Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

Practically surreal: Surreal arithmetic in Julia

Matthew Roughan

ARC Centre of Excellence for Mathematical & Statistical Frontiers (ACEMS), School of Mathematical Sciences, University of Adelaide, Adelaide, 5005, SA, Australia

A R T I C L E I N F O

Article history:
Received 26 September 2018
Received in revised form 17 January 2019
Accepted 6 March 2019

A B S T R A C T

This paper presents an implementation of arithmetic on Conway's surreal numbers. It also provides tools for visualising complicated surreals in the form of graph visualisations, and illustrates their use through several examples, and a small contribution to the theory of surreals.

Code metadata

Current code version	*v0.1.1*
Permanent link to code/repository used of this code version	https://github.com/ElsevierSoftwareX/SOFTX_2018_184
Legal Code License	*MIT "Expat" License*
Code versioning system used	git
Software code languages, tools, and services used	Julia, GraphViz
Compilation requirements, operating environments & dependencies	Julia v0.6, v0.7, v1.0 and GraphViz v2.38 (for visualisation)
If available Link to developer documentation/manual	https://github.com/mroughan/SurrealNumbers.jl
Support email for questions	matthew.roughan@adelaide.edu.au

1. Introduction

Surreal numbers were invented by John Conway [1], and named by Knuth in "Surreal Numbers: How Two Ex-Students Turned on to Pure Mathematics and Found Total Happiness" [2]. They are appealing because they provide a single construction for all of the numbers we are familiar with – the reals, rationals, hyperreals and ordinals – with only the use of set theory. Thus they provide an underpinning idea of what a number really is. And they have an elegant and complex structure that is worthy of study in its own right.

Surreal numbers are defined recursively: a surreal number x consists of an ordered pair of two sets of surreal numbers (call them the left and right sets, X_L and X_R, respectively) such that no member of the left set is $\geq$ any of the members of the right set. We write $x = \{X_L \mid X_R\}$ for such a surreal.

The starting point for the recursion is $\{\emptyset \mid \emptyset\}$ (where $\emptyset$ is the empty set E). A careful reading of the definition sees that no elements of one can be $\geq$ the other because there *are* no elements, so the comparison is automatically true. We call this initial number $\bar{0}$. Then on the "first day" a first generation of surreals can be created, building on $\bar{0}$. On the second day we create a second generation and so on. Each has a consistent meaning corresponding to traditional numbers, with respect to standard mathematical operators such as addition.

There are many works on the surreals, *e.g.*, [1–6]. Conway and many subsequent researchers seem mainly interested in them as a component of combinatorial game theory [1]. A few studies aim at deeper analysis problems. However, these works do not generally investigate the numbers themselves in detail. Despite the complexity of some components of the theory, they show no examples beyond a small basic set.

But examples are a key means to teach complicated ideas. We need them here, but do not have them. Why not? The lack arises from the complexity of the calculations, but also from the difficulty in visualising the results. The package presented here attacks those problems through an implementation of the surreals, and a visualisation from graph theory.

E-mail address: matthew.roughan@adelaide.edu.au.

https://doi.org/10.1016/j.softx.2019.03.005

We implement[1] an important subset of the surreal numbers in the new programming language Julia [8]. Julia should be ideal for this task, because it allows dynamic typing, scripting, and other features that allow it to be used easily and productively, but simultaneously allows the rigorous type definitions needed for the development of surreals, and it aims to be extremely efficient. The saying is "Julia walks like Python, but runs like C", expressing the speed of the language, coupled with an ease of use for mathematical programming. Julia is also free and open source. With the recent release of Julia 1.0, it seems interesting to understand whether it lives up to its hype.

The experience of using Julia on this project was almost entirely positive. Surreal arithmetic involves deep recursion, and complicated use of memory, and although we do not have a good benchmark to compare against, the performance of the code (given what it is doing) is good. Furthermore, writing the code was a much more enjoyable experience than it might have been in some of Julia's major competitors. For instance, the language has some elegant features that automatically extend the utility of functions, for example from scalars to vectors.

With Julia we can explore the surreal numbers in more detail. Apart from the educational value of this facility, it allows the construction of new hypotheses about the surreals, highlighting the benevolent relationship between computer science and mathematics. We explore one small example of this here.

Finally, the deep recursion of all operations on surreals makes them computationally challenging to work with. We illustrate this with a multiplication table showing how quickly the computational cost of operations grows. That raises the question of how we might improve algorithms in the future.

2. The surreal numbers

There are several good tutorials or books on the surreal numbers, *e.g.*,[1–6]. Tøndering [3] and Simons [6], in particular, provide excellent free introductions to surreals.

