Refinement Types for Elm

Master Thesis Report

Lucas Payr

13 April 2021

Topics of this Talk

- Background
- Formal Language Similar to Elm
- Extension of the Formal Language
- Demonstration

Background: Introduction to Elm

- Invented by Evan Czaplicki as his master-thesis in 2012.
- 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 (Functions are curried)
- 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)

Background: Introduction to Elm

Example

```
max =
  \a -> \b ->
    if
      (<) a b
    then
      b
    else
      a
```

Background: Introduction to Refinement Types

Restricts the values of an existing type using a predicate (refinement).

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

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

Refinement: $\nu = \mathit{Cons}\ (b : t)\ (c : \mathit{List}\ t) \wedge c = \mathit{Cons}\ (d : t)\ [\,]$

Background: Introduction to 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, Tuples and Functions.
- Allows predicates with logical operators and linear arithmetic.

Example

$$a :\{\nu : Int \mid 0 \leq \nu\} \rightarrow b : \{\nu : Int \mid 0 \leq \nu\}$$
$$\rightarrow \{\nu : Int \mid 0 \leq \nu \ \wedge \ a \leq \nu \ \wedge \ b \leq \nu\}$$

- Liquid Type Variables : a, b
- Refinements: $0 \leq \nu$ and $0 \leq \nu \ \wedge \ a \leq \nu \ \wedge \ b \leq \nu$

Background: Motivation

- Catching Division by zero in compile time

$$(//) : \mathit{Int} \to \{\nu : \mathit{Int} \mid \neg(\nu = 0)\} \to \mathit{Int}$$

- Catching index-out-of-bounds errors in compile time

$$\mathit{get} : \mathit{Array}\ \mathit{Int} \to \{\nu : \mathit{Int} \mid 0 \leq \nu \wedge \nu < 5\} \to \mathit{Int}$$

- Having natural numbers as a subtype of integers

$$\text{type alias } \mathit{nat} = \{\nu : \mathit{Int} \mid 0 \leq \nu\}$$

Background: Goals of Thesis

1. Formal language similar to Elm
   - Formal syntax
   - Formal type system
   - Denotational semantics
   - Proof that the type system is sound with respect to the semantics.
   - Small step semantics (using K Framework) for rapid prototyping of the language
2. Extension of the formal language with Liquid Types
   - Extending the formal syntax, formal type system and denotational semantic
   - Proof that the extension infers the correct types.
   - Implementation (of the core algorithm) written in Elm for Elm.

Formal Language Similar to Elm

Formal syntax

$$
\begin{aligned}
< \texttt{exp} > \; ::= \; & \texttt{"if"} < \texttt{exp} > \texttt{"then"} < \texttt{exp} > \texttt{"else"} < \texttt{exp} > \\
\mid \; & \ldots
\end{aligned}
$$

Formal Type System

$$
\frac{\Gamma, \Delta \vdash e_1 : Bool \quad \Gamma, \Delta \vdash e_2 : T \quad \Gamma, \Delta \vdash e_3 : T}{\Gamma, \Delta \vdash \texttt{"if"}\ e_1\ \texttt{"then"}\ e_2\ \texttt{"else"}\ e_3 : T}
$$

Judgment: $\Gamma, \Delta \vdash e : T$ (e has the type T with respect to Γ and Δ)

Formal Language Similar to Elm

Denotational Semantics

$$\left[\left[\begin{matrix}\text{"}\texttt{if}\text{"}\ e_1\ \text{"}\texttt{then}\text{"} \\ e_2\ \text{"}\texttt{else}\text{"}\ e_3\end{matrix}\right]\right]_{\Gamma,\Delta} = \begin{cases}[[e_2]]_{\Gamma,\Delta} & \text{if } b \\ [[e_3]]_{\Gamma,\Delta} & \text{if } \neg b\end{cases}$$

$$\text{with } [[e_1]]_{\Gamma,\Delta} = b$$

$$\text{where } b \in \text{value}(Bool)$$

Theorem (Soundness of <exp>)
Γ, Δ be type contexts, Δ' be a variable context similar to Δ with respect to Γ. Let $e \in <\texttt{exp}>$ and $T \in \mathcal{T}$. Assume $\Delta, \Gamma \vdash e : T$ can be derived.

Then $[[e]]_{\Gamma,\Delta'} \in \text{value}_\Gamma(\overline{\Gamma}(T))$.

