

What About the Natural Numbers?

JMCT

N

N?

“Some thirty years into the history of machine-independent programming language design, the treatment of numbers is still problematic.”

— *Colin Runciman, 1989*

“Some sixty years into the history of machine-independent programming language design, the treatment of numbers is still problematic.”

— *Me, just now*

Main takeaway

The number system we use should relate to the structures of the problem we're solving.

Main takeaway

For some domains, the use of Reals¹ may be appropriate:

¹or their approximation via **Floats**

Main takeaway

For some domains, the use of Reals¹ may be appropriate:

- e.g. physics calculations involving volume, speed, or mass

¹or their approximation via **Floats**

Main takeaway

For *many* problems **Integers** are appropriate:

Main takeaway

For *many* problems **Integers** are appropriate:

- Fixed-precision DSP

Main takeaway

For *many* problems **Integers** are appropriate:

- Fixed-precision DSP
- Bank account balance :’(

Main takeaway

Runciman's argument:

For many of the discrete structures involved in the day-to-day practice of programming, the **natural** numbers are the most appropriate number system.

How?

In the process of exploring the Natural Numbers, we'll be developing an API. As we progress we'll see how different representations affect our API.



#goals

#goals

1. Show you that the [lazy?] Ns are *Good* and *Proper*

#goals

1. Show you that the [lazy?] Ns are *Good* and *Proper*
2. Demonstrate that even *simple* choices of types for an API have deep consequences

#goals

1. Show you that the [lazy?] Ns are *Good* and *Proper*
2. Demonstrate that even *simple* choices of types for an API have deep consequences
3. Have you asking “What about the Natural Numbers?” next time you create an API.



Shape of things to come

Shape of things to come

1. Overview of the Ns themselves

Shape of things to come

1. Overview of the \mathbb{N} s themselves
2. Programming with `Nat`

Shape of things to come

1. Overview of the \mathbb{N} s themselves
2. Programming with `Nat`
3. Arithmetic with `Nat` and properties we care about

Shape of things to come

1. Overview of the `Ns` themselves
2. Programming with `Nat`
3. Arithmetic with `Nat` and properties we care about
4. How does `Nat` influence API design?

Shape of things to come

1. Overview of the `Ns` themselves
2. Programming with `Nat`
3. Arithmetic with `Nat` and properties we care about
4. How does `Nat` influence API design?
5. Implementation concerns

Shape of things to come

1. Overview of the `Ns` themselves
2. Programming with `Nat`
3. Arithmetic with `Nat` and properties we care about
4. How does `Nat` influence API design?
5. Implementation concerns
6. Beyond `Nat`

Shape of things to come

1. Overview of the `Ns` themselves
2. Programming with `Nat`
3. Arithmetic with `Nat` and properties we care about
4. How does `Nat` influence API design?
5. Implementation concerns
6. Beyond `Nat`
7. Conclude

Let's start

1. Overview of the `Ns` themselves
2. Programming with `Nat`
3. Arithmetic with `Nat` and properties we care about
4. How does `Nat` influence API design?
5. Implementation concerns
6. Beyond `Nat`
7. Conclude

What *are* they?

The *Natural* numbers have a few definitions:



What *are* they?

The *Natural* numbers have a few definitions:

1. Set Theoretic

What *are* they?

The *Natural* numbers have a few definitions:

1. Set Theoretic
2. Peano Axioms

Setting Yourself Up For Success

Setting Yourself Up For Success

Several possible Set-theoretic definitions, von Neumann proposed the following:

Setting Yourself Up For Success

Several possible Set-theoretic definitions, von Neumann proposed the following:

- $0 = \{\}$

Setting Yourself Up For Success

Several possible Set-theoretic definitions, von Neumann proposed the following:

- $0 = \{\}$
- $1 = 0 \cup \{0\}$

Setting Yourself Up For Success

Several possible Set-theoretic definitions, von Neumann proposed the following:

- $0 = \{\}$
- $1 = 0 \cup \{0\}$
- $2 = 1 \cup \{1\}$

Setting Yourself Up For Success

Several possible Set-theoretic definitions, von Neumann proposed the following:

- $0 = \{\}$
- $1 = 0 \cup \{0\} = \{0\} = \{\{\}\}$
- $2 = 1 \cup \{1\} = \{0, 1\} = \{\{\}, \{\{\}\}\}$

Setting Yourself Up For Success

oof

Setting Yourself Up For Success

In 1889 Giuseppe Peano published

“The principles of arithmetic presented by a new method”

Setting Yourself Up For Success

The two axioms we care about most (right now) are simple enough:

Setting Yourself Up For Success

The two axioms we care about most (right now) are simple enough:

- $0 \in \mathbb{N}$

Setting Yourself Up For Success

The two axioms we care about most (right now) are simple enough:

- $0 \in \mathbb{N}$
- $\forall n \in \mathbb{N}. S(n) \in \mathbb{N}$

Sign post

1. Overview of the `Ns` themselves
2. Programming with `Nat`
3. Arithmetic with `Nat` and properties we care about
4. How does `Nat` influence API design?
5. Implementation concerns
6. Beyond `Nat`
7. Conclude

Setting Yourself Up For Success

Okay, but we're concerned with the *practice* of *programming* ...

