doesn't have a well-defined value, it may be considered to be at least 4 (of course, an alternative interpretation may be preferred). On the other hand, little can usefully be said about $4 * \perp$.

1.2 Aims

The current state of affairs is rather unfortunate. Eagerness can unwittingly be added to an otherwise lazy program by no more than making use of arithmetic. Moreover, the eagerness is contagious: functions that make use of prelude functions that in turn use arithmetic also become tainted.

A minimal set of aims, then, includes the following:

- **Lazy arithmetic operators:** addition, subtraction, multiplication and quotient and remainder are all eager primitives. Additionally, the relational operators are required.

- **A lazy prelude:** the prelude exists to encourage programmers to use the definitions it contains. When those definitions are needlessly eager, we lose some of the power of our language. A side-effect (did I say side-effect?) of the point below about user-interface is that we might hope not to have to change the prelude.

We might also like to ensure:

- **A constructed natural type:** to be in keeping with the other types in our language, lazy naturals should be a constructed type. In addition to the all-too-often unrecognised elegance of uniformity, a constructed numeric type could offer the programmer a reduced cognitive load. It seems likely that function definitions similar to those for other constructed types might be available, and that similar laws will hold for lazy naturals as already hold for existing constructed types.

- **No fixed upper bound on the size of a number:** a great variety of arbitrary-precision integer arithmetic routines (see chapter 2) have been written to avoid the underflow and overflow problems mentioned above. An implementation of lazy naturals should at least offer its user this peace of mind.

- **Reasonable efficiency:** although no lazy natural implementation can hope to perform quite as efficiently (in terms of both space and time) as a competing machine integer scheme, the efficiency should be sufficient that the lazy naturals are perfectly suitable for everyday use.

- **A user-interface similar to primitive arithmetic:** it should not be necessary for the programmer to learn a whole new set of functions. It would be reasonable to expect, for example, that addition has its usual operator symbol, precedence, associativity, etc.

1.3 Overview

The next chapter constitutes a review of the area. The alternative approaches to implementing naturals are contrasted and discussed.

The sequel can be considered to be in two logical parts. The first, chapters 3-6 deal with the implementation of lazy naturals, taking it to be a foregone conclusion that lazy naturals are what we want. Chapter 3 begins by describing the Haskell system of numeric types. The other three chapters present two implementations and an advantageous modification to a Haskell implementation.

Chapters 7 and 8 form the second logical part, where the lazy naturals developed in part one are made use of. Chapter 7 concentrates on revising the Haskell prelude to exhibit increased laziness. Chapter 8 covers more general applications of lazy naturals.

A conclusion rounds off the report by evaluating the various implementations and suggesting future lines of enquiry.

Chapter 2

Review

The natural numbers may not have been invented by computer programmers but they have certainly been the most devoted supporters of the natural cause. Mathematicians have traditionally blurred the distinction between naturals and integers, and it is only quite recently that mathematicians have studied the naturals with an eye to proving the both the solidity of the foundations and the detail of the properties they assume of the operations they use.

In light of the outline of the problems with integers in the previous chapter, it is no wonder that support for naturals has always been an issue in programming language design (see [19]). Despite this, there has been next to no interest in lazy naturals. It is disappointing to reflect that even those languages that exhibit interesting and complicated type systems pass over naturals in favour of their integer cousins.

This chapter reviews the most relevant work in both the mathematical and computer programming arenas. In the mathematical sphere, only the work of Peano and Landau is considered. The consideration of the programming sphere runs the gamut from functional languages to Ada 95.

2.1 The mathematician's view

The naturals cannot be said to have excited the mathematicians. They seem unable even to decide whether the naturals are the non-negative integers (starting at zero) or the positive integers (starting at one). Much of our mathematical notation is surprisingly recent, so it should come as no surprise that there are also various ways of writing the set's name.

Two mathematicians whose names are intimately linked with natural numbers are Giuseppe Peano and Edmund Landau. Peano was the first to attempt to axiomatise the natural numbers as a basis for proof. Landau expanded on this by presenting a great number of theorems, taking Peano's axioms as his starting point.

Both Peano and Landau were concerned with the way mathematics was being taught and the fact that though almost everyone knows that, say, multiplication of two naturals is commutative, far fewer people are able to prove it.

Giuseppe Peano

The Italian Giuseppe Peano is most famous to non-mathematicians for his *"Arithmetices principia nova methodo exposita"* [16], often known in English as "Peano's axiomatisation of arithmetic", after the form that this "new method" took. His most interesting axioms for our purposes – rewritten in modern mathematical notation – are:

Axiom 1	$1 \in \mathbb{N}^+$
Axiom 6	$a \in \mathbb{N}^+ \Rightarrow a + 1 \in \mathbb{N}^+$

These two axioms define the set of natural numbers. Axiom 1 defines the least element, and axiom 6 tells us how to construct further elements. The complete set of Peano's axioms is – as he intended – a suitable foundation for most of the work on natural numbers.

Edmund Landau

The German Edmund Landau wrote a famous textbook called *"Grundlagen der Analysis"* ([13], known in English as "Foundations of analysis"). He took Peano's axioms for the natural numbers and developed many theorems, not only for natural numbers but also for fractions, cuts, reals and complex numbers. His use of more familiar notation than Peano's and his textbook presentation did much to popularise Peano's ideas. To distinguish the addition operator from successor, Landau used the notation x' for "successor of x".

2.2 Constructed types

Many functional programming textbooks give natural numbers as a familiar example of an algebraic ('constructed') type using a construction based on chains of successors, but fail to pursue their development.

Natural numbers in a functional language are dealt with in greater depth in [22], but the language is eager. The author makes explicit use of the successor construction (using the notation that Landau used for successor in [13]). Here, for example, is the definition of take (see chapter 7 for a description of take and other standard prelude functions):

```
take 0 x          =  [ ]
take n′ [ ]       =  [ ]
take n′ (u : x)   =  u : (take n x)
```

A different approach to constructed naturals is that taken by Knuth in a lecture at Stanford in December 1994. His "Computer Musing" was on the use of binary trees as a number representation for gigantic numbers, and the recursive algorithms for computing the sum and product of such numbers. In his system, the empty tree represents zero and the tree with left and right children a and b (where $0 \leq b < 2^a$) represents $2^a + b$. There are no algorithms presented for any kind of subtraction or quotient and remainder, either because of the difficultly of implementing them with the binary tree construction or simply because Knuth felt that they weren't appropriate.

It is obvious that Knuth's interest lies in the mathematical properties of his trees rather than their use in computation, and he goes as far as posing a number of research questions. Perhaps the greatest advantage of the notation is the fact that a small binary tree can have a gigantic value. Knuth states the existence of a six-node tree "whose numerical value involves more decimal digits than there are molecules in the universe". Though there is something to be said for the ability to efficiently handle such numbers (and to be able to compute with them), the absence – if we exclude multiplication by zero – of operators that can decrease the size of results is a significant drawback. There is no doubt that Knuth wins the award for the strangest constructed naturals.

Or does he? The pure lambda calculus can represent the natural n as n applications of a function f to an argument x. These are called the Church numerals, after Alonzo Church who first described the lambda calculus (in his book [5]). For example:

$$\begin{aligned} 0 &= \lambda f.\lambda x.x \\ 1 &= \lambda f.\lambda x.fx \\ 2 &= \lambda f.\lambda x.ffx \end{aligned}$$

Addition of two such numbers is defined as follows:

$$+ = \lambda n.\lambda m.\lambda f.\lambda x.n\ f\ (m\ f\ x)$$

To see how this works, start at the right-hand side: we see m unchanged by binding f and x to the arguments f and x of the new function. This is in turn used as x in the application of n where f is again bound to the argument f of the new function. To paraphrase: the second number is used as the argument x of the first, i.e. the second repeated application of f is tacked on to the end of the first.

Despite its appearance, this scheme isn't nearly as strange as it might seem at first sight. It is nothing more than an implementation of a constructed type without any syntactic sugar.

2.3 Imperative approaches

Back on the beaten track, responses to the need for naturals in imperative languages cover a wide spectrum. It isn't easy to guess what a particular language's naturals will look like, though there are general rules. Languages related in spirit to Pascal favour a sub-range type of the integers. The systems programming languages, on the other hand, favour the modulo arithmetic of 'unsigned' integers. Imperative languages have traditionally avoided built-in constructed types – though they can of course be simulated – preferring instead those types which most closely match the underlying hardware.

Sub-range types

Niklaus Wirth is an interesting case. He, like David Turner in the world of functional programming, is father to a line of programming languages from the typeless Euler[1] of his Ph.D. thesis, via Algol/W, Pascal, Modula, Modula-2, and the various versions of Oberon. The Pascal report [10] supports sub-range integer types but doesn't mention naturals at all, and Modula didn't have any notion of a natural either. Oberon – his most recent language, described in [25] – also fails to include naturals. Modula-2 is the only exception, with its 'cardinal' type (see [24]). Wirth then, is in the sub-range camp.

Luca Cardelli – one of the most important figures in type research – also seems to favour defining naturals as a sub-range of integers (see, for example, [4]).

[1] A tribute from one Swiss legend to another. Whether Euler will repay the compliment by giving up his place on the smallest denomination Swiss bank-note to Wirth remains to be seen.

Modulo arithmetic

C and C++ don't have naturals as such, preferring a type modifier `unsigned` that signifies arithmetic modulo 2^n, where n is the number of bits used to represent the type thus modified. Modulo arithmetic has no underflow or overflow conditions, and can have unexpected results for unwary programmers.

The definition of Standard ML [15] is interesting in that – though it mentions its `int` type – it doesn't specify the characteristics of this type. One is left to assume a signed two's-complement machine integer. More recently, AT&T's "SML Basis Library" has fixed this interpretation by defining ML's integer types as being to all intents and purposes two's-complement (by the round-about means of stating that integers need not be two's-complement, but must appear to be so to the programmer). They also specify that an exception is raised on overflow.

Java, the latest in line for the C throne and one of the moment's trendiest languages, ignores the naturals completely. There are no sub-range types, integers are fixed length and even C's `unsigned` modifier is gone. To quote the Java Language Specification [7]:

> The integral types are `byte`, `short`, `int`, and `long`, whose values are 8-bit, 16-bit, 32-bit and 64-bit signed two's-complement integers, respectively, and `char`, whose values are 16-bit unsigned integers representing Unicode characters.

Furthermore:

> The built-in integer operators do not indicate overflow or underflow in any way. The only numeric operators that can throw an exception are the integer divide operator / and the integer remainder operator `%` which throw an `ArithmeticException` if the right-hand operand is zero.

It seems that a perfectly valid way of ensuring portability is to assert that "all the world's a Sun".

Bridging the gap

Though Java side-steps the issue, Ada 95 – as usual – attempts to be all things to all men. The original 'designers' of Ada seem for once to have been concerned with making best use of a minimal set of facilities, leaving naturals to be a sub-range type of the predefined type `integer`. Ada 95 also offers the "modular integer", i.e. integer arithmetic modulo some number. An example from [2] is:

```
type Unsigned_Byte is mod 256;
```

This type corresponds to C's `unsigned char`. Notice though that Ada 95 allows the programmer to specify arithmetic modulo *any* number, not just powers of two.

2.4 Arbitrary-length arithmetic

LISP Bignums

A solution that addresses the problem of arbitrary length numbers is the so-called *bignum*. Initially added to dialects of LISP to support symbolic algebra work, bignums have since spread to a variety of programming languages. Common LISP [21] has bignums in addition to the range-restricted *fixnum* type, as does Haskell (which maintains the distinction between the two types) and Mathematica (which doesn't). A stated aim of the Common LISP system is that, although it provides the two types of integer, it "is designed to hide that distinction as much as possible". The programmer is at liberty to choose between the more efficient range-restricted fixnums and the less efficient but unbounded bignums. Given an identical interface, the choice of which to use can be made at a very late stage.

