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
April 10, 2023

Czech Technical University in Prague
Faculty of Information Technology
© 2023 Bc. Tat Dat Duong. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Duong Tat Dat. *Math expression evaluator for literal types in TypeScript.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 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 April 10, 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, a typed superset of JavaScript, is quickly gaining popularity in the JavaScript development ecosystem, and type-safety, the concept of validating data types, is "eating the world"[1]. As of 2023, over 66% of developers are using TypeScript most of the time, either avoiding JavaScript entirely or spending the majority of time working with TypeScript codebases [2]. Over the years, TypeScript has transformed from a simple type annotation tool to a full-fledged programming language within the type system itself. Multiple libraries with advanced TypeScript types have emerged to improve the developer experience. Libraries such as Prisma for database type-safety [3], Zod for combining schema validation and static type inference [4], or tRPC for API end-to-end type-safety across boundaries [5]. With intelligent suggestions in the editor of choice, TypeScript ensures high code quality 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 valuable suggestions, there is an incentive to utilise the type system instead of running a daemon alongside or adding a build step.

However, TypeScript is only as powerful as the types declared and received. A significant burden is laid on the maintainers of libraries to provide descriptive and valuable types. This thesis aims to lay out and highlight the capabilities and techniques of the TypeScript type system in a challenging type-only implementation of math expression evaluator, discussing the constraints and limitations found in TypeScript and possible workarounds useful for library maintainers.

1.2 What is a static 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 a program can manipulate.

In statically typed languages, the 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 inference), or a programmer explicitly specifies the data type of a variable before usage. Examples of such languages using static typing are, for instance, Java, C# or C++.

Whereas in dynamically typed languages, the type of a variable is determined at runtime based on the value being assigned, it does not need to be explicitly declared by the developer or known at compile time via type inference. 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 is 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 the code conforms to the expected data types. Developers can also refactor existing typed code more confidently, as the system gives developers direct feedback when refactoring.

Furthermore, by writing type annotations, developers are actively self-documenting the code, making the code more readable and easier to understand, especially when dealing with unfamiliar code. Finally, even though an initial commitment is necessary by writing type annotations at first, a more powerful type system can determine the developer's intent without writing additional code as the development progresses.

1.3 Structure of the work

This thesis will provide a comprehensive 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 themselves. To demonstrate the capabilities of the type system, an implementation of a generic math expression evaluator library that operates strictly on the type level is provided, followed by a discussion on testing and performance of the library and the impact on type checking and development 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 the 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 erasable, 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 generates 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` type checker 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 type checker 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 type checker 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 Usage of TypeScript

The TypeScript project is made of two major parts available to developers:

- `tsc`: the TypeScript Compiler, which is responsible for both type checking and outputting valid JavaScript files,
- `tsserver`: the TypeScript Standalone Server, which encapsulates the TypeScript Compiler and language services for use in editors and IDEs [26].

Whereas a type-checker is most likely executed manually more often and is the entry point for developers when using TypeScript, the language server is equally as useful, as it communicates with the editor via Language Server Protocol (LSP) to provide important language services. These include code completion, auto-importing, symbol renaming etc.

Unlike in the other languages, the compilation step itself is understood to only mean the type erasure itself. Even though the source code itself can have various type errors, `tsc` will still by default emit JavaScript files, as long as the input source file can be parsed by both the scanner and the parser. This allows developers to progressively update their code and iterate quickly on the functionality without having to deal with the type errors immediately, essentially acting more as a linter than a compiler. Regardless, in this thesis, “compiling” and “type-checking” will be used interchangeably.

[1] With a lax type checker configuration

2.3 Typescript syntax

In TypeScript, types are annotated using `:[type annotation]` syntax, adding annotations to any of the symbols found in JavaScript, such as variables, function parameters and function return values, to add constraints to values. Type annotations in TypeScript can be categorized into primitive types, literal types, data structure types, union types, intersection types and type parameters. In the following sections, we will explore each of these types in more detail. The following listing 2.1 shows a basic example of TypeScript annotations:

Listing 2.1 Basic TypeScript annotation example

```
const prefix: string = "Hello world"
const user: {
  name: string;
  age: number
}

function formatUserGreeting(
  user: {
    name: string;
    age: number;
  },
  message: string
): string {
  return [message, user.name].join(" ");
}

const greeting: string = formatUserGreeting(user, prefix);
```

At runtime, every variable has a single concrete value, but in TypeScript, the variable has only a type. A useful mental model for understanding types is to think of the type as a set of permitted values [27], effectively describing the domain of the type.

