Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Juuso Mikkonen

Statically typed programming languages in the JavaScript ecosystem:

A type system perspective

Master's Thesis
Espoo, November 27, 2019

Supervisor: Senior University Lecturer Arto Hellas
Advisor: D.Sc. (Tech) Otto Seppälä

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author: Juuso Mikkonen

Title:
Statically typed programming languages in the JavaScript ecosystem: A type system perspective

Date:	November 27, 2019	**Pages:**	v + 73
Major:	Computer Science	**Code:**	SCI3042
Supervisor:	Senior University Lecturer Arto Hellas		
Advisor:	D.Sc. (Tech) Otto Seppälä		

JavaScript is a ubiquitous programming language with usage in web, mobile applications and server software. The status of the language as the de-facto programming language of the web has made the language ecosystem advanced with a great number of userspace libraries and major companies working on efficient runtime systems. The core language, however, has numerous known difficulties caused by the initial design and persisted by the requirements for backwards-compatibility. In the last decade, a number of programming languages have chosen JavaScript as the compile target of the language.

Type theory and its application, programming language type systems, is an essential area of study in the design of programming languages. Every high-level programming language features a type system that greatly influences the ways of designing and implementing programs in the language. This thesis examines a group of selected statically-typed programming languages that compile to JavaScript. The core topics of research in this thesis are the motivation for new JS-compiled languages, the type system design of the languages, and the future direction of the JavaScript ecosystem based on the current trends and parallels to other programming ecosystems.

The results of the work include identifying several trends in type systems for the JS ecosystem and the web. These include unsound yet convenient partially inferred type systems for object-oriented and multi-paradigm programming and fully inferred extended Hindley-Milner type systems for primarily functional programming languages. Additionally, different options for the advancement of the programming ecosystem, including type annotations, inference of dynamically typed languages and new compile targets, are explored. Finally, based on the design choices of the languages researched, we provide several recommendations for safe and productive statically typed programming in the JavaScript ecosystem.

Keywords: programming language, type theory, type system, static typing, JavaScript, TypeScript, ReasonML, Elm, Dart

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tieto-, tietoliikenne- ja informaatiotekniikan maisteriohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä: Juuso Mikkonen

Työn nimi:
Staattisesti tyypitetyt ohjelmointikielet JavaScript-ekosysteemissä: tyyppijärjestelmien näkökulma

Päiväys:	27. marraskuuta 2019	**Sivumäärä:**	v + 73
Pääaine:	Computer Science	**Koodi:**	SCI3042
Valvoja:	Vanhempi yliopistonlehtori Arto Hellas		
Ohjaaja:	TkT Otto Seppälä		

JavaScript on laajalti käytetty ohjelmointikieli, jonka käyttö ulottuu web- ja mobiilisovelluksiin sekä palvelinohjelmistoon. Kielen asema web-kehityksen de-facto-ohjelmointikielenä on luonut sen ympärille laajan ohjelmistoekosysteemin, joka kattaa suuren määrän ohjelmistokirjastoja sekä tehokkaita ajoympäristöjä. Itse kieli aiheuttaa tästä huolimatta vaikeuksia alkuperäisten suunnitteluvirheiden ja vaaditun taaksepäinyhteensopivuuden vuoksi. Viimeisen vuosikymmenen aikana useampi ohjelmointikieli on alkanut käyttää JavaScriptia käännöskohteenaan.

Tyyppiteoria ja sen sovellus, ohjelmointikielten tyyppijärjestelmät, on tärkeä tutkimusala liittyen ohjelmointikielten suunnitteluun. Tyyppijärjestelmä on osa jokaista korkean tason ohjelmoinikieltä ja vaikuttaa täten suuresti itse ohjelmointikielen muihin ominaisuuksiin ja käyttöön. Tämä tutkimus käsittelee joukkoa staattisesti tyypitettyjä ohjelmointikieliä, jotka kääntyvät JavaScript-koodiksi. Tutkimuksen ytimessä ovat uusien kielten kehityksen motiivit, kielten tyyppijärjestelmien suunnittelu ja ominaisuudet sekä JavaScript-ekosysteemin mahdolliset tulevaisuuden suunnat.

Työn tuloksena tunnistamme useita trendejä tyyppijärjestelmien suunnittelussa JavaScript-ekosysteemiin. Näihin kuuluu käytännölliset, mutta teoriassa epäturvalliset tyyppijärjestelmät olio- ja moniparadigmaohjelmoinkieliin sekä funktionaalisten ohjelmointikielien Hindley-Milner-pohjaiset tyyppijärjestelmät, joissa muuttujien tyypit pystytään täysin päättelemään ilman ohjelman kirjoittajan annotaatioita. Lisäksi nostamme esiin useita tulevaisuuden suuntia, jotka voisivat viedä JS-ekosysteemiä eteenpäin. Näihin kuuluvat tyyppiannotaatiot, dynaamisten kielten tyyppi-inferenssi ja uudet käännöskohteet web-ekosysteemiin. Lopuksi annamme tutkimuksen perusteella suosituksia ominaisuuksista ja suunnitteluratkaisuista, jotka voisivat mahdollistaa tehokkaan ja turvallisen ohjelmistokehityksen JavaScript-ekosysteemissä tulevaisuudessa.

Asiasanat:	ohjelmointikieli, tyyppiteoria, tyyppijärjestelmä, staattinen tyypitys, JavaScript, TypeScript, ReasonML, Elm, Dart
Kieli:	Englanti

Contents

Chapter 1

Introduction

JavaScript (JS) is currently the most used programming language in the world. The language, created in 1995 for scripting in websites, has grown into a powerful general-purpose programming language used in web, mobile and desktop applications, server-side programming and scripts. Nowadays JavaScript is not only a language written by developers but also the compile target for several programming languages.

The JavaScript ecosystem is a good starting point for understanding the current state and evolution of programming languages. Being the biggest language in the industry, JavaScript heavily influences the direction of practical programming language development. JavaScript has several unique properties not shared by any other major programming language. Its position as the de-facto programming language of the web additionally separates it from other languages and imposes certain boundaries to the language evolution.

This thesis analyses several programming languages that compile to JavaScript. The languages are not inspected in isolation – instead, the interaction between each language and the JavaScript ecosystem is considered. The role of external ecosystem factors is important in the adoption of a programming language and a sophisticated and expressive programming language by itself is not enough for productive development in practice.

JavaScript is a dynamically typed programming language. This thesis focuses on statically typed programming languages developed to target the JavaScript language. The nature of the type system is an important factor in the programming language. Based on the subset of programming languages that target JavaScript the current trend in new programming languages is the use of robust static type systems to increase developer productivity while adding a minimal burden in the form of type annotations.

While type systems are at the focus of research in the thesis, the level of discussion is the language-level, not type-system level. This is highlighted in

the title which is not simply "Static type systems in the JavaScript ecosystem" but instead "Static type systems in the JavaScript ecosystem: A type system perspective".

1.1 Problem statement

JavaScript is a ubiquitous language with a vast amount of users and a major influence on the software industry. The language is however far from perfect and the options for improving it are limited by ecosystem restrictions. New programming languages in the previously JS-dominated areas, especially the web, can help in the evolution of tooling and engineering practices in software development but require strong interoperability with the existing ecosystem to be considered a viable alternative to JS.

The field of type theory and its practical applications in programming language type systems have made significant advancements since the introduction of JavaScript but the language, being dynamically typed, can not take advantage of these advancements. This places a burden on the maintainability and development of large-scale software in JavaScript. This is understandable as the language was not initially designed for such.

The JavaScript ecosystem continues to develop into different directions through new languages and tools gaining traction. Identifying the problems these languages aim to address and the solutions provided through the design choices in programming paradigm, type system and interoperability features help in shaping the future of the web and JavaScript ecosystem.

1.2 Structure of the thesis

Chapter 2 presents the relevant background for the rest of the thesis. This section covers two wide topics, programming languages and type systems. The former is discussed more briefly; the focus is on programming paradigms and language design aspects with implementation-related topics including compilers and execution omitted. The latter is a more in-depth introduction to the field of type theory and type systems with a focus on practical aspects relevant to the languages researched.

Chapter 3 introduces the methodology of the thesis. It explains the research questions and their motivation and the methods used for answering each of the questions. The chapter additionally gives a rationale for the scope choices made in the work. Chapter 4 discusses the JavaScript programming language through its history, features and ecosystem. This is an

essential premise for discussing languages utilizing JS as their compile target. The end of the chapter is spent exploring several known difficulties in the JavaScript language. These contribute to the motivation for creating new programming languages for the ecosystem.

In chapter 5, several programming languages targeting JavaScript are discussed in depth. The focus is on language features, type system and ecosystem containing interaction with JS as justified in the motivation chapter. The languages are described one at a time with a comparing summary in the end of the chapter. Chapter 6 presents analysis and results synthesized from the earlier discussion. The chapter discusses possible future directions of the JavaScript ecosystem as well as the viability of different options being adopted. Finally, chapter 7 summarizes the conducted research, discusses the validity of obtained results and presents ideas for future research on the subject.

Chapter 2

Background

2.1 Programming languages

The subject of research and discussion in this thesis is the construct of programming language. This section provides the necessary background for discussion. As the focus of the thesis is in the design of programming languages, the specifics of implementation are mostly omitted. This includes subjects like compiler design, interpreters, optimization and computer architecture.

2.1.1 Programming paradigms

Programming languages are typically categorized based on the programming paradigm they represent. Paradigms are formed by the features available in the programming language. The paradigm boundaries are not always definite: a language can incorporate features that make it a candidate for multiple paradigms. A program written in such a multi-paradigm language can utilize the features and patterns of one or multiple paradigms.

Imperative programming is a paradigm that approaches programming through the use of statements to alter the program state. Imperative programming typically involves the use of mutable state and effectful operations. Most imperative languages in active use are also categorized as procedural programming languages. This implies the availability of procedures, subroutines or functions depending on the used terminology. [2]

Object-oriented programming is a paradigm based on modeling program logic through entities that combine data (instance variables, fields) and functionality (methods), objects. The paradigm can be considered an extension to imperative programming with the imperative logic captured by the methods of objects and the mutable data contained in the fields of objects. [2]

Object-oriented programming is typically concerned with the relations of types of objects to each other [2]. In class-based languages, this is modeled with inheritance: a class may inherit another to extend its data fields or logic with additional properties or to overwrite existing object data. Not all object-oriented languages are class-based: prototypal object-oriented languages omit classes for prototypes that are inherent to objects. Extended objects can be instantiated directly from other objects without the class indirection based on the object prototype.

Other properties provided by objects include encapsulation of data and open recursion. Encapsulation is achieved by the use of private data and methods in object definitions – these properties are only accessible by other methods of the object. Different object-oriented languages have different approaches to encapsulation but most implement it to some degree. Open recursion is a typical object-oriented feature where the methods of an object may call other methods through a special variable (often `this` or `self`. Open recursion requires late binding of the special variable to enable calls to other methods defined (or overloaded) later in the inheritance chain. [30]

Declarative programming is often considered the alternative to imperative programming. Declarative programming focuses on *what* is the desired result of the program, omitting the specification of *how* the program should achieve this. The term is significantly broad and is used to describe a number of different sub-paradigms.

Functional programming is a more concrete subset of declarative programming. In functional programming, the program is modeled through a series of function applications [30]. Several practical general-purpose programming languages have functional features with some relying entirely on the functional paradigm. To be considered a functional language, several properties are required of the language. Most importantly, the functions in the language must be *first-class*, i.e., they can be treated like any other value type. This implies that a function

1. can be assigned to a variable
2. can be passed as a parameter to a function
3. can be the return value of a function.

A language with first-class functions can be considered functional. This includes programming languages such as ML, Lisp, JavaScript and Scala.

The functional programming paradigm is additionally separated into pure functional programming and impure functional programming. Pure functional languages don't support *mutability* of values or *side-effects*, i.e., imperative operations. Such languages operate entirely on the functional semantics

and are relatively rare in practical programming with more support in research languages. Haskell is the best known purely functional language. Impure languages include a variety of functional languages with imperative constructs available as an escape-hatch (OCaml, Clojure) and multi-paradigm languages (JavaScript, Scala). Most modern programming languages provide some facilities for functional programming but the amount of usage varies. E.g., Python, in theory, enables the use of functional programming but in practice, the language design encourages developers to mostly resort in other methods of computation.

2.2 Type systems

A type system is part of the foundations of any high-level programming language[1]. Thus a working knowledge of type systems is an essential prerequisite for discussing programming language design. Pierce [30] defines type systems in the context of programming languages as follows

> A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.

Type systems (or type theory) are a broader subject than the definition used for understanding type systems in programming languages. Type theory has a deep connection to mathematics and logic and is typically concerned with levels of abstraction not present in conventional programming languages [30]. In the context of computer science, a significant amount of research and theory in type systems focuses on pure type systems and lambda-calculi. While the concepts are partially shared with more practical development of type systems, practical languages typically sacrifice properties of purity for the expressiveness of the language. For instance, defining recursive functions sacrifices the property of always-terminating programs but is highly useful in translating ideas into implementation. While this thesis focuses on the practical aspects of type theory, it is important to acknowledge the deeper foundations of type theory that underlie the practical aspects of types in programming languages.

[1] Different definitions for high and low-level programming languages are used in different contexts. Here languages that operate on abstractions over actual computer hardware are considered high level. Examples of such languages are C, JavaScript and Python. Conversely, assembly languages are considered low-level languages.

2.2.1 Static and dynamic type systems

Type systems are often categorized as static or dynamic type systems. In static type systems, the type of each term in a program is known (through annotations or inference) at compile-time with programs that can not be typed being rejected by the typechecker: a static type system is essentially a set of constraints applied to the programming model. The type system can prove that a program has no type errors but can not generally prove the existence of type errors – thus some programs that can be proven to be valid by the developer are rejected by the typechecker. The type system is a trade-off between the expressiveness and safety of a programming language. More sophisticated type systems can allow a greater degree of safety without limiting expressiveness. [30]

Dynamically typed (or dynamically checked) programming languages defer type checking to runtime. This removes the need for type annotations in the source code and allows for greater expressiveness with the lack of a conservative typechecker. It adds overhead to the execution of a program in the form of type checking at runtime and opens the possibility of runtime type errors not present in safe static languages. Dynamically typed languages are sometimes referred to as untyped languages which is a misnomer as types do exist in the language during execution. Dynamic type systems are less academically researched than static type systems as types manifest only during runtime and thus can not be used to statically prove properties of the program.

```
if <complex expression that always evaluates to true>
    then true
    else -1
```

Listing 1: A pseudocode expression demonstrating the conservative nature of static typechecking.

Listing 1 demonstrates the tradeoff in expressiveness and safety. If the language used is statically typed, its typechecker must reject the program as a source of type error while the runtime behavior would be correct. If the language is dynamically typed the expression is valid and behaves well in runtime. The example could alternatively be used to demonstrates the benefits of sophisticated type systems for static typing: a type system with subtyping (discussed in section 2.2) may be able to deduce that the type of the complex expression is in fact a *subtype* of `boolean`, `true`. The compiler can then warn the programmer of an unreachable else branch in the source

code. The programmer can then simplify the expression to the value of the `then` branch, `true`.

