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

Supervisor:
Peter Bauer

Project Partner:
Dr. 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.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

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.

Zusammenfassung

Kurze Zusammenfassung unserer großartigen Arbeit. Auf Deutsch. Dies ist das einzige Mal, dass wir ein Bild in den Text einfügen müssen. Das Bild sollte in irgendeiner Weise Ihre Diplomarbeit darstellen. Dies ist untypisch für wissenschaftliche Arbeiten, aber von den zuständigen Stellen vorgeschrieben.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

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.

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. JavaScript can be found in a number of various applications that are no longer simply restricted to a browser. 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 challenging 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 typechecking 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.

Listing 1: Unchecked compile-time casts in TypeScript

```
class SuperClass {
```

```
  name: string = "Super class";
}

class MiddleClass extends SuperClass {
  superField: SuperClass = new SuperClass();

  constructor() {
    super();
  }
}

class LowerClass extends MiddleClass {
  classField: MiddleClass = new MiddleClass();

  constructor() {
    super();
  }
}

const c1 = <MiddleClass>new SuperClass(); // Unchecked cast
console.log(c1.superField.name); // Runtime Error! Doesn't actually exist

const c2 = <LowerClass>new MiddleClass(); // Unchecked cast
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 2:

Listing 2: Ambiguous dynamic data in TypeScript

```
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

3.1 Development Language

3.1.1 Selection criteria and weighing the options

3.1.2 Option - C++

3.1.3 Option - Java

3.1.4 Option - TypeScript

3.1.5 Result

3.2 Parser & Lexer Generator

3.2.1 Selection criteria and weighing the options

3.2.2 Option - Antlr4

3.2.3 Option - Coco

3.2.4 Result

4 Implementation

4.1 Internal Compiler

4.2 Semantic Analysis

4.3 Type Analysis

4.4 Output Generation

4.4.1 Introduction

4.4.2 Algorithms used for Output Generation

4.4.3 Types of Generated Statements

4.4.4 Differences between the Target Languages

4.4.5 Stylistic Choices

4.5 Type System

4.6 Integrated Runtime

4.6.1 Runtime Type Concept

The primary goal of the Kipper runtime type system is to allow the comparison of untyped objects with clearly defined types, such as primitives, classes and interfaces which allows to remove any ambiguity that could potentially lead to errors. When the user code is generated, all user-defined interfaces are translated into runtime types, which store the information necessary to do a type check against a given object. This allows the compiler to generate required checks against these runtime types in places where the user is performing a cast or match operation using an interface. In the case of user-defined classes, the compiler has no need to translate any structural information, as classes rely on a prototype system which means they can use simple "instanceof" (prototype equality) checks. Alongside the code generation for user-defined structures, the compiler also inserts the built-in primitive structures and generics such as the "Array" and "Func" type into the output code. The implementation for a simple runtime type can be seen in 3.

With the exception of interfaces and classes, types primarily differentiate themselves using their name, which means baseline checks are simply performed using equality checks. In more advanced use cases, such as interfaces, the required fields and methods are also taken into account; these represent the minimum blueprint an object must implement to match the given type. In this way, Kipper implements the same duck-typing system also present in TypeScript. To be future-proof in regards to inheritance, the "baseType" property is also available, although it remains unused as of today.

Listing 3: The structure of a runtime type

```
class KipperType {
  constructor(name, fields, methods, baseType = null,
     customComparer = null) {
    this.name = name;
    this.fields = fields;
    this.methods = methods;
    this.baseType = baseType;
    this.customComparer = customComparer;
  }

  accepts(obj) {
    if (this === obj) return true;
    return obj instanceof KipperType &&
       this.customComparer ? this.customComparer(this,
       obj) : false;
  }
}
```

Kipper uses a nominal type system for primitives. This means, that two types are compared by their name. Classes, on the other hand, are compared using JavaScripts prototype chain. Therefore they are equal, only if they are the same instance. To match interfaces and types against objects of an unknown type, it was necessary to implement duck typing. Duck typing compares the properties and methods of two objects for equality and ignores the type name. Both approaches and a third one will be further explained in chapter 4.6.2.

To check for type compatibility, every type brings its own comparer function. In case no comparer is supplied, it returns false.

A snippet of the implementation of the builtin runtime types can be seen in 4. Nominal comparison is implemented in the "___type_undefined" and the "___type_str" constants. "Any" is an exception, as it is necessary to cast it to a different type in order to return a useful value. By design "any" is as useless as possible, in order to force the developer into typechecking it.

Listing 4: The builtin runtime types

```
const __type_any =
  new KipperType('any', undefined, undefined);

const __type_undefined =
  new KipperType('undefined', undefined, undefined,
     undefined, (a, b) => a.name === b.name);

const __type_str =
  new KipperType('str', undefined, undefined,
     undefined, (a, b) => a.name === b.name);
```

4.6.2 Runtime Type Implementations in other Languages

Nominal type systems are used in most modern object-orientated programming languages like Java and C#. In these systems, types are compared by their name. Furthermore they are treated as equal, if one type is a subtype of the other one, as can bee seen in example 5. Here a Programmer is an Employee, but not the other way around. This means, that Programmers have all the properties and methods an Employee has, but can additionally bring their own. The relationships are inherited, so a SeniorDeveloper is still an Employee, and a Programmer at the same time. Even though the Senior Developer adds no new functionality to the Programmer, it is not treated the same. Nominal typing improves code readability and maintainability, due to the explicit inheritance declaration. On the other hand, this increases code redundancy for similar, but not related structures.