Importantly however, in much of the literature the idea of a surreal number is interwoven with its "form". This is best explained by an analogy to rational numbers. We can think of any given rational number in terms of its *form* p/q, but $2p/2q = p/q$, *i.e.*,there are two forms with the same *value*. In fact there are an infinite number of rational number forms with any given value. We usually refer to these as the same "number". Here, we implement surreal forms because it is in terms of these that Conway's surreal arithmetic operators are defined. Keddie [9] calls two forms identical if they are the same (*i.e.*,have the same left and right sets), and equal if they have the same value. We shall distinguish these two cases by writing equality of value as equivalence, $\equiv$, and identity by $==$. A single equal sign will denote assignment.

We denote numbers in lower case, and sets in upper case, with the convention that X_L and X_R are the left and right sets of x, and write a form as $x = \{X_L \mid X_R\}$. It is common to omit empty sets, but we prefer writing $\emptyset$ explicitly because it is a little clearer when writing complicated sequences.

The operations from traditional arithmetic: comparisons, addition, multiplication and so on are all defined for surreal forms [1,3,6]. This toolkit implements all of the common operations except arbitrary divisions (see below for the reason).

The left and right sets of a surreal form can be infinite, and this leads to many interesting facets of the surreal numbers (the ordinals, for instance), but here we will only consider surreal numbers with finite representations $\mathbb{S}$, the so called *short* numbers [10, Def 2.24]. These surreals are an important and useful subset because they correspond to the *dyadic* numbers $\mathbb{D}$, which are rational numbers of the form

$$x = \frac{n}{2^k},$$

where n and k are integers. To be more explicit, every short surreal has a dyadic real number with the same value [6, p. 27], and every dyadic real number can be mapped to a short surreal.

The standard mapping is called the Dali function [3], and is defined recursively by $d : \mathbb{D} \to \mathbb{S}$ where

$$d(x) \overset{\text{def}}{=} \begin{cases} \{\emptyset \mid \emptyset\}, & \text{if } x = 0, \\ \{d(n-1) \mid \emptyset\}, & \text{if } x = n, \text{ a positive integer}, \\ \{\emptyset \mid d(n+1)\}, & \text{if } x = n, \text{ a negative integer}, \\ \left\{ d\left(\frac{n-1}{2^k}\right) \middle| d\left(\frac{n+1}{2^k}\right) \right\}, & \text{if } x = \frac{n}{2^k} \text{ for } k > 0, n \text{ odd}. \end{cases}$$

The restriction of this toolkit to dyadics may seem significant, but (a) the dyadics are dense on the reals, and (b) floating point numbers are represented in computers as dyadics. Thus short surreals are a practical subset. However, the dyadics do not form a *field*, that is multiplicative inverses do not exist in all cases, and so arbitrary division is not possible.

As each number is really an equivalence class of forms, we will often need to be more specific about exactly which form we are referring to. The Dali construction is unique, and often important, so we refer to these surreal forms as *canonical* and denote them by putting a line above the number, *e.g.*,$d(1) \overset{\text{def}}{=} \bar{1} \overset{\text{def}}{=} \{\bar{0} \mid \emptyset\} == \{\{\emptyset \mid \emptyset\} \mid \emptyset\}$. Returning to the analogy to rational numbers, the canonical form is similar to a rational number expressed in *lowest terms*, *i.e.*,the form p/q where p and q are integers without any common factors. Other examples of canonical forms include

$$\begin{aligned} d(-1) &\overset{\text{def}}{=} \overline{-1} &== \{\emptyset \mid \bar{0}\}, \\ d(2) &\overset{\text{def}}{=} \bar{2} &== \{\bar{1} \mid \emptyset\}, \\ d(1/2) &\overset{\text{def}}{=} \overline{1/2} &== \{\bar{0} \mid \bar{1}\}, \end{aligned}$$

noting that each is defined in terms of earlier definitions.

The Dali construction erects a scaffolding of the simplest forms (for a given number), but even so it quickly leads to complicated expressions, and thus we need a means to illustrate surreal forms. We do so with pictures such as Fig. 1. The figure shows each surreal as a node in a graph. In particular, it is a connected Directed Acyclic Graph (DAG), though a DAG by itself would lose information. The graph would only specify parents, not left and

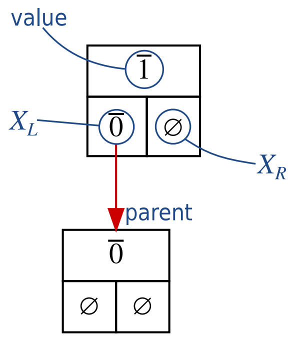


Fig. 1. A DAG depicting $\bar{1}$. Each box represents a surreal number with the value given in the top section, and the left and right sets shown in the bottom left and right sections, with links to each parent form shown by red and blue arrows, respectively.

