JFP **32**, e5, 19 pages, 2022.
doi:10.1017/S0956796822000028

FUNCTIONAL PEARL

Super-naturals

RALF HINZE
Fachbereich Informatik, Technische Universität Kaiserslautern, Germany
(*e-mail:* ralf-hinze@cs.uni-kl.de)

COLIN RUNCIMAN
Department of Computer Science, University of York, UK
(*e-mail:* colin.runciman@york.ac.uk)

1 Introduction

Functional programmers who enjoy working with lazy data structures sometimes wish numeric computations were lazy too. Usually they are not. Even in a "lazy" functional language such as Haskell, if numeric expressions are evaluated at all, they are evaluated completely. (At least, this is true for expressions of all predefined numeric types.) Such chunks of eagerness can play havoc with the fine-grained machinery of lazy evaluation. In particular, eager numeric operations are unsuitable for delicate measurements of lazy data structures. For example, an innocent-looking comparison such as $length\ xs > 0$ forces evaluation of the entire spine of the list *xs*—a needless expense.

One way to address this problem is to introduce an alternative numeric type whose values are indeed lazy data structures. Although some applications require negative or fractional values, the principal numeric requirement in general-purpose programming is support for the *natural* numbers (Runciman, 1989). Following Peano (1889), one specific option is to declare a datatype for successor chains with zero as a terminal value.

$$\textbf{data}\ Peano = O \mid S\ Peano$$

By default, *S* is non-strict: in accordance with an early mantra of laziness (Friedman & Wise, 1976), *successor does not evaluate its argument*. So if *length* computes a successor chain, $length\ xs > O$ evaluates almost immediately, as only the outermost constructor of $length\ xs$ is needed to decide whether $O > O = False$ or $S\ n > O = True$ applies. This is how lazy natural numbers are implemented in the *numbers* library for Haskell (Augustsson, 2007).

However, arithmetic succession is like a fixed small combinator—it is even called *S*! Performance is limited by such a fine-grained unit of work. We want a counterpart to the

$n = 1 :+ 3 :+ 5 :+ 1 :+ 4 :+ 2 :+ 1 :+ O$

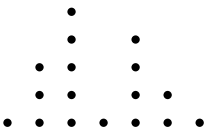


Fig. 1. A diagram of the super-natural n, for which $value\, n = 17$, $width\, n = 7$, and $height\, n = 5$. Unlike the Ferrers diagrams often used for partitions, we show the parts in representation order.

super-combinators of Hughes (1982), attuned to the coarser-grained units of work in the computation in hand. Meet super-naturals[1].

2 Super-naturals

Viewed another way, *Peano* is isomorphic to the type [()], and there are not many interesting operations on () values. A component type already equipped with arithmetic would offer more scope. So, generalising successor chains $1 + 1 + \cdots$, our lazy number structures are *partitions* $i + j + \cdots$ where the component values i, j, ... are positive whole numbers.

We call these structures *super-naturals*. They occupy a spectrum, at the extreme ends of which are counterparts of the eager monolithic *Natural*[2] and the lazy fine-grained *Peano*. But there are many intermediate possibilities, with partitions of different sizes.

```
infixr 6 :+
data Super natural = O | !natural :+ Super natural
```

A super-natural is either zero, represented by O, or of the form $p :+ ps$[3] where p is an evaluated part but ps may be unevaluated. The exclamation mark is a strictness annotation, forcing the first argument of $:+$ to be fully evaluated. The first, but not the second, so $:+$ is a form of delayed addition. The parametric *natural* type can, in principle, be any numeric type that implements mathematical naturals with operations for arithmetic and comparison.

Although the invariant that all parts of a super-natural are *positive* is not enforced by the type system, the smart constructor $.+$ takes care to avoid zero parts. An analogous smart product $.*$ will also be useful.

```
infixr 6 .+
infixr 7 .*

(.+), (.*) :: (Num natural, Eq natural) ⇒ natural → Super natural → Super natural
p .+ ps = if p == 0 then ps else p :+ ps
p .* ps = if p == 0 then O else if p == 1 then ps else fmap (p*) ps
```

The type *Super* is parametric, even functorial in the underlying number type, as witnessed by *fmap*:

[1] The name has also been used for Steinitz numbers, or for numbers of the form 10^{4n}, both otherwise unconnected with the super-naturals described here.
[2] Defined in *Numeric.Natural*, a basic Haskell library.
[3] In Haskell, symbolic data constructors, which are infix by default, must start with a colon.

https://doi.org/10.1017/S0956796822000028 Published online by Cambridge University Press

$$
\begin{aligned}
&\mathit{fmap} :: (a \to b) \to (\mathit{Super}\ a \to \mathit{Super}\ b)\\
&\mathit{fmap}\ f\ O \qquad\quad = O\\
&\mathit{fmap}\ f\ (p :\!\!+ ps) = f\ p :\!\!+ \mathit{fmap}\ f\ ps
\end{aligned}
$$

Indeed, super-naturals are like redecorated lists, in which $:\!\!+$ as "cons" evaluates the head but not the tail, and O replaces "nil". The functions *whole* and *parts* convert between the two views.

$$
\begin{aligned}
&\mathit{whole} :: (\mathit{Num}\ \mathit{natural}, \mathit{Eq}\ \mathit{natural}) \Rightarrow [\mathit{natural}] \to \mathit{Super}\ \mathit{natural}\\
&\mathit{whole} = \mathit{foldr}\ (:\!\!+)\ O\\
&\mathit{parts} :: \mathit{Super}\ \mathit{natural} \to [\mathit{natural}]\\
&\mathit{parts}\ O \qquad\quad = [\,]\\
&\mathit{parts}\ (p :\!\!+ ps) = p : \mathit{parts}\ ps
\end{aligned}
$$

The *value* function reduces a super-natural to its natural value. The *width* and *height* functions locate it in the representational spectrum. The dimensional terminology reflects a diagrammatic view of super-naturals: see Figure 1 for an example.

$$
\begin{aligned}
&\mathit{value}, \mathit{width}, \mathit{height} :: (\mathit{Num}\ \mathit{natural}, \mathit{Ord}\ \mathit{natural}) \Rightarrow \mathit{Super}\ \mathit{natural} \to \mathit{natural}\\
&\mathit{value}\ O \qquad\quad = 0\\
&\mathit{value}\ (p :\!\!+ ps) = p + \mathit{value}\ ps\\
&\mathit{width}\ O \qquad\quad = 0\\
&\mathit{width}\ (\_ :\!\!+ ps) \ = 1 + \mathit{width}\ ps\\
&\mathit{height}\ O \qquad\quad = 0\\
&\mathit{height}\ (p :\!\!+ ps) = \mathit{max}\ p\ (\mathit{height}\ ps)
\end{aligned}
$$

If $\mathit{width}\ n \leqslant 1$ then n is in its most compact form, and $\mathit{height}\ n = n$. If $\mathit{width}\ n = \mathit{value}\ n$ then n is like a *Peano* chain, and $\mathit{height}\ n \leqslant 1$.

3 Basic arithmetic

Let us first establish a few guiding principles we choose to adopt in our development of arithmetic and comparative operators for super-naturals.

As wide super-naturals may be derived from lazy data structures, operations on them should also be lazy, avoiding needless evaluation. However, we want to avoid the excessive widths of results that would arise if we merely replicated the unary chains of *Peano* arithmetic. Instead, we seek opportunities to express the arithmetic operations on super-naturals in terms of the corresponding natural operations applied to their parts. Striking a balance between the requirement for laziness, and the requirement to supply results of moderate width, we aim for arithmetic operations with results *no wider than their widest argument*. In particular, arithmetic on compact super-naturals (of width 0 or 1) should give compact results.

