FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Math expression evaluator for literal types in TypeScript
Student:	Bc. Tat Dat Duong
Supervisor:	Ing. Jaroslav Šmolík
Study program:	Informatics
Branch / specialization:	Web Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2023/2024

Instructions

Template literal types [1], introduced in TypeScript 4.1, expand on string literal types for narrowing down a type to a particular string constant, with the ability to expand into many string literal types.

1. Analyze and describe relevant constructs of the TypeScript type system (concatenation, recursive types, conditional types etc.)
2. Implement a typesafe math expression evaluator with a set of basic operations, using a string literal type both as the input and output of the evaluator.
3. Pick appropriate tools for testing type annotations and ensure the validity of the evaluator with functional tests.
4. Discuss the practical uses of implemented meta types and theoretical and practical shortcomings of the TypeScript type system.
5. Publish the implementation as an open-source TypeScript library, which can be used for meta-programming, including source code and corresponding documentation.

[1] https://www.typescriptlang.org/docs/handbook/2/template-literal-types.html

Electronically approved by Ing. Jaroslav Kuchař, Ph.D. on 6 January 2023 in Prague.

Master's thesis

MATH EXPRESSION EVALUATOR FOR LITERAL TYPES IN TYPESCRIPT

Bc. Tat Dat Duong

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Jaroslav Šmolík
March 26, 2023

Contents

List of Figures

List of Tables

List of Listings

Chtěl bych poděkovat především sit amet, consectetuer adipiscing elit. Curabitur sagittis hendrerit ante. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Cras pede libero, dapibus nec, pretium sit amet, tempor quis. Sed vel lectus. Donec odio tempus molestie, porttitor ut, iaculis quis, sem. Suspendisse sagittis ultrices augue.

Declaration

FILL IN ACCORDING TO THE INSTRUCTIONS. VYPLŇTE V SOULADU S POKYNY. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Curabitur sagittis hendrerit ante. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Cras pede libero, dapibus nec, pretium sit amet, tempor quis. Sed vel lectus. Donec odio tempus molestie, porttitor ut, iaculis quis, sem. Suspendisse sagittis ultrices augue. Donec ipsum massa, ullamcorper in, auctor et, scelerisque sed, est. In sem justo, commodo ut, suscipit at, pharetra vitae, orci. Pellentesque pretium lectus id turpis.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Curabitur sagittis hendrerit ante. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Cras pede libero, dapibus nec, pretium sit amet, tempor quis. Sed vel lectus. Donec odio tempus molestie, porttitor ut, iaculis quis, sem. Suspendisse sagittis ultrices augue. Donec ipsum massa, ullamcorper in, auctor et, scelerisque sed, est. In sem justo, commodo ut, suscipit at, pharetra vitae, orci. Pellentesque pretium lectus id turpis.

In Prague on March 26, 2023 ....

Abstract

Fill in abstract of this thesis in English language. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Cras pede libero, dapibus nec, pretium sit amet, tempor quis. Sed vel lectus. Donec odio tempus molestie, porttitor ut, iaculis quis, sem. Suspendisse sagittis ultrices augue.

Keywords enter, commma, separated, list, of, keywords, in, ENGLISH

Abstrakt

Fill in abstract of this thesis in Czech language. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Cras pede libero, dapibus nec, pretium sit amet, tempor quis. Sed vel lectus. Donec odio tempus molestie, porttitor ut, iaculis quis, sem. Suspendisse sagittis ultrices augue.

Klíčová slova enter, commma, separated, list, of, keywords, in, CZECH

Summary

Summary section

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem.

Summary section

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa.

Summary section

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

Summary section

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

Summary section

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Lorem lorem lorem.

