Making PT accessible

Implementing PT in TypeScript

Petter Sæther Moen

Thesis submitted for the degree of
Master in Informatics: Programming and System Architecture
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2021

Making PT accessible

Implementing PT in TypeScript

Petter Sæther Moen

Making PT accessible

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Acknowledgements

Contents

List of Figures

List of Tables

Preface

Part I

Introduction

Chapter 1

Introduction

1.1 What is PT?

1.2 Purpose of implementing PT in TS

Chapter 2

Background

2.1 Package Templates

essay

2.2 TypeScript

2.2.1 JavaScript / ECMAScript

When I talk about JavaScript in this thesis I will be refering to the ECMAScript standard (More precisely ES6(Kanskje ikke så viktig å spesifisere dette?)).

Part II

The project

Chapter 3

Planning the project

3.1 What Do We Need?

- The ability to add custom syntax (access to the tokenizer / parser)
- Some semantic analysis.

3.2 Syntax

For the implementation of PT we need syntax for the following:

- Defining packages (`package` in PTj)
- Defining templates (`template` in PTj)
- Instantiating templates (`inst` in PTj)
- Renaming classes (=> in PTj)
- Renaming methods (-> in PTj)
- Additions to classes (`addto` in PTj)

`template` and `inst` are both not in use nor reserved in the ECMAScript standard or in TypeScript, and can therefore be used in PTScript(Midlertidlig navn) without any issues.

The keyword `package` in TS / ES is as of yet not in use, however the ECMAScript standard has reserved it for future use. In order to "future proof" our implementation we should avoid using this reserved keyword, as it could have some conflicts with a potential future implementation of packages in ECMAScript. It could also be beneficial to not share the keyword in order avoid creating confusion between the future ES packages and PT Packages. `module` is also a keyword that could be used to describe a PT package, however this is already used in the ES standard, and should therefore also be avoided for similar reasons to `package`, to avoid confusion. We will therefore use (`pack` eller `bundle`? Må nok se litt mer på dette) instead.

For renaming classes PTj uses =>, however in ES this is used in arrow-functions[2]. To avoid confusion and a potentially ambiguous grammar we will have to choose a different syntax for renaming classes. PTj, for historical purposes, uses a different operator (->) for renaming class methods, however for keeping PTScript(Midlertidlig navn) simple we will stick to only having one common operator for renaming.

ECMAScript currently supports renaming of destructured fields using the :(colon) operator and aliasing imports using the keyword `as`. Even though we opted to choose a different keyword for packages, we will here re-use the already existing `as` keyword for renaming as the concepts are so closely related.

PTScript(Midlertidlig navn):

```
template T {
    class A {
        function f() : String {
            ...
        }
    }
}

pack P {
    inst T with A as A (f as g); // Function overloading not support
    addto A {
        i : number = 0;
    }
}
```

PTj:

```
template T {
    class A {
        String f() {
            ...
        }
    }
}

package P {
    inst T with A => A (f() -> g());
    addto A {
        int i = 0;
    }
}
```

3.3 TypeScript vs JavaScript

3.3.1 Verifying templates

One of the requirements for PT is that each template should be verifiable. There is no easy way to verify if some JavaScript code is verifiable without executing it. With TypeScript on the other hand, with the language being statically typed, we can, at least to a much larger extent, verify if some piece of code is valid. And thus we can also use this to validate each separate template in PT.

Now it should be noted that due to TypeScripts type system being unsound one could argue that this requirement of PT is not met. While this is true it still outperforms JavaScript on this remark, and we will later in section 4.2.1 on page 13 discuss more in-depth to what extent this requirement is met.

3.4 Choosing the right approach

Before jumping into a project of this magnitude it is important to find out what approach to use. The goal of this project is to extend TypeScript with the Package Templates language mechanism, this can be achieved as following:

- Making a fork of the TypeScript compiler
- Making a preprocessor for the TypeScript compiler
- Making a compiler plugin / transform
- Making a custom compiler from scratch

3.4.1 Preprocessor for the TypeScript Compiler

More work than ex plugin / transformer.

3.4.2 TypeScript Compiler Plugin / Transform

As of the time of writing this the official TypeScript compiler does not support compile time plugins. The plugins for the TypeScript compiler is, as the TypeScript compiler wiki specifies, "for changing the editing experience only"[4]. However there are alternatives that do enable compile time plugins / transformers;

- ts-loader[6], for the webpack ecosystem
- Awesome Typescript Loader[5], for the webpack ecosystem. Deprecated
- ts-node[7], REPL / runtime
- ttypescript[3], TypeScript tool TODO: Les mer på dette

Unfortunately ts-loader, Awesome Typescript Loader and ts-node does not support adding custom syntax, as it only transforms the AST produced by the TypeScript compiler. Because of this they are not a viable option for our use-case and will therefore be discarded.

