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

Contents

Abstract . . . iii
Acknowledgments . . . v

1 Introduction 1
1.1 What is PT? . . . 1
1.2 Purpose of Implementing PT in TS . . . 1
1.3 Structure of Thesis . . . 1

2 Background 3
2.1 Package Templates . . . 3
2.1.1 Syntax . . . 3
2.2 TypeScript . . . 9
2.2.1 JavaScript / ECMAScript . . . 9

3 The Language - PTS 11
3.1 Syntax . . . 11
3.2 The PTS Grammar . . . 12
3.3 Example programs . . . 14

4 Planning the project 17
4.1 What Do We Need? . . . 17
4.2 TypeScript vs JavaScript . . . 17
4.2.1 Verifying templates . . . 17
4.3 Choosing the right approach . . . 18
4.3.1 Preprocessor for the TypeScript Compiler . . . 18
4.3.2 TypeScript Compiler Plugin / Transform . . . 18
4.3.3 Babel plugin . . . 18
4.3.4 TypeScript Compiler Fork . . . 19

5 Implementation 21
5.1 Compiler Architecture . . . 21
5.2 Lexer and Parser . . . 21
5.2.1 Parser Generator . . . 21
5.3 Transforming Parse Tree to AST . . . 23
5.3.1 What Does the Parse Tree Look Like . . . 23
5.3.2 The AST Nodes . . . 24
5.3.3 Transforming . . . 25
5.4 Closing Templates . . . 25
5.4.1 Scoping . . . 26

5.4.2 Transforming Nodes to References . . . 26
5.4.3 Performing the rename . . . 26
5.4.4 Going back to the original AST . . . 26
5.5 Verification of Templates . . . 26
5.6 Code Generation . . . 26
5.7 Notes on Performance . . . 26
5.8 Testing . . . 27
5.8.1 Lexer and Parser . . . 27
5.8.2 Transpiler . . . 27

6 Does PTS Fulfill The Requirements of PT? 29
6.1 The Requirements of PT . . . 29
6.1.1 Parallel Extension . . . 29
6.1.2 Hierarchy Preservation . . . 29
6.1.3 Renaming . . . 29
6.1.4 Multiple Uses . . . 29
6.1.5 Type Parameterization . . . 30
6.1.6 Class Merging . . . 30
6.1.7 Collection-Level Type-Checking . . . 30
6.2 Verifying Templates . . . 30

7 Difference between PTS and PTj 31
7.1 Nominal vs. Structural Typing . . . 31
7.1.1 Advantages of Nominal Types . . . 31
7.1.2 Advantages of Structural Types . . . 33
7.1.3 What Difference Does This Make For PT? . . . 33
7.1.4 Which Better Fits PT? . . . 33

8 Results 35

Abstract

Acknowledgements

Chapter 1

Introduction

1.1 What is PT?

1.2 Purpose of Implementing PT in TS

1.3 Structure of Thesis

Chapter 2

Background

2.1 Package Templates

Krogdahl proposed Generic Packages in 2001, which is a language mechanism aimed at "large scale code reuse in object oriented languages"[7]. The idea behind this mechanism is to make modules of classes, called *packages*, that could later be imported and instantiated. This would make "textual copies" of the package body, and would also allow for further expanding the classes of the packages. Modularizing through Generic Packages made the programming more flexible as you would easily be able to write modules with a certain functionality and be able to later import it several times when there is a need for the functionality.

Generic Packages was later extended, and the mechanism is now called Package Templates (while the textual program modules themselves are simply called templates). The system is not fully implemented and there exists a number of proposals for extending it.

2.1.1 Syntax

In this section we will look at the syntax of Package Templates (further referred to as *PT*) in a Java-like language as proposed in [8].

Defining packages and package templates

Packages are defined by a set of classes similar to a normal Java package. Package templates(later just templates for short), are defined in a similar manner except for using the keyword *template*. Listing 1 on the following page shows an example of defining packages and templates. The contents of a package can be used as you would with a normal Java package.

