# Elm-Refine: Refinement Types for Elm

*Lucas Payr*

*2019-06-24*

## 1 Introduction

On 21. September 1997 the onboard computer of the USS Yorktown aircraft carrier threw an uncaught division by zero exception. This resulted in the board computer shutting down and the ship becoming unable to control until an engineer could restart the computer. Fortunately this happened during a training maneuver [Par19].

Nowadays, by IEEE 754 standard, systems are required to return `NaN`(Not a Number) or `Infinity` in case of a division by zero error. This way the computer would not have crashed, though it would have still caused unexpected behaviour. Typically, such errors can only be found by extensive testing. Instead one might try to use a more expressive type system that can ensure in compile time that division by zero and similar bugs are impossible to occur.

These more expressive types are called *Refinement Types*. Some authors of the papers, that will be cited in this thesis, also call them Dependent Types, tough Dependent Types are typically more general: They are used to prove specific properties by letting the type definition depend on a quantified predicate, whereas a Refinement Type takes an existing type and excludes certain values that do not ensure a specific property. To avoid confusion, we will use the term Refinement Types for types that depend on a predicate written in propositional logic. We will call such a predicate the *refinement* of the Refinement Type.

Refinement types were first introduced by Tim Freeman and Frank Pfenning in 1991[FP91]. In 2008 Patrick M. Rondon, Ming Kawaguchi and Ranjit Jhala from the university of California came up with the notion of Logically Quantified Data Types, or *Liquid Types* for short. Liquid Types limit themself to refinements on Integers and Booleans written in propositional logic together with order relations and addition.

Most work on Liquid Types was done in the UCSD Programming System research group, from which the original paper originated. The group has presented different implementations of Liquid types:

- **DSolve** for OCaml/ML [KRJ10]. This type checker originated from the original paper. In the original paper liquid types could only be ensured for a calculus called $\lambda_L$. It first translates OCaml to $\lambda_L$ and then checks for type safety.
- **CSolve** for C [Ron+12]. As a follow-up to DSolve, this checker implements Low-Level Liquid Type (LTLL) [RKJ10] for a formal language called *NanoC*, that extends $\lambda_L$ with pointer arithmetics.
- **LiquidHaskell** for Haskell [Vaz+14]. Extending $\lambda_L$, this type checker uses a new calculus called $\lambda_P$, a polymorphic $\lambda$-calculus.[VRJ13] Newer versions can also reason about termination, totalness (of functions) and basic theorems.
- **RefScript** for TypeScript [VCJ16]. In a two phase process, the dynamic typed language gets translated into a static typed abstract language [VCJ15]. This language can then be type checked using $\lambda_P$.

Outside of that research group, refinement types have also been implemented for Racket [KKT16], Ruby[Kaz+18] and F#[Ben+08].

## 2 Goals of the Thesis

The goal of this thesis is to adapt the concepts used in Liquid Haskell to a similar pure functional language called *Elm*. Elm was invented in 2012 by Evan Czaplicki as his master thesis. It is designed to be an introduction to functional programming. Most Elm programmers have either a JavaScript background or no prior programming experience at all. Therefore, the resulting type checker of this thesis should be usable without general knowledge about Liquid types.

Liquid Haskell requires quite a lot of knowledge about programming and Liquid Types in general. There are no save guards in place, and therefore its easy to write code that the type checker is not able to check. The resulting behaviour is not consistent: they are sometimes very cryptic ("Float is not numeric, because Float is not numeric" when comparing a float with a constant) or even wrong:

```
{-@ ax1 :: {x:Int | x*x <= 4} -> {x:Int | x*x <= x+x} @-}
ax1 :: Int -> Int
ax1 x = x
```

returns the error

```
Error: Liquid Type Mismatch
Inferred type
{v:Int | v * v <= 4 && v == x}

not a subtype of Required type
{VV:Int | VV * VV <= VV + VV}

In Context
x:{v:Int | v * v <= 4}
```

Without knowledge of Liquid Types, this message is hard to understand. It's also the wrong type of error. The types do match, its just that the type checker has trouble figuring that out. So instead an error message like "*could not prove*" or "*polynomial expressions are not allowed*" would be better.

