2 State of the art

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 is well formed. Russell's paradox gave a contradiction to this assumption:

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 *pricipia mathematica* [WR27], that at the time was currently in the printer. This rushed theory was not perfect, but it marks the first appearance of type theory.

2.1 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 element has exactly one type. Once the type of an element is given, it can not change. This is a big difference to sets, where elements can live in any amount of different sets.

In Russells original *ramified theory of types* defined a 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 functions 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 general referred as *simple type theory*. It does as follows

Definition 2.1: sime 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, $n = 0$ is allowed: then we obtain the simple type $()$;
3. All simple types can be constructed using rules 1 and 2.

We use $t, u, t_1, ...$ as metavariables over simple types [KLN04].

$()$ stands for the type of the propositions and the type of the individuals [KLN04].

At that time another method of dealing with Russell's paradox was Ernst Zermelo's axiomatic set theory. That then was further refined by Abraham Fraenkel in 1920 to what is now known as Zermelo-Fraenkels set theory (ZF). Mathematicians would then continue using ZF over type theory, as is was more expressive.

It was only in the 1950s that type theory found its use in Fortran's compiler and type

checker. A compiler turns language expressions into program code. A type checker ensures that an expression as a specific type and essentially proofs that the program is well-typed. Fortan is a well known programming language from the 1950 that is still in use. It's one of the first langauges with a compiler. And the first language to be used for a consumer and not just purely in academia.

Between 1934 and 1969 Haskell Curry and William Howard noticed that proofs could be represented as programs with the proven statement being the type of said program, more explicitly 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 exend it to be able to write statements in predicate logic. To do so, he introduced dependent types. Dependent types essentially extended typical type theory with quanfifiers. Note that dependent types to not use predicate logic but are rather dependent types are equivalent to statements written in predicate logic.

2.2 Dependent types and Refinement types

Because dependent types have the same expressivness as statements in predicate logic it can be used to check proofs written in predicate logic. This checking process in currently still far from automatic, but it has the benifit of producing bulletproof proofs. If instead we want to use types for automatic checking or even proving, then dependent types can't help. Instead we need a type system that includes a method of proving any statement written in it. Such types are called refinement types.

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 the predicated $\wedge$ and $\vee$ to reason about types. The underlying method for infering the types was based on the already implemented type inference for ML. A type inference finds the most general type for a given expression. This type inference, also called Hindley-Milner-Type inference has become essential not only for refinement types in gerneral, 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*. Where as the approach in 1991 used the Hindley-Milner Inference with small modifications, this new method extend the Inference with new methods to ensure that the new forms of types can be infered. These resulting refinement types now allow arbitary predicates written in propositional logic for integers with the order relations $\leq$ and $<$, addition and multiplaction with constants.

2.3 Introduction for Elm

The programming language Elm was developed by Evan Czaplicki as his master thesis. While it is a pure functional language with a lot of similarities to the programming language Haskell, the initial emphasis was on it being a reactive programming language.

In reactive programming effects are modelled as data streams, as a sequence of effects. Initially this was implemented in Elm by a data type called *Signal*. The signal had Addesses, that pointed to a single position in the event sequence. From there one could look at the next position or even the previous one. This allowed for "time-travel debugging", allowing the debugger not only to go step by step through the programm but also step by step backwards.

While this method of modelling events was a very direct implementation of the mathematical model(a sequence of effects), it turned out to be not very usefull and the community around Elm started to use a special architecture, now known as *The Elm Architecture* (TEA). Now a 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 with the `view` function. The result of the view function is a website written in HTML. This resulting HTML allows for messages (for example by pressing a button). These messages then get send one at the time though the `update` function. This in return changes the model, what again changes the HTML on the website.

Sadly the real world is not as simple. With this setup, there is no way to have time-related messages or randomness. For this there exist also Commands or `Cmd` for short, that can give the computer tasks, once the task is finished the computer returns a message. Some tasks should run continuesly (like for example the passing of time), for that its also possible to subscribe to a task. The extended Achitecture now looks as follows:

```
init : Model

update : Msg -> Model -> (Model, Cmd Msg)

subscription : Model -> Sub Msg

view : Model -> Html Msg
```

This achritecture is the main reason why good Elm programs seldomly use recursive functions. Instead of recursive function the TEA can be used: Using the Command `succeed` we can return the original Msg through the Command system back to update function again. As the Command system is asynciasly, this means that the computer only processes a command once there is enough working memory available. If a recursive function loops indeviently, this would not freeze the computer. On the other side this slows down recursive function calls, even if its not for long. So in some cases recursive function are still used in Elm, though this has also to do with Elm allowing recurisve types. For this thesis we will not look at the TEA and not at recursive functions or recurisve types.

While 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 savely ignore the first two types of runtime errors. The reason why Elm has problems comparing functions is because it uses the mathematically definition of equal. Two functions f and g are equal if $\forall x.f(x) = g(x)$. For infinite sets, like the reals, this is impossible to check numerically. Thus elm can not reason about $(\lambda x.x * 2) = (\lambda x.x + x)$. A relative easy way to solve this would be to not allow comparisons of function within the type system. As the time of this writing the development team behind Elm has desided against a fix in the type systems, this has todo with the effort of getting wrid of constraint type variables within the internal type system of elm and the fix would instead involve adding a new constraint type variable just to exclude functions.

Elm has a lot of hidden features that are intentionally not mentioned in the offical guides and will only be pointed out by the compiler once they are nessary. These Features include recursive types, opaque types, extendable records and constraint type variables. For this thesis we will not consider any of these features as they only effect more experienced Elm developers.

References

[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.

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

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