A. Appendix: Type System written in K Framework

K Framework[RS14] is a system for designing and formalizing programming languages. It uses rewriting systems that are written in its own K Language.

An K file contains modules with the base module name the same as the k file.

```
require "unification.k"
require "elm-syntax.k"

module ELM-TYPESYSTEM
  imports DOMAINS
  imports ELM-SYNTAX

  configuration <k> $PGM:Exp </k>
                <tenv> .Map </tenv>
  //..

  syntax KResult ::= Type
endmodule
```

You can specify the realm upon which the rewriting system can be executed by using the `configuration` keyword. Here we specify two parts: `<k></k>` containing the expression and `<tenv></tenv>` containing the environment.

We also need to specify the end result using the `KResult` keyword. Once the rewriting system reaches the such an expression, it will stop. If not specified the system might not terminate.

The formal grammar can be translated directly into the K language

```
syntax Type
  ::= "bool"
    | "int"
    | "{}Type"
    | "{" ListTypeFields "}Type" [strict]
    | Type "->" Type       [strict,right]
    | LowerVar
    | "(" Type ")"              [bracket]
    | ..
```

Additionally, we can include meta-information: `strict` to ensure the inner expression get evaluated first, `right`/`left` in which direction the expressions should be evaluated and `bracket` for brackets.

Rules are written as rewriting rules instead of inference rules.

```
syntax Exp ::= Type
rule E1:Type E2:Type
  => E1 =Type (E2 -> ?T:Type)
    ~> ?T
syntax KResult ::= Type
```

The rule itself has the syntax `rule . => .`. To ensure the rule gets only executed once all inner expression have been rewritten, we can include an additional `syntax` line before the rule and a `KResult` to ensure that rewriting system keeps on applying rules until a specific result has been reached. Only then it may continue.

Additionally, we have variables starting with an uppercase letter and existentially quantified variables starting with a question mark.

The system itself allows for a more untraditional imperative rewriting system using `~>`. This symbol has only one rule: `rule . ~> A => A` where `.` is the empty symbol. Thus saying, the left part need to be rewritten to `.` before the right part can be changed.

A.1. Implementing Algorithm J

In the original paper by Milner[Mil78] an optimized algorithm is presented for implementing polymorphism in a programming language. This algorithm is imperative but is typically presented as logical rules.

$$\frac{a : T_1 \quad T_2 = \mathit{inst}(T_1)}{\Gamma \vdash_J a : T_2} \qquad \text{[Variable]}$$

$$\frac{\Gamma \vdash_J e_0 : T_0 \quad \Gamma \vdash_J e_1 : T_1 \quad T_2 = \mathit{newvar} \quad \mathit{unify}(T_0, T_1 \to T_2)}{\Gamma \vdash_J e_0 e_1 : T_2} \qquad \text{[Call]}$$

$$\frac{T_1 = \mathit{newvar} \quad \Gamma, x : T_1 \vdash_J e : T_2}{\Gamma \vdash_J \texttt{\textbackslash}x\texttt{->}e : T_0 \to T_1} \qquad \text{[Lambda]}$$

$$\frac{\Delta_1 \vdash_J e_0 : T_1 \quad \Delta_1, a : \text{insert}_{\Delta_1}(\{T_1\}) \vdash_J e_1 : T_2}{\Delta \vdash_J \texttt{let}x\texttt{=}e_0\texttt{in}e_1 : T_2} \qquad \text{[LetIn]}$$

The imperative functions are *newvar*, *unify* and *inst*. *newvar* creates a new variable, *inst* instantiates a type with new variables and *inify* checks whether two types can be unified.

K Framework has a these imperative functions implemented in the `Unification.k` module. In order to use them, we need to first properly define poly types. This way, we only need to compute the bound variables of a type once.

```
syntax PolyType ::= "forall" Set "." Type
```

Next we tell the system that we want to use the unification algorithm on types.

```
syntax Type ::= MetaVariable
```

Once this is set up, we can use the function `#renameMetaKVariables` for *inst* and `?T` for *newvar*.

```
rule <k> variable X:Id => #renameMetaKVariables(T, Tvs) ...</k>
    <tenv>... X |-> forall Tvs . T
    ...</tenv>
```

```
rule <k> fun A:Id -> E:Type => ?T:Type -> E ~> setTenv(TEnv) ...</k>
    <tenv> TEnv:Map => TEnv [ A <- ?T ] </tenv>

syntax KItem ::= setTenv(Map)
  rule <k> T:Type ~> (setTenv(TEnv) => .) ...</k>
   <tenv> _ => TEnv </tenv>
```

Note that the `setTenv` function ensures that ?T is instantiated before its inserted into the environment.

For implementing insert$_\triangle$ we use `#metaKVariables` for getting all bound variables and `#freezeKVariables` to ensure that variables in the environment need to be newly instantiated when ever they get used.

```
rule <k> let X = T:Type in E => E ~> setTenv(TEnv)
    ...</k>
    <tenv> TEnv
      => TEnv[ X
        <- forall (#metaKVariables(T) -Set #metaKVariables(setTenv(TEnv))) .
          ( #freezeKVariables(T, setTenv(TEnv)):>Type
          )
      ]
    </tenv>
```

As for *unify*, we can take advantage of the build-in pattern matching capabilities:

```
syntax KItem ::= Type "=Type" Type
rule T =Type T => .
```

By using a new function =Type with the rewriting rule `rule T =Type T => .` we can force the system to pattern match when ever we need to. Note that if we do not use this trick, the system will think that all existentially quantified variables are type variables and will therefore stop midway.

References

[Mil78] Robin Milner. "A theory of type polymorphism in programming". In: *Journal of Computer and System Sciences* 17 (1978), pp. 348–375.

[RS14] Grigore Rosu and Traian-Florin Serbanuta. "K Overview and SIMPLE Case Study". In: *Electr. Notes Theor. Comput. Sci.* 304 (2014), pp. 3–56. DOI: 10.1016/j.entcs.2014.05.002. URL: https://doi.org/10.1016/j.entcs.2014.05.002.