Setting Yourself Up For Success

```
type Nat where
  Z      : Nat
  Succ  : Nat -> Nat
```

Setting Yourself Up For Success

Now we can easily represent any \mathbb{N} we want!

Setting Yourself Up For Success

Now we can easily represent any \mathbb{N} we want!

$$\llbracket Z \rrbracket = 0$$

$$\llbracket \text{Succ } n \rrbracket = 1 + \llbracket n \rrbracket$$

Talk over?

Talk over?

This is all very nice and elegant, but the ergonomics *suck*

RSI risk

Even just typing this slide made my RSI flare up:

- [illegible]



Spoonful of sugar

What do we do for other types?

Spoonful of sugar

```
type List elem where
  []      : List elem
  (:::)   : elem -> List elem -> List elem
```

Spoonful of sugar

Lists are flexible and easy to reason about, but they have the same problem!

Spoonful of sugar

Lists are flexible and easy to reason about, but they have the same problem!

- `type String = List Char`

Spoonful of sugar

Lists are flexible and easy to reason about, but they have the same problem!

- `type String = List Char`
- `initials = 'P' :: ('W' :: ('L' :: []))`

Spoonful of sugar

Because of this ubiquity of lists, compiler writers quickly came up with syntactic sugar for them:

Spoonful of sugar

Because of this ubiquity of lists, compiler writers quickly came up with syntactic sugar for them:

- `"PWL" ⇒ 'P' :: ('W' :: ('L' :: []))`

Spoonful of sugar

Because of this ubiquity of lists, compiler writers quickly came up with syntactic sugar for them:

- `"PWL" ⇒ 'P' :: ('W' :: ('L' :: []))`
- `[1..3] ⇒ 1 :: (2 :: (3 :: []))`

Spoonful of sugar

Similarly, we can implement syntactic sugar for the natural numbers:

Spoonful of sugar

Similarly, we can implement syntactic sugar for the natural numbers:

$$\blacksquare 3 \Rightarrow \text{Succ } (\text{Succ } (\text{Succ } Z))$$

Spoonful of sugar

We lose nothing with the syntactic sugar, we can still pattern match on naturals and retain all of our inductive reasoning.

Natural usage

```
... if length xs <= 5  
    then ...  
    else ...
```

Pattern Matching still available...

```
(<=) : Nat -> Nat -> Bool
Z      _ = True
(Succ _) Z = False
(Succ x) (Succ y) = x <= y
```

Sign post

1. Overview of the `Ns` themselves
2. Programming with `Nat`
3. Arithmetic with `Nat` and properties we care about
4. How does `Nat` influence API design?
5. Implementation concerns
6. Beyond `Nat`
7. Conclude

Reading, Writing and ...

Reading, Writing and ...

1. Programmers expect some arithmetic ‘out of the box’ when dealing with numbers.

Reading, Writing and ...

1. Programmers expect some arithmetic ‘out of the box’ when dealing with numbers.
2. At the very least they expect $+$, $-$, \times , \div

Real data structures

When programming with the discrete structures which are common in programming, there is a correspondence between the operations on numbers and the operations on the data structures.

Real data structures

When programming with the discrete structures which are common in programming, there is a correspondence between the operations on numbers and the operations on the data structures.

1. Think ‘array indices’, or ‘size’

Real data structures

When programming with the discrete structures which are common in programming, there is a correspondence between the operations on numbers and the operations on the data structures.

1. Think ‘array indices’, or ‘size’
2. What would a *negative* size mean?

Who would even do that?

Who would even do that?

```
length :: Foldable t => t a → Int
```

Returns the size/length of a finite structure as an `Int`.
no general way to do better.

Figure: lol

Exceptional negatives

Think of how many APIs return an “`Int`”.

Exceptional negatives

Think of how many APIs return an “`Int`”.

1. How many of these APIs only use the negative numbers to signal errors?



What do we want?

If we think a bit about arithmetic we may conclude the following:

What do we want?

