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].

Typically, such errors can only be found by extensive testing. Instead one might try to use a more expressive type system that can ensure at compile time that division by zero and similar bugs are impossible to occur.

These more expressive types are called *Refinement Types* [FP91]. Some authors also call them *Dependent Types* [Chr18], though 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 λ_L. It first translates OCaml to λ_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 λ_L with pointer arithmetics.
- **LiquidHaskell** for Haskell [Vaz+14]. Extending λ_L, this type checker uses a new calculus called λ_P, a polymorphic λ-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 λ_P.

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

1.1 Goals of the thesis

The goal of this thesis is to define Liquid Types for the pure functional language called *Elm* [CC13]. This includes a formal semantics of Elm, the extended type system, a proof that the type-system is sound with respect to the semantics, and the type inference to-

gether with a subset of the language on which the inference can be applied. To validate the results the rewriting framework *K Framework* [RS14] will be used and the finished result will be implemented as a proper type checker for Elm.

Elm was invented in 2012 by Evan Czaplicki as his master thesis [CC13]. It uses functional reactive programming to handle side effects, essentially modelling side effects as an infinite sequence. It is aimed at web development and for that, it invented *The Elm Architecture* (TEA). This architecture later got adopted by the JavaScript Framework *React* as the Library *Redux*.

Compared to the well-known functional programming language Haskell, Elm is simpler and has a lot of unique interpretations of Haskell concepts. Mainly the lacking of type classes and monads. The biggest difference to Haskell is that it does not allow runtime exceptions. That said, in theory Elm still has three types of runtime errors: out of memory, function equivalence and non-terminating functions. These errors can be completely avoided if the Elm guidelines are followed, thus its claim of having "no runtime errors".

From a non-technical stand point, Elm focuses heavily on learning. It designs its features to always be beginner friendly. This represents a unique position, as a proper implementation of Liquid Types in Elm would evidently need to be usable for academia and beginners alike.

Currently, the most advanced implementation of liquid types is Liquid Haskell[Vaz+14]. It allows any Haskell code even though its type inference is only sound for a subset. If one does write unsound code, there is no indication for it from the type checker. The resulting behaviour is very situation based: They are sometimes very cryptic ("Float is not numeric, because Float is not numeric" when comparing a Float with a constant) or even wrong. For example, the code fragment

```
{-@ 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 also indicates 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.

1.2 Expected results

The result will consist of three parts: A language with a formal syntax and semantics, a type checker and a case study.

1.2.1 The language

First a mathematically specified subset of Elm will be extended to include liquid types. In detail this includes:

1. A formal syntax written as Backus-Naur form (BNF).
2. A subset of the language that is equivalent to λ_L. This includes `Int`, `Bool`, `Tuple` and Lambda functions using constants `c`, variables, expressions (`+, * c, -`), relations (`<=, <, ==, /=`), predicates (`not, &&, ||`) and truth values (`True, False`).
3. A formal type system using Hindley Milner type inference. The liquid types will only be inferred for parts of the program that are written in said subset.
4. A high level denotational semantic.
5. A proof, using the denotational semantics, that the type system rules out runtime errors.
6. A low level small step semantic suitable to be used with K Framework. The implementation in K Framework will be using for development and rapid prototyping of the language as well as the type system.

1.2.2 The type checker

The second step will be to implement the type checker as part of a real Elm implementation. It 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*.

1.2.3 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.
```

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.

[CC13] Evan Czaplicki and Stephen Chong. "Asynchronous functional reactive programming for GUIs". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 2013, pp. 411–422. DOI: 10.1145/2491956.2462161. URL: https://doi.org/10.1145/2491956.2462161.

[Chr18] Daniel P. Friedman; David Thrane Christiansen. *The Little Typer*. Paperback. The MIT Press, 2018. ISBN: 0262536439,9780262536431.

[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.

[RS14] Grigore Rosu and Traian-Florin Serbanuta. "K Overview and SIMPLE Case Study". In: *Electr. Notes Theor. Comput. Sci.* 304 (2014), pp. 3–56. DOI: 10.1016/j.entcs.2014.05.002. URL: https://doi.org/10.1016/j.entcs.2014.05.002.

[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.