2.2.2 The advantages of static types

There is a good amount of empirical evidence on static type systems having several benefits over dynamic ones in software development. Kleinschmager et. al. [22] demonstrate that there's an increase in the maintainability of software when using statically typed languages for development. A large scale study of open-source software by Ray et. al. [32] shows an improvement in code quality for strongly typed languages[2].

A major part of empirical research on the benefits and costs of statically typed languages versus dynamic languages is based on imperative or object-oriented languages – often Java [22, 29]. This may downplay the true benefits of type systems. Subsection 2.2.5 introduces the Hindley-Milner type system used in several functional programming languages. It provides not only a sound and expressive type system but also full type reconstruction (type inference) which greatly reduces the amount of type annotation needed from the developer.

Some of the benefits of static type systems are emergent from the proofs derived for such type systems: under a sound type system, it is impossible to successfully compile software with type errors. Therefore a program compiling in a system is known to fulfill the requirements set by the compiler. Empirical evidence has shown that static typechecking can help a program avoid complex conceptual errors in addition to simple type errors as such errors often manifest as type errors in an expressive type system. [30]

In dynamically checked programs type errors manifest at runtime. Detecting and debugging runtime type errors is generally harder than fixing the errors at compile-time since a runtime error may manifest only in an obscure edge case that is not obvious when testing the program. Depending on the type error semantics of the language the error may additionally manifest far from the source of the bug. Powerful static checking helps not only in detecting the errors but also in locating and proposing fixes for them. [30]

In statically typed programming languages with a sound type system, the absence of compiler errors proves the absence of type errors [30]. This gives the programmer relatively strong guarantees on the behavior of the program. The actual guarantees depend on the classification of type errors in the language and the expressiveness of the type system. The language

[2]The study, however, acknowledges that there are possible biases in the studied data that can not be eliminated threatening the validity of the result.

safety property of type systems, including the definition of soundness, is discussed in subsection 2.2.3.

When the first high-level programming languages were developed, a major driver for static type systems was the performance of compiled programs. When the type of each variable is known at compile-time it is easier to reserve a correct amount of (either heap or stack) memory for the variable. Additionally, knowing the type of values passed to operators or functions in advance means that there is less (or no) need for checking the type at runtime. Distinguishing between different numeric types allowed Fortran, an early numerical computing language, to select the most performant algorithms to use at compile-time. [30]

Likewise, many of the optimizations performed by modern compilers are based on using the available type information to make valid assumptions on the program semantics and simplifying the generated code based on these[3]. This can be exploited to reduce the dynamic behavior needed by providing static and inlined implementations of dynamic methods by analyzing their use based on the type hierarchy [14]. Dynamically typed languages can similarly benefit from type reconstruction for optimization [1].

Type information of a program serves additionally as documentation of interfaces and functions of a program or library. Unlike other forms of documentation, including comments and external documents, types are always up-to-date and correct. The more expressive the type system, the better it serves as documentation. Documentation through types does not require explicit type annotations. Languages with strong type inference features can provide type-based interface documentation without annotating. The ML family of programming languages (or more generally Hindley-Milner type systems) provide sound typechecking with zero or minimal type annotations.

```
map :: (a -> b) -> [a] -> [b]

filter :: (a -> Bool) -> [a] -> [a]
```

Listing 2: The function signatures of list functions `map` and `reduce` in Haskell.

A programmer can deduce several properties related to the use of the functions from the signatures of Haskell functions shown in listing 2. The types `a` and `b` that appear as list elements are abstract and can be replaced by any type. Thus the functions certainly do not depend on the structure of

[3]A possible source of confusion is that immutability in functional programming is another important source of optimization. In statically typed functional languages the entirety of optimization capabilities should not, therefore, be attributed to the type system.

the members of the list. The arguments of the map function are a mapping function from type `a` to `b` and a list `a`. The return value is a list of type `b` which means that the elements are transformed from `a` to `b`. Filter operates on only one abstract type `a`. In addition to the list of `a` it requires a predicate function, i.e., a function that maps values of `a` to a truth value. With this input, it produces a new list of `a`. This information helps a programmer reason about the use of the functions even if they are unfamiliar with the functions' contents.

Readable type information potentially enables innovative documentation solutions: the Haskell programming language that features an advanced type system provides a search engine for looking up library functions based on type signature[4]. Modern integrated development environments (IDEs) can provide contextual help for users by leveraging the statically available type information. This has been empirically shown to improve the productivity of developers adopting new APIs when compared to dynamic languages using similar IDE tools [29].

2.2.3 Soundness and safety

The terms soundness and safety (and opposites) are often used to describe programming languages and type systems. Safety (or unsafety) is a context-dependent property of a programming language that depends on the abstraction level of operation. Pierce [30] defines a safe language as one that "protects its own abstractions". A programmer should be able to rely on the interfaces of the language hiding details of the underlying system in such a way that the developer need not be aware of the internals. Soundness is a defining property of static type systems.

The safety of a programming language does not require static typing: the property is also achievable by the means of runtime checks. In fact, guaranteeing safety by only static checking is not possible in practice but requires runtime checking to complement it. E.g. bounds-checking of arrays can not be (practically) achieved as a compile-time operation. Static checking can however greatly reduce the amount of runtime checking needed for safety. A typical example of an unsafe language by design is C in which pointer arithmetic without bound-checking can result in undefined behavior that affects the entire program. Such errors that the runtime can not recover from gracefully are also referred to as untrapped errors [30]. Out of bounds access of an array yields an untrapped error in C whereas in languages such as Python the error is trapped by the interpreter and the program execution

[4]https://hoogle.haskell.org

can continue safely. Safety by static analysis requires the compiler to be conservative and reject programs that it can not prove to be correct. Safety and unsafety can be combined in a coordinated fashion: Rust, a system programming language by Mozilla, is safe by design but provides semantics for performing unsafe operations using the `unsafe` keyword as an escape hatch [21].

Soundness is a property of a statically typed system. The condition of soundness is that a well-typed program in a language should never produce type errors. Again, the notion of type error is dependent on the language. However, it can be generalized to include any situation where a function is applied to an argument of invalid type or a non-function is applied [38]. An unsound type system can not provide the same guarantees for the correctness of a program that a sound type system can. There are however situations where an unsound type system can prove to be useful. This is explored further in chapter 5 through practical examples.

The research of type theory approaches the properties of safety and soundness through proofs: a type system should be formalized so that certain properties can be proven. Most practical programming languages, even the ones claiming soundness, have no formal proofs of their type system properties. The languages tend to be significantly more complex than languages researched in type theory rendering the formalization difficult. At times, the situation has resulted in soundness issues being detected after long periods of usage of a language. In 2016 Amin and Tate [3] discovered a previously unknown source of unsoundness in the Java and Scala type systems. The unsoundness was a result of multiple features, sound in isolation, interacting in an unexpected way. The bug had existed in the type systems for the previous 12 years with a massive amount of software relying on the type systems.

2.2.4 Static type system characteristics

Discussing type systems requires understanding several key properties and design choices made in constructing the system. This subsection very briefly introduces some core concepts relevant to the discussion in this thesis. These features revolve around type substitution, a fundamental language feature present in most static type systems of today.

Subtyping (also subtype polymorphism) is a type substitution feature available in several advanced programming language type systems. In subtyping, individual types form hierarchical relations based on the ability to substitute one type with the other. Subtyping is denoted $T <: S$ where type T is a subtype of type S. Other properties of the type system define the exact rules by which a type can be considered a subtype of another. The

evaluation rules similarly specify where substitution applies. [30]

The concrete implementation of subtyping varies among practical programming languages. A frequent categorization is the one of *nominal* (by-name) and *structural* subtyping. As the terms suggest, nominal subtyping is based on type names while structural subtyping relies on the structure of types. Nominal subtyping is generally preferred in object-oriented languages while functional languages and research languages tend to favor structural subtyping. Equation 2.1 presents one of the most common subtyping rules, width subtyping[5] [30]

$$\{x : T, y : U\} <: \{x : T\} \tag{2.1}$$

While it may seem initially counterintuitive that the more specific type is the subtype, this follows from the fact that the less specific *supertype* can be replaced with the subtype with all the functionality of the former still available – but not the other way around. In the demonstrated width subtyping a type that contains all the key-type pairs of another type – and additional key-value pairs – is a subtype of this type. Other general rules include depth subtyping and permutation subtyping presented below. [30]

$$\frac{T <: S}{\{x : T, y : U\} <: \{x : S\}}$$

Depth subtyping implies that types included in type T can also follow subtyping rules. The horizontal line separates the premises (above) and implications (below) of the rule. One subtlety in the relationship is that not all depth subtyping is covariant as in the record field depth subtyping. Generally, when applying subtyping where a type appears as an input instead of output, the subtyping relation is contravariant.

$$\frac{S_1 <: T_1 \qquad T_2 <: S_2}{T_1 \to T_2 <: S_1 \to S_2}$$

The permutation subtyping rule implies that for some types, the order of its components does not affect the essence of the type, e.g. the type of an object with field-type pairs can be replaced with a type having the same pairs in a different order. While the rule, applied to records, appears obvious it

[5]The subtyping rules presented here are not in their most generic forms for improved readability. E.g., in the width subtyping rule, both $x : T$ and $y : U$ should be replaced with sets of key-value pairs $l_i : T_i^{i \in 1..n}$ and $l_i : T_i^{i \in (n+1)..m}$ to obtain the general form.

introduces some difficulty in performant implementation of subtyping: varying order of fields implies the need for dynamic lookup of each field which can be costly for the type checking algorithm and execution on runtime. [30]

$$\frac{X : \{x : T, y : U\} \qquad Y : \{y : U, x : T\}}{X <: Y \qquad Y <: X}$$

To combine width, depth and permutation subtyping, a *transitivity* rule is required

$$\frac{X <: Y \qquad Y <: Z}{X <: Z}$$

Another option is to combine the three rules into one which avoids the need for the transitivity rule. While the resulting rule is less readable, it is more practical when implementing typecheckers. The transitivity rule alone is difficult in practice as it provides no context for when it should be applied and to which typing relations. [30]

Nominal subtyping requires the subtype relation to be explicitly stated. In Java, for example, subtyping is achieved by specifying the inheritance hierarchy in class definitions. All classes implicitly inherit the base `Object` class and are usable through subtyping when `Object` is expected.

```
class X { ... }

class Y extends X { ... }
```

Conversely, another class Z, which contains all the methods and instance variables with the same type signature as X is not a subtype of X unless the class is explicitly specified to inherit X or one of its subtypes, e.g. Y. Note that in Java the class names and types of instantiated objects are identical.

Nominal and structural types provide different frameworks for working with subtyping, both of which have their advantages. The intrinsic nature of structural subtyping makes the property emergent – it provides opportunities for versatile reuse without the initial intent of programmer. The advantages of nominal subtyping include enforcement of developer intention and better type error messages related to subtyping [24]. From type theory perspective structural typing is more convenient since for reasoning about type relations it suffices to know the structures in question without the need for an auxiliary data structure specifying the full inheritance hierarchy [30].

Both nominal and structural type systems are used in modern practical programming languages. Structural types are used by functional and multi-paradigm languages including OCaml, Haskell, Go and TypeScript. Nominal types can be seen in more object-oriented (or multi-paradigm) languages including Java, Rust and C++. While the separation is generally clear, there are nominal typing features in structural type systems and structural features in nominal type systems. [20]

Subtyping is a complex subject from a practical point of view. When used with records and advanced language features like parametric polymorphism, the implementation becomes complex [30]. Thus many programming languages with structural type equivalence do not implement subtyping. Another approach to the flexible use of records taken by programming languages is *row polymorphism* (also row-variable polymorphism) that defines an alternative logic for type substitution [37]. Row polymorphism enables preserving the specificity of a type when applied to a function expecting a more general type whereas subtyping always loses this information. Row polymorphism is additionally considered easier to combine with type inference which is why it is preferred in languages including PureScript and OCaml [28].

2.2.5 Hindley-Milner

The Hindley-Milner (HM) type system (also referred to as Damas–Milner and Damas–Hindley–Milner) is a widely used foundation for the type systems of several programming languages. The system was individually discovered by Hindley [18] and Milner [27] and later formalized and proven by Damas [11]. The type system provides extensive type inference requiring zero or minimal[6] type annotations from an author of a program. Hindley-Milner was first implemented in the ML ("Meta Language") programming language and has since become a signature feature of programming languages in the ML family, including Standard ML, OCaml and Haskell [30].

Hindley-Milner involves discovering a set of constraints for the types of each (non-annotated) construct in the program and using a unification procedure to check that the set of constraints implies a nonempty set of solutions. A central concept in the system is that of the *principal type*. If the set of solutions for a program is nonempty, there must be some solution σ that is more or equally general than any other solution. This is denoted $\sigma \sqsubseteq \sigma'$. The inference algorithm selects the most general solution, i.e. the principal type for each type. The unification algorithm originally proposed by Damas

[6] E.g., an unrestricted combination of polymorphism and general recursion in a HM type system requires type annotation.

and Milner is known as Algorithm W. [30]

The original ML language introduced a method of polymorphism that has since become strongly associated with the ML family and Hindley-Milner languages known as let-polymorphism. Consider the following OCaml expression:

```
let id = fun x -> x
  in id "this is " ^ string_of_bool (id true)
```

If the identity function `id` was assigned a single principal type based on the first usage, the signature would be `id :: string -> string`. The latter use of the function would reduce the answer set of type reconstruction to the empty set, signifying a type error. Instead, let-polymorphism associates a different type variable for each use of the function. These are then independently reconstructed to the appropriate types `id :: string -> string` and `id :: bool -> bool`. In practice, let-polymorphism must address some additional concerns: the trivial algorithm can result in an amount of work exponential to the term size with nested let expressions. Additionally, there are soundness concerns related to side-effectful functions in let expressions. The former of the issues can be addressed with optimized algorithms whereas the latter is typically mitigated by restricting the application of let-polymorphism to terms of a certain shape (*value restriction*) in the right-hand side of the let expression. [30]

With its roots in lambda calculus, Hindley-Milner at its essence is a type system for functional programming languages [30]. Procedural or object-oriented languages with a strong reliance on mutable data can not exploit the inference properties offered by Hindley-Milner. Some ML family languages include procedural "escape hatches" from functional programming for optimization and expressiveness, e.g., the mutable reference types and looping constructs in OCaml [28]. These features require an extension to the HM type system for sound type checking. Many of the conventional ML languages offer other extensions to the Hindley-Milner system, e.g., Haskell which supports higher-order type abstraction through type classes.

Chapter 3

Methodology

3.1 Research questions

Our analysis of statically typed languages with JavaScript as the compile target focuses on the following research questions

1. What is the motivation for creating new statically-typed languages to the JavaScript ecosystem?
2. What approaches in programming language design – and in particular, type system design – have been proposed to enhance development targeting JavaScript?
3. If the future of development in the JavaScript ecosystem is statically typed, what will the type system look like?