Developers can declare types directly in type annotations, but sometimes developers need to reuse the same type in multiple annotations. To avoid repeating the same declaration, we can use type aliases to refer to a type by a name. These type variables act as an alias, which can be used in place of the type itself. The listing 2.2 shows a refactored `formatUserGreeting` function of the previous listing 2.1 using type aliases.

2.3.1 Primitive Types

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. The TypeScript type system provides a comprehensive representation of these primitives, as seen in listing 2.3:

Some primitive values represent a singular data value, such as `null` or `undefined`, but many of these primitives can represent multiple values (`boolean` can represent either `true` or `false`), or even an infinite amount of values, like `number`, `bignumber` or `string`.

Listing 2.2 Type aliases

```
type User = {
  name: string;
  age: number
}

const prefix: string = "Hello world"
const user: User

function formatUserGreeting(
  user: User,
  message: string
): string {
  return [message, user.name].join(" ");
}
```

Listing 2.3 Primitive Types

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

2.3.2 Literal Types

To describe an exact possible value, we can use literal types. From the point of view of the type system, a literal type is a subset of one of the following primitive types: `string`, `number`, `bignumber` or `boolean`,[2] as seen in Listing 2.4.

Listing 2.4 Literal Types

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

// Valid code
const Valid: Literal = "foo"

// @ts-expect-error Type '"bar"' is not assignable to type 'Literal'
const Invalid: Literal = "bar"
```

2.3.3 Types for data structures

TypeScript also allows annotating data structures such as objects and arrays with four possible types, depending on the enumerability of items and their types. The syntax overview can be seen here in Listing 2.5.

- `tuple` type for describing an array with a fixed number of elements, possibly with a different type for each element,

[2] Both `null` and `undefined` are literal types as well

- `array` type for describing an array with an unknown length and the values are of the same type,
- `record` type for describing an object with an unknown number of keys and the values are of the same type,
- `object` type or an `interface` for describing an object with a finite set of keys with values of different types per key.

Listing 2.5 Data structures

```
type ObjectStructure =
  | { foo: string, bar: number }

type RecordStructure
  | { [key: string]: number }
  | Record<string, number>

type TupleStructure = [number, string]

type ArrayStructure = number[]
```

As we can see, two type syntaxes can be used for describing objects with a finite set of key-value pairs in TypeScript: `object` and `interface`. There are some key differences between the two syntaxes:

1. The `object` type uses the type alias syntax, whereas an interface is defined using a special `interface` keyword.
2. TypeScript allows multiple declarations of `interface`, which are later merged during type-checking. This can be especially useful when working with third-party libraries.
3. Even though both support object merging, `interface` can be implemented by classes, ensuring that the class adheres to the structure defined by the interface. `object` types cannot be directly implemented by a class.
4. Merging `interface` declarations is more performant when merging multiple declarations than an intersection of `object` types [28].

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, as seen in Listing 2.6. This is also known as duck typing, essentially: “If it walks like a duck and quacks like a duck, it is a duck.”

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, as seen in Listing 2.7.

Listing 2.6 Structured 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",
};
```

Listing 2.7 Nominal typing in TypeScript

```
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",
};
```

2.3.4 Union and intersection types

Revisiting the notion of types as sets of values, as seen in Listing 2.4, when attempting to assign a value not permitted by the `Literal` type, a type error occurs. In the world of TypeScript, a type is "assignable", if it is either a "member of" the set of permitted values defined by the type (when describing relationships between a value and a type) or it is a "subset of" the sets (when describing relationships between two types).

Sometimes, we need to describe a type, which is a combination of multiple types, combining two sets of values into a single set. This is achievable by using the union operator represented by the `|` symbol to describe a type that represents a value, which may be any of one of the combined types referred to as "union members" [29]. Essentially, `X | Y` can be read as a type for a value that can either be of type `X` or `Y`.