If we think a bit about arithmetic we may conclude the following:

1. Ideally, our operators would be *total*

What do we want?

If we think a bit about arithmetic we may conclude the following:

1. Ideally, our operators would be *total*
2. When possible, we want our operators to be *closed*

Why?

These properties, when combined, allow us to be confident that when we operate on two `Nats`, we get another `Nat`.

Why?

These properties, when combined, allow us to be confident that when we operate on two **Nats**, we get another **Nat**.

1. This isn't true for arithmetic over all number systems (nor should it be!)

Why?

These properties, when combined, allow us to be confident that when we operate on two **Nats**, we get another **Nat**.

1. This isn't true for arithmetic over all number systems (nor should it be!)
2. Many languages fail even where it should be!

Totality

Our functions being *total* gives us confidence that for any input, we get a result.

Closure

Our functions being *closed* means that the result values lie within the same number system as their arguments.

What do we want? (part 2)

“The aim is a total closed system of arithmetic with results that can be safely interpreted in the context of the discrete structures in general programming”

— *Colin Runciman, 1989*

Back to arithmetic

Addition and Multiplication present no difficulties.

Back to arithmetic

What about Subtraction?



Don't wait, saturate

Don't wait, saturate

```
(.-.) : Nat -> Nat -> Nat
n      .-.      Z = n
Z      .-.      _ = Z
(Succ n) .-. (Succ m) = n .-. m
```

Relate back to data structures

Relate back to data structures

```
drop : Nat -> List a -> List a
drop Z      xs      = xs
drop _      []      = []
drop (Succ n) (x::xs) = drop n xs
```

Relate back to data structures

We *want* a correspondence between operations on data structures and on numbers:

```
length (drop n xs) == length xs -. n
```


Relate back to data structures

These sorts of correspondences are what we use (often in our head) when programming or refactoring.

A divisive issue

Unlike Subtraction, division is *already* closed over Natural Numbers

A divisive issue

Unlike Subtraction, division is *already* closed over Natural Numbers (for the cases for which it is defined!)

Back to square zero

Some mathematicians define the Natural Numbers as starting from One! Would that save us from this issue?

Back to square zero

Maybe, but then we'd lose the important correspondence with real data structures.

Quick digression

Quick digression

Zero is not nothing!

Two solutions

Runciman proposes two solutions to the ‘division by zero’ problem:

Two solutions

Runciman proposes two solutions to the ‘division by zero’ problem:

1. based on viewing division on \mathbb{N} s as ‘slicing’

Two solutions

Runciman proposes two solutions to the ‘division by zero’ problem:

1. based on viewing division on \mathbb{N} s as ‘slicing’
2. based on using lazy `Nats`

Division as slicing

Think of dividing x by y as cutting x in y places.

Division as slicing

We can write a total division, $//$, in terms of a partial (fails when dividing by zero) division, $/$:

Division as slicing

We can write a total division, `//`, in terms of a partial (fails when dividing by zero) division, `/`:

$$x \text{ // } y = x \text{ / } (\text{Succ } y)$$



Umm...

We get one intuitive property

Umm...

We get one intuitive property

- Slicing zero times gets you the original thing back

... that's wrong

At the cost of it being *incorrect* at every other **Nat**

Let's fix it

We get back correctness by subtracting 1 from the divisor before passing it //

$$x ./\ y = x // (y .-. 1)$$



You coward!



You coward!

In a sense we've only side-stepped the problem!



You coward!

In a sense we've only side-stepped the problem!

- If you think this is the lazy solution...



Even lazier

Runciman proposes another solution to this problem:

Even lazier

Runciman proposes another solution to this problem:

- Lazy Natural Numbers

Lazy Nats

If we're in a lazy language we can have infinite structures!

Go infinity...

If we're in a lazy language we can have infinite structures!

```
infinity = Succ infinity
```


Back to division

`x ./ 0 = infinity`

`x ./ y = x / y`

No cheating

No cheating

```
x ./ y = if x < y  
         then 0  
         else Succ ((x .-. y) ./ y)
```

More power to you

Exponentiation is not closed over the Integers, but over Naturals it is!

More power to you

Exponentiation is not closed over the Integers, but over Naturals it is!

$$\begin{aligned}\text{pow } n \quad 0 &= 1 \\ \text{pow } n \text{ (Succ } p) &= n * \text{pow } n \text{ } p\end{aligned}$$

Laziness, revisited

Let's not start a war here

Laziness, revisited

Infinite values also allow you to avoid ‘cheating’ in some standard algorithms

Laziness, revisited

How many times have you seen `inf = 999999` in a graph algorithm?

