Constructing Clifford Algebras using the Super Tensor Product

Dan Piponi

February 2024 (originally March 2023)

Some literate Haskell but little about this code is specific to Haskell...

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE UndecidableInstances #-}

import GHC.TypeLits
```

1 Introduction

This is a followup to Geometric Algebra for Free and More Low Cost Geometric Algebra.

In those articles I showed how you could build up the Clifford algebras like so:

```
type Cliff1  = Complex R
type Cliff1' = Split R
type Cliff2  = Quaternion R
type Cliff2' = Matrix R
type Cliff3  = Quaternion Cliff1'
type Cliff3' = Matrix Cliff1
```

```
type Cliff4  = Quaternion Cliff2'
type Cliff4' = Matrix Cliff2
type Cliff5  = Quaternion Cliff3'
...
```

I used `CliffN` as the Clifford algebra for a negative definite inner product and `CliffN'` for the positive definite case. It's not a completely uniform sequence in the sense that `CliffN` is built from `CliffN'` for dimension two lower and you use a mix of `Matrix` and `Quaternion`.

The core principle making this work is that for type constructors T implemented like `Matrix`, `Quaternion` etc. we have the property that

$TU\mathbb{R} = T\mathbb{R} \otimes U\mathbb{R}$

eg. `Matrix (Quaternion Float)` is effectively the same thing as `Matrix Float` $\otimes$ `Quaternion Float`.

But John Baez pointed out to me that you can build up the `CliffN` algebras much more simply enabling us to use these definitions:

```
type Cliff1 = Complex Float
type Cliff2 = Complex Cliff1
type Cliff3 = Complex Cliff2
type Cliff4 = Complex Cliff3
type Cliff5 = Complex Cliff4

...
```

Or even better:

```
type family Cliff (n :: Nat) :: * where
  Cliff 0 = Float
  Cliff n = Complex (Cliff (n - 1))
```

But there's one little catch. We have to work, not with the tensor product, but the **super tensor** product.

We define `Complex` the same way as before:

```
data Complex a = C a a deriving (Eq, Show)
```

Previously we used a definition of multiplication like this:

```
instance Num a => Num (Complex a) where
```

```
C a b * C c d = C (a * c - b * d) (a * d + b * c)
```

We can think of `C a b` in `Complex R` as representing the element $1 \otimes a + i \otimes b$ $\mathbb{C} \otimes R$. The definition of multiplication in a tensor product of algebras is defined by $(a \otimes b)(c \otimes d) = (ac) \otimes (bd)$. So we have

$$\begin{aligned}
&(1 \otimes a + i \otimes b)(1 \otimes c + i \otimes d) \\
&= (1 \otimes a)(1 \otimes c) + (1 \otimes a)(i \otimes d) + (i \otimes b)(1 \otimes c) + (i \otimes b)(i \otimes d) \\
&= 1 \otimes (ac) + i \otimes (ad) + i \otimes (bc) + i^2 \otimes (bd) \\
&= 1 \otimes (ac - bd) + i \otimes (ad + bc)
\end{aligned}$$

This means that line of code we wrote above defining `*` for `Complex` isn't simply a definition of multiplication of complex numbers, it says how to multiply in an algebra tensored with the complex numbers.

2 Let's go Super!

A superalgebra is an algebra graded by $\mathbb{Z}_2$ where $\mathbb{Z}_2$ is the ring of integers modulo 2. What that means is that we have some algebra A that can be broken down as a direct sum $A_0 \oplus A_1$ (the subscripts live in $\mathbb{Z}_2$) with the property that multiplication respects the grading, ie. if x is in A_i and y is in A_j then xy is in A_{i+j}.

The elements of A_0 are called "even" (or bosonic) and those in A_1 "odd" (or fermionic). Often even elements commute with everything and odd elements anticommute with each other but this isn't always the case. (The superalgebra is said to be supercommutative when this happens. This is a common pattern: a thing X becomes a superX if it has odd and even parts and swapping two odd things introduces a sign flip.)

The super tensor product is much like the tensor product but it respects the grading. This means that if x is in A_i and y is in B_j then $x \otimes y$ is in $(A \otimes B)_{i+j}$. From now on I'm using $\otimes$ to mean super tensor product.

Multiplication in the super tensor product of two superalgebras A and B is now defined by the following modified rule: if b is in B_i and c is in A_j then $(a \otimes b)(c \otimes d) = (-1)^{ij}(ac) \otimes (bd)$. Note that the sign flip arises when we shuffle an odd c left past an odd b.

The neat fact that John pointed out to me is that $Cliff_n = \mathbb{C} \otimes \mathbb{C} \otimes \ldots$ n times $\ldots \otimes \mathbb{C}$.

We have to modify our definition of `\otimes` to take into account that sign flip.

I initially wrote a whole lot of code to define a superalgebra as a pair of algebras with four multiplication operations and it got a bit messy. But I noticed that the only specifically superalgebraic operation I ever performed on an element of a superalgebra was negating the odd part of an element.

So I could define `SuperAlgebra` like so:

