PTS (Package Template Script)

An Implementation of Package Templates 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

PTS (Package Template Script)

An Implementation of Package Templates in TypeScript

Petter Sæther Moen

PTS (Package Template Script)

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 Package Template Script 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.

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

pack P {
    inst T with A -> A (f -> g); // Function overloading not supported, so don'
    addto A {
        i : number = 0;
    }
}
```

Listing 1:

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 Package Template Script 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.

Package Template Script:

PTj:

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**??** on page ?? discuss more in-depth to what extent this requirement is met.

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

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

Listing 2:

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 could be achieved by one of the following methods:

- 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

At the time of writing 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"[5]. However, there are alternatives that do enable compile time plugins / transformers;

- ts-loader[9], for the webpack ecosystem
- Awesome Typescript Loader[6], for the webpack ecosystem. Deprecated

- ts-node[10], 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.

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

In this chapter we are going to look at the implementation of PT in TypeScript.

4.1 Architecture / Parts of the compiler / PP

4.1.1 Lexer and Parser

tree-sitter grammar extending tree-sitter-typescript

4.1.2 Instantiation and Renaming

Scoping

For creating scopes I chose the following node types for "making new scopes".

- `class_body`
- `statement_block`
- `enum_body`
- `if_statement`
- `else_statement`
- `for_statement`
- `for_in_statement`
- `while_statement`
- `do_statement`
- `try_statement`
- `with_statement`

```
===========================
Closed template declaration
===========================

template T {
    class A {
        i = 0;
    }
}

---
(program
    (template_declaration
        name: (identifier)
        body: (package_template_body
                (class_declaration
                    name: (type_identifier)
                    body: (class_body
                        (public_field_definition
                            name: (property_identifier)
                            value: (number)))))))
```

Listing 3: Example of tree-sitter grammar test

Transforming Nodes to Variable References

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 Notes on Performance

Very slow compiler/PP because of the chosen implementation, with tree traverser for every step.

4.3 Testing

4.3.1 Lexer and Parser

Tree-sitter tests are simple `.txt` files split up into three sections, the name of the test, the code that should be parsed, and the expected parse tree in S-expressions[8].

4.3.2 Transpiler

Started with jest, and used some time to get it to work with typescript files, however had to switch because jest doesn't handle native libraries (tree-sitter) too well. It `requires` the same native library several times, making the wrapping around the native program to break.

Chapter 5

Does PTS Fulfill The Requirements of PT?

5.1 The Requirements of PT

What are the requirements of PT?

As described in [**jot**]

- Parallel extension
- Hierarchy preservation
- Renaming
- Multiple uses
- Type parameterization
- Class merging
- Collection-level type-checking

5.1.1 Parallel Extension

5.1.2 Hierarchy Preservation

```
class A {
    ...
}
```

Listing 4: Example of hierarchy preservation

5.1.3 Renaming

5.1.4 Multiple Uses

In addition to using it for cities and roads, we could in the same program use it to form the structure of pipes and joints in a water distribution system.

5.1.5 Type Parameterization

5.1.6 Class Merging

5.1.7 Collection-Level Type-Checking

5.2 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 6

Difference between PTS and PTj

6.1 Nominal vs. Structural Typing

One of the most notable differences between PTS and PTj are the underlying languages type systems. PTS, as an extension of TypeScript, has structural typing, while PTj on the other hand, an extension of Java, has nominal typing.

Nominal and structural are two major categories of type systems. Nominal is defined as "being something in name only, and not in reality" in the Oxford dictionary. Nominal types area as the name suggest, types in name only, and not in the structure of the object. They are the norm in mainstream programming languages, such as Java, C, and C++. A type could be `A` or `BinTree`, and checking whether an object conforms to a type restriction, is to check that the type restriction is referring to the same named type, or a subtype. Structural types on the other hand is not tied to the name of the type, but to the structure of the object. These are not as common in mainstream programming languages, but are very prominent in research literature. However, in more recent (mainstream) programming languages, such as Go, TypeScript and Julia(at least for implicit typing), structural typing is becoming more and more common. A type in a structurally typed programming language, are often defined as records, and could for example be `name: string` .

6.1.1 Advantages of Nominal Types

Subtypes

In nominal type systems it is trivial to check if a type is a subtype of another, as this has to be explicitly stated, while in structural type systems this has to be structurally checked, by checking that all members of the super type, are also present in the subtype. Because of this each subtype relation only has to be checked once for each type, which makes it easier to make a more performant type checker for nominal type systems. However, it is also possible to achieve similar performance in structurally typed languages

