4 Liquid types for Elm

4.1 Formal definition of Elm

4.1.1 Syntax

Elm differentiates variables depending on the capitalization of the first letter. For the formal language we define <upper-letter> $\in \mathcal{V}$ for variables with the first letter capitalized and <lower-var>$\in \mathcal{V}$ for variables without.

Syntactically we can build our types from booleans, integers, lists, tuples, records, functions, custom types and type variables.

We will define our syntax in a Backus-Naur-Form [Bac59].

Definition 4.1: Type Signiture Syntax

Let <upper-letter> $\in \mathcal{V}$; <lower-var> $\in \mathcal{V}$.
—

We define the following types:

```
<list-lower-var> ::= "" | <lower-var> | <lower-var> <list-lower-var>

<list-type-fields> ::= "" | <lower-var> ":" <type>
                       |<lower-var> ":" <type> "," <list-type-fields>

         <list-type> ::= "" | <type>  |  <type> <list-type>

               <type> ::="Bool"
                        |"Int"
                        |"List" <type>
                        |"(" <type> "," <type> ")"
                        |"" <list-type-fields> ""
                        |<type> "->" <type>
                        |<upper-var> <list-type>
                        |<lower-var>
```

For matching expressions we allow various pattern.

Definition 4.2: Pattern Syntax

Let <upper-letter> $\in \mathcal{V}$; <lower-var> $\in \mathcal{V}$.
—

We define the following types:

```
     <list-pattern-list> ::= "" | <pattern>
                             | <pattern> "," <list-pattern-list>
```

```
<list-pattern-sort> ::= "" | <pattern> | <pattern> <list-pattern-sort>

    <list-pattern-vars> ::= "" | <lower-var>
                            | <lower-var> "," <list-pattern-vars>

            <pattern> ::= <bool>
                        | <int>
                        | "[" <list-pattern-list> "]"
                        | "(" <pattern> , <pattern> ")"
                        | <upper-var> <list-pattern-sort>
                        | <lower-var>
                        | <pattern> "as" <lower-var>
                        | "{" <list-pattern-vars> "}"
                        | <pattern> "::" <pattern>
                        | "_"
```

Because Elm is a pure functional programming language, a program contains just a single expression.

Definition 4.3: Expression Syntax

Let <upper-letter> $\in \mathcal{V}$; <lower-var> $\in \mathcal{V}$.

—

We define the following types:

```
  <list-exp-field> ::= <lower-var> "=" <exp>
                     | <lower-var> "=" <exp> "," <list-exp-field>

        <maybe-signature> ::= "" | <lower-var> ":" <type> ";"

       <list-case> ::= <pattern> "->" <exp>
                     | <pattern> "->" <exp> ";" <list-case>

                  <bool> ::= "True" | "False"

              <int> ::= "0" | "-1" | "1" | "-2" | "2" | ...

           <list-exp> ::= "" | <exp> | <exp> "," <list-exp>
```

The definition of <exp> can be found in figure 1.

Additionally, Elm also allows global constants, type aliases and custom types.

Definition 4.4: Statement Syntax

Let <upper-letter> $\in \mathcal{V}$; <lower-var> $\in \mathcal{V}$.

—

```
<exp> ::= "foldl"
        | "(::)"
        | "(+)" | "(-)" | "(*)" | "(//)"
        | "(<)" | "(==)"
        | "not" | "(&&)" | "(||)"
        | <exp> "|>" <exp>
        | <exp> ">>" <exp>
        | "if" <exp> "then" <exp> "else" <exp>
        | "" <list-exp-field> ""
        | ""
        | "" <lower-var> "|" <list-exp-field> ""
        | <lower-var> "." <lower-var>
        | "let" <maybe-signature> <lower-var> "=" <exp> "in" <exp>
        | "case" <exp> "of" "[" <list-case> "]"
        | <exp> <exp>
        | <bool>
        | <int>
        | "[" <list-exp> "]"
        | "(" <exp> "," <exp> ")"
        | "\" <pattern> "->" <exp>
        | <upper-var>
        | <lower-var>
```

Figure 1: Syntax for Expressions

We define the following types:

```
<list-sort> ::=<upper-var> <list-type>
              | <upper-var> <list-type> | <list-sort>

<list-statement> ::= "" | <statement> ";"
                     | <statement> ";" <list-statement>

<statement> ::= <maybe-signature> <lower-var> "=" <exp>
              | "type" "alias" <upper-var> <list-lower-var>
                "=" <type>
              | "type" <upper-var> <list-lower-var>
                "=" <upper-var> <list-sort>

<maybe-main-sign> ::= ""|"main" ":" <type> ";"

<program> ::= <list-statement> <maybe-main-sign> "main" "=" <exp>
```

Example 4.1

Using this syntax we can now write a function that reverses a list.

```
reverse : List a -> List a
reverse =
  foldl
    (::)
    [];

main : Int
main =
  case [1,2,3] |> reverse of
  [
    a :: _ ->
      a;
    _ ->
      -1
  ]
```

`foldl` iterates over the list from left to right. It takes the function `(::)`, that appends an element to a list, and the empty list as the starting list. The `main` function reverses the list and returns the first element: 3. Elm requires you also provide return values for other cases that may occur, like the empty list. In that case we just return -1. This will never happened, as long as the reverse function is correctly implemented.

References

[Bac59] John W. Backus. “The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference”. In: *IFIP Congress*. 1959, pp. 125–131.