Regular expressions of types

Oleg Grenrus
Me

- Oleg Grenrus, @phadej
Me

- Oleg Grenrus, @phadej
- I work at futurice
Me

- Oleg Grenrus, @phadej
- I work at futurice
- I co-organise the HaskHel-meetup 🌊
Me

- Oleg Grenrus, @phadej
- I work at futurice
- I co-organise the HaskHel-meetup 🌊
- I’m the “release manager” of servant 🤖
find - search for files

find [-H|-L] path... [operand_expression...]

is it just stringly typed?

is it

(s/
c at

(link

(s/
alt:

H:

L:

p ath

(s/
∗

]string?

(expr

-expr

expression?

)
find - search for files

find [-H|-L] path... [operand_expression...]
find - search for files

find [-H|-L] path... [operand_expression...]

- is it just stringly typed?
- is it

\[(s/cat :link \ (s/? \ (s/alt :H :H :L :L))
  :path \ (s/* string?)
  :expr \ operand-expression?)\]
Using find

- Shell:
  
  `find -L src -type f`
Using find

- Shell:
  
  ```shell
  find -L src -type f
  ```

- Clojure?

  ```clojure
  (find :L "src" :type :file)
  ```
Using find

- Shell:
  
  ```bash
  find -L src -type f
  ```

- Clojure?

  ```clojure
  (find :L "src" :type :file)
  ```

- Haskell!

  ```haskell
  (find_ #:L, "src", #:type, #:file)
  ```
Types
Lambda calculus

Example rule: introducing (making) a pair (or a product)

\[ \Gamma \vdash x : A \quad \Gamma \vdash y : B \quad \text{Pair} \]

\[ \vdash \text{pair } x \ y : A \times B \]

Let’s walk through the notation.
Lambda calculus

Example rule: introducing (making) a pair (or a product)

\[
\begin{array}{c@{}c@{}c}
\text{Premises} & \text{Rule name} & \text{Conclusion} \\
\hline
\end{array}
\]

We can conclude proposition below the line, if we have proof of propositions on top of the line.
Lambda calculus

Example rule: introducing (making) a pair (or a product)

\[ \Gamma \vdash x : A \]
\[ \vdash y : B \]
\[ \Gamma \vdash \text{pair } xy : A \times B \]

\( \Gamma \) is a context, “what is in the scope”, or “what we know”. For example \( x : \text{Int}, y : \text{Bool} \) - there are \( x \) and \( y \) in scope.
Lambda calculus

Example rule: introducing (making) a pair (or a product)

\[ \Gamma \vdash \text{Pair} \]

\(\vdash\), turnstile, “entails”. \(\Gamma \vdash \ldots\) is pronounced “In the context \(\Gamma \ldots\)”
Lambda calculus

Example rule: introducing (making) a pair (or a product)

\[ \Gamma \vdash \text{Pair} \]

\[ \Gamma \vdash \text{colon, “has type”} \]
Lambda calculus

Example rule: introducing (making) a pair (or a product)

\[ \Gamma \vdash \text{pair } x, y : \]

On the left hand side are expressions, I color them in red.
Lambda calculus

Example rule: introducing (making) a pair (or a product)

\[ \Gamma \vdash \text{Pair} : A \times B \]

On the right hand side are types, I color them blue.
Lambda calculus

Example rule: introducing (making) a pair (or a product)

\[ \Gamma \vdash \text{pair} : A \times B \]

On the left hand side are expressions, I color them in red. On the right hand side are types, I color them blue.
Lambda calculus

Example rule: introducing (making) a pair (or a product)

\[ \Gamma \vdash \text{pair} x y : \mathcal{A} \times \mathcal{B} \]

So...
Lambda calculus

Example rule: introducing (making) a pair (or a product)

\[ \Gamma \vdash \text{Pair} \]

In (any) context \( \Gamma \),
Lambda calculus

Example rule: introducing (making) a pair (or a product)

\[ \Gamma \vdash \text{pair } x \; y : A \times B \]

