HTL Leonding next level

JOHANNES KEPLER UNIVERSITÄT LINZ

HIGHER TECHNICAL FEDERAL COLLEGE LEONDING

Kipper - Programming Language for Improved Runtime Type-Safety

Diploma thesis

written as part of the

Matriculation and diploma examination

at the

Higher Department of Informatics

Handed in by:

Luna Klatzer
Lorenz Holzbauer
Fabian Baitura

Assisting Professor:

Dipl. Ing. Peter Bauer

Project Partner:

Prof. Dr. Dr. h.c. Hanspeter Mössenböck, Johannes Kepler Universität

Leonding, April 2025

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit 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 Arbeit wurde bisher in gleicher oder ähnlicher Weise keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Die vorliegende Diplomarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Leonding, April 2025 L. Klatzer & L. Holzbauer & F. Baitura

Abstract

Brief summary of our amazing work. In English. This is the only time we have to include a picture within the text. The picture should somehow represent your thesis. This is untypical for scientific work but required by the powers that are. Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Contents

1 Introduction

2 Background

2.1 Dissecting the current issues

2.1.1 The JavaScript problem

Currently, the web space is dominated by JavaScript, a language developed solely for the purpose of creating interactive websites which has become the standard for any modern browser. Originally, when Netscape started development in 1995, it wasn't even intended to get as big as it did, so it comes as no surprise that the programming language, which would become the future of the web, wasn't exactly properly future-proofed or secured for complex operations and architectures.

In the modern age of web development, JavaScript is no longer exclusively a front-end language. Wherever you go you will find JavaScript used in an application. Its usage has grown so much that there is now an incredibly large pool of available frameworks, technologies, and applications that you can use with the language. This though comes with a major problem, since the language powering so many systems today is a fairly harsh environment to work in, as it is filled with many problems ranging from minor inconveniences to major design issues that are impossible to ignore. The most egregious example of this is the type system of JavaScript. It provides neither type checks nor warnings, doesn't allow for objects to be matched against types and requires the user to always know what the value of a variable will be at runtime, making it a constant game of remembering and guessing.

Naturally, as a result, this has caused a lot of solutions to pop up, which all aim to resolve this issue. One of the most well-known and accepted solutions in this regard is TypeScript.

2.1.2 TypeScript - One of many solutions

TypeScript is as of now the most widely used alternative to JavaScript, or more accurately a super-set of it, allowing standard object-oriented functionality and compile-time type-checking similar to that present in Java or C#. In its core principle, TypeScript provides

everything that a developer needs for developing type-safe applications, as you can simply use the type annotations and let the TypeScript compiler check for your errors while working on your project. This though has certain limitations, as TypeScript is bound to the restrictions of a simple linter that aims to be fully compatible with JavaScript, no matter the circumstances. While that allows the developer to import any code from an old code base directly, it also heavily impacts and limits the functionality, which the TypeScript compiler can implement. As a result, the compiler is bound to the constraints of a language that is not even designed for type checking and type annotations. More specifically this means that all type checks are compile-time only and are not checked against at runtime, which means TypeScript works on a trust-based system, where the developer is often used as the root of trust.

Unchecked compile-time casts

As already mentioned TypeScript works on a compile-time-only basis, which does not allow for any runtime type checks. That also naturally means any standard functionality like casts can also not be checked for, since such type functionality requires the language to be able to reflect on its type structure during runtime. Given the fact though that casts, which allow the developer to narrow the type of a value down, are a necessity in everyday programs, TypeScript is forced to provide what you can call "trust-based casts". The developer can, like in any other language, specify what a specific value is expected to be, but unlike usual casts are primarily unchecked, meaning you can, if you want, cast anything to anything with no determined constraint.

While in principle this maintains the status quo and provides the developer with more freedom, it also opens up another challenge that must be looked out for when writing code. If one of those casts goes wrong and isn't valid, the developer will only know that at runtime and will have no assistance to fix it. To overcome this developers can themselves implement runtime type checks, which prevent type mismatches in ambiguous contexts. While it is a common approach, it is fairly impractical and adds a heavy burden on the developer as it requires constant maintenance and recurring rewrites to ensure the type checks are up-to-date and valid.