Extension of the Formal Language

Extending the Formal Syntax

$$
\begin{aligned}
&< \texttt{liquid} - \texttt{type} > ::= \\
&\quad "\{\texttt{v} : \texttt{Int}|" < \texttt{qualifier} - \texttt{type} > "\}" \\
&\quad | < \texttt{lower} - \texttt{var} > " : \{\texttt{v} : \texttt{Int}|" < \texttt{qualifier} - \texttt{type} > \\
&\qquad "- > " < \texttt{liquid} - \texttt{type} >
\end{aligned}
$$

Extension of the Formal Language

Formal Type System

$$\frac{\Gamma, \Delta, \Theta, \Lambda \vdash e_1 : \mathit{Bool} \quad e_1 : e_1' \qquad \Gamma, \Delta, \Theta, \Lambda \cup \{e_1'\} \vdash e_2 : T \quad \Gamma, \Delta, \Theta, \Lambda \cup \{\neg e_1'\} \vdash e_3 : T}{\Gamma, \Delta, \Theta, \Lambda \vdash \text{"}\mathtt{if}\text{"}\ e_1\ \text{"}\mathtt{then}\text{"}\ e_2\ \text{"}\mathtt{else}\text{"}\ e_3 : T}$$

$$\frac{\Gamma, \Delta, \Theta, \Lambda \vdash e : T_1 \quad T_1 <:_{\Theta,\Lambda} T_2 \quad \mathrm{wellFormed}(T_2, \Theta)}{\Gamma, \Delta, \Theta, \Lambda \vdash e : T_2}$$

Judgment: $\Gamma, \Delta, \Theta, \Lambda \vdash e : T$ (e has the type T with respect to Γ,Δ, Θ and Λ)

Extension of the Formal Language

Theorem (Soundness of Liquid Types)
Γ, Δ be type contexts, Δ' be a variable context similar to Δ with respect to Γ. Let $\Lambda \subset \mathcal{Q}$ and $\Theta : \mathcal{V} \nrightarrow \mathcal{Q}$. Let $e \in < \mathtt{exp} >$ and $T \in \mathcal{T}$. Assume $\Gamma, \Delta, \Theta, \Lambda \vdash e : T$ can be derived.

Then $[[e]]_{\Gamma,\Delta'} \in \mathrm{value}_{\Gamma}(\overline{\Gamma}(T))$.

Inference Algorithm for Liquid Types

$$\mathsf{Infer} : \mathcal{P}(\mathcal{C}) \to \ (\mathcal{K} \nrightarrow \mathcal{Q})$$

$$\mathsf{Infer}(C) =$$

$$\text{Let } V := \bigcup_{T_1 <:_{\Theta,\Lambda} T_2 \in C} \{a \mid (a, \_) \in \Theta\}$$

$$Q_0 := \mathit{Init}(V),$$

$$A_0 := \{(\kappa, Q_0) \mid \kappa \in \bigcup_{c \in C} \mathsf{Var}(c)\},$$

$$A := \mathsf{Solve}(\bigcup_{c \in C} \mathsf{Split}(c), A_0)$$

$$\text{in } \{(\kappa, \bigwedge Q) \mid (\kappa, Q) \in A\}$$

where $V \subseteq \mathcal{V}, Q_0, Q \subseteq \mathcal{Q}, A_0, A \in \mathcal{K} \nrightarrow \mathcal{Q}, \Theta$ is a type variable context and $\Lambda \subseteq \mathcal{Q}$.

Inference Algorithm for Liquid Types

1. (Split) Split the subtyping conditions over dependent function into subtyping conditions over simple liquid types.
2. (Init) Compute $Q = \mathit{Init}(V)$ where V is the set of all occurring variables and initiate the mapping A for very key κ_i with the set of resulting predicates with Q.
3. (Solve) Check for very subtyping condition if the current mapping A violates the subtyping condition. (SMT statement is satisfiable)
4. (Weaken) If so, weaken the mapping by removing any predicate that violates the subtyping condition (SMT statement is not satisfiable) and repeat
5. Once the algorithm terminates we have obtained the strongest refinements that can be build by conjunction over predicates in $\mathit{Init}(V)$.