In (any) context \( \Gamma \), \text{pair } x \; y
Lambda calculus

Example rule: introducing (making) a pair (or a product)

\[ \Gamma \vdash \text{pair } x \; y : A \times B \]

In (any) context \( \Gamma \), \text{pair } x \; y \text{ has type}
Lambda calculus

Example rule: introducing (making) a pair (or a product)

\[ \Gamma \vdash \text{pair } x \; y : A \times B \]

In (any) context \( \Gamma \), \textit{pair } \( x \; y \) has type \( A \times B \);
Lambda calculus

Example rule: introducing (making) a pair (or a product)

\[
\frac{
\Gamma \vdash \ \text{Pair}
}{
\Gamma \vdash \text{pair} \ x \ y : A \times B
}
\]

In (any) context \(\Gamma\), \(\text{pair} \ x \ y\) has type \(A \times B\); \textit{given that in the same context} \(\Gamma\),
Lambda calculus

Example rule: introducing (making) a pair (or a product)

\[
\frac{\Gamma \vdash x : A}{\Gamma \vdash \text{pair } x \ y : A \times B}
\]

Pair

In (any) context $\Gamma$, $\text{pair } x \ y$ has type $A \times B$; given that in the same context $\Gamma$, $x$ has type $A$
Lambda calculus

Example rule: introducing (making) a pair (or a product)

\[
\Gamma 
\vdash x : A \quad \Gamma 
\vdash \text{pair } x \ y : A \times B
\]

In (any) context \(\Gamma\), \(\text{pair } x \ y\) has type \(A \times B\); given that in the same context \(\Gamma\), \(x\) has type \(A\) and
Lambda calculus

Example rule: introducing (making) a pair (or a product)

\[
\Gamma \vdash x : A \quad \Gamma \vdash y : B \\
\Gamma \vdash \text{pair } x \ y : A \times B
\]

In (any) context \(\Gamma\), \(\text{pair } x \ y\) has type \(A \times B\); given that in the same context \(\Gamma\), \(x\) has type \(A\) and \(y\) has type \(B\).
Lambda calculus

Example rule: introducing (making) a pair (or a product)

\[
\frac{\Gamma \vdash x : A \quad \Gamma \vdash y : B}{\Gamma \vdash \text{pair } x \ y : A \times B}
\]

Pair

In (any) context \(\Gamma\), \(\text{pair } x \ y\) has type \(A \times B\); given that in the same context \(\Gamma\), \(x\) has type \(A\) and \(y\) has type \(B\). □
Lambda calculus: more rules

\[ \Gamma, x : A \vdash x : A \]

\[ \frac{\Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash x : A}{\Gamma \vdash f \, x : B} \quad \text{App} \]

\[ \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x : A \rightarrow e : A \rightarrow B} \quad \text{Lam} \]

\[ \frac{\Gamma \vdash x : A \quad \Gamma \vdash y : B}{\Gamma \vdash \text{pair} \, x \, y : A \times B} \quad \text{Pair} \]

\[ \frac{\Gamma \vdash x : A}{\Gamma \vdash \text{inl} \, x : A + B} \quad \text{Left} \]

\[ \frac{\Gamma \vdash y : B}{\Gamma \vdash \text{inr} \, y : A + B} \quad \text{Right} \]

\[ \frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \text{fst} \, p : A} \quad \text{Fst} \]

\[ \frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \text{snd} \, p : B} \quad \text{Snd} \]

\[ \frac{\Gamma \vdash \cdots}{\Gamma \vdash \cdots} \quad \text{Either} \]
Decidability of typed calculi

For the various type systems, we can ask three typical questions.
Decidability of typed calculi

1. Type-checking

\[ \Gamma \vdash x : \alpha \]
Decidability of typed calculi

2. Typability, or type-inference

\[ \begin{array}{c}
? \\
\Gamma \vdash x : ?
\end{array} \]
Decidability of typed calculi

