This is a follow-up to my previous post `https://shuangrimu.com/posts/elm-extensible-unions.html`

Since then there's been quite a lot of discussion on the Elm Discourse thread (`https://discourse.elm-lang.org/t/idea-extensible-union-types-and-benefits-they-bring-for-real-world-elm-code/6118`) about all this. In particular Evan Czaplicki replied there asking for more formal treatment of the informal type system I laid out in my previous post.

This post gives that. In order to write out the typing judgments effectively I've used LaTeX to generate a PDF instead. This PDF will be a fairly abbreviated treatment of the issue. In particular I'll be assuming familiarity with the formal treatment of the Hindley-Milner type system as well as familiarity with Daan Leijen's paper "Extensible records with scoped labels", which forms the theoretical basis of Elm's type system. You should definitely have the latter paper by your side while reading this, as I will constantly be making references to notation and examples in Leijen's paper.

If that sounds scary to you don't worry! This is just a demonstration of the formal foundations of the ideas presented in `https://shuangrimu.com/posts/elm-extensible-unions.html`. If you understood that post just fine, this PDF does nothing more than show that my assertions in that post are formally justified. On the other hand, if you are interested in a more approachable look at this topic, please let me know on the Elm discourse! I can try to carve out some time to write a more in-depth article covering the basics of type systems and type inference aimed at people who are not familiar with the formalization of those concepts, but are familiar with Elm.

This PDF is mainly aimed at people like Evan who are already familiar with these concepts and would just like to see how my pseudo-Elm from my earlier post maps onto these formal concepts.

In particular this document is about

- Showing the correspondence between my pseudo-Elm and the above formalisms.
- Providing worked examples of type checking and type inference according to these formalisms justifying my informal assertions of why certain expressions are given certain type signatures

In the interest of continuity I haven't made any syntax changes from my previous post (despite some excellent suggestions from contributors in the Discourse thread).

Given the close relationship between extensible unions and extensible records, it's no surprise that Leijen's paper actually mentions and provides a formalism for extensible unions as well (see section 5 of the paper "Variants," remember another name for extensible unions is polymorphic variants). In fact extensible unions uses the exact same type mechanism as extensible records! Typechecking and type inference proceed *identically*[1].

The split in type behavior arises solely from restricting which primitive functions are available for extensible records vs which primitives are available for extensible unions.

So luckily all the hard work is done for me! The correspondence between my pseudo-Elm and the formalisms presented in Leijen's paper is a straightforward desugaring.

N.B. from here on our I will be referring to extensible unions (such a name being chosen to be consistent with Elm's own naming conventions) instead as polymorphic variants because the latter is much more prevalent in CS literature and indeed is the terminology used by Leijen.

I won't provide a formal treatment for the desugaring (I'm pretty sure that would be overkill, but feel free to email me if you believe otherwise), instead I'm going to take the examples from my "What are extensible union types" section and write out the full typechecking and type inference process for them.

In particular we'll be using the following pseudo-Elm code from my previous post. Hopefully that is sufficiently illustrative of the desugaring process to convince readers that in fact I am using no new ideas apart from those already presented in Leijen's paper.

```
regularUser : a or @RegularUser Int
regularUser = @RegularUser 0

adminUser : a or @AdminUser String
adminUser = @AdminUser "admin"
```

[1]In fact one possible implementation of polymorphic variants is to simply provide Leijen's set of primitive functions, and then use an opaque wrapper type around Elm's usual records that cannot be unwrapped by normal user-land code. This requires absolutely no changes to Elm's type system. The only hard-coded change required in the compiler is the addition of the primitive functions. Of course there are various losses in developer ergonomics such as the quality of type messages in that approach, but it serves as an illustration that there is fundamentally nothing new going on at the type level.

```
regularUser1 : @RegularUser Int or @AdminUser String
regularUser1 = regularUser

toInt : @RegularUser Int or @AdminUser String -> Int
toInt user = case# user of
    @RegularUser x -> x
    @AdminUser str -> length str

thisTypeChecks : Bool -> a or @RegularUser Int or @AdminUser String
thisTypeChecks bool = case bool of
    True -> regularUser
    False -> adminUser

thisTypeChecksToo : @RegularUser Int or @AdminUser String or @SomeOtherTag Bool
thisTypeChecksToo = thisTypeChecks True

-- This was not in the original post, but serves a usual example of how an
-- extensible type variable interacts with a concrete input type in a function.
myInt : Int
myInt = toInt adminUser

-- This is a type error and is used mainly to demonstrate that the type errors
-- I presented in my post do in fact happen
failsToCompile : @RegularUser Int or @AdminUser String -> Int
failsToCompile user = case# user of
    @AdminUser str -> length str
```

