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

Autumn 2020

Making PT accessible

Implementing PT in TypeScript

Petter Sæther Moen

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

2.2 TypeScript

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)

`template` and `inst` are both not in used or reserved in the ECMAScript standard or in TypeScript, and can therefore be used in TemplateScript 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 to not create confusion between ES packages and PT Packages. `module` is also a keyword that could be used to describe a PT package, however this is also reserved in the ES standard, and should therefore also be avoided for similar reasons to `package`, to avoid confusion. We will therefore use (`pack` or `bundle`?) instead.

For renaming classes PTj uses =>, however in ES this is used in arrow-functions[1]. 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 TemplateScript 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.

3.3 Why TypeScript

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 end 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.

Unfortunately Babel plugins does not have any way of extending the tokenizer / parser and can not on its own supply a custom syntax, however it is possible to give a custom parser[2], a fork of the Babel-parser, in which the plugin should use. 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 will still make it easier to keep updated than through a fork of the TypeScript compiler.

Babel Plugin is not as nice as I first thought, it seems to be pretty hard to write third-party plugins as you have to make a parser fork for custom syntax, and there is a severe lack of documentation. Most examples of Babel plugins mostly use internal helpers and utils, which are hard to use for third-party plugins.

TODO: Does it support having multiple custom parser? E.g. babel-plugin-typescript + our custom 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 get 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.

Part III

Conclusion

Chapter 4

Results

Bibliography

[1] *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).

[2] Babel. *@babel/parser*. URL: https://babeljs.io/docs/en/babel-parser.

[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.