toward an algebraic understanding of physics analysis

Chris Pollard

some front matter:

- **I am not a mathematician or a category theorist but an interested (and fallible) physicist! I am certainly still learning this stuff. ;-)**

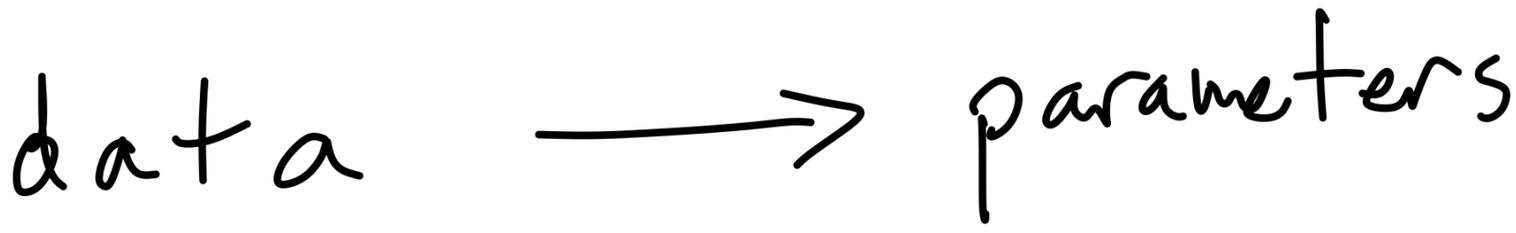
- **I also don't have a lot of experience explaining the concepts here: please ask questions as we go along if you get at all lost!**

- **it may seem like a whole lot of mumbo-jumbo to begin with, but I hope to convince you of the benefits thinking this way.**

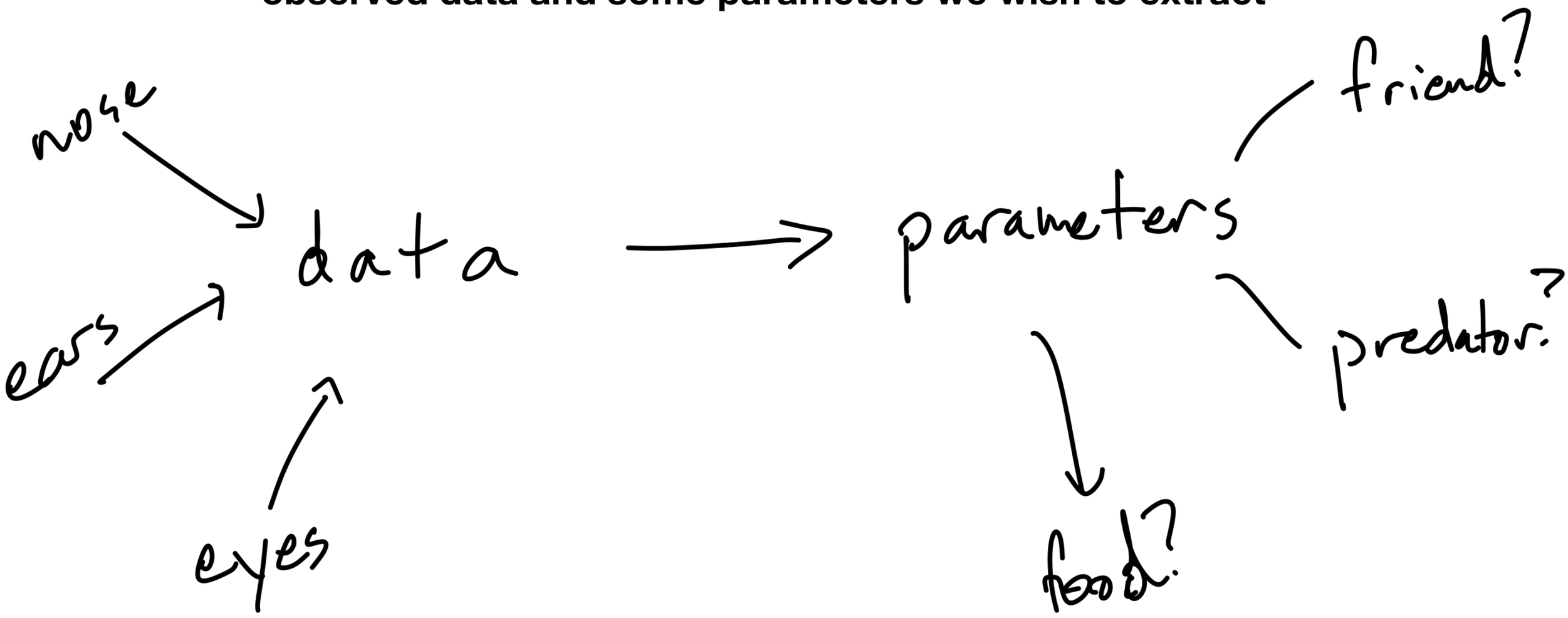
- **I cannot go into details and will do a lot of hand waving, but please let me know if I've piqued your interest!**

- **also apologies for all the words, but I thought people may want to go back and read some of the explanations.**

data analysis is about proving a relationship between observed data and some parameters we wish to extract



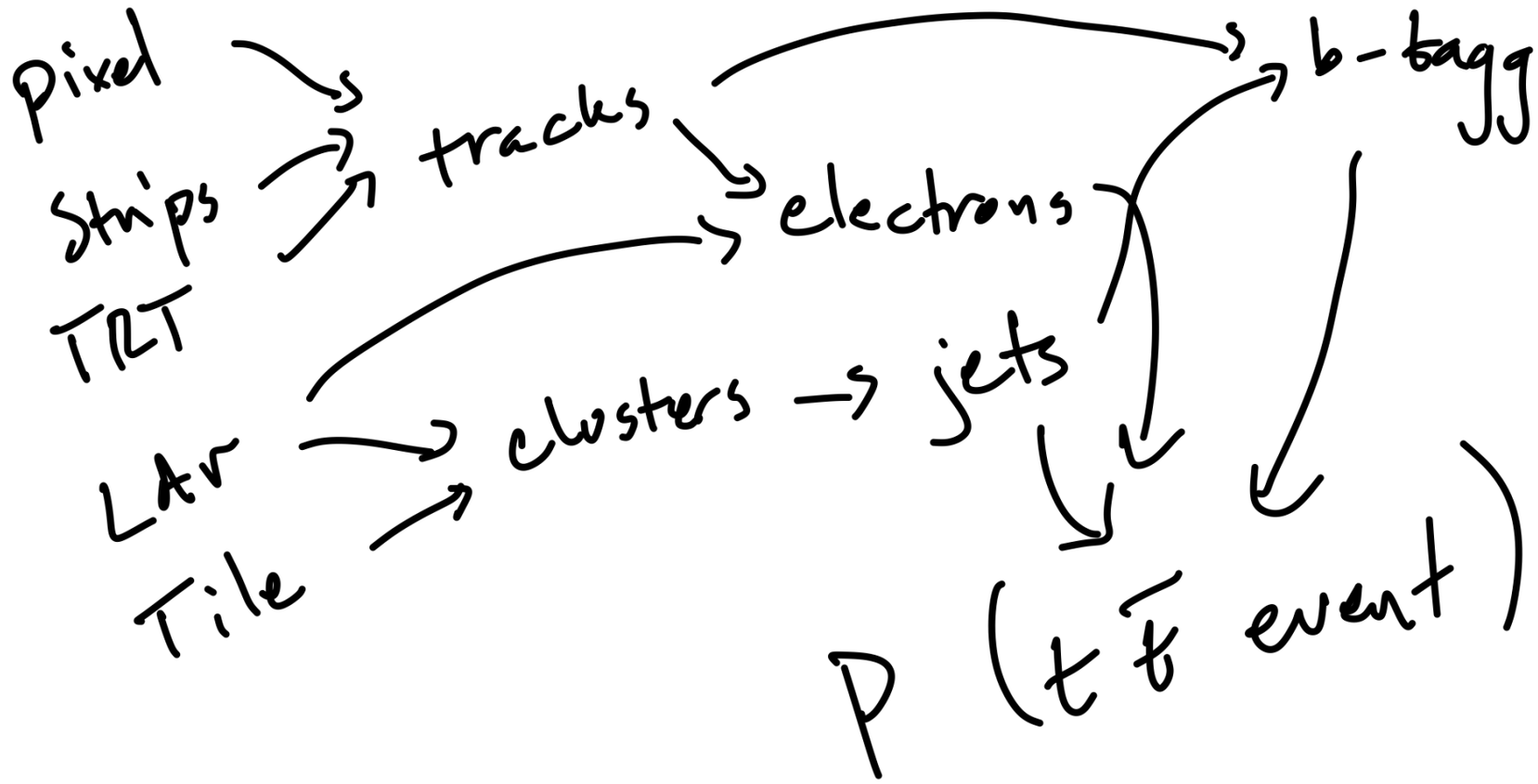
data analysis is about proving a relationship between observed data and some parameters we wish to extract



this is one way of thinking about human observation and modeling of our surroundings

