Refinement Types for Elm

Second Master Thesis Report

Lucas Payr

31 January 2019

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 implication
- `requires` denotes a precondition to the rule
- ?T deontes an existencially quantified variable

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

Example for Formally Infering the Type

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

$$
\begin{array}{rl}
\textbf{0.} & \Gamma := \varnothing, \Delta := \varnothing \\
\textbf{[Int] 1.} & \Gamma, \Delta \vdash \texttt{0} : \mathit{Int} \\
\textbf{[Record] 2.} & \Gamma, \Delta \vdash \{\texttt{counter} = \texttt{0}\} : \{\mathit{counter} : \mathit{Int}\} \\
\textbf{[LetIn] 3.} & \Delta := \Delta \cup (\mathit{model} \mapsto \{\mathit{counter} : \mathit{Int}\}) \\
\textbf{[Call] 4.} & \Gamma, \Delta \vdash (+)\ \texttt{1} : \mathit{Int} \to \mathit{Int} \\
\textbf{[Getter] 5.} & \Gamma, \Delta \vdash \texttt{model.counter} : \mathit{Int} \\
\textbf{[Pipe] 6.} & \Gamma, \Delta \vdash \texttt{model.counter} \mid > \ (+)\ \texttt{1} : \mathit{Int} \\
\textbf{[Setter] 7.} & \Gamma, \Delta \vdash \Delta(\mathit{model})\text{"} \supseteq \text{"}\{\mathit{counter} = \mathit{Int}\}
\end{array}
$$

Formal Inference Rules - Int, Pipe

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

```
rule i:Bool => bool
```