I mean to show that Leijen's formalism agree with the following characteristics.

1. Functions built with case expressions result in a concrete (i.e. monomorphic) input type
2. Exhaustivity checking still occurs in case matches
3. Type inference agrees with the type signatures presented

Again, I'll be assuming a standard Hindley-Milner (HM) type system on top of which we bolt Leijen's additional record system. This neatly models Elm type system with only a few minor variations. If people are interested are in these variations please let me know! I can include them, but for now I'll leave them out since they don't affect the core of the idea behind polymorphic variants.

Although I will generally hew very closely to Leijen's formalism, to highlight case match exhaustivity I will introduce a new primitive function *caseMatchFinished* for variants. Note that this is merely a more restricted version of Leijen's *error* function. I include it here to explicitly model Elm's promise of no runtime exceptions. *caseMatchFinished* allows us to follow Leijen's method of desugaring case statements without introducing something that looks operationally like a runtime exception. It also helps us infer closed row types in the input type of a case match statement[2].

$$caseMatchFinished :: \forall \alpha^{*}. \langle \llparenthesis\, \rrparenthesis \rangle \rightarrow \alpha^{*}$$

Note that this is a function that has no runtime evaluation significance because of its input type $\langle \llparenthesis\, \rrparenthesis \rangle$. There is no way of generating a value of type $\langle \llparenthesis\, \rrparenthesis \rangle$ outside of using Leijen's variant decomposition function $l \in \_ \; ? \; \_ : \_$ (which in turn only appears in case matches). However, that means we can only call *caseMatchFinished* when we've handled all other cases. In otherwords, at runtime we actually never call *caseMatchFinished*, it is purely a compile-time artifact. Hence *caseMatchFinished* can be given an arbitrary runtime implementation.

For simplicity's sake, instead of doing a full treatment of (closed, non-row-based) algebraic datatypes at the type level, I will assign a hard-coded primitive function that follows the usual function-based encoding[3] of algebraic datatypes, to be used in every case match instance.

[2]Assuming that Leijen intended *error* to be of type $\forall \alpha^{*}. String \rightarrow \alpha^{*}$, then Leijen's *showEvent* would be inferred as $\forall r^{\text{row}}. \langle key :: Char, mouse :: Point \mid r^{\text{row}} \rangle \rightarrow String$ (α^{*} unifies to a function type with an input variant type unlike in *caseMatchFinished* where the input is already a variant type) if it had not been annotated with a closed row type. This is a rather poor type signature for runtime safety, however, since any variant not tagged as *key* or *mouse* would hit the *error* branch and cause a runtime error!

[3]Technically a Boehm-Berarducci encoding, but since we don't have any recursive datatypes, all the usual function-based encodings based off of Church encodings work and agree.

Luckily in the pseudo-Elm code there is only one such instance, namely the case match on a boolean type. So we substitute instead the following function for the case match on a boolean type.

$$boolMatch :: \forall \alpha^{*}.Bool \to \alpha \to \alpha \to \alpha$$

The intended runtime interpretation of this function is that the first α is chosen if $Bool$ is $True$ otherwise the second α is chosen if $Bool$ is $False$.