Let's look at an example:

```
1 class SuperClass {
2   name: string = "Super class";
3 }
4
5 class MiddleClass extends SuperClass {
```

```
6     superField: SuperClass = new SuperClass();
7
8     constructor() {
9       super();
10    }
11  }
12
13  class LowerClass extends MiddleClass {
14    classField: MiddleClass = new MiddleClass();
15
16    constructor() {
17      super();
18    }
19  }
20
21  const c1 = <MiddleClass>new SuperClass(); // Unchecked
      cast
22  console.log(c1.superField.name); // Runtime Error!
      Doesn't actually exist
23
24  const c2 = <LowerClass>new MiddleClass(); // Unchecked
      cast
25  console.log(c2.classField.superField.name); // Runtime
      Error! Doesn't actually exist
```

Here we have a simple example of an inheritance structure, where we access the properties of a child that is itself also another object. Due to the nature of TypeScript operations such as casts are mostly unchecked and usually work on the base of trusting the developer to know what they're doing. That means that in the example given above, the compiler does not realise that the operation the developer is performing is invalid and will result in a failure at runtime (can't access property "name", c1.superField is undefined). Furthermore, given that JavaScript only reports on such errors when a property on an undefined value is accessed, the undefined variable may go unused for a while before it is the cause of any problem. This leads to volatile code that can in many cases not be guaranteed to work unless the developer actively pays attention to such errors and makes sure that their code does not unintentionally force unchecked casts or other similar untyped operations.

Ambiguous dynamic data

Another similar issue occurs when dealing with dynamic or untyped data, which does not report on its structure and as such is handled as if it were a JavaScript value, where all type checks and security measures are disabled. This for one makes sense given the goal of ensuring compatibility with the underlying language, but it also creates another

major problem where errors regarding any-typed values can completely go undetected. Consequently, if we were to receive data from a client or server we can not ensure that the data we received is fully valid or corresponds to the expected pattern. This is a problem which neither has a proper workaround nor a solution in TypeScript.

For example:

```
interface Data {
  x: number;
  y: string;
  z: {
    z1: boolean;
  }
}

function receiveUserReq(): object {
  // ...
  return {
    x: "1",
    y: "2",
    z: true
  }
}

var data = <Data>receiveUserReq(); // Unsafe casting with unknown data
console.log(data.z.z1); // No Runtime Error! But returns "undefined"
```

For the most part, developers are expected to simply watch out for such cases and implement their own security measures. There are potential libraries which can be utilised to add runtime checks which check the data received, but such solutions require an entirely new layer of abstraction which must be managed manually by a developer. This additional boilerplate code also increases the complexity of a program and has to be actively maintained to keep working.

Good examples of technologies that provide runtime object schema matching are "Zod" (https://github.com/colinhacks/zod) and "joi" (https://github.com/hapijs/joi). Both are fairly popular and actively used by API developers who need to develop secure endpoints and ensure accurate request data. While they are a good approach to fixing the problem after the fact, they still create their own difficulties. We will examine these later in the implementation section, where we will more thoroughly compare Kipper's approach to other tools.

2.2 How could it have been better

2.2.1 Case study: Java

Java is a statically typed OOP programming language, which is next to C# one of the primary languages used throughout the world of programming. It runs on a VM-based architecture designed to allow the developer to deploy cross-platform applications and work with powerful dynamic object structures. Like other high-level languages, it provides reflection, a concept essential for the purpose of runtime type checking and validations. Unlike TypeScript, operations in the code are always checked during compile time and runtime if necessary, and can never be simply ignored by the developer. As such when you work with the language and deploy an application you can be sure that the casts and type operations are safe, or at least will have an error thrown in the case of failure. This is a heavy contrast to the entirely dynamic and type-less structure present in JavaScript, which doesn't provide such safeguards and relies on the developer for security.

2.2.2 Case study: Rust

Rust is a systems programming language designed to offer memory safety without a garbage collector. One of the standout features of Rust is its ownership system, which enforces strict rules for memory allocation and deallocation, preventing common bugs like null pointer dereferencing or data races in concurrent programming. This guarantees memory safety at compile-time without needing a runtime environment to manage memory, unlike languages such as Java and JavaScript, which use garbage collection to manage memory dynamically.

Rust's type system is strongly and statically typed, like Java, but it emphasizes immutability and borrowing concepts to manage data lifetimes and concurrency safely. Unlike Java, Rust does not have reflection, but it provides powerful meta-programming features via macros. Rust also promotes zero-cost abstractions, ensuring that high-level abstractions have no runtime overhead, making it a popular choice for applications requiring both performance and safety. Despite working on the basis of a completely different programming paradigm it still manages to be type-safe or more accurately memory-safe. The compiler makes sure that there are no ambiguities left that could potentially lead to runtime errors and provides absolute safety in a way that still allows a certain freedom to the developer.

2.2.3 Drawing comparisons to JavaScript

Unlike the two languages we've just described, JavaScript is rather unique in its design and structure. As already mentioned, there is no proper reflection system, enforced type checks or type safety when running code, only really throwing errors when there is no other way around it. Moreover, you can say that JavaScript has no design or structure at all, and was more conceptualised as a fully dynamic type-less language with no OOP support in mind. This has caused quite a few problems in the years following the original version of JavaScript, as it has more and more developed into an OOP language while not providing any proper type functionality commonly present in such systems. Even languages like Python, which is also a dynamic interpreted language, provide static type hints and checks to ensure proper type safety when writing code.