Save yourself some computation

Are there more than 10 people in your company?

Save yourself some computation

Are there more than 10 people in your company?

```
... expensive > 10 ...
```

Laziness, revisited

Lazy numbers let us compare the sizes of things without necessarily fully computing the size!

Sign post

1. Overview of the `Ns` themselves
2. Programming with `Nat`
3. Arithmetic with `Nat` and properties we care about
4. How does `Nat` influence API design?
5. Implementation concerns
6. Beyond `Nat`
7. Conclude

APIs

We've already defined an API for arithmetic, with various tradeoffs.

APIs

We've already defined an API for arithmetic, with various tradeoffs.

1. Now let's define some non-arithmetic functions and see how the Nats guide us

Size

Size of structures is very straightforward

Size

```
size : List elem -> Nat
size [] = Z
size (x::xs) = Succ (size xs)
```


Position/Index

Finding the index of a thing is a little more interesting

Position/Index: Mark 1

```
position : elem -> List elem -> ??????????
```

Position/Index: Mark 1

```
position : elem -> List elem -> ??????????  
position a xs = pos xs 0  
  where  
    pos (x::xs) n =
```

Position/Index: Mark 1

```
position : elem -> List elem -> ??????????  
position a xs = pos xs 0  
  where  
    pos (x::xs) n =  
      if a == x  
      then n  
      else pos xs (Succ n)
```

Position/Index: Mark 1

```
position : elem -> List elem -> ??????????  
position a xs = pos xs 0  
  where  
    pos (x::xs) n =  
      if a == x  
      then n  
      else pos xs (Succ n)  
    pos [] n = ????
```

Position/Index: Mark 1

```
position : elem -> List elem -> Option Nat
position a xs = pos xs 0
  where
    pos (x::xs) n =
      if a == x
      then Some n
      else pos xs (Succ n)
    pos [] n = None
```

Thoughts: Mark 1

This is satisfying because we're explicit about the possibility of failure

Position/Index: Mark 2

It should really be `positions`!

Position/Index: Mark 2

It should really be **positions**!

```
positions : elem -> List elem -> List Nat
positions a xs = pos xs 0
  where
    pos (x::xs) n =
      if a == x
      then n :: pos xs (Succ n)
      else pos xs (Succ n)
    pos [] n = []
```

Thoughts: Mark 2

In a lazy language `positions` is strictly more flexible

Thoughts: Mark 1 & 2

Mind the gap

Thoughts: Mark 1 & 2

Mind the gap

- There were none!

sublist

Take the sublist of a list:

```
sublist m n = take (n - m+1) . drop m
```

sublist

The `sublist` function has invariants that the user has to keep in mind

```
sublist m n = take (n - m+1) . drop m
```

sublist

The `sublist` function has invariants that the user has to keep in mind

```
sublist m n = take (n - m+1) . drop m
```

- What if $n < (m-1)$?

sublist

The `sublist` function has invariants that the user has to keep in mind

```
sublist m n = take (n - m+1) . drop m
```

- What if $n < (m-1)$?
- `take` would be passed a negative argument!

sublist

Fix is straightforward

```
sublist : Nat -> Nat -> List elem -> List elem
sublist 0          n = take n
sublist (Succ m) n = take (n .-. m) . drop (Succ m)
```

Sign post

1. Overview of the `Ns` themselves
2. Programming with `Nat`
3. Arithmetic with `Nat` and properties we care about
4. How does `Nat` influence API design?
5. Implementation concerns
6. Beyond `Nat`
7. Conclude

Issues

Why don't we see Nats everywhere?

Issues

Why don't we see `Nats` everywhere?

- Language designers don't include them in the stdlibs

Issues

Why don't we see Nats everywhere?

- Language designers don't include them in the stdlibs
- Concerns about performance

Interesting observation

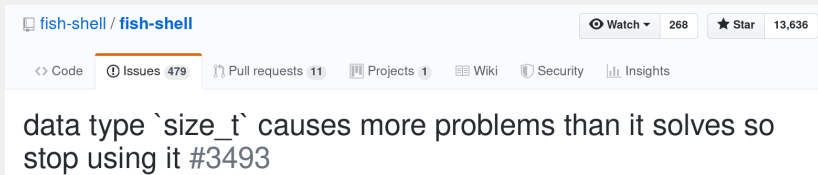
Even languages that *try* to have some sort of non-negative number end up tripping over themselves!

Interesting observation

Even languages that *try* to have some sort of non-negative number end up tripping over themselves!

