U051SS2EU : or would it? now I'm unsure
U3JURM9B6 : <@U051SS2EU> : optimizationsa-dvanced has me so scared I jus use (aget ... "field-name") everywhere
U050ECB92 : also if you suspect that something is generating invalid data, that is something clojure.spec + instrument is really good at catching
U11BV7MTK : but if you're scared of bad data getting into your data i think you're working on the wrong side
U3JURM9B6 : here's the thing; I'm dynamically checking some type constraints
U3JURM9B6 : but if I ask that question, ppl tell me to use spec or core.typed :slightly_smiling_face:
U11BV7MTK : so what happens if the types don't align?
U11BV7MTK : and who would introduce the bad data
U3JURM9B6 : throw an assertion
U3JURM9B6 : bad data introduced to my programming bugs
U051SS2EU : <@U3JURM9B6> ```(ins)dev:cljs.user=> (deftype Foo [a b])cljs.user/Foo
(ins)dev:cljs.user=> (aget (Foo. 1 2) "b")
2```

U3JURM9B6 : <@U051SS2EU> : nice; thanks!
U11BV7MTK : well that looks dangerous
U11BV7MTK : that looks like "b" may not track what the fields get renamed to in aggressive compilation
U050ECB92 : i still don't understand why you can't filter bad data at the edge
U3JURM9B6 : <@U050ECB92>: the problem is not bad user input
U051SS2EU : just spitballing: system that isn't designed with the right edges in place?
U3JURM9B6 : the problem is that I, the programmer, introduce bugs
U3JURM9B6 : these are bugs that a static type checker would catch at compile time
U3JURM9B6 : but in clojure, the next best thing I can do is to catch them at runtime
U3JURM9B6 : the input data is fine; it's internal functions that manip the data that violate type constraints
U1ACUMJKX : which of these two implementations would you prefer and why?```(defn inversion [nodes edges]
  (transduce
    (mapcat (fn [[k v]] (map #(vector % k) v)))
    (completing (fn [acc [k v]] (update acc k conj v)))
    (zipmap nodes (repeat #{}))
    edges))```
```(defn inversion [nodes edges]
  (reduce-kv
    (fn [acc k v] (reduce (fn [acc i] (update acc i conj k)) acc v))
    (zipmap nodes (repeat #{}))
    edges))```
expected shape of `nodes` is like `[:framebuffer-0 :framebuffer-1 :framebuffer-2 :texture-0 :texture-1 :texture-2 :texture-3 :program-0 :program-1]`
expected shape of `edges` is like
```
{:framebuffer-0 #{:texture-0 :texture-1 :program-0 :framebuffer-1}
 :framebuffer-1 #{:framebuffer-2 :texture-2 :texture-3 :program-1}
 :framebuffer-2 #{:texture-0 :program-1}}```
correct output for those specific inputs:
```
{:framebuffer-0 #{}
 :framebuffer-1 #{:framebuffer-0}
 :framebuffer-2 #{:framebuffer-1}
 :texture-0 #{:framebuffer-2 :framebuffer-0}
 :texture-1 #{:framebuffer-0}
 :texture-2 #{:framebuffer-1}
 :texture-3 #{:framebuffer-1}
 :program-0 #{:framebuffer-0}
 :program-1 #{:framebuffer-1 :framebuffer-2}}```

U051SS2EU : <@U3JURM9B6> I'd think accidentally calling an accessor and constructor would be a lot less likely than accidentally calling assoc on the wrong object
U3JURM9B6 : <@U051SS2EU> : I can make the constructor private, then have a function (which calls the constructor) do checks beforehand
```

U04VDQDDY : <@U3JURM9B6> For accessing object properties (as opposed to array elements), consider `goog.object/get` instead of `aget`.
U050ECB92 : <@U3JURM9B6> creating custom types that have their own field accessors completely negates the value of generic data access from the clojure std library
U050ECB92 : your code will essentially be a non-reusable DSL
U050ECB92 : I firmly believe you are going down the wrong path, and that the advice you're getting about specific implementation is misguided
U050ECB92 : you should test against invalid / incorrect data (even from your own code). Have you considered writing generative tests for your datastructures? (either from test.check or from spec)
U050ECB92 : test against bad data while allowing bad data to exist
U050ECB92 : Lots of things that are common in other languages (e.g. privileged data) are the exact wrong approach in Clojure.