JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Lucas Payr
K01556372

Submitted at
Research Institute for Symbolic Computation

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

March 11, 2021

REFINEMENT TYPES FOR ELM

Master Thesis

to obtain the academic degree of

Diplom-Ingenieur

in the Master's Program

Computer Mathematics

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

JOHANNES KEPLER
UNIVERSITY LINZ

Zusammenfassung

Das Ziel dieser Arbeit ist es, das Typensystem von Elm mit sogenannten “Refinement” Typen zu erweitern. Elm ist eine reine funktionale Programmiersprache welche das Hindley-Miler Typsystem benutzt. Refinement Typen sind eine Form von Subtypen, welche anhand eines Prädikats seine enthaltenen Werte bestimmt. Eine solche Klasse von Refinement Typen welche für Hindely-Miler Typsysteme ausgelegt ist, sind die sogenannten “Liquid” Typen. Für sie existiert ein Algorithmus, um für einen Ausdruck den dazugehörigen Typ herzuleiten. Dieser Algorithmus interagiert mit einem SMT Solver, um bestimmte Bedingungen für die Subtypen zu erfüllen. Typsysteme welche mit Liquid Typen erweitert werden, sind nicht länger vollständig. Stattdessen können sie nur für bestimmte Ausdrücke hergeleitet werden. Die Prädikate sind ebenfalls restringiert. Diese Arbeit liefert eine formale Definition von Elm und dessen Typsystem. Außerdem wird das System mit Liquid Typen erweitert. Hierfür wird eine Teilmenge der Ausdrücke sowie eine Teilmenge der Prädikate präsentiert, um die Wohldefiniertheit der Liquid Typen zu gewährleisten. Zum Überprüfen unserer Ergebnisse benützen wir das Softwaresystem “K Framework” sowie eine Implementierung in Elm des Algorithmus zum Lösen der Bedingungen für Subtypen. Als SMT Sovler benutzen wir Z3.

JOHANNES KEPLER
UNIVERSITY LINZ

Abstract

The aim of this thesis is to add refinement types to Elm. Elm is a pure functional programming language that uses a Hindley-Miler type system. Refinement types are subtypes of existing types. These subtypes are defined by a predicate that specifies which values are part of the subtypes. To extend a Hindley-Miler type system one can use so-called Liquid types. These are special refinement types that come with an algorithm for type inferring. This algorithm interacts with an SMT solver to solve subtyping conditions. A type system using Liquid types is not complete, meaning not every valid program can be checked. Instead, Liquid types are only defined for a subset of expressions and only allow specific predicates. In this thesis we give a formal definition of the Elm language and its type system. We extend the type system with Liquid types and provide a subset of expressions and a subset of predicates such that the extended type system is sound. We use the software system "K Framework" for rapid prototyping of the formal Elm type system. For rapid prototyping of the core algorithm of the extended type system we implemented said algorithm in Elm and checked the subtyping condition in the SMT solver Z3.

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

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.

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].

Elm is a pure functional programming language invented in 2012 by Evan Czaplicki as his master thesis *Elm* [CC13]. Elm has many similarities to Haskell but is simpler in nature. It uses a special architecture similar to state machines instead of monads used in Haskell. This architecture ensures that no runtime error can occur: Error are modelled as a type and therefore need to be handled. The language compiles to JavaScript. This gives the Elm community the unique position being the first contact with functional programming for many programmers.

This unique position gave rise to design philosophies for writing good functional code, not only for Elm. One such philosophie is “Make impossible states impossible”: Model your program in such a way that any possible state of the model represents a valid state of the program. Refinement types provide a way to achieve that rule for integers. Even simple types like a subtype containing all natural numbers or a type containing a range of numbers would be of help.

The goal of this thesis is therefore to define Liquid Types for Elm. In particular to define a set of predicates $\mathcal{Q}$ such that the type of ever Elm program with if-then-else conditions in $\mathcal{Q}$ can be inferred. $\mathcal{Q}$ needs to allow the definition of range types, natural numbers and non-zero integers.

Elm has changed a lot since 2012 and therefore the formal model described in the original thesis is outdated. We therefore start by formally defining the Elm language. This will include the syntax, denotational semantic, types and a system of inference rules to infer them. We then will introduce the formal notion of Liquid types to our type system. We also will define the subset of allowed predicates in an if-then-else condition and will extend the syntax with refinement types. Furthermore, we will present altered inference rules to ensure that Liquid types can be defined. Here we will introduce the notion of subtyping conditions. We will describe an algorithm to solve these conditions. Here we will specify a set of predicates that may be used for deriving a refinement solving the subtyping conditions. The algorithm generates SMT statements that can be checked using an SMT solver. To demonstrate everything in tandem we will provide an implementation of the subtyping algorithm using Z3 for solving the SMT statements.

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 3 we give the conclusion of the thesis.

2 State of the art

In this chapter 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 1.1: 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 1.1: Simple Type Theory

1. 0 is a type and () is a type;
2. If $T_1, \ldots, T_n$ are simple types, then also $(T_1, \ldots, T_n)$ is a simple type.

