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

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

JavaScript, originally developed by Netscape in 1995 to enable interactive web pages, has become the foundational programming language for modern web browsers with 60% of developers using the language in their profession as of 2023. [1] While its initial scope was limited to enhancing the functionality of websites, JavaScript has since evolved into a versatile language that serves as the foundation for diverse applications, including those outside traditional browser environments. However, this rapid expansion was not accompanied by fundamental changes to the language's initial design, leading to inherent limitations and challenges when addressing complex, large-scale systems. In modern web development, JavaScript's role extends far beyond front-end scripting. Its omnipresence is reflected in its adoption across back-end platforms (e.g. Node.js), desktop applications (e.g. Electron), and mobile development frameworks (e.g. React Native). Accompanying this growth is a vast ecosystem of libraries, frameworks, and tools that offer developers flexibility in solving specific challenges. Despite these advantages, the language presents significant difficulties for developers. One of the most prominent challenges is JavaScript's type system, which lacks static typing, runtime type checks, and robust safeguards against type mismatches. For example, developers must carefully manage the dynamic nature of variables, often encountering runtime errors caused by implicit type coercion or unexpected values. These issues complicate debugging and increase the risk of introducing bugs into production systems.

As a result, various solutions have emerged, aiming to enhance JavaScript's reliability and developer experience. One of the most well-known and accepted solutions in this regard is TypeScript.

2.1.2 TypeScript - One of many solutions

TypeScript has emerged as the most widely adopted enhancement to JavaScript, functioning as a statically typed superset of the language. It introduces features such as object-oriented programming constructs and compile-time type checking, aligning its capabilities with those of traditionally typed languages like Java or C#. By providing type annotations and a robust compilation process, TypeScript enables developers to build safer applications compared to their JavaScript counterparts. Errors related to type mismatches, for instance, can be identified during development, reducing the likelihood of runtime failures and improving overall code reliability.

Despite its advantages, TypeScript is constrained by its core design philosophy of maintaining full compatibility with JavaScript. This approach allows developers to easily integrate TypeScript with existing JavaScript codebases, promoting incremental adoption. However, it also imposes limitations on the language's capabilities. For example, because JavaScript was not originally designed with type safety in mind, the TypeScript compiler operates as a static analysis tool, enforcing type rules only at compile time. This design choice ensures compatibility but leaves runtime type enforcement unaddressed. Consequently, developers must rely on a "trust-based" system, wherein the correctness of types is assumed during runtime based on the accuracy of their compile-time annotations.

These constraints highlight the challenges inherent in adapting a dynamically typed language to support static typing. While TypeScript significantly mitigates many of JavaScript's shortcomings, its reliance on compile-time type checking alone limits its ability to provide comprehensive runtime guarantees, requiring developers to remain vigilant when integrating with dynamically typed JavaScript components.

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

```
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 that does not have a workaround or a solution in TypeScript.

For example 2:

Listing 2: Ambiguous dynamic data in TypeScript

```
1  interface Data {
2    x: number;
3    y: string;
4    z: {
5      z1: boolean;
6    }
7  }
8
9  function receiveUserReq(): object {
10   // ...
11   return {
12     x: "1",
13     y: "2",
14     z: true
15   }
16 }
17
```

```
18 var data = <Data>receiveUserReq(); // Unsafe casting
      with unknown data
19 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" [2] and "joi" [3]. 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

TypeScript has emerged as the most widely adopted enhancement to JavaScript, functioning as a statically typed superset of the language. It introduces features such as object-oriented programming constructs and compile-time type checking, aligning its capabilities with those of traditionally typed languages like Java or C#. By providing type annotations and a robust compilation process, TypeScript enables developers to build type-safe applications. Errors related to type mismatches, for instance, can be identified during development, reducing the likelihood of runtime failures and improving overall code reliability. Despite its advantages, TypeScript is constrained by its core design philosophy of maintaining full compatibility with JavaScript. This approach allows developers to seamlessly integrate TypeScript with existing JavaScript codebases, promoting incremental adoption. However, it also imposes limitations on the language's capabilities. For example, because JavaScript was not originally designed with type safety in mind, the TypeScript compiler operates as a static analysis tool, enforcing type rules only at compile time. This design choice ensures compatibility but leaves runtime type enforcement unaddressed. Consequently, developers must rely on a "trust-based"

system, wherein the correctness of types is assumed during runtime based on the accuracy of their compile-time annotations. These constraints highlight the challenges inherent in adapting a dynamically typed language to support static typing. While TypeScript significantly mitigates many of JavaScript's shortcomings, its reliance on compile-time type checking alone limits its ability to provide comprehensive runtime guarantees, requiring developers to remain vigilant when integrating with dynamically typed JavaScript components.

