U681TBBUP : The only notion needed for bounding something is comparability, maybe it doesn't make sense in some contexts but there isn't a reason to unnecessarily restrict it
U3ZNWN526 : Hmm... There's a strange thing happening (or maybe I'm just missing something obvious): I have an extensible record type `WithUserFields` and an extended record `UserFields`, which doesn't add any additional fields:
```
type alias WithUserFields a = { a | ... }
type alias UserFields = WithUserFields {}
```
But when I try to use `UserFields`, the compiler spits out "cannot find variable `UserFields`"

U2LAL86AY : how do you try to use `UserFields` can you show that particular example?
U2LAL86AY : and you can't have `WithUserFields` and `WithUserFields ` twice in the same module i think - ups i thougth is a `type` - delete that :smile:
U5QJW0DDE : Incidentally this is only true if your data model is structured the same as your UI. Rarely does a UI page equate to an exact segment of your data model.
U3ZNWN526 : Sure! The first case for example, is:
```
userFieldsFrom : User -&gt; UserFields
userFieldsFrom u =
  UserFields u.userRole u.username u.name u.email u.password u.passwordConfirmation
```

U3ZNWN526 : Well for `WithUserFields` the first time is defining it, and the second time I'm using it (to define `UserFields`)
U5QJW0DDE : I look forward to watching that.
U2LAL86AY : there is a small article about extensible records here. I can't find the other one i saved once - was much better at explaining how to compose extensible records.
<https://dennisreimann.de/articles/elm-data-structures-record-tuple.html>
U2LAL86AY : ok let check it out
U2LAL86AY : i simplified a bit:
```
type alias WithUserFields a =
    { a | name : String }


type alias UserFields =
    WithUserFields {}


userFieldsFrom : String -&gt; UserFields
userFieldsFrom string =
    WithUserFields string

```
error:
```
Cannot find variable `WithUserFields`
```

U2LAL86AY : so it seems like you can't have parametric types act as constructors
U3ZNWN526 : Oh, no, I have `UserFields` as the constructor within that last function, not `WithUserFields`
U2LAL86AY : Maybe is the same thing - since it gives the same error.
if you have `type alias Thing = String `
you get back a function named `Thing : String -&gt; Thing`.

But if you have `type alias Thing a  = String `
Seems you don't have this `Thing: String -&gt; Thing` anymore. It's how elm works.
 Let me try a few more things first

U3ZNWN526 : Ooh, so you think even with `type alias Thing = MetaThing {}` maybe wouldn't give you the constructor? I suppose that's possible... very unexpected though, and doesn't seem to be documented anywhere
U3ZNWN526 : I would think with a `type alias Thing = ` I could put anything after the `=` and it would give me a

constructor.
U5J0H2NS2 : I think this```
WithUserFields a = { a | ... }
```
means only, that the type has several known fields.
You did not specify their order. So you can not construct one just like you wanted.

U2LAL86AY : hmm yes, you don't get alias constructors for aliases with parametric values:```

type alias FirstAlias =
    { name : String }



type alias SecondAlias a =
    { a | name : String }



constructAValueOfFirstAlias : String -&gt; FirstAlias
constructAValueOfFirstAlias string =
    FirstAlias string
         --- works as expected because you have in the global scope a (function/variable) named: FirstAlias : String -&gt; FirstAlias



constructAValueOfSecondAlias : String -&gt; SecondAlias {}
constructAValueOfSecondAlias string =
    SecondAlias string
         -- error "Cannot find variable `SecondAlias`"

```

U3ZNWN526 : Oh well yes, that does make sense, because SecondAlias is not a type, you would only be able to construct other types using SecondAlias. But what I'm trying to do is define a second type and then use it!
U0J1M0F32 : ```
type SecondAlias a =
    SecondAlias { a | name : String }

constructAValueOfSecondAlias : String -&gt; SecondAlias {}
constructAValueOfSecondAlias string =
    SecondAlias { name = string }
```

U5J0H2NS2 : But this works```
type alias WithUserFields a = { a | name : String, email : String }
type alias UserFields = WithUserFields {}

change_name : String -&gt; UserFields -&gt; UserFields
change_name new_name user_fields = { user_fields | name = new_name }

get_user_fields : String -&gt; String -&gt; UserFields
get_user_fields name email =
    { name = name, email = email }
```

U3ZNWN526 : Yeah. That seems right. The problem I'm having is that for some reason... there's no `UserFields` constructor? (even though `UserFields` is just a plain old type alias)
U0CL0AS3V : &gt; all the structural problem i've been discussing started to surface in my company's app after it hit about 8000 lines of clojurescript (which have since doubled). I think none of the issues I present would be that apparent in smaller apps&gt; really the problems i'm talking grow exponentially with the size of your data model

we're at 150,000 lines of Elm code in production and these problems don't exist for us. I don't worry about the Elm Architecture scaling because my experience with it is that it's scaled better than any other front-end system I've ever worked on. :slightly_smiling_face:

U5QJW0DDE : Currently it is 15k lines. Do you find that either a) your data model lines up with your view tree much of the time or b) you pass the whole model state around often?
U0CL0AS3V : <@U5QJW0DDE> I get why you'd be worried about this, based on your experiences elsewhere, but the people saying "this is not a problem in Elm" are not making it up :wink:
U5QJW0DDE : <@U0CL0AS3V> I still haven't seen any examples that demonstrate an Elm solution to, for example, the distant descendant drop down list I mentioned earlier