Instantiating templates

Instantiating is what really makes PT useful. When defining packages and templates, PT allows for including already defined templates through instantiating. Instantiation is done inside the body of a package (or a template) with the use of a `inst`-clause. Instantiating a template will make

```
package P {
    interface I { ... }
    class A extends I { ... }
}

template T {
    class B { ... }
}
```

Listing 1: Defining a package P and a template T

"textual copies" of the classes, interfaces and enums from the instantiated template and insert them replacing the instantiation statement at compile time. Note that the template itself still exists and that it can be instantiated again in the same program. Listing 2 on the next page shows an example of instantiating a template inside a package. The resulting package `P` will then have the classes `A` and `B` from template `T` and its own class `C`.

Renaming

During instantiation it is possible to rename classes (as well as interfaces and enums) and class fields. Renaming is a part of the instantiation of templates, and will only affect the copy made for this instantiation, and it is done for the copy before it replaces the `inst`-statement. Renaming is denoted by an optional `with`-clause at the end of the `inst`-statement. In the `with`-clause one can rename classes using the following fat arrow syntax, `A => B`, where class `A` is renamed to `B`, and you can rename class fields with a similar arrow syntax, `i -> j`, where the field `i` is renamed to `j`. For method renaming you have to give the signature of the method so that it is possible to distinguish between overloaded versions, i.e. `m1(int) -> m2(int)`[1].

Field renaming comes after the class renaming enclosed by a set of parentheses. Renaming classes will also affect the signatures of any methods using this class. Listing 3 on page 6 shows an often used example of renaming, where a graph template is renamed to better fit a domain, in this case a road map. When renaming the class `Node` the signature of the methods in `Edge` using this `Node` was also changed to reflect this, i.e. the method `Node getNodeFrom()` would become `City getNodeFrom()` with the class rename, and `City getCityFrom()` with the method renaming.

Renaming makes it possible to instantiate templates with conflicting names of classes, or even instantiate the same templates multiple times. Listing 4 on page 7 shows an example of this where we instantiate the same template T twice without any issues.

[1] On a more technical level the compiler will find the class or field declaration that is gonna be renamed, and then find all name occurrences bound to this declaration and rename these.

```
// Before compile time instantiation of T
template T {
    class A { ... }
    class B { ... }
}

package P {
    inst T;
    class C {
        A a;
        B b() {
            ...
        }
    }
}

// After compile time instantiation of T
package P {
    class A { ... }
    class B { ... }
    class C { ... }
}
```

Listing 2: Instantiating template T in package P. Note that class C can reference class A and B as if they were defined in the same package, which they essentially are after the instantiation.

```
template Graph {
    class Node {
        ...
    }

    class Edge {
        Node getNodeFrom() { ... }
        Node getNodeTo() { ... }
    }

    class Graph {
        ...
    }
}

package RoadMap {
    ...
    inst Graph with
        Node => City,
        Edge => Road
            (getNodeFrom() -> getStartingCity(),
            getNodeTo() -> getDestinationCity()),
        Graph => RoadSystem;
    ...
}
```

Listing 3: Example of renaming classes during instantiation. This could be used to make the classes fit the domain of the project better.

```
template T {
    class A {
        void m() { ... }
    }
}

package P {
    inst T;
    inst T with A => B;
}

// package P after compile time instantiation and renaming
package P {
    class A {
        void m() { ... }
    }

    class B {
        void m() { ... }
    }
}
```

Listing 4: Example of instantiating the same template twice solved by renaming.

```
template T {
    class A {
        void someMethod() { ... }
    }
}

package P {
    inst T;
    addto A implements Runnable {
        int i;
        void someOtherMethod() { ... }
        void run() { ... }
    }
}
```

Listing 5: Adding new field and extending the implemented interface for the instantiated class A in package P