```
// Given the class A
class A {
    void f() { ... }
 }

// A subtype, B, in nominal typing
class B extends A { ... }

// A subtype, C, in structural typing
class C {
    void f() { ... }
    int g() { ... }
}
```

Listing 5: Example of subtype relations in nominal and structural typing, with a Java-like language.

```
interface BinTree<T> {
    getData(): T;
    getChildren(): [BinTree<T> | null, BinTree<T> | null];
}
```

Listing 6: Usage of a recursive type, BinTree, in TypeScript

through some clever representation techniques. We can see an example of subtype relations in both nominal and structural type systems, in a Java-like language, in listing 5. It is important to note that even though `C` is a *subtype* of `A` in a structural language, it is not a *subclass* of `A`.

Recursive types

Recursive types are types that mention itself in its definition, and are widely used in datastructures, such as lists and trees. Another advantage in nominal typing is how natural and intuitive recursive types are in the type system. Referring to itself in a type definition is as easy as referring to any other type. It is however just as easy to do this in structural type systems as well, however for calculi such as type safety proofs, recursive types come for free in nominal type systems, while it is a bit more cumbersome in structurally typed systems, especially with mutually recursive types[7]. Listings 6 and 7 on the next page shows the use of recursive types in TypeScript(structurally typed) and Java(nominally typed), respectively.

Runtime Type Checking

Often runtime-objects in nominally typed languages are tagged with the types(a pointer to the "type") of the object. This makes it cheap and easy to

```java
interface BinTree<T> {
    T getData();
    Pair<BinTree<T>, BinTree<T>> getChildren();
}
```

Listing 7: Usage of a recursive type, BinTree, in Java

do runtime type checks, like in upcasting or doing a `instanceof` check in Java.

6.1.2 Advantages of Structural Types

Tidier and More Elegant

Structural types carry with it all the information needed to understand its meaning.

Advanced Features

Type abstractions(parametric polymorphism, ADTs, user-defined type operators, functors, etc), these do not fit nice into nominal type systems.

More General Functions/Classes

See [4].

6.1.3 What Difference Does This Make For PT?

We are not going to go further into comparing nominal and structural type systems and "crown a winner", as there are a lot useful scenarios for both. We will instead look more closely into how a structural type system fits into PT, and what differences this makes to the features, and constraints, of this language mechanism.

6.1.4 Which Better Fits PT?

```
// 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
```

Listing 8: Example of a nominally typed program in Java

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

Listing 9: Example of a structurally typed program in a Java-like language

Part III

Conclusion

Chapter 7

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 Notices* 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] Donna Malayeri and Jonathan Aldrich. 'Is Structural Subtyping Useful? An Empirical Study'. In: *Programming Languages and Systems, 18th European Symposium on Programming, {ESOP} 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, {ETAPS} 2009, York, UK, March 22-29, 2009. Proceedings*. 2009, pp. 95–111. DOI: 10.1007/978-3-642-00590-9_8. URL: https://doi.org/10.1007/978-3-642-00590-9%7B%5C_%7D8.

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

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

[7] Benjamin C Pierce. *Types and Programming Languages*. MIT Press, 2002. ISBN: 9780262162098.

[8] *S-expression - Wikipedia*. URL: https://en.wikipedia.org/wiki/S-expression (visited on 25/01/2021).

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

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