3.4.3 Babel plugin

Babel isn't strictly for TypeScript, but for JavaScript as a whole, however we could write our plugin to be dependent on the TypeScript transformation plugin.

Making a Babel plugin will make it very accessible as most web-projects use Babel, and the upkeep is cheap, as plugins are loosely coupled with the core.

In order for a Babel plugin to support custom syntax it has to provide a custom parser, a fork of the Babel parser. Through this we can extend the TypeScript syntax with our syntax for PT. This is all hidden away from the user, as this custom parser is a dependency of our Babel plugin.

Seeing as we have to make a fork of the parser in order to solve our problem, the upkeep will not be as cheap as first anticipated. However being able to have most of the logic loosely coupled with the compiler core it will still make it easier to keep updated than through a fork of the TypeScript compiler.

TODO: Er det støttet å bruke flere plugins med forskjellige parsere? E.g. babel-plugin-typescript + vårt babel plugin?

3.4.4 TypeScript Compiler Fork

Possible, however not as accessible as other alternatives and will make upkeep expensive.

The TypeScript compiler is a monolith. It has about 2.5 million lines of code, and therefore has a quite steep learning curve to get into. If we were to go with this route it would be quite hard to keep up with the TypeScript updates, as updates to the compiler might break our implementation. However as we have seen, going the plugin / transform route also requires us to fork the underlying compiler and make changes to it, however with the majority of the implementation being loosely coupled it would still make it easier to keep up-to-date. That being said it will probably be a lot easier to do semantic analysis in a fork of the TypeScript compiler vs in a plugin / transform.

Chapter 4

Implementation

4.1 Architecture / Parts of the compiler / PP

Lexer, parser, instantiation and renaming,

4.1.1 Lexer and Parser

tree-sitter grammar extending tree-sitter-typescript

4.1.2 Instantiation and renaming

4.1.3 Verification of templates

ts api

4.1.4 Code generation

generate ts and compile ts to js through ts api.

4.2 Does PTS fulfill the requirements of PT?

4.2.1 Verifying templates

Talk about unsoudness of TypeScript. Talk about unsoudness of Java [1] Talk about since the requirement is met with Java we assume it is adequately met with TypeScript as well.

Chapter 5

Difference between PTS and PTj

5.1 Nominal vs. Structural Typing

Nominal and structural are two major categories of type systems. In nominal typing a type is

Write about pros and cons of both nominal and structural typing

nominal pros: Trivial to check if a type is a subtype of another Natural and intuitive recursive types (Structural typing also has recursive types, check if java and typescript support it) Often runtime-objects are tagged with the types, which are useful for multiple things like doing runtime instanceof checks. (This can also be used in structural typing)

5.1.1 What is nominal typing?

```
// Given the following class definitions for A, B and C:
class A {
    void f() {
        ...
    }
}

class B extends A {
    ...
}

class C {
    void f() {
        ...
    }
}

// And a consumer with the following type:
void g(A a) { ... }
```

```
// Would result in the following
g(new A()); // Ok
g(new B()); // Ok
g(new C()); // Error, C not of type A
```

5.1.2 What is structural typing?

```
// Given the same class definitions and the same consumer as in the
// Would result in the following
g(new A()); // Ok
g(new B()); // Ok
g(new C()); // Ok, because C is structurally equal to A
```

Part III

Conclusion

Chapter 6

Results

Bibliography

[1] Nada Amin and Ross Tate. 'Java and scala's type systems are unsound: the existential crisis of null pointers'. In: *ACM SIGPLAN Not.* 51.10 (Dec. 2016), pp. 838–848. ISSN: 0362-1340. DOI: 10.1145/3022671.2984004. URL: https://dl.acm.org/doi/10.1145/3022671.2984004.

[2] *Arrow function expressions - JavaScript | MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow%7B%5C_%7Dfunctions (visited on 19/10/2020).

[3] Cevek. *ttypescript*. URL: https://github.com/cevek/ttypescript.

[4] Microsoft. *microsoft/TypeScript*. URL: https://github.com/microsoft/TypeScript/wiki/Writing-a-Language-Service-Plugin.

[5] Stanislav Panferov. *Awesome TypeScript Loader*. URL: https://github.com/s-panferov/awesome-typescript-loader.

[6] TypeStrong. *ts-loader*. URL: https://github.com/TypeStrong/ts-loader.

[7] TypeStrong. *ts-node*. URL: https://github.com/TypeStrong/ts-node.