Refinement Types for Elm

Master Thesis Report

Lucas Payr

30 Oktober 2019

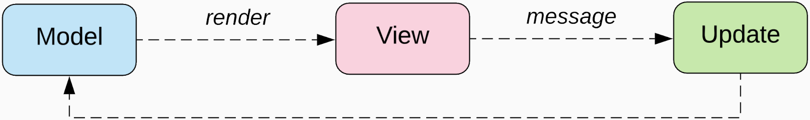
Background: Elm Programming Language

- 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



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: Refinement 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
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
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

Formalization of the Elm Type System

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

We say

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

Example

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

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

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

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

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

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

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*.

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

$$
\begin{aligned}
&\{\ \{(a : Nat, b : Nat)\ | a + b < 42\} \rightarrow \{(c : Nat, d : Nat) | c \leq d\} \\
&|\ (a = c \wedge b = d) \vee (b = c \wedge a = d) \\
&\}
\end{aligned}
$$

Revisiting the Problems

- Division by zero errors

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

- Off by one errors

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

- Proving the correctness of very simple programs

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

- Clearer interfaces

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

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 of 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.