The first research question searches for the premise of the current programming language evolution in the JavaScript ecosystem. The question is relevant and interesting for several reasons: JavaScript is the most popular programming language of today with a great number of developers and a mature ecosystem – why is there a significant drive for replacing programming languages happening? JavaScript has been popular for several years and has made major advancements in the last years – why is the transition to statically typed languages happening only now?

The second research question is evaluative by nature: it discusses the features and design decisions of programming languages that aim to improve development in the JS ecosystem. The focus of the research question is in the type system perspective. While several important language design aspects can be identified, the viewpoint of type systems is chosen for this thesis. Type theory and its practical applications in programming languages are a

rich field of research that bears an impact on the everyday work of software engineers and to the end results of this work, the software that powers the modern world.

The third research question is arguably the most difficult one of the three to answer. We attempt to form an answer to the question based on the language and type system design observed in the programming languages further studied in the languages introduced in chapter 5. Available information and research on similar development in other programming ecosystems is additionally used.

3.2 Approach

Programming languages are a difficult area of research for several reasons. With the rapid and largely industry-driven advancement of the field, it is challenging for academic research to keep up. Therefore many innovations and new approaches are documented informally if at all. Additionally, there are a number of difficulties in researching the practical usage of programming languages. For instance, metrics on the popularity of programming languages vary and are consistent only for established programming languages with a significant amount of usage. For a deeper understanding of the usage of programming languages, there are fortunately good corpuses available for programming language usage in the wild. However, comparative study is notably difficult due to inherent biases in technology choices as acknowledged in existing research of the field [32]. In this thesis, these approaches are ruled out by the relatively small amount of usage of some of the selected programming languages. Obtaining a representative data set of language usage would be difficult and prone to biases. Existing large-scale analysis of source code is used as a source when applicable, however.

To understand what drives the creation of new programming languages that target JavaScript, i.e., RQ1, we begin by examining JavaScript. Due to the language's immense popularity, different aspects of the language are well researched and documented. The supporting standard, ECMA-262 [15], provides a detailed and rigorous specification of the current state of the language while the previous standard versions display the development of the language. Books intended for JavaScript developers help in understanding the practical implications of different design choices. Chapter 4 is dedicated to the discussion of the JS programming language along with its core features, ecosystem and known difficulties.

There is a good amount of research available on different aspects of the JavaScript language due to its popularity. Researched properties of the lan-

guage relevant to this thesis include common causes of bugs [16], type coercion rules [31] and type inference [5, 17].

The information is collected primarily from the aforementioned sources. The focus on the collected information is on the core properties of the language and its ecosystem and on the issues commonly experienced by developers. Furthermore, the programming languages discussed later on in the thesis provide some added context for answering the question. The sources used for examining the statically typed languages are discussed more thoroughly when discussing the approach for RQ2. After the chapter, a reader should have acquired a background for understanding the context in which the programming languages discussed later have developed.

A significant research subject of the thesis is the programming languages chosen for inspection (RQ2). An acknowledged difficulty in evaluating programming languages and type systems is the varying level of documentation for the languages. The general documentation of programming languages tends to focus on the practical features and building blocks of programs rather than the design principles of the language. Fortunately, most programming languages of today – including each of the chosen subjects – are open-source. Therefore the source code is available for analysis. The source code is, however – especially if the level of comments is low – a difficult medium for deducing design principles of the language or the typechecker. Therefore direct analysis of source code is utilized only when other approaches fail.

Another method for assessing the capabilities of a programming language or a typechecker is writing and executing programs in the language. The power of the method in drawing conclusions is limited: a single program is rarely sufficient for proving a property of a system. In some cases, it does suffice: one property that is considered for type systems in the thesis is soundness. While any single program is insufficient in proving the soundness of a type system, a single counter-example does prove unsoundness. Programming and execution was used in this thesis to verify the state of soundness in several of the discussed languages as changes in the languages were not in all cases directly reflected in the current documentation. Additionally, the compiler output of written programs is used as a source of information on the languages. The compiled program can be exploited to obtain information on e.g., optimization techniques of the compiler (as seen in section 5.3). The reasoning on type system features when discussing the statically typed programming languages is based on the fundamental research in the field of type theory [18, 27, 30].

The approach for RQ3 is a synthesis of the results obtained for the first two research questions. As support for answering the question, parallels to other programming ecosystems are utilized: observing the language evolution

of the JVM ecosystem that has been active for a longer time provides perspective to the activity in the JS ecosystem. Java and the JVM ecosystem are well-researched subjects [3, 23, 29]. The well researched parallel ecosystem helps in generalizing the results and partially verifies that the development of the researched programming ecosystem is not simply an isolated anomaly but follows a logical path.

Another point of discussion for the third research question are solutions beyond static typing for improved language usage and static analysis. These include 3rd party static analysis tools for JavaScript and language development that includes type annotations for static typechecking to some degree. The discussion is based on the available tooling and research that helps establish the viability of different approaches for better static analysis.

As there is no definitive answer to what the future holds for the JavaScript ecosystem, we finally present some suggestions and recommendations for the future development of the ecosystem. These are based on the successful design choices observed in the programming languages researched for the previous research questions and the properties of JavaScript surveyed for background and RQ1.

3.3 Scope

The scope definition of the thesis consists of two limiting factors. These are the *properties* of the programming languages studied and the *languages* selected for inspection. The primary motivation for selecting the subjects of research is the relevance to the research questions. This section further defines the criteria used in the scoping decisions. The impact of the decisions on the validity of the results of the research is additionally discussed in section 7.1.

3.3.1 Programming languages

The programming languages further researched in chapter 5 are TypeScript, Elm, ReasonML and Dart. TypeScript is a superset of JavaScript developed by Microsoft for supporting large codebases in JavaScript development. Elm is a functional UI programming language developed by Evan Czaplicki with an ambitious goal of producing programs without any runtime errors. ReasonML is a programming toolchain for the web powered by OCaml, a robust functional language developed by INRIA. Dart is a programming language by Google with support for web, server-side and cross-platform mobile development. For each of the languages, the latest stable release was used as a

reference. The exact versions appear as footnotes in the introduction to each language.

The programming languages selected for inspection were chosen based on the following criteria:

1. is statically typed
2. compiles to JavaScript
3. has seen actual usage in production
4. has some unique aspect distinct from the other languages selected
5. is actively developed
6. is documented well enough to be evaluated.

The first criterion is based on the selection of type systems as an inspected property. Static type systems are generally more sophisticated and better researched than dynamically typed systems as explored in section 2.2. Additionally, statically typed programming languages are currently a relevant research subject in practical programming language design. Even without the scope selection most of the potential languages that satisfy the other criteria would be statically typed.

The second criterion is directly emergent from the inspection of the JavaScript ecosystem: it is essential for the language to be a part of the ecosystem to apply to the research question. There are languages that interact with JavaScript in other ways than by targeting the language as a compile target but such interaction is beyond the scope of the thesis.

Criteria three to five consider the relevancy of researching the languages. While languages that contradict the criteria, i.e., languages that are proof-of-concept or research-focused, languages that share core characteristics with each other and languages that are legacy, may be relevant to some research questions, research focused on the current evolution of a practical programming environment finds less benefit in such.

While condition 6 requires proper documentation for the language to be available, it does not specify that the language must be subject to academic research. Many of the relevant languages are novel and industry-driven with only informal definitions and documentation. In such cases, the available documentation and source code (if available) is used as the primary source for information.

Potential languages left outside the research include Flow, Scala.js and ClojureScript. Flow was eliminated due to the high level of similarities with

Language	Released	Created by	Paradigm	Types
TypeScript	2012	Microsoft	OOP, functional	Static
Elm	2012	Evan Czaplicki	Functional	Static
ReasonML	2016	Jordan Walke	Functional	Static
Dart	2011	Google	OOP, functional	Static
PureScript	2013	Phil Freeman	Functional	Static
Scala.js	2013	Martin Odersky	OOP, functional	Static
ClojureScript	2011	Rich Hickey	Functional	Dynamic

Table 3.1: Programming languages initially considered for research.

TypeScript: both provide a type-annotated superset of JavaScript with very similar syntax. Additionally, it is questionable if Flow should be considered a separate programming language from JavaScript. The description on the official documentation of Flow[1] refers to the project as "a static type checker for ... JavaScript code". There is research on the type systems of TypeScript and Flow that displays similar levels of bug detection by each using relatively similar annotations [16].

ClojureScript is an interesting approach to functional web programming. It leverages the Lisp syntax of Clojure, a dynamic programming language targeting the Java Virtual Machine (JVM). The dynamic type system of ClojureScript, however, rules it out of the scope of this thesis. Scala.js is a compiler for the Scala programming language targeting JavaScript. Scala is a multi-paradigm language with a highly sophisticated type system targeting JVM. While Scala.js is an interesting approach to supporting the web as a compile target for languages, the ecosystem of the web compile target is rather limited considering the age of the project (version 0.1 released in 2013). Research of the language would thus easily become research of the Scala programming language which is not intentional for this thesis.

3.3.2 Language properties

Another scoping decision in addition to selecting which programming languages to inspect is to choose the properties in which to focus on. It is essential to choose properties that support answering the research questions defined earlier. Not all aspects of programming languages can be covered extensively – focus is required.

As specified in the title of the thesis, type systems are one of the core features researched. The further scope limitation of statically typed pro-

[1] https://flow.org

gramming languages directs this to the static behavior of type theory. The type-theoretic background of the work is grounded in the theory in chapter 2. The type systems of each language are not only inspected in the context of the language itself but also in its execution environment, the JavaScript engine. Some type systems have gone to great lengths to enable the type-safe use of the programming styles typical to JavaScript.

The programming paradigm is another language property discussed. The paradigm used and the features of the type system interact with each other deeply which supports discussing paradigms in addition to type systems. There is a clear trend towards functional programming both in the web and in general but the selection of languages in the thesis includes both functional and object-oriented languages. The relation to the programming model of JavaScript (defined in chapter 4) is also explored.

To answer research question 3 we must look beyond the core semantics of the programming language. The entire *ecosystem* around the language has an influence on the adoption of the language and the viability of the language in practice. The ecosystem includes the package infrastructure, development tools and resources provided by the language but also its bindings to the JavaScript ecosystem. In some cases, the latter may prove to be more influential in the adoption of the language than the former.

Furthermore, none of the properties highlighted here are discussed in isolation due to their influence in each other: the programming paradigm of a language partially shapes the type system, type system limits the degree of interoperability with JavaScript and the ecosystem of a programming language is emergent from the inherent properties of the language. By understanding the language design holistically, we can make observations on the bigger picture of programming language design and evolution and on the future of programming.

Chapter 4

The JavaScript programming language

JavaScript is a dynamically typed interpreted programming language created in 1995 as a scripting language for web content in the NetScape browser. The language was initially developed by Brendan Eich. Since then, JavaScript has become a major programming language used not only in web applications but also server-side and in mobile development. JavaScript is currently the most used programming language in the world [34].

The popularity of JavaScript is explained by its place as the de-facto scripting language of the web: all major web browsers include a JavaScript interpreter. In the last 25 years, web pages have evolved from static documents into complex applications requiring and powered by large amounts of client-side logic.

4.1 Overview

JavaScript has several unique properties among general-purpose programming languages. It was strongly influenced by Java which was a popular language in the time JS was created but also by languages such as Smalltalk and the LISP family of programming languages [33]. This section introduces several of its major features and properties and discusses the evolution of the language. This information is provided as a background for research on the languages targeting JavaScript.

4.1.1 Standardization and language evolution

JavaScript was standardized in 1997 by Ecma in the ECMAScript specification. The standard specifies the syntax, features and semantics of JavasScript supporting the coexistence of multiple implementations of the language.

In 2011 the standard reached its fifth version. Since ECMAScript 6 (ES6, ES2015) in 2015, there has been a new standard version released each year. The ES2015 standard modernized JavaScript by introducing a number of new core features including `let` and `const` variable declarations, lambdas with arrow syntax, default parameters for functions and the class syntax. The more recent standard versions have introduced fewer additions each. The JavaScript specification is developed by Ecma's TC39 group which consists of software developers, academics and implementers of JS engines. The specification is developed with proposals for new features and changes which go through several stages of evaluation and discussion before possibly becoming parts of a new specification version.

While the development of JavaScript has been rapid in the last years with a new standard version each year, the implementations of JavaScript limit the usage of the new features for developers. Adopting the latest standard features before implementations are available in the popular browser implementation results in errors for unsupported browsers.

Due to the usage in web browsers, JavaScript has to maintain backwards-compatibility between all versions to ensure that older websites continue to function as before. This limits the degree of changes possible and has resulted in an append-only style of modifications to the language. Legacy features continue to exist along with newer counterparts while their use is often discouraged by developers.

4.1.2 Execution environments

Thanks to the standardization of the JavaScript specification there are multiple implementations of the language runtime available. The most prominent engines are V8 by Google, SpiderMonkey by Mozilla, JavascriptCore by Apple and Chakra by Microsoft. Each of these is used primarily in web browsers with V8 additionally serving as the JavaScript engine for Node.js, the popular runtime system for server-side JavaScript.

The standardization enables correct JavaScript programs to be executed on any of the engines. The features from the latest standard versions are however not implemented in the released versions of most engines. Due to

[1]Recent versions of Microsoft Edge have adopted V8 as JS engine. Internet Explorer continues to use Chakra

Engine	Author	License	Used in
V8	Google	BSD	Google Chrome, Node.js
SpiderMonkey	Mozilla	MPL 2.0	Mozilla Firefox
JavascriptCore	Apple	LGPLv2.1	Safari
Chakra	Microsoft	MIT	Edge, Internet Explorer[1]

Table 4.1: Current JavaScript engines

both the competition between browsers and the large interest in JavaScript by developers the JS engines of today are highly optimized and perform better than many other interpreted languages. V8, for instance, leverages just-in-time (JIT) compilation instead of interpreting for increased runtime performance. JavaScript engines are generally implemented in C or C++.

4.1.3 Language features

JavaScript is often described as a multi-paradigm programming language. It offers great freedom to developers in the choice of programming style through its large amount of features and a dynamic type system.

Type system

The dynamic nature of the type system of JavaScript means that variables do not have an immutable type in the language. Each value, however, has a distinct type that conveys what data the value stores. The JS type system has seven types of values. These types are introduced in table 4.2.

Type	Example value	typeof result	Pass by
number	3.141	number	value
string	“example”	string	value
boolean	false	boolean	value
undefined	undefined	undefined	value
null	null	object[2]	value
object	{ value: 1 }	object[3]	reference
symbol	Symbol(sym)	symbol	value

Table 4.2: JavaScript types

[2]The typeof result of null is known to be incorrect. It should intuitively be “null” instead of “object”. The error is maintained in the standard for backwards-compatibility

[3]There is an exception to this: functions are also members of the object type in JS but the typeof operator returns “function” for them. Similar to `typeof null`, this is an error

It is notable that JavaScript only has a single number type that is used to represent both integer and floating-point values [15]. The number type is a double-precision floating-point value as defined in IEEE 754-2008 standard [19]. In addition to the expected numeric values, it can also represent infinity, negative infinity and a special not-a-number (NaN) value.

