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 University

Leonding, April 4, 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 **1**

2 Background **2**

2.1 Dissecting the current issues 2

2.1.1 The JavaScript problem 2

2.1.2 TypeScript - One of many solutions 5

2.2 How could it have been better 8

2.2.1 Case study: Java 8

2.2.2 Case study: Rust 9

2.2.3 Drawing comparisons to JavaScript 10

2.3 Tackling the issue at its core 10

3 Technology **12**

3.1 Development Language 12

3.1.1 Selection criteria and weighing the options 12

3.1.2 Option - C++ 12

3.1.3 Option - Java 12

3.1.4 Option - TypeScript 12

3.1.5 Result 12

3.2 Parser & Lexer Generator 12

3.2.1 Selection criteria and weighing the options 12

3.2.2 Option - Antlr4 12

3.2.3 Option - Coco 12

3.2.4 Result 12

4 Implementation **13**

4.1 Internal Compiler 13

4.2 Semantic Analysis 13

4.3 Type Analysis 13

4.4 Output Generation . . . 13
4.4.1 Introduction . . . 13
4.4.2 Algorithms used for Output Generation . . . 13
4.4.3 Types of Generated Statements . . . 13
4.4.4 Differences between the Target Languages . . . 13
4.4.5 Stylistic Choices . . . 13
4.5 Type System . . . 13
4.6 Integrated Runtime . . . 13
4.6.1 Runtime Type Concept . . . 13
4.6.2 Runtime Type Implementations in other Languages . . . 14
4.6.3 Runtime Base Type . . . 17
4.6.4 Runtime Built-in Types . . . 18
4.6.5 Runtime Errors . . . 19
4.6.6 Runtime Generation for Interfaces . . . 20
4.6.7 Matches operator for Interfaces . . . 21
4.6.8 Typeof operator . . . 24

5 Compiler Reference **25**
5.1 Compiler API . . . 25
5.2 Target API . . . 25
5.3 Shell CLI . . . 25

6 Demo & Showcase **26**
6.1 Working example in the web . . . 26
6.2 Working example using Node.js . . . 26

7 Conclusion & Future **27**

Glossary **V**

Bibliography **VI**

List of Figures **VII**

List of Tables **VIII**

List of Source Code Snippets **IX**

Appendix **X**

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. This success has been largely made possible by its versatile and modifiable nature allowing the language to take many shapes with a relatively consistent and easy-to-use syntax.

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.

Type Ambiguity and Uncontrolled Flexibility

JavaScript's dynamic typing provides developers with flexibility and expressiveness in their code. However, this flexibility can also lead to ambiguity and unintended behavior. Unlike statically typed languages, JavaScript does not enforce type constraints, meaning variable types are determined at runtime. It permits almost any value to be assigned to any variable without restrictions, relying solely on the developer to manage type consistency. This lack of enforcement makes the language prone to errors caused by implicit type coercion or unexpected values. Additionally, JavaScript does not inherently

handle type errors and assumes that most operations are valid, even providing special cases for operations that would typically be considered invalid.

For instance:

- A variable intended to store a number can unexpectedly hold a string due to developer oversight or API misuse, as can be seen in listing 1.

Listing 1: Unchecked variable assignments due to missing type definitions

```
1 let id; // Variable allowing any value
2
3 ...
4
5 let resp = resp.json(); // Object: { id: "1234", ... }
6 id = resp.id; // Assigns a string even though that was not intended
```

- Implicit conversions, such as treating ‘null‘ or ‘undefined‘ as valid inputs in arithmetic operations, often yield confusing results, as can be seen in listing 2.

Listing 2: Broad ability to perform "invalid" operations despite clear error case

```
1 let discountRate = cart.appliedDiscount; // Let's assume it's actually "applied_discount" not "appliedDiscount" so it returns 'undefined'
2
3 return price * (1 - discountRate); // -> yields NaN (Not A Number)
```

Such flexibility complicates debugging, as issues may only surface during runtime. This increases the risk of critical bugs making it into production and requires developers to rely heavily on additional tooling or rigorous testing to mitigate the risks inherent in dynamic typing.