3. Inhabitation, or proof search

\[
\frac{?}{\Gamma \vdash ? : \alpha}
\]
Inhabitation, an example

Really nice feature of the system above is that everything is decidable
In particular: inhabitation.
Inhabitation, an example

\[ \vdash ? : (A \times B \to C) \to (A \to B \to C) \]
Inhabitation, an example

\[
\begin{align*}
\frac{f : A \times B \to C} & \vdash ? : A \to B \to C \\
\vdash \lambda f \to ? : (A \times B \to C) & \to (A \to B \to C)
\end{align*}
\]

Lam
Inhabitation, an example

\[
\begin{align*}
\Gamma \vdash \lambda f : A \times B \to C, x : A, y : B & \\
\Gamma \vdash f : A \times B \to C & \\
\Gamma \vdash \lambda x y \to ? : A \to B \to C & \\
\Gamma \vdash \lambda f x y \to ? : (A \times B \to C) \to (A \to B \to C) & \\
\end{align*}
\]
Inhabitation, an example

\[\Gamma \vdash f : A \times B \rightarrow C\]

\[\Gamma \vdash ? : A \times B\]

\[\Gamma \vdash f : C\]

\[f : A \times B \rightarrow C \vdash \lambda x y \rightarrow f : A \rightarrow B \rightarrow C\]

\[\vdash \lambda f x y \rightarrow f : (A \times B \rightarrow C) \rightarrow (A \rightarrow B \rightarrow C)\]

\[\Gamma = f : A \times B \rightarrow C, x : A, y : B\]
Inhabitation, an example

\[
\begin{align*}
\Gamma & \vdash f : A \times B \rightarrow C \\
\Gamma & \vdash ? : A \\
\Gamma & \vdash ? : B
\end{align*}
\]

\[
\frac{
\begin{align*}
\Gamma & \vdash \text{pair}?? : A \times B \\
\Gamma & \vdash f (\text{pair}??) : C
\end{align*}
}{
\Gamma \vdash \lambda x y \rightarrow f (\text{pair}??) : (A \times B \rightarrow C) \rightarrow (A \rightarrow B \rightarrow C)
}\]

\[\Gamma = f : A \times B \rightarrow C, x : A, y : B\]
Inhabitation, an example

\[
\Gamma \vdash f : A \times B \rightarrow C \\
\Gamma \vdash x : A \\
\Gamma \vdash ? : B \\
\Gamma \vdash \text{pair } x?: A \times B \\
\Gamma \vdash f(\text{pair } x?): C \\
\Gamma = f : A \times B \rightarrow C, x : A, y : B
\]
Inhabitation, an example

\[
\begin{align*}
\Gamma &\vdash f : A \times B \rightarrow C \\
\Gamma &\vdash x : A \\
\Gamma &\vdash y : B \\
\Gamma &\vdash \text{pair}_{xy} : A \times B \\
\Gamma &\vdash f(\text{pair}_{xy}) : C \\
\Gamma &\vdash \lambda x y \rightarrow f(\text{pair}_{xy}) : A \rightarrow B \rightarrow C \\
\Gamma &\vdash \lambda f x y \rightarrow f(\text{pair}_{xy}) : (A \times B \rightarrow C) \rightarrow (A \rightarrow B \rightarrow C)
\end{align*}
\]

\[\Gamma = f : A \times B \rightarrow C, x : A, y : B\]
Regular expressions
Stand back...

Not the Perl or POSIX regular expressions, but “real” regular expressions:

\begin{align*}
\text{regular expression} &::= \varepsilon & \text{empty string} \\
&\mid a & \text{"character", } a \in \Sigma \\
&\mid r_1 \cdot r_2 & \text{concatenation} \\
&\mid r_1 \lor r_2 & \text{alternation} \\
&\mid r_1^* & \text{Kleene closure}
\end{align*}
Stand back...

Not the Perl or POSIX regular expressions, but “real” regular expressions:

\[
\text{regular expression } r ::= \varepsilon \mid a \text{"character"}, a \in \Sigma \mid r_1 \cdot r_2 \text{concatenation} \mid r_1 \lor r_2 \text{alternation} \mid r_1^* \text{Kleene closure}
\]
Stand back...