Additions to a template

When instantiating a template you can also add fields to the classes of the template, as well as extending the classes implemented interfaces and this will only apply to the current copy. These additions are written inside an `addto`-clause. Extending the class with additional fields is done in the body of the clause, like you would in a normal Java class. If an addition has the same signature as an already existing method from the instantiated template class, then the addition will override the existing method, similarly to traditional inheritance. Extending the list of implemented interfaces for a class can be done by suffixing the `addto`-clause with `implements` and the list of interfaces. Snakk om hvorfor dette er veldig viktig/nyttig. Listing 5 shows an example of adding fields and implemented interfaces to an instantiated class. The resulting class `A` in package `P` would have the field `i`, methods `someMethod`, `someOtherMethod` and `run`, as well as implementing the interface `Runnable`.

The extension of classes using the `addto`-clause is done independently for each class, ignoring any inheritance-patterns. If there is a class `A` with a subclass `B`, they can both get extensions independently of each other. Any extensions made to class `A` would of course still be inherited to class `B`, as with normal Java inheritance.

Merging classes

If two or more classes in the same or in different instantiations share the same name they will be merged into one class. Through this mechanism PT achieves a form of static multiple inheritance. If two classes don't share the same name, it is still possible to force a merge through renaming them to the same name. In listing 6 on the next page we see an example of renaming

```
template T1 {
    class A {
        ...
    }
}

template T2 {
    class B {
        ...
    }
}

package P {
    inst T1 with A;
    inst T2 with B => A;
}
```

Listing 6: Instantiation with class merging through renaming

class `B` to `A` in order to merge them under the class name `A`. Merging the classes would simply lead to having a single class `A` with the fields from both classes. The two classes `A` and `B`, from templates `T1` and `T2` respectively, no longer exists in package `P`, but have formed the new class `A`, which is a union of both. Any pointers typed with the old `A` or `B` will now be pointing to the new merged class `A`.

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?)).

Chapter 3

The Language - PTS

While the majority of the semantics of PTS will be equal to that of PTj, we will have to change some syntax.

3.1 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)
- Specifying renaming for an instantiation (`with` in PTj)
- Renaming classes (`=>` in PTj)
- Renaming class fields (`->` in PTj)
- Additions to classes (`addto` in PTj)

`template`, `addto`, and `inst` are all 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 to 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 in order to avoid confusion. We will therefore use `pack` instead.

Renaming in PTj uses `=>`(fat-arrow) for renaming classes, and `->`(thin-arrow) for renaming class fields. PTj, for historical purposes, used two

different operators for renaming classes and methods, however in more recent PT implementations, such as [6], a single common operator is used for both. We will do as the latter, and only use a single common operator for renaming. Another reason for rethinking the renaming syntax is the fact that the `=>`(fat-arrow) operator is already in use in arrow functions[2], and reusing it for renaming could potentially produce an ambiguous grammar, or the very least be confusing to the programmer. JavaScript currently supports renaming of destructured fields using the `:`(colon) operator and aliasing imports using the keyword `as`. We could opt to choose one of these for renaming in PTS as well, however in order to keep the concepts separated, as well as making the syntax more familiar for Package Template users, we will go for the `->`(thin-arrow) operator.

The `with` keyword is currently in use in JavaScript for `with`-statements[10]. With it being a statement, we could still use it and not end up with an ambiguous grammar, however as with previous keywords, we will avoid using it in order to minimize concept confusion. Instead of this we will contain our instantiation renamings inside a block-scope(`{ }`). Field renamings for a class will remain the same as in PTj, being enclosed in parentheses(`( )`).

Another change we will make to renaming is to remove the requirement of having to specify the signature of the method being renamed. This was necessary in PTj as Java supports overloading, which means that several methods could have the same name, or a method and a field. Method overloading is not supported in JavaScript/TypeScript, and we do therefore not need this constraint.

3.2 The PTS Grammar

Now that we have made our choices for keywords and operators we can look at the grammar of the language.