Runtime-Bound Error Handling & Catching

Error handling in JavaScript is largely runtime-bound, except for syntax errors, and relies on tools like try-catch blocks and asynchronous error handling with Promise chains. This is because JavaScript is an interpreted language, meaning the program does not perform checks before executing each line. As a result, developers lack pre-execution validation.

While try-catch blocks can handle most errors, typically those defined by the developer, JavaScript allows certain operations that would be illegal in other languages, like

accessing non-existent properties by simply returning undefined instead of an error. This can make the underlying cause of an issue difficult to identify, with the code potentially failing at a different location, sometimes long after the original problem has occurred.

Examples include:

- Accessing a missing property, which returns ‘undefined‘, and later receiving an error due to ‘undefined‘ not being an object, as demonstrated in listing 3.

Listing 3: Accessing a missing property returns undefined which later causes an error

```
// Let's assume this is some kind of API response that incorrectly returned some data
let user = {
  name: "Alice"
};

...

const userAddress =  user.address; // Undefined property ('address' does not exist), returns simply 'undefined'

...

let shippingCost = getShippingCost(address.city); // -> TypeError: Cannot read property 'city' of undefined
```

- Misaligned function arguments, such as passing an object where a string was expected, going unnoticed until execution, as illustrated in listing 4.

Listing 4: Misaligned function arguments going unnoticed

```
function greet(name) {
  console.log("Hello, " + name + "!");
}

...

let user = { firstName: "Alice" };

greet(user); // Unintended behavior: "Hello, [object Object]!"
```

Modern approaches

These limitations force a defensive programming approach, requiring the developer to anticipate and safeguard against potential failures. While third-party tools such as linters or test frameworks can help identify issues, they are generally not equivalent to built-in, language-level type and error guarantees.

Given though that JavaScript is so prominent and can hardly be replaced in the modern web and server-side space, alternative solutions have been developed in response to these and other challenges to improve JavaScript's reliability and ease of use. One of the most prominent and widely adopted of these is TypeScript, which like the topic of this thesis, implements a transpilation-based language with independent libraries and functionality building on top of the existing JavaScript environment.

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 5.

Listing 5: 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 6:

Listing 6: 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" [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 untyped values to be compared with defined types, such as primitives, arrays, functions, classes, and interfaces, removing any ambiguities that could cause errors. During code generation, all user-defined interfaces are converted into runtime types that store the information needed to perform type checks. These are then utilised alongside the built-in runtime types, such as "num", "str" or "obj", to enable the compiler to add necessary checks and runtime references for any cast, match or typeof operation.

With the exception of interfaces, classes, and generics, types are primarily distinguished by their names. In these cases, type equality checks are performed using nominal comparisons, where the name acts as a unique identifier within the given scope e.g. type "num" is only assignable to "num". For more complex structures, additional information—such as members or generic parameters—is also considered.

In the case of interfaces, the names and types of fields and methods are used as discriminators. These fields and methods represent the minimum blueprint that an object must implement to be considered compatible with the interface and thus "assignable." In this regard, Kipper adopts the same duck-typing approach found in TypeScript.

For generics, which include "Array<T>" and "Func<T..., R>", the identifier is used alongside the provided generic parameters to determine assignability. This ensures that when one generic is assigned to another, all parameters must match. For instance, "Array<num>" cannot be assigned to "Array<str>" and vice versa, even if their overall structure is identical.

For user-defined classes, the compiler relies on the prototype to serve as the discriminator. In practice, this behaviour is similar to that of primitives, as different classes cannot be assigned to each other.

To ensure future compatibility with inheritance, Kipper also includes a "baseType" property, which allows types to be linked in an inheritance chain. However, this feature is currently unused.

4.6.2 Runtime Type Implementations in other Languages

Nominal Type Systems

Nominal type systems are used in most modern object-orientated programming languages like Java and C#. In these systems, types are identified by their unique names and can only be assigned to themselves. Additionally, two types are considered compatible, if one type is a subtype of the other one, as can bee seen in listing 7. Here a "Programmer" is an "Employee", but not the other way around. This means that "Programmer" instances have all the properties and methods an "Employee" has while also having additional ones specific to "Programmer". The relationships are as such 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 or even identical but not related structures.