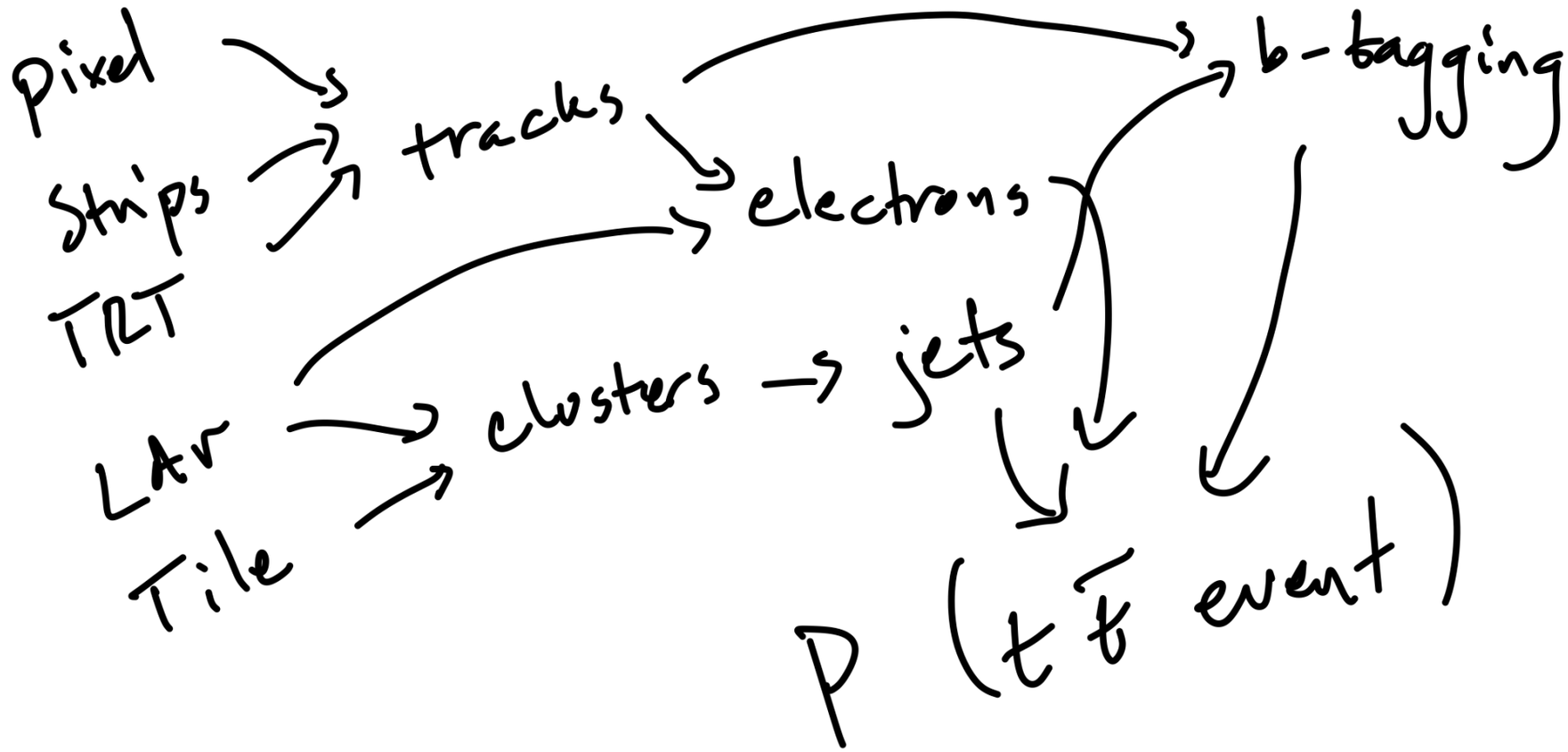
this relationship can actually be extremely complicated...

... in practice it's usually made up of many, many "sub"-relationships



this relationship can actually be extremely complicated...

... in practice it's usually made up of many, many "sub"-relationships



our brains can't really handle that many relationships at once

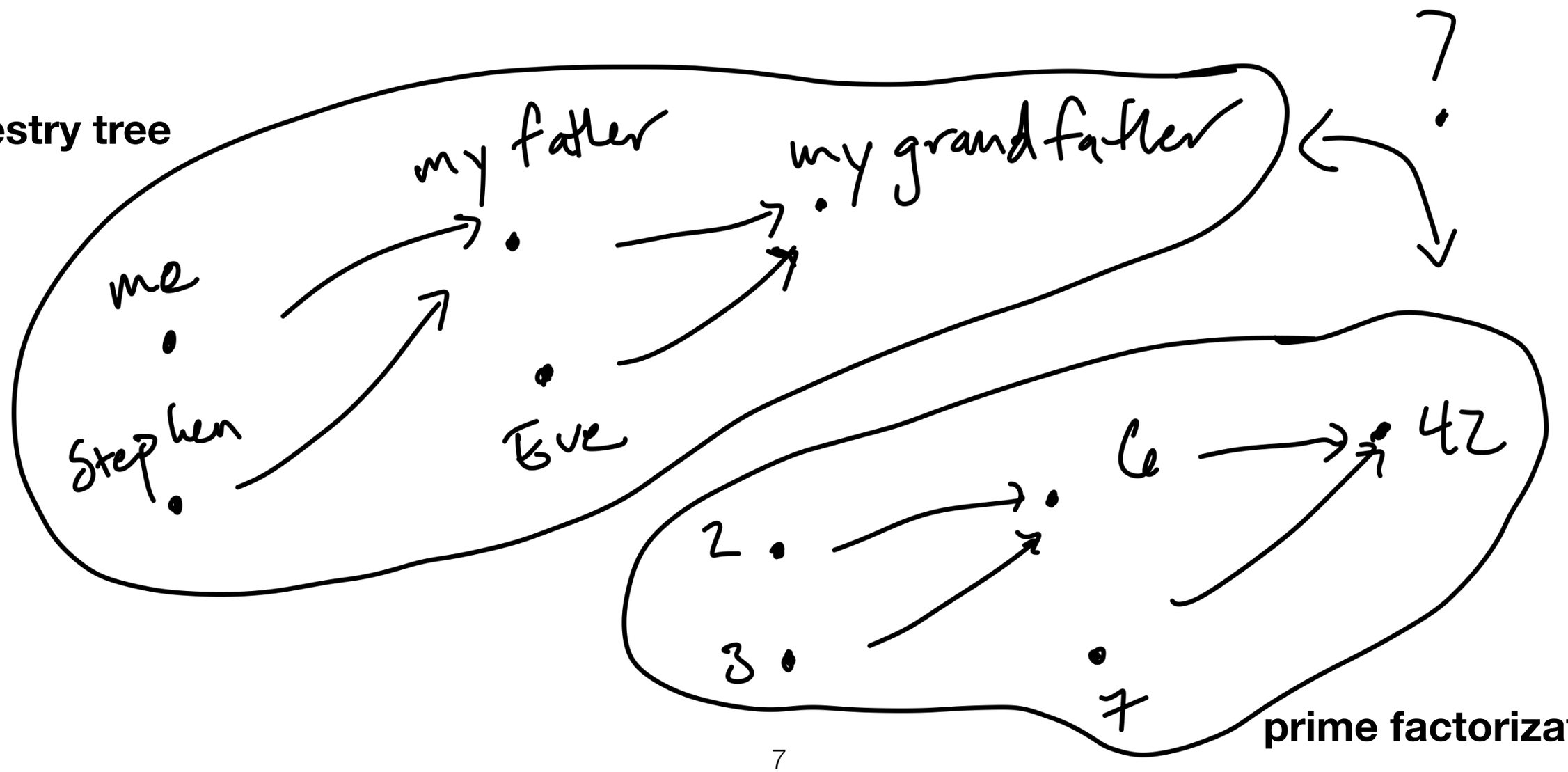
"divide-and-conquer" is almost always our go-to strategy for proving complex relationships:
breaking down then re-composing the problem is key!

category theory is (in my own words and opinion) an attempt
to reason formally about and study

abstract relationships
(throwing away unnecessary details)

how these relationships compose

and the **structure that results**
from these relationships and their composition



what is a category?

a collection of objects *A*, *B*, *C*, ...