The object type is the only non-primitive type in the system and is thus the most complex of the types. The array type of JavaScript is also an object with some special properties. Objects are essentially key-value containers that include some additional properties associated with the keys. In more recent ECMAScript versions they also support accessor type members with get and set methods. [15]

Functions

Functions in JavaScript are first-class members of the language. This means that function can be assigned to variables, passed as parameters to other functions and returned from functions. This property enables the functional programming paradigm and is also required for the asynchronous non-blocking I/O style of JavaScript. JavaScript does not enforce types or the number of parameters passed to a function.

Internally JavaScript functions are objects created using the `Function` constructor. There are several ways to define functions: the function declaration, the function expression, the arrow function expression and the function constructor. Additionally, there are also counterparts for each of these (excluding the arrow expression) for generator functions. [15]

Object-oriented features

One of the original design goals of JavaScript was to resemble Java while being more lightweight and simple [33]. Instead of the class-based object model that powers Java, JavaScript opted for a prototypal object model. This is a rather unique design choice among mainstream programming languages: well-known object-oriented languages such as Java, C++, Python, Ruby and PHP are all class-based.

The prototype-based object model is also known as instance-based. In class-based programming, new objects are instantiated based on an abstract definition of the object's behaviour – i.e., a class. In prototypal models, each object carries a description of this behavior – the prototype. New objects are instantiated from existing objects based on their prototype.

---

but has proven to be handy in practice in addition to being required for compatibility.

The ES2015 standard introduced the class syntax to JavaScript. This essentially allows developers to define objects in terms of class-based models. Internally the instantiation is prototypal regardless of the syntax used. The class representation enables representing traditional inheritance patterns of the object-oriented paradigm.

Concurrency

JavaScript is single-threaded by nature. There are no primitives or support for parallelism in the ECMAScript specification by design. Instead, JS concurrency is based on asynchronous, non-blocking operations. Functions that perform an asynchronous operation return immediately. Such functions typically accept a callback function parameter that is called once the operation finishes. [7]

```
performAsyncOperation(args, function (result, error) {
    if (error) {
        console.error("Operation resulted in error", error)
    } else {
        console.log("Operation completed with result", result)
    }
})
```

A more recent development in JS has led to the use of promise-based APIs over callbacks. A promise is an object that represents the result of a deferred operation. It can be in three distinct states: fulfilled, rejected or pending. Promises were initially introduced to JavaScript by 3rd party libraries such as jQuery (deferred objects) and Bluebird. They were standardized in ECMAScript 6 in 2015 and are implemented in all major JS engines nowadays.

```
performAsyncOperation(args)
    .then(function (result) {
        console.log("Operation completed with result", result)
    })
    .catch(function (error) {
        console.error("Operation resulted in error", error)
    })
```

ECMAScript 2017 introduced another abstraction on top of Promises for simplified control flow in asynchronous operations. The new async function type enables writing asynchronous code in a way that resembles synchronous

blocking code and makes it easier to follow the flow of control in execution. The new syntax addition consists of the async modifier for functions and the await keyword that enables the code to simulate waiting for an asynchronous value inside an async function.

```
async function main() {
    try {
        const result = await performAyncOperation(args)
        console.log("Operation completed with result", result)
    } catch (error) {
        console.error("Operation resulted in error", error)
    }
}
```

The design choice of asynchronous I/O over blocking and parallelism has a major impact on JavaScript software. Although the model was initially designed to enable easy event-driven interaction with the document object model (DOM) of websites, it has proven to be an efficient and productive paradigm for server software in the Node.js runtime. Server software written in Node.js does not suffer from thread-safety issues due to the single-threaded execution model and can achieve great performance in serving client requests [35].

4.1.4 JS-to-JS transpilation

The support of major browser JavaScript engines has for a long time been a limiting factor in adopting new language features by developers. This has been the case especially since the EcmaScript 6 (ES2015) standard which greatly renewed the language by introducing features such as `let` and `const` variable declarations, arrow functions, default parameters and the class syntax for objects. Implementing all features of the new specification took years from browser vendors and even then there was a significant market share of old browser versions with no ES6 support.

To overcome this limitation, some developers leveraged JavaScript-to-JavaScript transpilation: the source code was written using the new syntax features and then compiled to JavaScript that only uses the legacy features available in most browsers in the market. The most popular such project is Babel (originally 6to5)[4]. Babel is essentially a compiler that uses JavaScript as both input and target. While ES6 is fully supported by major browsers today, Babel has evolved to support varying newer JavaScript standards as

[4]https://babeljs.io

input. Additionally, it can be used to transpile completely unimplemented features that are still in the proposal phases of standardization or convenience features that aren't a part of the JavaScript language such as userland macros or the JSX syntax introduced by the React.js UI library.

Another reason for preprocessing JavaScript is that a syntax-level module system is a relatively novel feature in the language and especially in browser implementations of JavaScript [15]. The common practice for using modules in browser-targeting JS has been to use a userland module system and preprocess the source into a single file with all the dependencies. Node.js includes its own module system and doesn't require bundling files.

Transpilation and other preprocessing steps such as bundling JavaScript modules to a single source file have improved the productivity of JS development by allowing the use of more powerful features and automated workflows. However, it has also increased the complexity of typical web projects by introducing new tools and configurations. Most modern JavaScript projects involve a compilation or preprocessing step.

4.1.5 Ecosystem

For a long time, the dominant distribution method for JavaScript libraries was loading and executing library source files in the browser and assigning the interface to the library to a variable in the global scope. The files were loaded either from a content delivery network (CDN) or from the web server of the site. This was practical for websites with few dependencies (typically large general utility libraries such as jQuery). By using popular CDNs the library sources could in some cases be cached client-side if the same exact library source file was used in multiple websites.

With the emergence of JavaScript outside browsers, the browser-centric method of managing dependencies was no longer sufficient. in time the ecosystem transitioned from distributing libraries as independent source files to the usage of package managers and bundling assets partially due to the need to use JS libraries in environments other than browsers. This additionally enabled the increased use of smaller libraries as the work needed to maintain the dependencies became easier. The error-prone practice of exposing library code through the global scope was also abandoned as a result.

The Node Package Manager (NPM)[5] is a popular package registry used in JavaScript development. The command-line interface (CLI) to NPM is distributed with the Node.js JS runtime. The NPM registry is run by a US private company NPM Inc. Yarn is an alternative package manager

[5]https://www.npmjs.com

by Facebook that uses the NPM registry as a source for packages. NPM packages are nowadays the preferred method of distributing and consuming 3rd party JavaScript for the majority of developers and library authors both in browsers and server-side applications. With over 1 million available public modules, NPM is the biggest software registry in the world.

4.2 Known difficulties

The preceding sections introduced the JavaScript programming language and the conditions under which it was initially created and later developed. These conditions have resulted in a set of known difficulties in JavaScript development. In his 2008 book *JavaScript, the Good Parts* [7], software engineer Douglas Crockford writes

> JavaScript is a language with more than its share of bad parts. It went from non-existence to global adoption in an alarmingly short period of time. It never had an interval in the lab where it could be tried out and polished.

While the language has seen major advancement since then, many of Crorkford's ideas remain true for modern JavaScript. Meanwhile, the nature of applications using JavaScript and the general consensus on good software development practices has evolved.

4.2.1 Type coersion

Due to the lack of static typing and compile-time type checking it is the developer's responsibility to make sure that the type usage of a program is correct. As specified in the ECMAScript specification, JavaScript rarely throws an error if an operation is applied to values it was not intended for [15]. An example of this is the summation of a string and a number.

```
10 + "text" === "10text" // true
```

In this case, JavaScript is less strict than other popular dynamically typed languages. For example, in Python, the summation would result in an error. One can argue whether this is an expected or desired result but usage shows that many of the coercion rules are not intuitive to developers working on JS codebases [31]. Unwanted type coercion can result in bugs that are hard to discover. The ECMAScript specification also includes two different equality

operators `==` and `===`. The difference is that the double equality symbol performs type coercion before comparing the values while triple equality always results in false if the types of the values compared do not match. It is a common practice to only use the latter due to the complicated rules related to coercion [7].

Even when using the triple equality, there is some unintuitive behavior in equality. The best known of these is the equality of the `NaN` (not a number) value with itself, i.e., `NaN === NaN` results in `false` [7]. This sort of exceptions to general rules requires the developer to be very aware of the edge cases when checking the results of computations.

4.2.2 Incorrect assumptions of scope

In JavaScript, the scope of variables is handled differently than in most C-like languages[6]. While other languages of the category have block scope JavaScript originally did not [7]. Instead, all variables were defined in either the global scope or in function scope. The ECMAScript 6 standard introduced two new variable declaration keywords `let` and `const` which define block-scoped variables. The older `var` declaration keyword remains function-scoped for backwards-compatibility. [15]

```
function (x) {
    if (x > 10) {
        var y = 1
        let z = 1
    }
    y++ // y is accessible outside the if block
    // z is not accessible here
}
```

The impact of a scope misconception is highly dependent on the programming style and paradigm used. Blocks are mostly used together with flow control keywords `if`, `for` and `while`. The looping constructs are rarely used in functional style programs but can be an important part of more imperative programs. Similarly, conditionals are typically in an expression form instead of statements in functional programming. It is also possible to create a block without an associated flow control construct but this is rarely used in practice.

[6] C-style refers to the syntactic style of programming languages. Other categories for this include Lisp and ML style languages.

4.2.3 `this` semantics

One of the known difficulties in JavaScript language semantics is the use of the keyword `this`. As mentioned in section 2.1, the role of the keyword is self-referencing through open recursion used mainly in object-oriented programming. The rules on what value is bound to the keyword are contextual and somewhat complex. This often results in runtime errors when the object referenced is not the one expected by the developer. This is a problem especially for programs written in the object-oriented style.

- In global scope, `this` refers to the global object
- In function context, the behavior depends on how the function is called. For simple function calls, `this` refers to the global scope unless the function is in strict mode.
- If a function appears as an object method or in the object prototype chain, `this` refers to the enclosing object.

The above rules state the very essentials of `this` behaviour in JavaScript. The behaviour can be altered with by calling a function through `call` or `apply` indirections or by using the `bind` method introduced in ECMAScript 2015. Another addition to `this` handling added by the standard is the the arrow (`=>`) syntax for function declaration. Functions declared with the arrow syntax use the `this` value of the enclosing scope. [15]

4.2.4 Standard library

An important building block of any practical programming language is the standard library it provides. JavaScript, however, has no standard library. JavaScript provides a collection of global objects[7] that provides some common functionality. This has resulted in functionality typically provided by standard libraries to be implemented in 3rd party libraries, resulting in an increased need for dependencies in programs and fragmentation of practices in codebases.

Some of the built-in global functionality has confusing semantics that has either been maintained to support existing web software relying on the current behavior or later changed to a more understandable logic. An example of the latter is the global `parseInt(string, radix)` function that converts a string value to a number. A common source of confusion is the behavior

[7]Not to be confused with the global object accessed with the `this` operator in global scope in JavaScript.

when the radix is not defined. In the current standard, the radix defaults to 10 but this was not always the case: in earlier versions, the deduced radix would depend on the first digit of the string [7, 15].

There is an active proposal[8] for adding a standard library to JavaScript in a backwards-compatible way. If standardized, the proposal could result in JavaScript projects of the future adopting a more unified style of programming. It is however likely that the possible standardization and implementation of the JavaScript standard library is still far in the future and in the meantime, codebases have to rely on a number of userland libraries for common programming tasks typically available in standard libraries.

4.2.5 Conclusion

This section has introduced several issues that developers face when working with JavaScript. In the following section, we'll demonstrate how languages that compile to JavaScript attempt to handle these. The following summarizes the core issues discussed so far

- inability to enforce correct value types in development
- difficult type coersion rules
- scope rules differ from other languages
- complex `this` binding.

While the first issue can be considered fundamental and a direct result of the initial language design, the rest are more specific and situational. For clarity we can group them under a more general term:

- unintuitive language semantics
  - difficult type coersion rules
  - scope rules differ from other languages
  - complex `this` binding.

There are several additional features that could be categorized under unintuitive language semantics but are omitted because of their rarity in modern codebases or because of their small impact on code quality. These include the `with` statement, `eval` and typed wrappers (e.g., `new Boolean(false)`)

[8] https://github.com/tc39/proposal-javascript-standard-library

[7]. These features also contribute to the number of language constructs that add confusion and may be the source of bugs.

Additionally, we've observed a difficulty that has emerged whenever newer versions of the language have attempted to fix existing problems with the language:

- the need for complete backwards-compatibility preserves bad language design.

We've now identified several issues that developers face when working with JavaScript as well as some root causes of the situation. The identification of the issues will be useful when inspecting the programming languages that compile to JavaScript in the next sections.

Chapter 5

Languages targeting JavaScript

The previous chapter introduced the JavaScript programming language, its origin, development, syntax and problems. This chapter examines several programming languages that primarily target JavaScript as their compile target. This means that the runtime for the languages is the JavaScript engine used and the executed code is the intermediate JavaScript source generated by the compiler. The first JS-compiled language to gain popularity was CoffeeScript. The language predates the EcmaScript 2015 standard and introduced several features later standardized in JavaScript. The language was dynamically typed and was influenced in its syntax and features by the popular dynamically typed languages of the time, Ruby and Python. Since the development of CoffeeScript, several new programming languages compiling to JavaScript have surfaced providing a variety of features to differentiate from JavaScript including static typing, alternative syntax and pure functional paradigm.

Although JavaScript as a compile target is unlike any traditional target including assembly languages and virtual machine bytecode, the compilation process does not differ that greatly from other compile targets – only the primitives available are different. As a Turing complete language JavaScript has no limitations in the actual computation performed when compared to any other computer system. Only interfaces, resources and performance limit program execution in practice. The most important practical limitations from the perspective of compiling to JS are the single-threaded execution model and the APIs for I/O provided by the JavaScript implementation.

The focus of the chapter is on certain characteristics of the languages that are essential to the research questions specified in chapter 3. These include the syntax, type system, standard library and ecosystem of each language.

5.1 TypeScript

TypeScript[1] is an open-source multi-paradigm programming language by Microsoft. It was initially released in 2012 and has since been in active development. TypeScript is a superset of JavaScript – i.e., any valid JS program is also a valid TS program. TypeScript was created to make large JS codebases more maintainable by enabling type-checking using optional type annotations. The compiler of TypeScript is implemented in the language itself and runs as JavaScript. [6]

For most parts, the programming paradigm of TypeScript follows that described in chapter 4: all JavaScript idioms are usable in TS. A program written in TypeScript can exploit the prototypal OOP model, higher-order functions for functional programming and the non-blocking asynchronous programming patterns.