Not the Perl or POSIX regular expressions, but “real” regular expressions:

regular expression \( r \) ::= \( \varepsilon \) empty string
| \( a \) “character”, \( a \in \Sigma \)
| \( r_1 \diamond r_2 \) concatenation
| \( r_1 \lor r_2 \) alternation
| \( r^* \) Kleene closure
Matching

We formalise the what-matches -intuition using $\Delta \vdash r$ relation, which is read as “regular expression $r$ matches string $\Delta$”.

For example we can ask,

$[x, y, z] \vdash (x \lor y)^* \diamond z$
We formalise the what-matches -intuition using $\Delta \Vdash r$ relation, which is read as “regular expression $r$ matches string $\Delta$".
Matching

We formalise the what-matches-intuition using $\Delta \models r$ relation, which is read as “regular expression $r$ matches string $\Delta$”.

\[
\begin{array}{c}
\models r_1 \\
\models r_1 \diamond r_2 \\
\models r_1 \diamond r_2
\end{array}
\]
Matching

We formalise the what-matches - intuition using $\Delta \models r$ relation, which is read as “regular expression $r$ matches string $\Delta$.”

$$\begin{align*}
\Delta_1 & \models r_1 \\
\models r_1 \diamond r_2 & \text{ Concat}
\end{align*}$$
Matching

We formalise the what-matches -intuition using $\Delta \vdash r$ relation, which is read as “regular expression $r$ matches string $\Delta$”.

$$\frac{\Delta_1 \vdash r_1 \quad \Delta_2 \vdash r_2}{\vdash r_1 \diamond r_2} \text{ Concat}$$
Matching

We formalise the what-matches -intuition using $\Delta \vdash r$ relation, which is read as “regular expression $r$ matches string $\Delta$.”

$$\Delta_1 \vdash r_1 \quad \Delta_2 \vdash r_2 \quad \Delta_1 \bowtie \Delta_2 \vdash r_1 \diamond r_2 \quad \text{Concat}$$
And there are more rules

\[
\begin{align*}
\text{Atom} & : \quad \alpha \in \Sigma \\
& \quad \vdash [\alpha] \alpha \\
\text{Left} & : \quad \Delta \vdash r_1 \\
& \quad \Delta \vdash r_1 \vee r_2 \\
\text{Nil} & : \quad \dashv \vdash r^* \\
\text{Eps} & : \quad \dashv \vdash \varepsilon \\
\text{Right} & : \quad \Delta \vdash r_2 \\
& \quad \Delta \vdash r_1 \vee r_2 \\
\text{Cons} & : \quad \Delta_1 \vdash r \\
& \quad \Delta_2 \vdash r^* \\
& \quad \Delta_1 \uplus \Delta_2 \vdash r^*
\end{align*}
\]
A match

\[
\frac{[x] \vdash x}{[x] \vdash x \lor y} \quad \frac{[y] \vdash y}{[y] \vdash x \lor y} \quad \frac{[y] \vdash (x \lor y)^*}{\cdot \vdash (x \lor y)^*} \quad \frac{[y] \vdash (x \lor y)^*}{\text{Cons}} \quad \frac{[x, y] \vdash (x \lor y)^*}{\text{Cons}} \quad \frac{[z] \vdash z}{[x, y, z] \vdash (x \lor y)^* \diamond z}
\]
REList

\[\Delta_1 \vdash r_1 \quad \Delta_2 \vdash r_2 \quad \Delta_1 \cup \Delta_2 \vdash r_1 \Diamond r_2 \quad \text{Concat} \]

\[\Gamma \vdash x : \text{Int} \quad \Gamma \vdash y : \text{Bool} \quad \Gamma \vdash \text{pair } x \ y : \text{Int } \times \text{Bool} \]