Seznam zkratek

TC39	ECMA International, Technical Committee 39
W3C	World Wide Web Consortium

Chapter 1

Introduction

1.1 Motivation

TypeScript is a hot topic in the web development ecosystem and type-safety is eating the world [1]. As of 2023, majority of developers are using TypeScript most of the time, either avoiding JavaScript entirely or spending majority time working with TypeScript codebases [2]. Over the years, TypeScript has transformed from a basic type annotation tool to a full fledged programming language within the type system itself. Libraries such as Prisma for database type-safety [3], Zod for combining schema validation and static type inference [4] and tRPC for API end-to-end type-safety across boundaries [5] utilise the power of advanced TypeScript types to provide a better experience for developers. With smart suggestions being available right in the editor of choice, TypeScript ensures high quality of code while avoiding any runtime costs due to the type system being evaluated during compilation. With editors and IDEs using a language server powered by Language Server Protocol (LSP) to provide the developer with the smart suggestions, there is an incentive to utilise the type system instead of running a daemon alongside or adding an additional build step.

However, TypeScript is only as powerful as the types that you give to it. A great burden is laid to the maintainers of libraries to provide descriptive and useful types. The goal of this thesis is to laid out and highlight the capabilities of the TypeScript type system, discussing the constraints and limitations found in TypeScript.

1.2 What is static type system

But what actually is a type system? For years, type systems in programming languages have been a well-known and heavily discussed topic. The main goal of a type system is to provide a formal specification of the types of data that can be manipulated by a program.

In statically typed languages, a data type of a variable is known at compile time. The compiler uses the additional information about data types to verify the source code during compilation. The data type itself can be deduced from the usage in the code (type inferrence) or a programmer explicitly specifies the data type of a variable before usage. Example of such languages using static typing are for instance Java, C#, C++, etc.

Whereas in dynamically typed languages, the type of a variable is determined at runtime based on the value being assigned. Developers do not need to explicitly declare the type of a variable. Some of the popular dynamically typed languages include Python, Ruby, PHP and most notably JavaScript, which is widely used to create interactive and dynamic user interfaces on the web platform. Dynamically typed languages tend to be more flexible and allow developers, notably beginner developers, to write code faster and iterate quicker.

Static typing offers numerous compelling benefits, that can enhance the development process. First, a large class of errors are caught earlier in the development process. This reduces the likelihood of bugs and runtime issues that can be difficult to diagnose and debug.

With static typing, developers can rely on a compiler system to ensure that their code conforms to the expected data types. Developers can also refactor existing typed code with more confidence, as the system is giving developers direct feedback when refactoring.

Furthermore, by writing types developers are actively self-documenting the code, making the code more readable and easier to understand, especially when dealing wih previously unseen code. And even though developers might need to write more code to specify the types for the variable, the type system is able to determine the intent of the developer without writing additional code.

1.3 Strucute of the work

This thesis will provide a compherensive analysis of relevant advanced constructs found in the TypeScript type system, and how they can be used to allow robust meta-programming within the types itself. To demonstrate the capabilities of the type system and the usage of the constructs itself, we provide an implementation of a generic math expression evaluator library that operates strictly on the type level. We discuss how the library can be tested and the output validated and we evaluate the performance of implemented operations against other existing type level math libraries and the impact on which the library has on type checking and developer experience in the editor.

Chapter 2

Analysis

2.1 Static Typing in JavaScript

JavaScript is a dynamically typed programming language, where users do not need to assign types to a variable or a function and the type is inferred automatically by the JavaScript engine. This is a great feature of JavaScript, which lowers the barrier of entry to writing JavaScript code and allows developers to prototype and write code quickly, proven by the growth of popularity of JavaScript in the last decade, making it the most commonly used programming language according to the 2022 Stack Overflow Developer Survey [6].

However, dynamic typing has its drawbacks, as it is harder to spot trivial errors in the code without running it beforehand and it is more difficult to refactor the code without breaking it, which often leads to poor software quality [7]. Proponents of static typing insist that static types allow developers to spot potential bugs and mistakes earlier during development and that it allows for better tooling, such as more rich code completion and refactoring tools.

There is an upcoming TC39 proposal for adding type annotations, broadly inspired by TypeScript syntax [8]. These annotations are only useful for build-time tooling as they are ignored in runtime. The proposal suggests that these annotations should be erased by an additional compilation step. Even though users can already provide static types using JSDoc right now, the syntax is not as clean as the proposed TypeScript-like syntax.