The type system of TypeScript is not sound by design. This means that it is possible for a sufficiently typed program to encounter a runtime type error regardless of the compiler accepting the program. The sacrifice of soundness was necessary to support existing JavaScript code, APIs and patterns in TypeScript code – i.e., the existing ecosystem was prioritized over language safety. In addition to the support existing patterns unsoundness enables omitting strict typechecking where it is convenient. [6]

A notable goal of TypeScript is the full erasure of type information at runtime: the static types specified in the source code exist only at compile-time and are omitted in the resulting JavaScript. As there is no runtime representation for the types, type checking in runtime must be performed using the standard JS idioms available. Ideally, TypeScript should add no runtime overhead of boilerplate code and should compile to JS that resembles the original source code minus types. [6, 25]

```
function factorial(n: number): number {
  if (n === 0) {
    return 1
  } else {
    return n * factorial(n - 1)
  }
}
```

Listing 3: Factorial function in TypeScript. Note that both the argument and return type annotations are required for proper typing of the function.

[1] This thesis focuses on version 3.6 of the language.

5.1.1 Type system

Although not completely sound, the type system of TypeScript is highly sophisticated. It includes features such as structural type equivalence, subtyping, gradual typing and type operators [6]. To properly represent the highly dynamic type of programming used in typical JavaScript applications, the type system must simultaneously be lightweight and advanced. A lot of effort has been put into making the type annotation experience seamless. A static type system and its syntactic annotations are typically included in the initial design of a programming language – adding them afterward is a difficult task [30].

```
interface Container<T> { value: T }

function setEmpty(c: Container<{}>) {
  c.value = {}
}

const container: Container<{ a: boolean }> = {
  value: { a: true }
}

setEmpty(container)

container.value.a.valueOf()
```

Listing 4: A program displaying the unsoundness of the TS type system. The program passes compiler checks but throws a runtime type error for the last line. The guarantees of the type system do not hold for several contravariant cases, including assignment. The mutating function accepts its argument as it is a subtype of the parameter type expected although the assignment performed implies that contravariance or invariance is the correct rule for the call.

The typechecker of TypeScript is configurable. The compiler supports several configuration values that alter the strictness of checking. In this thesis, when referring to the capabilities of TypeScript to statically find programming errors, we assume the use of the strictest compiler settings[2]. Other

[2]The `strict` setting in TS compiler options enables the following settings: `--noImplicitAny`, `--noImplicitThis`, `--alwaysStrict`, `--strictBindCallApply`, `--strictNullChecks`, `--strictFunctionTypes` and `--strictPropertyInitialization`

options provided by the compiler include specifying the behaviour with JavaScript files in the project: the compiler can either reject JS files, include them in the proejct without any checking or attempt to typecheck JS source with the inference rules. [25]

The primitive types offered by the TS type system extend the family of types in JavaScript displayed in table 4.2. The additional primitives include the bottom type ($\bot$) `never`, top type ($\top$) `unknown` and the `any` type denotes a dynamic type [25]. While the top and bottom types are a fundamental part of type theory, they are not always implemented in practical type systems. Especially bottom – which is inhabited by zero values – is often omitted from type systems for simplicity [30].

The types of TypeScript follow a structural, not nominal, equivalence. The equivalence of types is determined by the members associated with the type instead of the name or constructor of the type [25]. This aligns well with the patterns used in JavaScript: objects are often created ad-hoc for data storage without using classes for construction. Nominal typing would not support existing JS practices in a similar way [6]. As a programming language with strong object-oriented features with structural type system TypeScript stands out among most mainstream programming languages.

In addition to structural typing, TypeScript relies on subtyping in its typing rules. Instead of each value inhabiting a single type, values are part of families of types with their unidirectional hierarchies. Subtyping is a well-researched section of type systems typically associated with object-oriented programming languages. In the example below, $C_{3D} <: C_{2D}$, i.e., the 3D coordinate is a subtype of 2D. Subtyping is not limited to object types in TypeScript: other subtype relations include a string literal as a supertype of string and an array of type T as a subtype of tuple with members of T. Functions follow subtyping principles with contravariant argument types and a covariant return type as introduced in subsection 3.3.2.

TypeScript supports the use of an advanced type system feature called type operators [25]. Unlike normal operators of JavaScript or TypeScript, these are resolved at typecheck time instead of evaluation level. Type operators help in strongly typed data manipulation without having to repeat type information in multiple types. The language offers type-level operators for union and intersection types and constructing types with parameter polymorphism. Type operators are a relatively rare feature in conventional programming languages.

Another important type system feature in TypeScript is the concept of conditional types. They allow selecting the result of a type-level computation based on a type relation (subtyping) check. The feature is essential in representing several possible type-level operations, one of which is demonstrated

```
function print2DCoordinate(c: Coordinate2D) {
  return "(" + c.x + ", " + c.y + ")"
}

interface Coordinate2D {
  x: number
  y: number
}

interface Coordinate3D extends Coordinate2D {
  z: number
}
```

Listing 5: Subtype relation in TypeScript. Note that the subtype does not need to be explicitly introduced as subtype. The type system infers such relations from the structure of a type automatically. The function `print2dCoordinate` can be supplied with a 3D coordinate without type errors.

in listing 6. Conditional types combined with the `infer` type operation enable the TypeScript type system to model important non-trivial relations at the type level. [25]

```
type NonNullable<T> = T extends undefined | null ? never : T

type X = NonNullable<number> // number
type Y = NonNullable<number | undefined> // number
type Z = NonNullable<null> // never
```

Listing 6: The definition of the predefined conditional type `NonNullable` that omits nullable values from the parameter type. Note the usage of the bottom type `never` in the definition. In the case of an union type, the options that are transformed to `never` are omitted from the results since the value is implicitly a subtype of any type.

The TypeScript type system supports some degree of type inference to improve the readability of code: without inference, the annotations would make codebases verbose and hard to read and write. The power of type inference in TypeScript is lesser than in Hindley-Milner type systems. The types of function arguments are not generally inferred in TypeScript and always require annotations to avoid dynamic type behavior in the function

body. The types of local variables and function return value can often be inferred based on the annotations of the arguments. Additionally annotating the return type in the function signature can be used for readability and to ensure the correct type behavior of the function. [25]

The design of the TypeScript language and its type system sacrifice some of the guarantees in correctness for allowing existing development styles. This is manifested in the index access signature of TypeScript arrays. The signature for `Array<T>` is `T` while a semantically correct one would be `T | undefined` where `|` is the union type operator. This can result in runtime errors in programs that pass the compiler typecheck. This behavior follows the convention of both JavaScript and several strongly typed languages where the responsibility of handling out-of-bounds access of arrays is left for the author of a program. The index signatures are not the only source of unsoundness in sufficiently typed TS programs: there are variance-related edge-cases where the typechecking fails by design.

```
const a = [1, 1, 3]; // a: Array<number>

let b = a[3].toString(); // b: string
// TypeError: Cannot read property 'toString' of undefined
```

5.1.2 Ecosystem

The design decision of gradual unsound typing pays off in the ecosystem potential of TypeScript. The interoperability of TypeScript with existing JavaScript code is seamless at its best. Existing JavaScript, being a subset of TS, is readily usable from TS code making the entire JavaScript ecosystem usable in TypeScript. Additionally, parts of the JavaScript source can possibly be typed due to type inference and the availability of type declarations for the language globals. Because of this TypeScript has no separate package manager or ecosystem but instead leverages the Node Package Manager.

TypeScript libraries are typically distributed as compiled JavaScript bundled together with a TS declaration file containing types of the public API (exported members) of the library [25]. This enables easy usage of the library from both JS and TS codebases while preserving the essential type information for TS use. The internal type details of the library should not be a concern for the library's consumer in good modular program design. Consuming libraries distributed as TypeScript source in TS projects would also require the configurations for the TS compiler to match, making compiled source files with type declarations a better choice for portability.

Similarly to JavaScript, TypeScript provides no standard library. Adding one would go against the design philosophy of full runtime erasure of TypeScript. The globals provided by JavaScript are available in TypeScript and type declarations are available for the globals of most common JavaScript runtimes. TypeScript is not limited to usage in browser: the language is usable in all the platforms where a JavaScript runtime is available. TypeScript is relatively popular in Node.js applications with detailed type declarations available of the Node.js APIs.

5.2 Elm

Elm[3] is a statically typed functional programming language for constructing user interfaces. Unlike the other programming languages introduced in this thesis, Elm is not designed to function as a general-purpose programming language but to be used exclusively in UI programming [10]. The language was created in 2012 by Evan Czaplicki [9]. The language is syntactically a member of the ML family of programming languages. Elm has its roots in the functional reactive programming (FRP) paradigm but has since evolved to a more conventional functional programming paradigm [9]. Elm follows a very opinionated and structured architecture for building applications and is powered by a static type system with features such as type inference and row polymorphism. The compiler of Elm is mainly written in Haskell, i.e., Elm is not self-hosted.

The Elm documentation claims that the compiler of the language produces high-performance JavaScript with no runtime errors. Additional features include automatic semantic version enforcement (using static analysis) and interoperability with existing JavaScript libraries. The programming model of Elm is highly declarative: a program mainly specifies what the layout and data should be, not how the state should be achieved. The programming model has many similarities with modern JavaScript UI libraries including React. The Elm architecture has inspired some of the solutions used for application state management in JS libraries, most notably Redux. [8]

Unlike in TypeScript, an Elm project compiled to JavaScript comes with a runtime library of its own. The runtime provides the core of the Elm architecture and utilities including data types and standard library functions. The Elm documentation claims that the runtime overhead produced by the language is minimal due to optimizations performed. [8]

[3]Version 0.19 of the language is used as a reference for the discussion on Elm.

```
factorial n = case n of
  0 -> 1
  x -> x * factorial (x - 1)
```

Listing 7: Factorial function in Elm. The types of the expression are fully inferred with no annotations needed. The function is written using ML-style pattern matching instead of the `if` control expression.

5.2.1 Type system

Elm, similarly to other languages of the ML family, bases its type system on the Hindley-Milner type system. The language, therefore, features strong type inference and soundness. Similarly to most practical ML languages, Elm extends the HM type system with additions including row polymorphism.

The Elm programming model encourages using the type system extensively to model the logic of programs. Elm does not provide a mechanism for runtime errors – instead, computations with the possibility of failure should be modeled using types that convey the possibility [8]. Such types include `Maybe` and `Result`. `Maybe` is a variant (also sum, union) type with the following definition: `type Maybe a = Some a | Nothing`. Consider indexed array access `get: Int -> Array a -> Maybe a`. The result of the operation is returned as an optional value of the array item type `a`. Out-of-bounds access is trapped by the runtime system and the unit value `Nothing` is returned. The typechecker ensures that code calling the array index access operator handles the faulty case somehow. The use of sum types to handle possibly failing computations is not unique to Elm – in fact, most functional programming languages support such constructs. However, the complete lack of an exception construct is a rather unique feature among mainstream programming languages.

The type system lacks higher-kinded polymorphism, a key feature in many statically typed functional programming. The absence of the property prevents the instantiation of generic functions over containers, including `map` and `filter`. Instead, each data type must declare its own functions. Similarly, the concept of type classes is missing from the language. Similarly to Haskell, Elm implements row polymorphism for records, also known as extensible records in the language. Row polymorphism enables function to operate on record types with at least the required fields as demonstrated in listing 8.

One of the focuses in the design of Elm is the ability to produce highly

```
point2d = { x = 1, y = 2 }

point3d = { x = 1, y = 2, z = -1 }

printPoint2d : { a | x: Int, y: Int } -> String
printPoint2d p =
  "(" ++ String.fromInt p.x ++ ", " ++ String.fromInt p.y ++ ")"

main = printPoint2d point3d
```

Listing 8: Row polymorphism in Elm. Without the row type specifier `a` the call to the function would be a compile-time type error due to the additional fields not specified in the input type. Elm is able to infer a row-polymorpic type for the function without the explicit annotation.

helpful compiler error messages[4]. A large number of compiler errors are related to invalid use of types as many categories of errors from simple typos to incoherent design manifest as type errors in strongly typed programming languages. The Elm compiler is often able to deduce not only the location and sort of the error but also the intent of the programmer and a probable fix for the problem. Structural type systems, like that of Elm, are often considered inferior to nominal type systems in producing human-readable type errors [24]. The Elm compiler goes to great lengths to make the structural type errors readable by e.g. displaying only the difference of the expected and received type and using a heuristic for determining possibly misspelled record field names.

An important addition to the Elm type system is the runtime decoding facilities provided by the `elm/json` package. All external data passed to Elm must go through decoding to be used in the code – there is no method for simply asserting the type. This ensures that foreign values do not add inconsistency to the type behavior of the application. Consequently, the forced requirement for type-safe decoding causes friction in adding new data sources, reducing the viability of Elm for rapid prototyping.

5.2.2 Ecosystem

Elm provides its own package manager[5] with a public library for publishing software packages and a command-line tool for installing packages. The

[4]https://elm-lang.org/news/compiler-errors-for-humans
[5]https://package.elm-lang.org

size of the package ecosystem is notably small with the number of packages available through the official package registry measuring in hundreds.

Elm has features for interoperating with JavaScript code. The port construct in Elm enables one-way message passing between Elm and JS. The port mechanism requires creating the binding for messages on the JS side in addition to the logic in Elm. The interoperability is thus not as seamless as in languages including TypeScript. Elm code can also be interfaced from JavaScript code. This requires compiling the project with a JavaScript output option. The program can then be initialized from JS with support for passing JS values (called flags in Elm) to the program initialization.

Other compile targets besides JavaScript are not on the Elm roadmap. According to the language documentation, this is due to the need for a viable ecosystem for new targets, not because of the technical difficulty of implementing a new target for the Elm compiler. Server-side use of Elm is similarly not a goal of the language at the moment. [8]

The Elm package ecosystem is small compared to most languages. However, the language requires a smaller amount of libraries as it is only intended to be used in web-targeting user interface programming. Additionally, the language runtime provides higher-level tools than those of most general-purpose programming languages. These include rendering to the DOM from the declarative HTML interface which in JavaScript is implemented by user-land libraries, most notably React.

5.3 ReasonML

ReasonML[6] (also Reason) is a combination of syntax extension and toolchain that forms a programming language of its own. It is based on the OCaml programming language developed at Inria in 1996. ReasonML adapts the ML-syntax of OCaml to resemble JavaScript. Code written in ReasonML is compiled to JavaScript using the Bucklescript OCaml to JavaScript compiler. ReasonML was created in 2016 by Jordan Walke at Facebook.

It is useful to consider ReasonML as a separate programming language from OCaml in the context of this thesis. The target platform, standard library and ecosystem differences that are further explored in the following sections make it more intuitive to consider ReasonML as a new language with its roots in OCaml. The design goal of the language is to enable a relatively familiar development experience to JavaScript developers while providing the rich type system of OCaml to enhance the development workflow. The

[6]Version 3.5.0 of the language is used as a reference in this thesis.

specification and documentation of the OCaml language and type system can, however, be used as a reference when discussing the ReasonML language.