Also, not included in the pseudo-Elm code, but something worth pointing out is that variant tags introduced by `@` uniformly desugar to Leijen's variant injection function. That is for `@SomeTag`, Leijen's equivalent formalism is

$$\langle SomeTag = \_ \rangle :: \forall \alpha r^{\text{row}}.\alpha \to \langle \, (\!| SomeTag :: \alpha \mid r |\!) \, \rangle$$

With that in mind let's present the full desugaring of our pseudo-Elm. I follow almost exactly Leijen's own desugaring, in Section 5 of his "Extensible records with scoped labels" paper. In particuar, I follow his method of desugaring a case match in his Morrow language into a series of nested $\in$ applications. The only difference we make, as I noted earlier, is to replace his use of $error$ with $caseMatchFinished$.

$$\begin{aligned}
regularUser &:: \forall r^{\text{row}}.\langle \, (\!| RegularUser :: Int \mid r^{\text{row}} |\!) \, \rangle \\
regularUser &= \langle RegularUser = 0 \rangle
\end{aligned}$$

$$\begin{aligned}
adminUser &:: \forall r^{\text{row}}.\langle \, (\!| AdminUser :: String \mid r^{\text{row}} |\!) \, \rangle \\
adminUser &= \langle AdminUser = \text{“admin”} \rangle
\end{aligned}$$

$$\begin{aligned}
regularUser1 &:: \langle \, (\!| RegularUser :: Int \mid \, (\!| AdminUser :: String \mid \, (\!| \, |\!) \, |\!) \, |\!) \, \rangle \\
regularUser1 &= regularUser
\end{aligned}$$

$$\begin{aligned}
toInt &:: \langle \, (\!| RegularUser :: Int \mid \, (\!| AdminUser :: String \mid \, (\!| \, |\!) \, |\!) \, |\!) \, \rangle \to Int \\
toInt\ user &= (RegularUser \in user \ ? \ (\backslash x \to x) : \\
&\qquad (\backslash user' \to (AdminUser \in user' \ ? \ (\backslash str \to length\ str) : \\
&\qquad\quad caseMatchFinished)))
\end{aligned}$$

$$\begin{aligned}
thisTypeChecks &:: \forall r^{\text{row}}.Bool \to \langle \, (\!| RegularUser :: Int \mid \, (\!| AdminUser :: String \mid r^{\text{row}} |\!) \, |\!) \, \rangle \\
thisTypeChecks\ bool &= boolMatch\ bool\ regularUser\ adminUser
\end{aligned}$$

$$\begin{aligned}
thisTypeChecksToo &:: Bool \to \langle \, (\!| RegularUser :: Int \mid \, (\!| AdminUser :: String \mid \, (\!| \, |\!) \, |\!) \, |\!) \, \rangle \\
thisTypeChecksToo &= thisTypeChecks\ True
\end{aligned}$$

$$\begin{aligned}
myInt &:: Int \\
myInt &= toInt\ regularUser
\end{aligned}$$

$$\begin{aligned}
failsToCompile &:: \langle \, (\!| RegularUser :: Int \mid \, (\!| AdminUser :: String |\!) \, |\!) \, \rangle \to Int \\
failsToCompile &= AdminUser \in user' \ ? \ (\backslash str \to length\ str) : caseMatchFinished
\end{aligned}$$

Note that although it is not defined in the original pseudo-Elm code, $length$ has the following type signature:

$$length :: String \to Int$$

For type checking it is mostly sufficient to run the type inference algorithm sketched in Figure 2 of Leijen's paper and verify that the types inferred agree with our annotated types up to bound type variables (by the soundness of our inference algorithm we know that these types must then be correct).

The two exceptions are $regularUser1$ and $thisTypeChecksToo$, which are annotated with a more specific type than is inferred. However, as we shall see, a simple use of the instantiation `Inst` rule from HM is sufficient to prove that $thisTypeChecksToo$ indeed has a valid type signature.

Leijen leaves unsaid the entire framework for type inference apart from unification, but we can use the usual type inference method for HM-like systems. Of note, we will always first infer a monotype before closing over all free type variables to form a polytype.

1. $regularUser$: Let's begin by examining $regularUser$. It is an application of $\langle l = \_\rangle$ to a single argument[4].

   Given a fresh type variable β that we wish to solve for, we have the following constraint

   $$\alpha \to \langle \llparenthesis \mathit{RegularUser} :: \alpha \mid r^{\text{row}} \rrparenthesis \rangle \sim \beta \to \langle \llparenthesis \mathit{RegularUser} :: \beta \mid r^{\text{row}} \rrparenthesis \rangle$$

   with an empty substitution set. One use of Leijen's *uni-app* followed by a use of *uni-varl* yields our unifier $[\beta \mapsto \mathit{Int}]$.

