

Refinement Types for Elm

Master Thesis

Lucas Payr

Background: Introduction to Elm

- Invented by Evan Czaplicki as his master-thesis in 2012.
- Pure Functional Language
- Compiles to JavaScript
- ML-like Syntax
- Simpler than Haskell
- Website: elm-lang.org

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).

Liquid Types (Logically Quantified Data Types) introduced in 2008

- Invented by Patrick Rondon, 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

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

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

Background: Motivation

- Catching Division by zero in compile time

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

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

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

- Having natural numbers as a subtype of integers

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

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

$$\begin{aligned} \langle \text{exp} \rangle &::= \text{"if"} \langle \text{exp} \rangle \text{"then"} \langle \text{exp} \rangle \text{"else"} \langle \text{exp} \rangle \\ &\quad | \dots \end{aligned}$$

Formal Type System

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

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

Denotational Semantics

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

with $[[e_1]]_{\Gamma, \Delta} = b$
where $b \in \text{value}(\text{Bool})$

Theorem (Soundness of $\langle \text{exp} \rangle$)

Γ, Δ be type contexts, Δ' be a variable context similar to Δ with respect to Γ . Let $e \in \langle \text{exp} \rangle$ and $T \in \mathcal{T}$. Assume $\Delta, \Gamma \vdash e : T$ can be derived.

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

Extending the Formal Syntax

```
< liquid - type > ::=  
    "{v : Int|" < qualifier - type > "}"  
    | < lower - var > " : {v : Int|" < qualifier - type >  
        "- > " < liquid - type >
```

Extension of the Formal Language

Formal Type System

$$\begin{array}{c} \Gamma, \Delta, \Theta, \Lambda \vdash e_1 : Bool \quad e_1 : e'_1 \\ \Gamma, \Delta, \Theta, \Lambda \cup \{e'_1\} \vdash e_2 : T \quad \Gamma, \Delta, \Theta, \Lambda \cup \{\neg e'_1\} \vdash e_3 : T \\ \hline \Gamma, \Delta, \Theta, \Lambda \vdash \text{"if" } e_1 \text{"then" } e_2 \text{"else" } e_3 : T \\ \Gamma, \Delta, \Theta, \Lambda \vdash e : T_1 \quad T_1 <_{\Theta, \Lambda} T_2 \quad \text{wellFormed}(T_2, \Theta) \\ \hline \Gamma, \Delta, \Theta, \Lambda \vdash e : T_2 \end{array}$$

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

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 $\Theta : \mathcal{V} \rightarrow \mathcal{Q}$. Let $e \in \langle \text{exp} \rangle$ and $T \in \mathcal{T}$. Assume $\Gamma, \Delta, \Theta, \Lambda \vdash e : T$ can be derived.

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

Inference Algorithm for Liquid Types

$\text{Infer} : \mathcal{P}(\mathcal{C}) \rightarrow (\mathcal{K} \multimap \mathcal{Q})$

$\text{Infer}(C) =$

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

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

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

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

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

where $V \subseteq \mathcal{V}, Q_0, Q \subseteq \mathcal{Q}, A_0, A \in \mathcal{K} \multimap \mathcal{Q}, \Theta$ 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 $Q = \text{Init}(V)$ where V is the set of all occurring variables and initiate the mapping A for every key κ_i with the set of resulting predicates with Q .
3. (Solve) Check for every 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 $\text{Init}(V)$.

Theory (Verification) - Part 1

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

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

Theory (Verification) - Part 2

For every subtyping condition $(T_1 <_{:\Theta, \wedge} T_2) \in C$, let

$$\Theta' := \{ (a, r)$$

$$| r \text{ has the form } q \wedge (a, q) \in \Theta \wedge q \in \mathcal{Q}$$

$$\vee r \text{ has the form } [[k]_S]_{S_0} \wedge (a, q) \in \Theta$$

$$\wedge q \text{ has the form } [k]_{S_0} \wedge k \in \mathcal{K} \wedge S_0 \in \mathcal{V} \not\rightarrow IntExp \}$$

and $\{(b_1, r'_1), \dots, (b_n, r'_n)\} = \Theta'$.

Inference Algorithm for Liquid Types

Theory (Verification) - Part 3

We then have the following correctness property.

$$\begin{aligned} & [T_1]_s \in \mathcal{T} \wedge [T_2]_s \in \mathcal{T} \\ & \wedge [T_1]_s <_{:\Theta', \wedge} [T_2]_s \\ & \wedge \forall S' \in (\mathcal{V} \rightarrow \mathcal{Q}). (\forall a \in V. \exists Q \in \mathcal{P}(A_1(a)). S'(a) = \bigwedge Q) \\ & \wedge [T_1]_{s'} \in \mathcal{T} \wedge [T_2]_{s'} \in \mathcal{T} \\ & \wedge ([T_1]_{s'} <_{:\Theta', \wedge} [T_2]_{s'}) \\ & \Rightarrow \forall a \in V. \forall \nu \in \mathbb{Z}. \\ & \quad \forall i_1 \in \text{valuer}(\{\nu : \text{Int} \mid r'_1\}). \dots \forall i_n \in \text{valuer}(\{\nu : \text{Int} \mid r'_n\}). \\ & \quad [[S(a)]]_{\{(\nu, z), (b_1, i_1), \dots, (b_n, i_n)\}} \Rightarrow [[S'(a)]]_{\{(\nu, z), (b_1, i_1), \dots, (b_n, i_n)\}} \end{aligned}$$

Demonstration

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.