As we aspire to laziness, given arguments in constructed form (O or $p :\!\!+ ps$), we aim to produce a result in constructed form quickly. More specifically, we aim wherever possible to define super-natural operators with a *bounded-demand property*: only when the kth part of a result is demanded need any kth part of an argument be evaluated.

Another desirable characteristic of super-natural arithmetic is that operations satisfy the usual algebraic equivalences in a *strong sense*. Not only are the natural values of

https://doi.org/10.1017/S0956796822000028 Published online by Cambridge University Press

equated expressions the same (as they must be for correctness) but when evaluated they also have *identical representations* as super-naturals. Such strong equivalences give assurance to programmers choosing the most convenient arrangement of a compound arithmetic expression: they need not be unduly concerned about the pragmatic effect of their choice.

Addition. For the *Peano* representation, addition is concatenation of *S*-chains. For super-naturals, merely concatenating lazy partitions would miss an opportunity. We add super-naturals by *zipping them together*, combining their corresponding parts under natural addition, applying the middle-two interchange law $(i + m) + (j + n) = (i + j) + (m + n)$.[4]

$$\begin{aligned} O \quad & + qs \quad & = qs \\ ps \quad & + O \quad & = ps \\ (p :\!\!+ ps) & + (q :\!\!+ qs) & = (p + q) :\!\!+ (ps + qs) \end{aligned}$$

Viewing super-naturals as elements of a positional base 1 number system, our super-natural addition amounts to the standard school algorithm for adding multidigit numbers, but with two simplifications. (1) As the base is 1, positions do not matter, so there is no need to align digits. (2) There is no need for "carried" values as the digits can be arbitrarily large.

Instead of addition leading to wider and wider partitions, we have $width\ (m + n) = max\ (width\ m)\ (width\ n)$. We also have $height\ (m + n) \leqslant height\ m + height\ n$—not an equality as the highest parts of m and n may be in different positions.

This super-natural $+$ is strongly commutative and associative. Because of the *zip*-like recursion pattern, addition also has the bounded-demand property.

Natural subtraction. It is tempting to extend the zippy approach to natural subtraction, sometimes called *monus* and written $\dot{-}$. Tempting, but *wrong* as the counterpart of the middle-two interchange law $(i + m) \dot{-} (j + n) = (i \dot{-} j) + (m \dot{-} n)$ simply does not hold, e.g. $(2 + 3) \dot{-} (4 + 1) = 0 \neq 2 = (2 \dot{-} 4) + (3 \dot{-} 1)$. Often, properties linking addition and subtraction are conditional as subtraction is not a full inverse of addition. Specifically, $(a \dot{-} b) + b = a$ only holds if $a \geqslant b$. However, there are two useful value-preserving transformations on super-naturals.

$$[\![p_1 :\!\!+ (p_2 :\!\!+ ps)]\!] = [\![(p_1 + p_2) :\!\!+ ps]\!] \qquad \text{(join)}$$

$$[\![p :\!\!+ ps]\!] = [\![k :\!\!+ (p - k) :\!\!+ ps]\!] \qquad \text{if } 0 < k < p \qquad \text{(split)}$$

The Oxford brackets $[\![-]\!]$ are the semantic counterpart of *value*, mapping a super-natural to its denotation, an integer value—not a natural number, for reasons to become clear later.

By recursive splitting, we can reduce natural subtraction of super-naturals to natural subtraction of their constituent parts.[5]

[4] We *overload* the symbol '+', using it to denote both addition of super-naturals and addition of their natural parts. In Haskell, predefined *type classes* specify the expected signatures of numeric operators. To introduce a new type of number, further overloading these signatures, one declares *class instances* defining the operators for this type without any further signature declarations. See the Appendix for details of the class instances for super-naturals.

[5] We are also *overloading* the symbol '$\dot{-}$', introducing a suitable type class called *Monus*, see Appendix.

https://doi.org/10.1017/S0956796822000028 Published online by Cambridge University Press

$$
\begin{aligned}
O \quad & \dot{-}\, qs \quad && = O \\
ps \quad & \dot{-}\, O \quad && = ps \\
(p :\!\!+ ps) \dot{-} (q :\!\!+ qs) & && = \textbf{case}\ compare\ p\ q\ \textbf{of} \\
& && \quad LT \to ps \dot{-} ((q \dot{-} p) :\!\!+ qs) \\
& && \quad EQ \to ps \dot{-} qs \\
& && \quad GT \to ((p \dot{-} q) :\!\!+ ps) \dot{-} qs
\end{aligned}
$$

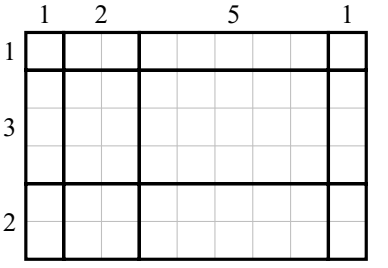
Despite the possible splitting of arguments in recursive calls, the result of a super-natural subtraction has dimensions bounded by those of the first argument: $width\ (m \dot{-} n) \leqslant width\ m$ and $height\ (m \dot{-} n) \leqslant height\ m$.

However, natural subtraction is necessarily more eager than addition. It must *fully evaluate at least one argument* before it can determine even the outermost construction of a result.

Of course subtraction is neither commutative nor associative. However, "subtractions commute" as we have the strong equivalence $(k \dot{-} m) \dot{-} n = (k \dot{-} n) \dot{-} m$. These subtraction chains are also strongly equivalent to $k \dot{-} (m + n)$.

Multiplication. Turning to super-natural multiplication, we need to distribute one sum over the other as illustrated in the following diagram.

$$(1 :\!\!+ 3 :\!\!+ 2 :\!\!+ O) * (1 :\!\!+ 2 :\!\!+ 5 :\!\!+ 1 :\!\!+ O)$$



One option is to enumerate the rectangles of the matrix in one way or another:

$$
\begin{aligned}
ps * qs &= whole\ [p * q \mid p \leftarrow parts\ ps, q \leftarrow parts\ qs] \\
ps * qs &= whole\ [p * q \mid q \leftarrow parts\ qs, p \leftarrow parts\ ps]
\end{aligned}
$$

Here, $width\ (m * n) = width\ m * width\ n$. Such expansion is needlessly extravagant, missing opportunities for combination of natural values. To decrease the width of the resulting product, we could sum rows or columns of the matrix.

$$
\begin{aligned}
ps * qs &= whole\ (map\ sum\ [[p * q \mid q \leftarrow parts\ qs] \mid p \leftarrow parts\ ps]) \\
ps * qs &= whole\ (map\ sum\ [[p * q \mid p \leftarrow parts\ ps] \mid q \leftarrow parts\ qs])
\end{aligned}
$$

Expansion is avoided, as $width\ (m * n) = width\ m$ or $= width\ n$. But now either $(m*)$ or $(*n)$ is hyper-strict—assuming natural $*$ is bi-strict, so natural *sum* is hyper-strict. To avoid the bias toward an argument we could sum the diagonals instead (a suitable definition of *diagonals* is left as an exercise for the reader).

$$ps * qs = whole\ (map\ sum\ (diagonals\ [[p * q \mid q \leftarrow parts\ qs] \mid p \leftarrow parts\ ps]))$$

Expansion is moderate as for non-O arguments: $width\ (m * n) = width\ m + width\ n - 1$. However, even this expansion is questionable, and diagonalisation is quite costly.

https://doi.org/10.1017/S0956796822000028 Published online by Cambridge University Press

All of these approaches are unsatisfactory in one way or another. Our preferred method of super-natural multiplication exploits the distribution of multiplication over addition in its simplest form: $(i + m) * (j + n) = i * j + m * j + i * n + m * n$.

$$\begin{aligned} O \quad & * qs \quad = O \\ ps \quad & * O \quad = O \\ (p :+ ps) * (q :+ qs) &= (p * q) .+ ((q .* ps) + (p .* qs) + (ps * qs)) \end{aligned}$$

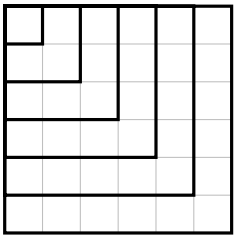
Observe that three variants of multiplication are used: $*$ is overloaded to denote both multiplication of naturals and of super-naturals, $.*$ multiplies a super-natural by a natural.

