JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Lucas Payr
K01556372

Institute
Research Institute for Symbolic Computation

Supervisor and First Examiner
A.Univ.-Prof. DI Dr.
Wolfgang Schreiner

March 8, 2021

REFINEMENT TYPES FOR ELM

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

JOHANNES KEPLER
UNIVERSITY LINZ

Abstract

Liquid types provide a theory for implementing refinement types for a Hindley-Miler type system. Elm is a pure functional programming language using such a type system. In this thesis we discuss the extension of Elm with Liquid types.

JOHANNES KEPLER
UNIVERSITY LINZ

Thanks

I would like to thank my supervisor Wolfgang Schreiner for his continuous advice and suggestions throughout numerous revisions. Your input helped a lot.

I would also like to thank my friends and my girlfriend for this lovely time I had at university. I would not have been able to finish my degree if it would not have been without your love and support. It was a great time.

Last but not least, a big thanks to my family. I am privileged to have you all.

JOHANNES KEPLER
UNIVERSITY LINZ

Contents

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 be controlled until an engineer was able to 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" within this thesis. The predicate describing the valid values of a refinement type is called a *refinement*.

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 a variant of refinement types called 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].

The goal of this thesis is to define Liquid Types for the pure functional language called *Elm* [CC13]. In Chapter 2 we will give a quick history of type systems and the Elm language. In Chapter **??** we will formally define the type system of Elm. In Chapter **??** we introduce refinement types and describe how we can extend the type system using liquid types. In Chapter **??** we discuss the implementation and provide a demonstration. In Chapter 6 we give the conclusion of the thesis.

2 State of the art

In this section we give a quick history of type theory and the Elm langauge.

2.1 Type Theory

In 1902, Bertrand Russell wrote Gottlob Frege a letter, pointing out a paradox in Gottlob's definition of sets in his paper *Begriffsschrift* [Fre84]. Up until this point mathematicians expected that the naive theory of sets was well-formed. Russell's paradox gave a contradiction to this assumption:

Theorem 2.1: Russell's Paradox

Given the set of all sets that do not contain themselfs $R = \{x | x \notin x\}$, then $R \in R$ and $R \notin R$.

To fix this problem, Russell added a basic theory of types in the appendix of *Principia Mathematica* [WR27], which at the time was already in the printer. This rushed theory was not perfect, but it marks the first appearance of type theory.

Russell thought that the main pain point of set theory was that sets could be defined implicitly, (without listing all elements). Type theory is therefore by design constructive. Every value has exactly one type. Once the type of value is given, it can not change. This is a big difference to sets, where elements can live in any amount of different sets.

In Russell's original *ramified theory of types* he defined an order amongst the predicates. The lowest predicates could only reason about types. Higher predicates could

reason only about lower predicates. This made the type of functions not only depend on the types of its arguments but also on the types of predicates contained in the definition of the function. Thus, the type of a function could get very complicated even for simple functions [KLN04].

In 1921 Leon Chwistek and Frank Ramsey noticed that if one would allow recursive type definitions, the complexity could be avoided. This new theory is in general referred to as *Simple Type Theory*.

Definition 2.1: Simple Type Theory

1. `0` is a type;
2. If $T_1, \ldots, T_n$ are simple types, then also $(T_1, \ldots, T_n)$ is a simple type. For $n = 0$ we obtain the simple type ();
3. All simple types can be constructed using rules 1 and 2.

[KLN04].

Note that () stands for the type of all propositions and `0` is the type of all variables. $(T_1, \ldots, T_n)$ denotes the type of n-nary relations between values in T_i, for i from 1 to n. For example the predicate $a < b$ for variables a, b would be of type $(0, 0)$. For P and Q of type (0), the predicate $P(a) \wedge Q(b)$ would be of type $((0))$ [KLN04]. The theory of types has changed a lot since then.