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 most effective approach to ensuring a safe development environment for developers is to build upon JavaScript by extending its standard functionalities through a custom, unofficial system that incorporates the necessary structures and safety measures.

This is where Kipper comes into play—a language that implements a custom system, which is later transpiled into JavaScript or TypeScript. By incorporating an additional runtime and non-standard syntax, Kipper enables runtime type checks, addressing gaps left by TypeScript's compile-time-only system. This approach is particularly advantageous, as it allows the system to extend beyond JavaScript standards, introducing structures tailored to the requirements of modern programs. Accordingly, developers can rely on Kipper to enhance code security and ensure that no dynamic structures remain unchecked or bypass 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

```
1  class KipperType {
2    constructor(name, fields, methods, baseType = null,
         customComparer = null) {
3      this.name = name;
4      this.fields = fields;
5      this.methods = methods;
6      this.baseType = baseType;
7      this.customComparer = customComparer;
8    }
9
10   accepts(obj) {
11     if (this === obj) return true;
12     return obj instanceof KipperType &&
           this.customComparer ? this.customComparer(this,
           obj) : false;
13   }
14 }
```

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

```
1   const __type_any =
2     new KipperType('any', undefined, undefined);
3
4   const __type_undefined =
5     new KipperType('undefined', undefined, undefined,
        undefined, (a, b) => a.name === b.name);
6
7   const __type_str =
8     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

```
1 class Employee {
2   public float salary;
3 }
4
5 class Programmer extends Employee {
6   public float bonus;
7 }
8
9 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 Interfaces

The generation of runtime interface types allows for dynamic type checking against interfaces in the target languages. This process is managed by the RuntimeTypesGenerator 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

```
1  const __intf_Car = new __kipper.Type(
2    "Car",
3    [
4      new __kipper.Property("brand", __kipper.builtIn.str),
5      new __kipper.Property("year", __kipper.builtIn.num),
6    ],
7    [
8      new __kipper.Method("honk", __kipper.builtIn.void,
9        [
10         new __kipper.Property("volume",
               __kipper.builtIn.num),
11       ]
12     ),
13   ]
14 );
15
16 const __intf_Person = new __kipper.Type(
17   "Person",
18   [
19     new __kipper.Property("name", __kipper.builtIn.str),
20     new __kipper.Property("age", __kipper.builtIn.num),
21     new __kipper.Property("car", __intf_Car),
22   ],
23   []
24 );
```

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

```
1  return [
```

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

4.6.4 Runtime Generation for Builtin Types

Builtin runtime types are the used as parts of more complex constructs like interfaces. Comparing builtin runtime types at runtime happens by comparing their references, as they are uniquely defined at the start of the target language output in a global scope. They are implemented like the examples in 4. In addition to these types there are also builtin implementations for generic types. These include arrays and functions.

Listing 11: Generic builtin types

```
const __type_Array = new KipperGenericType('Array',
    undefined, undefined, {T: __type_any});
const __type_Func = new KipperGenericType('Func',
    undefined, undefined, {T: [], R: __type_any});
```

As can be seen in example 11, they are implemented using a special KipperGenericType class. This class extends the KipperType and includes an additional field for generic arguments. Most importantly, it includes the method changeGenericTypeArguments, which allows for modifying a types generic arguments at runtime. It is used for example in lambdas and arrays, where the builtin generic runtime type is used and the modified to include the required types. When, for example an array is initialized, it first gets assigned the "___type_Array" runtime type, which is then modified by the changeGenericTypeArguments method. Arrays for example use the specified type for their elements, whilst functions require a return type as well as an array of arguments, which have types as well. The "___type_Func" type is used by lambdas, which are user defined functions without a name.

Listing 12: Generic Kipper Type

```
class KipperGenericType extends KipperType {
```

```
2     constructor(name, fields, methods, genericArgs,
          baseType = null) {
3       super(name, fields, methods, baseType);
4       this.genericArgs = genericArgs;
5     }
6     isCompatibleWith(obj) {
7       return this.name === obj.name;
8     }
9     changeGenericTypeArguments(genericArgs) {
10      return new KipperGenericType(
11      this.name,
12      this.fields,
13      this.methods,
14      genericArgs,
15      this.baseType
16      );
17    }
18  }
```

Other builtin types are error type. They are used in the error handling system to represent errors. The base "KipperError" type has a name property and extends the target language's error type as can be seen in code snippet 13. Additional error types inherit of this base type. The "KipperNotImplementedError" for example just changes the name of the error. As of now it is not possible for the user to create custom error types, due to inheritance not being implemented yet . The user will have to use the builtin error type and change it's name.