Because behind a union type may be a value of any of the union member types, TypeScript will allow only operations, which are valid for every union member. If we want to perform an operation which valid for some of the union members, we must perform type narrowing, which refines a broader type to a more specific narrow one, capturing a subset of values of the original broader type.

An example can be seen in Listing 2.8, where the function `printUserId` can accept both a `string` or a `number` as an argument. To invoke `toUpperCase()`, a method valid only for values of `string` type, we must perform a check, if the parameter is a `string`. Afterward, TypeScript is smart enough to infer that the type of the checked value must be necessary a `string` and permits the invocation of `toUpperCase()`.

Listing 2.8 Union types with simple narrowing

```
function printUserId(id: string | number) {
  if (typeof id === "string") {
    return id.toUpperCase()
  } else {
    return id
  }
}
```

Whereas an intersection of types can be represented by the `&` operator. Similarly to the union type, `X & Y` can be read as a type for a value that can simultaneously belong to type `X` and `Y`. These intersection types are of particular interest when working with object types, as an intersection of two object types has all properties of both object types, as an object with both of the properties can be assigned to both of the intersection member types. For this particular reason, intersection types are commonly used to merge two object types, as seen in 2.9.[3]

Listing 2.9 Intersection types

```
type Intersection = { a: string } & { b: number }
const item: Intersection = { a: "a", b: 1 }
```

2.3.5 `keyof` type and indexed access types

To access a specific property of a record or a tuple, we use the indexed access type. The syntax of indexed access types mirrors the syntax for accessing an object in JavaScript, as seen in Listing 2.10. We can also use unions as keys to get types of multiple properties of an object type.

Listing 2.10 Indexed access types

```
type User = { firstName: string; lastName: string; age: number }

type Age = User["age"]
type Names = User["firstName" | "lastName"]
```

What if we need to get all possible keys of an object type? For that, we can use the `keyof` keyword operator. This will return an union of all keys of the provided data structure type. These are especially useful when working with mapped types later on. An example can be seen in Listing 2.11.

TODO: indexed access type

[3] We can also use `extends` keyword to merge two interfaces

Listing 2.11 Usage of `keyof`

```
type User = { firstName: string; lastName: string; age: number }
type Keys = keyof User
//   ^? "firstName" | "lastName" | "age"
```

2.3.6 Special data types

When working with unions and intersections, we need to be able to describe a type, which can describe a union of all possible types or a type, which is created by intersecting two types with no related properties. We refer to these types as universal supertypes and universal subtypes respectively. Universal supertypes, also known as top types, are types that are a superset of all other types and are used to represent any possible value. Whereas universal subtypes, also known as bottom types, are types that are a subset of all other types and are often used to describe a type that has no permitted values.

TypeScript includes two top universal supertypes: `any` and `unknown`. In the case of `any`, every type is assignable to type `any` and type `any` is assignable to every type [30]. `any` is acting as an escape hatch to opt out of type checking. This does have unintended consequences, as `any` is assignable to every type, it can be assigned to a different type without any warnings. This is especially problematic when dealing with external data as the return type of `JSON.parse()` is `any`. An example of assignability can be seen at Listing 2.12.

Listing 2.12 Assignability of any

```
let data: any = JSON.parse("...")

// all of these are valid TypeScript code
data = null
data = true
data = {}

// still valid code, opting out of type checking
const a: null = data
const b: boolean = data
const c: object = data
```

`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, which can be seen at Listing 2.13. To assign `unknown` to a different type, we must narrow the types using either type guards, type assertions, equality checks or other assertion functions.

Finally, `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. No other type can be assigned to `never` nor `never` cannot be assigned to any other type. `never` can be found when attempting to intersect two types that have no properties in common, such as `string & number`.

`void` is a specific type used to signify a function, which does not return a value. There is

Listing 2.13 Assignability of unknown

```
let data: unknown = JSON.parse("...")

// all of these are valid TypeScript code
data = null
data = true
data = {}

// not valid, as unknown is not assignable to any other type
const a: null = data
const b: boolean = data
const c: object = data
```

a notable difference between the usage of `void` when used in context, describing a type for a function with `void` return type, and when used in the function declaration, as seen in Listing 2.14. The former is used to describe a situation when an implementation of a "void function" does return a value but should be ignored. The latter does enforce that a function should not return a value at all.