Regardless, many languages aim to introduce static typing to JavaScript, such as Flow or TypeScript, or alternative languages which compile back to JavaScript, such as Elm or ReScript.

2.1.1 Elm

Elm is a functional programming language designed specifically for building web applications [9]. The language compiles to JavaScript and has a strong static Hindley-Milner-based type system, which allows inferring types more often and reliably. Elm does not provide any escape hatches such as `any` in TypeScript, thus it is harder to write unsafe code, as the types must be valid for the code to be compiled.

Elm also includes a lot of quality-of-life improvements and benefits, for instance: enforced purity of functions, out-of-the-box immutability, `case` pattern matching, JSON decoders and encoders for strict parsing, `Maybe` and `Result` monads for avoiding `null` and `undefined` references or its own virtual DOM implementation for efficient rendering of interactive user interfaces. Notably, the Elm Architecture, where the application code is organized into three parts: model, update and view [10], has greatly inspired other libraries and frameworks like Redux [11].

2.1.2 ReScript

ReScript is a programming language built on top of the OCaml toolchain. Unlike Flow or TypeScript, ReScript is not a superset of JavaScript, instead, the language compiles to JavaScript. ReScript was created as a spin-off from the Reason programming language and accompanying BuckleScript compiler, aiming to vertically integrate and streamline the adoption barrier caused by the need to be familiar with multiple unrelated tools and toolchains [12].

The language aims to be more sound with more powerful type inference than TypeScript, borrowing the Hindler-Milner type system from OCaml implementation [13, 14], thus most of time the types can be inferred automatically without the need to annotate them explicitly, whereas TypeScript utilizes bidirectional type checking [15].

2.1.3 Flow

Flow is a static type checker for JavaScript [16, 17], which allows developers to annotate their code with static types. Flow is developed by Meta and is internally used in production by Facebook, Instagram and React Native. Type annotations in Flow are fully eraseable, which means that the type annotations can be fully removed from the Flow code to emit valid JavaScript code. The checking of these types is occurring at compile-time before removal in build-time. Flow is also a superset of JavaScript, which means any JavaScript code is a valid Flow code.

One of the primary goals of Flow is to provide type soundness; the ability to catch every error that might happen in runtime at compile-time, no matter how likely it is to happen. This means, that a valid Flow code can provide developers some guarantees about the type a value has in runtime, at the expense of catching errors, which are unlikely to happen in runtime.

Both Flow and TypeScript are similar regarding features at the time of writing. Most of the soundness differences between Flow and TypeScript have been addressed with the newer versions of TypeScript, even though soundness is a specific non-goal by the TypeScript team [18]. However, developers must opt-in to these features by setting `"strict"` to `"true"` in `tsconfig.json`, whereas in Flow these features are enabled by default.

2.1.4 TypeScript

TypeScript is a statically typed programming language developed and maintained by Microsoft [19]. It is a language that compiles down to JavaScript and adds static type checking to JavaScript [20]. Unlike Elm or ReScript, TypeScript is a syntactical superset of JavaScript, which means that any valid JavaScript code can be a valid TypeScript code[1]. Similar to Flow, type annotations provided by the developer are fully erasable either by the TypeScript `tsc` compiler or by other community build tools, such as `babel`[21], `esbuild`[22] or `swc`[23].

Type system in TypeScript is considered to be less sound and more forgiving, as soundness is stated as an explicit non-goal for the design team of TypeScript [18], with emphasis on striking a balance between productivity and correctness. By default, the TypeScript compiler is not strict and the language itself includes an escape hatch for developers to opt out of type checking by using the `any` type or using `@ts-ignore` comment annotations. Nevertheless, with proper compiler configuration, the type system of TypeScript can be as sound as in Flow.

Both Flow and TypeScript support advanced features such as generics and utility types, with the latter supporting template string literal types and better support for conditional types, unlocking the potential of writing more expressive types, which this master thesis will further explore in more detail.

