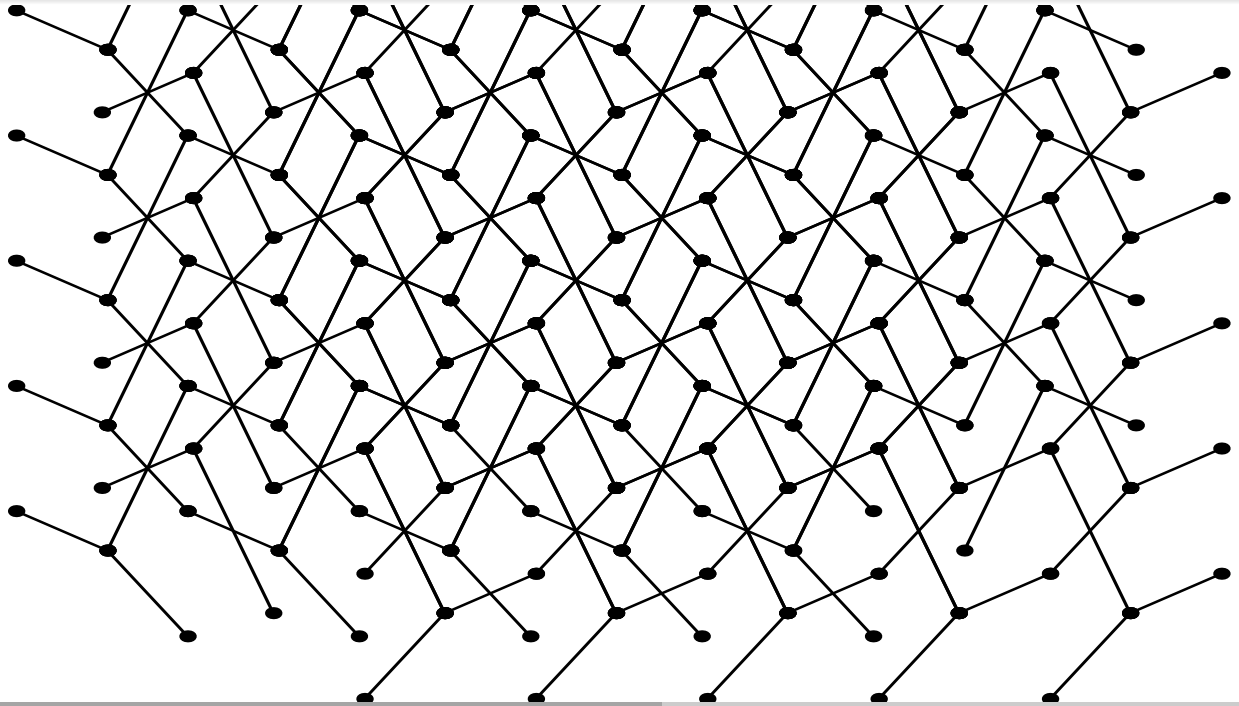
Functional Geometry (folds in computed forms)

Pablo Colapinto
Robert W. Deutsch Fellow at the AlloSphere Research Group
UCSB, Media Arts and Technology Program
C++Now May 16, 2015



1. Motivations
2. Geometry
3. Arrows
4. Folds

The study of models is the study of man
–Robert Rosen, Anticipatory Systems

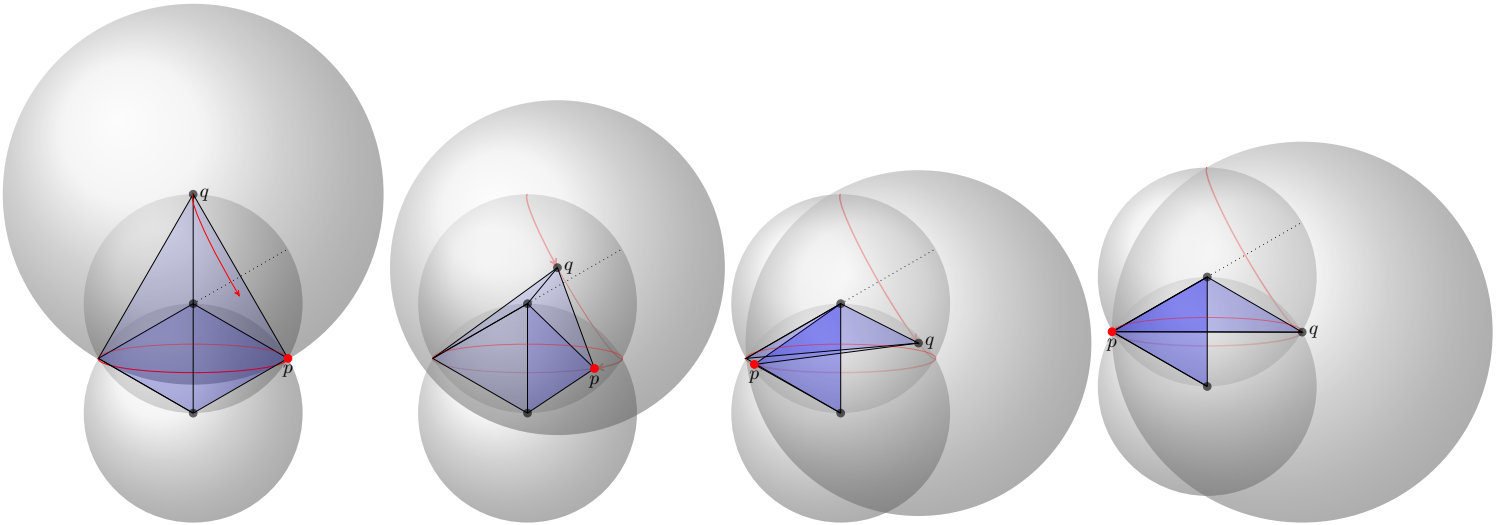
The obstacles of achieving a facile relationship of people and things seems to inhere not so much in the structure of things themselves as the structure of our ideas and values.

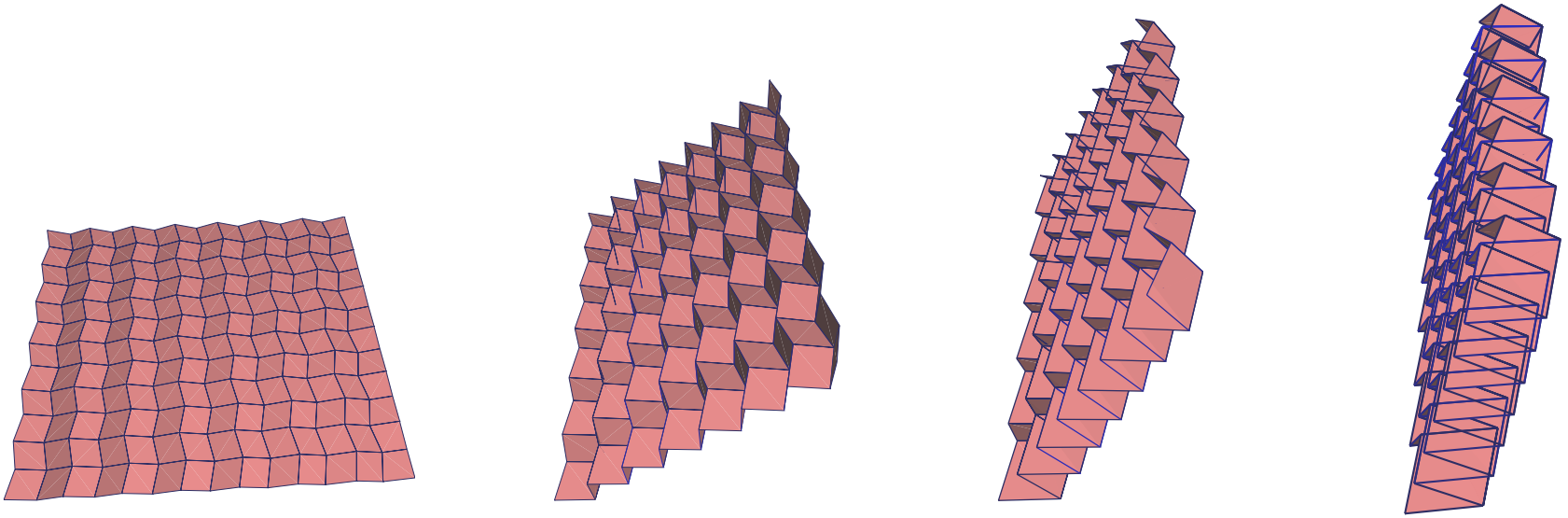
– Ron Resch, *The Topological Design of Sculptural and Architectural Systems*, 1973.