Additionally, the syntax of Liquid Haskell is not suitable for Elm. A small example written in Liquid Haskell would be the following:

```
{- Positive Integers -}
{-@ type Nat = {x:Int | x > 0} @-}

{- Increases the value by one -}
{-@ increase :: in:Nat -> {out:Nat | out == in + 1} -@}
increase = (+) 1
```

There are two problems with that design.

- Elm uses so called Extendible Records, for example `{s|num:Int}`. For someone who's not familiar with Liquid Haskell or programming in general, Extendible Records and Liquid Types look very similar.

- Elm uses an auto formatter that automatically inserts an empty line after a comment. Additionally, Documentation comments (written `{-| something -}`) have no following empty line, but instead but be followed by the type signature.

An alternative Syntax that would work for Elm is the following:

```
{-| Positive Integers

@refinedBy `\x -> x > 0`
-}
type alias Nat =
    Int

{-| Increases the value by one

@refinedBy `\in out -> out == in + 1`
-}
increase : Nat -> Nat
increase = (+) 1
```

Here the last line of the documentation contains the type definition. The keyword `@refinedBy` would fit right in with other documentation keywords. `\.->.` is the syntax for a lambda function.

## 3 Expected Results

The result will consist of three parts: A Formal Language, a Type Checker and a Case Study.

### 3.1 The Formal Language

In the first step, a formal language equivalent to Elm will be extended to support Liquid Types. The type system and type inference will be formally proven with respect to the semantic. Additionally, the derived rules will be implemented in K system to validate that the system actually works.

Liquid Haskell has `Int`, `Boolean`, `List`, `Tuple`, lambda functions and custom data types. The formal language will include the same types as well as records.

The formal Language will allow refining `Bool`, `Int`, `Tuple a` and functions types using constants `c`, variables, expressions `+, * c, -`, relations `<=, <, ==, /=`, predicates `not, &&, ||` and truth values `True, False`.

### 3.2 The Type Checker

The second step will be to implement the type checker. The implementation will be done in Elm using the GitHub project *stil4m/elm-analyse* as a foundation. Elm-analyse is a refactoring tool that enforces coding rules and good practices. It exposes the Abstract Syntax Tree (AST) using *stil4m/elm-syntax* and has a finished interface that can display error messages together with automatic scripts that can fix the bugs.

The type checker will also work for `Char` and include the relations `>, >=`.

The final result will be published on the GitHub repository *Orasund/elm-refine*.

## 3.3 The Case Study

For the third step, parts of the core elm package will be adopted to various refinement types and checked using the type checker. The goal of the case study is to improve the type checker with useful error messages and present a set of refinement types that can be used by the elm community without knowing anything about refinement types. For the error messages, the type checker will link to documentation, also located in the GitHub repository. This style of error messages matches the way Elm uses its error messages:

Elm allows for debugger-guided learning, where the debugger will reveal new features when necessary and references the specific pages of the documentation.

Here is an example error message:

```
This `case` does not have branches for all possibilities:

22|>    case list of
23|>        a :: _ ->
24|>            a

Missing possibilities include:

    []

I would have to crash if I saw one of those. Add branches for them!

Hint: If you want to write the code for each branch later, use `Debug.todo` as a
placeholder. Read <https://elm-lang.org/0.19.0/missing-patterns> for more
guidance on this workflow.
```

# 4 Structure

The thesis will have the following structure.

1. Introduction (3 pages)
2. State of the Art (5 pages)
   - Type Theory
     - Why was type theory invented?
     - How does it compare to set theory?
     - What are Dependent types?
     - What are Refinement types?
     - Introduction to definitions that will be used in the thesis.
   - Introduction to Elm
     - A Mathematical view on Elm.
     - How does Elm deal with Side Effects?
     - Three ways to get run-time errors in Elm.
     - Overview of features that will be ignored in this Thesis. (Cmd Type, Extendable Records, Opaque Types/modules, Recursive Types, Recursive Functions, special Type Classes)
   - Liquid Haskell
     - History of Liquid Haskell
     - measuring the length of a list