[1] The toolkit (v0.1.1) is released under the MIT license, and available at https://github.com/mroughan/SurrealNumbers.jl. It works for Julia 0.6, 0.7 and 1.0, and has been tested under various versions of Linux and MacOS, and has no dependencies other than the Graphviz toolbox [7], which is used solely to produce the illustrations of graphs.

right parents. So in displaying the DAG, we show a box for each surreal number, with the value given in the top section and the left and right sets shown in the bottom left and right sections, respectively. Left and right parents are shown by red and blue arrows.

Additional examples are shown in Fig. 2. The first shows a form equivalent to zero, *i.e.*, $\{\overline{-1} \mid \overline{1}\} \equiv \{\emptyset \mid \emptyset\}$. To understand why these are equivalent, one needs to understand the definition of $\leq$ for surreal numbers. Conway [1] defined $x \leq y$ to mean that there is no member $x_L \in X_L$ such that $y \leq x_L$ and no member $y_R \in Y_R$ such that $y_R \leq x$. In the example y would be $\{\emptyset \mid \emptyset\}$, which has only empty sets and so there are no members y_R to deal with, and thus we only need note that $-1 \leq 0$ to see that $\{-1 \mid 1\} \leq 0$. We then flip the relationship around to see that also $0 \leq \{-1 \mid 1\}$, and combining the two inequalities we get $\{-1 \mid 1\} \equiv 0$.

Note that in the above calculations we do not need to restrict ourselves to canonical forms (those denoted by an overline) as the statements are true for any forms of 1 and 0, but the distinction between the canonical forms and surreal numbers (which consist of a class of forms, of which the canonical form is but one) is an important one to note in what follows.

The graph in Fig. 2(a) shows that a value can reappear at multiple places in the structure of the form: in this case surreals with the value 0 appear both at the top and the bottom of the DAG. The information characterising the surreal form is not its value, or even the values of its subsets, but the whole structure of the DAG that describes it. So the two '0' nodes in the graph are different (non-identical) surreals that just happen to have the same value.

The next example, shown in Fig. 2(b), gives the DAG for the Dali construction of 3/2:

$$\overline{3/2} == \left\{\overline{1} \mid \overline{2}\right\} == \left\{\left\{\{\emptyset \mid \emptyset\} \mid \emptyset\right\} \mid \left\{\{\{\emptyset \mid \emptyset\} \mid \emptyset\} \mid \emptyset\right\}\right\}.$$

Fig. 2(c) and (d) show two other constructions of 3/2 resulting from simple arithmetic. We can see how quickly arithmetic increases the complexity of the resulting forms. A core aim here is to be able to play with such arithmetic to gain a better understanding of surreal numbers.

We can see from the examples that a graph is a better way to understand surreal forms than a complicated series of brackets and empty sets. We can, for instance, see that the 1s appearing in Fig. 2(a), (b) and (c) are identical forms, but there are two different forms for 1 in (d): one is the canonical form $\overline{1}$ (appearing near the bottom of the figure) and the other appears in the third row from the top.

3. Implementing the surreals in Julia

The goal of this work is to explore surreal number forms in detail. There are several prior implementations of the surreals in various programming languages [11–17]. Most of these are very incomplete. For instance, most deal only with integers, *e.g.*, [14,17], and/or have quite limited sets of facilities [16]. Perhaps the most complete is in Coq by Mamane [13], but this is aimed at proving properties of surreals rather than exploring the numbers themselves. Furthermore, none of these packages try to visualise complicated surreal outputs.

The choice of Julia as programming language was driven by the aim to have a tool in which to explore the surreals, and so an interpreted language was preferred. However, the recursion of even the simplest operations means that computational efficiency is an issue. Julia claims to provide both C-like computational performance, and a dynamic, REPL (read–evaluate–print–loop) interpreter suitable for exploration [8,18]. Surreal numbers seemed an interesting test of the new language.

```
**Input**: $x = \{X_L \mid X_R\}$
**Output**: $m = \lfloor x \rfloor$
i = 0
**while** $x \geq 2^{(i+1)}$ **do**
  | i = i + 1 // *exponential search to for bounding interval*
**end**
a = $2^i$; b = $2^{i+1}$ // *binary search on interval* $[2^i, 2^{i+1})$
**for** $j = 1$ *to* $i$ **do**
  midpoint = (a+b)/2
  **if** $x <$ midpoint **then**
    | b = midpoint
  **else**
    | a = midpoint
  **end**
**end**
m = a
```

Algorithm 1: Calculate $m = \lfloor x \rfloor$, *i.e.*,the largest integer less that x, for $x \geq 1$. The case $0 \leq x < 1$ is handled by an initial check, and negative values are handled by a mirror image of this algorithm.