Listing 2.14 Return type void

```
type voidFn = () => void

// Valid code
const fn1: voidFn = () => true

function fn2(): void {
  // @ts-expect-error Not valid, as void functions cannot return a value
  return true
}
```

2.3.7 Enumerations

`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. In numeric enums, each member is assigned a numeric literal value, as seen in Listing 2.15. Each member can have an optional initializer to specify an exact number corresponding to a member. If omitted, the 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, as seen in Listing 2.16.

Listing 2.15 Numeric enums

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

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.

Listing 2.16 String-based enums

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

2.3.8 Generic Types

Sometimes we need to write code, which needs to work and accept types we don't know in advance. Generic types allow the development of such reusable components that can work over a variety of types rather than a single one. Generic types are created by defining a type parameter that can be used as a placeholder for a specific type. The consumers will be then able to replace the placeholder with their own desired types when using the component. In TypeScript, generic types can be defined on interfaces, functions and classes.

To illustrate the point, consider the implementation of the built-in `Array` type found in the `lib.*.d.ts` files (a subset can be seen at Listing 2.17). The `Array<T>` is a generic type, which accepts a single type argument `T` and is used to describe the type of the elements in the array. The type argument `T` is later used both in arguments and return types of the methods of the `Array<T>` type: `push()` accepts only elements of the same type as the array while `pop()` will return an element of the same type.

Generic types can be interpreted as functions in a meta-programming language found inside the TypeScript type system itself. The meta-programming language implements some of the key concepts found in the functional programming paradigm.

Generic types are considered first-class citizens in the language, being able to be passed as arguments into other generic types, similar to functions in a functional programming language. Generic types are also pure and cannot have any side effects during type checking. We also use recursion in the meta-programming language to break down complex problems into smaller ones and solve them independently.

Listing 2.17 Array type

```
interface Array<T> {
  push(...items: T[]): number;

  pop(): T | undedfined;
}

const strArr: Array<string> = []
const numArr: Array<number> = []

strArr.push("one", "two")
numArr.push(1, 2)

const a = strArr.pop()
//    ^? string

const b = numArr.pop()
//    ^? number
```

There is a notable omission, however: generic types cannot receive other generic types as type arguments [31]. Thus, higher-order functions are not permitted.[4]

2.3.9 Type constraints with `extends`

When writing generic types, sometimes we need to be able to describe some expectations that a type argument must satisfy. For example, we might want to accept types, which do have a certain property, such as `length` as seen in Listing 2.18. To achieve this, we use the `extends` keyword to describe our constraints to the type.

Listing 2.18 Type constraints with `extends`

```
function getLength<T extends HasLength>(obj: T): number {
  return obj.length
}

const a = getLength("hello")
const b = getLength([1, 2, 3])
const c = getLength({ length: 10 })

// @ts-expect-error
// Argument of type '{ foo: string; }' is not
// assignable to parameter of type 'HasLength'.
const d = getLength({ foo: "bar" })
```

The generic function will not be able to accept any types anymore, as desired and we must only pass types, which satisfy the constraints instead.

[4]There is a way to create such type using HOTScript, more on that later

2.3.10 Conditional types

Within the TypeScript meta-language, developers can write conditions and branching logic using conditional types. Conditional types follow a syntax similar to the conditional ternary operators with another case of overloading the `extends` keyword: `Input extends Expect ? A : B`. This can be read as "If type Input is assignable to type Exepct, then the type resolves to type A, otherwise to type B." An example can be seen in Listing 2.19, where the `IsString<T>` type will resolve to `true` if the type argument `T` is assignable to `string` and to `false` otherwise.

Listing 2.19 Conditional types

```
type IsString<T> = T extends string ? true : false
```

We can use the `infer` keyword to deduce or extract a specific type within the scope of conditional types, essentially acting as a way to perform pattern matching. With `infer` we introduce a new generic type variable, which can be later used within the true branch of the conditional type, as seen in the implementation of the `ReturnType<T>` utility type in Listing 2.20. The `ReturnType<T>` type will resolve to the return type of the type argument `T`.