Other languages

Many libraries are available to add equivalents of bignums to legacy languages such as Ada and C. Recent editions of the infamous "Numerical Recipes" have included example implementations. The typical approach is to use base-n arithmetic, where n is some value convenient for the particular implementation. Machine arithmetic instructions are used on these base-n 'digits' (which are in practice either bytes or words) while a higher-level algorithm deals with intricacies such as propagating carries between digits. The algorithms most often used are so well-known that they are often referred to as the "classical algorithms". Section 4.3 of volume 2 of Knuth's legendary series of books [12] is an excellent introduction to the algorithms involved in arbitrary-precision arithmetic.

2.5 Discussion

Of the languages that do attempt to offer naturals, those using sub-range types are to be preferred over those offering only unsigned integers. Unsigned types fail to correspond to the situations in which programmers generally find themselves. Even the embedded system or operating system programmers who might be considered most likely to use an unsigned type tend to use them in situations to which arrays of bits would be better suited. Just as high-level programmers find themselves using integers when what they really want is naturals, the low-level programmers find themselves using unsigned integers when what they really want is a bit array. Only Ada 95 offers modulo arithmetic with sufficient generality that it can be used for such purposes as circular buffers and cryptography.

In practice, because they correspond to data structures or real-world values, programmers can state both upper and lower bounds for most of the numeric variables they use. This makes the use of sub-range types safe and efficient.

The languages that offer sub-range types also offer exceptions, in the main. These two features tie together to give the programmer a means of identifying out-of-range numbers. The major problems with the sub-range type approach are that it imposes an upper bound on the naturals (which, even when finite, might not be known), and the problem of type equivalence. A standard 'natural' type would allow naturals to be communicated amongst independently designed routines. As it is, a plethora of conversion routines is often required: few languages (either in definition or implementation) have the sophistication required to recognise interconvertability. Still others view such automatic conversion as inherently wrong.

The neglect shown to the natural numbers is yet another example of our hubris. Without having properly considered – let alone secured – our foundations, we focus our attentions on "higher things". It is interesting to note that in the recent flood of 'object-oriented' languages, basic types seem very much to be second-class citizens.

Chapter 3

Haskell Numerics

A stated aim of this project is that the user-interface to the lazy naturals should be familiar. This chapter discusses Haskell numerics in the context of its class system, the mechanism by which the standard numeric types are defined, and by which user types are able to imitate these standard types. In this chapter, *emphasis* is given to words defined by the Haskell report [9].

3.1 The class system

The Haskell type system has a notion of a *type class* (or class). A class is a set of *class methods* which must be implemented by any type claiming to be an instance of that class. An *instance* is simply a set of bindings corresponding to the methods defined by a particular class.

A *default class method* may be provided by a class. An instance is then at liberty to omit a definition, in which case the default definition will be used. In the absence of a default method, an instance must define the method.

The class Eq is the class of all types for which equality and inequality are defined. The type variable a stands for the type proposed to belong to Eq. Notice the default class method for inequality:

```
class Eq a where
  (==), (≠)        ::  a → a → Bool

  x ≠ y            =   not (x==y)
```

A type wishing to demonstrate its membership of Eq need only define equality. (An implementer who knows a more efficient way of implementing inequality is at liberty to provide one, however.)

Imagine the type Edge used in a window manager:

```
data Edge  =  North | East | South | West
```

To be able to compare values of type Edge for equality, Edge must be an instance of class Eq. An example instance declaration for Edge is:

```
instance Eq Edge where
  (==) North North    =  True
  (==) East East      =  True
  (==) South South    =  True
  (==) West West      =  True
  (==) _ _            =  False
```

Of course, for types where structural equality suffices, this would be tedious were it necessary. In these cases (and for the classes Eq, Ord, Enum, Bounded, Show, or Read) Haskell can derive any necessary definitions (see appendix D of [9] for more information):

```
data Edge  =  North | East | South | West
              deriving Eq
```

It is possible that a class has prerequisite classes. That is, a type must be an instance of those prerequisite classes before it can be an instance of the class itself. The list of prerequisites is a list of *class assertions* forming a *context*.

3.2 Standard Prelude classes

For the purposes of implementing lazy naturals, there are relatively few classes of interest (see figure 3.1). The class Eq has already been mentioned, and the related class Ord is those types over which a total ordering can be defined. Class Enum contains all types which are enumerable (i.e. there is a notion of successor). The Show class offers conversion of a type to a string. Given membership of these and in addition the class Eval (for which an instance is automatically defined for every type), a type can be considered a Haskell numeric type just like any other.

The definitions required are few: a definition of equality, of less-than or equal-to, of enumFrom and enumFromThen ([n..] and [n, n′..]), and finally

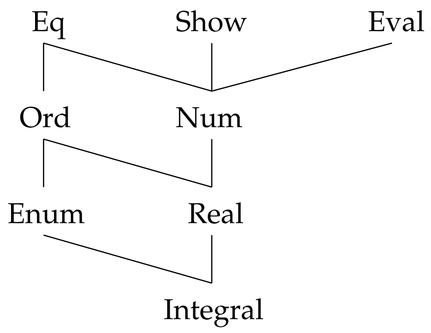

Figure 3.1: The Haskell numeric class hierarchy

of the arithmetical operators themselves. Although these definitions are perfectly applicable to the case of lazy naturals, the water muddies as we consider the arithmetical operators. The design seems to have been heavily influenced by the requirements of the built-in numeric types. The basic class Num – which doesn't include quotient or remainder – does include negation, absolute value and a function to return the sign of a number.

The class Num also has a fromInteger member. An integer numeric literal in a Haskell program represents fromInteger applied to the Integer value of the integer. This allows the literal to have any appropriate type, the actual type being inferred from context. In particular, it allows programs to contain lazy natural literals.

The Integral class has members for division (quotient) and remainder. The default definition of division uses quotient and manipulates the result depending on sign. This means that lazy naturals need only define quotient and remainder.

3.3 Summary

The Haskell class system offers everything one could wish for in this situation. The possibility of overloading the standard arithmetic operations and the ability to use literal values of a non-standard type combine to realise the hope of fully integrating lazy naturals with the basic types.

Chapter 4

Successor Naturals

Peano's axiomatisation – described in chapter 2 – may have caused a stir in its day, but it seems a conservative option to the functional programmer. In particular, it can be seen as the definition of an algebraic type. The two axioms on page 7 define our base element and how to construct further elements. It makes good sense to explore the possibilities of Peano's approach before considering more outlandish constructions.

In Haskell, the axioms are equivalent to:

$$\textbf{data}\ \mathsf{Positive} = \mathsf{One} \mid \mathsf{Succ}\ \mathsf{Positive}$$

Using this representation, the number 1 is One, 2 is Succ One, 3 is Succ Succ One, etc. There are many partially defined values – successors of an undefined computation – and an infinite value. The definition of infinity is rather elegant:

$$\infty = \mathsf{Succ}\ \infty$$

You will have noticed that Peano has defined the positives, but our intention is to implement the naturals. This means that we need a representation of zero. It isn't difficult to see that Peano's axioms are not harmed by taking the symbol One to represent the number zero: One is simply a symbolic name for the least element in the set. Although perfectly logical, this convention would be at best confusing and certainly a poor start on the road to lazy naturals with a familiar user interface!

(In his later work [17], Peano gives inductive definitions of addition, multiplication, exponentiation and factorial. He also makes use of the number zero. As ever, we're not the first to come this way.)

4.1 The data type

Peano's definition is easily amended to start from zero: the alteration is simply a matter of replacing one name with another. The new type is called SuccNat[1]:

$$\textbf{data}\ \mathsf{SuccNat} = \mathsf{Zero} \mid \mathsf{Succ\ SuccNat}$$

Values of type SuccNat are easily converted to and from Haskell integers:

```
fromInteger 0        =  Zero
fromInteger (n+1)    =  Succ (fromInteger n)

toInteger Zero       =  0
toInteger (Succ n)   =  1 + toInteger n
```

The traversal of a successor natural exemplified by toInteger is a common requirement, as is the analogous operation on lists. This list operation is called foldr.

In essence, foldr reconstructs a list. It can be thought of as replacing each cons with a function application and the empty list with a user-supplied zero element. The successor natural analogue replaces each Succ with a function application, and Zero with a user-supplied zero element.

The analogue is called fosdr (pronounced 'foster'). Replace the 'l' for list in foldr with the 's' for successor natural if you don't understand the name. The definition is simple and almost identical to that of foldr:

```
fosdr :: (α → α) → α → SuccNat → α
fosdr f z Zero       =  z
fosdr f z (Succ n)   =  f (fosdr f z n)
```

The type of fosdr is somewhat simpler than foldr because the chain of successors carries no information (that's to say, always carries the same information), whereas the function applied by foldr takes a list item as a parameter.

More uses for fosdr will be found later, but for now it will serve to demonstrate that toInteger can be re-written:

```
toInteger   =   fosdr (1+) 0
```

[1]The intention being that the programmer can select a preferred implementation using, for example, **type** Nat = SuccNat.

Another important fundamental operation is higher-order comparison (or “three-way switch”). This function has five parameters: three functions (lt, eq and gt) and the two numbers. Both numeric arguments are counted down to zero (making this function strict). If they both reach zero in the same step, the eq function is applied to their difference (zero). If the first reaches zero before the second, the lt function is applied to their difference (what remains of the second). If the second reaches zero before the first, the gt function is applied to their difference (what remains of the first):

```
switch :: (SuccNat → α) → (SuccNat → α) → (SuccNat → α) →
            SuccNat → SuccNat → α
switch lt eq gt (Succ x) (Succ y)  =  switch lt eq gt x y
switch lt eq gt Zero Zero          =  eq Zero
switch lt eq gt Zero y             =  lt y
switch lt eq gt x Zero             =  gt x
```

An alternative definition doesn’t apply the eq function to the constant Zero — after all, why bother with a parameter that always has the same value? The uniformity of all three functions having the same type is sufficient reason to prefer this definition. This will become evident later in this chapter, when many arithmetic functions are defined using little more than switch.

4.2 Addition

Addition is the least problematic of the arithmetical operators, though even in its definition there lurk a number of pitfalls and trade-offs. Some of these will be made apparent by the presentation of a number of possible solutions.

Peano’s lazy addition

In [17], Peano gives a definition of addition analogous to this:

```
add (Succ x) y  =  Succ (add x y)
add Zero y      =  y
```

Peano’s turns out to be a useful definition. It doesn’t use accumulation to produce its result, avoiding one source of eagerness. It does however have the unfortunate property of victimising its left-hand parameter, forcing its evaluation.

The problem arises because the case analysis is all on the left-hand parameter. In order to select an equation, the functional program needs to know how one of its arguments is constructed at its outermost level. Is it a Zero or a Succ? To answer this question, some evaluation of an argument must occur: addition has to be strict in one parameter, but the choice of which is arbitrary. This arbitrariness is what I refer to as victimization.

Take, for example, the case of $\mathsf{u} + \mathsf{e}$ where the left-hand argument is unevaluated while the right-hand argument is fully evaluated. The definition of $+$ forces the evaluation of u before attaching e to its end without so much as looking at it. Addition would have been lazier had case analysis been on the right-hand parameter: then u would be attached – still unevaluated – to the end of e. The case $\mathsf{e} + \mathsf{u}$, on the other hand, results in maximal laziness with the current definition.

Without knowing in advance to what extent each of its arguments have already been evaluated, this addition cannot be maximally lazy.