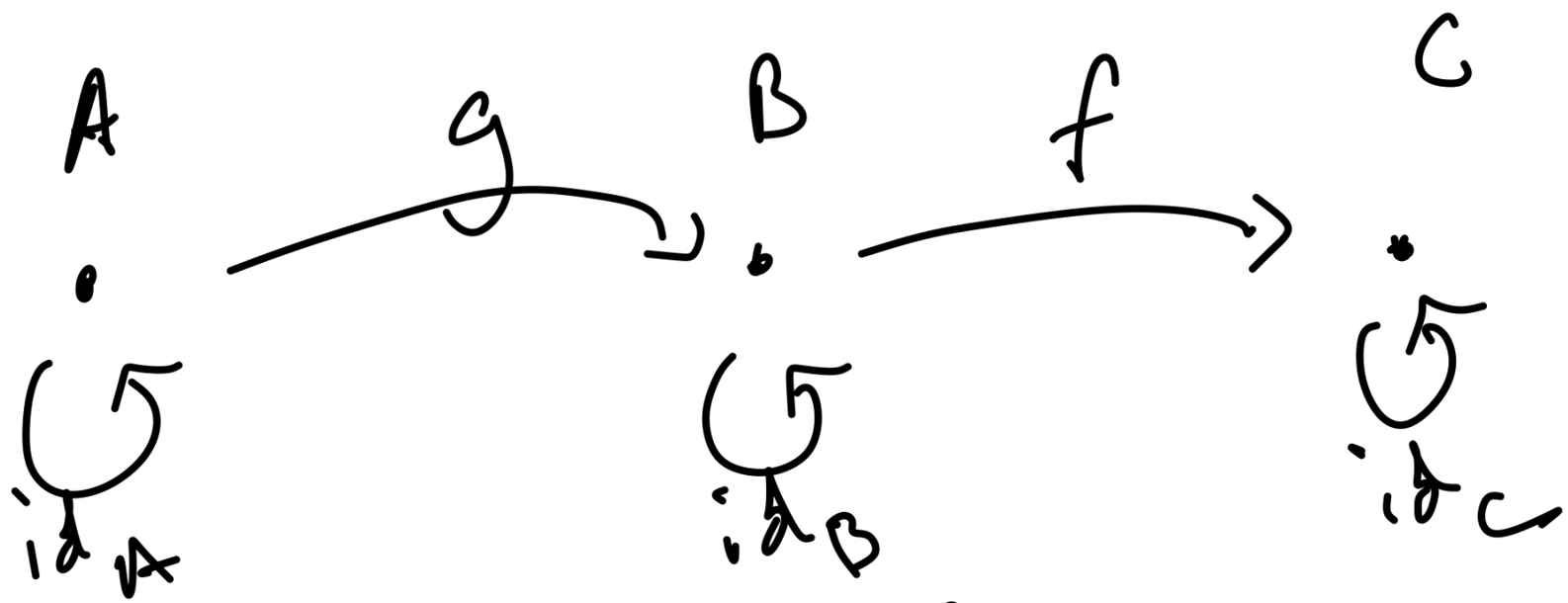
and a collection of morphisms (aka relationships) connecting objects *f*, *g*, *h*, ...

morphisms between compatible objects compose:

***f . g,* meaning "f after g"**

**their composition must also be a morphism in the category,
and composition is associative**

the identity morphism exists for each object



what is a category?

a collection of objects *A*, *B*, *C*, ...

and a collection of morphisms (aka relationships) connecting objects *f*, *g*, *h*, ...

morphisms between compatible objects compose:

f . g, meaning "f after g"

their composition must also be a morphism in the category,
and composition is associative

the identity morphism exists for each object

notice the similarity to a group:

a set with an associative binary operation on its members such that
the composition of two members is also a member
and with an identity element

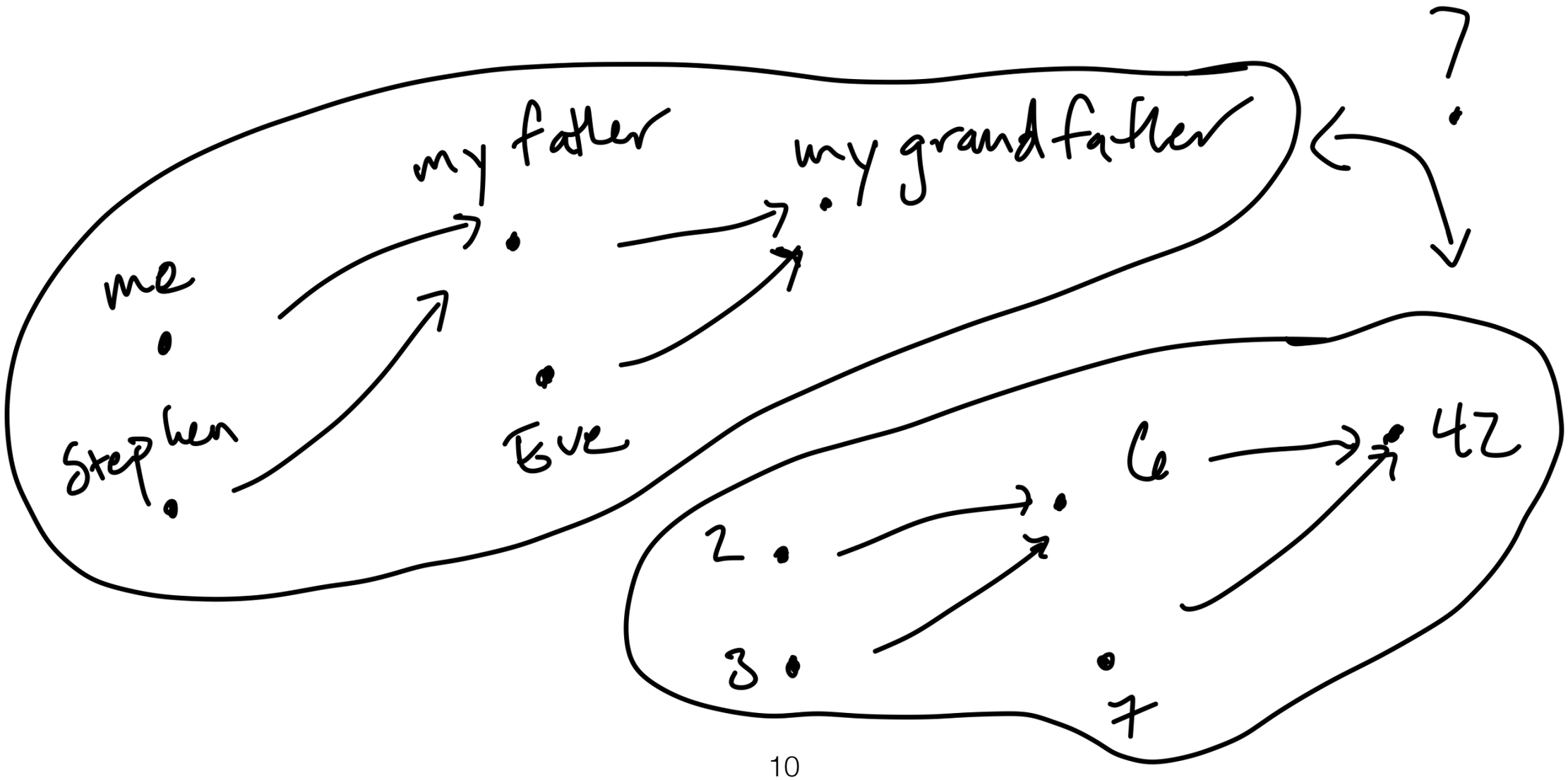
but missing inverses! (i.e. a monoid)

going back to the previous example: my family members can be cast as objects in a category, where morphisms represent ancestry

ancestry composes: an ancestor of my ancestor is also my ancestor

the same is true of prime factorization

what's clear is that these two categories have the same structure, neglecting the detailed meaning of the arrows and object labels

