Refinement Types for Elm

Second Master Thesis Report

Lucas Payr

31 January 2020

Current State

1. Formal language similar to Elm
    1.1 A formal syntax **(DONE)**
    1.2 A formal type system **(DONE)**
    1.3 A denotational semantic **(DONE)**
    1.4 A small step semantic (using K Framework) **(WORK IN PROGRESS)**
    1.5 Proof that the type system is valid with respect to the semantics.
2. Extension of the formal language with Liquid Types
3. A type checker implementation written in Elm for Elm.

Topics of this Talk

- Quick introduction to K Framework
- Discuss the formal inference rules based on an example
- Compare the implemented type checker in K Framework with the formal type system
- Live demonstration using the given example.

K Framework

- Created in 2003 by Grigore Rosu
- Maintained and developed by the research groups FSL (Illinois,USA) and FMSE (Lasi,Romania).
- Framework for designing and formalizing programming languages.
- Based on Rewriting systems.

K Framework - 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
```

K Framework - Syntax

`syntax` denotes a syntax

- `strict` - Evaluate the inner expression first
- `right/left` - Evaluate left/right expression first
- `bracket` - Notation for Brackets

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

K Framework - Rules

- rules will be executed top to bottom
- `rule . => .` denotes a rewriting rule
- `. ~> .` denotes a concatenation of two processes(KItems)
- `.` denotes the empty process (`rule . ~> A => A`)
- `requires` denotes a precondition to the rule
- `?T` denotes an existentially quantified variable

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

Example for Formally Inferring the Type

```
let
  model = { counter = 0 }
in
{ model | counter = model.counter |> (+) 1 }
```

0. $\Gamma := \varnothing, \Delta := \varnothing$

[Int] 1. $\Gamma, \Delta \vdash \mathtt{0} : \mathit{Int}$

[Record] 2. $\Gamma, \Delta \vdash \{\mathtt{counter} = \mathtt{0}\} : \{\mathit{counter} : \mathit{Int}\}$

[LetIn] 3. $\Delta := \Delta \cup (\mathit{model} \mapsto \{\mathit{counter} : \mathit{Int}\})$

[Call] 4. $\Gamma, \Delta \vdash (+)\ \mathtt{1} : \mathit{Int} \rightarrow \mathit{Int}$

[Getter] 5. $\Gamma, \Delta \vdash \mathtt{model.counter} : \mathit{Int}$

[Pipe] 6. $\Gamma, \Delta \vdash \mathtt{model.counter} \mid > (+)\ \mathtt{1} : \mathit{Int}$

[Setter] 7. $\Gamma, \Delta \vdash \Delta(\mathit{model})'' \supseteq ''\{\mathit{counter} = \mathit{Int}\}$

Formal Inference Rules - (+), Int

$$\frac{}{\Gamma, \Delta \vdash "(+)" : \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int}}$$

```
rule (+)
  => int -> int -> int
```

$$\frac{i : T}{\Gamma, \Delta \vdash i : T} \quad \frac{}{i : \mathit{Int}}$$

```
rule intValue i:Int => int
```

Formal Inference Rules - Call, Pipe

$$\frac{\Gamma, \Delta \vdash e_1 : T_1 \rightarrow T_2 \quad \Gamma, \Delta \vdash e_2 : T_1}{\Gamma, \Delta \vdash e_1\ e_2 : T_2}$$

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

$$\frac{\Gamma, \Delta \vdash e_1 : T_1 \quad \Gamma, \Delta \vdash e_2 : T_1 \rightarrow T_2}{\Gamma, \Delta \vdash e_1\ "| > "\ e_2 : T_2}$$

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

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

Formal Inference Rules - Record

$$\frac{\Gamma, \Delta \vdash \mathit{lef} : \{a_1 : T_1, \ldots, a_n : T_n\}}{\Gamma, \Delta \vdash "\{" \ \mathit{lef} \ "\}" : \{a_1 : T_1, \ldots, a_n : T_n\}}$$

```
rule { LEF:ListTypeFields }Exp => { LEF }Type
```

$$\frac{\Gamma, \Delta \vdash e : T}{\Gamma, \Delta \vdash a \ " = " \ e : \{a : T\}}$$

$$\frac{\Gamma, \Delta \vdash \mathit{lef} : T \quad \Gamma, \Delta \vdash e : T_0 \quad \{a_0 : T_0, \ldots, a_n : T_n\} = T}{\Gamma, \Delta \vdash a_0 \ " = " \ e \ "," \ \mathit{lef} : T}$$

