3.6 Type System Written in K Framework

K Framework[RS14] was created in 2003 by Grigore Rosu. It is a research language and a system of tools for designing and formalizing programming languages. These include tools for parsing, execution, type checking and program verification[Ste+16]. Most of the features of the system are done by rewriting systems that are specified using its programming language called “K Language”.

The main usage besides the creation and formalization of new languages is to create formal languages of existing programming languages. These include C[HER15], Java[BR15], JavaScript[PSR15], Php[FM14], Python[Gut13] and Rust[Kan+18].

The project was developed by the Formal Systems Laboratory Research Group and the University of Illinois, USA. The project itself is open source while the various more specialized tools are published under the Runtime Verification Inc. These include an analysing tool for C called RV-Match that is based on the formal C language written in K language[Gut+16] and more recently a tool for verifying smart contract written for the crypto-coin Ethereum [Hil+18].

We will be using K Framework to express small step semantics of the denotational semantics from the earlier section. We can validate the semantic by letting K Framework apply the rewriting rules upon some examples.

```
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
```

One 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 type context.

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

3.6.1 Implementing the Formal Language

To implement the formale Elm language in K Framework we need to translate the formal grammar 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 gets evaluated first, `right/left` to state in which direction the expressions should be evaluated and `bracket` for brackets that will be replaced with meta level brackets during paring.

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 . => .`. The inner expressions need to be rewritten (into types) for the outer rule can be applied. 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,d the left part needs to be rewritten to `.` before the right part can be changed.

With all of this applied, the type system can infer types by applying rules as long as possible. But this only holds true for mono types. For poly types we need to implement the polymorphism, in particular instantiation and the generalization. The inference rules that we have presented in the section about type inference are not monomorphic and therefore can't be implemented. So in order to implement them we need to modify the slightly.

3.6.2 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:

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

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

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

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

So all we need to do, is to replace the rules of *let in*, *lambda*, *call* and *variable* with the rules above. The imperative functions are *newvar*, *unify* and *inst*:

- *newvar* creates a new variable.
- *inst* instantiates a type with new variables.
- *inify* checks whether two types can be unified.

K Framework has these imperative functions implemented in the `Unification.k` module. In order to use them, we need to first properly define poly types.

```
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 unification we use `#metaKVariables` for getting all bound variables and `#freezeKVariables` to ensure that variables in the environment needs to be newly instantiated whenever 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.

3.6.3 Example

We will now showcase how K-Framework infers types using the following example:

```
let
  model = []
in
(::) 1 model
```

We first need to write the example into a form that K-Framework can parse. Using the following syntax:

```
syntax Exp
    ::= "let" LowerVar "=" Exp "in" Exp               [strict(2)]
      | Exp Exp                                       [left,strict]
      | "[]Exp"
      | "intExp" Int
      | "(::)"
      | "variable" LowerVar
      | ..
```

Translating the program into our K-Framework syntax, this results in the following file.

```
<k>
let
  model = []Exp
in
((::) (intExp 1)) (variable model)
</k>
<tenv> .Map </tenv>
```

Here `.Map` denotes the empty type context. Also note that we have already applied the `left` rule. K-Framework uses this rule in parse-time, so this is just syntax sugar.

K-Framework will now walk through the abstract syntax tree to find the first term it can match. By specifying `strict(2)` we tell the system that `let in` can only be matched once `[]Exp` is rewritten. By appling the rule

```
rule []Exp => list ?A:Type
```

K-Framework obtains the following result.

```
<k>
let
  model = list ?A0:Type
in
((::) (intExp 1)) (variable model)
</k>
<tenv> .Map </tenv>
```

The system remembers the type hole `?A0` and will fill it in as soon as it finds a candidate for it. By using the rule

```
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>
```

the system rewrites the `let in` expression.