    - proving termination of lists
    - simple theorem proving
3. The Formal Language (10 Pages)
    - Liquid types For OCaml
        - How does the original paper work?
        - $\lambda_L$ Calculus
        - Hindley Milner type inference
        - Predicate Abstraction, used for generating Loop Invariants
    - Liquid types for Haskell
        - How did they adopted the original type system to Haskell?
        - Abstract Liquid Types for Type Classes
    - Liquid types for Elm
        - The formal language
        - The $\lambda$ calculus as subset of the formal language
        - The Type System
        - The Type inference
4. The Implementation (10 Pages)
5. The Case study (10 Pages)
    - Implementing various modules of the core package using different refined types.
    - core/Basics module implemented for `Int\{0}`
    - core/Basics module implemented for `Range`
    - core/Char module implemented for refined `Char` (the resulting types will be sharp)
    - core/Bitwise module implemented for `4Bit` integers
    - Short presentation of the error messages
6. Conclusions and Future Work (1 page)

## 5 Time Frame

July 2019

- Finishing the formal language (without refinement types)

August 2019

- Working on the K Framework implementation

September 2019 - Holidays

October 2019

- Start writing the *State of the Art* chapter
- Writing the formal language

November 2019

- Start writing the *Liquid types for Elm* chapter
- Implement the formal language for K Framework

December 2019

- Start writing the *The Formal Language* chapter
- Start with the implementation of the type checker

January 2020

- start writing the *The Implementation* chapter
- working on the type checker

February 2020

- finishing the type checker

March - July 2020 - Break

August 2020

- Starting with the case study
- implementing the core package for `Int\{0}`
- implementing the core package for `Range`

September 2020

- implementing chars for `Char` refinement types.

October 2020

- implementing "Bitwise" module for `4Bit` integers

November 2020

- start writing the *The Case study* chapter
- writing the *Conclusion and Future Work* chapter

Dezember 2020

- finalize thesis
- presentation

February 2020

- Master's examination

## References

[Ben+08] Jesper Bengtson et al. "Refinement Types for Secure Implementations". In: *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008*. 2008, pp. 17–32. DOI: 10.1109/CSF.2008.27. URL: https://doi.org/10.1109/CSF.2008.27.

[FP91] Timothy S. Freeman and Frank Pfenning. "Refinement Types for ML". In: *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*. 1991, pp. 268–277. DOI: 10.1145/113445.113468. URL: https://doi.org/10.1145/113445.113468.

[Kaz+18] Milod Kazerounian et al. "Refinement Types for Ruby". In: *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings*. 2018, pp. 269–290. DOI: 10.1007/978-3-319-73721-8_13. URL: https://doi.org/10.1007/978-3-319-73721-8%5C_13.

[KKT16] Andrew M. Kent, David Kempe, and Sam Tobin-Hochstadt. "Occurrence typing modulo theories". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 2016, pp. 296–309. DOI: 10.1145/2908080.2908091. URL: https://doi.org/10.1145/2908080.2908091.

[KRJ10] Ming Kawaguchi, Patrick Maxim Rondon, and Ranjit Jhala. "Dsolve: Safety Verification via Liquid Types". In: *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*. 2010, pp. 123–126. DOI: 10.1007/978-3-642-14295-6_12. URL: https://doi.org/10.1007/978-3-642-14295-6%5C_12.

[Par19] Matt Parker. *Humble Pi:A Comedy of Maths Errors*. ISBN 978-0241360194. Allen Lane, 2019.

[RKJ10] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. "Low-level liquid types". In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. 2010, pp. 131–144. DOI: 10.1145/1706299.1706316. URL: https://doi.org/10.1145/1706299.1706316.

[Ron+12] Patrick Maxim Rondon et al. "CSolve: Verifying C with Liquid Types". In: *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. 2012, pp. 744–750. DOI: 10.1007/978-3-642-31424-7_59. URL: https://doi.org/10.1007/978-3-642-31424-7%5C_59.

[Vaz+14] Niki Vazou et al. "Refinement types for Haskell". In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*. 2014, pp. 269–282. DOI: 10.1145/2628136.2628161. URL: https://doi.org/10.1145/2628136.2628161.

[VCJ15] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. "Trust, but Verify: Two-Phase Typing for Dynamic Languages". In: *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*. 2015, pp. 52–75. DOI: 10.4230/LIPIcs.ECOOP.2015.52. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2015.52.

[VCJ16] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. "Refinement types for TypeScript". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 2016, pp. 310–325. DOI: 10.1145/2908080.2908110. URL: https://doi.org/10.1145/2908080.2908110.

[VRJ13] Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. "Abstract Refinement Types". In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 2013, pp. 209–228. DOI: 10.1007/978-3-642-37036-6_13. URL: https://doi.org/10.1007/978-3-642-37036-6%5C_13.