Our **Program** is To Map Function to Structure

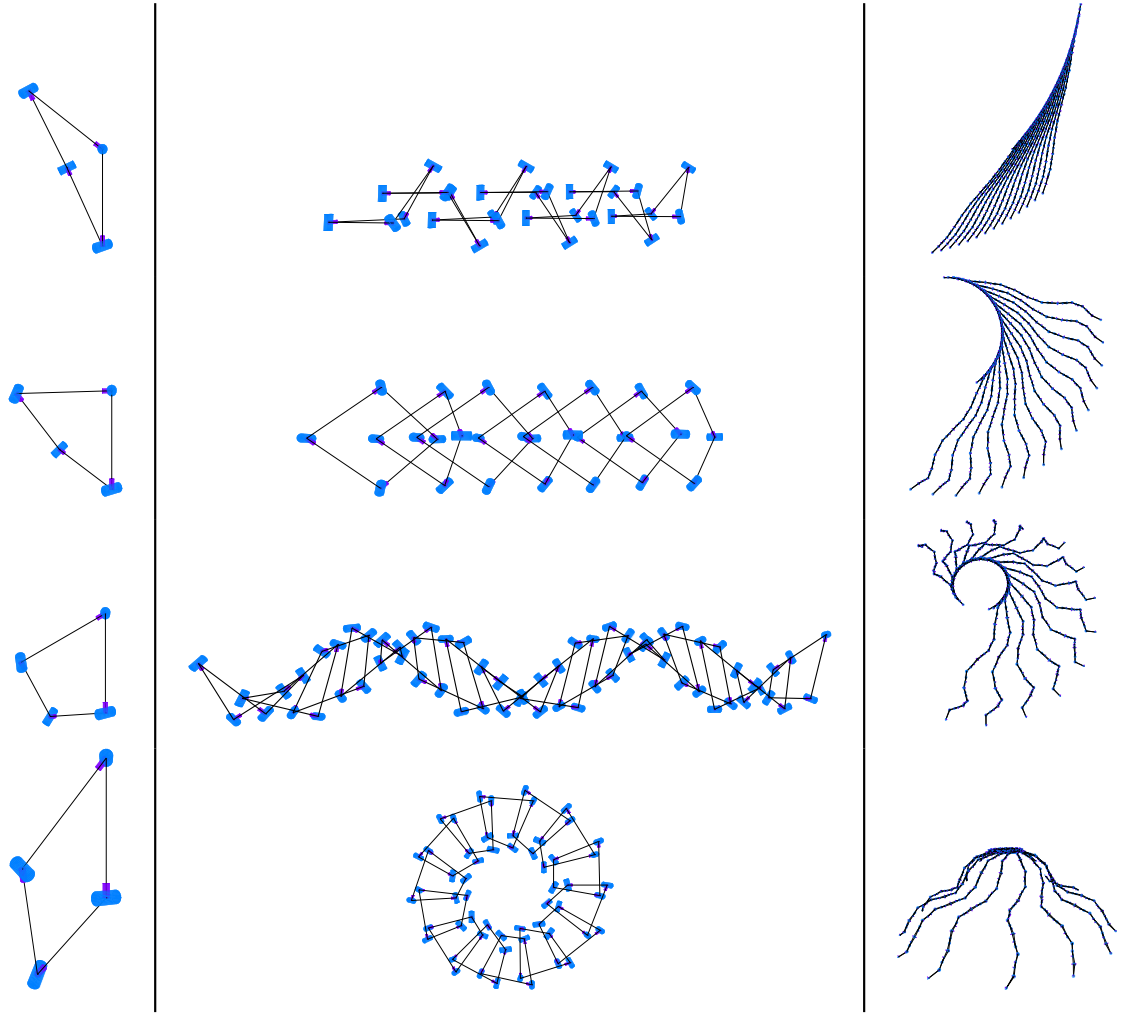
$$f : a \longrightarrow b$$

Structured Morphisms → Morphing Structures



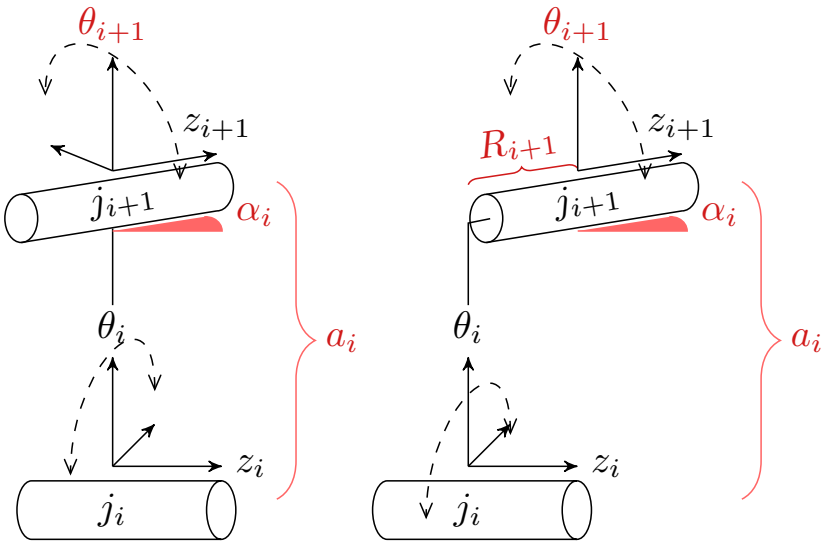


Bennett Spatial Linkage Mechanism



Networked Bennett mechanisms inspired by the work of Z. You and Y. Chen's *Motion Structures* (Spon Press, 2012).