TypeScript has become the de-facto standard for writing JavaScript code with static types. With deep integration with Visual Studio Code [24], the rich build ecosystem and high compatibility with existing JavaScript libraries and tools, TypeScript has become one of the fastest growing languages in terms of usage according to the 2022 Octoverse report by Github [25].

2.2 Typescript syntax

Typescript adds optional static types to JavaScript, which can be applied to add constraints to code constructs such as functions, variables and properties, so that compilers, linters and other development tools can provide better and useful insights during software development [26].

Types in TypeScript can be categorized into primitive types, literal types, data structure types, union types, intersection types and type parameters. All of these types introduce static constraints on their values.

A primitive value is data, that is not an object and has no methods or properties. These primitives are immutable, thus they cannot be altered. Similar to JavaScript, TypeScript has the following types of primitive values:

Listing 2.1 Primitive Types

```
type Primitive =
  | string | number | bigint
  | boolean | undefined | symbol | null;
```

Some primitive values represent a singular data value, such as `null` or `undefined`, but many of these primitives can represent an infinite number of values, like `number`, `bignumber` or `string`.

Literal types are a subset of primitive values, which are used to describe an exact possible value.

To represent data structures such as objects and arrays, we can use the following types: objects, records, tuples and arrays.

[1]With a lax compiler configuration

Listing 2.2 Literal Types

```
type Literal = "foo" | 42 | true | 100n;
```

Listing 2.3 Data structures

```
type Structures =
  | { foo: string, bar: number }      // object
  | { [key in keyof Keys]: number }   // record
  | [number, string]                  // tuple
  | number[]                          // array
```

TypeScript uses structured typing, which means that TypeScript only validates the shape of the data. Essentially, if the data has the same shape as the type, it is considered to be of that type. This is also known as duck typing, essentially if it walks like a duck and quacks like a duck, it is a duck.

Listing 2.4 Duck typing

```
type DuckLike = { quack: () => void; type: string };

const Duck: DuckLike = {
  quack: () => console.log("duck!"),
  type: "duck",
};

// this will be still valid
const Goose: DuckLike = {
  quack: () => console.log("goose!"),
  type: "goose",
};
```

Structured typing does include some drawbacks unlike in nominal type systems, where each type is unique and the same data cannot be assigned across types, but these can be easily mitigated using literal types to act as brands.

Types can be generalized into sets, where each type can contain a set of values. A type can be a subset or a superset of another type.

Similar to other sets in mathematics, types can be combined using unions. With unions, we can broaden the scope of the type to represent multiple values.

Types can also intersect each other to create a narrower type.

TODO: type intersection

TypeScript includes two top types, also known as universal types or supertypes: `any` and `unknown`. These types are used to represent any possible value.

`any` is a top type, where every type is assignable to type `any` and type `any` is assignable to every type [27]. `any` is acting as an escape hatch to opt out of type checking. This does have unintended consequences, especially when dealing with external data, most notably when the return type of `JSON.parse()` and `Response.json()` is `any`. As `any` is assignable to every type,

Listing 2.5 Nominal typing in TS

```
type DuckLike = { quack: () => void; type: "duck" };

const Duck: DuckLike = {
  quack: () => console.log("duck!"),
  type: "duck",
};

// this will not be valid
const Goose: DuckLike = {
  quack: () => console.log("goose!"),
  type: "goose",
};
```

it can be assigned to a different type without any warnings.

TODO: Add any code example

`unknown` acts as a more restrictive version of `any`. Every type is assignable to type `unknown`, but `unknown` is not assignable to any other type. To assign `unknown` to a different type, we must narrow the types using either type guards, type assertions, equality checks or other assertion functions.

TODO: Add unknown code example

`never` is a bottom type, acting as a subtype of all other types, representing a value that should never occur. In the context of the theory of mathematical logic, `never` acts as a logical contradiction, describing a value that may never exist, which is especially useful to model stop gaps within type-level functions. No other type can be assigned to `never` nor `never` cannot be assigned to any other type.