Now $width\,(m * n) \leqslant max\,(width\,m)\,(width\,n)$, a property acquired by the use of super-natural addition. As there is no multiplicative increase in *width*, there must be one in *height*. Indeed, the tightest bound on *height* in terms of argument dimensions is

$$height\,(m * n) \leqslant 2 * min\,(width\,m)\,(width\,n) * height\,m * height\,n.$$

The following example is instructive. The height bound is $2 * min\ 6\ 6 * 1 * 1 = 12$.

$$\begin{aligned} &\gg whole\,[1, 1, 1, 1, 1, 1] \\ &1 :+ 1 :+ 1 :+ 1 :+ 1 :+ 1 :+ O \\ &\gg it * it \\ &1 :+ 3 :+ 5 :+ 7 :+ 9 :+ 11 :+ O \end{aligned}$$



Do you recall the visual proof that the first n odd numbers sum to n squared? Well, this is exactly what is going on here. Super-natural multiplication sums up the L-shaped areas.

Multiplication defined in this way is strongly associative and commutative, and it is strongly distributive over addition. Multiplication also has the bounded-demand property.

Natural division. We defer a discussion of natural division and modulus/remainder as it will be useful to talk about comparison first.

4 Ordering

Evaluation of arithmetic expressions is typically triggered by comparisons[6], and the representation of super-naturals enables us to make these comparisons lazy. In particular, to decide whether a super-natural is zero, we only need to know the outermost constructor: any super-natural of the form $p :+ ps$ must be greater than zero, as all parts of a super-natural are positive. If we had *negative* parts, then comparison would be hyper-strict in both arguments, defeating the whole point of the exercise.

[6] The numeric interfaces in Haskell feature a multitude of constructors and modifiers, but only a few observers, the methods of the classes *Eq* and *Ord*.

https://doi.org/10.1017/S0956796822000028 Published online by Cambridge University Press

Comparison. The function *compare* employs the same recursion scheme as natural subtraction, using both comparison and subtraction over the underlying natural components.

$$
\begin{aligned}
&\mathit{compare}\ O\ \ O\ &&= \mathit{EQ}\\
&\mathit{compare}\ O\ \ (\_ :\!+ \_)\ &&= \mathit{LT}\\
&\mathit{compare}\ (\_ :\!+ \_)\ \ O\ &&= \mathit{GT}\\
&\mathit{compare}\ (p :\!+ ps)\ (q :\!+ qs) &&= \mathbf{case}\ \mathit{compare}\ p\ q\ \mathbf{of}\\
&&&\qquad \mathit{LT} \to \mathit{compare}\ ps\ ((q \dot{-} p) :\!+ qs)\\
&&&\qquad \mathit{EQ} \to \mathit{compare}\ ps\ qs\\
&&&\qquad \mathit{GT} \to \mathit{compare}\ ((p \dot{-} q) :\!+ ps)\ qs
\end{aligned}
$$

Comparison fully evaluates the spine of one of its arguments, but rarely of both. So it exhibits similar strictness behaviour to natural subtraction, which is hardly surprising since $a \leqslant b$ if and only if $a \dot{-} b = 0$.

Minimum and maximum. With *compare* defined, Haskell defaults to generic definitions of all the boolean-valued comparison operators. These simply inspect *compare* results in the obvious way. There are also generic defaults for *min* and *max*

$$
\begin{aligned}
\mathit{min}\ m\ n &= \mathbf{if}\ m \leqslant n\ \mathbf{then}\ m\ \mathbf{else}\ n\\
\mathit{max}\ m\ n &= \mathbf{if}\ m \geqslant n\ \mathbf{then}\ m\ \mathbf{else}\ n
\end{aligned}
$$

but we must *not* settle for these defaults. For the super-naturals they are excessively strict, giving no result until both m and n are evaluated to the extent required for *compare* to give its atomic result. Instead, we can give *max* and *min* their own explicit lazier definitions, re-using the recursion scheme of *compare*.

$$
\begin{aligned}
&\mathit{min}\ O\ \ qs\ &&= O\\
&\mathit{min}\ ps\ \ O\ &&= O\\
&\mathit{min}\ (p :\!+ ps)\ (q :\!+ qs) &&= \mathbf{case}\ \mathit{compare}\ p\ q\ \mathbf{of}\\
&&&\qquad \mathit{LT} \to p :\!+ \mathit{min}\ ps\ ((q \dot{-} p) :\!+ qs)\\
&&&\qquad \mathit{EQ} \to p :\!+ \mathit{min}\ ps\ qs\\
&&&\qquad \mathit{GT} \to q :\!+ \mathit{min}\ ((p \dot{-} q) :\!+ ps)\ qs\\
&\mathit{max}\ O\ \ qs\ &&= qs\\
&\mathit{max}\ ps\ \ O\ &&= ps\\
&\mathit{max}\ (p :\!+ ps)\ (q :\!+ qs) &&= \mathbf{case}\ \mathit{compare}\ p\ q\ \mathbf{of}\\
&&&\qquad \mathit{LT} \to p :\!+ \mathit{max}\ ps\ ((q \dot{-} p) :\!+ qs)\\
&&&\qquad \mathit{EQ} \to p :\!+ \mathit{max}\ ps\ qs\\
&&&\qquad \mathit{GT} \to q :\!+ \mathit{max}\ ((p \dot{-} q) :\!+ ps)\ qs
\end{aligned}
$$

These definitions can be seen as *productive variants* of comparison. Their correctness hinges on the fact that addition distributes over both minimum $\downarrow$ and maximum $\uparrow$, e.g.:

$$
\begin{aligned}
&(i + m) \downarrow (j + n)\\
=\ &\{\ \text{assume } i < j \text{ and property of monus}\ \}\\
&(i + m) \downarrow (i + (j \dot{-} i) + n)\\
=\ &\{\ \text{addition distributes over minimum: } x + (y \downarrow z) = (x + y) \downarrow (x + z)\ \}\\
&i + (m \downarrow ((j \dot{-} i) + n)),
\end{aligned}
$$

https://doi.org/10.1017/S0956796822000028 Published online by Cambridge University Press

We saw that addition reduces the width of both arguments in lockstep — a *zip*-like pattern. By contrast, most recursive calls in the *compare* family of functions only reduce the width of one argument—a *merge*-like pattern. We see the impact in the worst-case widths of results, which for *max* are the *sums* of argument widths, and for *min* just one less:

$$\begin{aligned}&\gg max\,(1 :\!\!+ 2 :\!\!+ 2 :\!\!+ 2 :\!\!+ 2 :\!\!+ O)\,(2 :\!\!+ 2 :\!\!+ 2 :\!\!+ 2 :\!\!+ 2 :\!\!+ O)\\&1 :\!\!+ 1 :\!\!+ 1 :\!\!+ 1 :\!\!+ 1 :\!\!+ 1 :\!\!+ 1 :\!\!+ 1 :\!\!+ 1 :\!\!+ 1 :\!\!+ O\\&\gg min\,(1 :\!\!+ 2 :\!\!+ 2 :\!\!+ 2 :\!\!+ 2 :\!\!+ O)\,(2 :\!\!+ 2 :\!\!+ 2 :\!\!+ 2 :\!\!+ 2 :\!\!+ O)\\&1 :\!\!+ 1 :\!\!+ 1 :\!\!+ 1 :\!\!+ 1 :\!\!+ 1 :\!\!+ 1 :\!\!+ 1 :\!\!+ 1 :\!\!+ O\end{aligned}$$

The first of these examples also illustrates a curious and disappointing feature of *max*: it factors out the *minimum* of the first parts in each step, not the maximum as one might reasonably expect.