You only need REList! No pair, sums, ordinary lists...
REList

\[
\frac{\Delta_1 \vdash r_1 \quad \Delta_2 \vdash r_2}{\Delta_1 \oplus \Delta_2 \vdash r_1 \diamond r_2} \quad \text{Concat} \quad \frac{\Gamma \vdash x : \text{Int} \quad \Gamma \vdash y : \text{Bool}}{\Gamma \vdash \text{pair } x \ y : \text{Int} \times \text{Bool}}
\]

\[
\frac{\Gamma \vdash x : \text{REList } r_1 \quad \Gamma \vdash y : \text{REList } r_2}{\Gamma \vdash \text{pair } x \ y : \text{REList } (r_1 \diamond r_2)}
\]
RELList

\[
\Delta_1 
\vdash 
\Gamma 
\vdash 
\Delta_2 
\vdash 
\Gamma 
\vdash 
x : \text{Int} 
\Gamma 
\vdash 
y : \text{Bool} 
\Gamma 
\vdash 
\text{pair } x \ y : \text{Int } \times \text{Bool}
\]

\[
\Delta_1 
\vdash 
\Gamma 
\vdash 
\Delta_2 
\vdash 
\Gamma 
\vdash 
x : \text{RELList } r_1 
\Gamma 
\vdash 
y : \text{RELList } r_2 
\Gamma 
\vdash 
\text{pair } x \ y : \text{RELList } (r_1 \diamond r_2)
\]

You only need RELList! No pair, sums, ordinary lists …
What’s the point?

pair (left 1) (cons ’a’ (cons ’b’ nil)) : REList ((Int ∨ Bool) ◇ Char*)
What’s the point?

\[
\text{pair (left 1) (cons ’a’ (cons ’b’ nil)) : REList ((Int } \lor \text{ Bool)} \diamond \text{ Char}^*)}
\]

Nicer expression syntax!

\[
[1, ’a’, ’b’] : \text{REList ((Int } \lor \text{ Bool)} \diamond \text{ Char}^*)
\]
Does it work?

Yes!

\[ r \] is decidable

And I have made a type-checker GHC plugin making these decisions!
Does it work?

Yes!
Does it work?

Yes! $\Delta \vdash r$ is decidable
Does it work?

Yes! \( \Delta \vdash r \) is **decidable**

And I have made a type-checker GHC plugin making these decisions!
find - again

find [-H|-L] path... [operand_expression...]

(-H ∨ -L ∨ ε) String* operand-expression

(s/cat :link (s/? (s/alt :H :H :L :L))
  :path (s/* string?)
  :expr operand-expression?)
Lets put regular expressions into types!

\((-H \lor -L \lor \varepsilon) \diamond \text{String}^* \diamond \text{operand-expression}\)
Lets put regular expressions into types!

\((\neg H \lor \neg L \lor \varepsilon) \diamond \text{String}^* \diamond \text{operand-expression}\)

type FIND

\(= (E \lor \text{Flag "H"} \lor \text{Flag "L"})\)

\(\diamond S \diamond \text{String}\)

\(\diamond \text{OPERAND_EXPRESSION}\)
Lets put regular expressions into types!

\((-H \lor -L \lor \epsilon) \diamond \text{String}^* \diamond \text{operand-expression}\)

type \text{FIND} = (E \lor \text{Flag "H"} \lor \text{Flag "L"})
\diamond S \diamond \text{String}
\diamond \text{OPERAND_EXPRESSION}

\text{find} \_ :: \text{REList \text{FIND}} \rightarrow \text{IO ()}
Conclusion
This all is cool
This all is cool

Which one is “better”?

\[ find_\_
:: \text{RELList\ FIND} \rightarrow \text{IO ()} \]

-- or

\[ find_\_
:: \text{Maybe\ FollowLinks} \rightarrow [\text{Path}] \rightarrow \text{Expr} \rightarrow \text{IO ()} \]
This all is cool

Which one is “better”?

\[
\text{find} \_ \text{:: REList FIND} \to \text{IO ()}
\]

-- or

\[
\text{find} \_ \text{:: Maybe FollowLinks} \to \text{[Path]} \to \text{Expr} \to \text{IO ()}
\]

Missing bit is implementing the rule

\[
\frac{\Delta \vdash r_1 \quad r_1 \leq r_2}{\Delta \vdash r_2} \quad \text{Sub}
\]

so we could use relist syntax but have Haskell types e.g.

\[\text{[1, ’a’, ’b’]} : (\text{Either Int Bool, [Char]})\]
Thank you!
Questions?