- e.g. C with `size_t` and `ssize_t`

I'm not making this up



fish-shell / fish-shell

Watch 268 Star 13,636

Code Issues 479 Pull requests 11 Projects 1 Wiki Security Insights

data type `size_t` causes more problems than it solves so stop using it #3493



anordal commented on Oct 26, 2016

Contributor + 😊 ...

Just to present an alternative view, I think the *signed* ints are the dangerous ones, and that `int` should be forbidden (except APIs that specify just that). In my own code, I use `unsigned` as the generic type all over the place, and `signed` where I actually mean that. I find that very few numbers have to be signed, except results of subtraction.

Remember, "unsigned integer" is just programming jargon for "natural number".



20



5



6



The reality is that there are a large number of places where we currently combine unsigned and signed ints in arithmetic expressions. And absent explicit typecasts those expressions are likely to produce the wrong result. Which is what lead to the bug I introduced when I made a change to silence such warnings.

Reality check

This person is not wrong!

Reality check

This person is not wrong!

- understanding the behavior of casts (especially implicit ones) is hard!



Is all hope lost?

The issue is twofold:

Is all hope lost?

The issue is twofold:

- Unsigned values can be coerced away

Is all hope lost?

The issue is twofold:

- Unsigned values can be coerced away
- Programmers aren't forced to recon with 0!



All hope is not lost

Some languages do it right!



All hope is not lost

Some languages do it right!

- Idris and Agda compile Peano **Nats** to machine words

What about the *lazy* Nats?

There are issues with implementing the lazy Nats

What about the *lazy* Nats?

There are issues with implementing the lazy Nats

- Lazy languages can have poor memory usage if lazy structures are implemented naively

What about the *lazy* Nats?

What are the alternatives?

What about the *lazy* Nats?

What are the alternatives?

1. a machine number

What about the *lazy* Nats?

What are the alternatives?

1. a machine number
2. an unevaluated computation (i.e. a *thunk*)

What about the *lazy* Nats?

What are the alternatives?

1. a machine number
2. an unevaluated computation (i.e. a *thunk*)
3. a pair (m,n) of machine number and thunk

Why not machine?

Why not machine?

1. Suitable for eager languages (IMO)

Why not machine?

1. Suitable for eager languages (IMO)
2. We lose *infinity* in lazy languages

Why not thunks?

Why not thunks?

1. Uses $O(n)$ space

Why not thunks?

1. Uses $O(n)$ space
2. where n is the *value* of the `Nat`!

Perfect pair?

This leaves some combination of machine number and thunk

Perfect pair?

This leaves some combination of machine number and thunk

1. Static analyses can help make it more efficient

Perfect pair?

This leaves some combination of machine number and thunk

1. Static analyses can help make it more efficient
2. ‘dirty’ implementation techniques can be hidden from the user

Sign post

1. Overview of the `Ns` themselves
2. Programming with `Nat`
3. Arithmetic with `Nat` and properties we care about
4. How does `Nat` influence API design?
5. Implementation concerns
6. Beyond `Nat`
7. Conclude

What else

What else

1. Sets!



Consider the following:

Consider the following:

1. A function from an API you're using returns a `List`

Consider the following:

1. A function from an API you're using returns a **List**
2. Does order matter?

Consider the following:

1. A function from an API you're using returns a **List**
2. Does order matter?
3. What does a duplicate element signal?



Mind the gap!

Mind the gap!

1. What if the same function returned a **Set**?

Mind the gap!

1. What if the same function returned a **Set**?
2. No order in the representation

Mind the gap!

1. What if the same function returned a **Set**?
2. No order in the representation
3. No duplicate elements

Picking up the signals

Picking up the signals

1. Every data structure is signaling *something*

Picking up the signals

1. Every data structure is signaling *something*
2. Asking the consumers of your API to *ignore* a signal only serves to increase the cognitive burden of your API

Picking up the signals

1. Every data structure is signaling *something*
2. Asking the consumers of your API to *ignore* a signal only serves to increase the cognitive burden of your API
3. Try choosing structures that are *necessary and sufficient*

Picking up the signals

1. Every data structure is signaling *something*
2. Asking the consumers of your API to *ignore* a signal only serves to increase the cognitive burden of your API
3. Try choosing structures that are *necessary and sufficient*
4. This way, all signals are meant to be heeded



Closing thoughts

Closing thoughts

No one seems to disagree, and yet...



Ahead of his time

Ahead of his time

“The benefits of lazy evaluation generally are now widely recognised (though still regarded as controversial by some)”

— *Colin Runciman, The year of TS's birth*

Smash that subscribe button

Thanks for your time!

You can read more of my rants [@josecalderon](#)

N!