Inference Algorithm for Liquid Types

Theory (Verification) - Part 1
$C \subseteq \mathcal{C}^-$ be a set of well-formed conditions, $A_1, A_2 : \mathcal{K} \nrightarrow \mathcal{Q}$ and $V := \bigcup_{T_1 <:_{\Theta,\Lambda} T_2 \in C} \{a \mid (a, \_) \in \Theta\}$. Let for all $a \in V$, $A_1(a)$ be well-defined. Let $A_2 = \text{Solve}(C, A_1)$ and $S = \{(\kappa, \bigwedge Q) \mid (\kappa, Q) \in A_2\}$.

Then for every $a \in V$, $A_2(a) \subseteq A_1(a)$.

Inference Algorithm for Liquid Types

Theory (Verification) - Part 2
For every subtyping condition $(T_1 <:_{\Theta,\Lambda} T_2) \in C$, let

$$\begin{aligned}\Theta' := \{\ &(a, r) \\ &\mid r \text{ has the form } q \wedge (a, q) \in \Theta \wedge q \in \mathcal{Q} \\ &\vee\ r \text{ has the form } [[k]_S]_{S_0} \wedge (a, q) \in \Theta \\ &\qquad \wedge\ q \text{ has the form } [k]_{S_0} \wedge k \in \mathcal{K} \wedge S_0 \in \mathcal{V} \nrightarrow \mathit{IntExp}\}\end{aligned}$$

and $\{(b_1, r_1'), \ldots, (b_n, r_n')\} = \Theta'$.

Theory (Verification) - Part 3
We then have the following correctness property.

$$
\begin{aligned}
&[T_1]_S \in \mathcal{T} \wedge [T_2]_S \in \mathcal{T} \\
&\wedge\ [T_1]_S <:_{\Theta',\Lambda} [T_2]_S \\
&\wedge\ \forall S' \in (\mathcal{V} \to \mathcal{Q}).(\forall a \in V.\exists Q \in \mathcal{P}(A_1(a)).S'(a) = \bigwedge Q) \\
&\quad \wedge [T_1]_{S'} \in \mathcal{T} \wedge [T_2]_{S'} \in \mathcal{T} \\
&\quad \wedge ([T_1]_{S'} <:_{\Theta',\Lambda} [T_2]_{S'} \\
&\qquad \Rightarrow \forall a \in V.\forall \nu \in \mathbb{Z}. \\
&\qquad\quad \forall i_1 \in \mathrm{value}_\Gamma(\{\nu : \mathit{Int} \mid r_1'\}).\ldots.\forall i_n \in \mathrm{value}_\Gamma(\{\nu : \mathit{Int} \mid r_n'\}). \\
&\qquad\qquad [[S(a)]]_{\{(\nu,z),(b_1,i_1),\ldots,(b_n,i_n)\}} \Rightarrow [[S'(a)]]_{\{(\nu,z),(b_1,i_1),\ldots,(b_n,i_n)\}})
\end{aligned}
$$

Conclusion: The Good

- Can catch index-out-of-bounds errors in compile time
- Can catch (some) division by zero errors in compile time
- Can define the natural numbers as a subtype of the integers.

Conclusion: The Bad

Liquid Types have three weaknesses:

- Capabilities of liquid types depend on the initial set of predicates $Init(V)$.
- Increasing the size of $Init(V)$ increases the computation time by a quadratic amount.
- The type system is no longer complete (Not every liquid type can be checked using a type checker).

Liquid Haskell

- Uses a specific initial set $init(V)$ tailored to a specific use-case.
- Developed in Haskell (Not in Elm) thus its faster.

Conclusion: The Ugly

The following code can not be checked in Liquid Haskell.

```
fun : {v:Int | 0 <= v && v*v <= 4} -> {v:Int | v*v <= v+v}
fun =
  \x -> x
```

Liquid Haskell returns the following Error:

```
Error: Liquid Type Mismatch
Inferred type
  {v:Int | v = x }
not a subtype of Required type
  {VV:Int | VV * VV <= VV + VV}
In Context
  x:{v:Int | 0 <= v && v * v <= 4 }
```

Conclusion: The Ugly

The following can be checked in Liquid Haskell.

```
fun : {v:Int | 0 <= v && v*v <= 4} -> {v:Int | v*v <= v+v}
fun =
  \x ->
    if v <= 2 then
      x
    else
      0 --dead branch
```

Conclusion: The Ugly

- The user needs to know about the inner workings of the type checker.
- Liquid types are an overkill for the use cases of Elm.

I therefore come to the conclusion, that liquid types are not a proper fit for Elm.

- LiquidHaskell has the same problems, but targets more the academic world.
- Main target of Elm: Javascript programmers.