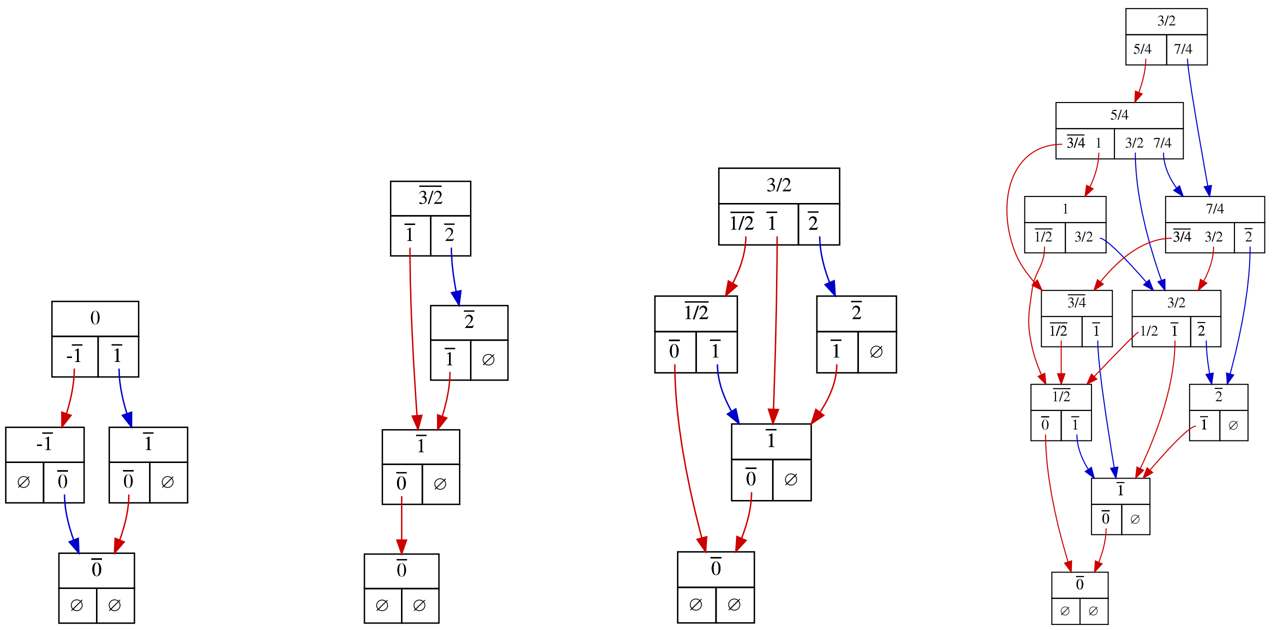
In the literature on surreal numbers there was one major missing piece: an algorithm to find the value of an arbitrary surreal form. The value is not a simple function of the left and right sets, but rather it is the value of the "simplest" number x that satisfies the criteria $X_L < x < X_R$ [3,4]. The problem is that this definition is not constructive. The obvious recursive approach was tested, but found to be very slow. The final approach adopted was a exponential/binary search to find the dyadic $x = n/2^k$ that satisfies this relationship, and which has the smallest floor $\lfloor x \rfloor$, and then the smallest (non-negative) k.

The floor algorithm (Algorithm 1) first finds a bounding pair of integer powers of 2 through an exponential search, and then refines this to the single integer interval through a binary search over this range. Now, given $m = \lfloor x \rfloor$ we know that $x \in [m, m+1)$, and that we are looking for the dyadic number with the smallest value of k. We again use a binary search similar to that above, but with respect to the paired bounds. The proof that this algorithm works requires derivation of the generation/birthday of canonical surreals. We omit this here in order to preserve space to discuss more immediate issues.

A key point in working with surreals is brought up by the algorithm. It uses comparisons, and these also are defined recursively. This is faster than alternatives we trialled, but still imposes a computational load even for superficially simple cases. Thus, in many operations we cache results, storing them in a dictionary. The dictionary look-up itself could become onerous, however, so we perform this using a hash function, which is calculated once the first time it is needed for a given surreal form, and then stored in the surreal form's data structure to facilitate fast look-ups.

The resulting toolkit is open source, and available at https://github.com/mroughan/SurrealNumbers.jl. It is installed using the standard Julia package manager, and has many of the functions and operators expected for a numerical type. A simple example of code is included below. The SurrealShort function is a constructor taking three arguments: a string giving the shorthand for the surreal (which is used only for neat output), and the left and right sets. Alternatively we can convert a standard numeric type into its canonical form.

```
julia> using SurrealNumbers
julia> x0 = SurrealShort("0", ∅, ∅)
julia> x1 = SurrealShort("1", [x0], ∅)
julia> x1_alt = convert(SurrealShort, 1)
julia> x1 == x1_alt
julia> surreal2dag(x1)
```



(a) $0 == \{\overline{-1} \mid \overline{1}\}$. *(b)* $\overline{3/2}$. *(c) The form of* $\overline{1} + \overline{1/2}$. *(d) The form of* $\overline{3/4} + \overline{3/4}$.