Listing 7: 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

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 its object subsystem being typed this way. Classes in OCaml only serve as functions for creating objects. In example 8 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 8: 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 Typed Systems - Duck Typing

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 of that type. The most prominent language utilizing Duck Typing is TypeScript. As can be seen in listing 9, 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.

Listing 9: Example of duck typing in TypeScript

```
interface Duck {
  quack(): void;
}

const duck: Duck = {
  quack: function () {
    console.log("Quack!");
  }
};

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

const dog: Duck = {
  bark: function () {
    console.log("Woof!");
  }
}; // <- causes an error in the static type checker
```

Given that duck typing allows dynamic data to be easily checked and assigned to any interface, Kipper adopts a similar system to that of TypeScript but introduces notable differences in how interfaces behave and how dynamic data is handled. For instance, casting an "any" object to an interface in Kipper will result in a runtime error if the object does not possess all the required members. In contrast, TypeScript permits such an operation without performing any type checks at runtime.

4.6.3 Runtime Base Type

In practice, all user-defined and built-in types inherit from a basic "KipperType" class in the runtime environment. This class is a simple blueprint of what a type could do and what forms a type may take on. A simple version of such a class can be seen in listing 10.

Listing 10: The structure of a runtime type

```
1  class KipperType {
2    constructor(name, fields, methods, baseType =
       undefined, customComparer = undefined) {
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 }
```

As already mentioned types primarily rely on identifier checks to differentiate themselves from other types. Given though that there are slight differences in how types operate, they generally define themselves with what they are compatible using a comparator function. This comparator is already predefined for all built-ins in the runtime library and any user structures build on top of the existing rules established in the library.

Type "any" is an exception and is the only type that accepts any value you provide. However, assigning "any" to anything other than "any" is forbidden and it is necessary to cast it to a different type in order to use the stored value. By design "any" is as useless as possible, in order to force the developer into typechecking it.

Furthermore, classes are also exempt from this comparator behaviour, as classes behave like a value during runtime and provide a prototype which can simply be used to check if an object is an instance of that class.

4.6.4 Runtime Built-in Types

Built-in runtime types serve as the foundation of the type system and make up the parts of more complex constructs like interfaces. Built-in runtime types are compared at runtime by comparing their references, as they are uniquely defined at the start of the output code and available in the global scope. The implementations of such structures can be seen in the listing 11 down below.

Listing 11: Examples for the built-in 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);
```

In addition to the core primitive types—such as "bool", "str", "num", and others—there are built-in implementations for generic types, including "Array<T>" and "Func<T..., R>". These additionally define their generic parameters which generally default to a standard "any" type as can be seen in listing 12.

Listing 12: Generic built-in types

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

As can be seen in listing 12, generic types are implemented using a special "KipperGenericType" class. This class, shown in listing 13, extends the "KipperType" and includes an additional field for generic arguments. Most importantly, it includes the method "changeGenericTypeArguments", which allows for modifying a type's generic arguments at runtime. It is used in lambda and array definitions, where the built-in generic runtime type is used and then modified to represent the specified generic parameters. When for example an array is initialized, it first gets assigned the default "Array<any>" runtime type, which is then modified by the "changeGenericTypeArguments" method to create the required type, such as "Array<num>". Arrays for example use the specified type for their elements, whilst functions require a return type as well as an array of argument types. The "Func<T..., R>" type on the other hand is used by lambda definitions,

which are user-defined functions with a specific return type and arguments without a name.

Listing 13: Generic Kipper Type

```
class KipperGenericType extends KipperType {
  constructor(name, fields, methods, genericArgs,
     baseType = null) {
    super(name, fields, methods, baseType);
    this.genericArgs = genericArgs;
  }
  isCompatibleWith(obj) {
    return this.name === obj.name;
  }
  changeGenericTypeArguments(genericArgs) {
    return new KipperGenericType(
    this.name,
    this.fields,
    this.methods,
    genericArgs,
    this.baseType
    );
  }
}
```

4.6.5 Runtime Errors

Other built-ins include error classes, which are used in the error handling system to represent runtime errors caused by invalid user operations. The base "KipperError" type has a name property and extends the target language's error type as can be seen in listing 14. Additional error types inherit this base type and extend it with additional error information. For instance, the "KipperNotImplementedError" is used whenever a feature that is not yet implemented is used by the developer.

Listing 14: Kipper error types