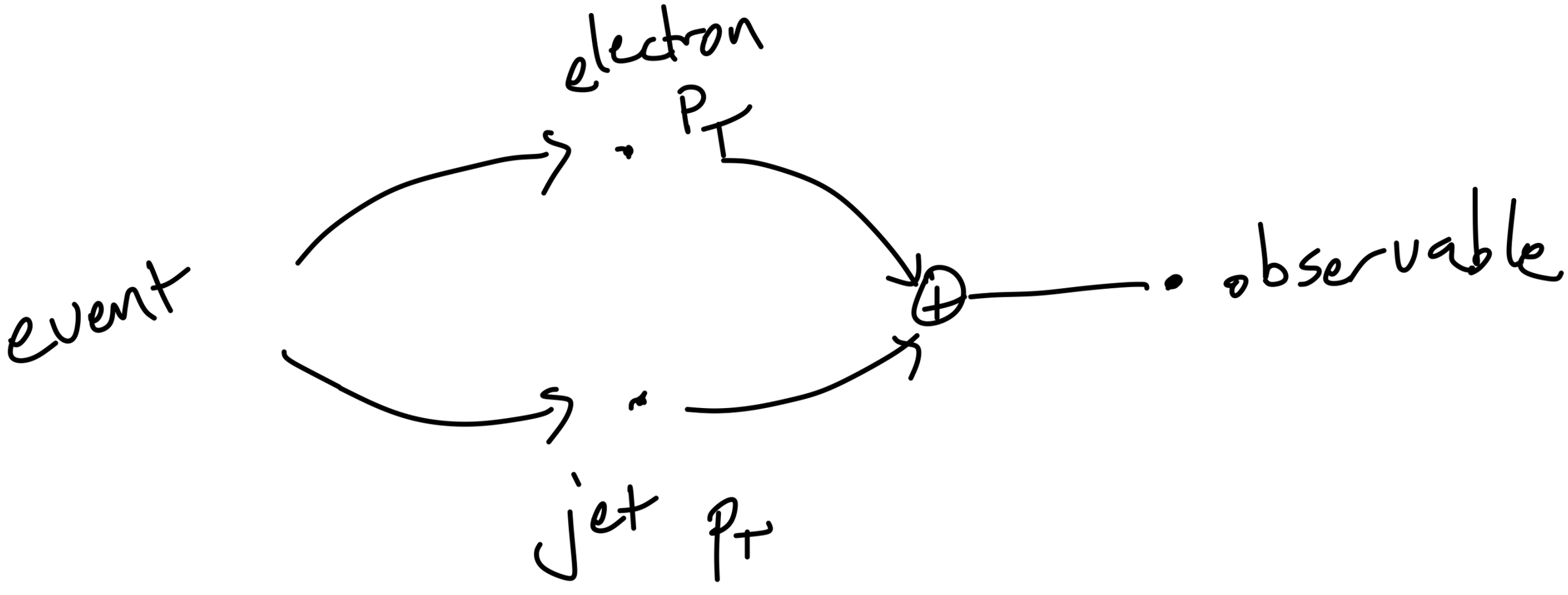


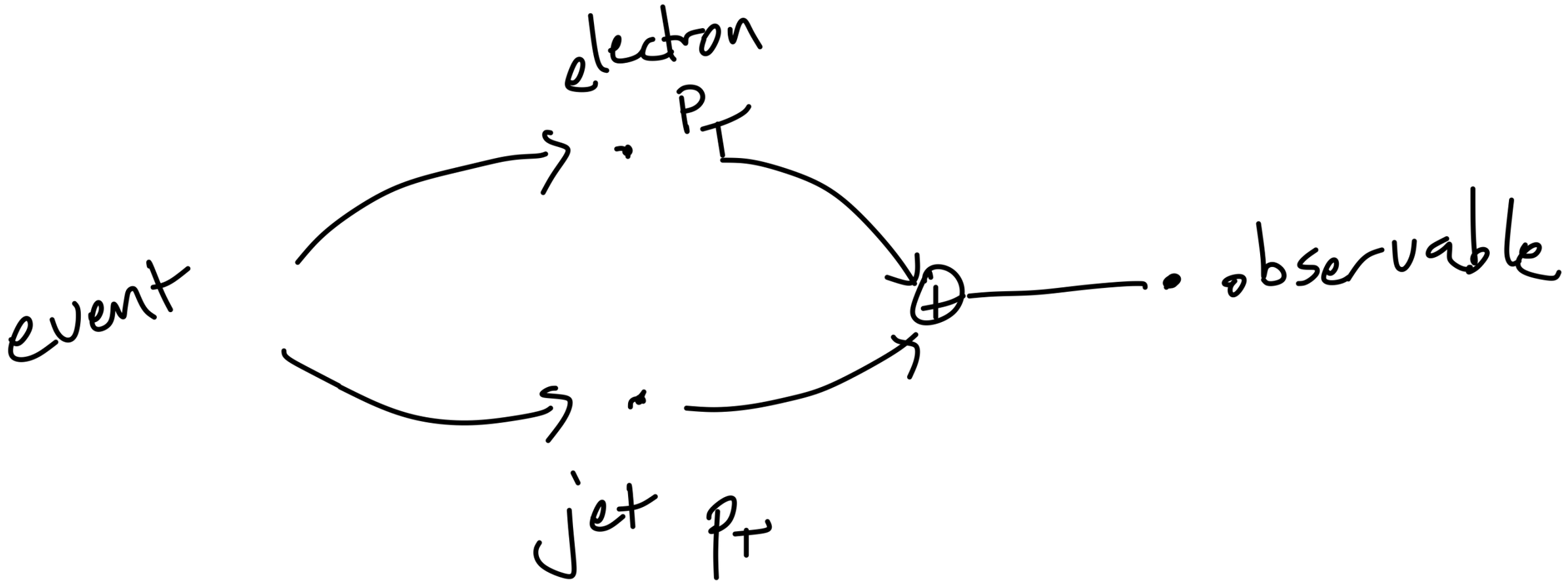
why is this interesting?

when we perform physics analyses,
we are attempting to prove a very complicated relationship between
e.g. energy deposits in the detector and some physical parameter.

the translation of this process to CPU commands tends to be fairly ad-hoc.

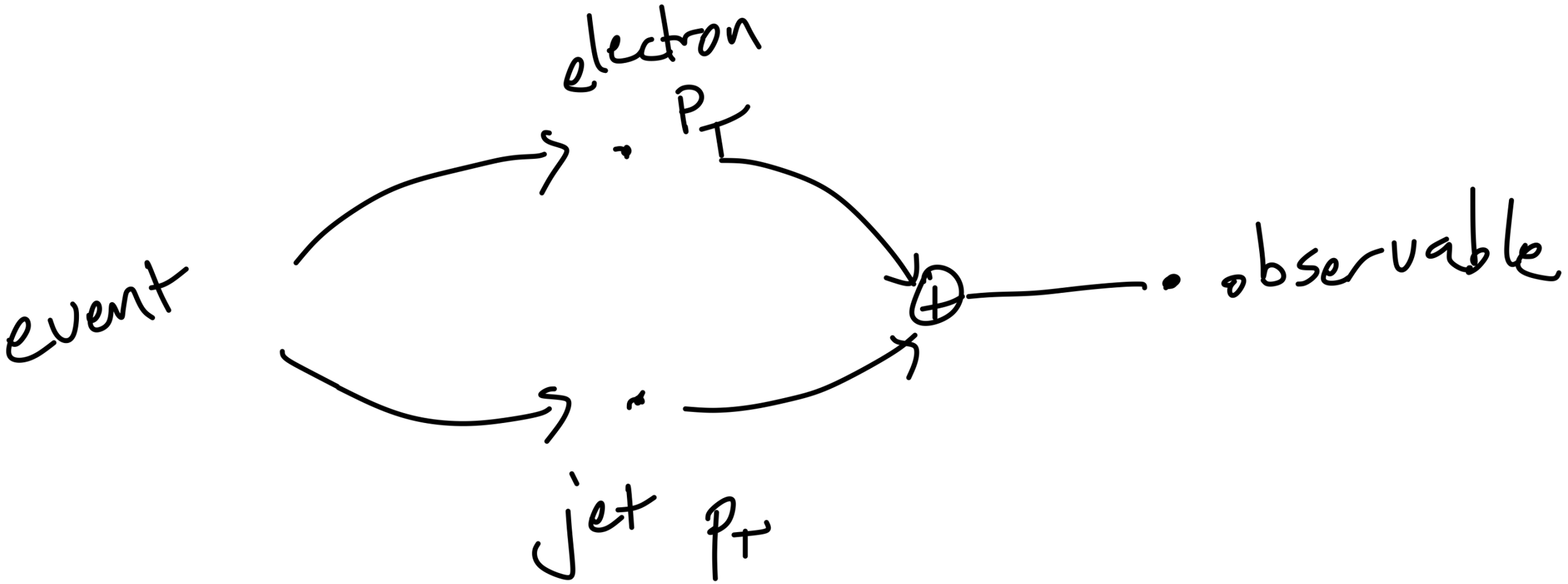
by breaking our proof down into small, understandable, composable chunks,
we can make this process much more precise
and gain important insights in so doing!





a procedure that can be represented as a graph like this one (a string diagram) can be realized in a class of category with a few constraints on its structure, called cartesian closed categories.

there is a famous correspondence ("Curry–Howard–Lambek correspondence") between cartesian closed categories <=> intuitionist logic <=> typed lambda calculus