   I introduce a new rule called *uni-function* which is just a specialization of repeated usages of *uni-app* where we think of the function arrow $\to$ as a type constructor of two arguments.

   $$\textit{uni-function}: \dfrac{\tau_1 \sim \tau_1' : \theta_1 \qquad \theta_1\tau_2 \sim \theta_1\tau_2' : \theta_2}{\tau_1 \to \tau_2 \sim \tau_1' \to \tau_2' : \theta_2 \circ \theta_1}$$

   Although it can be derived from *uni-app*, it makes a lot of our resulting work easier to write out.

   Because of the usual abuse of notation using proof tree notation to denote a procedural algorithm, we will need to split up our unification over several different proof trees to show each step as it's run.

   We first apply *uni-function*, where the first precondition is satisfied by *uni-varl*.

   $$\textit{uni-function:}\ \dfrac{\alpha^* \sim \mathit{Int} : [\alpha^* \mapsto \mathit{Int}] \qquad [\alpha^* \mapsto \mathit{Int}]\langle \llparenthesis \mathit{RegularUser} :: \alpha^* \mid r^{\text{row}} \rrparenthesis \rangle \sim [\alpha^* \mapsto \mathit{Int}]\beta : ?}{\alpha^* \to \langle \llparenthesis \mathit{RegularUser} :: \alpha^* \mid r^{\text{row}} \rrparenthesis \rangle \sim \mathit{Int} \to \beta : ?}$$

   After applying our substitution set $[\alpha^* \mapsto \mathit{Int}]$, we invoke *uni-varl* again.

   $$\textit{uni-function}:\ \dfrac{\alpha^* \sim \mathit{Int} : [\alpha^* \mapsto \mathit{Int}] \qquad \textit{uni-varl:}\ \dfrac{\begin{array}{c}\langle \llparenthesis \mathit{RegularUser} :: \mathit{Int} \mid r^{\text{row}} \rrparenthesis \rangle \sim \beta : \\ [\beta \mapsto \langle \llparenthesis \mathit{RegularUser} :: \mathit{Int} \mid r^{\text{row}} \rrparenthesis \rangle]\end{array}}{\langle \llparenthesis \mathit{RegularUser} :: \mathit{Int} \mid r^{\text{row}} \rrparenthesis \rangle \sim \beta : ?}}{\alpha^* \to \langle \llparenthesis \mathit{RegularUser} :: \alpha^* \mid r^{\text{row}} \rrparenthesis \rangle \sim \mathit{Int} \to \beta : ?}$$

   And cascading our substitution set down yields our final proof tree.

   $$\dfrac{\alpha^* \sim \mathit{Int} : [\alpha^* \mapsto \mathit{Int}] \qquad \dfrac{\langle \llparenthesis \mathit{RegularUser} :: \mathit{Int} \mid r^{\text{row}} \rrparenthesis \rangle \sim \beta : [\beta \mapsto \langle \llparenthesis \mathit{RegularUser} :: \mathit{Int} \mid r^{\text{row}} \rrparenthesis \rangle]}{\langle \llparenthesis \mathit{RegularUser} :: \mathit{Int} \mid r^{\text{row}} \rrparenthesis \rangle \sim \beta : [\beta \mapsto \langle \llparenthesis \mathit{RegularUser} :: \mathit{Int} \mid r^{\text{row}} \rrparenthesis \rangle]}}{\alpha^* \to \langle \llparenthesis \mathit{RegularUser} :: \alpha^* \mid r^{\text{row}} \rrparenthesis \rangle \sim \mathit{Int} \to \beta : [\alpha^* \mapsto \mathit{Int}, \beta \mapsto \langle \llparenthesis \mathit{RegularUser} :: \mathit{Int} \mid r^{\text{row}} \rrparenthesis \rangle]}$$

   That gives us our final unifier of

   $$[\alpha^* \mapsto \mathit{Int}, \beta \mapsto \langle \llparenthesis \mathit{RegularUser} :: \mathit{Int} \mid r^{\text{row}} \rrparenthesis \rangle]$$