At that time another method for dealing with Russell's paradox was invented by Ernst Zermelo: His axiomatic set theory. It was further refined by Abraham Fraenkel in 1920 to what is now known as Zermelo-Fraenkels set theory (ZF). Mathematicians nowadays prefer using ZF over type theory, as it is more expressive.

Type theory lost any relevance for about 30 year. Then in the 1950s type theory finally found its use amongst computer scientists, when type checkers were added to compilers. A type checker ensures that an expression has a specific type and therefore proves that the program is well-typed. One of the first programming languages with a type checker was Fortran. Earlier type checkers existed, but only in the realm of academia.

Between 1934 and 1969 Haskell Curry and William Howard noticed that proofs could be represented as programs: A theorem can be represented by a type and the corresponding proof can be represented as a program of said type. More explicitly they noticed a relation between natural deduction and lambda calculus. This realization, now known as the *Curry-Howard-Correspondence*, resulted in the invention of proof checking programs.

The next big step for type theory was in 1972 as Per Martin-Löf introduced a new form of type theory, now known as the Martin-Löf type theory. Martin-Löf took the Curry-Howard-Correspondence and extended it such that it is able to write statements in predicate logic. To do so, he introduced dependent types.

2.2 Dependent types and Refinement types

Because dependent types have the same expressiveness as statements in predicate logic it can be used to check proofs written in predicate logic. This checking process is currently still far from automatic, but it has the benefit of producing bulletproof proofs. Note that dependent types can not be automatically checked. In comparison, an extension to type theory that can be checked automatically are so-called *refinement types*. The idea behind *refinement types* is to use a predicate to restrict possible values of an existing type. Refinement types are therefore a form of *subtyping*.

The main theory behind refinement types was developed by Tim Freeman and Frank Pfenning in 1991 in the paper titled *Refinement Types for ML* [FP91]. The original paper only allowed predicates using $\wedge$ and $\vee$. The underlying method for inferring the types was based on the already implemented type inference for the programming language ML. A type inference finds the most general type for a given expression. The type inference for ML, also called *Hindley-Milner-Type inference* has become essential not only for refinement types, but also for all modern functional programming languages.

Modern refinement types are mostly influenced by a paper in 2008 by Patrick Rondan, Ming Kawaguchi and Ranji Jhala titled *Liquid Types*. Whereas the approach in 1991 used the Hindley-Milner inference with small modifications, this method introduced an additional theory on top of the Hindley-Milner inference. This new form of refinement types allow predicates written in propositional logic for integers with the order relations $\leq$ and $<$, addition and multiplication with constants. In theory one can extend the realm of allowed relations to any relation that can be reasoned upon using an SMT-Solver.

2.3 Introduction for Elm

The programming language Elm was developed by Evan Czaplicki as his master thesis in 2012. It was initially developed as a reactive programming language. In reactive programming effects are modelled as data streams (a sequence of effects). This was implemented in Elm by a data type called *Signal*. Signals allowed for "time-travel debugging": One can step forwards and backwards through the events.

While signals where a very direct implementation of the mathematical model(a sequence of effects), it turned out to be not very useful. Thus, it got replaced by a special architecture, now known as *The Elm Architecture* (TEA). Nowadays, an Elm program is more like a state machine with events being state transitions:

We call the state of a program the `Model`, the events are called messages or `Msg` for short. The TEA has three functions:

```
init : Model

update : Msg -> Model -> Model

view : Model -> Html Msg
```

The program start with an `init` model. The model then gets displayed on screen using the `view` function. The result of the view function is an HTML document. This resulting HTML allows the user to send messages (for example by pressing a button). These messages then get send one at the time through the `update` function. The `update` function changes the model and then calls the `view` function, resulting in an update of the HTML document. To allow impure effects like time-related actions or randomness, Elm can send and receive asynchronous messages from and to a local or remote computer.