PTS is an extension of TypeScript, and the grammar is therefore also an extension of the TypeScript grammar. There is no published official TypeScript grammar (other than interpreting it from the implementation of the TypeScript compiler), however up until recently there used to be a TypeScript specification[12]. This TypeScript specification was deprecated as it proved a to great a task to keep updated with the ever-changing nature of the language. However, most of the essential parts are still the same. The PTS grammar is therefore based on the TypeScript specification, and on the ESTree Specification[13].

In figure 3.1 on the facing page we can see the PTS BNF grammar. This is not the full grammar for PTS, as I have only included any additions or changes to the original TypeScript/JavaScript grammars. More specifically the non-terminal ⟨declaration⟩ is an extension of the original grammar, where we also include package and template declarations as legal declarations. The productions for non-terminals ⟨id⟩, ⟨class declaration⟩, ⟨interface declaration⟩, and ⟨class body⟩ are also from the original grammar.

⟨declaration⟩ ⊨ ... | ⟨package declaration⟩ | ⟨template declaration⟩
⟨package declaration⟩ ⊨ `pack` ⟨id⟩ ⟨PT body⟩
⟨template declaration⟩ ⊨ `template` ⟨id⟩ ⟨PT body⟩
⟨PT body⟩ ⊨ `{` ⟨PT body decls⟩ `}`
⟨PT body decls⟩ ⊨ ⟨PT body decls⟩ ⟨PT body decl⟩ | λ
⟨PT body decl⟩ ⊨ ⟨inst statement⟩ | ⟨addto statement⟩ |
⟨class declaration⟩ | ⟨interface declaration⟩
⟨inst statement⟩ ⊨ `inst` ⟨id⟩ ⟨inst rename block⟩
⟨inst rename block⟩ ⊨ `{` ⟨class renamings⟩ `}` | λ
⟨class renamings⟩ ⊨ ⟨class rename⟩ | ⟨class rename⟩ , ⟨class renamings⟩
⟨class rename⟩ ⊨ ⟨rename⟩ ⟨field rename block⟩
⟨field rename block⟩ ⊨ `(` ⟨field renamings⟩ `)` | λ
⟨field renamings⟩ ⊨ ⟨rename⟩ | ⟨rename⟩ , ⟨field renamings⟩
⟨rename⟩ ⊨ ⟨id⟩ `->` ⟨id⟩
⟨addto statement⟩ ⊨ `addto` ⟨id⟩ ⟨addto heritage⟩ ⟨class body⟩
⟨addto heritage⟩ ⊨ ⟨class heritage⟩ | λ

Figure 3.1: BNF grammar for PTS. The non-terminals ⟨declaration⟩, ⟨id⟩, ⟨class declaration⟩, ⟨interface declaration⟩, and ⟨class body⟩ are productions from the TypeScript grammar. The ellipsis in the declaration production means that we extend the TypeScript production with some extra choices.
Legend: Non-terminals are surrounded by ⟨angle brackets⟩. Terminals are in `typewriter font`. Meta-symbols are in regular font.

3.3 Example programs

Figure 7 on the next page.

```
// PTS
template T {
    class A {
        function f() : string {
            ...
        }
    }
}

pack P {
    inst T { A -> A (f -> g) };
    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;
    }
}
```

Listing 7: An example program with instantiation, renaming, and addition-classes in PTS vs. PTj

Chapter 4

Planning the project

4.1 What Do We Need?

There are a lot of approaches one can take when working with TypeScript, however due to the nature of this project there are some restrictions we have to abide by. Our approach should allow the following:

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

In addition to these we would also like to look for some other desirable traits for our implementation:

- Loosely coupled implementation (So that new versions of typescript not necessarily breaks our implementation).
- Mer?

4.2 TypeScript vs JavaScript

4.2.1 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 type secure. 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.