TODO: Add unknown code example

`enum` type is a distinct subtype used to describe a set of named constants. Instead of using individual variables for each constant, an `enum` provides an organized way to express a collection of related values. `enum` is one of the few TypeScript features which introduce an additional code added to the compiler output and enums refer to real objects at runtime.

An `enum` type consists of members and their corresponding initializers for the runtime value of the member. There are two types of enums in TypeScript: numeric enums and string-based enums:

Listing 2.6 Numeric enums

```
enum Direction {
  Up = 1,
  Down,
  Left,
  Right,
}
```

In numeric enums, each member is assigned a numeric literal value. Each member can have an optional initializer to specify an exact number corresponding to a member. If omitted, the

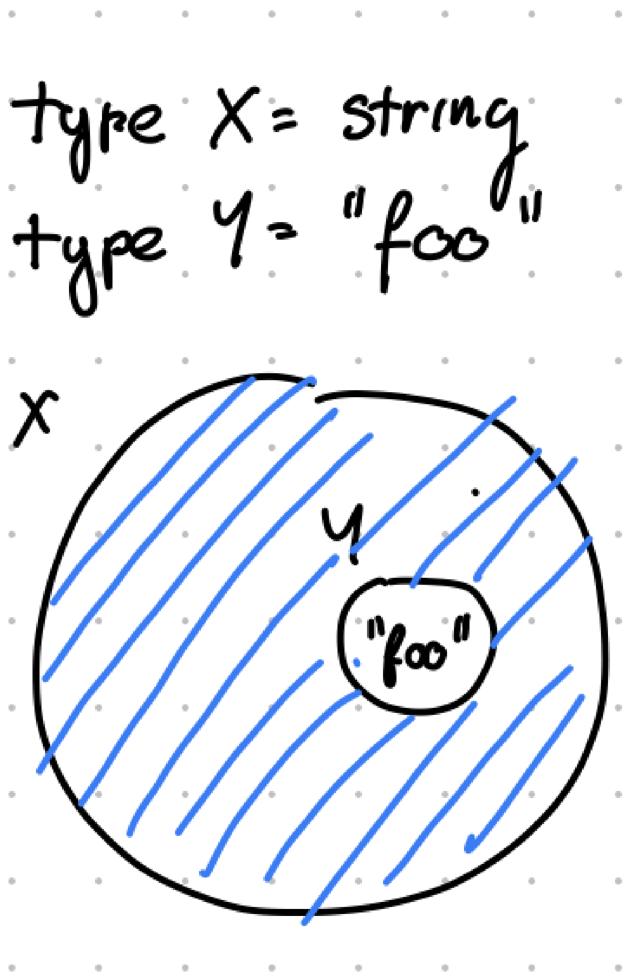

Figure 2.1 Types represented as sets

value of the member will be generated by auto-incrementing from previous members.

String-based enums are similar in nature, where each member is assigned a string literal value instead. Each member thus must have an initializer with a string literal. The key benefit of string-based enums is that they tend to keep their semantic value well when serializing, which is especially helpful when debugging, as the values of numeric enums tend to be opaque.