Nonetheless, as JavaScript is currently one of the most important languages out there, the system can under no circumstances be changed as it would break backwards compatibility with previous systems and destroy the web as we know it today. This has caused quite a dilemma, which persists until today. Many tools like TypeScript have been developed since then and are seen as the de-facto solution for these problems, but it's a rather bad solution given all the current restraints, unavoidable edge cases and vulnerabilities that can be easily introduced.

2.3 Tackling the issue at its core

As we have already mentioned, JavaScript is a language that can under no circumstances be changed or it would mean that most websites would break in newer browser versions. This phenomenon is also often described as "Don't break the web", the idea that any new functionality must incorporate all the previous standards and systems to ensure that older websites work and look the same. Naturally this also then extends to TypeScript, which has at its core a standard JavaScript system that can also not be changed or altered to go against the ECMAScript standard. Consequently, the only real way to provide a safe development environment to the developer is to build on top of JavaScript and extend the standard with a custom unofficial system implementing the required structures.

Here comes Kipper into play, which is such a language implementing a custom system that is translated to JavaScript or TypeScript after the fact. With an additional runtime and non-standard syntax, Kipper can provide runtime type checks and fill in

all the holes that TypeScript can not fill with its compile-time-only system. This is particularly powerful as it allows the system to go beyond the JavaScript standards and implement structures which are in line with the requirements of a modern developer. Subsequently, the developer can rely on the language to secure the code and make sure that no dynamic structures go unchecked or fall through the type system due to edge cases.

3 Technology

4 Implementation

Siehe tolle Daten in Tab. 1.

Siehe und staune in Abb. 1. Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Donec odio elit, dictum in, hendrerit sit amet, egestas sed, leo. Praesent feugiat sapien aliquet odio. Integer vitae justo. Aliquam vestibulum fringilla lorem. Sed neque lectus, consectetuer at, consectetuer sed, eleifend ac, lectus. Nulla facilisi. Pellentesque eget lectus. Proin eu metus. Sed porttitor. In hac habitasse platea dictumst. Suspendisse eu lectus. Ut mi mi, lacinia sit amet, placerat et, mollis vitae, dui. Sed ante tellus, tristique ut, iaculis eu, malesuada ac, dui. Mauris nibh leo, facilisis non, adipiscing quis, ultrices a, dui.

	Regular Customers	**Random Customers**
Age	20-40	>60
Education	university	high school

Table 1: Ein paar tabellarische Daten

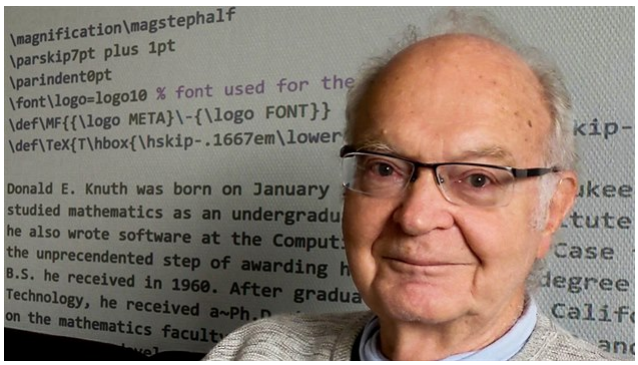

Figure 1: Don Knuth – CS Allfather

Morbi luctus, wisi viverra faucibus pretium, nibh est placerat odio, nec commodo wisi enim eget quam. Quisque libero justo, consectetuer a, feugiat vitae, porttitor eu, libero. Suspendisse sed mauris vitae elit sollicitudin malesuada. Maecenas ultricies eros sit amet ante. Ut venenatis velit. Maecenas sed mi eget dui varius euismod. Phasellus aliquet volutpat odio. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Pellentesque sit amet pede ac sem eleifend consectetuer. Nullam elementum, urna vel imperdiet sodales, elit ipsum pharetra ligula, ac pretium ante justo a nulla. Curabitur tristique arcu eu metus. Vestibulum lectus. Proin mauris. Proin eu nunc eu urna hendrerit faucibus. Aliquam auctor, pede consequat laoreet varius, eros tellus scelerisque quam, pellentesque hendrerit ipsum dolor sed augue. Nulla nec lacus. Dann betrachte den Code in Listing 1.

Listing 1: Some code

```
# Program to find the sum of all numbers stored in a
    list (the not-Pythonic-way)

# List of numbers
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

# variable to store the sum
sum = 0

# iterate over the list
for val in numbers:
    sum = sum+val

print("The sum is", sum)
```

5 Compiler Reference

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

6 Demo & Showcase

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

7 Conclusion & Future

Bibliography

List of Figures

List of Tables

Source Code References

Appendix