Fig. 2. DAGs depicting the forms of surreal numbers. Note that there are equivalent forms, *e.g.*,$\{\emptyset \mid \emptyset\} \equiv \{\overline{-1} \mid \overline{1}\}$, within a form.

The statement `x1 == x1_alt` tests whether the two alternative constructions are identical; it returns `true`, as both are canonical. The last statement creates DOT output for input to the GraphVis program [7], which can layout the nodes of the DAG.

Some aspects of the implementation may be unexpected. First, Julia has a `Set` type that could be used to implement forms according to the definition, however the `SurrealShort` structure instead uses sorted arrays of surreal numbers. This allows us to replace an $O(NM)$ algorithm for comparison operations between surreals with an $O(1)$ operation (the order spoken of here is the number of initial comparisons at the root level, each of which requires recursive evaluation, so the reduction of complexity is even more important than it might seem). Comparisons are used frequently, and therefore speeding these up results in a large overall performance improvement.

Second, the `SurrealShort` structure is implemented as a mutable type. Julia can create faster code when immutable types are used, but putting hash values in the type itself avoids (recursive) calculation of hashes more than once per form, which improved performance in our tests.

4. Surreal arithmetic

The goal of this work was at least in part to be able to generate more examples of surreal arithmetic. A few small examples are included below: for instance, the commands

```
julia> x = dali(1) + dali(1/2)
julia> expand(x; level=2)
```

gives the following (the `expand` command writes it in full)

$$\overline{1} + \overline{1/2} == \{\{\{\emptyset \mid \emptyset\} \mid \{\{\emptyset \mid \emptyset\} \mid \emptyset\}\}, \{\{\emptyset \mid \emptyset\} \mid \emptyset\} \mid \{\{\{\emptyset \mid \emptyset\} \mid \emptyset\} \mid \emptyset\}\}.$$

The complexity of such expressions hinders our understanding, and so we display the form as a DAG in Fig. 2(c).

Multiplication can be likewise explored, but these become even more complicated: for instance

```
julia> x = dali(2) * dali(2)
julia> expand(x; level=2)
```

results in

$$\overline{2} \times \overline{2} == \{\{\{\{\{\{\emptyset \mid \{\emptyset \mid \emptyset\}\} \mid \{\{\emptyset \mid \emptyset\} \mid \emptyset\}\} \mid \{\{\{\emptyset \mid \emptyset\} \mid \emptyset\} \mid \emptyset\}\} \mid \{\{\{\{\emptyset \mid \emptyset\} \mid \emptyset\} \mid \emptyset\} \mid \emptyset\}\} \mid \{\{\{\{\{\emptyset \mid \emptyset\} \mid \emptyset\} \mid \emptyset\} \mid \emptyset\} \mid \emptyset\}\} \mid \emptyset\}.$$

This is the most complicated example previously published [19].

Multiplication grows in complexity *very* rapidly, so it is not surprising that more complicated results have not been seen. We can see this increase in complexity in DAG form in Fig. 3, which shows $\overline{2} \times \overline{3}$, and in Table 1, which shows a sample of simple multiplications that illustrate the growth rate of computational times.[2] The table shows, for instance, the size of the resulting DAGs in terms of numbers of nodes and edges as $s(\cdot)$, and $e(\cdot)$ respectively. These grow exponentially.

The table also illustrates the importance of the DAG representation combined with caching. If one used naïve recursion for the calculation, this would be the same as treating these structures as trees. Recursion through these would be the same as tracing all paths through the DAGs from the root node to $\overline{0}$. The number of such paths is given by $p(\cdot)$ in the table. Remember that recursive operations are defined in terms of other recursive operations so this growth means that even computations such as $\overline{3} \times \overline{3}$ are impractical without the DAG representation.

We can see this explicitly in the Haskell surreal numbers package [15], which uses the direct definition of multiplication. The Haskell code is actually faster for the smallest case,[3] because it avoids caching overhead, but the computation load grows so quickly, that it cannot compute $\overline{3} \times \overline{3}$: the process attempting to perform the calculation grew to consume 80 GB of RAM (the assigned memory limit for a single process on this machine) before

[2] Times were measured on Julia 1.0 on a single core of a Linux Mint v18 Intel i7-6900 K computer running at 3.2 GHz using Julia's @timed macro.