Comparison with list append

It isn't unreasonable to expect to see a family resemblance between addition of constructed numbers and appending lists. Expectations are raised in particular by the law:

$$\mathsf{length\ xs} + \mathsf{length\ ys} \quad = \quad \mathsf{length\ (xs} +\!\!+ \mathsf{ys)}$$

The likeness does extend to the definitions. The Zero constructor corresponds to the empty list, and the Succ constructor to cons (:). The similarity between a lazy natural definition and a list definition is quite striking. Here is an example definition of list append:

$$\begin{array}{lcl} \mathsf{(x{:}xs)} +\!\!+ \mathsf{ys} & = & \mathsf{x{:}(xs} +\!\!+ \mathsf{ys)} \\ [\,] +\!\!+ \mathsf{ys} & = & \mathsf{ys} \end{array}$$

Using the same correspondence between zero and the empty list, we see the similarity between the left- and right-identity laws for list append and natural addition:

$$\begin{array}{lcl} \mathsf{xs} +\!\!+ [\,] & = & \mathsf{xs} \\ \mathsf{x}+0 & = & \mathsf{x} \\ \\ [\,] +\!\!+ \mathsf{xs} & = & \mathsf{xs} \\ 0+\mathsf{x} & = & \mathsf{x} \end{array}$$

Another useful law is associativity. Both list append and natural addition are associative:

$$\begin{array}{lcl} \mathsf{xs} \mathbin{+\!\!+} (\mathsf{ys} \mathbin{+\!\!+} \mathsf{zs}) & = & (\mathsf{xs} \mathbin{+\!\!+} \mathsf{ys}) \mathbin{+\!\!+} \mathsf{zs} \\ \mathsf{x}+(\mathsf{y}+\mathsf{z}) & = & (\mathsf{x}+\mathsf{y})+\mathsf{z} \end{array}$$

Re-writing Peano addition

Just as list append can be defined using foldr, natural addition can be defined using fosdr. We 'replace' each Succ in the first number with Succ, and use the other number as our zero element. That's to say: we attach one number in place of the other number's Zero. Choosing once more to victimise the left-hand parameter, we come by the following definition (with an analogous definition of list append for comparison):

```
add x y     =  fosdr Succ y x
(++) xs ys  =  foldr (:) ys xs
```

This definition is broadly similar to the original Peano definition. As before, if we knew the relative extents to which the parameters have been evaluated, we could make a better choice of which parameter to victimise.

Even-handed addition

Here is an alternative definition of addition which is even-handed in its demands on its arguments:

```
add (Succ x) (Succ y)  =  Succ (Succ (add x y))
add Zero y             =  y
add x Zero             =  x
```

By taking a successor from each argument, it might seem superior to the previous definition: after all, doesn't it move twice as quickly towards a result?

The answer is no. This definition is in general worse than Peano's. Whereas Peano's addition victimises one parameter and leaves the other alone, this addition victimises both of its parameters simultaneously. (That's where the intuitive factor of two goes.) Like Stalin, this definition achieves equity by mistreating all. Peano offers the chance of maximal laziness on more occasions than this definition, though this definition is superior in the case where the two arguments are a shared reference to the same structure. If doubling was seen to be particularly useful in some application, then a definition along these lines would be eminently suitable.

4.3 Multiplication

Peano multiplication

Peano tried his hand at multiplication too. Had he been alive today, he would be writing Haskell rather than mathematics, and his definition would have looked like this:

```
greedymul x y  =  fosdr (x+) Zero y
```

That's to say: multiplication is repeated addition. fosdr replaces each Succ in its second argument with an addition of its first argument.

An interesting etymological detail in the context of this definition is the fact that English word multiplication derives in part from the Latin word plicare, meaning to fold. (This in turn, may have derived from the Greek $\pi\lambda\iota\sigma\sigma o\mu\alpha\iota$, to spread the legs, but that's only of prurient interest.)

It's not surprising that both arguments will be evaluated: it's not possible to say anything about the result of multiplying two constructed numbers without knowing something about both of them (because either could be zero, and that decides what the outermost constructor in the result will be).

Having admitted that multiplication is necessarily strict in both parameters, is there any improvement we can make to Peano's multiplication? In figure 4.1 you can see how Peano multiplication proceeds. If multiplication is seen as filling a rectangle, and the two arguments as perpendicular lines demarcating two edges, Peano's scheme works by laying down strips of one length, moving in the direction of the other edge. This means that Peano multiplication must evaluate the entirety of the argument chosen as the 'strip'.

Even-handed multiplication

When we have to cause evaluation of our arguments, we want the greatest return on this investment. Going back to the analogy with covering a rectangle, we can think of the perimeter of the covered area as the amount of evaluation we've caused (actually twice the amount, but that isn't important). An improvement in our return is brought about by making use of the observation that a square has the highest area-to-perimeter ratio of any rectangle. A square expanding out from the corner covers the area optimally for a given amount of evaluation. We finish off with Peano multiplication because by that time one of our arguments is fully exhausted,

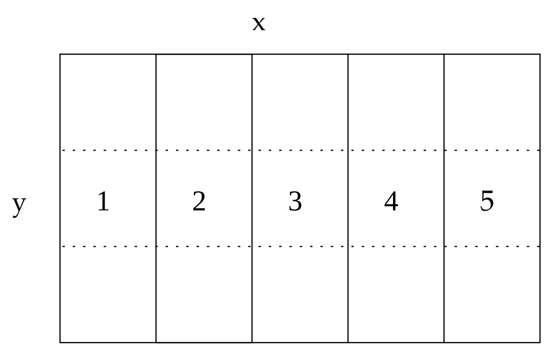

Figure 4.1: Peano multiplication of 3 by 5. The numbers show the order in which the strips are laid down.

and can be offered for sacrifice. Here is the definition:

```
mul x y  =  emu 1 x y where
              emu c 0 y'                  =  greedymul x y'
              emu c x' 0                  =  greedymul y x'
              emu c (Succ x') (Succ y')   =  c+emu (c + 2) x' y'
```

Figure 4.2 shows multiplication of 3 by 5. Notice how previously evaluation of four successors (one in the x direction and three in the y direction) contributed three successors to the result, whereas now evaluation of four successors (two in each direction) contributes four. Unfortunately, we now demand evaluation of both arguments at once.

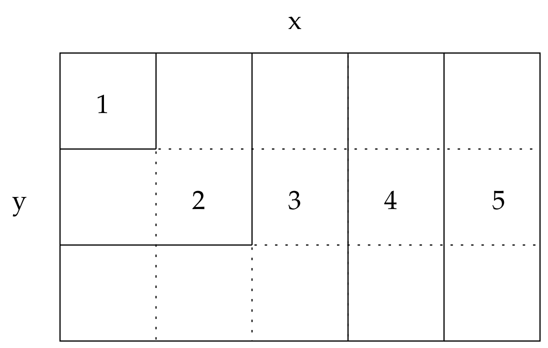

Figure 4.2: Even-handed multiplication of 3 by 5. The numbers show the order in which the strips are laid down.

Diagonalized multiplication

We can easily make our multiplication less voracious. The idea is still to move out from the corner, but this time we keep our demands for evaluation under control. When expanding out, we have two choices: we either expand equally in both directions or we don't. We've tried expanding equally in both directions and didn't much care for it. The alternative is to alternate between arguments and avoid victimising either argument (or indeed both).

Diagonalized multiplication alternately lays down rectangles alongside and underneath the previous rectangle (see figure 4.3). When one of the arguments has been exhausted, we again revert to Peano multiplication.

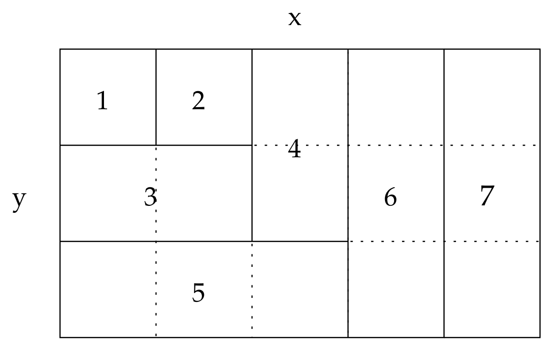

Figure 4.3: Diagonalized multiplication of 3 by 5. The numbers show the order in which the strips are laid down. Notice that the last strips are special: because the arguments weren't equal, we end by multiplying a fully-evaluated number by a partially evaluated one. We can use Peano multiplication to do this.

The definition of diagonalized multiplication is only slightly more complicated than that for even-handed multiplication. The auxiliary `sidebottom` performs the laying down of strips. The Boolean first argument is `True` for a side strip, `False` for a bottom strip. The parameter `c` accumulates the strip length, which increases on each side-to-bottom transition. The two arguments to the initial multiplication are evaluated alternately as needed. When one of the arguments has been fully evaluated, Peano's multiplication is called in to finish the job (by this time we know which argument to offer for victimization):

```
mul (Succ x) (Succ y)            = sidebottom True Zero (Succ x) (Succ y)
where
  sidebottom _ c Zero y'         = greedymul c y'
  sidebottom _ c x' Zero         = greedymul c x'
  sidebottom True c (Succ x') y' = c+sidebottom False (Succ c) x' y'
  sidebottom False c x' (Succ y') = c+sidebottom True c x' y'
mul _ _                          = Zero
```

An example reduction is that of $3 * 4$ (with numeric literals taking the place of successor chains for clarity). The subscript numbers relate reductions with strips in figure 4.3.

$$
\begin{array}{ll}
 & \mathsf{mul}\ 3\ 4 \\
\rightarrow & \mathsf{sidebottom}\ \mathsf{T}\ 0\ 4\ 3 \\
\rightarrow & 0 + \mathsf{sidebottom}\ \mathsf{F}\ 1\ 3\ 3 \\
\rightarrow_1 & 0 + 1 + \mathsf{sidebottom}\ \mathsf{T}\ 1\ 3\ 2 \\
\rightarrow_2 & 0 + 1 + 1 + \mathsf{sidebottom}\ \mathsf{F}\ 2\ 2\ 2 \\
\rightarrow_3 & 0 + 1 + 1 + 2 + \mathsf{sidebottom}\ \mathsf{T}\ 2\ 2\ 1 \\
\rightarrow_4 & 0 + 1 + 1 + 2 + 2 + \mathsf{sidebottom}\ \mathsf{F}\ 3\ 1\ 1 \\
\rightarrow_5 & 0 + 1 + 1 + 2 + 2 + 3 + \mathsf{sidebottom}\ \mathsf{T}\ 3\ 1\ 0 \\
\rightarrow & 0 + 1 + 1 + 2 + 2 + 3 + \mathsf{greedymul}\ 3\ 1 \\
\rightarrow_6 & 0 + 1 + 1 + 2 + 2 + 3 + 3 \\
\rightarrow^* & 12
\end{array}
$$

Note that reduction does not necessarily proceed in such a way that all addition occurs after all multiplication.

4.4 Subtraction

Subtraction is the most problematic of our operations over the naturals in that it isn't closed. To avoid these exceptional conditions, the monus operator is often used in preference (see [14] for more about monus). Monus is defined as zero if its second argument is greater than its first. A job easily performed by switch and the K combinator.

The K combinator[2] is defined as:

$$
\begin{array}{l}
\mathsf{k} :: \alpha \rightarrow \beta \rightarrow \alpha \\
\mathsf{k\ x\ y} \;=\; \mathsf{x}
\end{array}
$$

[2]The Haskell prelude already contains a definition of the K combinator, under the name const. Despite having lived with Ken Thompson's choice of `creat` over `create` in the UNIX C library, I still favour shorter names.