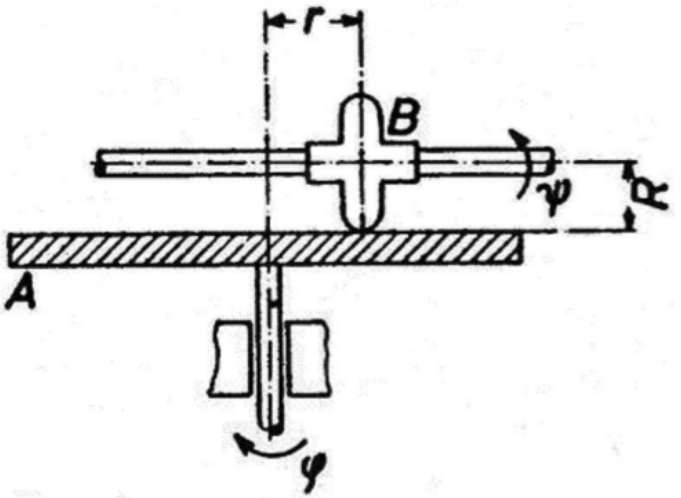
Fold-Expressions→ Folding Transformations



$$\mathcal{M}_i = \mathcal{M}_{i-1}\mathcal{M}_{L_{i-1}}\mathcal{M}_{J_i} \tag{1}$$

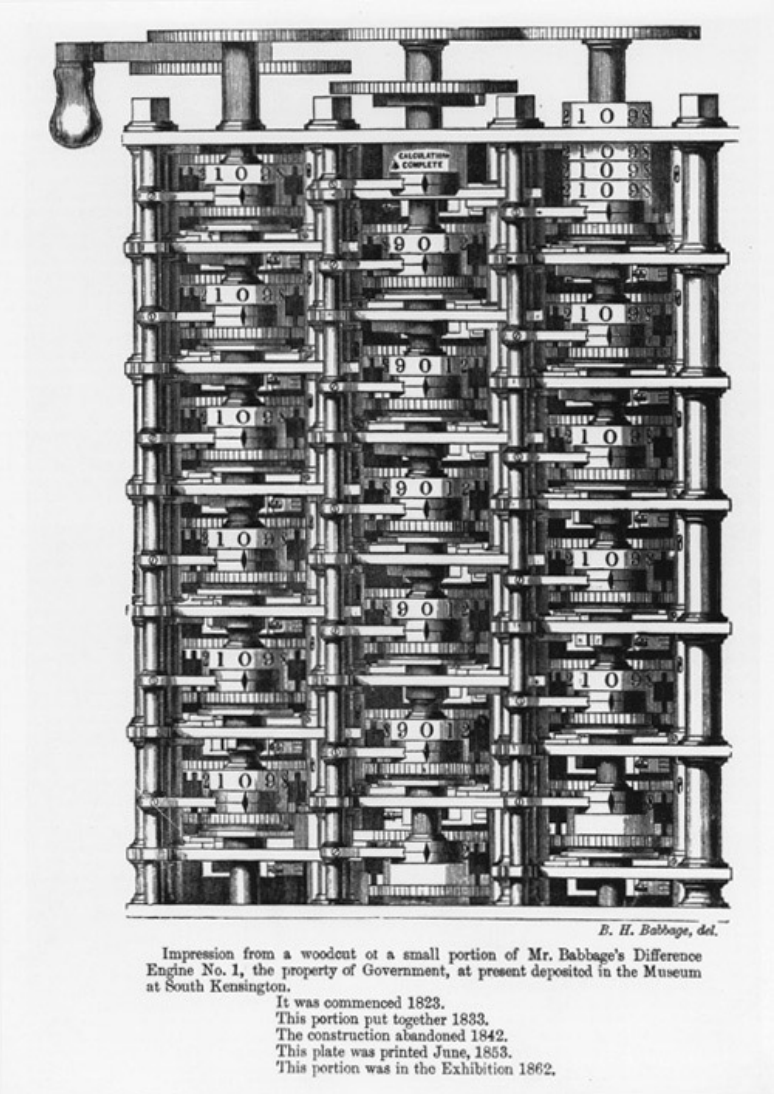
Space, Which Calculates

$$R\,d\psi = r\,d\varphi$$
$$\psi = \frac{1}{R}\int r\,d\varphi$$



Integration Calculations. Image from Konrad Zuse *Calculating Space* "Rechnender Raum", 1969 better translated as: *Space which Calculates*

Computing with Gears in 1822



B. H. Babbage, del.

Impression from a woodcut of a small portion of Mr. Babbage's Difference Engine No. 1, the property of Government, at present deposited in the Museum at South Kensington.

It was commenced 1823.
This portion put together 1833.
The construction abandoned 1842.
This plate was printed June, 1853.
This portion was in the Exhibition 1862.

Computing with Linkages in 1876

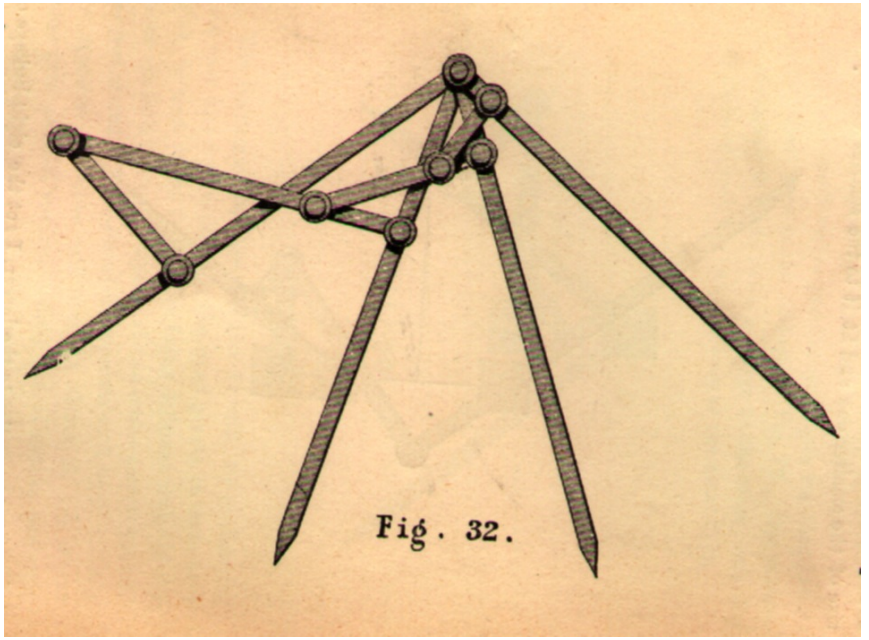


Figure: Kempe's Angle **Multiplicator**. He also invented an **Additor** and **Translator**. In 1876, Kempe (mis)proved the *Universality Theorem* – there is a linkage which can trace any planar curve.