Minimum revisited. Can we rework the definition of *min* so that it uses a *zip*-like recursion pattern? Yes, we can! Consider the first recursive call in the definition of *min*. Instead of combining $q \dot{-} p$ and *ps* to form the super-natural $(q \dot{-} p) :\!\!+ ps$ we simply keep the summands apart. To this end, alongside each of the original *min* arguments we introduce an auxiliary "accumulator" argument.

$$[\![min\text{-}acc\ m\ ps\ n\ qs]\!] = (m + [\![ps]\!]) \downarrow (n + [\![qs]\!]),$$

Correctness relies on an applicative invariant that at least one accumulator is zero. Initially, both are zero.[7]

$$\begin{aligned}&newmin :: (Monus\ natural) \Rightarrow Super\ natural \to Super\ natural \to Super\ natural\\&newmin\ ps\ qs = min\text{-}acc\ 0\ ps\ 0\ qs\\&min\text{-}acc :: (Monus\ natural) \Rightarrow\\&\qquad natural \to Super\ natural \to natural \to Super\ natural \to Super\ natural\\&min\text{-}acc\ m\ O\ \ \ \ n\ O\ \ \ \ = O\\&min\text{-}acc\ m\ O\ \ \ \ n\ (q :\!\!+ qs) = dotMin\ m\ ((n+q) :\!\!+ qs)\\&min\text{-}acc\ m\ (p :\!\!+ ps)\ n\ O\ \ \ \ = dotMin\ n\ ((m+p) :\!\!+ ps)\\&min\text{-}acc\ m\ (p :\!\!+ ps)\ n\ (q :\!\!+ qs) = min\ p'\ q' :\!\!+ min\text{-}acc\ (p' \dot{-} q')\ ps\ (q' \dot{-} p')\ qs\\&\quad \textbf{where}\ p' = p + m\\&\qquad\ \ q' = q + n\end{aligned}$$

Just as we defined $:\!\!+$ and $:\!\!*$ earlier, to combine natural and super-natural arguments, here we need an asymmetric version of minimum, called *dotMin*.

$$\begin{aligned}&dotMin :: (Monus\ natural) \Rightarrow natural \to Super\ natural \to Super\ natural\\&dotMin\ 0\ \ qs\ \ \ \ \ = O\\&dotMin\ m\ O\ \ \ \ \ = O\\&dotMin\ m\ (q :\!\!+ qs) = \textbf{if}\ m \leqslant q\ \textbf{then}\ m :\!\!+ O\ \textbf{else}\ q :\!\!+ dotMin\ (m \dot{-} q)\ qs\end{aligned}$$

Maximum revisited. Theoretically, maximum is dual to minimum. However, when we define them over super-naturals, there are fundamental obstacles in the way of a symmetric

[7] As a reminder, the type class *Monus* introduces natural subtraction $\dot{-}$.

https://doi.org/10.1017/S0956796822000028 Published online by Cambridge University Press

formulation. There is no convenient greatest element dual to the least element O[8]. For a non-zero super-natural, the first part gives a lower bound for the entire value, but there is no dual representation of an upper bound[9].

So implementation of *max* differs in some interesting ways from *min*. Recall that *min* factors out the *smaller* of the first parts, e.g. reducing $min\,(9 :+ m)\,(2 :+ n)$ to $2 :+ min\,((9 \dot{-} 2) :+ m)\,n$. This suggests that *max* should factor out the *larger* part, e.g. reducing the call $max\,(9 :+ m)\,(2 :+ n)$ to $9 :+ max\,m\,((2 - 9) :+ n)$. But now the alarm bells are ringing, as $2 - 9$ is *negative*, beyond the realm of *natural* numbers!

One way out of this difficulty is to interpret the accumulators used for *max* "negatively". Whereas for *min-acc* the argument-and-accumulator pairs represented *natural sums*, now, for *max-acc*, let these pairs represent *integer differences*:

$$[\![max\text{-}acc\ ps\ m\ qs\ n]\!] = ([\![ps]\!] - m) \uparrow ([\![qs]\!] - n).$$

Viewing super-naturals as base 1 numerals, in *max-acc* the m and n arguments are *borrowed* values, whereas in *min-acc* they are *carried* values.

Turning these ideas into equational definitions:

$$\begin{aligned}
&newmax :: (Monus\ natural) \Rightarrow Super\ natural \to Super\ natural \to Super\ natural\\
&newmax\ ps\ qs = max\text{-}acc\ ps\ 0\ qs\ 0\\
&max\text{-}acc :: (Monus\ natural) \Rightarrow\\
&\qquad Super\ natural \to natural \to Super\ natural \to natural \to Super\ natural\\
&max\text{-}acc\ O\ \ \ m\ qs\ \ \ n = qs \mathbin{.\dot{-}} n\\
&max\text{-}acc\ ps\ \ \ m\ O\ \ \ n = ps \mathbin{.\dot{-}} m\\
&max\text{-}acc\ (p :+ ps)\ m\ (q :+ qs)\ n = max\ (p \dot{-} m)\ (q \dot{-} n) :+ max\text{-}acc\ ps\ (b \dot{-} a)\ qs\ (a \dot{-} b)\\
&\quad \textbf{where}\ a = p + n\\
&\qquad\ \ b = q + m
\end{aligned}$$

Again, correctness relies on the invariant that $m = 0$ or $n = 0$. This invariant implies $max\,(0 - m)\,(i - n) = max\,0\,(i - n)$ justifying the first base case, where *max-acc* specialises to monus! Once more we need an operator, this time $.\dot{-}$, to combine natural and super-natural values:

$$\begin{aligned}
&(\mathbin{.\dot{-}}) :: (Monus\ natural) \Rightarrow Super\ natural \to natural \to Super\ natural\\
&ps\ \ \ \ \mathbin{.\dot{-}}\ 0\ = ps\\
&O\ \ \ \ \mathbin{.\dot{-}}\ m = O\\
&(p :+ ps) \mathbin{.\dot{-}} m = \textbf{if}\ p > m\ \textbf{then}\ (p \dot{-} m) :+ ps\ \textbf{else}\ ps \mathbin{.\dot{-}} (m \dot{-} p)
\end{aligned}$$

Look again at the last equation of *max-acc*. As its argument–accumulator pairs represent *integer* differences, we factor out the maximum of $p - m$ and $q - n$ to form the first part. However, this maximum difference is sure to be positive—recall the applicative invariant for *max-acc*, and the data invariant that super-natural parts are positive. So replacing any negative difference by zero does not alter *max*'s choice: it returns the other (positive) difference. Writing $max\,(p \dot{-} m)\,(q \dot{-} n)$ for $(p - m) \uparrow (q - n)$ is not only valid, it avoids undefined results when standard subtraction is partially defined for *natural*.

[8] Though there are many infinite super-naturals, there is no argument pattern by which they can be recognised.
[9] We thank an anonymous reviewer for this observation about bounds.

https://doi.org/10.1017/S0956796822000028 Published online by Cambridge University Press

As the definition of *max-acc* is probably the most complicated in the paper, we provide a derivation in Appendix 2.