Listing 13: Kipper error types

```
1  class KipperError extends Error {
2    constructor(msg) {
3      super(msg);
4      this.name = "KipError";
5    }
6  }
7  class KipperNotImplementedError extends KipperError {
8    constructor(msg) {
9      super(msg);
10     this.name = "KipNotImplementedError";
11   }
12 }
```

4.6.5 Matches operator

The main feature of the Kipper programming language is it's runtime type comparison. There are multiple ways to compare objects at runtime. The instanceof operator compares by reference. This means, that an object is an instance of a class, when

there is a reference to that class. This works by utilizing JavaScripts prototype system. Another way to compare objects would be comparison by structure. This aproach treats two objects as equal, when they share the same structure, as in the same properties and methods. Kipper allows for comparison with both options. Comparison by reference is implemented using the 'instanceof' operator and only utilized for comparing classes. Comparison by structure is what we call 'matching' and is used for primitives and interfaces. Matching array and function types is not yet supported in Kipper. The 'matches' operator in Kipper implements this type of comparison.

Listing 14: The Kipper matches operator

```
interface Y {
  v: bool;
  t(gr: str): num;
}

interface X {
  y: Y;
  z: num;
}

var x: X = {
  y: {
    v: true,
    t: (gr: str): num -> {
      return 0;
    }
  },
  z: 5
};

var res: bool = x matches X; // true
```

As can be seen in example 14, the matches operator can compare interfaces by properties and methods. It takes two arguments, an object and a type which it should match. Properties are compared recursivly and methods are compared by name, arguments and return type.

Comparison works by iterating over the methods and properties. When iterating over the properties, it checks for the property's name being in the type it should check against 15. The order of properties does not matter. When the name was found, it checks for type equality. This checking is done using the aforementioned runtime types and nominal type comparison. In case a non-primitive is detected as the properties type, the matches function will be recursively executed on non-primitive.

Listing 15: Matches operator property comparison

```
for (const field of pattern.fields) {
  const fieldName = field.name;
  const fieldType = field.type;

  if (!(fieldName in value)) {
    return false;
  }

  const fieldValue = value[fieldName];
  const isSameType = __kipper.typeOf(fieldValue) ===
     field.type;

  if (primTypes.includes(field.type.name) &&
     !isSameType) {
    return false;
  }

  if (!primTypes.includes(fieldType.name)) {
    if (!__kipper.matches(fieldValue, fieldType)) {
      return false;
    }
  }
}
```

After checking the properties, the matches expression iterates over the methods. It first searches for the method name in the target type. If found, it compares the return type. Then each argument is compared by name. As the methods signatures need to be exactly the same, the amount of parameters is compared as well. This ensures, that one method can not be a subset of the other one.

Listing 16: Matches operator method comparison

```
for (const field of pattern.methods) {
  const fieldName = field.name;
  const fieldReturnType = field.returnType;
  const parameters = field.parameters;

  if (!(fieldName in value)) {
    return false;
  }

  const fieldValue = value[fieldName];
  const isSameType = fieldReturnType ===
     fieldValue.__kipType.genericArgs.R;

  if (!isSameType) {
    return false;
  }

```

```
17    const methodParameters =
          fieldValue.__kipType.genericArgs.T;
18
19    if (parameters.length !== methodParameters.length) {
20      return false;
21    }
22
23    let count = 0;
24    for (let param of parameters) {
25      if (param.type.name !==
            methodParameters[count].name) {
26        return false;
27      }
28      count++;
29    }
30  }
```

When none of these condition is false, the input object matches the input type and they can be seen as the equal.

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

[1] JetBrains s.r.o. (2024) The State of Developer Ecosystem in 2023 Infographic. Available online: https://www.jetbrains.com/lp/devecosystem-2023/javascript/

[2] Colin McDonnell. (2024) zod. Available online: https://github.com/colinhacks/zod

[3] hapi.js. (2024) joi. Available online: https://github.com/hapijs/joi

List of Figures

List of Tables

List of Source Code Snippets

Appendix