Refinement Types for Elm

Master Thesis Report

Lucas Payr

30 Oktober 2019

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

Characteristics

- Pure Functional Language (immutable, no side effect, everything is a function)
- Statically typed
- 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.)

Introduction to Type Theory

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. $\text{splitAt} : \forall a.Nat \to List\ a \to (List\ a, List\ a)$

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

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

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

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

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

Refinement Types

- Restricts the values of an existing type
- Liquid Types (Logically Quantified Data Types)

Definition (Liquid Types)
Let T be a Type Application of *Int*, tuples and functions. Let q be a predicate 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 $0, 3, 42, \dots$
- Bound variables $a, b, c, \dots$

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

Refinement Types

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

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

Solvable Problems

- Division by zero errors

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

- Off by one errors

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

- Proving the correctness of very simple programs

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

- Clearer Interfaces

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

Goal and Current State

1. A formal syntax. **(DONE)**
2. A formal type system. **(WORK IN PROGRESS)**
3. A high level denotational semantic.
4. A proof, using the denotational semantics, that the type system rules out runtime errors.
5. A low level small step semantic (with the help of K Framework).
6. A type checker for Elm

Started thesis in July 2019

Expected finish at the end of 2020