Computing with Folds in 1936

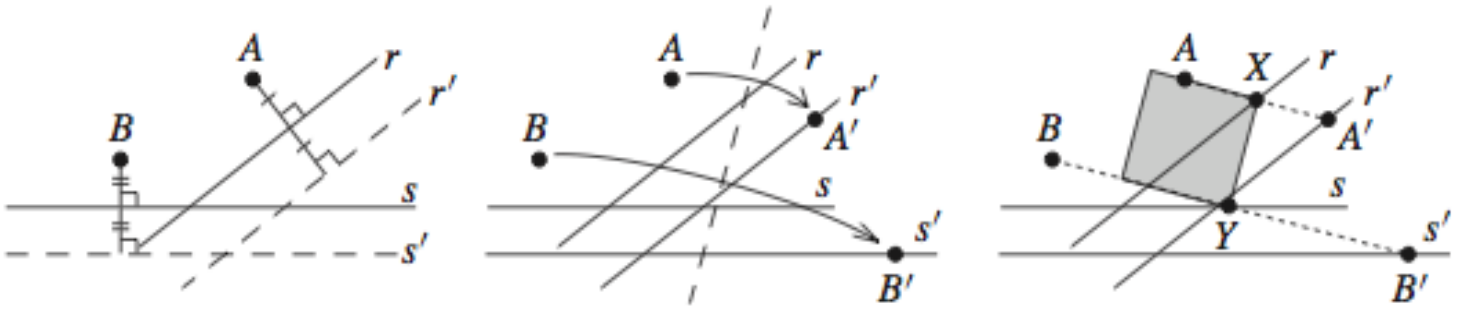
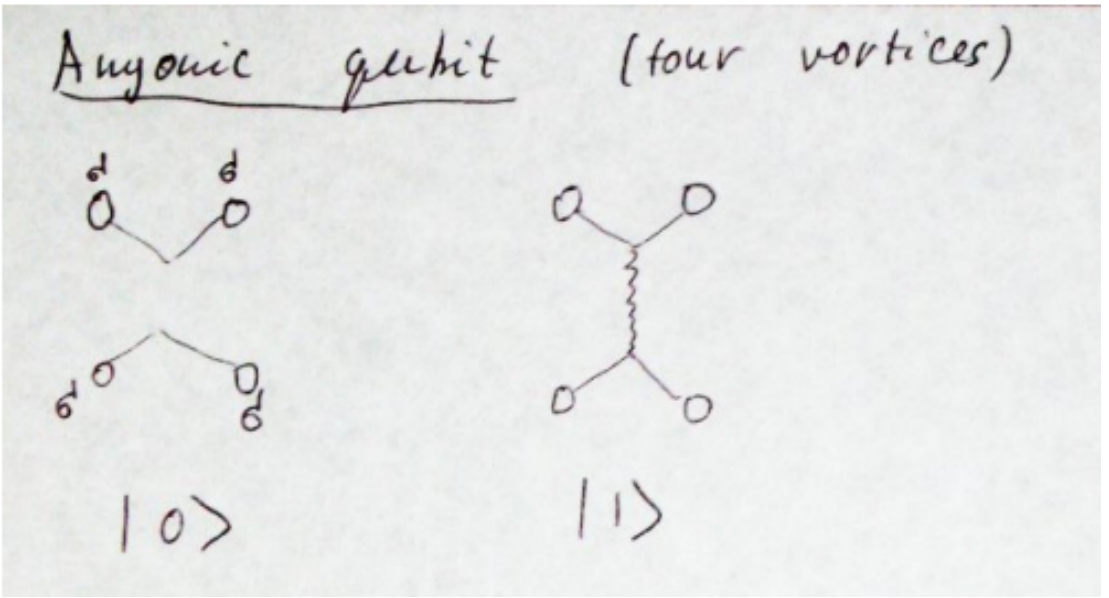


Figure: Beloch's fold to solve the cube-root of 2. From Hull, Thomas C., "*Solving Cubics With Creases: The Work of Beloch and Lill*", The American Mathematical Monthly 118, 4 (2011), pp. pp. 307-315.

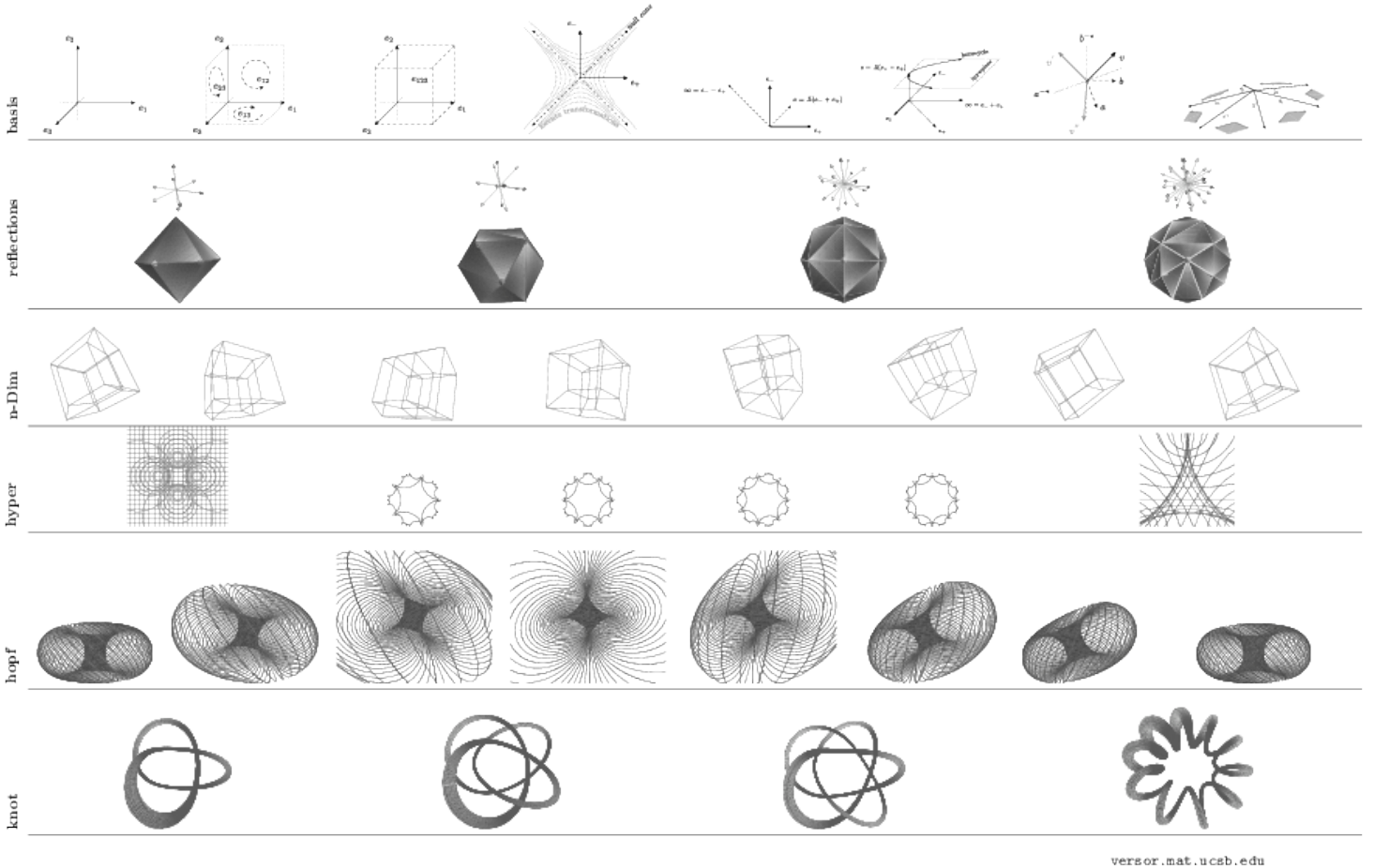
Computing with Braids in 1997



Kitaev's Topological Quantum Computation

Image: Kitaev's Lecture Notes at
http://pitp.physics.ubc.ca/archives/misc/kitaev/