   A single application of that unifier to β indeed yields the type signature

   $$\langle \llparenthesis \mathit{RegularUser} :: \mathit{Int} \mid r^{\text{row}} \rrparenthesis \rangle$$

   , showing that our annotated type signature for $regularUser$ is indeed correct (after closing over free type variables with HM's generalization rule to generate the polytype

   $$\forall r^{\text{row}}.\langle \llparenthesis \mathit{RegularUser} :: \mathit{Int} \mid r^{\text{row}} \rrparenthesis \rangle$$

).

[4]It is indeed one argument, not two arguments, i.e. l itself is not an argument. This is because formally $\langle l = \_\rangle$ is a syntactic convenience that denotes a family of functions parametrized by l and is *syntactically* specialized to a single function at a call site.

2. *adminUser*: The same procedure for *regularUser* suffices to show that our type signature for *adminUser* is correct.

3. *regularUser*1: This follows by *uni-varl*. Alternatively, with a type signature for *regularUser*, a single (kind-aware) use of the HM instantiation rule[5] immediately yields *regularUser*1.

4. *toInt*: For *toInt*, we have a lambda abstraction rather than application. Following the usual HM abstraction rule this means means at the top-level we introduce a fresh type variable γ with a new typing environment $\Gamma' = \Gamma \cup \{\mathit{bool} : \gamma\}$[6] that we use for the remainder of this expression.

   The top-level of *toInt*, i.e.

   $$(\mathit{RegularUser} \in \mathit{user}\ ?\ (\backslash x \to x) : \ldots)$$

   is a lambda application that then proceeds in a similar fashion to the process laid out for *regularUser*.

   The proof trees for *toInt* become extremely cumbersome to write out, so I'll appeal to the reader's trust that we just alternate instances of HM's abstraction rule introducing a new typing environment and the lambda application inference process laid out in *regularUser*.

   The only point I wish to emphasize here is that *toInt* ultimately has no row type variables at the top-level, i.e. we have entirely closed row types in *toInt*'s signature rather than any instances of r^{row}. The reason behind this is *caseMatchFinished*, which forces an instantiation of the row type variable introduced by nested $\in$s to a $\llparenthesis\ \rrparenthesis$, effectively "removing" the type variable.

5. *thisTypeChecks*: *thisTypeChecks* is mostly straightforward. Assuming that we already have the types for *regularUser* and *adminUser* in our typing environment (whose types we'll call R and A respectively to make our proof tree size more manageable), we get the following series of proof trees where ν_i is a fresh type variable for $i \in \{1, 2, \ldots\}$.

   We solve repeated partial applications from left to right. First partially applying to *bool* using *uni-function*.

   $$\frac{\mathit{Bool} \sim \mathit{Bool} : [] \qquad (\alpha^* \to (\alpha^* \to \alpha^*)) \sim \nu_1^* : [\nu_1^* \mapsto (\alpha^* \to (\alpha^* \to \alpha^*))]}{\mathit{Bool} \to (\alpha^* \to (\alpha^* \to \alpha^*)) \sim \mathit{Bool} \to \nu_1^* : [\nu_1^* \mapsto (\alpha^* \to (\alpha^* \to \alpha^*))]}$$

   Then we partially apply *regularUser*.

   $$\frac{\alpha^* \sim \langle \llparenthesis \mathit{RegularUser} :: \mathit{Int} \mid r^{\text{row}} \rrparenthesis \rangle : [\alpha^* \mapsto \langle \llparenthesis \mathit{RegularUser} :: \mathit{Int} \mid r^{\text{row}} \rrparenthesis \rangle] \qquad (\alpha^* \to \alpha^*) \sim \nu_2^* : ?}{\alpha^* \to (\alpha^* \to \alpha^*) \sim \langle \llparenthesis \mathit{RegularUser} :: \mathit{Int} \mid r^{\text{row}} \rrparenthesis \rangle \to \nu_2^* : ?}$$