[3] The comparison between compute times in Julia and Haskell should be taken with pinch of salt because (i) they use different timing mechanisms (Julia uses the @timed macro, and Haskell uses the criterion package [20]), and (ii) Haskell uses lazy evaluation, so to see performance we have to trigger evaluation, which in this case induces a small amount of overhead not included in the Julia code.

Table 1
Size and computational characteristics of simple product of surreal forms. The generation (or depth) $g(\cdot)$, and size of the DAG in terms of number of nodes $s(\cdot)$ and edges $e(\cdot)$ are given. Calculation of the number of paths from the root of the graph is given in $p(\cdot)$. Time (in seconds) for the computation and the number of unique surreal addition, multiplication and construction operations are also reported. Not all computations result in the construction of a form because some are 'short-circuited'.

$x \times y$	Measures of the size of the resultant form				Time (s)		Operations		
	g(xy)	s(xy)	e(xy)	p(xy)	Julia	Haskell [15]	+	×	Construction
$\overline{1} \times \overline{3}$	3	4	3	1	0.00001	0.000001	3	7	3
$\overline{2} \times \overline{3}$	12	45	82	625	0.00028	0.0004	83	9	69
$\overline{3} \times \overline{3}$	31	737	2052	9.4E+10	0.13	–	3419	10	3182
$\overline{4} \times \overline{3}$	64	16958	64653	6.1E+27	120.5	–	286358	14	281775
$\overline{5} \times \overline{3}$	115	430591	2094360	2.8E+37	102553.0	–	22024924	18	21919195

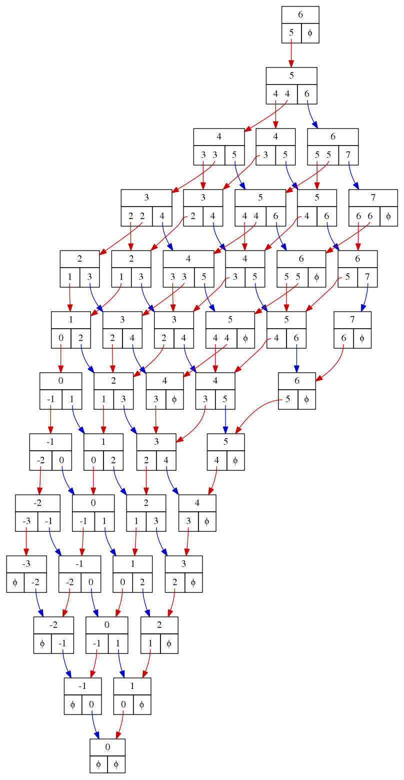


Fig. 3. The form of $\overline{2} \times \overline{3}$. Note that in these, we frequently see forms that might superficially appear identical, *e.g.*,$4 = \{3 \mid 5\}$ appears three times, but these are not identical forms, which is implicit in the DAG descending from them.

being terminated without outcome. The only other implementation of multiplication [11,14] use the same direct implementation. Ironically, in reading the comments in the Haskell implementation after compiling and testing, we found the statement "Don't you dare do 3 * 3 *[sic]*".

Caching allows us to reuse sub-components and thus perform the calculations much faster. The effectiveness of the caching can be seen Table 1 in the final three columns.[4] Caching multiplications reduces their number substantially. However, multiplication is defined in such a way that its outputs are then used as operands in summations in many combinations, leading to an explosion of different surreal forms in the final DAG. Thus the computation costs still grow very rapidly.

The size of problem grows so quickly it is difficult to benchmark, and there is little to compare against. The only other examples of multiplication [11,14,15] all use the direct implementation, which cannot solve simple cases such as $\overline{3} \times \overline{3}$ (an early Julia direct implementation of multiplication suffered the same fate, so this is not a programming language problem).

The problem at the heart is that even though we cache *results* of calculations, there are many calculations that lead to the same surreal form, but are not cached. For instance, in calculating 2×4, we get a form for 8 that has ancestor 5, but there are multiple different calculations in the product that arrive at the same form of 5:

$$\begin{aligned} 5 &== ((\overline{3} + ((\overline{2} + \overline{2}) + -\overline{1})) + -\overline{1}) \\ &== (\overline{2} + ((\overline{1} + (\overline{2} + \overline{2})) + -(\overline{1} + \overline{1}))) \\ &== (\overline{4} + ((\overline{1} + \overline{2}) + -(\overline{1} + \overline{1}))) \\ &== ((\overline{3} + (\overline{2} + \overline{2})) + -(\overline{1} + \overline{1})) \\ &== (\overline{3} + ((\overline{2} + \overline{2}) + -(\overline{1} + \overline{1}))). \end{aligned}$$

Caching avoids performing $\overline{2} + \overline{2}$ multiple times, and can remove some computations altogether: for instance commutativity means that we can cache $y + x$ whenever we calculate $x + y$. Using commutativity and other tricks we save many repeated calculations, though care must be taken though because many of the reported properties of surreal arithmetic are properties of arithmetic on the values, not the forms. For instance forms of 0 other than $\overline{0}$ are additive identities for surreal *numbers* but not for *forms*, a fact that can now be quickly verified with the toolkit.