```
class SuperAlgebra a where
  conjugation :: a -> a
```

where `conjugation` is the negation of the odd part.

(I'm not sure if this operation corresponds to what is usually called conjugation in this branch of mathematics.)

But there's a little efficiency optimization I want to write. If I used the above definition, then later I'd often find myself computing a whole lot of negations in a row. This means applying `negate` to many elements of large algebraic objects even though any pair of them cancel each other's effect. So I add a little flag to my conjugation function that is used to say we want an extra negation and we can accumulate flips of a flag rather than flips of lots of elements.

```
class SuperAlgebra a where
  conjugation :: Bool -> a -> a
```

Here's our first instance:

```
instance SuperAlgebra Float where
  conjugation False x = x
  conjugation True x = negate x
```

This is saying that the conjugation is the identity on `Float` but if we want to perform an extra flip we can set the flag to `True`. Maybe I should call it `conjugationWithOptionalExtraNegation`.

And now comes the first bit of non-trivial superalgebra:

```
instance (Num a, SuperAlgebra a) =>
         SuperAlgebra (Complex a) where
  conjugation e (C a b) = C (conjugation e a)
                            (conjugation (not e) b)
```

We consider 1 to be even and i to be odd. When we apply the conjugation to $1 \otimes a + i \otimes b$ then we can just apply it directly to a. But that $i\otimes$ flips the "parity" of b (because tensor product respects the grading) so we need to swap when we use the conjugation. And that should explain why `conjugation` is defined the way it is.

Now we can use the modified rule for $\mathbb{C}\otimes$ defined above:

```
instance (Num a, SuperAlgebra a) => Num (Complex a) where
  fromInteger n = C (fromInteger n) 0
  C a b + C a' b' = C (a + a') (b + b')
  C a b * C c d = C (a * c - conjugation False b * d)
                    (conjugation False a * d + b * c)
  negate (C a b) = C (negate a) (negate b)
  abs = undefined
  signum = undefined
```

For example, `conjugation False` is applied to the first b on the RHS because d implicitly represents an id term and when expanding out the product we shuffle the (odd) i in id left of b. It doesn't get applied to the second ib because ib and c remain in the same order.

That's it!

3 Tests

I'll test it with some examples from `Cliff3`:

```
class HasBasis a where
  e :: Integer -> a

instance HasBasis Float where
  e = undefined

instance (Num a, HasBasis a) => HasBasis (Complex a) where
```

```
  e 0 = C 0 1
  e n = C (e (n - 1)) 0

make a b c d e f g h =
  C (C (C a b) (C c d))
    (C (C e f) (C g h))

e1, e2, e3, e21, e31, e32, e321 :: Cliff 3
e1 = e 0
e2 = e 1
e21 = e2 * e1
e3 = e 2
e31 = e3 * e1
e32 = e3 * e2
e321 = e3 * e2 * e1

main = do
    print (e1 * e1 + 1 == 0)
    print (e31 * e31 + 1 == 0)
    print (e3 * e3 + 1 == 0)
    print (e21 * e21 + 1 == 0)
    print (e2 * e2 + 1 == 0)
    print (e32 * e32 + 1 == 0)
    print (e321 * e321 - 1 == 0)
    print (e3 * e2 * e1 - e321 == 0)
    print (e2 * e1 - e21 == 0)
    print (e3 * e1 - e31 == 0)
    print (e3 * e2 - e32 == 0)
    print (e21 * e32 - e31 == 0)
```

4 Observation

The implementation of multiplication looks remarkably like it's the Constructing Clifford Algebras using the Super Tensor Product construction. It can't be (because iterating it three times gives you a non-associative algebra but the Clifford algebras are associative). Nonetheless, I think comparison with Cayley-Dickson may be useful.

5 Efficiency

As mentioned above, before I realised I just needed the `conjugation` operation I wrote the above code with an explicit split of a superalgebra into two pieces intertwined by four multiplications. I think the previous approach may have a big advantage - it may be possible to use variations on the well known "speed-up" of complex multiplication that uses three real multiplications instead of four. This should lead to a fast implementation of Clifford algebras.

Also be warned: you can kill GHC if you turn on optimization and try to multiply elements of high-dimensional Clifford algebras. I think it tries to inline absolutely everything and you end up with a block of code that grows exponentially with n.

Note also that this code translates directly into many languages.