K (cancellator) can be seen as discarding its second argument. In conjunction with switch, a partial application of k disposes of the information about *how* different the parameters were — remember that this is the value to which switch applies its function arguments. Instead, k allows us to return a value of our choosing. In the case of monus, this is Zero in the less-than and equal-to cases:

$$\mathsf{monus} \;=\; \mathsf{switch\ (k\ Zero)\ (k\ Zero)\ id}$$

Monus isn't the only subtractive operator we can define over the naturals. An example alternative is the δ function, defined as the absolute difference between its two arguments:

$$\delta \;=\; \mathsf{switch\ id\ id\ id}$$

Interestingly, the following quaint law demonstrates the interrelation between δ and monus:

$$(\mathsf{x}-\mathsf{y})+(\mathsf{y}-\mathsf{x}) \;=\; \mathsf{x}\ \delta\ \mathsf{y}$$

Comparison with list suffix

Monus shares some properties with the common drop function on lists. Bearing in mind that the arguments to drop are in the opposite order to those for monus (the amount removed being the first argument to drop, rather than the second), we have the right-identity law:

$$\begin{aligned}\mathsf{(flip\ drop)\ xs\ 0} &= \mathsf{xs}\\ (-)\ \mathsf{x}\ 0 &= \mathsf{x}\end{aligned}$$

Accepting the equivalence of the empty list and zero, we also have a similar left-identity law (again, 'left' from the point of view of monus):

$$\begin{aligned}\mathsf{(flip\ drop)}\ [\,]\ \mathsf{x} &= [\,]\\ (-)\ 0\ \mathsf{x} &= 0\end{aligned}$$

4.5 Quotient and remainder

Quotient and remainder share a similar relationship with monus as multiplication does with addition. The idea is to count the number of times we can switch before we fully evaluate one or other of the arguments. What's left of the other argument is the remainder. The definition is straightforward:

$$\begin{array}{lcl} \mathsf{qr\ x\ y} & = & \mathsf{switch\ (k\ (Zero, x))\ (k\ (Succ\ Zero, Zero))\ step\ x\ y} \\ \textbf{where}\ \mathsf{step\ x'} & = & \mathsf{(Succ\ q, r)}\ \textbf{where}\ \mathsf{(q, r) {=} qr\ x'\ y} \end{array}$$

As one would expect, the remainder isn't made available until the input has been fully evaluated. The quotient, on the other hand, has a successor added each time that switch declares x to be larger than y.

Notice how this definition gives us $(1, 0)$ as the result of dividing 0 by 0, and a combination of infinite quotient and undefined remainder for any other value divided by 0. The reason behind the unexpected behaviour at zero is that the simple eq function gives the qr function the belief that $\forall x \bullet x/x = 1$. In actual fact, this belief only holds for finite x because switch cannot prove the equality of infinite values in finite time.

An alternative definition would add an equation for the case of division by zero. This might be useful if it were required that $x \bmod 0 = x$, as Knuth (in [8]) claims mathematicians expect.

Dividing x (where x is finite) by ∞ gives $(0, x)$ — a perfectly reasonable result.

4.6 The relational operators

It's easy to see how the relational operators can be implemented using switch and the K combinator: here, for example, is equality (the relational operators are shown after η-reduction):

$$(==) \;=\; \mathsf{switch\ (k\ False)\ (k\ True)\ (k\ False)}$$

Similarly, weak less-than is easily defined:

$$(\leq) \;=\; \mathsf{switch\ (k\ True)\ (k\ True)\ (k\ False)}$$

The four other relational operators can either be defined in truth-table manner as combinations of k True and k False as the three functions, or more entertainingly as:

$$\begin{array}{lcl} (\neq) & = & \mathsf{(not\ .)\ .\ (==)} \\ (\geq) & = & \mathsf{flip}\ (\leq) \\ (<) & = & \mathsf{(not\ .)\ .}\ (\geq) \\ (>) & = & \mathsf{(not\ .)\ .}\ (\leq) \end{array}$$

The Haskell Ord class offers its own default implementation of the relational operators, also in terms of weak less-than. The formulation here is slightly more direct and saves a small number of reduction steps when using the Hugs interpreter.

4.7 Miscellaneous functions

As was touched on in chapter 3, Haskell expects instances of class Num to offer three extra functions. The function abs (absolute value) is simply defined for naturals: id fits the bill perfectly. The sign function signum is also simple:

```
signum Zero  =  Zero
signum _     =  Succ Zero
```

Rather more troublesome is negate. The default method for subtract which uses addition and negate gives the user an unreasonable expectation of us. We cannot in general supply an additive inverse. Under the circumstances, perhaps the only reasonable solution is to report an error:

```
negate  =  error "undefined"
```

4.8 Summary

This chapter has shown that a successor-chain construction based on Peano's axioms for the natural numbers is perfectly suitable for implementing lazy naturals.

The constructed type has enabled us to bring out the correspondence between the lazy natural and the list — our fundamental data structure. Not only are the definitions of analogous operations similar, but some laws that hold for lists are found also to hold for lazy naturals.

We have seen the usefulness of higher-order functions demonstrated by fosdr and switch, in particular the way in which switch embodies the computation of a wide range of operations.

The next chapter examines an implementation of lazy naturals that builds upon the work in this chapter.

Chapter 5

Mixed Naturals

No matter how well we implement the operators dealing with successor naturals, we will be inefficient because of our representation. The representation itself has the problem that its size is proportional to the number it represents (in a sense, it is a unary representation where the Succ constructor is a '1' and Zero marks the end of a number). The functions fromInteger and fromInt which create values of type SuccNat are wasteful: they throw away a computed result, insisting on constructing a chain of successors of Zero.

The "mixed naturals" are based on a proposal in [19]. They combat waste by maintaining two separate components: an evaluated non-negative integer and a successor natural. The value of the mixed natural is the sum of the values of its constituents. The evaluated part allows more efficient work with ready-computed results, while the successor natural part allows laziness to be maintained.

If a thing looks too good to be true, you may be certain that it is. We hope to gain efficiency by incorporating machine integers into our lazy naturals, because we can use fast primitive arithmetic on these integers. This arithmetic – as the introduction explained – is eager. We must exercise great caution to avoid degenerating into primitive arithmetic on numbers. If we aren't careful, we will find ourselves performing eager arithmetic while being forced to carry around the baggage of a rarely-used successor natural.

In practice, avoiding this degeneration is likely to entail that we must sometimes sacrifice our familiar user-interface and deal directly with the structure of the mixed natural. In this way we can force greater use of the successor natural component, injecting laziness where a more abstract definition would be eager.

5.1 The data type

The Haskell type uses an infix constructor :+ to tie together an Integer and a SuccNat (as defined in the previous chapter). The constructor's shape is a reminder that the mixed natural's value is the sum of its two components' values. The type Integer is used in preference to Int that we may comply with our stated aim of imposing no arbitrary limit on the size of a natural. The type is called MixNat:

$$\textbf{data}\ \mathsf{MixNat} \;=\; \mathsf{Integer :\!+ SuccNat}$$

Mixed naturals can be converted to and from integers:

$$\begin{aligned}
\mathsf{fromInteger\ 0} &= \mathsf{0 :\!+ Zero}\\
\mathsf{fromInteger\ (n{+}1)} &= \mathsf{(n{+}1) :\!+ Zero}\\[1em]
\mathsf{toInteger\ (e :\!+ u)} &= \mathsf{e + fosdr\ (1{+})\ 0\ u}
\end{aligned}$$

What about fosdr for mixed naturals? The two applications we found for fosdr were in addition and greedy multiplication. The fact that the mixed naturals are a pair consisting of an integer and a successor natural combined with a desire to keep the two parts separate makes fosdr otiose: we can use built-in arithmetic on the integers and the definitions from the previous chapter on the successor naturals.

Although a mixed natural fosdr turned out not to be useful, a switch analogue is every bit as useful here as it was with the successor naturals. Its definition is rather more complicated — here's the type of the function we require:

$$\begin{aligned}
&\mathsf{mswitch} :: (\mathsf{MixNat} \to \alpha) \to (\mathsf{MixNat} \to \alpha) \to (\mathsf{MixNat} \to \alpha) \to\\
&\quad \mathsf{MixNat} \to \mathsf{MixNat} \to \alpha
\end{aligned}$$

There are various ways of implementing this function. An obvious approach would be to convert our mixed naturals into successor naturals. That way we could simply re-use the switch function already written. The disadvantage is that it would mean undoing our good work in holding on to the integer parts of our mixed naturals.

An alternative scheme does a rather complicated case analysis. Judicious use of auxiliaries allows this to written rather cleanly, and it is this implementation that is presented.

The function begins by comparing the integer parts of the mixed naturals. The case where the integers are equal is clear: they may safely be ignored, and we switch on the successor natural parts instead. If the first integer part is less than the second then we must demand the difference from the successor natural belonging to the smaller integer. If the demand cannot be met, the first number is smaller than the second and we apply lt to what remains of the second number. Otherwise, we have now exhausted both integers and fall back on comparing successor naturals. The case where the second integer part is larger than the second is similar, and the law of trichotomy means that our definition is complete:

```
mswitch lt eq gt (n :+ u) (m :+ v)
          | n < m         =  demand
                               (λr → (switch′ lt eq gt r v))
                               (λr → lt (r :+ u)) (m−n) u
          | n == m        =  switch′ lt eq gt u v
          | n > m         =  demand
                               (λr → (switch′ lt eq gt u r))
                               (λr → gt (r :+ u)) (n−m) v
```

The auxiliary demand is a simple two-way switch whose numeric parameters are of different types. As its name suggests, it demands that the successor natural be at least as large as the integer. If this is the case, it applies t (for true) to the remainder of the successor natural, otherwise f (for false) is applied to that part of the integer which remains unsatisfied despite the successor natural's being exhausted:

```
demand :: (SuccNat → α) → (Integer → α) → Integer → SuccNat → α
demand t f 0 u              =  t u
demand t f (n+1) (Succ u)   =  demand t f n u
demand t f (n+1) Zero       =  f (n+1)
```

Although switch has already been defined for comparing successor naturals, it is of no use to us because its three continuations (lt, eq and gt) have type SuccNat → α. Our function must take continuations of type MixNat → α. In order that we may re-use switch,the auxiliary switch' uses lambda abstractions to repackage successor naturals as mixed naturals. The successor naturals from switch are thus given to the user's lt, eq and gt functions as mixed naturals:

```
switch′ :: (MixNat → α) → (MixNat → α) → (MixNat → α) →
              SuccNat → SuccNat → α
switch′ lt eq gt u v  =  switch  (λr → lt   (0 :+ r))
                                 (λr → eq   (0 :+ r))
                                 (λr → gt   (0 :+ r)) u v
```

5.2 Addition

As ever, a mixed natural function can be implemented by converting the arguments to successor naturals and then using the successor natural function. The reason behind the addition of the integer part was that we wanted to facilitate the use of primitive arithmetic, and here we see an opportunity to do so. The interpretation given to the two parts of a mixed natural (that their sum is the value of the mixed natural) means that we can add integer and successor natural parts independently of one another:

$$(+)\ (\mathsf{n} :+ \mathsf{u})\ (\mathsf{m} :+ \mathsf{v}) \quad = \quad (\mathsf{n}+\mathsf{m}) :+ (\mathsf{u}+\mathsf{v})$$

Reliance upon the quality of definition of successor natural addition is implicit in this definition. The problem with this definition is that though we may come to evaluate some of the sum u+v, it will remain in the successor natural part.

Do the laws noted in chapter 4 still hold? Left- and right-identity certainly do, because both built-in addition and successor natural addition have these properties. Associativity also still holds.

5.3 Multiplication

Multiplication can also utilise the integer parts to get a head-start. The head-start involves (recalling the rectangle-filling analogy of the previous chapter) starting with a ‘free’ rectangle in the top-left corner. This rectangle will have the area of the product of the two integer parts.