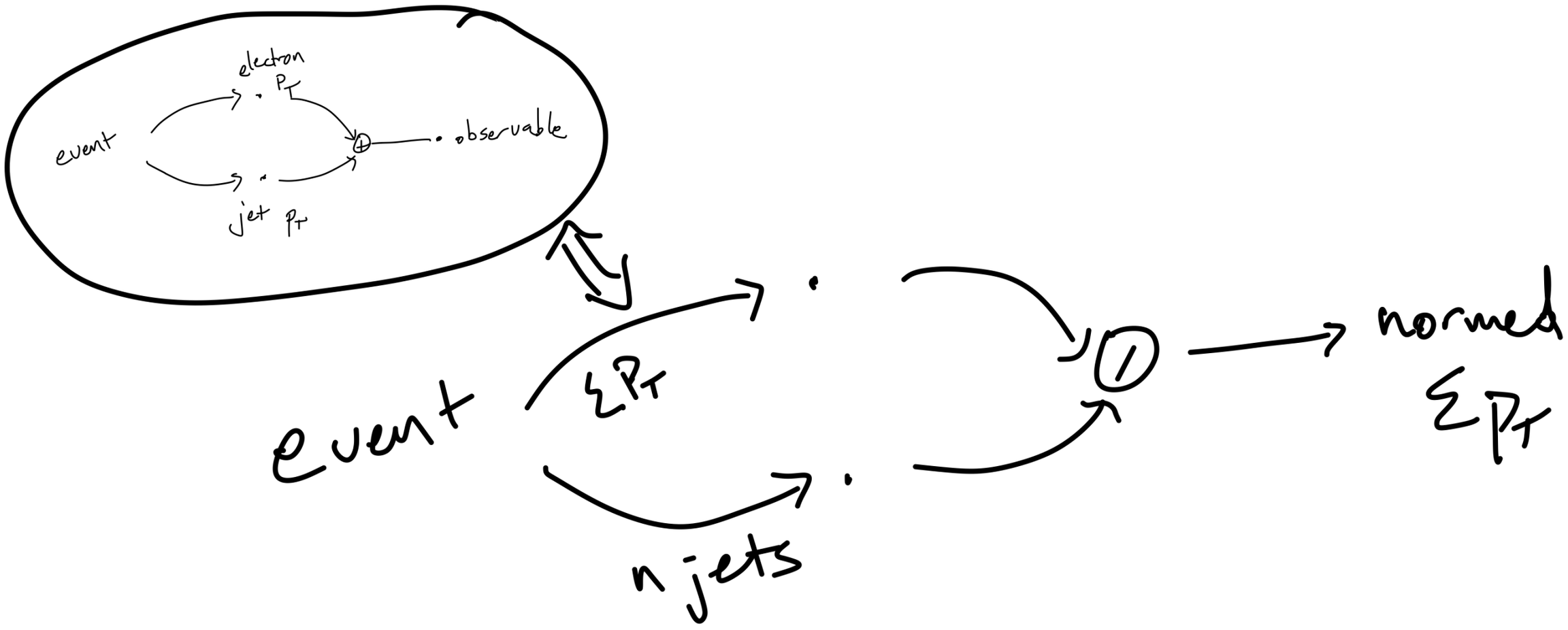


cartesian closed categories <=> intuitionist logic <=> typed lambda calculus

the Curry–Howard–Lambek correspondence shows that diagrams above can be translated to and evaluated as computer programs where

1) objects are labeled sets of values (types)
2) morphisms are functions between types

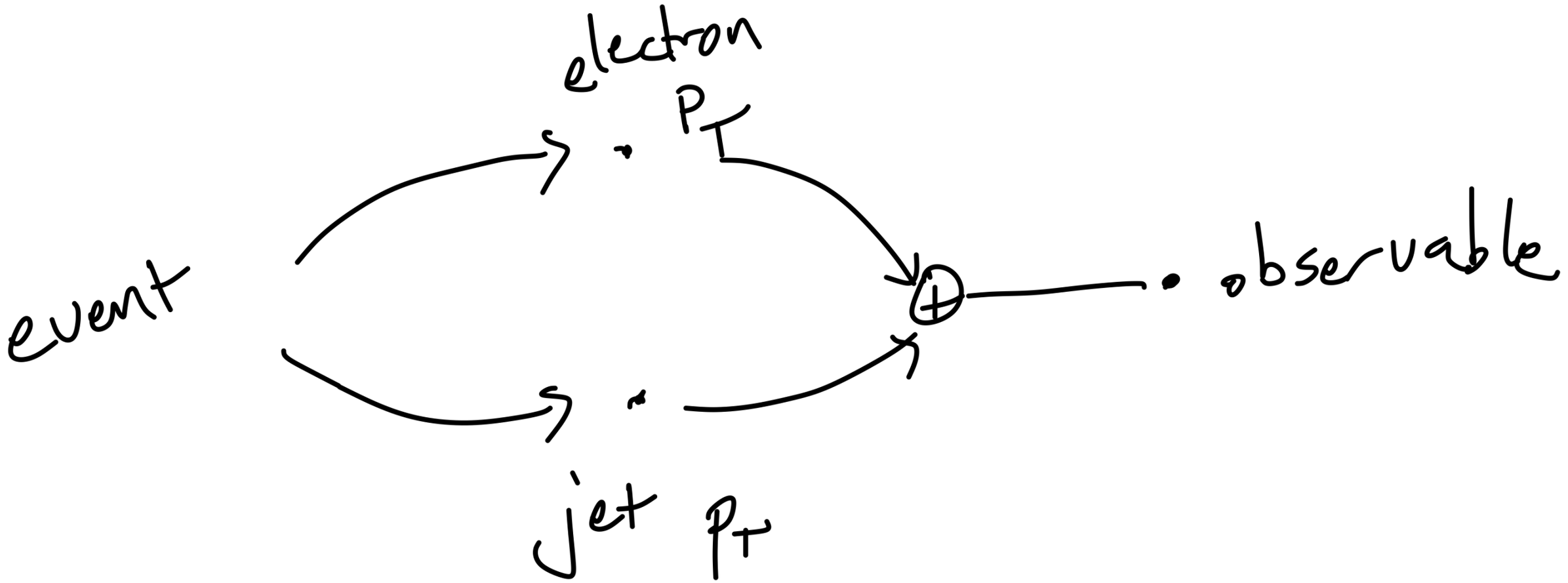
in other words, if we can draw a diagram like above describing a physical process, and if we can provide an implementation of each morphism, then we can translate such diagrams into a suitable programming language!



we should not forget that, because everything here is very carefully composable, physics processes that can be represented this way also compose.

i.e. this particular process may be embedded in a much larger one, or morphisms in this particular process may actually be compositions of more "fundamental" morphisms.

however, we can reason locally knowing that we can always insert our local understanding into the bigger picture!

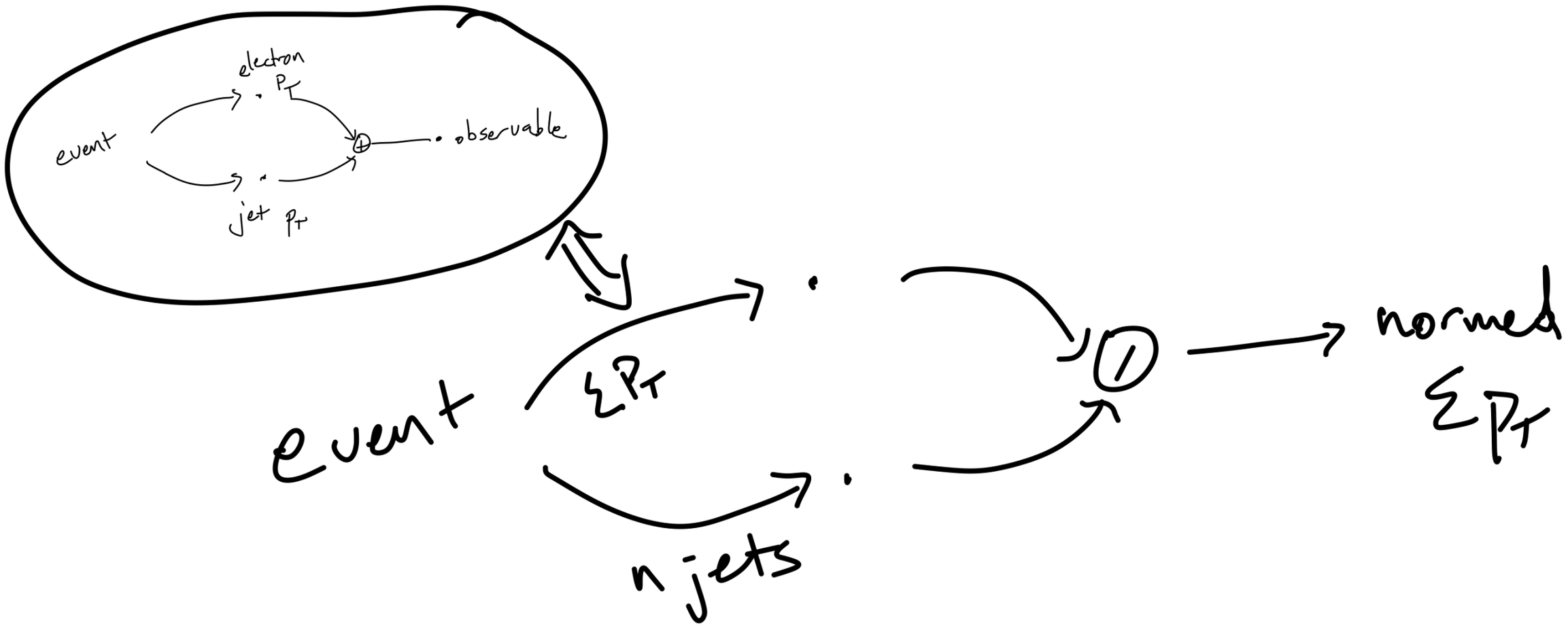


thankfully, others have already done the hard work: translating this onto the CPU

they even provide a nice, visual syntax!