[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 $\lambda a, b.a < b$ for variables a, b would be of type $(0, 0)$. For P of type (0) and Q of type (), the predicate $\lambda a, b.P(a) \wedge Q(b)$ 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 and automated reasoning.

2.2 Hindley Milner Types System

One of the first systems for automated reasoning was LCF (Logic for Computable Functions) invented in 1973 by Robin Milner. To realize LCF he invented a functional programming language called ML (Meta Language). ML uses a special system of types known as the *Hindley-Milner Type System*. This system introduces *polymorphic types*: Types that can be instantiated to obtain new types. As an example, we consider the simple type theory extended by polymorphic types, as used in the Hindley-Milner type system.

Definition 2.1: polymorphic types for Simple Type Theory

1. Let T be a type. Then $\forall a.T$ is a polymorphic type. We call a a *type variable.*
2. Let T_0 be a type, let T be a polymorphic type. Then $T\ T_0$ is a type. Such a type is called a *type application.*
3. We call types defined in 1.1 *monomorphic* types.

With this definition, the predicate $\wedge$ has type $\forall a.\forall b.(a, b)$. For the predicate P and Q of type (0), the predicate $\lambda a.\lambda b.P(a) \wedge Q(b)$ has type $\Big(\big(\forall a.\forall b.(a,b)\big)\ (0)\Big)\ (0)$. Note that we have derived two types for the same expression. The Hindley-Milner type system comes with two equivalence rules to ensure that every expression has a unique type. The first equivalence rule says that any bound variables may be renamed and the second that type applications can be eliminated by instantiating the polymorphic type.

$$\Big(\big(\forall a.\forall b(a,b)\big)\ (0)\Big)\ (0) = \big(\forall a.((0),a)\big)\ (0) = ((0),(0))$$

Nowdays ML exists in different dialects such as SML(Standard ML), OCaml and F#.

Additionally, ML also provides rules for inferring the type for a given expression. These inference rules provide the most general type possible. This means it will only instantiate a polymorphic type if necessary.

2.3 Dependent types and Refinement types

In 1972 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.

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 type inference for ML.

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.

2.4 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.

3 Conclusion

In this thesis we have looked at the type system for the Elm language and discussed extending it using refinement types.

The original intent was to have an implementation of the type checker for Liquid types. We expected that the resulting Liquid types are defined such that non-negative integers, range types and non-zero integers can be defined. We expected to implement Liquid types for `Int`, `Bool` and tuples of Liquid types. Additionally, we expected that the inferred type of the `max` function can be sharp. Such a sharp refinement is $(a \leq \nu) \wedge (b \leq \nu) \wedge (\nu = b \vee \nu = a)$.

The resulting type system is capable of defining all described integer types and also allows inferring Liquid types for functions over integers. It does not include Liquid types over `Bool` and tuples. Additionally, the inferred type of the max function is not sharp: $(a \leq \nu) \wedge (b \leq \nu)$. Adding these features would have been too time-consuming and would not have added any new theory to the system. The inclusion of booleans would have meant that we would have needed to type check the liquid expressions. This would have added unnecessary complexity, as we are mainly interested in subtypes of integers. Tuples would have been easy to add but would not add additional behaviour: Tuples as arguments can be flattened and then transformed into a list of arguments. Tuples as a return argument are syntax sugar for defining multiple functions with the same input values. To infer a sharp refinement for the `max` function one can simply add $\nu = b \vee \nu = a$ to the search space, but that is not very sophisticated. Another way would be to add $P \vee Q$ for a specific set of allowed predicates for P and Q. We used our Elm implementation to test this for the definition of P and Q not containing $\vee$. The resulting refinement was sharp but included a lot of trivial conditions. The search space increased by a factor of four. This factor could be decreased by ensuring that no two predicates are equivalent.

While working on the thesis it became clear that the original expectations did not match the possibilities of Liquid types. In particular, the expressiveness of Liquid types is 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. Opposite to our original expectation, the set of predicates allowed in If-branches $\mathcal{Q}$ is a superset of the predicates allowed in refinements. Additionally, the search space for the derived predicates must be finite. 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.

There are multiple future topics that can be explored.

- The set of allowed expressions $\mathcal{Q}$ and the search space for the inferred refinements can be extended to sharpen the inferred predicates. At some point this would also need to include an algorithm to simplify the inferred predicates. Otherwise, the inferred predicates can only be hardly read by humans.
- The current implementation in Elm can be extended to a full type checker by using the Elm-in-Elm compiler [Jan19]. This would require some changes to 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.
- One can try to implement a specific type in Elm without Liquid types. Liquid types make a type system incomplete. Therefore, this is a far better solution of implementing a specific type. For the range type, the author of this thesis has actually found another way. Namely, to implement these types using phantom types (an algebraic type were not all type variables are used) [Pay21].

Bibliography

[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.