Listing 5: Example of nominal typing in java

```
class Employee {
  public float salary;
}

class Programmer extends Employee {
  public float bonus;
}

class SeniorDeveloper extends Programmer { }
```

Structural type systems compare types by their structure. This means, if two differently named types have the same properties and methods, then they are the same type. An example of this would be OCaml, with it's object subsystem being typed this way. Classes in OCaml only serve as functions for creating objects. In example 6 there is a function that required a function "speak" returning the type "string". Both the "dog" object as well as the "cat" object fulfill this condition, therefore both are treated equal. Most importantly, these compatibility checks happen at compile time, as OCaml is a static language. Structural typing allows for a lot of flexibility as it promotes code reuse. Furthermore it avoids explicit inheritance hierarchies.

Listing 6: Example of structural typing in Ocaml

```
let make_speak (obj : < speak : string >) =
obj#speak

let dog = object
method speak = "Woof!"
end

let cat = object
method speak = "Meow!"
end

let () =
print_endline (make_speak dog);
print_endline (make_speak cat);
```

Duck Typing is the usage of a structural type system in dynamic languages. It is the practical application of the Duck Test, therefore if it quacks like a duck, and walks like a duck, then it must be a duck. In programming languages this means that if an object has all methods and properties required by a type, then it is that type. The most prominent language utilizing Duck Typing is JavaScript. As can be seen in example 7, the duck and the person have the same methods and properties, henceforth they are of the same type. The dog object on the other hand does not implement the "quack" function, which equates to not being a duck. Duck typing simplifies the code by removing type constraints, while still encouraging polymorphism without complex inheritance. Due to the type checking happening at runtime, errors only surface at this time, potentially causing failures. For developers, the lack of type information makes understanding the code harder and more tedious to maintain.

Listing 7: Example of duck typing in JavaScript

```
const duck = {
```

```
  quack: function () {
    console.log("Quack!");
  }
};

const person = {
  quack: function () {
    console.log("I'm a person but I can quack!");
  }
};

const dog = {
  bark: function () {
    console.log("Woof!");
  }
};
```

4.6.3 Runtime Generation for Builtin Types

4.6.4 Runtime Generation for Interfaces

The generation of runtime interface types allows for dynamic type checking against interfaces in the target languages. This process is managed by the !!!RuntimeTypes-Generator!!! class, which is called by the code-generator and adds a JavaScript object to the compiled code that represents the structure of the interface. It therefore includes the interface's methods and properties with their respective types.

Listing 8: Example interfaces in the Kipper language

```
interface Car {
  brand: str;
  honk(volume: num): void;
  year: num;
}

interface Person {
  name: str;
  age: num;
  car: Car;
}
```

At compile time, the generator function iterates over the interface's members and differentiates between properties and methods. The function keeps separate lists of already generated runtime representations for properties and methods.

If it detects a property, the type and semantic data of the given property is extracted. When the property's type is a built in type, the respective runtime type of the previous section is used. If not, we can assume the property's type is another interface, which has it's own runtime type already. This data is stored in "___kipper.Property", which is finally added to the list of generated runtime type representation

In case a method is detected, the generator function fetches the return type and the method's name. If the method has arguments, the name and type of each argument gets evaluated and stored in the same internal Kipper type as above, because the type representation of properties and arguments is equivalent. After that, it gets added to the list of type representations as well.

Listing 9: The runtime representation of the previous interfaces

```
const __intf_Car = new __kipper.Type(
  "Car",
  [
    new __kipper.Property("brand", __kipper.builtIn.str),
    new __kipper.Property("year", __kipper.builtIn.num),
  ],
  [
    new __kipper.Method("honk", __kipper.builtIn.void,
      [
        new __kipper.Property("volume",
           __kipper.builtIn.num),
      ]
    ),
  ]
);

const __intf_Person = new __kipper.Type(
  "Person",
  [
    new __kipper.Property("name", __kipper.builtIn.str),
    new __kipper.Property("age", __kipper.builtIn.num),
    new __kipper.Property("car", __intf_Car),
  ],
  []
);
```

The generated output code in the target language is a string array, which later gets concatenated into a single output string as can bee seen in 10. The properties and methods are wrapped into a Kipper type object that represents the final interface runtime type. It is saved into a constant with a ___intf_ prefix. This runtime type-checking object code is then placed under the compiled interface's code. Notable usages for runtime typechecking include the "matches" operator 4.6.5 and the typeof operator 4.6.6.

Listing 10: Code generation statement

```
return [
  [
    "const ",
    identifier,
    ` = new ${TargetJS.internalObjectIdentifier}.Type("`
      + interfaceName + '"',
    ", [",
    ...propertiesWithTypes,
    "], [",
    ...functionsWithTypes,
    "])",
  ],
];
```

4.6.5 Matches operator

4.6.6 Typeof operator

5 Compiler Reference

5.1 Compiler API

5.2 Target API

5.3 Shell CLI

6 Demo & Showcase

6.1 Working example in the web

6.2 Working example using Node.js

7 Conclusion & Future

Bibliography

List of Figures

List of Tables

List of Source Code Snippets

Appendix