---

title: Pro Git author: Scott Chacon rights: Creative Commons Non-Commercial Share Alike 3.0 language: en-US

A formalism for data systems

This is considerations for formalizing some ideas found in modern data systems.

The code examples are done using mathematical notations and Haskell functions, which should be clear enough in most of the case.

General principles

Distribution of computation Representation is not observable

Setting

We consider operations over some sets of values, in which we want to follow some basic principles. All the operations operate either on values or sets of values, and data processing consists in writing some functions that transform sets into other sets, or into values. Here are a couple of assumptions that we will work on: - the exact representation of the data is not observable. This is often not verified in such systems. - the data is considered static. It does not change over time.

The first point has profound implications, in that datasets do not have first or last elements. They do not behave like regular collections, which are usually sequential in nature. However, this provides a lot of advantages in the case of big data processing.

We consider here a simple formalism: call $\mathcal{U}$ the set of all the values, called the *universe*. We assume that the universe is countable and separable. Subsets of the set of all values can be tagged with *types*. A type is essentially a subset of all the values. Types are mostly useful when writing programs and for clarity, not so much for our mathematical derivations.

A dataset is a *finite measure* over the set of values. $d : U \to \mathbb{N}$ with the restriction:

$$\sum_{x \in \mathcal{U}} d(x) < \infty$$

Note Basic set arithmetic ensures that the function defined above is a measure in the sense of distributions. In a sense, we consider here some easy subset of distributions.

Basic operations over distributions:

Union The union of two datasets is straightforward:

```
union :: Dataset a -> Dataset a -> Dataset a
(union d1 d2) x = d1 x + d2 x
```

This operation is of course symmetric, associative, commutative and has a zero element (the empty set). This is the most important of all the general operations, because it conveys some useful structure over datasets (a monoid and a semi-group).

Dirac The dirac of a dataset is the extraction of a single value at a given point. The name `dirac` comes from the theory of measures, which itself comes from physics.

```
dirac :: a -> Dataset a
dirac x y = if x == y then 1 else 0
```

One trivial consequence is the fact that a dataset is the (finite) sum of diracs. This fact will have its importance later.

Mass The mass of a dataset is the count of all its points. The name sounds related to physics and we will see the intuition behind it later.

```
mass :: Dataset a -> Nat
```

Point transforms

With this structure, there is not much we can do so far, apart from querying the value of a dataset at various points. We are going to add many more operations now. One of the simplest and yet most effective ways to transform a dataset is to apply some operation to all its points.

We are going to add a twist to it though, relative to the distribution principle: if a dataset is splittable, we would like the transform to be splittable too:

```
f :: Dataset a -> Dataset b
f (d1 U d2) = (f d1) U (f d2)
```

It turns out that this adds a lot of structure to all the possible point transforms; in particular the transforms have to be automorphisms over the dataset monoid.

Proposition: neutral element:

```
f {} = {}
```

Proposition: characterization: All the point transforms can be uniquely characterized by their operations on elements. More precisely, by the point function:

```
point_f :: a -> [b]
```

(up to a permutation of the elements in the return list). TODO

The last proposition shows that the *only* valid transforms a fully described by what happens on each point, and that each point can only get mapped to a collection of other points. We can now fully qualify transforms about how many points they produce for each point: exactly one (mass-preserving), at most once (mass-shrinking) or arbitrary.

Mass-preserving transform A mass-preserving transform obeys the following law: forall d, `mass (f d) == mass d`

It is fully characterized by a function `point_f :: a -> b`. Conversely, any such function defines a mass-preserving transform. This is the functorial `fmap` that is familiar to functional programmers:

```
map :: (a -> b) -> Dataset a -> Dataset b
```

Shrinking transform A mass-shrinking transform (or shrinking transform in short) reduces the mass of a dataset: `forall d, mass (f d) <= mass d`. Similarly, it is fully characterized by its optional return on each point:

```
mapMaybe :: (a -> Maybe b) -> Dataset a -> Dataset b
```

Reductions

Reductions correspond to the action of taking a dataset and condensing its information into a single value. This is the second fundamental operation one can do on a dataset.

```
f :: Dataset a -> b
```

Again, we would like this transform to obey some distribution principle: reducing a dataset to a single value should be done independently from the structure of the dataset, and it should be run in parallel if the dataset is the union of subsets. This is expressed through the following definition:

Definition A reduction `f` is said to be monoidal, or universal, if there exists a function `+_f :: b -> b -> b` that obeys the following distributivity rule:

```
f (d1 union d2) = (f d1) +_f (f d2)
```

A monoidal reduction can be interpreted as carrying over through computations of the underlying structure of the dataset. It turns out that the property above is quite strong and imposes a lot of constraints on the definition of `+_f`:

Monoid structure The relation `(Im f, +_f)` describes a monoid over `Im f`. The neutral element of this monoid is `f {}`.

Characterization This morphism is fully described by the values of `f` over diracs and `{}`.

Unicity This monoidal law is unique. TODO

Composition This trivial property has important performance consequences: If `f1` and `f2` are both monoidal reductions, then `f1 &&& f2` is also a monoidal reduction.

Reductions - examples

Here are a couple of fundamental examples of reductions.

Joins

```
join :: Dataset (k, a) -> Dataset (k, b) -> Dataset (k, Maybe a, Maybe b)
```

It follows some distributivity law, as we would expect:

```
join (a1 U a2) b == (join a1 b) U (join a2 b)
join a b = join b a
mass (join a b) == max (mass (key a1 U key a2))
```

TODO: is it enough to defin the union through axioms?

Groups

A group corresponds to the notion that data points can somehow be associated together. It allows to express transforms on potentially many datasets all at once.

Definition: group A group is a function from datasets to datasets:

```
type Group k v = k -> Dataset v
```

Of course, this is a very high-level definition, and we will see how it works out in practice. A practical implementation of a group may be done as follow, thanks to Currification: a group is the type `k -> v -> Nat`, or through Currying: `(k, v) -> Nat`, which suggest an alternative representation for groups:

```
type Group k v = Dataset (k, v)
```

We will use one representation or the other according to the situation.

Reductions over groups. Define the following operator:

```
shuffle :: Reduce a b -> Group k a -> Dataset (k, b)
```

```
groupBy :: (v -> k) -> Dataset v -> Group k v
```

Distributivity of shuffle

This distributivity law shows the power of the universal reductions.

```
shuffle f (g1 `union` g2) = map (+_f) (join (shuffle f g1) (shuffle f g2))
```

Filter

```
filter :: (a -> Bool) -> Dataset a -> Dataset a
```

This is a simple example of contraction.

Ordered reductions

```
orderedReduce :: Ord a => Dataset (a, b) -> [b]
```

Canonical representation

```
shuffle count . groupBy id
```

Indexing and counting

Here are a couple of operations one can do to as substitutes for usual operations in SQL, using the basic transforms outlined above.

Substitutes for random operations