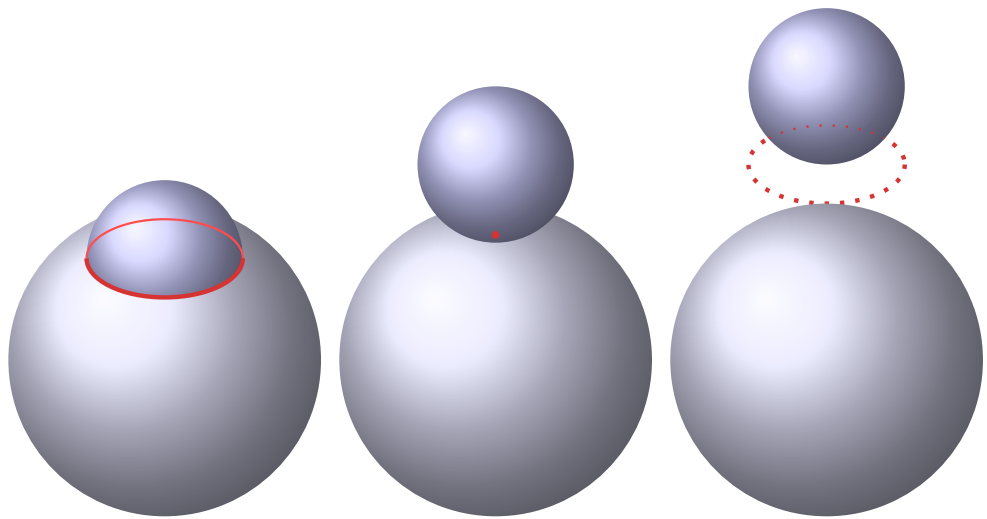
Versor is a C++ library for *forming*



The PROBLEM of **SPATIAL** ARTICULATION

Q: How do we **pose** questions to space?

A: Use **forms** in **well-formed** formulas.

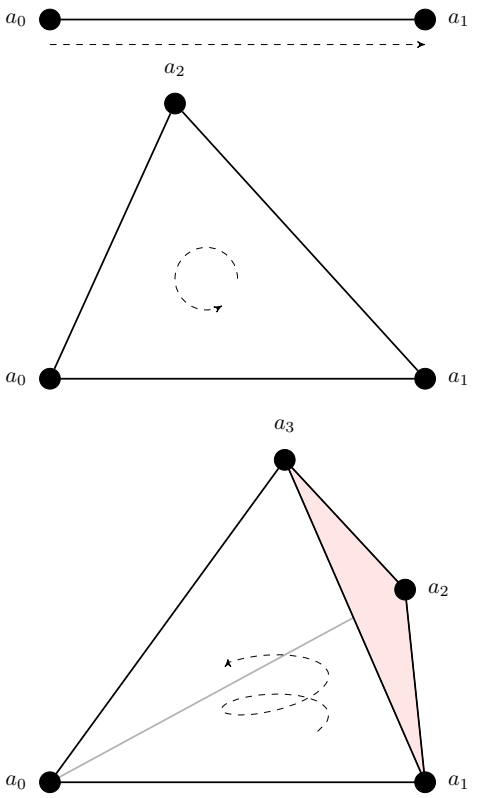


The Problem of **Spatial Articulation** is Two-fold

1. A Computation Problem: Articulating **Movements**
2. A Design Problem: Articulating **Concepts**

The Nature of Space is (a) Complex

- **Complicate**: to fold
- **Evolve**: to unfold



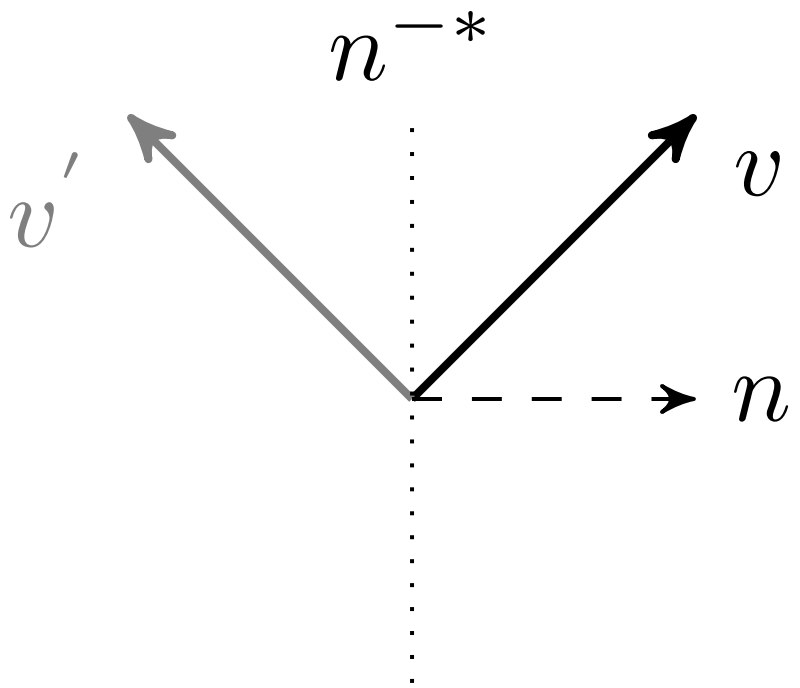
Geometry $\Longleftrightarrow$ Function

GA		**FP**
language of space		language of process
coordinate free	$\int_M dL = \int_{\partial M} L$	point free
covariant	$F(va) = F(v)F(a)$	composable
implicit	$\kappa = p_a \wedge p_b \wedge p_c$	declarative
n-dimensional		generic
denotes form		denotes process
universal construction of relations		
STRUCTURE PRESERVING MAPPINGS		

Point-free Expressions

sum x:xs = x + sum xs
sum = fold (+) 0
gets rid of need to specify “head” and “tail”

Coordinate-free Expressions



- $v' = -nvn$
- n is a **versor**
- n^{-*} is the **dual** of n

gets rid of need to specify x y and z (or **dimension** or **metric**)

N-dim

rotations in any dimension

```
1  /*! ND Rotor from ND Bivector b*/
2  template<class A>
3  auto rot ( const A& b ) ->
4          decltype( b + 1 )
5  {
6      A::value_t  c = sqrt(- ( b.wt() ) );
7          A::value_t  sc = -sin(c);
8          if (c != 0) sc /= c;
9          return b * sc + cos(c);
10 }
```

David Hestenes Resurrects Clifford's Geometric Algebra in the 60s

- *Multivector Calculus*
- *Spacetime Algebra*
- *New Foundations for Classical Mechanics*
- *Geometric Algebra to Geometric Calculus*

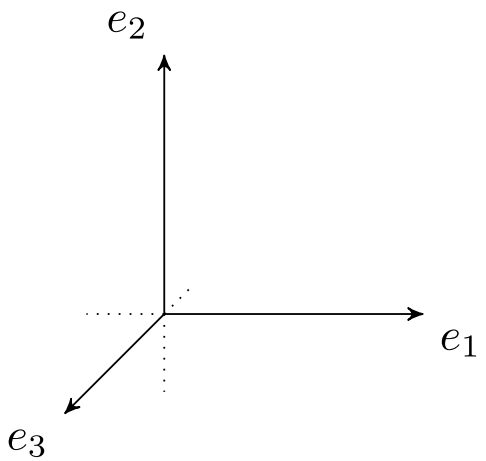
See Also:

- G. Sobczyk, A. Lasenby, J. Lasenby, E. Bayro-Corrochano, C. Doran, C. Perwass, L. Dorst, S. Mann, D. Fontijne, R. Wareham, J. Cameron, ...