```
syntax ListExpField ::= ListTypeFields
rule A:Id = E:Type , => A : E ,
rule A:Id = E:Type , LEF:ListTypeFields => A : E , LEF
syntax KResult ::= ListTypeFields
```

Formal Inference Rules - LetIn

$$\frac{(a, \_) \notin \Delta \quad \Gamma, \Delta \vdash e_1 : T_1 \quad mes : T_1 \vdash a : T_1 \qquad \Gamma, \text{insert}_\Delta(\{(a, T_1)\}) \vdash e_2 : T_2}{\Gamma, \Delta \vdash \text{"}\mathtt{let}\text{"}\ mes\ a\text{"} = \text{"}\ e_1\ \text{"}\mathtt{in}\text{"}\ e_2 : T_2}$$

```
rule <k> let A:Id = E1:Type in E2:Exp
  => E2
  ...</k>
  <tenv> TEnv:Map => TEnv [ A <- E1 ] </tenv>
```

- Assume $\text{insert}_\Delta(A) := \Delta \cup A$

Formal Inference Rules - Getter

$$\frac{(a_1, \{a_2 : T, \dots\}) \in \Delta}{\Gamma, \Delta \vdash a_1 "." a_2 : T}$$

```
rule <k> A1:Id get A2:Id => getField LTF A2 ...</k>
  <tenv>... A1 |-> { LTF:ListTypeFields }Type ...</tenv>

syntax Map ::= "getField" ListTypeFields Id
  rule getField (A:Id : T:Type ,) A => T
  rule getField (A1:Id : T:Type , LTF:ListTypeFields) A2
    => T
    requires A1 ==K A2
  rule getField (A1:Id : T:Type , LTF:ListTypeFields) A2
    => getField LTF A2
    requires A1 =/=K A2
```

Formal Inference Rules - Setter

$$\frac{\begin{array}{c}\Gamma, \Delta \vdash \mathit{lef} : \{a_1 : T_1, \ldots, a_n : T_n\} \\ \Gamma, \Delta \vdash a \sqsubseteq_\Delta T_0 \quad T_0 = \{a_1 : T_1, \ldots, a_n : T_n, \ldots\}\end{array}}{\Gamma, \Delta \vdash \text{"\{"}\ a\ \text{"|"}\ \mathit{lef}\ \text{"\}"} : T_0}$$

```
rule <k> ( { A:Id | LEF:ListTypeFields } )
  => { LTF }Type
  ...</k>
  <tenv>... A |-> { LTF:ListTypeFields }Type ...</tenv>
  requires (containsFields LTF LEF)
```

- Assume $a \sqsubseteq_\Delta T := (a, T) \in \Delta$
- We skip the definition of `containsFields`

Polymorphism & Demonstration

Polymorphism is not yet implemented

```
let
  model = {list = []}     --forall a.List a
in
{ model | list = [ 1 ] }  --List Int
```

Demonstration

Instantiation

Definition (Instantiation)
Let $\Delta : \mathcal{V} \nrightarrow \mathcal{T}$ be a type context, $T \in \mathcal{T}$ and e be an expression.

Then we define

$$e \sqsubseteq_{\Delta} T :\Leftrightarrow \exists T_0 \in \mathcal{T}.(e, T_0) \in \Delta \wedge T_0 \sqsubseteq T$$

Note that Δ is a partial function and therefore $\Delta(e)$ would only be defined if T_0 exists. If T_0 does not exist, then this predicate will be false.

- Used in the inference rule **[Setter]**

Generalization

Definition (Generalization)
Let Δ_1, Δ_2 be type contexts, $a \in \mathcal{V}$.

We define

$$
\begin{aligned}
\mathsf{insert}_{\Delta_1}(\Delta_2) := \Delta_1 \cup \\
&\{(a, \forall b_1 \ldots \forall b_n . T') \\
&\mid (a, \forall a_1 \ldots \forall a_m . T') \in \Delta_2 \wedge T' \text{ is a mono type} \\
&\wedge \ \{b_1, \ldots, b_n\} = \{b \mid b \in \mathsf{free}(T') \wedge (b, \_) \notin \Delta_2\} \\
&\}
\end{aligned}
$$

- Used in the inference rule **[LetIn]**

Current State

1. Formal language similar to Elm
    1.1 A formal syntax **(DONE)**
    1.2 A formal type system **(DONE)**
    1.3 A denotational semantic **(DONE)**
    1.4 A small step semantic (using K Framework) **(WORK IN PROGRESS)**
    1.5 Proof that the type system is valid with respect to the semantics.
2. Extension of the formal language with Liquid Types
3. A type checker implementation written in Elm for Elm.

Started thesis in July 2019

Expected finish in Summer 2021