```
class KipperError extends Error {
  constructor(msg) {
    super(msg);
    this.name = "KipError";
  }
}
class KipperNotImplementedError extends KipperError {
  constructor(msg) {
    super(msg);
    this.name = "KipNotImplementedError";
  }
}
```

4.6.6 Runtime Generation for Interfaces

Unlike TypeScript, in Kipper all interfaces possess a runtime counterpart, which stores all the required information to verify type compatibility during runtime. This process is managed by the Kipper code generator, which adds custom type instances to the compiled code that represent the structures of the user-defined interfaces with all its methods and properties including their respective types.

Now take for example the given interfaces seen in listing 15.

Listing 15: Example interfaces in the Kipper language

```
1  interface Car {
2    brand: str;
3    honk(volume: num): void;
4    year: num;
5  }
6
7  interface Person {
8    name: str;
9    age: num;
10   car: Car;
11 }
```

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 already provided by the Kipper runtime library is used. If not, we can assume the property's type is a reference to another type structure, which will be simply referenced in our new type structure. This data is stored in an instance of "___kipper.Property", which is finally added to the list of properties in the interface.

In case a method is detected, the generator function fetches the return type and the method's name. If the method has any arguments, the name and type of each argument also gets evaluated and then included in the definition of the "___kipper.Method". After that, it gets added to the interface as well and is stored in its own separate method list.

If we translate the interfaces shown above in listing 15 it would look similar to that in listing 16.

Listing 16: 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  );
```

As shown in listing 16, the properties and methods of an interface are encapsulated within a "KipperType" instance, identified by the "___intf_" prefix. The code for this runtime interface is included directly in the output file, where it can be accessed by any functionality that requires it. To reference the generated interface, the compiler maintains a symbol table that tracks all defined interfaces. The code generator then inserts runtime references to these interfaces wherever necessary.

Notable usages for runtime typechecking include the "matches" operator 4.6.7 and the "typeof" operator 4.6.8.

4.6.7 Matches operator for Interfaces

The primary feature of the Kipper programming language is its runtime type comparison. There are multiple approaches for comparing objects at runtime. One method is comparison by reference, which is implemented using the "instanceof" operator. This method determines that an object is an instance of a class if there is a reference to that class, leveraging JavaScript's prototype system.

Another approach is comparison by structure, where two objects are considered equal if they share the same structure, meaning they have the same properties and methods. Kipper supports both methods of comparison. Reference-based comparison is implemented via the "instanceof" operator and is exclusively used for class comparisons. Structural comparison, referred to as "matching," is applied to primitives and interfaces. Structural comparisons are implemented using the matches operator as can been seen in listing 16.

Listing 17: 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 17, 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 recursively 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 present in the type it should check against. The order of properties does not matter. When the name is 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-primitives.

This property match algorithm is implemented as can seen in listing 18.

Listing 18: 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.

Listing 19: 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 compatible.

4.6.8 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

Glossary

transpilation	Act of compiling high-level language code to high-level code of another language. This term is mostly used in context of JavaScript and its subsidary languages building on top of the language.

Bibliography

[1] JetBrains s.r.o. (2024) The state of developer ecosystem in 2023 infographic. [Online]. Available: https://www.jetbrains.com/lp/devecosystem-2023/javascript/

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

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

List of Figures

List of Tables

List of Source Code Snippets

1 Unchecked variable assignments due to missing type definitions 3
2 Broad ability to perform "invalid" operations despite clear error case . . 3
3 Accessing a missing property returns undefined which later causes an error 4
4 Misaligned function arguments going unnoticed 4
5 Unchecked compile-time casts in TypeScript 6
6 Ambiguous dynamic data in TypeScript 7
7 Example of nominal typing in java 15
8 Example of structural typing in Ocaml 15
9 Example of duck typing in TypeScript 16
10 The structure of a runtime type 17
11 Examples for the built-in runtime types 18
12 Generic built-in types 18
13 Generic Kipper Type 19
14 Kipper error types 19
15 Example interfaces in the Kipper language 20
16 The runtime representation of the previous interfaces 20
17 The Kipper matches operator 22
18 Matches operator property comparison 23
19 Matches operator method comparison 23

Appendix