Properties revisited. So have we curtailed the sum-of-widths behaviour of *max* and *min*? Yes, as desired, we now have $width\,(newmax\,m\,n) \leqslant max\,(width\,m)\,(width\,n)$ and $width\,(newmin\,m\,n) \leqslant max\,(width\,m)\,(width\,n)$. With *min* on the left, one might expect *min* on the right, but that inequality does not hold: e.g. $newmin\,(1 :+ 1 :+ O)\,(2 :+ O) = 1 :+ 1 :+ O$. We also have corresponding bounds on *height* as $height\,(newmax\,m\,n) \leqslant max\,(height\,m)\,(height\,n)$ and $height\,(newmin\,m\,n) \leqslant max\,(height\,m)\,(height\,n)$. Here again, for *newmin* a minimum depth variant is plausible but wrong: e.g. $newmin\,(1 :+ 3 :+ O)\,(2 :+ 2 :+ O) = 1 :+ 3 :+ O$.

As for equivalences, *newmin* is strongly commutative, associative and idempotent. However, *newmax* only scores two out of three, as it is *not* strongly associative. Here is a counter-example:

$$\begin{aligned}
&\gg newmax\,(newmax\,(2 :+ O)\,(1 :+ 1 :+ 1 :+ O))\,(1 :+ 1 :+ 2 :+ O)\\
&2 :+ 1 :+ 1 :+ O\\
&\gg newmax\,(2 :+ O)\,(newmax\,(1 :+ 1 :+ 1 :+ O)\,(1 :+ 1 :+ 2 :+ O))\\
&2 :+ 2 :+ O
\end{aligned}$$

As for laziness, *newmin* has the bounded-demand property, but *newmax* does *not* share this property. Here is a counter-example:

$$\begin{aligned}
&\gg newmax\,(3 :+ O)\,(1 :+ 1 :+ \bot)\\
&3 :+ \bot
\end{aligned}$$

To determine the second part (if any) of the result, *newmax* needs to evaluate beyond the second part of the second argument. Does the first part of the result already hold the entire maximum value, or will the second argument eventually exceed it?

We deliberately made the first part of a non-zero *newmax* result the natural maximum of the first parts of its arguments. This choice is consistent with laziness: it derives as much information as possible from immediately available argument parts, reducing the likelihood that any further result part (and therefore further argument parts) will be needed. Putting the previous example in a larger context:

$$\begin{aligned}
&\gg newmax\,(3 :+ O)\,(1 :+ 1 :+ \bot) > (2 :+ O)\\
&True
\end{aligned}$$

The result would have been $\bot$ using our original super-natural *max*. However, in other contexts, where more parts of a *newmax* result are *needed* and an initially maximal argument is exhausted, the other argument may have to be evaluated beyond the limits for a bounded-demand operation. We cannot have our cake *and* eat it!

5 Super-natural division

Super-natural division with remainder is a more tricky problem. Resorting to repeated subtraction would be no more attractive than mere repeated addition was for multiplication. Instead we seek a way to apply natural division to component parts.

https://doi.org/10.1017/S0956796822000028 Published online by Cambridge University Press

As a warm-up, let us first complete our family of point-wise operations: *dotDiv* is a special case of natural division where the divisor is a natural, rather than a super-natural.

$$
\begin{aligned}
&\mathit{dotDiv} :: (\mathit{Integral}\ \mathit{natural}) \Rightarrow \mathit{Super}\ \mathit{natural} \to \mathit{natural} \to \mathit{Super}\ \mathit{natural}\\
&\mathit{dotDiv}\ O \qquad\qquad\ \ d = O\\
&\mathit{dotDiv}\ (p :\!\!+ O) \qquad d = (p\ \mathbf{div}\ d) :\!\!+ O\\
&\mathit{dotDiv}\ (p_1 :\!\!+ p_2 :\!\!+ ps)\ d = q :\!\!+ \mathit{dotDiv}\ ((r + p_2) :\!\!+ ps)\ d\\
&\quad \mathbf{where}\ (q, r) = \mathit{divMod}\ p_1\ d
\end{aligned}
$$

The recursive case is derived by join and split transformations: the super-natural dividend $p_1 :\!\!+ p_2 :\!\!+ ps$, which equals $((p_1\ \mathbf{div}\ d) * d + p_1\ \mathbf{mod}\ d) :\!\!+ p_2 :\!\!+ ps$, is rewritten to the super-natural $(p_1\ \mathbf{div}\ d) * d :\!\!+ (p_1\ \mathbf{mod}\ d + p_2) :\!\!+ ps$.

Concerning division of super-naturals, $a\ \mathbf{div}\ b$ cannot be lazier than $a < b$ because we can implement $<$ in terms of **div**: $a < b$ if and only if $a\ \mathbf{div}\ b = 0$. To compute $a\ \mathbf{mod}\ b$, complete evaluation of a is required; indeed, there is no sensible, lazy *dotMod* operation.

$$
\begin{aligned}
&\mathit{divMod}\ \_\ O = \mathit{error}\ \texttt{"division by zero"}\\
&\mathit{divMod}\ a\ b\ = \mathbf{case}\ \mathit{compare}\ a\ b\ \mathbf{of}\\
&\qquad\qquad\qquad\quad \mathit{LT} \to (0, a' :\!\!+ O)\\
&\qquad\qquad\qquad\quad \mathit{EQ} \to (1, 0)\\
&\qquad\qquad\qquad\quad \mathit{GT} \to (\mathit{dotDiv}\ a\ b', (a'\ \mathbf{mod}\ b') :\!\!+ O)\\
&\qquad\qquad\qquad\quad \mathbf{where}\ a' = \mathit{value}\ a\\
&\qquad\qquad\qquad\qquad\quad\ \ b' = \mathit{value}\ b
\end{aligned}
$$

This definition squeezes out the last bit of laziness: if $a < b$ then $a\ \mathbf{mod}\ b = a$; we can return a in compact form as it is fully evaluated by the comparison, but b need not be fully evaluated. However, if $a > b$ then $a\ \mathbf{mod}\ b$ is inevitably eager.

Like subtraction, division gives results with dimensions bounded by those of the first argument, despite possible additions to its parts in recursive calls. That is, assuming non-O n, we have both $\mathit{width}\ (m\ \mathbf{div}\ n) \leqslant \mathit{width}\ m$ and $\mathit{height}\ (m\ \mathbf{div}\ n) \leqslant \mathit{height}\ m$. All remainders are compact: $\mathit{width}\ (m\ \mathbf{mod}\ n) \leqslant 1$ and $\mathit{height}\ (m\ \mathbf{mod}\ n) < \mathit{value}\ n$.

Also like subtraction, division is neither commutative nor associative, yet "divisions commute" as the strong equivalence $(k\ \mathbf{div}\ m)\ \mathbf{div}\ n = (k\ \mathbf{div}\ n)\ \mathbf{div}\ m$ holds. These division chains are also strongly equivalent to $k\ \mathbf{div}\ (m * n)$. Any equivalence between **mod** results is guaranteed to be a strong equivalence, as all such results are compact.

6 Application: measures

Returning to our introductory motivation, in this section we explore the use of super-natural arithmetic to measure lazy data structures.