Elm claims that it has no runtime errors. This is technically not true. There exist three types of runtime errors: running out of memory, non-terminating functions and comparing functions. For this thesis we can safely ignore the first two types of runtime errors. The reason why Elm has problems comparing functions is that it uses the mathematical definition of equal. Two functions f and g are equal if $\forall x.f(x) = g(x)$. For infinite sets, like the real numbers, this is impossible to check numerically. Thus, elm can not reason about $(\lambda x.x \cdot 2) = (\lambda x.x + x)$. For our thesis we do not allow comparisons between functions.

Elm has a lot of hidden features that are intended for advanced programmers and therefore mostly syntax sugar or quality-of-life features. These features include recursive types, opaque types, extendable records and constraint type variables. For this thesis we will not consider any of these features.

6 Conclusion

In this thesis we have looked at the type system for the Elm language and discussed extending it using refinement types. The final implementation was done in Elm in order for it to be included into the Elm-in-Elm compiler [Jan19].

We have seen that liquid types are refinement types that can reason upon the if-then-else branches and derive sharper subtypes for each of the branches. In combination with an SMT-solver this means that one can have different range types as subtypes of integers.

The expressiveness of liquid types are directly dependent on the initial set of predicates and the allowed expressions in $\mathcal{Q}$. Extending the allowed expressions in $\mathcal{Q}$ requires that the SMT solver can still cope with them. Therefore, we can expect to see more complicated use cases for liquid types as SMT solvers get better. We also know that the search space for the derived predicates must be finite. We will always have a maximal recursion depth with respect to $\mathcal{Q}$ for expressions in our search space. This means that no matter how big the space we are considering is, there will always be a predicate in $\mathcal{Q}$ that can not be found.

When considering if one would want to add liquid types to a language (like Elm), one must keep in mind that liquid types can not always be checked. Thus, the type system is no longer complete. Note that we have only proven soundness under the assumption that all if-then-else conditions are in $\mathcal{Q}$. Therefore, one must consider if the benefits, in particular having subtypes of integers, out way the downsides, in particular the type system being no longer complete.

For Elm the biggest benefit would be to have range types. This would allow writing enum types and give more compile type guaranties.

To turn the implementation described in Section **??** into a usable product, one would need to rewrite the type-checker part of the Elm-in-Elm compiler. The updated checker would need to collect the subtyping conditions while inferring the type. (As discussed in Section **??**) This can not be done by simply traversing the abstract syntax tree. Such an addition would be simple but tedious, as every type inference rule would need to be updated.

A simpler solution for adding a range type in Elm has been found by the author while working on this thesis: One can use a phantom type (an algebraic type were not all type variables are used) to encode the length of an integer into its type:

```
type One = One


type OnePlus n = OnePlus n


type Range n = Range Int


type Const n = Const Int
```

```
one : Const One
one =
  Const 1

plus1 : Const n -> Const (OnePlus n)
plus1 (Const n) =
  Const (n + 1)

fromModBy : Const n -> Int -> Range n
fromModBy (Const n) int =
  Range (modBy n int)
```

An additional benefit of this approach is that Elm removes single constructor types in compile time. This means that `Range` and `Const` are modelled as `Int` in the compiled source code.

The author has published a usable package containing said types under the name "Elm-Static-Array" [Pay21].

JOHANNES KEPLER
UNIVERSITY LINZ

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

.....

Lucas Payr

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.

[Fre84] Gottlob Frege. *Die Grundlagen der Arithmetik – Eine logisch mathematische Untersuchung über den Brgiff der Zahl*. Breslau: Verlag von Wilhelm Koebner, 1884. URL: http://www.gutenberg.org/ebooks/48312,%20https://archive.org/details/diegrundlagende01freggoog.

[Jan19] Martin Janiczek. *Elm-In-Elm*. https://github.com/elm-in-elm/compiler. 2019.

[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.

[KLN04] Fairouz Kamareddine, Twan Laan, and Rob Nederpelt. *A Modern Perspective on Type Theory: From its Origins until Today*. Applied Logic 29. Kluwer, 2004.

[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.

[Pay21] Lucas Payr. *Elm-Static-Array*. https://github.com/Orasund/elm-static-array. 2021.

[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.

[WR27] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, 1925–1927.