However, even though all of the above expressions for 5 have the same result, we cannot cache that in advance, because before performing the computation we do not know the result. This is not onerous in the case above, but as the size of the surreal forms increases, the number of possibilities in such calculations expands dramatically. This leads to the observed explosion in addition operations, and the resulting computational crisis. One approach to improve the situation would be to further exploit the algebraic rules of surreal numbers to create larger tables of results from their components, but to be much more valuable than the current tables, they would need to go beyond 2D, and thus there will be a complex memory/computation trade-off that goes beyond the scope of this paper.

5. Discussion

The headline point of the implementation is that the DAG structure, which we initially used only as a visualisation aid, is also important in order to make the computations efficient. However there are other issues we care about.

In particular speed of computation is sometimes traded off against ease in writing code. The experience of implementing surreals in Julia was largely positive. Julia provides all the facilities one expects of a modern programming language, plus some additional features such as automatic extension of functions to array operations. Most syntax is natural for someone experienced in languages such as Python, R or Matlab. The closest is Matlab, but there are some notable differences that (once accepted) are more natural.

[4] Numbers of operations were counted using lightweight instrumentation built into the operators, which create a small additional overhead.

However, there were some negatives. The most notable were changes in Julia versions including breaking changes to the core language. However, with the release of 1.0 we expect these to cease. Also these changes came with large improvements. Comparing computation times for $\overline{3} \times \overline{5}$, these were reduced from 39.9 to 28.5 h in going from Julia 0.6 to 1.0, a very substantial improvement.

Another experience that could improve is the addition of better tools to understand memory usage. As with many such languages Julia takes care of memory allocation and garbage collection. It provides some tools to understand how much memory is used, but in complicated situations these could provide clearer information.

Part of the intention here is also to show the benevolent relationship between computing and mathematics. A computer program can provide examples, which lead to hypotheses, which lead eventually to new theorems. One such example concerns surreal *birthday* arithmetic.

The construction of surreals leads to the notion of *birthdays*: take 0 to be born on day 0, and ± 1 to be born on day 1, and so on, then we can assign a birthday to all surreals. I prefer the term *generation* over birthday if only because it links up to the notion of parents and children more cleanly. In the graphs above, it is the length of the longest path from the top to the bottom of the DAG, *i.e.,*its *depth*.

Formally, if we denote the parents of x by $X_P = X_L \cup X_R$, then we can define mathematically the generation/birthday function of a surreal number form x by

$$g(x) = \sup_{s \in X_P} g(s) + 1, \tag{1}$$

where $g(\overline{0}) = 0$. In terms of $g(\cdot)$, a surreal form x is *simpler* or *older* than y means $g(x) < g(y)$. For a dyadic $x = n/2^k$ (presuming n odd and $k \geq 0$)

$$g\big(d(x)\big) = \lceil\, |x| \,\rceil + k.$$

For instance, the generation of canonical integers is equal to their absolute value $g(\overline{n}) = |n|$. Generations of some other examples are listed in Table 1.

There is some interest in "birthday arithmetic", *i.e.,* consideration of the birthdays of outputs of surreal operations. For instance, Simons [6, pp. 25–26] proves $g(x+y) \leq g(x)+g(y)$. Proofs of surreal properties are usually inductive, and Simons presents an elegant and general formulation of such.

With the toolbox presented here it is easy to generate a table of additions, and their birthdays/generations. One quickly notes that these satisfy $g(x+y) = g(x)+g(y)$. It is quick work to extend Simons' proof to demonstrate equality, but one might never have thought to do so without seeing several examples.

Simons goes on to ask whether $g(xy) \leq g(x)g(y)$, stating "It is an obvious enough conjecture, with no obvious counter-examples". However, quick examination of Table 1 shows that $g\big(\overline{2} \times \overline{3}\big) = 12$, and $g\big(\overline{2}\big)g\big(\overline{3}\big) = 6$. This is an immediate counter example to Simons' conjecture (there are several others listed in the table). Gonshor [21, Theorem 6.2] provides a legitimate bound: $g(xy) \leq 3^{g(x)+g(y)}$, but the results in the table show that this is a *very* weak upper bound. For instance for $\overline{4} \times \overline{3}$ Gonshor's bounds would be $3^{12} \simeq 500,000$, the actual value being 64.