We require an additional accumulating parameter to implement this, because the side strip and the bottom strip are no longer necessarily the same length. The accumulating parameter c is replaced by s (the side strip) and b (the bottom strip). Transitions between side strips and a bottom strips are now accompanied by increments to either the side or bottom strip length.

An extra optimisation comes with the realisation that there is now a choice of starting with either side or bottom strips. The following definition elects to start with whichever is larger (via the first argument to sb):

```
(*) (n :+ u) (m :+ v)      =  (n*m) :+
                                 sb (m > n) (fromInteger m) (fromInteger n) u v
where
  sb _ s b Zero y          =  greedymul b y
  sb _ s b x Zero          =  greedymul s x
  sb True s b (Succ x) y   =  s+sb False s (Succ b) x y
  sb False s b x (Succ y)  =  b+sb True (Succ s) b x y
```

Figure 5.1 shows an example multiplication of two mixed naturals. The eagerly-evaluated part is shown in the top right-hand corner, with the strips completing the rectangle. Note that the difference in length between side and bottom strips is greater than one. This is why the two accumulating parameters are necessary.

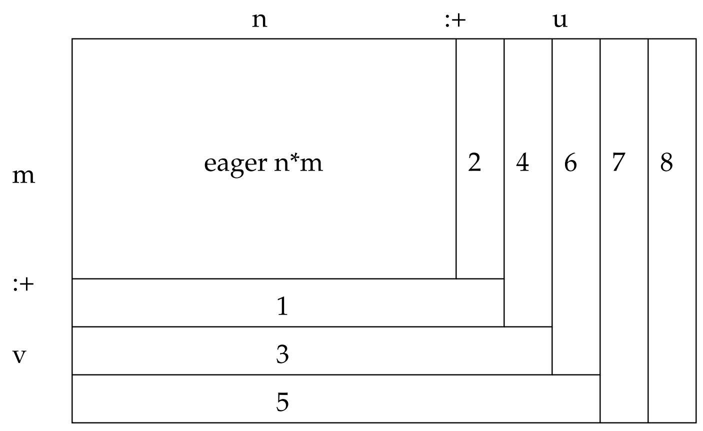

Figure 5.1: Diagonalized multiplication of a mixed natural. The numbers show the order in which the strips are laid down.

This definition still uses greedymul to finish off a multiplication once an argument has been exhausted. As before, no laziness is lost because the first argument to greedymul is fully evaluated.

The reader's eyebrow may have been raised by the use of fromInteger to turn our hard-won integer part into a successor chain. The reason for this is that the part of the result calculated by sb must be a successor natural to maintain laziness. An alternative formulation was tried, performing the computation with integers before converting the result into a successor natural. In practice this definition performs less well because if a program needs to call on the successor natural part of the result, then the whole result must be computed (because integer arithmetic is eager). In cases where the successor natural part of the result is not needed, there is no difference between the definitions.

5.4 Subtraction

Monus can be defined almost as before, thanks to mswitch:

$$(-) \quad = \quad \mathsf{mswitch}\ (\mathsf{k}\ 0)\ (\mathsf{k}\ 0)\ \mathsf{id}$$

It is easy to see from the definition that monus' left- and right-identity laws hold for mixed naturals just as they did for successor naturals:

$$\begin{aligned} \mathsf{x} - 0 &= \mathsf{x} \\ 0 - \mathsf{x} &= \mathsf{x} \end{aligned}$$

Any alternate subtractive operators work here just as they did with successor naturals, thanks to the three-way switch. As before, we also see that the investment in writing mswitch is amortised with monus, quotient and remainder, and the relational operators.

5.5 Quotient and remainder

Quotient and remainder also needs no more than cosmetic alteration (indeed, had we written 0 in place of Zero in the original definitions they could be repeated verbatim).

```
quotRem n m  =  mswitch (k (0, n)) (k (1, 0)) step n m where
  step n'    =  let (q :+ qu, r)=quotRem n' m in ((q+1) :+ qu, r)
```

The unusual behaviour seen with successor naturals is mirrored by the mixed naturals: though computing quotient and remainder of zero by zero gives (0, 0), any other value as the dividend gives an infinite quotient and an undefined remainder.

An infinite divisor again results in a zero quotient and a remainder equal to the dividend, for any finite dividend.

5.6 The relational operators

The relational operators are also unchanged beyond replacing switch with mswitch:

```
(==)  =  mswitch (k False) (k True) (k False)
(≤)   =  mswitch (k True) (k True) (k False)
(≥)   =  flip (≤)
(<)   =  (not .) . (≥)
(>)   =  (not .) . (≤)
```

5.7 Discussion

It is easy to see that the mixed naturals just presented are, in general, not equivalent to the successor naturals. The obvious definition of list length, for example, is only lazy with the successor naturals:

```
length [ ]       =  0
length (_ : xs)  =  1+length xs
```

The reason is that the integer part of the second argument to the addition isn't evaluated until the entire list has been examined, at which time the chain of ones propagates back. Without removing the eager addition from mixed natural addition, there's nothing we can do (but see chapter 7 for alternative definitions of list length).

Clearly, mixed naturals can degenerate to either integers (as here) or successor naturals, but in most cases are likely to lie somewhere in-between. The application programmer takes responsibility for exactly how lazy their naturals are, accepting this responsibility in return for the *possibility* of increased efficiency. To draw a comparison with fire: humans would never have become as dominant as they are today had they not mastered it, but if you play with fire, you have to expect to get hurt.

For an indication of whether we do in fact gain increased efficiency, see the examples in chapter 8.

Perhaps the most disappointing property of the mixed naturals is their inability to make full use of any evaluation they cause. The definitions presented in this chapter have not been able to rework evaluated successor chains into the integer part of either their arguments or their results. This inability to capitalise on – or even recognise – evaluation will be addressed in the next chapter.

Chapter 6

Implementation Assistance

Neither of the two implementations of lazy naturals presented thus far is ideal. The successor naturals are unable to offer maximal laziness while the mixed naturals don't even attempt to do so. At this point, it might seem that we can go no further.

As characters faced with such a situation in a Greek tragedy, we might call upon some *deus ex machina* for help. And that is precisely what we will do here. If you recall, the reason that the successor naturals are not maximally lazy is that there is no way of knowing the extent to which their arguments are already evaluated. Without that information there is little chance of us making a rational decision when faced with the dichotomy of which argument to select for case analysis while designing the operators. About the best we can hope for is consistency (e.g. we select the first argument) and a similarity with the analogous list functions (e.g. we select whichever argument our list analogue would).

Although the mixed naturals are an attempt to address the problem of time-efficiency, no attention has been given to the space-efficiency of lazy naturals. Successor naturals take up heap space proportional to their value and mixed naturals – being part successor natural – suffer the same fate whenever laziness is introduced.

This chapter concentrates on improving laziness, though an outline will be given of how the implementation assistance employed could also be applied to improving space-efficiency (and thereby time-efficiency).

6.1 A new primitive

Care must be exercised that we do not find ourselves carried away — this isn't a fairy-tale where we find ourselves granted three wishes. We are to restrict ourselves to a single wish on grounds of good taste. The problem is to decide what it is that would be of most use to us. Obvious candidates include:

- a means of measuring the degree of evaluation of an expression,
- a more efficient representation of successor chains on the heap,
- or a garbage collector aware of lazy naturals and able to normalise them (by collecting the evaluated part of successor chains and increasing the corresponding integer part appropriately).

Of these only the measure of degree of evaluation appears to have wider applicability, and for this reason it is such an approach that will be investigated.

For the purposes of implementing lazy naturals, it would suffice to know whether or not an expression has a Succ or Zero constructor outermost. Such an expression has been evaluated to a sufficient extent that we can perform case analysis (because our case analysis just asks whether we have Succ or Zero outermost).

If both arguments have a constructor outermost, then it is of no consequence which we select. If only one argument has a constructor outermost, it should be selected in preference to the other argument. If neither argument has a constructor outermost then we must use our wits and take our chances, as we did before. (That's to say: though we might well show a preference for one argument over another, the method is unlikely to be as systematic as we would hope for when using the new primitive.)

The prime candidate for the position as oracle is a primitive that tests whether or not an expression is in *weak head-normal form* (WHNF). An expression is in WHNF if it is a constant, a lambda expression or a partial application of a constant function. Our oracle will go against oracular tradition and give answers as unambiguous as true or false. Sometimes a metaphor can be taken too far!

6.2 The implementation

The Haskell interpreter Hugs does not provide any means by which a program can determine whether or not an expression is in WHNF. It does, however, provide a relatively simple mechanism by which one can add new primitives.

Hugs, like Gofer before it ([11]), evaluates expressions by traversing program graphs. Given the root of an expression, the evaluator reduces the graph to WHNF by working its way down the spine of the graph until it reaches the head. Depending on what it finds at the head, the evaluator either performs some computation (applying a function, for example) or returns the graph as already being in WHNF.

A WHNF primitive need only mirror the evaluator's work up to the point where the evaluator finds the head of the expression. The primitive can then inspect the head to determine whether its answer is true or false.

The abstract interpretation functions used in strictness analysis should satisfy the two important properties of safety and informativeness (see chapter 22 of [18] for more on abstract interpretation functions). Reformulating these properties to describe a WHNF primitive, we have that:

1. It *must* be 'safe': it should never declare an expression to be in WHNF when it isn't.

2. It *should* be as 'informative' as possible: it should detect expressions in WHNF in as many cases as possible.

To this end, the WHNF primitive errs on the side of caution: if it isn't *certain* that an expression is in WHNF, it returns false. In order that the primitive can return a Boolean result, false is reused for the cases where the best that can be said is "true or false".

The code itself is quite short (see figure 6.1), but it makes use of numerous Hugs macros. The macro `primArg` is used to access a primitive's arguments, numbered from one. We use `whatIs` to determine the type of a cell, while the macros `fun` and `arg` retrieve the function and expression from cells representing function applications and indirections (used to implement sharing) respectively. Testing whether a name corresponds to a constructor function is done with `isCfun`, and `updateRoot` overwrites a primitive's application with its return value.

6.3 Applications

There are few places where we can make use of the new `whnf` primitive. Looking through the successor naturals' implementation, we see that though `switch`, `monus`, and `qr` all have two arguments, they are always obliged to examine both. The `add` and `mul` functions are the only opportunities we have to apply `whnf`.

```
primFun(primWhnf) {
    Cell n = primArg(1);            /* Get our argument. */
    Bool isWhnf = FALSE;

    /* Traverse the expression until we find out whether it's in WHNF. */
enh:switch(whatIs(n)){
        case AP:
            /* An application: follow the graph and see what function */
            /* is being applied — it might be a constructor. */
            n = fun(n);
            goto enh;
        case INDIRECT:
            /* An indirection's value is the value of the expression */
            /* that it references, so we follow the indirection. */
            n = arg(n);
            goto enh;
        case NAME:
            /* A name is in WHNF only if it's the name of a constructor */
            /* function. */
            isWhnf = isCfun(n);
            break;
        case CHARCELL:              /* Character. */
        case INTCELL:               /* Integer literal. */
        case FLOATCELL:             /* Floating point literal. */
        case STRCELL:               /* String literal. */
            isWhnf = TRUE;
            break;
        /* Assume that anything else is not in WHNF. */
        default: break;
    }

    /* Replace the application of whnf with a truth value. */
    updateRoot(isWhnf ? nameTrue : nameFalse);
}
```

Figure 6.1: The implementation of the WHNF primitive.

Addition

A revised definition of addition using whnf need not arbitrarily select its first argument for case analysis:

```
add x y  =  if whnf x then add′ y x else add′ x y
            where
                add′ (Succ x) y  =  Succ (add x y)
                add′ Zero y      =  y
```

Multiplication

There are two different multiplication algorithms used by the mixed naturals: there's greedymul which we needn't consider since it is only ever applied to a fully-evaluated first argument, and there's the diagonalized multiplication.