   Setting $\mathit{RUr}^{\text{row}} = \llparenthesis \mathit{RegularUser} :: \mathit{Int} \mid r^{\text{row}} \rrparenthesis$ to make the proof tree less intimidating:

   $$\frac{\alpha^* \sim \mathit{RUr}^{\text{row}} : [\alpha^* \mapsto \mathit{RUr}^{\text{row}}] \qquad [\alpha^* \mapsto \mathit{RUr}^{\text{row}}](\alpha^* \to \alpha^*) \sim [\alpha^* \mapsto \mathit{RUr}^{\text{row}}]\nu_2^* : ?}{\alpha^* \to (\alpha^* \to \alpha^*) \sim \mathit{RUr}^{\text{row}} \to \nu_2^* : ?}$$

   $$\frac{\alpha^* \sim \mathit{RUr}^{\text{row}} : [\alpha^* \mapsto \mathit{RUr}^{\text{row}}] \qquad (\mathit{RUr}^{\text{row}} \to \mathit{RUr}^{\text{row}}) \sim \nu_2^* : [\alpha^* \mapsto \mathit{RUr}^{\text{row}}, \nu_2^* \mapsto \mathit{RUr}^{\text{row}} \to \mathit{RUr}^{\text{row}}]}{\alpha^* \to (\alpha^* \to \alpha^*) \sim \mathit{RUr}^{\text{row}} \to \nu_2^* : [\alpha^* \mapsto \mathit{RUr}^{\text{row}}, \nu_2^* \mapsto \mathit{RUr}^{\text{row}} \to \mathit{RUr}^{\text{row}}]}$$

   This yields $\forall r^{\text{row}}.\mathit{RUr}^{\text{row}} \to \mathit{RUr}^{\text{row}}$ (after substitution for ν_2 and closing over our free type variables with $\forall$) as our inferred type after partial application of *regularUser*.

   Finally applying *adminUser* is the one interesting part, since we have to make use of Leijen's *uni-row* rule, which up to now we haven't seen. We'll use the abbreviation

   $$\mathit{AUw}^{\text{row}} = \langle \llparenthesis \mathit{AdminUser} :: \mathit{String} \mid w^{\text{row}} \rrparenthesis \rangle$$

   to again make the initial *uni-function* rule shorter.

[5]We need to extend the usual $\sqsubseteq$ specialization operator with Leijen's $\cong$ operator but otherwise things remain the same.

[6]Γ is the implicit environment we've been carrying around that among other things includes the typings for $l \in \_?\_ : \_$ and *caseMatchFinished*.

$$\frac{RUr^{\text{row}} \sim AUw^{\text{row}} : ? \qquad RUr^{\text{row}} \sim \nu_3^* : ?}{RUr^{\text{row}} \to RUr^{\text{row}} \sim AUw^{\text{row}} \to \nu_3^* : ?}$$

Focusing on running *uni-row* on $RUr^{\text{row}} \sim AUw^{\text{row}} : ?$ we get (after one invocation of *uni-app* on the $\langle\rangle$ constructor strips us to bare row types) the following. I notate the three remaining conditions we're not yet using as B, C, and D to keep the proof tree manageable in size.

$$\textit{uni-row:}\ \frac{\llparenthesis AdminUser :: String \mid w^{\text{row}} \rrparenthesis \simeq \llparenthesis RegularUser :: \tau' \mid s' \rrparenthesis : ? \qquad \text{B} \qquad \text{C} \qquad \text{D}}{\llparenthesis RegularUser :: Int \mid r^{\text{row}} \rrparenthesis \sim \llparenthesis AdminUser :: String \mid w^{\text{row}} \rrparenthesis : ?}$$

To solve for $\llparenthesis AdminUser :: String \mid w^{\text{row}} \rrparenthesis \simeq \llparenthesis RegularUser :: \tau' \mid s' \rrparenthesis : ?$ we first run *row-swap.* We leave τ' as is and match s' to $\llparenthesis AdminUser :: String \mid t^{\text{row}} \rrparenthesis$ for a new t^{row}.