translating to haskell syntax:

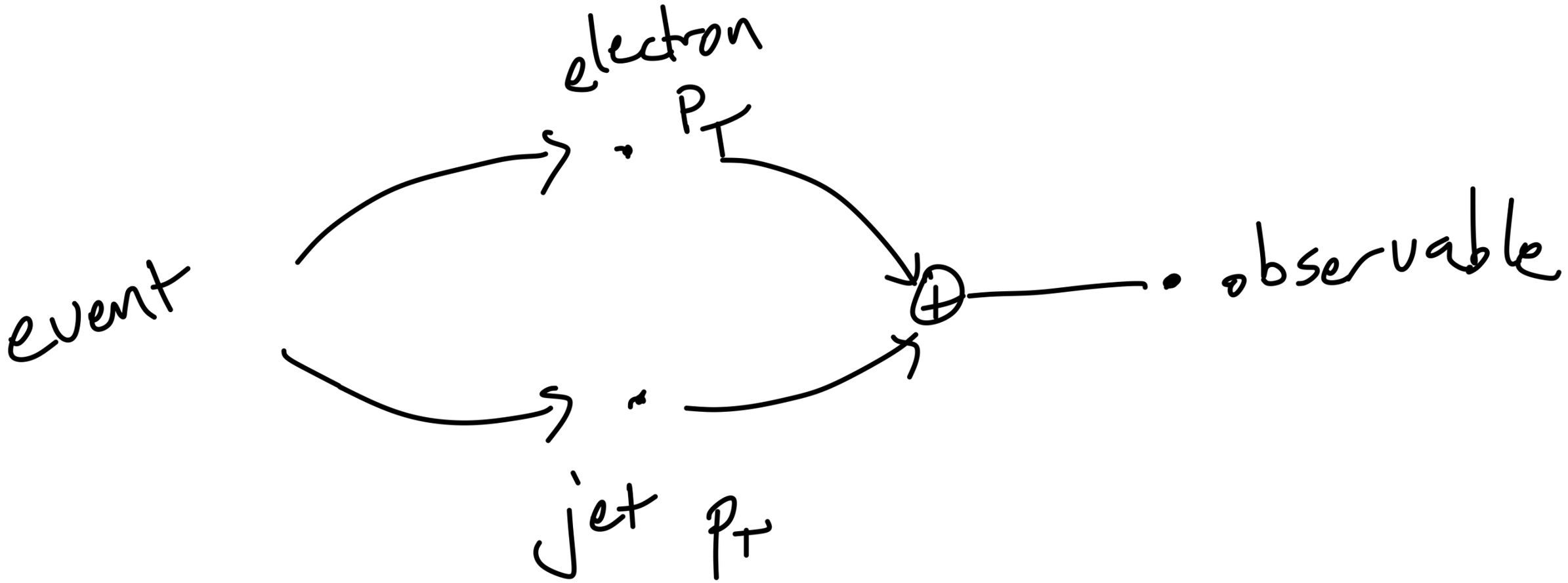
```
ptsum = proc event → do
  ept ← elPt ≺ event
  jpt ← jetPt ≺ event
  returnA ≺ ept + jpt
```



remember:
everything here composes

translating to haskell syntax:

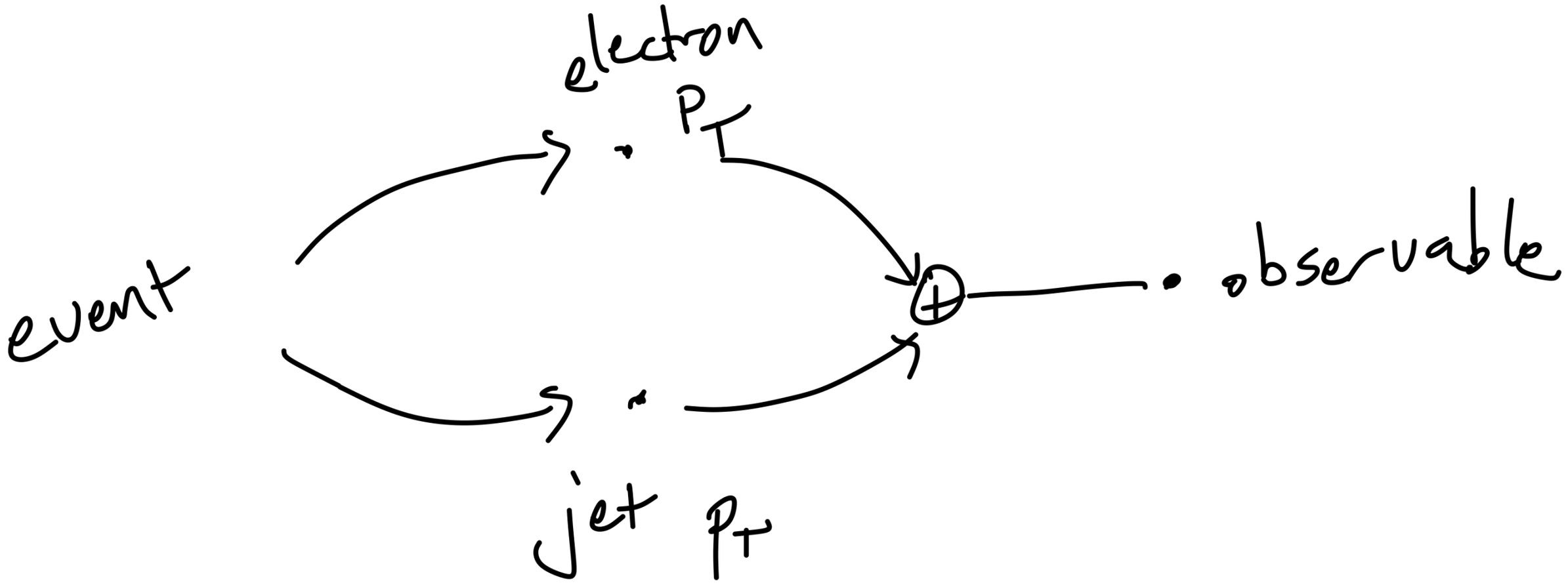
```
ptsumnorm = proc event → do
  ps ← ptsum ⤙ event
  nj ← njets ⤙ event
  returnA ⤙ ps / nj
```



so far we've just been working with "simple" functions on sets as morphisms

i.e. you give me an event
-> I know how to extract the electron pt
-> I know how to extract the jet pt
-> I can sum these two to get my final observable

this is already powerful, but we can take things to another level.



when I see a process like this many things pop into my head:

what assumptions are built into each relationship?

are there uncertainties related to each arrow?

how would I predict such an observable with e.g. MC?

well... are "simple" functions the only morphisms we know how to compose?

are "simple" functions the only morphisms we know how to compose?

no!

let's consider efficiency corrections ("scale factors", "SF") for reconstructing electrons

now instead of simply being a function from an event to the pT of an electron

Event -> pT

our electron pT function returns an additional *context*

Event -> (SF, pT)

all we need to figure out is how to compose morphisms like this...

```
f :: a -> (SF, b)
g :: b -> (SF, c)

(g . f) :: a -> (SF, c)
```

but we know how to do this!

- run the function *f* on the input *a*, resulting in a SF and *b*
- plug the resulting *b* into the function *g*, resulting in another SF and *c*
- multiply the SFs returned by *f* and *g*
- and return the combined SF along with the value *c*

```
f :: a -> (SF, b)
g :: b -> (SF, c)

(g . f) :: a -> (SF, c)
```

but we know how to do this!

- run the function *f* on the input *a*, resulting in a SF and *b*
- plug the resulting *b* into the function *g*, resulting in another SF and *c*
- multiply the SFs returned by *f* and *g*
- and return the combined SF along with the value *c*

strangely (or not?) this is a well-established structure in the haskell world:
if you have functions returning some additional information for which there is a well-defined multiplication (i.e. a monoid), there is an obvious way to compose these functions.

my morphism source code requires no changes;
the elPt and jetPt morphisms are free to return SF-decorated values, and a SF-decorated pt sum will be returned!

```
ptsum = proc event → do
  ept ← elPt ⤙ event
  jpt ← jetPt ⤙ event
  returnA ⤙ ept + jpt
```

this type of structure is called a "Writer" in the haskell community because each morphism is allowed to "write" some additional information to a "generalized log"

in this case the "log" was accumulated multiplicative scale factors, but it's easy to see that this is powerful in a more general way...

for instance, imagine one wants to keep track of the set of relevant assumptions that were applied for a particular event, e.g.

