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

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
2.2.1 JavaScript / ECMAScript . . . 5

II The project **7**

3 Planning the project **9**
3.1 What Do We Need? . . . 9
3.2 Syntax . . . 9
3.3 TypeScript vs JavaScript . . . 11
3.3.1 Verifying templates . . . 11
3.4 Choosing the right approach . . . 11
3.4.1 Preprocessor for the TypeScript Compiler . . . 11
3.4.2 TypeScript Compiler Plugin / Transform . . . 11
3.4.3 Babel plugin . . . 12
3.4.4 TypeScript Compiler Fork . . . 12

4 Implementation **13**
4.1 Architecture / Parts of the compiler / PP . . . 13
4.1.1 Lexer and Parser . . . 13
4.1.2 Instantiation and Renaming . . . 13
4.1.3 Verification of Templates . . . 13
4.1.4 Code Generation . . . 13
4.2 Notes on Performance . . . 13

5 Does PTS Fulfill The Requirements of PT? **15**
5.1 The Requirements of PT . . . 15
5.2 Verifying Templates . . . 17

6 Difference between PTS and PTj **19**
6.1 Nominal vs. Structural Typing 19
6.1.1 Advantages of Nominal Types 19
6.1.2 Advantages of Structural Types 21
6.1.3 What Difference Does This Make For PT? 21
6.1.4 Which Better Fits PT? 22

III Conclusion **23**

7 Results **25**

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**??** on page ?? 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[7], for the webpack ecosystem
- Awesome Typescript Loader[5], for the webpack ecosystem. Deprecated
- ts-node[8], 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

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.

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**]

From [**jot**]

> The following is a list of properties that such a mechanism should have. For each property we first state it for a collection C containing classes A and B (and in the last property also a collection D with a class E), and we then illustrate it with one of the examples above.
>
> - **Parallel extension:** When using the collection C in a certain setting we can add attributes to A and B. These additions should also have effect for the code of C, e.g. so that we by means of an A-variable defined in C can directly (without casting) access the attributes added to A.
>
> - - In the graph example, assume that we have added the int variable length to Edge, and that n is a Node-reference. With this property we can conveniently specify directly: "n.firstEdge().length = 5;", as firstEdge is typed with the extended Edge class.
>
> - **Hierarchy preservation:** The mechanism should allow B to be a subclass of A, and if additional attributes are given to A and to B, then the B with additions should be a subclass of the A with additions. Note that this will not be the case if we just use the collection C with the classes A and B and then define subclasses A' and B' to A and B, respectively, with the additions we want in these subclasses. B' will then not be a subclass of A'.
>
> - - In the compiler example this is exactly what we need in order to be able to add attributes as explained in the example.
>
> - **Renaming:** When C is used, we should be able to change the name of A and B, and of their attributes, so that they fit with the specific use situation.

- - For the graph example, the renaming property makes it possible to rename the nodes and edges to cities and roads.

- **Multiple uses:** It should be possible to use the classes of C for different and independent purposes in the same program, and so that each purpose have different additions and renamings. The compiler should be able to check that each use implies a different set of classes as if they are defined in separate hierarchies.

- - In a program we may need the basic graph structure for different purposes. 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.

- **Type parameterization:** It should be possible to write a collection of classes that assumes the existence of classes that have some required attributes, but are not yet completely defined. In each use of this collection, one can provide specific classes that have at least the required attributes.

- - In the compiler example we assume that the front end shall always produce Java Bytecode, and we use some readymade mechanism for packing the code to a correctly formatted class file. However, a number of such packers may be available, and we do not want to choose which to use while implementing the front end class collection.

- **Class merging:** Assume we have the two collections C (with classes A and B) and D (with class E). When they are both used in the same program, we should be able to merge e.g. the classes A and E so that the resulting class gets the union of the attributes, and so that we via an E-variable defined within D can also directly see the A attributes (and similarly for an A-variable in C).

- - Assume that we in addition to the graph collection have a collection Persons with a class Person. In a program handling personal relations we then want to use both collections together so that we obtain a new class, say PersonNode, which has all the attributes of Node and Person, and where a Person-variable p defined in Persons gives access directly to the Node attributes, e.g. “p.firstEdge”.

In addition to these properties, it is important that such collections of classes can be separately type-checked. We also prefer a mechanism that allows only single inheritance, as the merge property described above to a large extent will take care of the need for combining code from different sources (for which purpose multiple inheritance is often used). Finally, the type system should be as simple and intuitive as possible.

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

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 6.1. 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`.

Listing 6.1: Example of subtype relations in nominal and structural typing, with a Java-like language.

```
// 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() { ... }
}
```

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[6]. Listings 6.1.1 and 6.1.1 shows the use of recursive types in TypeScript(structurally typed) and Java(nominally typed), respectively.

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

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

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

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 [**malayeri**].

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.

```
// 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

// 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
```

6.1.4 Which Better Fits PT?

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

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

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