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

Making PT accessible

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

Printed: Reprosentralen, University of Oslo

Abstract

Acknowledgements

Contents

I Introduction **1**

1 Introduction **3**
1.1 What is PT? . . . 3
1.2 Purpose of implementing PT in TS . . . 3

2 Background **5**
2.1 Package Templates . . . 5
2.2 TypeScript . . . 5

II The project **7**

3 Planning the project **9**
3.1 Implementing PT as a TS library . . . 9
3.1.1 Defining templates . . . 9
3.1.2 Renaming classes and class attributes . . . 9
3.1.3 Instantiating templates . . . 11
3.1.4 Merging classes . . . 11
3.1.5 Conclusion . . . 11
3.2 What Do We Need? . . . 11
3.3 Syntax . . . 11
3.4 TypeScript vs JavaScript / ECMAScript . . . 13
3.5 Choosing the right approach . . . 13
3.5.1 Preprocessor for the TypeScript Compiler . . . 13
3.5.2 TypeScript Compiler Plugin / Transform . . . 13
3.5.3 Babel plugin . . . 14
3.5.4 TypeScript Compiler Fork . . . 14

III Conclusion **15**

4 Results **17**

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 Implementing PT as a TS library

In order to implement PT we need to be able to handle the following:

- Defining templates
- Renaming classes and class attributes
- Instantiating templates
- Merging classes

3.1.1 Defining templates

For defining templates we would like a construct that can wrap our template classes in a scope. We will also need to be able to reference the template. ECMAScript has three options for this, an array, an object or a class. It should however also be possible to inherit from classes in your own template, which pretty much rules out both arrays and objects, as there is no way of referencing other members during definition of the array/object. Templates could there be defined as ECMAScript classes, where each member of the template is a static attribute of the template. In listing 1 on the following page we see an example of how this could be done.

We are making the templates classes static in order to be able to rename them, see section 3.1.2.

3.1.2 Renaming classes and class attributes

Renaming of classes is possible to an extent. Since we made the classes static attributes of the template class we could easily just create a new static field on the template class (with decleration merging to get the correct type for the new name) and `delete`[3] the old field. We can see an example of this in listing 2 on the following page.

Even though we were able to give the class a "new name", this would still not actually rename the class. Any reference to the old names would be left unchanged, and thus we are not able to achieve renaming in TypeScript.

```
class T1 {
  static A = class {
    i = 1;
  };

  static B = class extends T1.A {
    b = 2;
  };
}
```

Listing 1: Example of defining a template

```
class T1 {
  static A = class {
    i = 0;
  };
  static B: any;
}

interface T1 {
  B: typeof T1.A;
}

const classRef = T1.A;
delete T1.A;
T1.B = classRef;
```

Listing 2: Example of renaming a template class

```
class T2 {
  static A = class {};
}

const P = class {};
for (let attr of Object.keys(T2)) {
  P[attr] = T2[attr];
}
```

Listing 3: Example of instantiating a template

3.1.3 Instantiating templates

As with renaming, we are also able to instantiate templates to an extent. We are able to iterate over the attributes of the template class, and populate a package/template with references to the template. An example of this can be seen in listing 3.

The instantiation will only contain references to the instantiated templates classes, while PT instantiations make textual copies of the templates content. Only having references to the original template could mean that if a template that has been instantiated is later renamed, then the instantiated template might lose some of its references.

3.1.4 Merging classes

For merging of types you would use the built-in decleration merging[7]. Implementation merging is also possible because ECMAScript has open classes. For implementation merging you would create an empty class which has the type of the merged declerations, and then assign the fields and methods from the merging classes to this class. There are several libraries that supports class merging, such as mixin-js[5].

3.1.5 Conclusion

Since we are not able to support renaming and instantiations we can conclude that Package Templates can not be implemented as a library in TypeScript, we will need to implement this as part of a compiler.

3.2 What Do We Need?

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

3.3 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[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 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;
        }
    }
```

3.4 TypeScript vs JavaScript / ECMAScript

3.5 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

3.5.1 Preprocessor for the TypeScript Compiler

More work than ex plugin / transformer.

3.5.2 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[8], for the webpack ecosystem
- Awesome Typescript Loader[6], for the webpack ecosystem. Deprecated
- ts-node[9], REPL / runtime
- ttypescript[2], 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.

3.5.3 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?

3.5.4 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.

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

[3] *delete operator - JavaScript | MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/delete (visited on 29/10/2020).

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

[5] *mixin-js - npm*. URL: https://www.npmjs.com/package/mixin-js (visited on 29/10/2020).

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

[7] *TypeScript: Handbook - Declaration Merging*. URL: https://www.typescriptlang.org/docs/handbook/declaration-merging.html (visited on 28/10/2020).

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

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