"1 TeV electron found; assume SFs from Z->ee extrapolate to this range"
-or-
"assume mJJ and jet substructure are uncorrelated"

it's straightforward to collect these and to validate they were consistent for each event passing through analysis code.

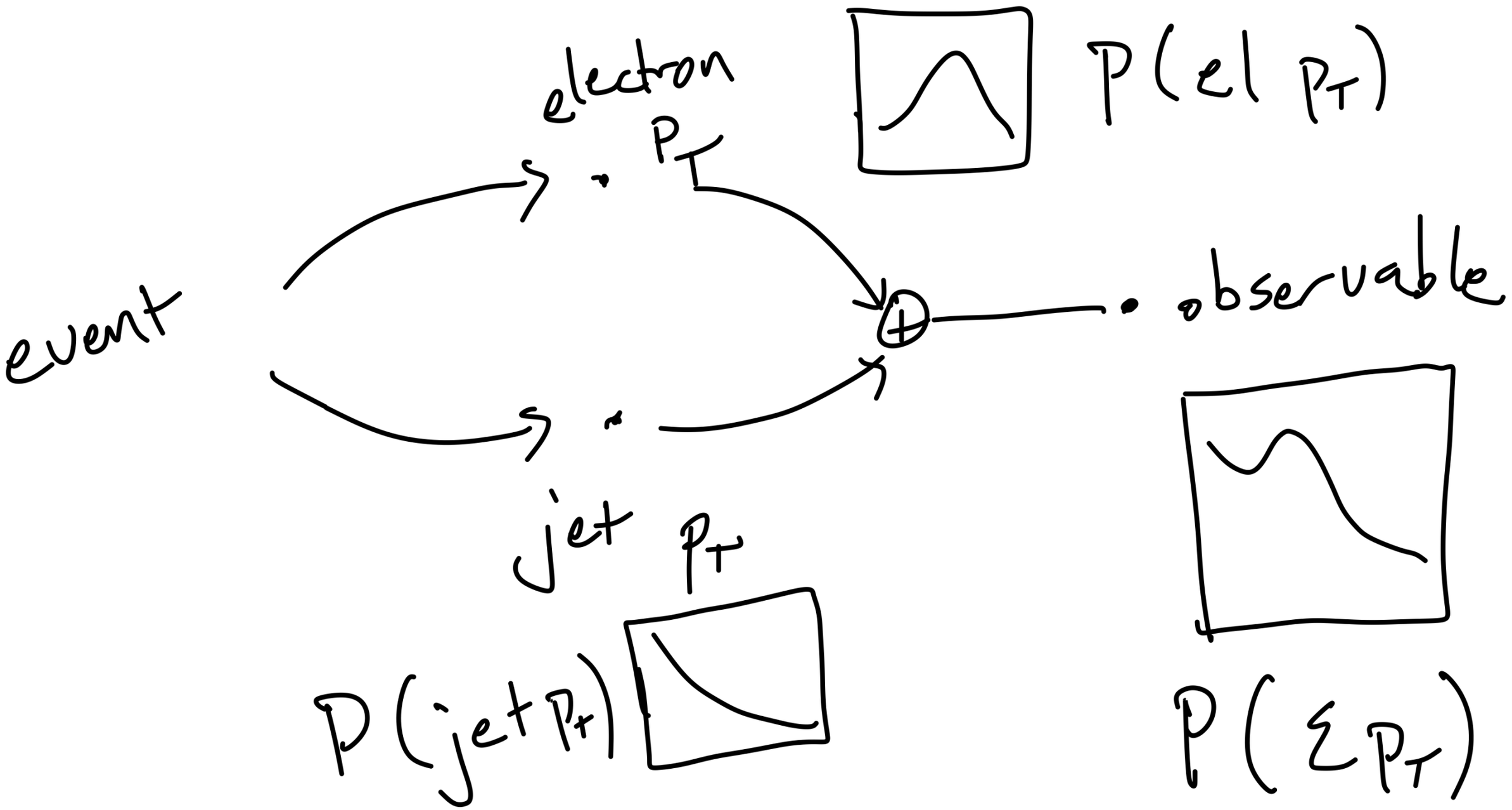
anything that has some form of "multiplication" can be accumulated over the analysis! job configuration, conditions information, application of cuts, etc.

the Writer structure is not at all unique!

there are well-established constraints on structures that yield correctly-composing morphisms.

probability distributions
systematic variations/nuisance-parameter dependences
cut requirements
data-store accesses
data streams
...

are just a few examples (off the top of my head).



Probability is easy.

in my physics analysis, I use a hybrid structure that keeps track of cut requirements, scale factors, and systematic variations:

each sub-morphism of my analysis is allowed to

(1) pass or reject an object due to cut requirements,
(2) apply scale factors and
(3) add systematic variations to the computation.

these effects all compose into one big morphism between an event and the set of observables I extract from it.

I would even go so far as to say this is the "most correct" way to encode corrections, uncertainties, etc.

I haven't said anything yet about histogramming...

folds (or "reduce"s, of which histograms are a subset) are certainly composable!

my (current) preferred way of encoding folds is through Mealy machines, which are basically functions

Mealy a b = a -> (b, Mealy a b)

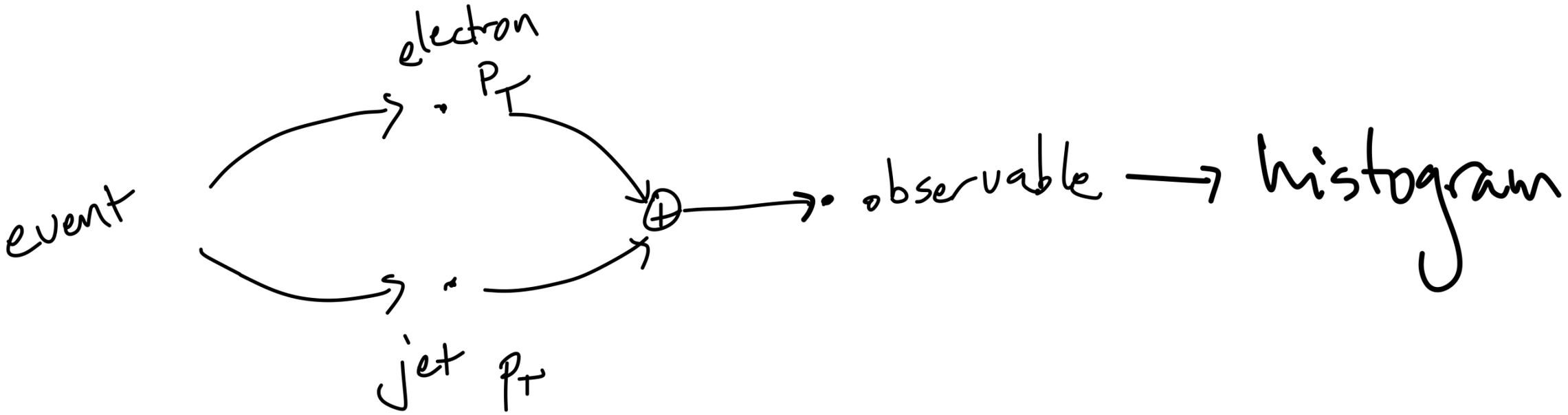
so, you give me something of type *a* and I'll give you a *b* back in addition to a brand new machine!

for histogramming, the *a* type is whatever we have as an input to the histogram, and the *b* is the state of the histogram after the filling has occurred.

the "new machine" has already taken the previous fill into account and is ready to perform the next fill, so in this sense the machine is a generalized accumulator.