Diagonalized multiplication can be more discriminating when choosing a successor chain to begin work on. If both are in WHNF, then the new definition chooses the one corresponding to the smaller integer to accrue the benefit of adding a strip the size of the larger integer. If neither is in WHNF, the same choice is made: if we have to cause evaluation, we want to get our money's worth. If only one of u and v is in WHNF then it is the one which is is selected for case analysis. The whole thing works by sorb ("s or b") being true or false, depending on whether work should start on the side or bottom:

```
(∗) (n :+ u) (m :+ v) = (n∗m) :+ rest
   where
    rest                    =  if eu == ev then pick (n<m)
                               else pick eu
    eu                      =  whnf u
    ev                      =  whnf v
    pick sorb               =  sb sorb (fromInteger m) (fromInteger n) u v
    sb _ s b Zero y         =  greedymul b y
    sb _ s b x Zero         =  greedymul s x
    sb True s b (Succ x) y  =  s+sb False s (Succ b) x y
    sb False s b x (Succ y) =  b+sb True (Succ s) b x y
```

Why do we only consider the degree of evaluation when deciding how to start? Why not expand the eagerly evaluated rectangle in both directions as far as evaluation that has already occurred will allow? The reason, of course, is that there is no guarantee that our successor chains aren't infinite.

Despite the cries of *"lasciate ogni speranza, voi ch'entrate"*[1] [6], we should at least consider this alternative now that we have come so far.

The auxiliary skim is used to skim off the evaluated part into the first component of its result, leaving the unevaluated successor natural as its second component.

```
skim (i, s)  =  if whnf s
                then case s of
                   Succ s' → skim (1 + i, s')
                   Zero → (i, s)
                else (i, s)
```

'Skimming' multiplication, then, can be thought of as applying skim to normalize the two mixed natural arguments to the multiplication. The side consists of an evaluated part s_i and an unevaluated part s_s, distinct from the original integer and successor natural parts (though *i* and *s* stand for integer and successor natural, respectively). The bottom is treated similarly, and the rest of the definition follows from our previous attempt:

```
(*) (n :+ u) (m :+ v) = ready :+ rest
   where
    ready                 =  s_i * b_i
    rest                  =  sb (s_i > b_i) (fromInteger s_i)
                                (fromInteger b_i) b_s s_s
    (b_i, b_s)            =  skim (n, u)
    (s_i, s_s)            =  skim (m, v)
    sb _ s b Zero y       =  greedymul b y
    sb _ s b x Zero       =  greedymul s x
    sb True s b (Succ x) y  =  s+sb False s (Succ b) x y
    sb False s b x (Succ y) =  b+sb True (Succ s) b x y
```

Do we have to worry about infinite values? No we don't, as it happens. When the whnf primitive travels along a cyclic value, the cycle – the reference to the structure by its name – is visible. Even mutually cyclic structures are no problem: the implementation still sees a reference to a name that isn't a constructor function. Although a Haskell program has no direct access to this information, it's clearly visible to the primitive. In the C code for the primitive that was presented in the previous section, we come across a NAME cell where the name is not a constructor function.

There is still one dark cloud left on our horizon. What about knot-tying cycles? In a functional programming language using graph-reduction,

[1] Abandon all hope, ye who enter.

there are two obvious structures to use when representing a function such as:

$$\mathsf{knot}\ \mathsf{f} \;=\; \textbf{let}\ \mathsf{x} = \mathsf{f}\ \mathsf{x}\ \textbf{in}\ \mathsf{x}$$

Either we have a graph of repeated applications of f (so the definition is cyclic, but the structure isn't) or we have a graph where the argument of the application of function f is the application itself, giving a truly cyclic structure. This second case is depicted in figure 6.2.

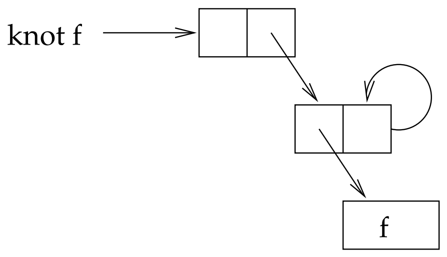

Figure 6.2: The structure of a knot-tied cycle. The topmost cell is an indirection cell.

We can allow ourselves a smug grin at this point. What was the biggest problem facing us when applying the whnf primitive has (almost) vanished into thin air. We need no longer be scared of cyclic structures, unless we have a knot-tied cycle in our graph.

Mixed naturals

The mixed naturals simply make use of the successor naturals to operate on their SuccNat components. Their laziness comes about by virtue of the successor naturals' laziness.

As was hinted in the introduction to this chapter, whnf could also be used to improve the space-efficiency of successor naturals in the context of mixed naturals. A normalization function could be imagined, where long chains of Succ constructors are gathered up into the Integer component of the mixed naturals, similar to the operation of skim in the last definition of multiplication found in the previous section.

We can write norm quite easily in terms of skim (assuming that skim is seen to be of sufficient generality to be lambda-lifted out of multiplication):

$$\mathsf{norm}\ (\mathsf{e} :\!+\ \mathsf{u}) \;=\; \textbf{let}\ (\mathsf{e}', \mathsf{u}') = \mathsf{skim}\ (\mathsf{e}, \mathsf{u})\ \textbf{in}\ (\mathsf{e}' :\!+\ \mathsf{u}')$$

Given that there is little problem with cyclic structures, the question is not when to stop but rather when to start. Knowing at what point it makes most sense to normalize is quite difficult. To avoid unnecessary loss of laziness, the power, and therefore the responsibility, has to be in the hands of the programmer. Without adding extra syntax to the language, use of normalization would greatly detract from a program's readability.

The only example use of (something similar to) `norm` in the mixed naturals is the multiplication already mentioned.

6.4 Summary

The addition of a new primitive to the Hugs Haskell interpreter has allowed us to make an omelette while only breaking eggs if need be. The use of `whnf` allows increased laziness at those points where an arbitrary decision between arguments used to take place.

The fact that there were only two places in the implementation of mixed naturals where the new primitive could be used is pleasing in that we haven't had to tarnish much of the implementation with our implementation-dependent feature.

Chapter 8 discusses the effect on performance of using the WHNF primitive.

Chapter 7

Applications in the Haskell Prelude

In this, the second part of the report, the focus switches from the implementation of lazy naturals to their use. Starting with this chapter, examples are presented of lazy naturals in situations where their presence is anything from invaluable to intolerable!

There are two distinct properties of lazy naturals to be explored. Firstly, they guard against accidental loss of laziness. That's why they're called lazy. Secondly, they lead to definitions with fewer equations and fit neatly into inductive proof. That's the natural bit.

When considering laziness, we might like to examine the following points:

- Where can use be made of laziness? Operations over lazy data structures are the obvious candidates, but is that all?

- Laziness allows computation over infinite structures in certain cases: is that ever useful? That is: is it as useful to be able to define functions that compute infinite results from infinite data as it is to be able to define the infinite data structures in the first place?

- Does lazy arithmetic simplify the task of writing lazy programs? Is it possible to take a lazy but complicated program and remove complexity by using lazy naturals?

On the topic of naturalness, interesting questions include:

- Is it easier to spot an incomplete or incorrect definition using explicit successor notation or using integers, $(\mathsf{n} + \mathsf{k})$ patterns and additional guards?

- Is notation like (Succ n) an acceptable alternative to (n + k) patterns?
- Monus is an interesting alternative to subtraction, but is it the only useful "natural subtraction"?

We begin by considering the Haskell prelude. The way in which prelude functions embody the most common numerical computations allows many functional programs to make very little direct use of arithmetic. To help programmers take best advantage of the laziness of lazy naturals, the prelude functions should be examined for eagerness. At the same time, other modifications to the standard Haskell prelude will be explored, hopefully resulting in simpler definitions.

In this chapter excerpts from the standard Haskell prelude are presented using `monospaced` type, while their replacements use sans-serif type.

7.1 List index (subscript)

The list index operator is usually defined as follows:

```
(x:_)  !! 0          = x
(_:xs) !! n | n > 0  = xs !! (n-1)
(_:_)  !! _          = error "!!: negative index"
[]     !! _          = error "!!: index too large"
```

The use of lazy naturals precludes the possibility of a negative index, meaning that we can define index with just three equations and without the need for a guard on the recursive equation:

```
index (x : _) Zero      = x
index (_ : xs) (Succ n) = index xs n
index _ _               = error "index : index too large"
```

7.2 List length

List length, despite being much used as an example in introductions to functional programming, is the least amenable to rewriting of any of the prelude functions. The definition in Hugs' prelude is eager with both implementations of lazy naturals because of the left fold:

```
length = foldl' (\n _ -> n + 1) 0
```

The Haskell definition works for successor naturals, but not for mixed naturals. It fails there because the numbers it adds consist solely of integer parts, and the addition degenerates to eager addition of built-in integers:

```
length []     = 0
length (_:xs) = 1 + length xs
```

If we're prepared to accept a mixed-natural specific definition of length, the following will suffice:

$$\mathsf{length\ xs} \;=\; \mathsf{0 :\!+ \; foldr\ (\lambda\_ \to (1+))\ 0\ xs}$$

This is the opposite degeneration: where the computation works as if a mixed natural were a successor natural with some baggage to carry around.

7.3 List prefix and suffix

List prefix and suffix are both similar in definition to one another. List prefix (take) takes the first *n* elements from its second argument, where *n* is its first argument. List suffix (drop) takes all but the first *n* elements from its second argument, where once again *n* is its first element.

The standard Haskell definitions require four equations each:

```
take 0 _              = []
take _ []             = []
take n (x:xs) | n > 0 = x : take (n-1) xs
take _     _          = error "take: negative argument"

drop 0 xs             = xs
drop _ []             = []
drop n (_:xs) | n > 0 = drop (n-1) xs
drop _     _          = error "drop: negative argument"
```

The lazy natural definitions are, by contrast, much simpler. Only two equations are required. As usual, no exceptional equation is needed for the negative case, and the observation that for non-positive cases the result is equal to the second argument reduces two of the remaining equations to one:

$$\begin{array}{lcl} \mathsf{take\ (Succ\ n)\ (x : xs)} & = & \mathsf{x : take\ n\ xs} \\ \mathsf{take\ \_\ \_} & = & \mathsf{[]} \\ \\ \mathsf{drop\ (Succ\ n)\ (\_ : xs)} & = & \mathsf{drop\ n\ xs} \\ \mathsf{drop\ \_\ xs} & = & \mathsf{xs} \end{array}$$

The availability of infinity as a usable value allows two new laws to be stated:

$$\begin{aligned} \mathsf{drop}\ \infty\ \mathsf{xs} &= [] \\ \mathsf{take}\ \infty\ \mathsf{xs} &= \mathsf{xs} \end{aligned}$$

No claim is made for the usefulness of these laws, but it's nice to know they're there.

7.4 List split

The function splitAt splits a list into a pair of lists. The size of the first list is determined by the first argument to splitAt. The second list contains the remainder of the argument list:

```
splitAt 0 xs      = ([],xs)
splitAt _ []      = ([],[])
splitAt n (x:xs)
      | n > 0     = (x:xs',xs'')
                    where
                    (xs',xs'') = splitAt (n-1) xs
splitAt _ _       = error "splitAt: negative argument"
```

The lazy natural solution has no need of the special case of a negative argument, and once again it is possible to combine two of the cases and present the positive case first:

$$\begin{aligned} \mathsf{splitAt}\ (\mathsf{Succ}\ \mathsf{n})\ (\mathsf{x} : \mathsf{xs}) &= (\mathsf{x} : \mathsf{xs}', \mathsf{xs}'') \\ \quad \textbf{where} &\quad (\mathsf{xs}', \mathsf{xs}'') = \mathsf{splitAt}\ \mathsf{n}\ \mathsf{xs} \\ \mathsf{splitAt}\ \_\ \mathsf{xs} &= ([\,], \mathsf{xs}) \end{aligned}$$

Note how the new definition excludes silly applications — despite what Hugs (for example) thinks, there are no minus three initial elements of the empty list:

```
? splitAt (-3) ([]::[Int])
([],[])
```

Being allowed to take or drop a negative number of list elements is of dubious utility. Indeed, it's easy to suspect that the use of monus rather than subtraction would prevent the situation from ever arising. An implementation of enumFromTo (aka '..') might take from an infinite list, for example. Subtraction would require take to be defined for a negative number of list elements so that an expression like 6..4 would reduce to the empty list. Monus, on the other hand, would require only that a take of zero elements be defined.