Listing 2.20 Infer in conditional types

```
type ReturnType<T> = T extends (...args: any) => infer R ? R : never;
```

Since TypeScript version 4.7 [32], we can also add an additional type constraint for the inferred type, which will be checked before the conditional type is resolved. This is useful when we want to avoid an additional nested conditional type, as seen in Listing 2.21, where we want to return the first element of the tuple type only if it is a string.

Listing 2.21 Type constraints within infer

```
type FirstIfString<T> =
  T extends [infer S extends string, ...unknown[]]
    ? S
    : never;

// is equivalent to
type FirstIfString<T> =
  T extends [infer S, ...unknown[]]
    ? S extends string ? S : never
    : never;
```

When given a union type within the conditional type, the conditional type will be resolved for each member type in the union separately, essentially distributing the union type. To prevent such behavior, we can wrap the type argument in a tuple or any other structure type.

Listing 2.22 Distributing union types

```
type ToArray<Type> = Type extends any ? Type[] : never;

// $ExpectType string[] | number[]
type A = ToArray<string | number>

type ToArrayNonDist<Type> = [Type] extends [any] ? Type[] : never;

// $ExpectType (string | number)[]
type B = ToArrayNonDist<string | number>
```

2.3.11 Mapped types

Sometimes we need to transform a type into another type. For example, we might want to create a new type, which is a copy of the original type, but with all properties being optional. This can be achieved using mapped types. Mapped types are types, which are created using the syntax for index signatures, commonly used in JavaScript for properties not declared ahead of time. An example is shown in Listing 2.23, where the generic type `ToBoolean<T>` will create a new type which will take all properties from `T` and change their values to `boolean`.

We can also specify mapping modifiers to affect the mutability or optionality of a property: `readonly` and `?` respectively. Prefixing the modifier with either `+` or `-` will either add or remove the modifier to the property.[5] This can be seen in the `Optional<T>` type in Listing 2.23, which will create a new type, which is a copy of the original type, but with all properties being optional.

Listing 2.23 Mapped types

```
type ToBoolean<T> = {
  [K in keyof T]: boolean
}

type Optional<T> = {
  [K in keyof T]+?: T[K]
}
```

Introduced in TypeScript 4.1 [33], we can also use the `as` keyword to re-map keys in mapped types. This can allow us to create, transform or filter out keys when creating a new type. An example is shown in Listing 2.24, where the `Omit<T, Key>` creates a new object type based on type `T` with omitted properties which are assignable to `Key`.

Listing 2.24 Using as in mapped types

```
type Omit<T, Key> = {
  [K in keyof T as Exclude<K, Key>]: T[K]
}
```

[5] + is assumed by default if omitted

2.3.12 Recursive Types

A recursive data type is a data type that includes a reference to itself within the type definition. Recursive types are useful for modeling complex or hierarchical data structures, such as linked lists or trees.

An example can be seen in Listing 2.25, where the `Tree<Value>` generic type represents an object with a value of type `Value` and optional left and right subtrees of the same type.

Listing 2.25 Modeling a binary tree with recursive types

```
type Tree<Value> = {
  value: Value,
  left?: Tree<Value>,
  right?: Tree<Value>
}
```

Using recursive types combined with generic types, we can implement typical recursive algorithms useful for this thesis. One such example can be seen at Listing 2.26, where we implement a `FromEntries<Entries>` generic type, converting a list of `[Key, Value]` tuples into a single object type.

First, we define an additional optional generic type parameter `Accumulator` with an initial type value of . For every tuple in a list, we create an object type containing the current key-value pair with `{ [K in Key]: Value }` and merge it with the accumulator using the `&` operator. The merged object type is then passed as the accumulator to the next iteration. Finally, when the list is empty, we return the accumulator, which will be the final object type.

Listing 2.26 Reduce example

```
type FromEntries<Entries, Accumulator = {}> =
  Entries extends [infer Entry, ...infer Rest]
    ? FromEntries<
        Rest,
        Entry extends [infer Key, infer Value]
          ? { [K in Key]: Value } & Accumulator
          : Accumulator
      >
    : Accumulator;
```

There are some limitations regarding recursive types. To prevent infinite recursion, TypeScript limits the instantiation depth to ensure a consistent and performant developer experience. As of writing, the limit is set to 100 levels for type aliases and 5 million type instantiations [34]. Thanks to the tail-recursion elimination optimization, the limit is set to 1000 levels for tail-optimized recursion types. Thus, it is desired to use tail recursion whenever possible.