```
<k>
((::) (intExp 1)) (variable model)
</k>
<tenv>
  [model <- forall A0 . (list (#freeze(A0))]
</tenv>
```

Note that we have just witnessed generalization: The free variable `?A` of the type got bound resulting in a poly type. These poly types only exist inside the type inference system.

The rule `Exp Exp` is strict, we therefore need to first rewrite `(::) (intExp 1)` and `variable model`. By appling the rules

```
rule (::) => ?A:Type -> ( list ?A ) -> ( list ?A )
rule intExp I:Int => int
```

the left expression can be rewritten.

```
<k>
((?A1:Type -> ( list ?A1 ) -> ( list ?A1 )) int) (variable model)
</k>
```

```
<tenv>
  [model <- forall A0 . (list (#freeze(A0))]
</tenv>
```

We can apply the expression using the rule

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

and by pattern matching we fill in the type hole `?A1` with `int`.

```
<k>
(( list int ) -> ( list int )) (variable model)
</k>
<tenv>
  [model <- forall A0 . (list (#freeze(A0))]
</tenv>
```

Next we need to get `model` out of the type context. By the rule

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

we obtain the following expression.

```
<k>
(( list int ) -> ( list int )) (list ?A2)
</k>
<tenv>
  [model <- forall A0 . (list (#freeze(A0))]
</tenv>
```

Note how the poly type was only used to store the variables that have been frozen. As we take a copy out of the type context, we instantiate the poly type resulting in a new type hole `?A1`.

Finally, we apply the expressions and again fill the type hole `?A2 = int` resulting in in our final expression.

```
<k>
list int
</k>
<tenv>
  [model <- forall A0 . (list (#freeze(A0))]
</tenv>
```

Here the rewriting system terminates, and the inferred type is `list int`.

References

[BR15] Denis Bogdanas and Grigore Roşu. "K-Java: A Complete Semantics of Java". In: *SIGPLAN Not.* 50.1 (Jan. 2015), pp. 445–456. ISSN: 0362-1340.

DOI: 10.1145/2775051.2676982. URL: https://doi.org/10.1145/2775051.2676982.

[FM14] Daniele Filaretti and Sergio Maffeis. “An Executable Formal Semantics of PHP”. In: *ECOOP 2014 – Object-Oriented Programming.* Ed. by Richard Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 567–592. ISBN: 978-3-662-44202-9.

[Gut+16] Dwight Guth et al. “RV-Match: Practical Semantics-Based Program Analysis”. In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I.* Vol. 9779. LNCS. Springer, July 2016, pp. 447–453. DOI: http://dx.doi.org/10.1007/978-3-319-41528-4_24.

[Gut13] Dwight Guth. “A formal semantics of Python 3.3”. In: 2013.

[HER15] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. “Defining the Undefinedness of C”. In: *SIGPLAN Not.* 50.6 (June 2015), pp. 336–345. ISSN: 0362-1340. DOI: 10.1145/2813885.2737979. URL: https://doi.org/10.1145/2813885.2737979.

[Hil+18] Everett Hildenbrandt et al. “KEVM: A Complete Semantics of the Ethereum Virtual Machine”. In: *2018 IEEE 31st Computer Security Foundations Symposium.* IEEE, 2018, pp. 204–217.

[Kan+18] Shuanglong Kan et al. “K-Rust: An Executable Formal Semantics for Rust”. In: *CoRR* abs/1804.07608 (2018). arXiv: 1804.07608. URL: http://arxiv.org/abs/1804.07608.

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

[PSR15] Daejun Park, Andrei Stefănescu, and Grigore Roşu. “KJS: A Complete Formal Semantics of JavaScript”. In: *SIGPLAN Not.* 50.6 (June 2015), pp. 346–356. ISSN: 0362-1340. DOI: 10.1145/2813885.2737991. URL: https://doi.org/10.1145/2813885.2737991.

[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.

[Ste+16] Andrei Stefănescu et al. “Semantics-Based Program Verifiers for All Languages”. In: *SIGPLAN Not.* 51.10 (Oct. 2016), pp. 74–91. ISSN: 0362-1340. DOI: 10.1145/3022671.2984027. URL: https://doi.org/10.1145/3022671.2984027.