Measuring binary search trees. There are at least three different measures of a binary search tree: the number of keys stored in it (*size*), the length of the shortest path from the root to a leaf (*min-height*), and the length of the longest one (*max-height*).

https://doi.org/10.1017/S0956796822000028 Published online by Cambridge University Press

$$
\begin{aligned}
&\textbf{data}\ \mathit{Tree}\ \mathit{elem} = \mathit{Leaf} \mid \mathit{Node}\ (\mathit{Tree}\ \mathit{elem})\ \mathit{elem}\ (\mathit{Tree}\ \mathit{elem})\\
&\mathit{size}, \mathit{min\text{-}height}, \mathit{max\text{-}height} :: (\mathit{Monus}\ \mathit{natural}) \Rightarrow \mathit{Tree}\ \mathit{elem} \to \mathit{Super}\ \mathit{natural}\\
&\mathit{size}\ \mathit{Leaf} = 0\\
&\mathit{size}\ (\mathit{Node}\ l\ x\ r) = 1 \dot{+} (\mathit{size}\ l + \mathit{size}\ r)\\
&\mathit{min\text{-}height}\ \mathit{Leaf} = 0\\
&\mathit{min\text{-}height}\ (\mathit{Node}\ l\ x\ r) = 1 \dot{+} \mathit{min}\ (\mathit{min\text{-}height}\ l)\ (\mathit{min\text{-}height}\ r)\\
&\mathit{max\text{-}height}\ \mathit{Leaf} = 0\\
&\mathit{max\text{-}height}\ (\mathit{Node}\ l\ x\ r) = 1 \dot{+} \mathit{max}\ (\mathit{max\text{-}height}\ l)\ (\mathit{max\text{-}height}\ r)
\end{aligned}
$$

These definitions may look quite standard, but there is an important twist: in some places they use delayed addition $\dot{+}$ rather than overloaded addition $+$. Why? Because if these functions used *only* the methods of the predefined numeric classes, they would exhibit the very strict behaviour we deprecated in our introduction.

The *size* function uses both delayed and overloaded addition, which has an interesting effect. To showcase its behaviour, we first define an *insert* function as a way to grow trees.

$$
\begin{aligned}
&\mathit{insert} :: (\mathit{Ord}\ a) \Rightarrow a \to \mathit{Tree}\ a \to \mathit{Tree}\ a\\
&\mathit{insert}\ x\ \mathit{Leaf} = \mathit{Node}\ \mathit{Leaf}\ x\ \mathit{Leaf}\\
&\mathit{insert}\ x\ (\mathit{Node}\ l\ y\ r) = \textbf{if}\ x \leqslant y\ \textbf{then}\ \mathit{Node}\ (\mathit{insert}\ x\ l)\ y\ r\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \mathit{Node}\ l\ y\ (\mathit{insert}\ x\ r)
\end{aligned}
$$

Now we generate a list of a million random elements, fold this list by insertion into a large search tree, and measure it.

$$
\begin{aligned}
&\gg \textbf{let}\ \mathit{rs} = \mathit{take}\ 1000000\ (\mathit{randomRs}\ (0, 999999)\ (\mathit{mkStdGen}\ 19650702))\\
&\gg \textbf{let}\ t = \mathit{foldr}\ \mathit{insert}\ \mathit{Leaf}\ \mathit{rs}\\
&\gg \mathit{size}\ t\\
&1 \dot{+} 2 \dot{+} 4 \dot{+} 8 \dot{+} 16 \dot{+} 32 \dot{+} 64 \dot{+} 125 \dot{+} 247 \dot{+} 480 \dot{+} 910 \dot{+} 1662 \dot{+} 2924 \dot{+} 5007 \dot{+}\\
&8120 \dot{+} 12603 \dot{+} 18514 \dot{+} 26210 \dot{+} 35123 \dot{+} 44581 \dot{+} 54252 \dot{+} 63068 \dot{+} 70193 \dot{+}\\
&74763 \dot{+} 76634 \dot{+} 75907 \dot{+} 72847 \dot{+} 67298 \dot{+} 60435 \dot{+} 52327 \dot{+} 44229 \dot{+} 36074 \dot{+}\\
&28318 \dot{+} 21560 \dot{+} 15683 \dot{+} 10892 \dot{+} 7388 \dot{+} 4774 \dot{+} 2957 \dot{+} 1769 \dot{+} 982 \dot{+} 499 \dot{+}\\
&258 \dot{+} 144 \dot{+} 72 \dot{+} 31 \dot{+} 11 \dot{+} 2 \dot{+} O
\end{aligned}
$$

By the combined application of $\dot{+}$ and $+$, *size* processes the tree *breadth-first*, and the k-th part of its result is the number of elements on level k! The first 7 parts are exact powers of two, which tells us that the first 7 levels of the tree are full. But then the leaves start, so the tree's *minimal* height is 7. Finally, the width of this super-natural *size* gives us the *maximal* height of the tree—it is 48.

The two functions measuring path lengths behave quite differently. Thanks to lazy evaluation, *min-height* is a lot faster than *max-height*—around 3× faster, for this example.

$$
\begin{aligned}
&\gg \mathit{min\text{-}height}\ t\\
&1 \dot{+} 1 \dot{+} 1 \dot{+} 1 \dot{+} 1 \dot{+} 1 \dot{+} 1 \dot{+} O\\
&\gg \mathit{max\text{-}height}\ t\\
&1 \dot{+} 1 \dot{+} 1 \dot{+} 1 \dot{+} 1 \dot{+} 1 \dot{+} 1 \dot{+} 1 \dot{+} 1 \dot{+} 1 \dot{+} 1 \dot{+} 1 \dot{+} 1 \dot{+} 1 \dot{+} 1 \dot{+} 1 \dot{+} 1 \dot{+} 1 \dot{+} \cdots\\
&\gg \mathit{width}\ \mathit{it}\\
&48
\end{aligned}
$$

https://doi.org/10.1017/S0956796822000028 Published online by Cambridge University Press

These measure functions also work smoothly with infinite trees. To illustrate, *grow-except* grows an infinite tree with only a *single leaf* at some given position.

```
grow-except :: (Num elem, Eq elem) ⇒ elem → Tree elem
grow-except n = grow 0
  where grow k = if k == n then Leaf
                           else Node (grow ((2 ∗ k) + 1)) k (grow (2 ∗ (k + 1)))
```

Any result of *grow-except* has infinite size and infinite maximal height, but because of the leaf it has finite minimal height.

```
≫ min-height (grow-except 4711)
1 :+ 1 :+ 1 :+ 1 :+ 1 :+ 1 :+ 1 :+ 1 :+ 1 :+ 1 :+ 1 :+ O
≫ max-height (grow-except 4711) > it
True
≫ size (grow-except 10000000) > 3
True
```

As expected, computations requiring only a finite prefix of infinite *max-height* or *size* results do not diverge. The last call, for example, returns almost instantly.

However, these example evaluations also demonstrate that the height functions return *only successor chains*. Such chains may be tolerable for height measures of "bushy" structures such as trees. But what about structures with a unit branching factor, such as the ubiquitous list?

Measuring lists. For lists, the three measure functions *size*, *min-height* and *max-height* coincide: all are equivalent to *length*. So first, let us similarly define a variant of Haskell's *length* function that gives a super-natural result, just replacing the single + in the usual definition by :+.

```
length :: (Monus natural) ⇒ [elem] → Super natural
length [ ]      = 0
length (x : xs) = 1 :+ length xs
```

The length of a list is now given by an *equally long* successor chain. Even though super-natural arithmetic is quite well-behaved—we have been careful to ensure that the results of arithmetic operations are no wider than their widest argument—by pursuing the goal of laziness, we seem to have boxed ourselves into a corner with Peano!