$$\textit{row-swap:}\ \frac{RegularUser \neq AdminUser \qquad w^{\text{row}} \simeq \llparenthesis RegularUser :: \tau' \mid t^{\text{row}} \rrparenthesis : ?}{\llparenthesis AdminUser :: String \mid w^{\text{row}} \rrparenthesis \simeq \llparenthesis RegularUser :: \tau' \mid \llparenthesis AdminUser :: String \mid t^{\text{row}} \rrparenthesis \rrparenthesis : ?}$$

Solving for $r^{\text{row}} \simeq \llparenthesis RegularUser :: \tau' \mid t^{\text{row}} \rrparenthesis : ?$ requires an invocation of *row-var.*

$$\textit{row-var:}\ \frac{\text{fresh}(\tau') \qquad \text{fresh}(t^{\text{row}})}{w^{\text{row}} \simeq \llparenthesis RegularUser :: \tau' \mid t^{\text{row}} \rrparenthesis : [w^{\text{row}} \mapsto \llparenthesis RegularUser :: \tau' \mid t^{\text{row}} \rrparenthesis]}$$

. This then lets us fully solve *row-swap.*

$$\textit{row-swap:}\ \frac{RegularUser \neq AdminUser \qquad \begin{array}{c} w^{\text{row}} \simeq \llparenthesis RegularUser :: \tau' \mid t^{\text{row}} \rrparenthesis : \\ [w^{\text{row}} \mapsto \llparenthesis RegularUser :: \tau' \mid t^{\text{row}} \rrparenthesis] \end{array}}{\begin{array}{c} \llparenthesis AdminUser :: String \mid w^{\text{row}} \rrparenthesis \simeq \llparenthesis RegularUser :: \tau' \mid \llparenthesis AdminUser :: String \mid t^{\text{row}} \rrparenthesis \rrparenthesis : \\ [w^{\text{row}} \mapsto \llparenthesis RegularUser :: \tau' \mid t^{\text{row}} \rrparenthesis] \end{array}}$$

We then bubble up to *uni-row.* I'll use the abbreviation $W = [w^{\text{row}} \mapsto \llparenthesis RegularUser :: \tau' \mid t^{\text{row}} \rrparenthesis]$ to make the proof tree easier to handle.

$$\textit{uni-row:}\ \frac{\begin{array}{c} \llparenthesis AdminUser :: String \mid w^{\text{row}} \rrparenthesis \simeq \llparenthesis RegularUser :: \tau' \mid \llparenthesis AdminUser :: String \mid t^{\text{row}} \rrparenthesis \rrparenthesis : W \\ \text{tail}(\llparenthesis \rrparenthesis) \notin \text{dom}([w^{\text{row}} \mapsto \llparenthesis RegularUser :: \tau' \mid t^{\text{row}} \rrparenthesis]) \\ W Int \sim [w^{\text{row}} \mapsto \llparenthesis RegularUser :: \tau' \mid t^{\text{row}} \rrparenthesis]\tau' : [] \\ [](W r^{\text{row}}) \sim [](W \llparenthesis AdminUser :: String \mid t^{\text{row}} \rrparenthesis) : ? \end{array}}{\llparenthesis RegularUser :: Int \mid r^{\text{row}} \rrparenthesis \sim \llparenthesis AdminUser :: String \mid w^{\text{row}} \rrparenthesis : ?}$$

Now we need to solve for

$$[](W r^{\text{row}}) \sim [](W \llparenthesis AdminUser :: String \mid t^{\text{row}} \rrparenthesis) : ?$$

which reduces to solving for

$$r^{\text{row}} \sim \llparenthesis AdminUser :: String \mid t^{\text{row}} \rrparenthesis : ?$$

which follows immediately from *uni-varl* as

$$r^{\text{row}} \sim \llparenthesis AdminUser :: String \mid t^{\text{row}} \rrparenthesis : [r^{\text{row}} \mapsto \llparenthesis AdminUser :: String \mid t^{\text{row}} \rrparenthesis]$$