ReasonML (with the underlying Bucklescript) is not the first implementation of OCaml on JavaScript. In addition to the Bucklescript project, the Js_of_ocaml package compiles OCaml to JS. The main difference between Bucklescript and Js_of_ocaml is that the former operates on OCaml source code[7] while the latter transpiles bytecode generated by the OCaml compiler.

The programming paradigm of ReasonML and OCaml is typically considered to be functional programming. However, from a strict perspective, the language is not purely functional. By design, ReasonML provides several non-functional constructs including reference types and mutable records. The "O" in OCaml originally stood for objective as the language introduced several object-oriented constructs to the ML programming language family. ReasonML can thus be alternatively considered a multi-paradigm language with its combination of functional, imperative and object-oriented features. The functional language features are however highly preferred in good programming style. Most programs can be written in a purely functional style without the usage of the imperative features. [28]

```
let rec factorial = n => switch (n) {
    | 0 => 1
    | n => n * factorial(n - 1)
};
```

Listing 9: Factorial function in ReasonML. Both the argument and return type are inferred based on the usage of the arithmetic operators. However, the function declaration must be annotated with the `rec` modifier.

ReasonML adapts the ML-style syntax to more closely resemble modern JavaScript. The changes are entirely cosmetic and any program can be expressed in either of the syntaxes. The ReasonML core ecosystem includes tools for converting between the two syntaxes programmatically. [36]

Due to the large number of features in the OCaml language, the syntax of the language has become somewhat crowded. Parentheses are often needed in expressions to ensure correct parsing and some side-effectful statements require the use of two consecutive semicolons. In nested `case` expressions it is easy to add cases to an expression not intended[8]. ReasonML

[7] More accurately Bucklescript processes the internal lambda representation of the ReasonML source code produced by the OCaml compiler. This is still a higher level representation of the source than bytecode.

[8] Though the typechecker should be able to catch such errors immediately.

attempts to clarify the syntax by introducing C-style constructs like braced code blocks. Several language constructs and operators have additionally been transformed to resemble JavaScript instead of ML: these include equality operators, comment syntax, list syntax and variant constructors. [36]

5.3.1 Type system

OCaml and ReasonML share an advanced Hindley-Milner based type system. The core features of HM, completeness and strong type inference, are thus present in the language. OCaml and ReasonML include several imperative features that require extending the HM type system in a type-safe way. These include mutable references, looping constructs and mutable objects. [36]

Some practical trade-offs are required to enable the sound and inferred typing of programs. One of these is the monomorphism of operators. This is demonstrated by the numeric operators that differ for integer and floating-point numbers.

```
let a: int = 1 // The type annotation is optional
let b = 2
a + b

let r: float = 10.0
let pi = 3.141
let c = pi *. r *. r
```

Type annotations are rarely needed in ReasonML but can be useful for documentation and verifying purposes. Some advanced features, including combining recursion and polymorphism, require annotations for type inference. Additionally, recursive functions require a special `rec` keyword in the definition. Mutually recursive functions must be defined together. [36]

The record type of ReasonML, unlike the rest of the language, is nominally subtyped. This design decision has some important implications. First, the use of records is a bit less versatile as the condition for subtyping is stricter. Second, and importantly, this enables the optimization of records: the record can be transformed to a tuple in the compiled code with the labels omitted. The lookup of a record value is now a constant-time operation in runtime and the required memory of the object is smaller. The following record instance

```
type language = {name: string, created: int};

let re: language = {name: "ReasonML", created: 2016};
```

produces the following JavaScript (with comments omitted):

```
'use strict';

var re = ["ReasonML", 2016];

exports.re = re;
```

While the optimization originates from the native OCaml implementation where record field lookup can be compiled to a very small amount – ideally 2 – machine instructions [36], it applies to generated JavaScript, too. The gained benefit can be observed both in reduced computational complexity and source file size, which is an important metric for web content delivered via network. In this specific case, it is justifiable to declare that ReasonML produces better JavaScript than the equivalent written directly in JS.

5.3.2 Ecosystem

Discussing the ReasonML ecosystem requires understanding multiple partially overlapping programming ecosystems of their own: the OCaml ecosystem, the Bucklescript ecosystem and the JavaScript ecosystem. ReasonML exists in an intersection of all of the former with full or partial support for each.

The ReasonML package ecosystem is directed by the use of Bucklescript which focuses on the existing JavaScript ecosystem and NPM. The native OCaml package manager (OPAM) is less focused on – it is possible to compile some native OCaml libraries to JavaScripr with Bucklescript but in general this is limited by missing primitives in the JS compile target. Primitives not translated include threading and I/O constructs not available in JavaScript runtime.

Both ReasonML and Bucklescript are designed for light interoperability with the JS ecosystem. There are bindings available for several JS libraries, including the immensely popular UI library React.js (also by Facebook). ReasonML additionally supports a syntax corresponding with the JSX syntax which enables a familiar syntax for HTML elements and custom components to be rendered.

ReasonML supports embedding arbitrary JavaScript code inside the source file. This provides an easy escape hatch when working with existing JS. Additionally, ReasonML code can interact with the embedded JS by e.g. binding the result of a JS expression to a variable (annotated with a type). While such features can help in the initial integration of ReasonML and JavaScript,

this is not a recommended long-time solution as it is a possible source of type errors not caught by the typechecker. Instead, JS code should be interfaced through foreign function interface (FFI) external values. ReasonML provides methods for declaring single external values, e.g., JavaScript globals or providing bindings for entire libraries. The main target of the ReasonML project is the client-side web. The workflow for targeting Node.js using the language is still under development and has not stabilized. [36]

One of the design goals of Bucklescript is to produce clean and human-readable JavaScript with a clear one-to-one mapping to the original ReasonML or OCaml source code. This helps in integrating ReasonML source in existing JavaScript projects. The compiler includes analysis tools for eliminating unused code which prevents the addition of unnecessary standard library overhead. The mapping between JavaScript and ReasonML primitive types is simple and ideally adds little overhead.

5.4 Dart

Dart[9] is a multi-paradigm programming language created by Google in 2011. Dart targets multiple platforms with its support for JS compilation, native compilation and a virtual machine of its own (Dart VM). According to Google, Dart shares the design goal of TypeScript – that is, Dart intends to "make building large-scale web apps easier". Several major products of Google, including Google AdWords, AdSense, and Google Assistant, use Dart. Dart has undergone major changes between versions 1 and 2 including a transition from optional to mandatory static typing. [13]

Dart is an object-oriented language. Every value, including primitives, functions and null values, is an object instantiated from a class. Functions as objects subsequently implies that they are first-class members of the language. Additionally, Dart supports anonymous functions and lexical closures for nested functions, making functional programming viable in the language. Therefore Dart should be considered a multi-paradigm language. [13]

The Dart language is standardized in the ECMA standard ECMA-408. The latest published standard version (4th edition) describes the 1.11 version of Dart which is generally incompatible with version 2 of the language. The 5th edition of the standard [12] describes version 2.2 of the language – however, the standard is still at the draft phase.

Dart supports asynchronous programming using futures and async/await [13]. The programming model is familiar to JavaScript developers: futures

[9]Language version 2.5.2 is used as a reference.

represent the same deferred computation concept that promises do in JavaScript while async/await provides a very similar API to that presented in section 4.1.3

```
factorial(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

Listing 10: Factorial function in Dart. Argument type is needed for static typing while the return value type is inferred.

5.4.1 Type system

The type system of Dart, similarly to that of TypeScript, is unsound by design. In Dart, the intentional unsoundness is a result of the allowed covariance in generic classes (listing 11): the design prioritizes convenience over static correctness in the subject. Type errors in the use of covariant generic classes are caught at runtime. Due to the different meanings given to the term sound, the Dart documentation claims the type system to be sound due to the runtime trapping of the illegal covariance cases. The 5th Dart specification draft conversely mentions the unsoundness of the static type system in this regard along with an explanation of static safety getting in the way of developers in the case. From the perspective of the thesis, related to static typechecking, the type system of Dart is unsound. [13]

Dart provides the `dynamic` type specifier which instructs the typechecker to make no assumptions on the type of the variable [13]. This can be used to e.g. instantiate a list of varying element types `List<dynamic>`. The construct corresponds to the `any` type of TypeScript.

The Dart type system includes type inference features that significantly reduce the number of type annotations needed. When the typechecker can not infer a type, the type is assumed to be `dynamic`. The behavior of the compiler can be adjusted with configuration to the static analyzer to enable warnings for implicit casts and implicit dynamic types. [13]

The Dart type system follows the pattern of most object-oriented languages by the use of nominal subtyping. This means that only classes in the

```
class PredicateContainer<T> {
  PredicateContainer(this.predicate);
  bool Function(T) predicate;
}

