Refinement Types for Elm

Master Thesis

Lucas Payr

Background: Introduction to Elm

- Invented by Evan Czaplicki as in his master thesis in 2012.
- Pure Functional Language
- Compiles to JavaScript
- Based on Haskell
- Website: elm-lang.org

```
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 Rondan, Ming Kawaguchi and Ranji Jhala
- Initial concept done in OCaml. Later also Haskell.
- Operates over Integers and Booleans.
- Allows predicates with logical operators and linear arithmetic.

Example

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

Background: Motivation

- Catching Division by zero at compile time

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

- Catching index-out-of-bounds errors at 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 definition of a 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 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, by using Linear Arithmetic in the external SMT Solver Z3

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)

Type contexts: Γ contains type aliases, Δ contains types of variables.

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

Proof: See thesis.

Extension of the Formal Language

Extending the Formal Syntax

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

Extension of the Formal Language

Formal Type System

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

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

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

Θ contains the refinements of liquid type variables, Λ contains if-conditions.

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 $e \in < \texttt{exp} >$ and $T \in \mathcal{T}$. Assume $\Gamma, \Delta, \Theta, \Lambda \vdash e : T$ can be derived.

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

Proof: See thesis.

Inference Algorithm for Liquid Types

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

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

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

$$Q_0 := Init(V),$$

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

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

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

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

$\mathrm{Solve}(C, A) =$

Let $S := \{(k, \bigwedge Q) \mid (k, Q) \in A\}$.

If there exists $(\{\nu : Int \mid q_1\} <:_{\Theta,\Lambda} \{\nu : Int \mid [k_2]_{S_2}\}) \in C$ such that

$\neg(\forall z \in \mathbb{Z}.\forall i_1 \in \mathrm{value}_\Gamma(\{\nu : Int \mid r'_1\}). \ldots \forall i_n \in \mathrm{value}_\Gamma(\{\nu : Int \mid r'_n\}).$

$[[r_1 \wedge p]]_{\{(\nu,z),(b_1,i_1),\ldots,(b_n,i_n)\}} \Rightarrow [[r_2]]_{\{(\nu,z),(b_1,i_1),\ldots,(b_n,i_n)\}})$

for $r_2 := \bigwedge [S(\kappa_2)]_{S_2}$, $p := \bigwedge \Lambda$,

$$r_1 := \begin{cases} \bigwedge [S(k_1)]_{S_1} & \text{if } q_1 \text{ has the form } [k_1]_{S_1} \text{ for } k \in \mathcal{K} \text{ and} \\ & S_1 \in \mathcal{V} \nrightarrow IntExp \\ q_1 & \text{if } q_1 \in \mathcal{Q} \end{cases},$$

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

$\{(b_1, r'_1), \ldots, (b_n, r'_n)\} = \Theta'$

then $\mathrm{Solve}(C, \mathrm{Weaken}(c, A))$ else A

where $k, k_2 \in \mathcal{K}, S : \mathcal{K} \nrightarrow \mathcal{Q}, Q, \Lambda \subseteq \mathcal{Q}, S_2 : \mathcal{V} \nrightarrow \mathit{IntExp}, q_1 \in \mathcal{K} \cup \mathcal{Q}$, Θ be a type variable context, $r_1, p, r_2 \in \mathcal{Q}, a \in \mathcal{V}, \Theta' : \mathcal{V} \nrightarrow \mathcal{Q}, r \in \mathcal{Q}$, $n \in \mathbb{N}, b_i \in \mathcal{V}, r_i \in \mathcal{Q}$ for $i \in \mathbb{N}_0^n$ and $[t]_A$ denotes the substitution for the term t with a substitution A.

Inference Algorithm for Liquid Types

we can use an SMT solver to validate

$$\neg(\forall z \in \mathbb{Z}. \forall i_1 \in \mathsf{value}_\Gamma(\{\nu : \mathit{Int} \mid r_1'\}). \ldots \forall i_n \in \mathsf{value}_\Gamma(\{\nu : \mathit{Int} \mid r_n'\}).$$
$$[[r_1 \wedge p]]_{\{(\nu,z),(b_1,i_1),\ldots,(b_n,i_n)\}} \Rightarrow [[r_2]]_{\{(\nu,z),(b_1,i_1),\ldots,(b_n,i_n)\}})$$

by deciding the satisfiablity of

$$((\bigwedge_{j=0}^{n} [r_j']_{\{(\nu,b_j)\}}) \wedge r_1 \wedge p) \wedge \neg r_2$$

with free variables $\nu \in \mathbb{Z}$ and $b_i \in \mathbb{Z}$ for $i \in \mathbb{N}_1^n$.

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

Theorem (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 $a \in V$, $A_2(a) \subseteq A_1(a)$.

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

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

$$
\begin{aligned}
&[T_1]_S \in \mathcal{T} \land [T_2]_S \in \mathcal{T} \\
&\land\ [T_1]_S <:_{\Theta',\Lambda} [T_2]_S \\
&\land\ \forall S' \in (\mathcal{V} \to \mathcal{Q}).(\forall a \in V.\exists Q \in \mathcal{P}(A_1(a)).S'(a) = \bigwedge Q) \\
&\quad \land [T_1]_{S'} \in \mathcal{T} \land [T_2]_{S'} \in \mathcal{T} \\
&\quad \land ([T_1]_{S'} <:_{\Theta',\Lambda} [T_2]_{S'} \\
&\qquad \Rightarrow \forall a \in V.\forall \nu \in \mathbb{Z}. \\
&\qquad\quad \forall i_1 \in \mathrm{value}_\Gamma(\{\nu : \mathit{Int} \mid r'_1\}). \dots .\forall i_n \in \mathrm{value}_\Gamma(\{\nu : \mathit{Int} \mid r'_n\}). \\
&\qquad\qquad [[S(a)]]_{\{(\nu,z),(b_1,i_1),\dots,(b_n,i_n)\}} \Rightarrow [[S'(a)]]_{\{(\nu,z),(b_1,i_1),\dots,(b_n,i_n)\}})
\end{aligned}
$$

Proof: See thesis.

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 using my type checker.

```
fun : {v:Int | True} -> {v:Int | True}
  -> {v:Int | (||) ((<=) a v) ((<=) b v) }
fun =
  \a -> \b -> max a b
```

The type checker assumes that `max a b` has type `{v:Int | True}`

Conclusion: The Ugly

```
fun : {v:Int | True} -> {v:Int | True}
  -> {v:Int | (||) ((<=) a v) ((<=) b v) }
fun =
  \a -> \b ->
    let
      z = max a b
    in
    if (||) ((<=) a z) ((<=) b z)  then
      z
    else
      a -- dead branch
```

The user needs to know about the inner workings of the type checker.

Conclusion: The Ugly

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.

Further Work

- Implementation in a lower level programming language.
- Tailering the initial set $Init(V)$ towards a specific use-case.
- Better error messages (Figure out why a the type checker might have failed).