yielding the following as our final proof tree for *uni-row.*

$$\frac{\begin{array}{c} \llparenthesis AdminUser :: String \mid w^{\text{row}} \rrparenthesis \simeq \llparenthesis RegularUser :: \tau' \mid \llparenthesis AdminUser :: String \mid t^{\text{row}} \rrparenthesis \rrparenthesis : W \\ \text{tail}(\llparenthesis \rrparenthesis) \notin \text{dom}(W) \\ W Int \sim W\tau' : [\tau' \mapsto Int] \\ [\tau' \mapsto Int](W r^{\text{row}}) \sim [\tau' \mapsto Int](W \llparenthesis AdminUser :: String \mid t^{\text{row}} \rrparenthesis) : [r^{\text{row}} \mapsto \llparenthesis AdminUser :: String \mid t^{\text{row}} \rrparenthesis] \end{array}}{\begin{array}{c} \llparenthesis RegularUser :: Int \mid r^{\text{row}} \rrparenthesis \sim \llparenthesis AdminUser :: String \mid w^{\text{row}} \rrparenthesis : \\ [r^{\text{row}} \mapsto \llparenthesis AdminUser :: String \mid t^{\text{row}} \rrparenthesis, w^{\text{row}} \mapsto \llparenthesis RegularUser :: \tau' \mid t^{\text{row}} \rrparenthesis] \end{array}}$$

This leads to either

$$\langle \, (\!| RegularUser :: Int \mid (\!| AdminUser :: String \mid t^{\text{row}} |\!) \, |\!) \, \rangle$$

or

$$\langle \, (\!| AdminUser :: String \mid (\!| RegularUser :: Int \mid t^{\text{row}} |\!) \, |\!) \, \rangle$$

, depending on which side we choose to use the substitution set. This would ordinarily be a problem, since unification is supposed to give equality, but they are equivalent under Leijen's $\cong$ relation[7].

6. *thisTypeChecksToo*: This one is pretty straightforward. Inference leads us to

$$\langle \, (\!| RegularUser :: Int \mid (\!| AdminUser :: String \mid (\!|\,|\!) \, |\!) \, |\!) \, \rangle$$

. Then we are justified in annotating with the specialized type signature via HM's instantiate rule.

7. *myInt*: Also pretty straightforward, follows a similar path to *thisTypeChecks*. We introduce a fresh type variable δ which will ultimately be the inferred type of *myInt*. An application of *uni-function* leads us to unify

$$\langle \, (\!| AdminUser :: String \mid r^{\text{row}} |\!) \, \rangle \sim \langle \, (\!| RegularUser :: Int \mid (\!| AdminUser :: String \mid (\!|\,|\!) \, |\!) \, |\!) \, \rangle$$

. An application of *uni-row* followed by *row-swap* and *row-var* gives us the substitution set

$$[r^{\text{row}} \mapsto (\!| RegularUser :: \gamma \mid (\!|\,|\!) \, |\!)]$$

which the third condition of *uni-row* then adds $\gamma \mapsto Int$ resulting in the final substitution set

$$[r^{\text{row}} \mapsto (\!| RegularUser :: Int \mid (\!|\,|\!) \, |\!)]$$

. This unifies our function inputs in *uni-function* and our function output results in the substitution set

$$[\delta \mapsto Int]$$

which agrees with our type annotation.

8. *failsToCompile*: This function will be inferred as $\langle \, (\!| AdminUser :: String \mid (\!|\,|\!) \, |\!) \, \rangle \to Int$. If we try to unify *failToCompile*'s type annotation with the inferred type, an application of *uni-function* ultimately will try to unify

$$(\!| AdminUser :: String \mid (\!|\,|\!) \, |\!) \sim (\!| RegularUser :: Int \mid (\!| AdminUser :: String \mid (\!|\,|\!) \, |\!) \, |\!)$$

However, that will fail to unify in *uni-row* on *row-swap*, which will ask us to do the following impossible rewrite.

$$(\!|\,|\!) \simeq (\!| RegularUser :: Int \mid (\!|\,|\!) \, |\!)$$

And that's it folks! Please let me know in the Elm discourse thread if there's anything else you'd like me to show.

[7]In case you're wondering, this is why Leijen mentions " To use our framework with standard Hindley-Milner type rules we need to make the implicitsyntactic equality between mono types explicit with our equality relation defined in Figure 1." and gives the modified *app* example. This is what lets us confidently say that it doesn't matter to the type system at all which order we choose.