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

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

//pattern matching
syntax KItem ::= Type "=Type" Type
rule T =Type T => .
```

Formal Inference Rules - Record

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

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

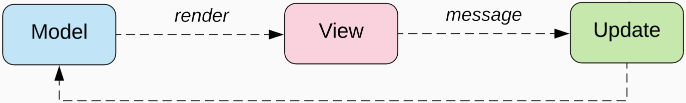
Background: Elm Programming Language

- Invented by Evan Czaplicki as his master-thesis in 2012.
- Goal: Bring Function Programming to Web-Development
- Side-Goal: Learning-friendly design decisions
- Website: elm-lang.org

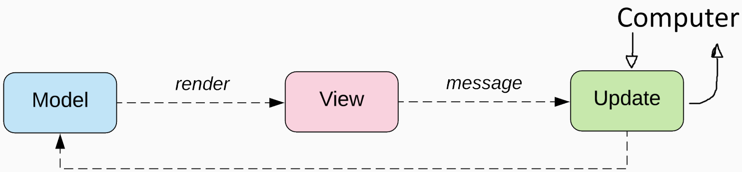
Characteristics

- Pure Functional Language (immutable, no side effect, everything is a function)
- Compiles to JavaScript (in the future also to WebAssembly)
- ML-like Syntax (we say `fun a b c` for $fun(a, b, c)$)
- Simpler than Haskell (no Type classes, no Monads, only one way to do a given thing)
- "No Runtimes errors" (running out of memory, function equality and non-terminating functions still give runtime errors.)

Background: The Elm Architecture



Background: The Elm Architecture



Example

- Online Editor: ellie-app.com

Background: Refinement Types

Restricts the values of an existing type using a predicate.

Initial paper in 1991 by Tim Freeman and Frank Pfenning

- Initial concept was done in ML.
- Allows predicates with only $\wedge, \vee, =$, constants and basic pattern matching.
- Operates over algebraic types.
- Needed to specify **explicitly** all possible Values.

Example

$$\{a : (Bool, Bool) |\ a = (True, False) \vee a = (False, True)\}$$

$$\forall t.\{a : List\ \ t | a = Cons\ (b : t)\ (c : List\ \ t) \wedge c = Cons\ (d : t)\ [\,]\}$$

Background: Liquid Types

Liquid Types (Logically Quantified Data Types) introduced in 2008

- Invented by Patrick Rondan, Ming Kawaguchi and Ranji Jhala
- Initial concept done in OCaml. Later also C, Haskell and TypeScript.
- Operates over Integers and Booleans. Later also Tuples and Functions.
- Allows predicates with logical operators, comparisons and addition.

Example

$$\{(a : Bool, b : Bool) | (a \vee b) \wedge \neg(a \wedge b)\}$$

$$\{(a : Int, b : Int) | a \leq b\}$$

Goals of Thesis

1. Formal language similar to Elm
    1.1 A formal syntax
    1.2 A formal type system
    1.3 A denotational semantic
    1.4 A small step semantic (using K Framework) for rapid prototyping the language
    1.5 Proof that the type system is valid with respect to the semantics.
2. Extension of the formal language with Liquid Types
    2.1 A formal syntax
    2.2 A formal type system
    2.3 A denotational semantic
    2.4 A small step semantic (using K Framework) for rapid prototyping the type checker
    2.5 Proof that the extension infers the correct types.
3. A type checker implementation written in Elm for Elm.

Problems Addressed by the Type System

- Division by zero errors
- Off by one errors
- Proving the correctness of very simple programs
- Clearer interfaces

Theory: Formalization of the Elm Type System

We will use the Hindley-Milner type system (used in ML, Haskell and Elm)

We say

$$
\begin{aligned}
T \text{ is a } \textit{mono type} :\Leftrightarrow\ & T \text{ is a type variable} \\
\lor\ & T \text{ is a type application} \\
\lor\ & T \text{ is a algebraic type} \\
\lor\ & T \text{ is a product type} \\
\lor\ & T \text{ is a function type} \\
T \text{ is a } \textit{poly type} :\Leftrightarrow\ & T = \forall a. T' \\
& \text{where } T' \text{ is a mono type or poly type} \\
& \text{and } a \text{ is a symbol} \\
T \text{ is a } \textit{type} :\Leftrightarrow\ & T \text{ is a mono type } \lor\ T \text{ is a poly type.}
\end{aligned}
$$

Example

1. $Nat ::= \mu C.1 \mid Succ\ C$
2. $List = \forall a.\mu C.Empty \mid Cons\ a\ C$
3. $\text{splitAt} : \forall a.Nat \rightarrow List\ a \rightarrow (List\ a, List\ a)$

Theory: Formalization of the Elm Type System

The *values* of a type is the set corresponding to the type:

$$\text{values}(Nat) = \{1, Succ\ 1, Succ\ Succ\ 1, \ldots\}$$

$$\text{values}(List\ Nat) = \bigcup_{n \in \mathbb{N}} \text{values}_n(List\ Nat)$$

$$\text{values}_0(List\ Nat) = \{[\ ]\}$$

$$\text{values}_n(List\ Nat) =$$

$$\{[\ ]\} \cup \{Cons\ a\ b | a \in \text{values}(Nat), b \in \text{values}_{n-1}(List\ Nat)\}$$

Theory: Definition of Liquid Types

Definition (Sketch)
Let T be a Type Application of *Int*, tuples and functions. Let q be a logical formula consisting of

- Logical operations: $\neg, \wedge, \vee$
- Logical constants: *True*, *False*
- Comparisons: $<, \leq, =, \neq$
- Integer operations: $+, \cdot c$ where c is a constant
- Integer constants: $0, 3, 42, \ldots$
- Bound variables: $a, b, c, \ldots$

Then we call the syntactic phrase $\{a : T | q(a)\}$ a *Liquid Type*.

Theory: Definition of Liquid Types

Example
Let $Nat = \{a : Int | a > 0\}$ in

$$\{\ \{(a : Nat, b : Nat)\ | a + b < 42\} \to \{(c : Nat, d : Nat) | c \leq d\}$$
$$|\ (a = c \wedge b = d) \vee (b = c \wedge a = d)$$
$$\}$$

Theory: Revisiting the Problems

- Division by zero errors

$$(/) : Int \to \{a : Int | a \neq 0\} \to Int$$

- Off by one errors

$$\text{Let } Pos = \{a : Int | 0 \leq a \wedge a < 8\} \text{ in}$$
$$get : (Pos, Pos) \to Chessboard \to Maybe\ Figure$$

- Proving the correctness of very simple programs

$$\text{swap} : \{(a : Int, b : Int) \to (c : Int, d : Int) | b = c \wedge a = d\}$$

- Clearer interfaces

$$\text{length} : List\ a \to \{a : Int | a \geq 0\}$$

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 **(WORK IN PROGRESS)**
    1.4 A small step semantic (using K Framework) for rapid prototyping the language
    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 at the end of 2020