The Haskell definition (using integers) could also have been written with two equations. Perhaps the explicit use of naturals encourages one to think – and program – in terms of them? One thing that an integer could never offer is the law:

$$\mathsf{splitAt}\ \infty\ \mathsf{xs} \quad = \quad (\mathsf{xs}, [\,])$$

7.5 Discussion

I think that even these short examples demonstrate the utility of a *natural* type over an integer type. All of the computations here were more sensibly defined with naturals than integers: a fact recognised by the writers of the Haskell prelude who included all the checks for negative arguments. The fact that the functions' type signatures more sensibly contain naturals than integers does not necessarily imply that the natural *definitions* are better. Though in all but the case of list length, I would argue that they are.

Although standard Haskell allows us to write definitions using case analysis on the naturals by means of the patterns 0 and (n+k), this requires extra syntax and an underlying evaluation machine extended to support (n+k) patterns as a special case. The constructed lazy naturals make use of the standard mechanisms for constructed types.

Only with list length was laziness a real advantage, though depending on the computation of the number of items required, take and splitAt could also benefit. The drop function, and its special-case, the indexing operator, are unable to gain any laziness: there is no way that they can produce a result until they have seen their 'index' in full.

It is interesting just how difficult it was to arrive at a lazy implementation of list length with mixed naturals. It would have been nice to include definitions of sum and product, minimum and maximum in this chapter but their definitions proved even less amenable to conversion to mixed natural form.

This is the price that one pays for increased efficiency. Exactly what increase in efficiency we gain is covered in the next chapter.

Chapter 8

Other Applications

This chapter takes a number of longer functional programs and explores the applicability of lazy naturals to such real-world situations. The aim is to present a representative sample of functional programs to help assess those areas to which lazy naturals are most (and least) well-suited. Some programs are from published papers and books, while others were tailor-made to demonstrate a particular point.

The performance figures given in this chapter were taken from Mark Jones' Hugs[1] interpreter version 1.3β (incorporating the WHNF primitive described in chapter 6). The hardware and system software used is not relevant. Because the raw numbers have a tendency to be large, ratios are given in parentheses in each of the tables. Where mention is made of "time performance", this is assumed to be directly proportional to number of reduction steps.

8.1 Word count

The `wc` program is similar to the UNIX utility of the same name. It counts lines, words and characters in the specified files. For example:

```
? main ["hugs_diff.hs","diff.hs"]
        62       425      2162 hugs_diff.hs
        72       469      2396 diff.hs
```

The program is typical of a class of small but useful programs that perform only basic arithmetic (in this case, nothing more than addition). It is

[1]See `http://www.cs.nott.ac.uk/Department/Staff/mpj/hugs.html` for details of Hugs, including downloading information.

```
main (f:fs) = readFile f >>= \s ->
putStr (wc s) >> putStrLn (' ':f) >> main fs
main [] = done

-- showIn shows n in a field of width w, padded to the
-- left with spaces.
showIn w n =
    (replicate (w - length (show n)) ' ') ++ (show n)

data LWC = LWC !Nat !Nat !Nat

wc = wc' (LWC 0 0 0) False

wc' (LWC l w c) _ [] =
    (showIn 10 l) ++ (showIn 10 w) ++ (showIn 10 c)
wc' (LWC l w c) inAWord (x:xs)
    | isSpace x = let l' = if x=='\n' then l+1 else l in
        wc' (LWC l' w (c+1)) False xs
    | otherwise = let w' = if inAWord then w else w+1 in
        wc' (LWC l w' (c+1)) True  xs
```

Figure 8.1: The source to wc.

highly unlikely that this implementation of wc could benefit by using lazy naturals — the paper from which it is taken ([20]) demonstrates that the very reason for its efficiency is the use of Haskell's strictness annotations to force evaluation of the three counts, rather than storing up the computation until asked to display the results. The source to a version modified for Hugs is shown in figure 8.1.

Table 8.1 compares the number of reduction steps and heap cells required by four versions of the program. All four versions are the same basic program, the only difference being that a type synonym Nat was allowed to vary over the three types Integer, SuccNat and MixNat (once with each of the two implementations of MixNat).

Of particular interest is the huge performance difference between the successor naturals and mixed naturals. Though the successor naturals are two orders of magnitude worse than the integers in both time and space, the mixed naturals are within a factor of two or three of the integers. The mixed naturals making use of the WHNF primitive perform slightly less well because their defining equations are longer and more numerous, re-

Type	Reductions		Cells	
Integer	32908	(1.00)	87107	(1.00)
SuccNat	3555800	(108.05)	7082293	(81.30)
MixNat	83530	(2.54)	175703	(2.02)
WHNF	89854	(2.73)	182027	(2.09)

Table 8.1: Comparison of `wc` on `succ.hs` using four different numeric types.

quiring more reduction steps. This could well be optimised away by an intelligent compiler.

Experiments using integers and mixed naturals with various files taken as input to `wc` show that the ratios (figures in brackets) remain very similar, indicating that the mixed naturals do not affect the program's time or space complexity other than the constant multiple of extra effort required by the mixed naturals' implementation.

The figures bear out our hopes for the mixed naturals in relation to their successor natural cousins: their performance is significantly better. The worsening of time performance by a factor of three and in space performance by a factor of two when compared with built-in integer arithmetic are both well within an order of magnitude. This is a typical goal of bignum implementations. To quote [23]:

> [The] goal has been to be *not* two orders of magnitude slower than 'best possible' [referring to a hand-coded implementation by experts], and *not* an order of magnitude slower, but something within the range of a half order of magnitude — something more akin to the relative discrepancy observed between highly "tuned" machine language and the output of a good compiler.

Examining space performance first, it is easy to see that the space required by a mixed natural is the sum of the space required by an integer, the space required by a successor natural, the space required by the :+ constructor function and the space required by all the application cells that glue the structure together (see chapter 10 of [18] for more information on concrete representations of functional programs). In this program the successor natural parts remain zero throughout. For this reason, mixed naturals require more space than do integers. With the Hugs interpreter, they require five times the space (see figure 8.2).

As for time performance, for every n reduction steps that integer addition requires, mixed natural arithmetic requires these steps (recall the

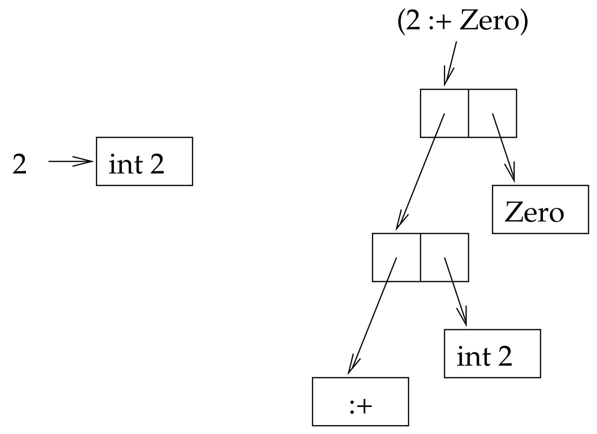

Figure 8.2: A comparison of the structure of the fully-evaluated integer 2 and the fully-evaluated mixed natural (2 :+ Zero) when using Hugs. These structures were confirmed by the addition of a structure-displaying primitive.

definition of mixed natural addition in section 5.2) plus the reduction of the equation for mixed natural addition, plus the application of the :+ constructor function to the result.

Although there is no good reason to use lazy naturals in this program, it is comforting to observe that the penalty for doing so is not too severe. A factor of two or three is soon swallowed up by the increased performance of successive years' machines.

8.2 Page up

A common idiom in C (and other languages devoid of any natural type) is that of assigning the maximum of zero and some expression to a variable. As an example, take the case of a document editor with a "page up" facility. The idea is that the display should move 'up' a page (implying that pages are shown vertically in sequence) at the user's request. An obvious analogous implementation in Haskell would be:

```
type Position = (Nat, Nat)

pageHeight = (1024::Nat)

pageUp :: Position -> Position
pageUp (x, y) = (x, y - pageHeight)
```

Imagine the case where the current y-position is less than pageHeight from the beginning of the document. There isn't anything to see before the document, so we should only jump back as far as zero. Using Haskell's integer type for our naturals, we can't do this. The y-position would wrap around and leave the user at the opposite end of the document (or more likely, past the end of the document). The usual solution when using signed integers is to write something of the following form:

```
pageUp (x, y) = max (x, 0) (x, y - pageHeight)
```

This is quite clearly the behaviour of monus: the 'obvious' implementation given above would have been correct if a natural type had been used in place of integers.

Using fully-evaluated mixed naturals, we suffer only a small constant time overhead in comparison with the integer implementation using max; about three times as many reductions. We gain the freedom to say what we mean and have our implementation worry about the details.

8.3 File difference

The authors of [20] present a Haskell implementation of an algorithm due to Allison, modified to emulate the UNIX `diff` program: a differential file comparator.

In his paper [1], Allison notes that the computation of min3 is of paramount importance in maintaining laziness. In general, all three parameters would need to be examined, but extra knowledge of the relationship between the three variables allows him to optimize away some comparison.

It would be unreasonable to expect lazy naturals to compete against human ingenuity, but lazy naturals may still have something to add. There is an interesting subtraction in the first line of the program (note that this is an excerpt from a larger expression):

```
diagFor (length xs - length ys)
```

The function diagFor is defined:

```
diagFor 0         = prince
diagFor d | d > 0 = lowers !! ( d-1)
          | d < 0 = uppers !! (-d-1)
```

It might seem unfortunate that monus appears to be useless in this case: we want to know the difference between the two lengths. The observant reader – I've always wanted to write that – will have noticed that our three-way switch embodies this pattern exactly. Using lazy naturals, we have:

```
mswitch  (λ d → uppers!!(d−1))
         (k prince)
         (λ d → lowers!!(d−1))
         (length xs) (length ys)
```

Here the subtractions of 1 from d are performed by monus, and we know that in both cases d is at least 1. The generality of the three-way switch allows it to be used in unanticipated situations. Offering the programmer access to this function is preferable by far to attempting to enumerate all of its possible applications. (A full listing of the modified program is to be found in appendix A.)

8.4 Number theory

At the other end of the spectrum, we might expect a program dealing almost exclusively with arithmetic to suffer more when lazy naturals are used. A simple example is generating the first four perfect numbers:

```
? take 4 (filter perfect [1..])
[1, 6, 28, 496]
```

Here are the necessary definitions (as a reminder, a perfect number is one whose proper divisors add up the the number itself):

```
perfect :: Nat → Bool
perfect n          =  n == sum (properDivisors n)

properDivisors :: Nat → [Nat]
properDivisors n   =  1 : [x | x ← [2..n − 1], n'rem'x == 0]
```

As one would expect from the increased use of arithmetic, the lazy naturals have a harder time of things here. Even the mixed naturals are an order of magnitude worse than the built-in integers:

The reason behind the mixed naturals' disappointing showing is the use of the Haskell prelude's eager sum. Had a lazy sum been developed in the previous chapter, the mixed naturals' performance would have improved. (In fact, switching from a left to a right fold in the standard definition offers some improvement, but that effect is only minor.)