How can we compute narrower and higher super-natural lengths, without abandoning laziness and reverting to compact atomic values? Recall our width-limiting approach to arithmetic. Given two super-naturals to be added, for example, we *zipped* them together, typically obtaining a higher result. Though we only have a single super-natural length, we can zip it with *itself* by lazily splitting its parts into two super-natural "halves" and adding them together. Instead of fiddling with the *length* definition, we prefer a modular approach. We introduce a general-purpose combinator for super-natural compression.

https://doi.org/10.1017/S0956796822000028 Published online by Cambridge University Press

$$
\begin{aligned}
&\mathit{compress} :: (\mathit{Monus}\ \mathit{natural}) \Rightarrow \mathit{Super}\ \mathit{natural} \to \mathit{Super}\ \mathit{natural}\\
&\mathit{compress}\ \mathit{ps} = \mathit{odds}\ \mathit{ps} + \mathit{evens}\ \mathit{ps}\\
&\mathit{odds}, \mathit{evens} :: \mathit{Super}\ \mathit{natural} \to \mathit{Super}\ \mathit{natural}\\
&\mathit{odds}\ \ O\ \ = O\\
&\mathit{odds}\ \ (p :\!\!+ \mathit{ps}) = p :\!\!+ \mathit{evens}\ \mathit{ps}\\
&\mathit{evens}\ O\ \ = O\\
&\mathit{evens}\ (\_ :\!\!+ \mathit{ps}) = \mathit{odds}\ \mathit{ps}
\end{aligned}
$$

Now there is a simple and modular way to obtain more compact super-natural lengths: apply $\mathit{compress} \cdot \mathit{length}$. If even compressed super-natural lengths are still too wide, here is one of many ways to achieve greater compaction: the progressively tighter *squeeze* function halves width again for each successive part.

$$
\begin{aligned}
&\mathit{squeeze} :: (\mathit{Monus}\ \mathit{natural}) \Rightarrow \mathit{Super}\ \mathit{natural} \to \mathit{Super}\ \mathit{natural}\\
&\mathit{squeeze}\ O\ \ = O\\
&\mathit{squeeze}\ (p :\!\!+ \mathit{ps}) = p :\!\!+ \mathit{squeeze}\ (\mathit{compress}\ \mathit{ps})
\end{aligned}
$$

For example, squeezing an infinite successor chain yields the powers of two.

$$
\begin{aligned}
&\gg\!\!= \mathit{squeeze}\ (\mathit{whole}\ [1\,..])\\
&1 :\!\!+ 2 :\!\!+ 4 :\!\!+ 8 :\!\!+ 16 :\!\!+ 32 :\!\!+ 64 :\!\!+ 128 :\!\!+ 256 :\!\!+ 512 :\!\!+ 1024 :\!\!+ \ldots\\
&\gg\!\!= \mathit{squeeze}\ (\mathit{whole}\ [1\,..]) > 1000000\\
&\mathit{True}
\end{aligned}
$$

There is not much difference in running time between $\mathit{squeeze}\ (\mathit{whole}\ [1\,..]) > 1000000$ and $\mathit{whole}\ [1\,..] > 1000000$. But the squeezed version consumes less memory.

7 Summary and concluding remarks

There is quite a gulf between atomic numeric values and Peano chains. We set out to span this gulf by developing numeric operations for a flexible intermediate representation—the super-natural number. Allowing many alternative representations for the same value gives programmers flexibility. They can choose, in any specific application, what degree of strictness or laziness they prefer. In practice, their overall choice is most simply expressed as a particular combination of individual choices between delayed and zipped super-natural addition. Even the two extreme options of atomic values (always zip, stay compact) and Peano chains (always delay, stay unary) are still available.

Non-zero super-naturals are simply partitions, freely ordered in any convenient sequence, and typically evaluated in that order. We never seriously considered any stronger data invariant. We did briefly consider a weaker one, allowing zero parts, but rejected it as it seemed to cause more problems than it solved.

In our development of the other numeric operations, we aimed to be even handed. They had to be lazy, but not so lazy as to expand super-natural widths beyond those of the widest operands. We tried hard to define operations to satisfy all the equivalences a programmer might reasonably expect. We checked all the properties we claim by testing the first million cases, and some by constructing a general proof. Addition, multiplication and minimum are particularly well-behaved; subtraction, division and remainder are of necessity stricter

https://doi.org/10.1017/S0956796822000028 Published online by Cambridge University Press

and asymmetric, yet still satisfy some pleasing laws; maximum is the worst-behaved—if any reader thinks they know how to reform it, please tell us!

Thinking of the relationship between numbers and lazy data structures, various data structures have been modelled after number systems. We have already observed in more than one context that the structure of lists matches the *Peano* number type. The textbook by Okasaki (1998) contains a wealth of further examples. In due time, perhaps someone will devise a general-purpose data structure modelled after super-naturals.

Meanwhile, super-naturals themselves are already data structures. They are containers of part values of their parametric natural type. They even meet their own requirements for such parts. *Super-super-naturals* anyone?

Acknowledgements

We thank Jeremy Gibbons, John Hughes and three anonymous reviewers for their helpful comments on a previous version of this article.

Conflicts of interest

None.

References

Augustsson, L. (2007) *Numbers: various number types*. Accessed April 28, 2021. Available at: http://www.haskell.org/package/numbers

Friedman, D. P. & Wise, D. S. (1976) CONS should not evaluate its arguments. In *Proceedings of 3rd International Colloqium on Automata Languages and Programming,* pp. 257–284.

Hughes, R. J. M. (1982) Super-combinators: A new implementation method for applicative languages. In *Proceedings of ACM Symposium on Lisp and Functional Programming,* pp. 1–10.

Okasaki, C. (1998) *Purely Functional Data Structures*. Cambridge University Press.

Peano, G. (1889) The principles of arithmetic, presented by a new method. In *A Source Book in Mathematical Logic, 1879–1931*, van Keijenoort, J. (ed). Harvard University Press, pp. 83–97.

Runciman, C. (1989) What about the natural numbers? *Comput. Lang.* **14**(3), 181–191.

1 Appendix: Class instances

Haskell predefines an elaborate hierarchy of numeric type classes: *Num*, *Real*, *Integral*, *Fractional*, *Floating*, *RealFrac*, and *RealFloat*. Unfortunately, even the base class *Num* includes negation, absolute value, and sign. So natural number types do not enjoy natural instance definitions. For example, the predefined type of natural numbers (in *Numeric.Natural*) has a partial subtraction operation: applications such as $11 - 47$ raise an arithmetic underflow exception.

https://doi.org/10.1017/S0956796822000028 Published online by Cambridge University Press

```
instance Functor Super where
  fmap f  O       = O
  fmap f  (p :+ ps) = f p :+ fmap f ps
instance (Show natural) ⇒ Show (Super natural) where
  show O = "0"
  show n = "(" ++ show' n ++ ")"
    where show' O       = "0"
          show' (p :+ ps) = show p ++ " :+ " ++ show' ps
```

Fig. 2. Super-naturals: non-numeric instances.

To avoid assumptions about subtraction in the underlying natural arithmetic, for the super-naturals, we introduce a custom type class for natural subtraction.

```
infixl 6 ∸
class (Num natural, Ord natural) ⇒ Monus natural where
  (∸) :: natural → natural → natural
  a ∸ b = if a ⩽ b then 0 else a − b
```

The default definition of monus uses explicit case distinction, rather than $max\ 0\ (a - b)$, as *max* is typically bi-strict.

```
instance Monus Int
instance Monus Integer
instance Monus Natural
instance (Monus natural) ⇒ Monus (Super natural) where
  O ∸ qs              = O
  ps ∸ O              = ps
  (p :+ ps) ∸ (q :+ qs) = case compare p q of
                            LT → ps ∸ ((q ∸ p) :+ qs)
                            EQ → ps ∸ qs
                            GT → ((p ∸ q) :+ ps) ∸ qs
```

Figures 2, 3, 4, and 5 give the instance declarations for non-numeric type classes (*Functor* and *Show*), basic arithmetic (*Num*), equality and comparison (*Eq* and *Ord*), and enumeration and integral operations (*Enum*, *Real*, and *Integral*). Each numeric type-class instance requires that the underlying *natural* parts type is an instance of *Monus*.