main() {
  PredicateContainer<num> pc = PredicateContainer<int>((i) =>
    i.isEven);
  bool Function(num) f = pc.predicate;
}
```

Listing 11: A minimal program displaying the compile-time unsoundness of the Dart type system. The generic class, parametrized by the type parameter T, considers its type parameter covariant. However, as established earlier, function parameters are contravariant by nature. The assignment of the function in the last line of the `main` function is therefore invalid. This program compiles but throws a type error at runtime for the invalid assignment.

same inheritance hierarchy can exist in a subtype hierarchy – subtyping is explicit. [13]

5.4.2 Ecosystem

The Dart ecosystem is heavily influenced by its multiple compile targets. Dart targets JavaScript, native CPU architecture and its own virtual machine. Additionally, Dart is the programming language of choice for Flutter, a cross-platform mobile application framework by Google that targets iOS and Android devices. [13]

Another cause of divergence in the package ecosystem is the major language updates of Dart 2. Many packages developed for 1.x versions are incompatible with the new language versions. The presence of multiple targets – especially the mobile platform support – helps with the adoption of the language and increases the number of available packages. The core language is the same for each target enabling cross-platform development to some degree. Differences in platform APIs, however, results in platform-specific code. The Dart ecosystem features its own package repository[10] that hosts packages for both web and Flutter projects as well as cross-platform packages. Dart supports JavaScript interoperability through the `js` package which is still beta software according to the package documentation.

[10] https://pub.dev

5.5 Summary

The programming languages discussed in this chapter represent a wide range of programming styles and paradigms. The static type systems in the languages differ from each other significantly. The properties the languages have in common are the list of factors defined in section 3.3 – most importantly, the languages compile to JavaScript and feature a static type system.

Language	Type soundness	Type equivalence	Subtyping
TypeScript	No	Structural	Yes
Elm	Yes	Structural	Row polymorphism
ReasonML	Yes	Structural[11]	Row polymorphism
Dart	No	Nominal	Yes

Table 5.1: Core type system features of the researched languages.

Table 5.1 demonstrates the key differences in the type systems of the languages. The variety of type system features and design choices implies that there is no definite path for type-safe JavaScript development. Choices in type system features involve compromises in language features, safety and runtime performance as discussed in the chapter and in the background (especially section 2.2). The features presented in the table are far from exhaustive when considering the differences in type systems. Properties omitted from the table but presented in the chapter include type inference, type operators and interoperability features. These features also present important solutions in the design of the type systems.

Language	Syntax	JS interop	Registry	Target platforms
TypeScript	C	Direct	npm	Any JS runtime
Elm	ML	Ports	Own	Web
ReasonML	ML	Direct, annotated	npm	Any JS runtime[12]
Dart	C	dart:js module	Own	Web, native

Table 5.2: Programming language and ecosystem properties in languages researched. This table extends table 3.1.

In addition to the core language design, the choices related to the ecosystem surrounding the language play an important role in determining the usefulness of the language. Table 5.2 presents some key ecosystem properties

[11] ReasonML records are compared nominally.

[12] The primary focus being on the client-side web.

along with language features relevant to the programmer. Similarly to the compared type system properties, these present a diverse range of solutions for design choices in package management and JS interoperability. Of the languages discussed in the chapter, TypeScript is currently easily the most popular [34].

Chapter 6 further discusses the implications of the differences observed in the programming languages targeting JavaScript and explores how these insights could be used to develop the JS ecosystem towards an increasingly safe and convenient programming model.

Chapter 6

Analysis and results

After exploring JavaScript and a number of languages that target the language, we provide analysis on the state of the programming ecosystem and its options regarding type systems. The chapter is split into four sections. The first one discusses how the issues highlighted in JavaScript can be solved using the researched statically typed programming languages. The second section introduces an alternative method of addressing issues in JavaScript: static analysis of dynamically typed programs. Two applications of this, code quality analysis without type information and partial typechecking based on external type information, are discussed. These approaches are subsequently compared to using statically typed languages for the same problems.

The third section considers different future directions for the entire JavaScript ecosystem. Replacing JS with a statically typed language is only one possible alternative with several other options available. The section considers language extensions, type inference and a different compile target instead of JS. Additionally, the language evolution of Java and the JVM ecosystem is considered as a parallel to the JS ecosystem evolution. Finally, the last section presents several recommendations for languages targeting the ecosystem based on the observations from the discussed statically typed languages.

6.1 Solving JavaScript issues with static languages

In chapter 4 we explored the JavaScript programming language along with its issues in development. Chapter 5 introduced several modern programming languages that target the JavaScript ecosystem. This chapter provides a synthesis of how the introduced languages solve or mitigate the problems of JavaScript described based on the evaluation of the language features.

6.1.1 Runtime type errors and maintainability

As established earlier, the highly dynamic nature of JavaScript has proven to make large programs and codebases written in the language difficult to maintain. The documentation of two of the four languages described in chapter 5 explicitly mentions supporting the development of large-scale programs targeting JavaScript as a design goal or motivation of the creation of the language.

Each of the languages researched succeeds in preventing type-related runtime errors with static analysis which improves the maintainability of software. It is debatable whether type systems providing stronger guarantees (Elm) are superior in this regard when compared to languages with unsound practical type system features (TypeScript, Dart) but each obviously performs the task better than dynamically typed JavaScript. This includes basic errors including referencing undefined variables, calling non-function values and null-pointer issues but also more complex problems in software design can be revealed by the type system.

Of the languages described, TypeScript and Dart mention supporting large-scale software in their design goals. Both research and practice show that compile-time typechecking indeed improves the maintainability of software especially as the scale grows. Improved static analysis, including type analysis, additionally enables better tooling for programming tasks including refactoring.

6.1.2 Confusing language semantics

One of the challenges identified in JavaScript development was the presence of legacy language constructs and features that are not intuitive to developers, both of which can result in bugs. A significant amount of work has been put into recent JavaScript specifications to reduce the harm caused by e.g. function scope.

Each of the programming languages presented earlier – apart from TypeScript – manages to avoid the issues simply by relying on different syntax and operational semantics entirely. It is safe to say that programming language developers are well aware of the issues caused by the mentioned design decisions in JavaScript. Modern languages tend to discourage type coercion except in the most obvious places or make it explicit rather than implicit. Similarly, scoping rules tend to follow the practices set by previous languages. The usage of global variables is also mostly discouraged making access to `this` object unnecessary outside object methods in object-oriented languages.

TypeScript maintains the language features of JavaScript with its superset design principle. This includes the `var` scoping, `this` semantics and function scope. Fortunately, the type system helps in catching misuse of the features. Many coercion errors are caught by rigorous typing of the variables and the implicit `this` argument of functions can be typed statically. In general, the problems categorized as legacy features, are highly JavaScript-specific. E.g., function scope is a design choice not present in any other mainstream programming language.

6.1.3 The problem of backwards-compatibility

The need to maintain versionless backwards-compatibility was earlier identified as one of the causes of the complexity of the JavaScript language and its syntax. New features and improvements to previous ones are mostly added in an append-only style by introducing new APIs or language constructs. This requirement is based on the nature of the web platform but influences all JavaScript runtimes through the standardization process.

One cause of the need for versionless backwards-compatibility in the web is the interpreted nature of JavaScript: compiled languages offer a natural solution to the backwards-compatibility issue through versioned compiler updates with only the compile target remaining fixed. The Elm language has experienced major breaking changes between versions, most importantly when transitioning from the functional reactive programming paradigm to functional programming. Regardless of this, the applications developed on earlier versions of Elm and deployed to the web continue to function due to the compilation step. Non-backwards-compatible version updates in programming languages can result in divergence of the ecosystem as libraries and community have to either support multiple versions which requires additional effort or to choose a version while ignoring the others – this can be observed in both Elm and Dart ecosystems both of which have experienced major changes since creation. The option of breaking changes, however, empowers the evolution of programming languages over time.

Of the languages discussed in chapter 5, Elm, ReasonML and Dart can be considered to solve the issue of versionless backwards-compatibility. None is bound by the semantics of JavaScript. In fact, such languages can benefit from the stable change policy in JavaScript as the produced output of the compiler remains valid in the future. On the contrary, TypeScript is not free of the constraints of JavaScript language evolution. Being a superset of JavaScript, TypeScript is bound to preserve the known bad parts of JavaScript and rely on static analysis to steer users to prefer the features considered superior. It is possible that the future direction of TypeScript involves drop-

ping support for features deemed unproductive but so far there have been no signs of this.

6.2 Static analysis of dynamically typed languages

Static typing is not a requirement for static analysis of code. Dynamically typed languages, including JavaScript, Python and Ruby, have tools for detecting possible programming errors before runtime and for performing mundane tasks, including refactoring and renaming. The degree of static analysis capability available is however reduced by the use of dynamic typing. Without the type information available statically (at analysis-time), it is more difficult to deduce properties of the code. The combination of static and dynamic programming constructs (and languages) may result in better static analysis than the use of purely dynamic languages. This synergy is already in use in IDE tools of selected programming languages. In this section, we introduce two methods of improving dynamically typed programming process through static analysis. The first uses only information statically available without static type information and the second refines the process through external static type information that is made available in the dynamically typed source code.

6.2.1 Code quality analysis

Tools for static analysis in dynamically typed languages have existed for a long time. These rely on properties other than types, e.g., the structure of single statements and expressions. This type of static analysis can yield several interesting results that help the programmer without any type reconstruction and can address several of the issues discussed in section 4.2.

In compiled languages, the static analysis is typically performed by the compiler which can issue warnings and errors based on the results. This was not always the case: in early compilers, the analysis capabilities were primitive and external programs (e.g., the Lint program for the C language) were created and used for further analysis. Interpreted languages have no compile step where such warnings could be issued[1]. [7]

For JavaScript, there are several static analysis tools for enforcing code

[1]The interpreter can issue runtime warnings for statically or dynamically analyzed problems. This is however closer to runtime analysis.

style available including JSLint [7], JSHint[2] and ESLint[3]. These are collectively referred to as linting tools or linters after the C analyzer. The tools are highly configurable and detect code issues based on a set of rules. By integrating the tools into the development environment, they can provide similar guidance to that of an IDE with statically typed languages. The set of programming errors that can be caught using this strategy includes syntax errors, usage of uninitialized or undefined[4] variables and stylistic problems defined by the tool or the configuration.

Linting tools provide a viable solution to many of the identified JavaScript issues without static typing. Especially when combined with the feature additions of the ES2016 specification it is relatively easy to enforce the usage of new features over legacy alternatives, e.g., `let` and `const` over `var`. This removes issues related to JS scoping rules. Similarly, the use of the coercing equality operator can be discouraged to partially mitigate the issue of accidental type coercion.

However, the tools are inefficient in addressing some fundamental issues we have discussed. A majority of type errors can not be mitigated by the static analysis without type reconstruction. Type errors related to referring to undefined variables can be caught but a majority of other errors, including attempting to call a value that is not a function or accessing a nonexistent object field, are beyond the capabilities of the analysis. The help offered by linting tools for the scalability of software projects is limited due to this.

6.2.2 Type reconstruction for dynamic languages

Several dynamic languages, including JavaScript, support an informal annotation style for documenting the type signature for functions. JSDoc[5] is a commonly used annotation specification for JS using comments for annotating code. These comments can then be used to generate documentation for the codebase.

Modern IDEs can use the types defined in the comment annotations for static type reconstruction [26]. With JSDoc, a function may, for instance, have its parameter and return value types documented. In a limited number of cases, this information can be used in two different purposes: the parameter type information can be used to perform local inference to determine if the return type defined is correct. More importantly, the analysis can be ex-

[2]https://jshint.com

[3]https://eslint.org

[4]In this context, we use the word undefined to denote variables that have not been introduced in the code. The identifier `undefined` is used for the similarly named JS value.

[5]https://jsdoc.app

tended beyond the function scope by substituting the default dynamic type of calls to the function with the defined return type. Ideally, if the JSDoc annotations are correct and sufficient, they can be used for full type reconstruction in the codebase, making the code statically typed. This approach, however, relies on the correctness of the JSDoc annotations which is hard to verify – especially in living codebases. Additionally, the annotation language is mostly concerned with rather primitive type constructs and is thus unable to present more complex types available in advanced type systems.

Another prominent source of type information for statically typed languages – and especially JavaScript – is the use of libraries and APIs written in a statically typed language. Currently, the most promising form of this is the use of libraries written in TypeScript in JS. The libraries are compiled to JS before use but retain the declaration file that exposes the types of the exported members of the library. Alternatively, the libraries can be written in JavaScript and the types of exported constructs (as deduced by the programmer) can be encoded in hand-written TS declaration files.

The approach of using TS type declarations of functions is exploited by the Visual Studio Code IDE by Microsoft. It essentially treats JS code as TypeScript with implicit `any` values enabled and attempts to gradually reconstruct the types. This approach can improve the programmer's understanding of the codebase and help in avoiding programming errors. In addition to the used libraries, type information can be inferred from variable declarations and the usage of primitive values. With TS declarations of JavaScript global primitives (e.g. `parseInt`) and platform-specific global values (e.g. `window`, `process`) available, the amount of available type information further increases. [26]

6.3 Future directions for the JavaScript ecosystem

This section provides a synthesis of the explored programming language design in this thesis. It presents possible future directions for JavaScript and the larger ecosystem surrounding the language. The insight from obtained by observing the type systems of the researched languages is combined with results from other programming language ecosystems as the design space is interconnected and mainstream programming languages influence each other significantly.

6.3.1 Type annotation extensions

One of the possible courses of programming language development is the extension of existing programming languages with types. In the JavaScript ecosystem, both TypeScript and Flow are prime examples of this. While TypeScript is generally considered a separate language from JavaScript, Flow is often described as a static analysis tool or typechecker for the JS language. Flow notably supports including its annotations inside JavaScript comments, making the syntax entirely valid JS without a compilation step.

Outside the JS ecosystem, another popular dynamic scripting and backend language, Python, has adopted the use of optional type annotations[6]. Unlike in JavaScript, the annotations are part of the core language specification. They are not enforced at runtime but are exclusively intended for static analysis, similarly to Flow or TypeScript. Clojure, a Lisp style programming language for the Java Virtual Machine (JVM) also supports type annotations as a library[7].

Improving a programming language through optional type support comes with several benefits over transitioning to a new language. The most important ones are the compatibility with existing package ecosystems and libraries and the familiarity to a wide audience of developers. However, achieving soundness in such a setting can be a difficult or completely unrealistic goal depending on the language. Additionally, simply annotating each expression with a type will not result in static safety: many of the common dynamic idioms of dynamically typed languages will require rewriting to fit into the stricter set of valid programs approved by the typechecker.

6.3.2 Type inference

A notable trend that is observed in the languages discussed in this thesis is the increased usage of type inference in modern statically typed programming languages[8]. This can be considered a rational development: if type information can be obtained without programmer interaction – as is often the case – this method should be preferred over type annotation.

There is a significant difference in the degree to which different programming languages provide inference: in languages with a Hindley-Milner type

[6]https://docs.python.org/3/library/typing.html

[7]https://github.com/clojure/core.typed

[8]The adoption of type inference can be considered modern development only in the context of mainstream programming languages – in academia, the topic of type reconstruction has been researched for several decades and the feature is incorporated in multiple research programming languages. See subsection 2.2.5 for one such system.

system, the type inference is generally complete while imperative languages often provide inference to a much lesser extent – e.g., exclusively for local variables and function return values assuming argument types are known.

The increased availability of type inference should intuitively lower the barrier for adopting static typing in software engineering. The rationale is that a developer will, in any program, statically or dynamically typed, consider the types and not e.g. attempt to pass an integer for a function expecting a string. Thus removing the requirement for annotating this intention in static typing context essentially results in zero-cost typechecking. Unfortunately, a type system with complete inference of types places restrictions on the language features available. The adoption of the functional programming paradigm increases the degree of inference without sacrificing expressiveness.

It is safe to assume that future programming language development involves type inference. The trade-offs made between language features and inference depend on the chosen programming paradigms and discoveries in the research of type systems. The ReasonML (and OCaml) programming language demonstrates how a single language can incorporate strong type inference, an expressive type system, object-oriented and imperative features (in addition to the functional core) and good performance with a combination of smart design choices.

Furthermore, type inference for JavaScript has been a subject of research for years with some work preceding the emergence of modern web applications and server-side usage of JS [4, 5]. Type inference for the language has been researched for, in addition to correctness, performance [17]. Due to the highly dynamic nature of JavaScript, complete type inference of the entire language is unlikely to be achieved. A usable subset of the language could, however, be typed soundly without any annotation as Anderson et. al. [4, 5] have demonstrated.

6.3.3 Parallels to the JVM ecosystem

The inspection of JavaScript, its ecosystem and languages that target it raises an interesting parallel with the ecosystem of Java Virtual Machine (JVM) and Java. Java is an object-oriented programming language created in the 1990s. One of its goals was “write once, run everywhere” – i.e., a program written and compiled for one target OS should be executable on another one without modifications [23]. The goal was approached though JVM, a virtual machine that consumes its proprietary bytecode format, therefore enabling bytecode execution in any operating system and architecture where the virtual machine is available.

Java became one of the most widely used programming languages in the

Language	Released	Type system	Paradigm
Java	1995	Static	Object-oriented
Scala	2004	Static	Object-oriented, functional
Groovy	2007	Static, dynamic	Object-oriented, functional
Clojure	2007	Dynamic	Functional
Kotlin	2011	Static	Object-oriented, functional

Table 6.1: Languages of the JVM ecosystem. Compare to 3.1