Functions and `void`

Void TODO: Enums

Arrays

`any`, `never`, `unknown`

Object

Generics

- Terminology - generics, type arguments, return type
- Conditional types

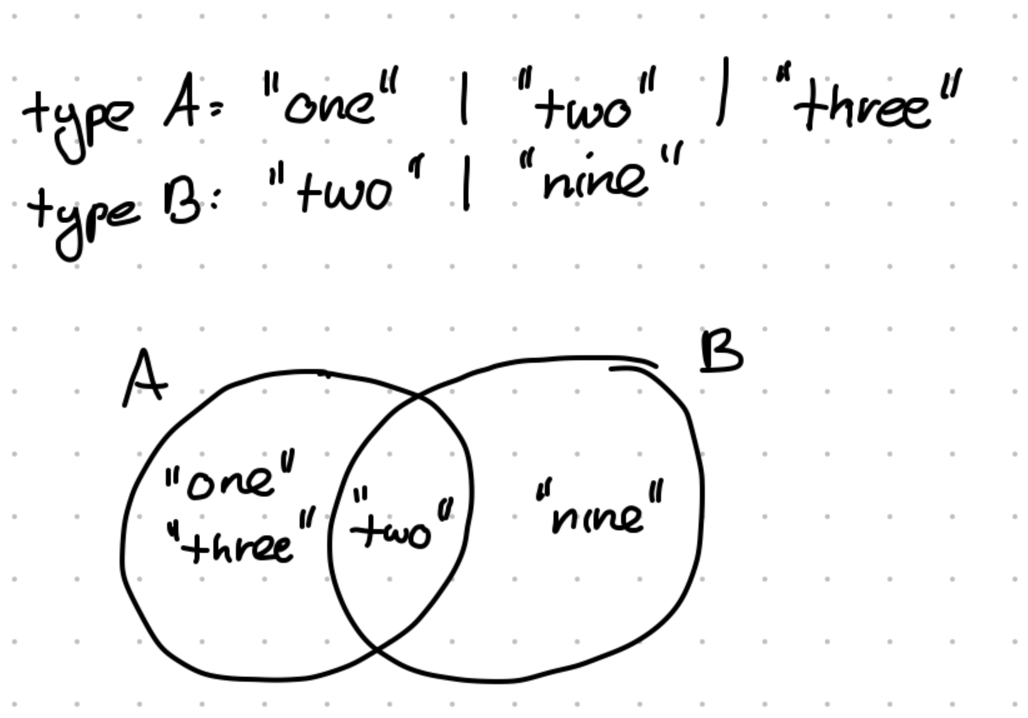

Figure 2.2 Union of types

Listing 2.7 Numeric enums

```
enum Direction {
  Up = "UP",
  Down = "DOWN",
  Left = "LEFT",
  Right = "RIGHT",
}
```

- Recursive types
- Mapped types
- Template Literal Types

2.2.1 Types and their assignability

- Primitive Types
- Literal Types
- `unknown`, `never`, `any`
- Structures (nominal vs structural typing)
- Unions and Intersections

2.2.2 Conditional Types

2.2.3 Conditional Types

2.2.4 Recursive Types

2.2.5 Mapped Types

2.2.6 Template Literal Types

2.3 Prior Art

- `kawayiLinLin/typescript-lodash`
- `arielhs/ts-arithmetic`
- `ts-belt`
- `type-fest`

Chapter 3

Implementation

Chapter 4

Testing

- Unit tests, integration tests (`eslint`, `eslint-plugin-expect-type`)
- Github Actions
- Performance Testing (performance tracing, extended diagnostics)
- Comparison between existing TS math libraries

Chapter 5

Conclusion

5.1 Advantages and disadvantes of TS

5.2 Future work

Appendix A

Nějaká příloha

Sem přijde to, co nepatří do hlavní části.

Bibliography

1. JSWORLD CONFERENCE (director). *Fred K. Schott - Type-safety Is Eating the World* [online]. 2023. [visited on 2023-03-25]. Available from: `https://www.youtube.com/watch?v=DqYxbjTM2vw`.
2. *The State of JS 2022: Usage* [online]. [visited on 2023-03-25]. Available from: `https://2022.stateofjs.com/en-US/usage/`.
3. *Prisma/Prisma: Next-generation ORM for Node.Js & TypeScript — PostgreSQL, MySQL, MariaDB, SQL Server, SQLite, MongoDB and CockroachDB* [online]. [visited on 2023-03-25]. Available from: `https://github.com/prisma/prisma`.
4. MCDONNELL, Colin. *Zod* [online]. 2023. [visited on 2023-03-25]. Available from: `https://github.com/colinhacks/zod`.
5. *tRPC* [online]. tRPC, 2023. [visited on 2023-03-25]. Available from: `https://github.com/trpc/trpc`.
6. *Stack Overflow Developer Survey 2022* [online]. Stack Overflow. [visited on 2023-01-29]. Available from: `https://survey.stackoverflow.co/2022/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2022`.
7. FARD, Amin Milani; MESBAH, Ali. JSNOSE: Detecting JavaScript Code Smells. *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)* [online]. 2013, pp. 116–125 [visited on 2023-03-25]. ISBN 9781467357395. Available from DOI: `10.1109/SCAM.2013.6648192`.
8. *ECMAScript Proposal: Type Annotations* [online]. Ecma TC39, 2023. [visited on 2023-01-29]. Available from: `https://github.com/tc39/proposal-type-annotations`.
9. *Elm - Delightful Language for Reliable Web Applications* [online]. [visited on 2023-01-31]. Available from: `https://elm-lang.org/`.
10. *The Elm Architecture · An Introduction to Elm* [online]. [visited on 2023-01-31]. Available from: `https://guide.elm-lang.org/architecture/`.
11. *Prior Art — Redux* [online]. 2022-02-07. [visited on 2023-01-31]. Available from: `https://redux.js.org/understanding/history-and-design/prior-art`.