Many Applications of Geometric Algebra

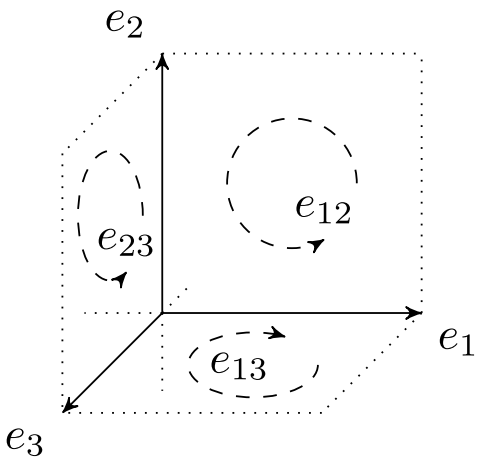
- Classical Mechanics and Particle Physics
- Molecular Modeling and Crystallography
- Electrodynamics and Optics
- Digital Signal Processing and Computer Vision
- Robotics and Kinematics
- Relativistic Physics and Gauge Theory
- 3D Computer Graphics and Experimental Visualization

↗ Vectors are Directed Magnitudes



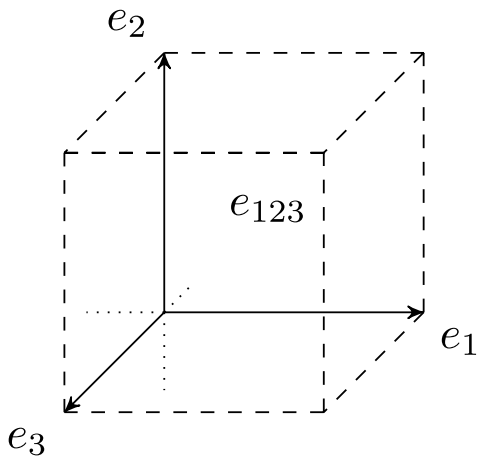
Linear combinations of these basis blades define a vector :
$\mathbf{v} = \alpha \mathbf{e}_1 + \beta \mathbf{e}_2 + \gamma \mathbf{e}_3$.

Bivectors are Directed Areas



Basis 2-blades e_{12},e_{13}, and e_{23} in G^3 represent directed unit areas. Linear combinations of these basis blades define a plane or bivector : $\mathbf{B} = \boldsymbol{v}_a \wedge \boldsymbol{v}_b = \alpha e_{12} + \beta e_{13} + \gamma e_{23}$.

The Pseudoscalar is a Directed Volume



The basis trivector e_{123} in G^3 is also known as the pseudoscalar I. As the highest grade blade I is sometimes referred to as the *tangent space*.

$$I = \bigwedge_{i=1}^{n} e_i = e_1 \wedge e_2 \wedge e_3 = e_{123} \tag{2}$$

Simple Rules To Type Generic Spaces

The **inner product** of basis blades (in a Euclidean Metric) :

$$e_i \cdot e_j = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases}$$

The **outer product** annhilates if the blades are the same :

$$e_i \wedge e_j = \begin{cases} e_{ij} & i \neq j \\ 0 & i = j \end{cases}$$

The **geometric product**, unique to GA, is the sum of these two :

$$e_i e_j = e_i \cdot e_j + e_i \wedge e_j = \begin{cases} e_{ij} & i \neq j \\ 1 & i = j \end{cases}$$

A **ring of types**

Anti-commutation powers the orientability of the algebra (really useful for spatial calculations !)

$$e_{ij} = -e_{ji}$$

so, for example :

$$e_{23}e_2 = e_{232} = -e_{223} = -e_3$$

and :

$$e_{12}^2 = e_{12}e_{12} = e_{1212} = -e_{1122} = -1$$

where 1 is the *zero object*

Compile-Time Combinatorics

$$
\begin{aligned}
ab &= (a_1 e_1 + a_2 e_2 + a_3 e_3) * (b_1 e_1 + b_2 e_2 + b_3 e_3) \\
&= a_1(b_1 e_1 e_1 + b_2 e_1 e_2 + b_3 e_1 e_3) + a_2(b_1 e_2 e_1 + b_2 e_2 e_2 + b_3 e_2 e_3) \\
&\quad + a_3(b_1 e_3 e_1 + b_2 e_3 e_2 + b_3 e_3 e_3) \\
&= a_1(b_1 + b_2 e_{12} + b_3 e_{13}) + a_2(-b_1 e_{12} + b_2 + b_3 e_{23}) + a_3(-b_1 e_{13} - b_2 e_{23} + b_3) \\
&= (a_1 b_1 + a_2 b_2 + a_3 b_3) + (a_1 b_2 - a_2 b_1) e_{12} \\
&\quad + (a_1 b_3 - a_3 b_1) e_{13} + (a_2 b_3 - a_3 b_2) e_{23}
\end{aligned}
$$

Generate Compile-Time Combinatorics

some combinatorics : product of bivectors a and b in twistor algebra :

```
-a[5] * b[5] /*s*/ a[4] * b[4] /*s*/ a[3] * b[3] /*s*/ a[2] * b[2]
/*s*/ a[1] * b[1] /*s*/ -a[0] * b[0] /*s*/
-a[4] * b[3] /*e12*/ a[3] * b[4] /*e12*/ -a[2] * b[1] /*e12*/ a[1] *
b[2] /*e12*/ -a[5] * b[3] /*e13*/ a[3] * b[5] /*e13*/ -a[2] * b[0]
/*e13*/ a[0] * b[2] /*e13*/ -a[5] * b[4] /*e23*/ a[4] * b[5]
/*e23*/ a[1] * b[0] /*e23*/ -a[0] * b[1] /*e23*/ a[5] * b[1]
/*e14*/ -a[4] * b[0] /*e14*/ -a[1] * b[5] /*e14*/ a[0] * b[4]
/*e14*/ a[5] * b[2] /*e24*/ a[3] * b[0] /*e24*/ -a[2] * b[5]
/*e24*/ -a[0] * b[3] /*e24*/ a[4] * b[2] /*e34*/ a[3] * b[1]
/*e34*/ -a[2] * b[4] /*e34*/ -a[1] * b[3] /*e34*/ a[5] * b[0]
/*e1234*/ -a[4] * b[1] /*e1234*/ a[3] * b[2] /*e1234*/ a[2] * b[3]
/*e1234*/ -a[1] * b[4] /*e1234*/ a[0] * b[5] /*e1234*/
```

Product of Two Bases

basis blades combine using xor

```
1  template<short x>    struct blade{
2                  static const short value = x;
3  };
4
5  template<typename a, typename b>
6  struct geometric_product{
7        using type = blade<a::value ^ b::value>;
8  };
```

Round and Flat Multivectors (and more)

Graphic Symbol	Geometric State	Grade	Algebraic Form	Abbr.
	Point	1	$p = n_o + \boldsymbol{a} + \frac{1}{2}\boldsymbol{a}^2 n_\infty$	`Pnt`
	Point Pair	2	$\tau = p_a \wedge p_b$	`Par`
	Circle	3	$\kappa = p_a \wedge p_b \wedge p_c$	`Cir`
	Sphere	4	$\Sigma = p_a \wedge p_b \wedge p_c \wedge p_d$	`Sph`
	Flat Point	2	$\Phi = p \wedge n_\infty$	`Flp`
	Line	3	$\Lambda = p_a \wedge p_b \wedge n_\infty$	`Lin`
	Dual Line	2	$\lambda = \boldsymbol{B} + \boldsymbol{d} n_\infty$	`Dll`
	Plane	4	$\Pi = p_a \wedge p_b \wedge p_c \wedge n_\infty$	`Pln`
	Dual Plane	1	$\pi = \boldsymbol{v} + \delta n_\infty$	`Dlp`
	Minkowski Plane	2	$E = n_o \wedge n_\infty$	`Mnk`

Basic Rounds and Flats in 5D Conformal Geometric Algebra and their Algebraic Constructions. Bold symbols represent Euclidean elements.