1990s. The library ecosystem and the virtual machine went through major development resulting in a highly advanced programming environment for Java. However, over time the core language became burdensome for developers. Some of the design decisions of the language didn't age well: developers preferred more compact languages while Java was relatively heavy syntax-wise with static types and no type inference available. Similarly, Java's choice of allowing null values to be assigned for any reference variables made development error-prone and greatly reduced the usefulness of the static type system in asserting correctness.

The burden of using Java to target the existing environment was removed by creating new programming languages to target JVM. Currently, the JVM ecosystem is a host for a number of languages, the most widely used ones being Java, Scala, Clojure and Kotlin. In addition to the new languages, several existing programming languages have been adapted to run on JVM with the Jython implementation of Python and JRuby version of Ruby among the most prominent ones. [23]

Initially, the virtual machine running the languages was highly optimized for only Java. This resulted in inferior performance for languages with major core differences with Java. E.g., the dynamic type system of Clojure caused overhead in execution. Since then the development of JVM has added support for several features that have enabled high performance using alternative JVM languages – including support for dynamic types. [23]

The parallel that emerges from the JavaScript and Java ecosystems includes the initial popular programming language (JavaScript, Java), the advanced and ubiquitous language runtime (JS engines, JVM) and a number of individually developed programming languages that target the runtime for existing ecosystem benefits or the external requirement to target the runtime (the browser environment, Android).

One conclusion that can be reached by inspecting the development of the two ecosystems is that the popularity of a programming environment inevitably results in the development of advanced tools to increase the produc-

tivity of programmers in such an environment. The programming language used is one of the key tools of a programmer and at some point, the rational investment in tooling is to develop an improved programming language to replace the existing one. While Java appears to have maintained its position as the biggest JVM language [34], the constant increase in the use of its alternatives shows a new path of evolution in the ecosystem.

6.3.4 Future compile targets

While JavaScript has been the ubiquitous language of the web for the last decades, this is not necessarily the case in the future. New APIs and standards are making their way to the browser. The most prominent contender for JavaScript is WebAssembly (WA), a standard for a binary format for executing high-performance programs in browsers.

If WebAssembly reaches major browser support and the format and platform further evolves, it will be possible to author web application code in any language that targets WebAssembly. At the moment the language support for WebAssembly is limited. One of the major limitations is the lack of garbage collection in WA. Mainstream programming languages without garbage collections are limited to languages with manual memory management (C, C++) and languages with reference counting based memory management (Rust, Swift). There is an active proposal for garbage collection in WebAssembly which may in the future result in the availability of GC in WebAssembly[9]. This would greatly increase the number of languages able to target WA.

With the possible implementation of GC in WebAssembly, compiled statically typed languages including Haskell, OCaml and Dart could target web using WebAssembly. Static typing is not, however, the only possible future of web with WA: dynamically typed languages like Python and Ruby can be brought to WebAssembly by implementing the interpreter of the language in WA regardless of the availability of GC – many of the dynamic language interpreters are already implemented in C or C++ thus not requiring GC.

Regardless of the outcome, the future of WebAssembly is not the future of the JS ecosystem. It will however greatly impact the latter by diversifying the programming environment of the web and enabling other languages to compete in a space previously dominated by JavaScript. Languages compiling to JavaScript may, in the end, be a phase that precedes languages with a lower level compile target for the web.

[9]https://github.com/WebAssembly/gc

6.4 Recommendations

So far we have covered the motivation for replacing JavaScript with a statically typed language, a number of languages with their own unique properties for improving the developer productivity and code safety, and considered the viability of different tracks for evolution in the JavaScript ecosystem. This section presents a number of ideas and practices found in the languages researched that we considered useful for future programming languages in the JS ecosystem. Some of the features could alternatively be incorporated in a future version of JavaScript with static types. The recommendations additionally take into account the nature of JavaScript and its current usage.

JavaScript was initially designed as a scripting language and it is still heavily used for the purpose. Consequently, it is expected to enable rapid prototyping with minimal overhead in code. While the requirement does not rule out static types, it does place some limits on the strictness of the typechecker. E.g., Elm, a language highly concerned with correctness, requires decoding for external values which results in overhead in the amount of code required for interacting with auxiliary data sources, including files and network data. The overhead limits the usability of the language in rapid prototyping and scripting.

Enabling potentially unsafe operations in the language is likely required to be able to address all the current use-cases of JavaScript. Additionally, based on the usage of languages discussed earlier, we recommend that the usage of unsafe operations is made as explicit as possible at the syntax level. This is moderately well executed in ReasonML where calls to `bs.raw` indicate unsafe code from JavaScript. In TypeScript the boundary of safe and unsafe logic is not that clear: without the strictest compiler settings, it is easy to implicitly declare values of the dynamic `any` type. Especially the boundary of library code and user-defined code can be unsafely typed if the library code handles values of type `any` at some point. E.g., passing type parameters to library functions often signify unsafe casts which may not be intended by the programmer. Handling unsafe operations explicitly in isolation improves the programmer's ability to properly debug the code and verify the safety of operations that can not be proven by the type system. The goal of a clear boundary between the safe and unsafe may conflict with the property of gradual typing in languages including TypeScript[10].

Subsection 4.1.3 covered the asynchronous non-blocking nature of Java-

[10]However, gradual typing is not the only source of non-isolated unsoundness in TypeScript. E.g., the type assertion operation `T as U` enables asserting the type to both a subtype and supertype creating ambiguity on which assertions are safe.

Script concurrency that heavily influences the programming style of JS. A language targeting JavaScript needs to similarly conform to the model. This can be achieved with plain callback functions (see 4.1.3) or higher-level constructs, including tasks and promises. The JavaScript language, however, has made significant improvements in regards to managing concurrency by introducing the native promise construct and async/await, the latter of which is a major upgrade involving a number of new syntax constructs and evaluation rules. A language with static types would likely need to offer a sufficiently high-level API for managing asynchronous execution to be considered a viable replacement for JavaScript. Thus we recommend that a language targeting JavaScript offers support for async/await-like constructs. Of the languages described in this thesis, TypeScript and Dart support async/await, ReasonML does not but the documentation hints that this may change in the future [36] and Elm offers a different model for asynchronous logic.

The syntax of the languages discussed can be broadly divided into C-style and ML-style syntax. JavaScript, TypeScript and Dart closely follow the C-style with their bracketed blocks with lists of statements and expressions. Elm is syntactically a pure ML language while ReasonML aims to take the ML-style syntax of OCaml closer to C-style languages, namely JavaScript. It can be characterized as a hybrid of the two. The differences between syntax styles in programming languages are mostly aesthetic preferences. Any style of syntax should be convertible to another (e.g. the ReasonML to OCaml conversion). In this thesis, we thus make no involved recommendations for the syntax style but encourage a preference for a more familiar style over an obscure one.

For the features of type system, it is difficult to recommend a single set of properties among the different solutions proposed by the languages described. As described in section 6.3, type inference is a powerful feature that should likely be incorporated in the type system. The degree of inference applied depends on the rest of the features – the more there are advanced features implemented, the more difficult inference becomes. No direct recommendation is presented for the choice between nominal and structural type equivalence. Both strategies have found their use in successful programming languages and a hybrid model may enable gaining the benefits of both as observed in ReasonML.

The requirement for soundness is a fundamental design choice for the type system for a language. Both TypeScript and Dart argue for programming flexibility over absolute soundness and each language has proven to be usable for large-scale projects regardless of the lack of soundness. Functional languages, including the ones discussed in this thesis, ReasonML and Elm, argue for a sound type system with roots in academic type system research.

Both approaches have been effective in software development and neither of the choices necessarily limits the usefulness of the programming language so neither is explicitly recommended over the other. However, for languages with unsound type system features, we recommend making these parts explicit. The documentation of the language should ideally specify all possible sources of unsoundness. Additionally, static analysis tools could be provided for detecting sources of unsoundness in a codebase for debugging purposes.

Chapter 7

Conclusions

In the thesis, we examined programming language development in the JavaScript ecosystem from a type system perspective focusing on statically typed programming languages. These languages offer an alternative set of tools for engineers working on platforms where JS has a dominant position and furthermore provide paths of advancement for the programming ecosystem. Our approach to the subject included three main research questions.

The first research question that sets the premise for understanding the development in the field is *"what is the motivation for creating new statically-typed languages to the JavaScript ecosystem?"* This question is mainly discussed in chapter 4. There are several identified properties of JavaScript that contribute to the need for new programming languages in the ecosystem and that motivate the language features, including type system features, present in the languages discussed in the thesis. Most of these stem from the initial design decisions of JavaScript that have not aged well in combination with the requirement of versionless backwards-compatibility imposed by the web platform. Other requirements arise from the dynamic nature of the JS programming language which is hard to scale to complex programs: the need to support large-scale programs in JavaScript was found to be a significant motivation for the programming languages discussed.

The second research question discussed in chapter 5 is *"what approaches in programming language design – and in particular, type system design – have been proposed to enhance development targeting JavaScript?"* From the type system perspective, a few distinct niches were detected among statically typed JS-compiled languages. These include the object-oriented languages with unsound but flexible type systems containing partial type inference and advanced type system features (Dart, TypeScript) and the functional languages inspired by the ML family of programming languages with their emphasis on functional programming and complete type inference

(Elm, ReasonML). Additionally, different approaches to type equivalence and substitution were identified inside the groups.

The final question, *"if the future of development in the JavaScript ecosystem is statically typed, what will the type system look like?"*, is more open-ended. The approach used is to attempt to obtain a holistic view of the possibilities of static typing in the ecosystem by understanding the obstacles and possibilities of transitioning to a type-safe programming model.

We observed several difficulties in adopting statically typed languages in the ecosystem. These include the overhead of type annotations, the challenges of decoding external data and foreign function interface, the current reliance on highly dynamic behavior and major paradigm and syntax differences between JS and majority of the compiled languages. The languages discussed focus on addressing different challenges related to the developer experience: often there is a trade-off between the properties of correctness and ease of use. New ideas in language design are needed to omit the trade-off and improve one without sacrificing the other. Some powerful concepts, including type inference, are gaining popularity in mainstream programming languages and enabling static typechecking with minimal loss of expressiveness.

There are several viable paths for the JS ecosystem to move towards a productive type-safe future. These include the discussed compiled languages, external tools for static analysis, extending the ECMAScript standard with type annotations and transitioning to a WebAssembly-first ecosystem with bindings to existing JavaScript code. Each of the options comes with their own advantages, disadvantages and limitations. Our last contribution in the thesis is a list of recommendations (section 6.4) for productive and safe statically typed programming in the JS ecosystem based on a number of features and design choices included in the programming languages researched in the thesis.

While the emergence and increased popularity of the languages discussed in this thesis is a promising signal of the possibility of future advancement in the JavaScript ecosystem, it is unlikely that the use of JavaScript sees a major decrease in the near future. In the long term, we, however, find likely that JavaScript will either incorporate static type analysis or be surpassed by a statically typed programming language.

7.1 Validity

This section presents the acknowledged threats to the internal and external validity of the research. Concerning the motivation for new languages in the JS ecosystem, as discussed in RQ1, it is often difficult to outline the

full motivation of the language creators. Three out of the four programming languages were created by a major software company with their own interest in the future of software development. This threatens the validity of parts of the discussion on motivation for new languages in the JS ecosystem. However, the validity of the technical reasoning stands as it can be verified from the source code (and secondarily from documentation). As the technical motivation of language creation is the main focus of the thesis, the threat to validity can be considered minor.

Regarding RQ 2 and 3, some bias is inevitably created by the choice of programming languages for the research. The choices are however justified thoroughly in section 3.3. The major languages of the ecosystem are discussed along with several potential languages of the future. A possible threat is that the development in small niche languages is overstated due to the difficulty to measure programming language popularity. This is partially mitigated by primarily highlighting features that appear in more than one researched language.

In some cases, the designed use of a programming language differs from the actual usage. Ray et. al. [32] excluded TypeScript from their research comparing statically and dynamically typed languages due to the high levels of usage of the dynamic `any` construct observed in TS codebases. This property does not directly threaten the validity of the results in this thesis as it does not attempt to quantify the current use of TypeScript as much as it describes the capabilities of the language and its type system. Gradual typing and dynamic constructs in static type systems are addressed as a feature of the language, not as a property that prevents considering the language statically typed. The bias of using TypeScript as a dynamically typed language is however present in the language popularity metrics used, implying that the amount of usage of TS as a static language is not as high as the metrics display.

7.2 Future research

Several paths for future research can be identified. One interesting approach to assessing the potential of a language is to attempt to rewrite a meaningful piece of JavaScript code in it with minimal changes. A reasonable hypothesis is that some languages – e.g., ReasonML – might be able to replicate existing code with a relatively small amount of syntax differences and no additional type annotations. This process, applied to several codebases, could potentially highlight which aspects of the dynamic language are the easiest and the hardest to map to a statically typed context.

Another prominent topic of study is the formalization of the type systems of modern programming languages – including the ones studied in this thesis. Several mainstream programming languages only provide very informal documentation on the type system properties and assumptions. Bierman et. al. [6] defined the TypeScript programming language rigorously and were able to point out sources of unsoundness among the typing rules. Functional languages with their roots in academia are typically defined formally in the language design process while robust multi-paradigm languages originating from industry may lack such formalization.

The potential of providing type reconstruction for dynamically typed languages via static typing of external values is a practice discussed in the thesis and observed in the wild (e.g., with JavaScript and Visual Studio Code). It has however received little attention in academia and could prove to be an interesting topic of research.

Bibliography

[1] Aiken, A., and Murphy, B. Static type inference in a dynamically typed language. In *POPL* (1991), vol. 91, pp. 279–290.

[2] Ambler, A. L., Burnett, M. M., and Zimmerman, B. A. Operational versus definitional: A perspective on programming paradigms. *Computer 25*, 9 (1992), 28–43.

[3] Amin, N., and Tate, R. Java and Scala's type systems are unsound: The existential crisis of null pointers. *Acm Sigplan Notices 51*, 10 (2016), 838–848.

[4] Anderson, C. *Type inference for Javascript*. PhD thesis, Imperial College London, 2006.

[5] Anderson, C., Giannini, P., and Drossopoulou, S. Towards type inference for JavaScript. In *European conference on Object-oriented programming* (2005), Springer, pp. 428–452.

[6] Bierman, G., Abadi, M., and Torgersen, M. Understanding TypeScript. In *European Conference on Object-Oriented Programming* (2014), Springer, pp. 257–281.

[7] Crockford, D. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.

[8] Czaplicki, E. Elm programming language documentation. Documentation. `https://elm-lang.org/docs` [Accessed 10.10.2019].

[9] Czaplicki, E. *Elm: Concurrent FRP for functional GUIs*. Senior thesis, Harvard University, 2012.

[10] Czaplicki, E., and Chong, S. N. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation-PLDI'13* (2013), ACM Press.

[11] Damas, L. *Type assignment in programming languages*. PhD thesis, The University of Edinburgh, 1985.

[12] Dart team. Dart programming language 5th edition specification draft. Specification, 2019. `https://dart.dev/guides/language/specifications/DartLangSpec-v2.2.pdf` [Accessed 5.11.2019].

[13] Dart team. Dart programming language documentation. Documentation, 2019. `https://dart.dev/guides` [Accessed 10.10.2019].

[14] Dean, J., Grove, D., and Chambers, C. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming* (1995), Springer, pp. 77–101.

[15] ECMA International. ECMAScript language specification. Specification, 2019. `https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf` [Accessed 12.11.2019].

[16] Gao, Z., Bird, C., and Barr, E. T. To type or not to type: Quantifying detectable bugs in JavaScript. In *Proceedings of the 39th International Conference on Software Engineering* (2017), IEEE Press, pp. 758–769.

[17] Hackett, B., and Guo, S.-y. Fast and precise hybrid type inference for JavaScript. *ACM SIGPLAN Notices 47*, 6 (2012), 239–250.

[18] Hindley, R. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society 146* (1969), 29–60.

[19] IEEE 754-2008 Standard for Floating-Point Arithmetic. Standard, IEEE Standards Association, 2008.

[20] Jones, T., Homer, M., and Noble, J. Brand objects for nominal typing. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)* (2015), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[21] Klabnik, S., and Nichols, C. *The Rust Programming Language*. No Starch Press, 2018.

[22] Kleinschmager, S., Robbes, R., Stefik, A., Hanenberg, S., and Tanter, E. Do static type systems improve the maintainability of

software systems? An empirical study. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)* (2012), IEEE, pp. 153–162.

[23] Li, W. H., White, D. R., and Singer, J. JVM-hosted languages: they talk the talk, but do they walk the walk? In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (2013), ACM, pp. 101–112.

[24] Malayeri, D., and Aldrich, J. Integrating nominal and structural subtyping. In *European Conference on Object-Oriented Programming* (2008), Springer, pp. 260–284.

[25] Microsoft. TypeScript langauge documentation. Documentation. `http://www.typescriptlang.org/docs/home.html`, [Accessed 10.10.2019].

[26] Microsoft. Visual Studio Code IDE documentation. Documentation, 2019. `https://code.visualstudio.com/docs` [Accessed 12.11.2019].

[27] Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17*, 3 (1978), 348–375.

[28] Minsky, Y., Madhavapeddy, A., and Hickey, J. *Real World OCaml: Functional programming for the masses*. O'Reilly Media, Inc., 2013.

[29] Petersen, P., Hanenberg, S., and Robbes, R. An empirical comparison of static and dynamic type systems on API usage in the presence of an IDE: Java vs. Groovy with Eclipse. In *Proceedings of the 22nd International Conference on Program Comprehension* (2014), ACM, pp. 212–222.

[30] Pierce, B. C. *Types and programming languages*. MIT press, 2002.

[31] Pradel, M., and Sen, K. The good, the bad, and the ugly: An empirical study of implicit type conversions in JavaScript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)* (2015), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[32] Ray, B., Posnett, D., Filkov, V., and Devanbu, P. A large scale study of programming languages and code quality in GitHub. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), ACM, pp. 155–165.

[33] Severance, C. Javascript: Designing a language in 10 days. *Computer 45*, 2 (2012), 7–8.

[34] Stack Overflow. Stack overflow developer survey. Survey, 2019. `https://insights.stackoverflow.com/survey/2019` [Accessed 10.10.2019].

[35] Tilkov, S., and Vinoski, S. Node. js: Using javascript to build high-performance network programs. *IEEE Internet Computing 14*, 6 (2010), 80–83.

[36] Walke, J. ReasonML programming language documentation. Documentation. `https://reasonml.github.io/docs/en/overview` [Accessed 10.10.2019].

[37] Wand, M. Type inference for record concatenation and multiple inheritance. *Information and Computation 93*, 1 (1991), 1–15.

[38] Wright, A. K., and Felleisen, M. A syntactic approach to type soundness. *Information and computation 115*, 1 (1994), 38–94.