Another key limitation related to recursive generic types is that the variables declared with `infer` do not inherit the constraints of the parent type, as seen in Listing 2.27. As the `Tail` type lost the type constraint of `Haystack`, we cannot pass the tail as the new haystack of the

FilterWrong type. To remedy this issue, we need to add an additional type constraint to the inferred type.

Listing 2.27 Recursive types and type constraints

```
type FilterWrong<Haystack extends string[], Needle extends string> =
  Haystack extends [infer Head, ...infer Tail]
    ? Head extends Needle
      // $ExpectError Type 'Tail' does not satisfy the constraint 'string[]'.
      ? [Head, ...FilterWrong<Tail, Needle>]
      : FilterWrong<Tail, Needle>
    : [];

type FilterCorrect<Haystack extends string[], Needle extends string> =
  Haystack extends [infer Head, ...infer Tail extends string[]]
    ? Head extends Needle
      ? [Head, ...FilterCorrect<Tail, Needle>]
      : FilterCorrect<Tail, Needle>
    : [];
```

2.3.13 Template Literal Types

Finally, template literal types are based on the string literal types and allow string interpolation and manipulation within the TypeScript type system. For this thesis, we use template literal types to create a parser of mathematical expressions, but template literal types can be used to create fully typed string-based Domain Specific Languages (DSLs).

Similar to the syntax of JavaScript template literal strings, we use backticks to create a new template literal type. When used with a string literal type, a template literal will create a new string literal type by concatenation [35]. For example, the type `` `Hello ${"World"}` `` will create a new string literal type `"Hello World"`.

Template literal types can be used with primitive types as well, the only limitation being that the primitive type must be stringifiable. That includes all of the primitive types except the `symbol` type. When created, these types are as a subset of their primitive type and can be used to work as a validation mechanism matching a string of an expected format. For instance, the type `` `localhost:${number}` `` will create a new string literal type that will match a string of the format `localhost:PORT`, where `PORT` is a number.

The distributive nature of union types applies to template literal strings as well: the type will be applied for every member type of the union to the template literal, as seen in the Listing 2.28, where we create a new `Style` type with all of the possible combinations of the `Variants` and `Weights` types. Generally, it is preferable to avoid combinations of big union types, as it can lead to worse type-checking performance or an error if a union type reaches 1 000 000 member types.

Finally, we can use inference in template literal types to perform pattern matching within string literals with the combination of conditional types and the `infer` keyword. In Listing 2.29, we create a generic type `SplitString`, which splits a string literal type into a tuple of substrings

Listing 2.28 Distributive nature of unions in template literal types

```
type Variants = "primary" | "secondary"
type Weights = 100 | 200 | 300

type Style = `${Variants}-${Weights}`
//   ^? | "primary-100" | "primary-200" | "primary-300"
//      | "secondary-100" | "secondary-200" | "secondary-300"
```

with a space as the delimiter. We attempt to pattern match a string with `Head`, containing the left side of the split, and `Rest`, including the rest of the split string, as the two inferred types as the result. We also apply type constraints for the inferred types to ensure the types are assignable to `string`.[6] Both of the inferred types are used to create a new tuple type, with `Head` being the first element of the tuple and `Rest` used in a recursive call to split the rest of the string.

Listing 2.29 Pattern matching with template literal types

```
type SplitString<Input extends string> =
  Input extends `${infer Head extends string} ${infer Rest extends string}`
    ? [Head, ...SplitString<Rest>]
    : [Input];
```

2.4 Prior Art

There are multiple basic implementations of math operations in TypeScript. Tasks regarding basic math operations are even part of the TypeChallenges collection[36]. However, most of the do work only on integers, as they work on tuple expansion, which will be further discussed in the implementation part of this thesis.

Nevertheless, multiple libraries in the NPM registry do provide basic math calculations within the TypeScript type system, but none of them provide a fully typed parser of mathematical expressions. Some of the libraries we have found do provide type utilities that operate on floating-point numbers instead of integers, such as `type-fest` [37] or `typescript-lodash`[38]. The most comprehensive implementation of math operations can be found in the `ts-arithmetic` library [39], which provides a fully typed implementation of division.