Conformal Versors Completely Represent All Euclidean Transformations

Graphic Symbol	Geometric State	Grade(s)	Algebraic Form	Abbr.
	Rotor	0, 2	$\mathcal{R} = e^{-\frac{\theta}{2}\boldsymbol{I}} = cos\frac{\theta}{2} - sin\frac{\theta}{2}\boldsymbol{I}$	`Rot`
	Translator	0, 2	$\mathcal{T} = e^{\frac{\boldsymbol{d}}{2}\infty} = 1 - \frac{\boldsymbol{d}}{2}\infty$	`Trs`
	Motor	0, 2, 4	$\mathcal{M} = e^{\boldsymbol{B}+\boldsymbol{d}\infty}$	`Mot`
	Dilator	0, 2	$\mathcal{D} = e^{\frac{\lambda}{2}E} = cosh\frac{\lambda}{2} + sinh\frac{\lambda}{2}E$	`Dil`
	Boost	0, 2	$\mathcal{B} = e^{o\boldsymbol{t}} = 1 + o\boldsymbol{t}$	`Trv`

Versor Enables Working in any Dimension or Metric

Instantiating an 3-Dimensional Euclidean Space

```
1 //.........................<p>, field>
2 using ega = algebra<metric<3>, double>;
```

Versor Enables Working in any Dimension or Metric

Instantiating an 4-Dimensional Spacetime

```
1 //.........................<p,q>, field>
2 using sta = algebra<metric<1,3>, double>;
```

Versor Enables Working in any Dimension or Metric

Instantiating a 5-Dimensional Conformal Space

```
1 //.........................<p,q,conf>, field>
2 using cga = algebra<metric<4,1,true>, double>;
```

Versor Enables Working in any Dimension or Metric

Instantiating an 6-Dimensional Twistor Space (quantum gravity!)

```
1 //.............................<p,q,conf>, field>
2 using twistor = algebra<metric<2,4,true>, double>;
```

- Is there an even more generic way to build compile-time algebras?

3 Elements of Programming

...

(,)

(,...)

*incomplete

A C++ syntax of point-free programming

...	stream
(,)	pair
(,...)	fold

Universal Categories (maybe? if not **why not**)

... firstness

(,) secondness

(,...) thirdness

*see Kumiko Tanaka-Ishii's *Semiotics of Programming*

... a search for the most **general**, most **terse**, most **expressive** syntax of structure

Arrows are a **secondness** for building computations

- John Hughes, *Generalising Monads to Arrows*, 2000
- Useful for thinking of computations in terms of **streams**
- Can be used to organize template compilation
- Helps us communicate programming logic
- There is also an Arrow *calculus* if you like ...

Arrows are **processes** built from 3 combinators

combinator	description
`arr`	**lifts** a process
`pipe`	**connects** arrows
`first`	**selects** stream

The Product of Two Lists by writing `head` and `tail`

Distributive property (mainloop):
$(a+b+c)(d+e+f) = a(d+e+f)+b(d+e+f)+c(d+e+f)$

```
procedure Product(A, B)
    if A=∅ then
        return ∅
    else
        first ← SubProduct(A::HEAD, B)
        rest ← Product(A::TAIL,B)
        return cat(first, rest)
```

Disributing over One List

Distributive property (subloop): $a(d + e + f) = ad + ae + af$

```
procedure SubProduct(a, B)
    if B=∅ then
        return ∅
    else
        sign ← sign(a, B::HEAD)
        first ← evaluate(a, B::HEAD)
        rest ← SubProduct(a, B::TAIL)
        return cat(first, rest)
```

We could use the hana library

lifting a metafunction

```
1  //Lift product
2  constexpr auto geometric_product =
3  metafunction<vsr::geometric_product>;
4  //a basis . . .
5  constexpr auto vec = basis<1,2,4>;
6  //compile-time distributed multiplication
7  constexpr auto result = ap(
8    lift<Tuple>(geometric_product), vec, vec );
```

But I Need to back **trace** the evaluation

- Maybe I want **ARROWS** to keep track of which indices from which input types contribute to which indices of which return types.
- exec_list< computation_instruction<A, B::HEAD, idxA, idxB> >

As a reminder we want to do this at compile-time because it makes fast code and all sorts of researchers use it.
Also its good for you.

Arrows help with **tacit**, **point-free** programming

Distributing a metafunction over two lists:

```
1  template<class A, class B> struct pair{};
2  template<class A, class B> struct all_pairs{};
3
4  template<typename ... xs, typename ... ys>
5  struct all_pairs< list<xs...>, list<ys...> >{
6    using type = list<
7      typename partial< pair, xs >::
8        template eval< ys ...> // expands ys
9      ... >; // expands xs
10 };
```

What is this `partial` function?

arr: $(A \times B \to C) \to (B \to C^A)$

arr curries n-ary functions into unary functions

```
1  template<
2    template<class...> class F, //<-- metafunction
3    typename ... a>  //<-- n-1 arguments
4  struct arr{
5
6          using type = arr<F,a...>;
7
8          template<typename x>
9          using eval = typename F<a...,x>::eval;
10 };
```

arr: $(a \times b \to c) \to (arrow\ b\ c)$

arr can be used to apply any function over a list

```
1  using pair_with_zero = arr< pair , int_<0> >;
2
3  template< typename ... xs >
4  using pair_me = list<
5          typename pair_with_zero::
6            template eval<xs>...
7          >;
8
9  using paired = pair_me<int_<1>,int_<2>,int_<3>>;
```

list: (pair:< 0 , 1 >) (pair:< 0 , 2 >) (pair:< 0 , 3 >)

stream : $(arrow\ \ b\ \ c \times [b]) \rightarrow [c]$

stream will take any arrow process and map it over a list

```
1 ///process is a type with Arrow concept
2 template<typename process, typename ... xs>
3 using stream = list<
4         typename process::template eval<xs>...
5         >;
```

OK, so

a list of streams

```
1  template<class A, class B> struct all_pairs{};
2  template<typename ... xs, typename ... ys>
3  struct all_pairs< list<xs...>, list<ys...> >{
4    using type = list<
5                   stream< arr< pair, xs >, ys...>
6                     ...>;
7  };
```

list: (pair:< 0 , 5 >) (pair:< 0 , 6 >) (pair:< 0 , 7 >)
list: (pair:< 1 , 5 >) (pair:< 1 , 6 >) (pair:< 1 , 7 >)
list: (pair:< 2 , 5 >) (pair:< 2 , 6 >) (pair:< 2 , 7 >)

pipe: $((arrow\ b\ c) \times (arrow\ c\ d)) \rightarrow (arrow\ b\ d)$

pipe feeds the output of one arrow into the input of another

```
1  template< class PA, class PB > //<-- two arrows
2  struct pipe{
3
4    using processA = PA;
5    using processB = PB;
6
7    template<typename x> using eval =
8      typename processB::template eval<
9        typename processA::template eval<x>
10     >;
11 };
```