Type	Reductions		Cells	
Integer	1731725	(1.00)	3950988	(1.00)
SuccNat	44754512	(25.84)	58221703	(14.74)
MixNat	27185272	(15.70)	79932570	(20.23)
WHNF	27432455	(15.84)	81141492	(20.54)

Table 8.2: Comparison of finding the first four perfect numbers using two different numeric types.

8.5 Discussion

It is unfortunate that lazy algorithms don't lend themselves to complexity analyses. The time and space required by the evaluation of an expression depends not only on the input (something we have to contend with even in imperative programming) but also the context in which the result is used. When a result is never used, we achieve a remarkable best-case figure!

That said, it is clear that the mixed naturals outperform the successor naturals, both in time and space. It could be said that those programs which make light use of arithmetic (such as `wc`) do not suffer too greatly when using mixed naturals, but that those programs making heavy use of arithmetic (such as the perfect numbers example) do. What should be borne in mind, though, is that `wc` only used arithmetic operations for which lazy definitions were available, while the perfect numbers example make use of an eager `sum`.

We have seen that though the mixed naturals are more efficient in both time and space than the successor naturals, where we fail to avoid eagerness we can expect to pay a high price.

The next, final, chapter looks again at all the implementations of lazy naturals, both relative to one another and relative to the original aims of the project. Suggestions are also made of future work that might prove profitable.

Chapter 9

Conclusion

9.1 Comparisons

Laziness

All solutions offer at least some degree of laziness. Although we have to be careful of such eagerness-inducing activities as left folds, the successor naturals in particular offer an easy route to laziness.

The mixed naturals are a different kettle of fish: the competing desires to on the one hand make best use of any fully-evaluated integers and on the other maintain as much laziness as possible are a great source of difficulty. The eager integer arithmetic is at our heels once again, forcing us to be exercise (what should be) undue care and attention. With successor naturals, we can forget such concerns. The moment we switch to mixed naturals for efficiency's sake, we shy from using successor naturals at all.

The WHNF primitive offered extra laziness: the ideal would have been successor naturals making use of the WHNF primitive, if only the performance hit were less harsh. It is hard to conceive of a lazier natural type than this without knowledge of the relative expense of the computations that constitute unevaluated expressions.

None of the programs in chapter 8 made extensive use of the laziness of the lazy naturals, and it would be interesting to see whether or not the lazy naturals have their "killer application".

A constructed type

The implementation of lazy naturals as constructed types was both good and bad. The successor naturals had elegant associated definitions: a `Zero` case and a `Succ n` case. A slightly modified syntax to our language would allow us to write programs in exactly the form used by Landau for his proofs (using prime to represent successor, and the numeral 0 to represent zero).

Explicit use of a successor notation is a viable alternative to the $(n + k)$ pattern, at least where $k = 1$. It is easy to have confidence in such definitions, knowing that there are no negative cases to worry about and that for once structural induction and induction over the naturals were one and the same.

With a constructed type, there is no need for $(n + k)$ pattern support in the Hugs virtual machine.

The "double construction" of mixed naturals was an annoyance. It is something that the implementor of the arithmetic module *must* face, but which the user shouldn't need to. Unfortunately, laziness comes about through the successor natural part, so it needs to be accessible to the programmer. The definition of list length in chapter 7 is a good example of this: had we not had access to the two parts of the mixed natural, we would have been unable to formulate a lazy definition.

Arbitrary precision

The size of a successor natural is limited only by available memory, while the mixed naturals rely on the availability of arbitrary precision integers in the Haskell implementation. This might seem an unwarranted assumption, but the Haskell 1.3 standard guarantees the availability of arbitrary precision integers, considering them a "basic type".

Efficiency

As already mentioned, the mixed naturals are generally more efficient than the successor naturals (both in time and space). This increased efficiency comes at a price: that of increased reliance on the programmer, of increased complexity and difficulty of definition.

Lazy naturals are unlikely ever to compete with eager naturals in performance terms, but the mixed naturals are certainly efficient enough for much internal data-structure work. If it weren't so difficult to construct lazy definitions of certain functions (the important group of functions that perform some operation based on an entire list seem to belong to this class) then the mixed naturals would be entirely adequate.

As it is, the programmer has to ask themselves how important the laziness of a particular definition is.

Another efficiency consideration is the fact that some of the recursive definitions used with lazy naturals can require large amounts of space on the control stack. To make serious use of successor naturals, Hugs would have to be altered to dynamically increase the size of the control stack to avoid constantly overflowing it.

User-interface

Haskell's class system allows lazy naturals to fit neatly into the standard system of types. The class hierarchy seemed all-too-clearly to reflect the functions that the designer wanted to incorporate, but – apart from such notable examples as the inclusion of negate in class Num – the standard hierarchy accepted naturals without complaint.

The fromInt and fromInteger functions help by facilitating the use of numeric literals in programs. Above Succ Zero, the alternative becomes unwieldy.

The lack of ad-hoc function overloading can be seen as a shame: it prevents us from having different implementations of functions for different natural types unless we introduce a new class.

With the mixed naturals, the :+ representation needed to be available to offer laziness in all cases. The programmer has no choice but to grasp the nettle and understand the implementation if they want to be certain of laziness. With the successor naturals, the system can be trusted on its own.

9.2 Summary

Just as the LISP community 'settled' on including both fixnums and bignums in their language, future functional programming languages could offer a similar choice between fixnums (Int), bignums (Integer) and 'lazynums' (the WHNF-using MixNat?).

The successor naturals have ideal laziness properties, and they are a pleasure to program with. The mixed naturals generally have reasonable time and space efficiencies. Unfortunately, neither is satisfactory from both points of view.

9.3 Further work

The most serious problem facing the implementor of lazy naturals is not laziness but efficiency. The mixed naturals attempt to address this, but fall short of being a totally acceptable solution.

A different internal representation of mixed naturals could cut down on their space usage. A specific MIXNAT cell with pointers to an integer cell and a chain of successor natural constructors would be an obvious example.

There may be alternatives to constructed types, but the fact that any reasonable implementation of a functional programming language includes constructed types and the associated pattern-matching mechanism makes the use of a constructed type an elegant solution.

Even were we able to solve the problem of space inefficiency, we would still have to deal with the fact that the mixed naturals are difficult to use. This is, I believe, inherent in the structure of mixed naturals. Maybe if mixed naturals were – even in the worst case – not noticeably less space efficient than successor naturals, this problem would attract less attention. We could degenerate to a successor natural solution, and be thankful that there are cases where we do experience an efficiency gain.

If a language were to include numerous numeric types, would abstract interpretation enable a clever compiler to intuit which type was most suitable at a particular point in a program? If, as with strictness analysis, we had a test to tell us how much precision we need, whether we need laziness, etc., we could hand over the responsibility for choosing an implementation of the natural numbers to the compiler or interpreter. Of course, we would also require transformations that take definitions using one type to definitions using another type. This is likely to prove difficult in practice, witness for example my inability to produce an efficient and lazy definition of `sum`. (Such an idea was presented in [19], but was not addressed in this report.)

On a different tack, not directly connected with lazy naturals, it might be interesting to explore wider uses of the WHNF primitive. Though something to be used only sparingly, the WHNF primitive offers extra laziness when it is most important — and we only optimize when we really need to, don't we? It would also be interesting to make a better study of the usefulness of the WHNF primitive to the lazy naturals. There is a marked absence of programs relying on the laziness of their naturals in chapter 8.

Definitions

Enough of Peano, here are *i miei definizioni...*

Accumulating parameter An auxiliary parameter added to a function to accumulate a result.

Algebraic type A type defined by a finite number of type constructors: a sum-of-products type. Also called structured types or free data types.

Continuation A function used to 'continue' (usually finish) the computation of another. Continuation passing is a method of directing flow-of-control.

η**-reduction** Elision of a function's outermost parameter. Valid because $\lambda x.E\ x = E$ if x is not free in E.

K-combinator Cancellator combinator. Discards its second argument: $K = \lambda x.\lambda y.x$.

Strictness A function is strict in a parameter if it requires some evaluation of that parameter. A function is **hyper-strict** in a parameter if it requires that the argument be fully evaluated.

Weak head-normal form An expression is in WHNF if it is a constant, a lambda expression or a partial application of a constant function.

Bibliography

[1] Allison, L. *"Lazy dynamic programming can be eager"*, Information Processing Letters **43** (1992) 207-212.

[2] John Barnes (1996) *"Programming in Ada 95"*, Addison-Wesley.

[3] Richard Bird & Philip Wadler (1988) *"Introduction to Functional Programming"*, Prentice Hall.

[4] Luca Cardelli (1993) *"Typeful Programming"*, Digital SRC Research Report 45.

[5] Alonzo Church (1941) *"The Calculi of Lambda Conversion"*, Princeton University Press.

[6] Dante Alighieri (1308) *"La Divina Commedia, I: Inferno"*, canto III.

[7] James Gosling, Bill Joy & Guy Steele (1997) *"The Java Language Specification"*, Addison-Wesley.

[8] Ronald L. Graham, Donald E. Knuth & Oren Patashnik (1994) *"Concrete Mathematics"*, Addison-Wesley.

[9] The Haskell 1.3 Report (1996) `http://haskell.cs.yale.edu/haskell-report/haskell-report.html`.

[10] Kathleen Jensen & Niklaus Wirth (1978) *"Pascal User Manual and Report"*, Springer.

[11] Mark P. Jones (1994) *"The implementation of the Gofer functional programming system"*, Yale Research Report YALEU/DCS/RR-1030.

[12] Donald E. Knuth (1981) *"The Art of Computer Programming: Seminumerical Algorithms"*, Addison-Wesley.

[13] Edmund Landau (1970) *"Grundlagen der Analysis"*, Wissenschaftliche Buchgesellschaft.

[14] Zohar Manna & Richard Waldinger (1985) *"The Logical Basis for Computer Programming, Volume 1: Deductive Reasoning"*, Addison-Wesley.

[15] Robin Milner, Mads Tofte & Robert Harper (1990) *"The Definition of Standard ML"*, MIT Press.

[16] Giuseppe Peano (1889) *"Arithmetices principia nova methodo exposita"*.

[17] Giuseppe Peano (1921) *"Le definizioni in mathematica"*, Periodico di mathematiche (4), 1 (1921), pp. 175-89.

[18] Simon L. Peyton Jones (1987) *"The implementation of functional programming languages"*, Prentice Hall.

[19] Colin Runciman (1989) *"What about the* natural *numbers?"*, Comput. Lang. Vol. 14, No. 3, pp. 181-191.

[20] Colin Runciman & Niklas Röjemo (1996) *"Heap Profiling for Space Efficiency"*, Springer LNCS 1129, Lecture notes of 2nd International Summer School on Advanced Functional Programming, Olympia WA, August 1996, pp. 159-183.

[21] Guy Steele (1984) *"Common LISP: The Language"*, Digital Press.

[22] Raymond Turner (1991) *"Constructive Foundations for Functional Languages"*, McGraw-Hill.

[23] Jon White (1986) *"Reconfigurable, Retargetable Bignums: A Case Study in Efficient, Portable Lisp System Building"*, Proceedings of the 1986 ACM Conference on LISP and Functional Programming, ACM Press.

[24] Niklaus Wirth (1985) *"Programming in Modula-2"*, Springer.

[25] Niklaus Wirth & Jürg Gutknecht (1992) *"Project Oberon — The Design of an Operating System and Compiler"*, Addison Wesley.

Appendix A

Collected Source Code

The source code for the implementations of lazy naturals presented in chapters 4-6 begins overleaf.