12. *BuckleScript & Reason Rebranding* [online]. ReScript Blog. [visited on 2023-01-29]. Available from: `https://rescript-lang.org/blog/bucklescript-is-rebranding`.
13. *Efficient and Insightful Generalization* [online]. [visited on 2023-02-12]. Available from: `https://okmij.org/ftp/ML/generalization.html`.
14. *History — ReScript* [online]. 2022-02-12. [visited on 2023-02-12]. Available from: `https://github.com/rescript-lang/rescript-compiler/blob/master/CREDITS.md`.
15. *Reconstructing TypeScript, Part 0: Intro and Background* [online]. [visited on 2023-01-24]. Available from: `https://jaked.org/blog/2021-09-07-Reconstructing-TypeScript-part-0`.
16. CHAUDHURI, Avik; VEKRIS, Panagiotis; GOLDMAN, Sam; ROCH, Marshall; LEVI, Gabriel. Fast and Precise Type Checking for JavaScript. *Proceedings of the ACM on Programming Languages* [online]. 2017, vol. 1, 48:1–48:30 [visited on 2023-01-29]. Available from DOI: `10.1145/3133872`.
17. *Flow* [online]. Meta, 2023. [visited on 2023-01-29]. Available from: `https://github.com/facebook/flow`.
18. *TypeScript Design Goals* [online]. GitHub. [visited on 2023-01-29]. Available from: `https://github.com/microsoft/TypeScript/wiki/TypeScript-Design-Goals`.
19. *TypeScript: JavaScript With Syntax For Types* [online]. [visited on 2023-02-01]. Available from: `https://www.typescriptlang.org/`.
20. *Documentation - TypeScript for JavaScript Programmers* [online]. [visited on 2023-01-14]. Available from: `https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html`.
21. *Babel · The Compiler for next Generation JavaScript* [online]. [visited on 2023-02-01]. Available from: `https://babeljs.io/`.
22. *Esbuild - An Extremely Fast Bundler for the Web* [online]. [visited on 2023-02-01]. Available from: `https://esbuild.github.io/`.
23. *SWC - Rust-based Platform for the Web* [online]. [visited on 2023-02-01]. Available from: `https://swc.rs/`.
24. *Visual Studio Code - Code Editing. Redefined* [online]. [visited on 2023-02-01]. Available from: `https://code.visualstudio.com/`.
25. *Octoverse 2022: The State of Open Source* [online]. The State of the Octoverse. [visited on 2023-01-29]. Available from: `https://octoverse.github.com/`.
26. HEJLSBERG, Anders; LUCCO, Steve. TypeScript Language Specification. [N.d.].
27. *The Top Types ‘any‘ and ‘unknown‘ in TypeScript* [online]. [visited on 2023-03-26]. Available from: `https://2ality.com/2020/06/any-unknown-typescript.html`.

Obsah přiloženého média

```
readme.txt ........................ stručný popis obsahu média
exe ............................... adresář se spustitelnou formou implementace
src
  impl ............................ zdrojové kódy implementace
  thesis .......................... zdrojová forma práce ve formátu LaTeX
text .............................. text práce
  thesis.pdf ...................... text práce ve formátu PDF
```