2 Appendix: Derivation of *max-acc*

The Oxford brackets map a partition to its denotation, an integer value.

$$[\![O]\!] = 0 \tag{2.1a}$$

$$[\![p :+ ps]\!] = p + [\![ps]\!] \tag{2.1b}$$

Recall that the argument-and-accumulator pairs of *max-acc* represent *integer differences*:

$$[\![max\text{-}acc\ ps\ m\ qs\ n]\!] = ([\![ps]\!] - m) \uparrow ([\![qs]\!] - n). \tag{2.2}$$

https://doi.org/10.1017/S0956796822000028 Published online by Cambridge University Press

```
instance (Monus natural) ⇒ Num (Super natural) where
  O + qs                 = qs
  ps + O                 = ps
  (p :+ ps) + (q :+ qs) = (p + q) :+ (ps + qs)
  (−) = (∸)
  O ∗ qs                 = O
  ps ∗ O                 = O
  (p :+ ps) ∗ (q :+ qs)  = (p ∗ q) :+ ((q .∗ ps) + (p .∗ qs) + (ps ∗ qs))
  negate n = 0
  abs n = n
  signum O         = O
  signum (_ :+ _) = 1
  fromInteger n = fromInteger n :+ O
```

Fig. 3. Super-naturals: basic arithmetic.

```
instance (Monus natural) ⇒ Eq (Super natural) where
  m == n = compare m n == EQ
instance (Monus natural) ⇒ Ord (Super natural) where
  compare O        O        = EQ
  compare O        (_ :+ _) = LT
  compare (_ :+ _) O        = GT
  compare (p :+ ps) (q :+ qs) = case compare p q of
                                  LT → compare ps ((q ∸ p) :+ qs)
                                  EQ → compare ps qs
                                  GT → compare ((p ∸ q) :+ ps) qs
  min ps qs = min-acc 0 ps 0 qs
  max ps qs = max-acc ps 0 qs 0
```

Fig. 4. Super-naturals: equality and comparison.

```
instance (Enum natural, Monus natural) ⇒ Enum (Super natural) where
  fromEnum = fromEnum · value
  toEnum i  = whole [toEnum i | i > 0]
instance (Real natural, Monus natural) ⇒ Real (Super natural) where
  toRational = toRational · value
instance (Integral natural, Monus natural) ⇒ Integral (Super natural) where
  divMod a b = case compare a b of
                 LT → (0, a′ :+ O)
                 EQ → (1, 0)
                 GT → (dotDiv a b′, (a′ mod b′) :+ O)
                 where a′ = value a
                       b′ = value b
  quotRem =  divMod
  toInteger = toInteger · value
```

Fig. 5. Super-naturals: enumeration and integral operations.

https://doi.org/10.1017/S0956796822000028 Published online by Cambridge University Press

The derivation of *max-acc* relies furthermore on the applicative invariant that at least one accumulator is zero: $m = 0$ or $n = 0$.

Base case: we reason

$$
\begin{aligned}
& [\![\mathit{max\text{-}acc}\ O\ m\ qs\ n]\!] \\
= \quad & \{\ \text{specification of } \mathit{max\text{-}acc}\ (2.2)\ \} \\
& ([\![O]\!] - m) \uparrow ([\![qs]\!] - n) \\
= \quad & \{\ \text{definition of } [\![-]\!]\ (2.1a)\ \} \\
& (0 - m) \uparrow ([\![qs]\!] - n) \\
= \quad & \{\ \text{applicative invariant: } m = 0 \text{ or } n = 0, \text{ and data invariant: } m \geqslant 0 \text{ and } [\![qs]\!] \geqslant 0\ \} \\
& 0 \uparrow ([\![qs]\!] - n) \\
= \quad & \{\ \text{definition of monus: } x \mathbin{\dot{-}} y = 0 \uparrow (x - y)\ \} \\
& [\![qs]\!] \mathbin{\dot{-}} n.
\end{aligned}
$$

The proof for the other base case is similar.

Inductive case: we factor out the larger of the differences, $(p - m) \uparrow (q - n)$, to form the first part, and then work towards a situation where we can apply the specification (2.2) from right to left. We reason

$$
\begin{aligned}
& [\![\mathit{max\text{-}acc}\ (p :\!\!+ ps)\ m\ (q :\!\!+ qs)\ n]\!] \\
= \quad & \{\ \text{specification of } \mathit{max\text{-}acc}\ (2.2)\ \} \\
& ([\![p :\!\!+ ps]\!] - m) \uparrow ([\![q :\!\!+ qs]\!] - n) \\
= \quad & \{\ \text{definition of } [\![-]\!]\ (2.1b)\ \} \\
& (p + [\![ps]\!] - m) \uparrow (q + [\![qs]\!] - n) \\
= \quad & \{\ \text{define } p' := p - m \text{ and } q' := q - n\ \} \\
& (p' + [\![ps]\!]) \uparrow (q' + [\![qs]\!]) \\
= \quad & \{\ \text{addition distributes over maximum: } x + (y \uparrow z) = (x + y) \uparrow (x + z)\ \} \\
& (p' \uparrow q') + ((p' - (p' \uparrow q') + [\![ps]\!]) \uparrow (q' - (p' \uparrow q') + [\![qs]\!])) \\
= \quad & \{\ \text{arithmetic: } x - y + z = z - (y - x)\ \} \\
& (p' \uparrow q') + (([\![ps]\!] - ((p' \uparrow q') - p')) \uparrow ([\![qs]\!] - ((p' \uparrow q') - q'))) \\
= \quad & \{\ \text{subtraction distributes over maximum: } (x \uparrow y) - z = (x - z) \uparrow (y - z)\ \} \\
& (p' \uparrow q') + (([\![ps]\!] - (0 \uparrow (q' - p'))) \uparrow ([\![qs]\!] - ((p' - q') \uparrow 0))) \\
= \quad & \{\ \text{define } a := p + n \text{ and } b := q + m \text{ and note that } p' - q' = a - b\ \} \\
& (p' \uparrow q') + (([\![ps]\!] - (0 \uparrow (b - a))) \uparrow ([\![qs]\!] - ((a - b) \uparrow 0))) \\
= \quad & \{\ \text{definition of monus: } x \mathbin{\dot{-}} y = 0 \uparrow (x - y)\ \} \\
& (p' \uparrow q') + (([\![ps]\!] - (b \mathbin{\dot{-}} a)) \uparrow ([\![qs]\!] - (a \mathbin{\dot{-}} b))) \\
= \quad & \{\ \text{specification of } \mathit{max\text{-}acc}\ (2.2)\ \} \\
& (p' \uparrow q') + [\![\mathit{max\text{-}acc}\ ps\ (b \mathbin{\dot{-}} a)\ qs\ (a \mathbin{\dot{-}} b)]\!] \\
= \quad & \{\ \text{applicative invariant: } m = 0 \text{ or } n = 0, \text{ and data invariant: } p > 0 \text{ and } q > 0\ \} \\
& ((p \mathbin{\dot{-}} m) \uparrow (q \mathbin{\dot{-}} n)) + [\![\mathit{max\text{-}acc}\ ps\ (b \mathbin{\dot{-}} a)\ qs\ (a \mathbin{\dot{-}} b)]\!]
\end{aligned}
$$

https://doi.org/10.1017/S0956796822000028 Published online by Cambridge University Press

$$= \quad \{ \text{ definition of } [\![-]\!] \text{ (2.1b) and } x \uparrow y = max\ x\ y \text{ for non-negative operands } \}$$
$$[\![max\ (p \dot{-} m)\ (q \dot{-} n) :\!+ max\text{-}acc\ ps\ (b \dot{-} a)\ qs\ (a \dot{-} b)]\!].$$

The final rewrites aim to avoid the use of integer subtraction in the body of *max-acc*. Finally, observe that the applicative invariant is maintained in the recursive call, as at least one of $a \dot{-} b$ and $b \dot{-} a$ is zero.

https://doi.org/10.1017/S0956796822000028 Published online by Cambridge University Press