[6] Albeit unnecessarily, as TypeScript automatically applies the `string` type constraint in this instance

Chapter 3

Implementation

Chapter 4

Testing

4.1 Developer experience

By using `tsserver`, we can see and verify the types representing a symbol during development, by hovering on top of a symbol. This does provide some useful feedback during development but does require significant context switching with the mouse pointer, especially when switching back and forth from implementation to testing. There are various plugins for editors, that are able to display the inferred types in a different manner. One such key plugin used thoroughly during development is `vscode-twoslash-plugins` [40], which allows inserting a `// ^?` comment to display the inferred type of an expression right in the editor.

TODO: Add a screenshot of the vscode-twoslash-plugins in action

4.2 Testing with eslint

TODO: Describe $ExpectType

To remedy the issue, we are using `eslint` together with `@typescript-eslint/parser` as the source code parser and `eslint-plugin-expect-type` plugin to create unit tests for each of the math methods.

- Developer experience
- 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/Babel* [online]. Babel, 2023. [visited on 2023-02-01]. Available from: `https://github.com/babel/babel`.
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. *Standalone Server (Tsserver)* [online]. GitHub. [visited on 2023-03-27]. Available from: `https://github.com/microsoft/TypeScript/wiki/Standalone-Server-(tsserver)`.
27. VANDERKAM, Dan. *Effective TypeScript: 62 Specific Ways to Improve Your TypeScript*. First edition. Beijing [China] ; Sebastopol, CA: O'Reilly Media, 2019. ISBN 978-1-4920-5374-3.

28. *Performance* [online]. GitHub. [visited on 2023-04-02]. Available from: https://github.com/microsoft/TypeScript/wiki/Performance.

29. *Documentation - Everyday Types* [online]. [visited on 2023-03-27]. Available from: https://www.typescriptlang.org/docs/handbook/2/everyday-types.html.

30. *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.

31. *Type Inference for Higher-Order, Generic Curried Function Breaks down When the Function Is Applied to Another Generic Function · Issue #49312 · Microsoft/TypeScript* [online]. GitHub. [visited on 2023-03-28]. Available from: https://github.com/microsoft/TypeScript/issues/49312.

32. ROSENWASSER, Daniel. *Announcing TypeScript 4.7* [online]. TypeScript, 2022-05-24. [visited on 2023-03-29]. Available from: https://devblogs.microsoft.com/typescript/announcing-typescript-4-7/.

33. ROSENWASSER, Daniel. *Announcing TypeScript 4.1* [online]. TypeScript, 2020-11-19. [visited on 2023-03-29]. Available from: https://devblogs.microsoft.com/typescript/announcing-typescript-4-1/.

34. *Implementation of Checker.Ts* [online]. Microsoft, 2023. [visited on 2023-03-31]. Available from: https://github.com/microsoft/TypeScript/blob/55867271933d603f6c29b8eb7399960a71e96ccc/src/compiler/checker.ts.

35. *Documentation - Template Literal Types* [online]. [visited on 2023-04-02]. Available from: https://www.typescriptlang.org/docs/handbook/2/template-literal-types.html.

36. *Type-Challenges/Type-Challenges* [online]. Type Challenges, 2023. [visited on 2023-04-02]. Available from: https://github.com/type-challenges/type-challenges.

37. SORHUS, Sindre. *Sindresorhus/Type-Fest* [online]. 2023. [visited on 2023-04-02]. Available from: https://github.com/sindresorhus/type-fest.

38. KAWAYILINLIN. *kawayiLinLin/Typescript-Lodash* [online]. 2023. [visited on 2023-01-15]. Available from: https://github.com/kawayiLinLin/typescript-lodash.

39. ARIEL. *Type Level Arithmetic* [online]. 2023. [visited on 2023-01-15]. Available from: https://github.com/arielhs/ts-arithmetic.

40. THEROX, Orta. *Vscode-Twoslash-Queries* [online]. 2023. [visited on 2023-03-26]. Available from: https://github.com/orta/vscode-twoslash-queries.

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
```