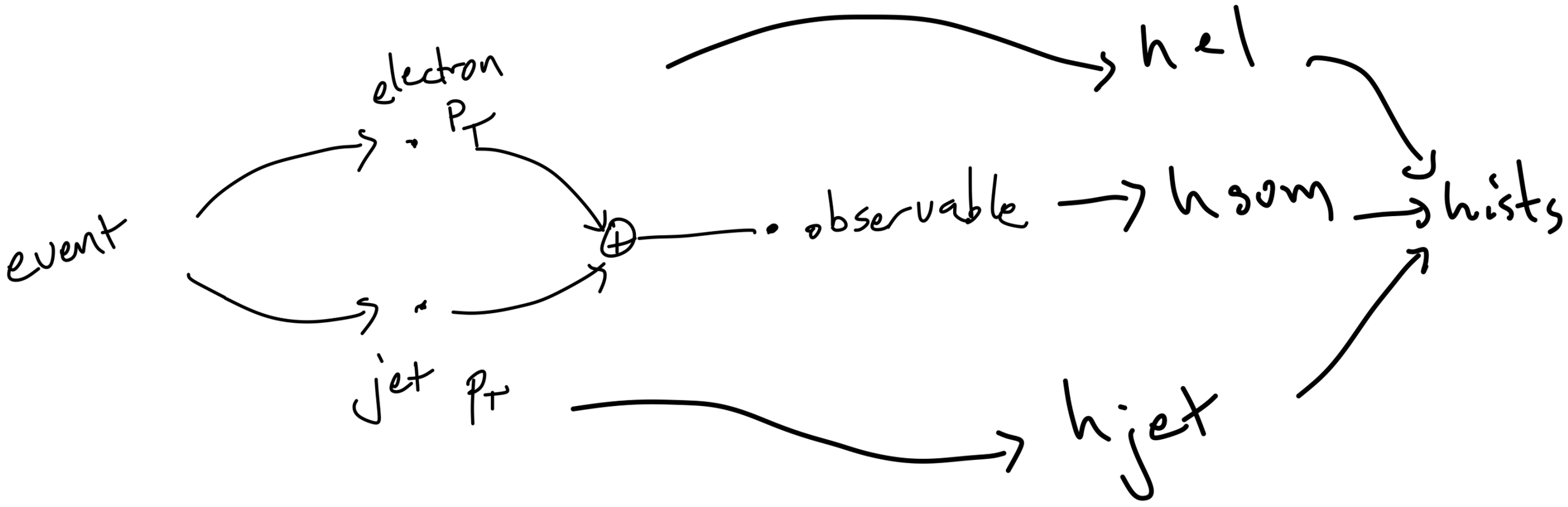
when you are done folding over your dataset, just take the final returned histogram and ignore the new machine.

(or keep it if you want to be able to continue filling at some future point)



the main point being that Mealy machines are
also morphisms in a cartesian closed category!

i.e. I can "compile" the morphisms of any of my analysis diagrams into Mealy machines and reuse exactly the same code I had before to go from an event to e.g. sumpt.



I can attach a histogram filler (or three) to my previous morphism and voila! every time an event comes in, I get a new histogram with the fill included

as well as a brand new mealy machine that will gobble up the next event

i.e. event -> (histograms, Mealy event histograms)

it's also possible to thread all the effects (e.g. systematic variations) through Mealy machines, so we have not lost any generality at all.

a few words about haskell...

traditional programming languages (fortran, C(++), even python)
I believe started from the CPU hardware and its limitations
and evolved layers of abstraction to make it easier
for us to write and understand commands.

haskell (I would claim) is in a class of languages that began with a set of desired mathematical constructs, and a lot of effort was invested into translating these into calculations that can be run on a CPU.

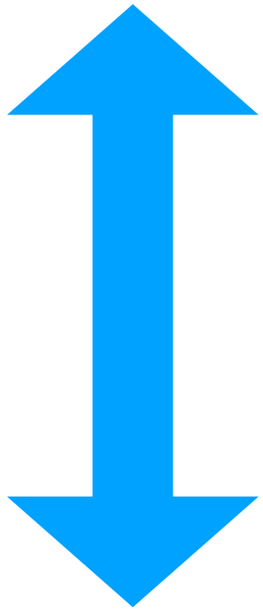
haskellers often speak in terms of the concrete category *Hask*,
which, with a few caveats, has types as objects and functions as morphisms.

the compiler (GHC) explicitly checks that your morphism composition is allowed.

functions in haskell are *pure* unless they are carefully annotated,
meaning there are no global variables or side effects:
if I give the same input to a function a million times, I will always get the same result.

the language has an extremely complete set of available libraries, package management, etc; the ideas presented here are very widespread in the community.

fortran, c, c++, python, java, etc



haskell, etc

over the last several years there has been a significant migration of ideas from haskell (and similar) into both old and new languages.

personally I think many of these constructs will be considered "core" to languages in the (distant) future.

Influenced

Agda,[5] Bluespec,[6] C++11/Concepts,[7]
C#/LINQ,[8][9][10][11] CAL,[*citation needed*]
Cayenne,[8] Clean,[8] Clojure,[12]
CoffeeScript,[13] Curry,[8] Elm,
Epigram,[*citation needed*] Escher,[14] F#,[15]
Frege,[16] Hack,[17] Idris,[18] Isabelle,[8]
Java/Generics,[8] LiveScript,[19] Mercury,[8]
Ωmega,[*citation needed*] Perl 6,[20] PureScript,[21]
Python,[8][22] Rust,[23] Scala,[8][24] Swift,[25]
Timber,[26] Visual Basic 9.0[8][9]

however!

I'd be lying if I said it was all milk and honey

there are a number of drawbacks to a language like haskell:

- **the learning curve is extremely steep, especially if you're coming from a more traditional programming language.**

- **the language feels extremely restrictive to start, because correctness rules are brutally enforced on the programmer. this can be difficult to come to terms with.**

- **evaluation is *non-strict*; i.e. functions are only evaluated when the values they return are needed; this can be difficult to reason about.**

- **speed: in fact haskell is extremely fast (it's compiled), although comparisons to hand-tuned code like BLAS/LAPACK etc is probably not totally fair.**

- **however, because the language is very restrictive in some senses, the compiler is able to perform some substantial improvements behind the scenes.**
  in many cases computations are automatically parallelized!

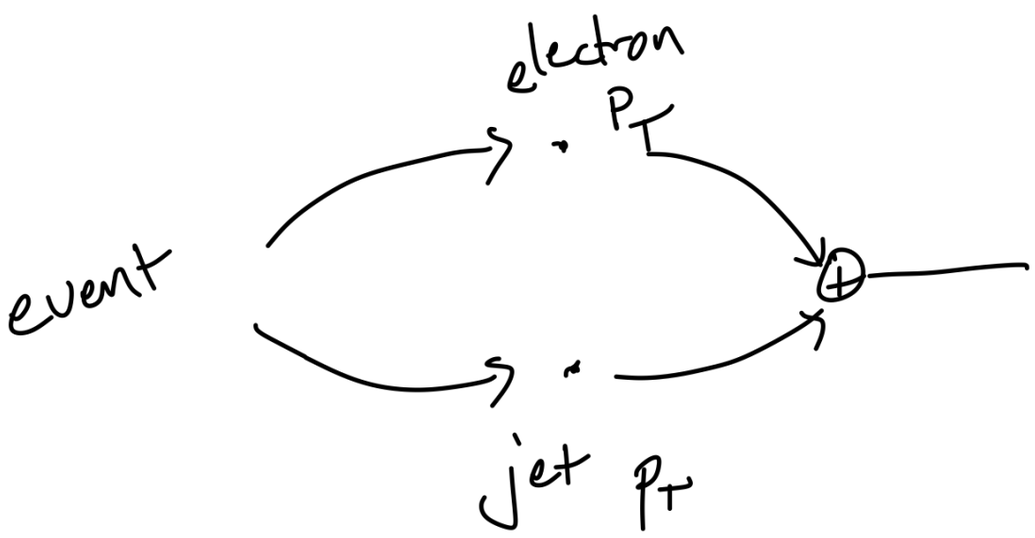
- **probably many more...**

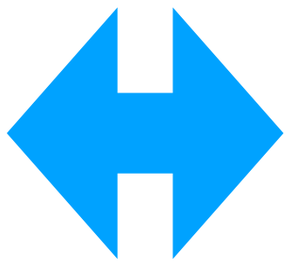
I would not advocate whole-sale migration to a language like haskell for many of the reasons outlined above.

that said, to me it's clear that we should be learning from the concepts and ideas developed by that and similar communities.

maybe we can begin to integrate such ideas in the not-too-distant future without entirely disrupting our code ecosystem.

I still feel there is a lot to be gained from having a very abstract representation of processes that can be cast in many different lights.





```haskell
ptsum = proc event → do
  ept ← elPt ≺ event
  jpt ← jetPt ≺ event
  returnA ≺ ept + jpt
```

to summarize:

ideally source code should be a faithful, useful representation of the mathematical process of physics analysis.

there should be a one-to-one correspondence between the process and the implementation---I hope we can move more in this direction!

thinking compositionally about physics analysis (and mathematical processes more generally) comes with enormous benefits.

adding abstraction, purity, and composability to our code also makes many other interesting (useful!) things possible:

probabilistic programming
syntax tree construction, manipulation, and storing
automatic differentiation
etc

the haskell community has already made huge efforts to make these ideas concretely possible, so we don't have to start from scratch by any means.

and beyond programming, I think we can learn a lot from casting a more formal light on how we interpret our data!

useful references

categorical rosetta stone (Baez):

https://arxiv.org/pdf/0903.0340.pdf

category theory for programmers:

https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/