`first`: $(arrow\ b\ c) \rightarrow (arrow\ (b, d)\ (c, d))$

`first` applies an arrow to the first part of a pair

```
1  template<class PA> //<-- an arrow
2  struct first{
3
4    using process = PA;
5
6    template<typename x> using eval = //<-- a pair
7      pair<
8        typename process::
9          template eval<typename x::first>,
10         typename x::second //<-- pass through
11     >;
12 };
```

A Simple Example of Arrows in Action

pipe an **arrow** to a **first**

```
1 //lift a pair<> by partially applying it
2 using pair_with_zero = arr<pair,int_<0>>;
3 //lift a successor function
4 using add_one = arr<succ>;
5 //pipe together
6 using proc = pipe<pair_with_zero, first<add_one>>;
7 //apply process to a stream
8 using stream< proc,int_<1>,int_<2>,int_<3>>;
```

list: (pair:< 1 , 1 >) (pair:< 1 , 2 >) (pair:< 1 , 3 >)

The succ function

succ

```
1  template<int N>
2  struct int_{
3    using type = int_<N>;
4    using one = int_<1>;
5    static const int eval = N;
6  };
7
8  template<typename seq>
9  struct succ{
10   using eval = decltype(
11         seq() + typename seq::one());
12   using type = succ<seq>;
13 };
```

splitand:$(arrow\ b\ c \times arrow\ b\ d) \rightarrow arrow\ b\ (c, d)$

splitand sends one input to two processes

```
1  ///PA &&& PB
2  template< class PA, class PB >
3  struct splitand{
4          using processA = PA;
5          using processB = PB;
6    template<typename x> using eval = pair<
7          typename processA::template eval<x>,
8          typename processB::template eval<x> >;
9  };
```

parallel:$(arrow\ b\ c \times arrow\ b\ d) \rightarrow arrow\ b\ (c, d)$

parallel sends an input to two processes

```
1  ///PA *** PB
2  template< class PA, class PB >
3  struct parallel{
4          using processA = PA;
5          using processB = PB;
6    template<typename x> using eval = pair<
7          typename processA::template eval<x::first >,
8          typename processB::template eval<x::second>
9    >;
10 };
```

$(,\ldots)$

Let's take stock of some patterns

- Lists (...) (of arguments, of types, etc)
- Pairs (,) (of functions, of results, etc)

How can these combine to make a **third** computational strategy?
(because we're not fully point-free yet really)

Fold-Expressions in C++17!

Takes a binary operator like + and nests it

```
1 template<typename ... xs>
2 using fold = decltype( (xs() + ...) );
3
4 using result = fold<int_<1>, int_<1>, int_<1> >;
```

3

Fold can be used to make a simple **filter**

+ operator on list<xs...>, list<ys...> returns list<xs...,ys...>

```
1 //, ... uses right fold
2 template<typename process, typename ... xs>
3 using filter = decltype(
4         (typename list<
5           typename process::
6                 template eval<xs>>::type() + ...) );
```

Fold can be used to make a simple **filter**

a remove func lifted into an arrow

```
1
2  template< typename T, typename S>
3  struct remove{
4          using eval =
5          typename either<
6                  std::is_same<T,S>::value,
7              nothing, S >::type;
8  };
9
10 using result =
11   filter< arr<remove, byte<1>>,
12       byte<1>, byte<2>, byte<1>, byte<3>>;
```

2 3

What about a + operator for our **Pairs**?

perhaps a + operator on pairs calculates pairwise +

```
1  template<class a, class b, class c, class d>
2  constexpr auto operator +(
3          pair<a,b>&& pa, pair<c,d>&& pb){
4   return typename pair<
5          decltype(a() + c()),
6          decltype(b() + d())
7   >::type();
8  }
```

A Generic **Sink** should just Fold a Stream...

folding a pair with sink

```
1  template<typename process, typename ... xs>
2  using sink = decltype(
3    (typename process::template
4                    eval<xs>() + ...)
5          );
6
7  using proc = sink<
8          arr<pair, int_<1>>, int_<3>, int_<5> >;
```

(pair:< 2 , 8 >)

But there is something missing

1. We can only apply our simple arrows before the evaluated result is folded, not *while* it is being folded
   - Hmmm, if could we lift arrows to arrows...
2. No way to keep track of computation results
   - Maybe just another type of fold will do the trick?

Nesting Pairs To Scan a Fold

A different operator: every recursion, value is stored

```
1  template<class a, class b, class c, class d>
2  constexpr auto operator <<( //another operator
3          pair<a,b>&& pa, pair<c,d>&& pb){
4           return typename pair<
5                   decltype(a() + c()),
6                   pair<a,b>              //extension
7           >::type();
8  }
9
10 template<typename process, typename ... xs>
11 using sinkL = decltype(
12   (... <= typename process::template eval<xs>())
13 ); //a left fold
```

Stream of ints → Scanned fold

- sinkL (arr(\x → <x,void>)) (1,1,1)
  - (pair:< 3 , (pair:< 2 , (pair:< 1 , >) >) >)
- sinkR (arr(\x→ <void,x>)) (1,1,1)
  - (pair:< (pair:< (pair:< , 1 >) , 2 >) , 3 >)
- its an impulse train . . .

Still not close to an Index Sort

the things we tmp'ers do to insert-sort at compile time

```
1 insert
2 insert_impl<true|false>
3 sorting_index
4 sorting_index_already_exists
5 sorting_index_end_check<true|false>
6 sorting_index_impl<true_false>
7 ... and limit cases <>
```

We want to be able to fold arrows themselves, not just their evaluations

- we could overload the + operator for pipes

consider unfolding a pair into a list

```
1  using unf1 = first<arr<id>>;
2  using unf2 = pipe<unf1, second<unf1> >;
3  using unf3 = pipe<unf2, second<unf1> >;
4  ....
5  using unfN = pipe<unfN-1, second<arr<make_void>>>;
6
7  //
8  using pattern = pipe<unf1, second<unf1>>;
9  template<typename x>
10 using source = ?// no parameter pack to unfold
```

Pipe + Pipe = Pipe

pipe a piping (to be folded in with a stream, to later unfold it...)

```
1  template< class PA, class PB, class PC, class PD>
2  constexpr auto operator +
3          (pipe<PA,PB> pa, pipe_<PC,PD> pb)
4  {
5           return pipe<decltype(pa), decltype(pb) >();
6  };
```

Summary of This Simple Construction So Far

- Products (pairs) and Arrows Seem Destined To Work Together
- Both are good for folding and unfolding
- We can fold Arrows too (we think)

Questions:

- What is missing from this Universal Construction?
- Does Folding Pairs provide any compile-time advantage / disadvantage?
- Could we build a spectral basis for Algebras? Keep track of indices more naturally?
- Is this recursive style really **pointless** (in the *right* way)?

The Tacit Point of it all...

the demonstration of a pointless style towards the formulation of algebraic and geometric points