These results on birthdays arithmetic are relatively trivial in the general scheme of things, but they serve to illustrate the value of being able to perform computations with surreal forms. Ability to play with examples can lead to the better hypotheses, and thence new results.

6. Conclusion

This paper presents a toolkit for working with surreal number forms. It provides most of the facilities for working with short surreals (surreals with a finite representation) that one would expect, and thus provides a means to generate examples in order to learn about the surreal numbers.

Short surreals are limiting — we cannot do division without long surreals, but these require infinite sets. Although it is not possible to explicitly list an infinite set, we could still represent it using lazy evaluation. Thus there is a way forward towards representing long surreals yet to be explored. But before this there is still room to make the computations more efficient. At present we can perform only small multiplications, and this should be improved, though there appear to be fundamental limitations simply because surreal forms resulting from multiplication grow in size so quickly. Parallelisation of algorithms should provide some improvements, but requires some deeper thought in order to minimise communication overhead.

References

[1] Conway JH. On numbers and games. Peters series, Taylor & Francis; 2000, ISBN: 9781568811277.

[2] Knuth DE. Surreal numbers: How two ex-students turned on to pure mathematics and found total happiness: A mathematical novelette. Addison-Wesley Publishing Company; 1974, ISBN: 9780201038125.

[3] Tøndering Claus. Surreal numbers – an introduction, version 1.6. 2013, https://www.tondering.dk/download/sur16pdf, [Accessed 25 September 2018].

[4] Grimm Gretchen. An introduction to surreal numbers. 2012, https://www.whitman.edu/Documents/Academics/Mathematics/Grimm.pdf, [Accessed 25 September 2018].

[5] Proof of Conway's simplicity rule for surreal numbers. 2014, https://math.stackexchange.com/questions/816540/proof-of-conways-simplicity-rule-for-surreal-numbers, [Accessed 6 June 2018].

[6] Simons Jim. Meet the surreal numbers. April 2017, https://www.m-a.org.uk/resources/.../4H-Jim-Simons-Meet-the-surreal-numbers.pdf.

[7] Ellson John, Gansner Emden, Koutsofios Lefteris, North Stephen C, Woodhull Gordon. Graphviz–open source graph drawing tools. In: Mutzel Petra, Jünger Michael, Leipert Sebastian, editors. Graph drawing. Lecture notes in computer science, vol. 2265, Berlin, Heidelberg: Springer; 2002, p. 483–4. http://dx.doi.org/10.1007/3-540-45848-4_57,

[8] Bezanson Jeff, Edelman Alan, Karpinski Stefan, Shah Viral B. Julia: a fresh approach to numerical computing. SIAM Rev 2017;59(1):65–98.

[9] Keddie P. Ordinal operations on surreal numbers. Bull Lond Math Soc 1994;26(6):531–8.

[10] Schleicher Dierk, Stoll Michael. An introduction to Conway's games and numbers. Mosc Math J 2006;6(2):359–88.

[11] Surreal numbers (in Lua). 2014, https://githubcom/Ryan1729/lua-Surreal-numbers?files=1, [Accessed 6 September 2018].

[12] Surreal numbers calculator (in Javascript). 2018, http://www.iwriteiam.nl/SurrealNumbers.html, [Accessed 6 September 2018].

[13] Mamane Lionel Elie. Surreal numbers in Coq. In: Proceedings of the 2004 international conference on types for proofs and programs. TYPES'04, Berlin, Heidelberg: Springer-Verlag; 2006, p. 170–85, ISBN: 3-540-31428-8, 978-3-540-31428-8, http://dx.doi.org/10.1007/11617990_11.

[14] A Haskell library representing the class of surreal numbers. 2014, https://github.com/Lacaranian/surreal, [Accessed 6 September 2018].

[15] Haskell-surreals. 2014, https://github.com/elfakyn/Haskell-surreals, [Accessed 6 September 2018].

[16] Cavines Ken. Generating the surreal numbers (in mathematica). 2014, http://demonstrations.wolfram.com/GeneratingTheSurrealNumbers/, [Accessed 6 September 2018].

[17] Surreal numbers in Python. 2010, https://githubcom/codeinthehole/python-surreal, [Accessed 6 September 2018].

[18] Karpinski Stefan. Man creates one programming language to rule them all. WIRED 2014.

[19] Chu-Carroll Mark. Arithmetic with surreal numbers. August 2006, http://goodmath.scientopia.org/2006/08/21/arithmetic-with-surreal-numbers/.

[20] O'Sullivan Bryan. Criterion: a Haskell microbenchmarking library. 2019, http://www.serpentine.com/criterion/, [Accessed 10 January 2019].

[21] Gonshor Harry. An introduction to the theory of surreal numbers. London mathematical society lecture note series, Cambridge University Press; 1986.