4.3 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

4.3.1 Preprocessor for the TypeScript Compiler

More work than ex plugin / transformer.

4.3.2 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"[11]. However, there are alternatives that do enable compile time plugins / transformers;

- ts-loader[19], for the webpack ecosystem
- Awesome Typescript Loader[15], for the webpack ecosystem. Deprecated
- ts-node[20], REPL / runtime
- ttypescript[4], 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.

4.3.3 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.

4.3.4 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 5

Implementation

In this chapter we are going to look at the implementation of PTS.

5.1 Compiler Architecture

5.2 Lexer and Parser

5.2.1 Parser Generator

There are a lot of parser generators out there, but there is no one-size-fits-all solution. In order to navigate through the sea of options we need to set some requirements in functionality, so that we can more easily find the right tool for the task.

As we talked about in section 4.1 on page 17, we set ourselves the goal to find an approach that would allow us to create an implementation that was loosely coupled with TypeScript. TypeScript is a large language that is constantly updated, and is getting new features fairly often. Because of this one of the requirements for our choice of parser generator is the possibility for extending grammars. This is important because we want to keep our grammar loosely coupled with the TypeScript grammar, and don't want to

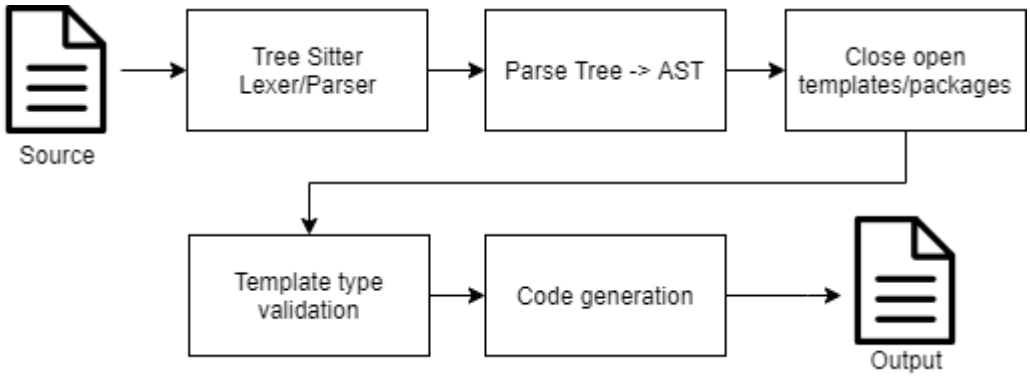

Figure 5.1: Overview of the compiler

be forced to rewrite the entire TypeScript grammar, as well as keeping it up-to-date.

We will be working with the TypeScript API, which only has a runtime library in JavaScript/TypeScript. Therefore, another desired attribute is a runtime library in TypeScript. This is not as essential as the requirement above, as we can get around using the TypeScript API and instead use the TypeScript CLI, or create a CLI in the language of the parser generator tools runtime library.

ANTLR4

ANTLR, ANother Tool for Language Recognition, is a very powerful and versatile tool, used by many, such as Twitter for query parsing in their search engine[16].

ANTLR supports extending grammars, or more specifically importing them. Importing a grammar works much like a "smart include". It will include all rules that are not already defined in the grammar. Through this you can extend a grammar with new rules or replacing them. It does not however support extending rules, as in referencing the imported rule while overriding[16]. This isn't a major issue however as you could easily rewrite the rule with the additions.

The only supported runtime library in ANTLR is in Java. This does not mean that you won't be able to use it in any other language, as you could simply invoke the runtime library through command line, however it is worth keeping in mind.

Overall ANTLR seems like a good option for our project, but the lack of a runtime library in TypeScript is a hurdle we would rather get a round if we can.

Bison

Bison is a general-purpose parser generator. It is one of many successors to Yacc, and is upwards compatible with Yacc[5].

Bison does not support extending grammars. The tool works on a single grammar file and produces a C/C++ program. There is a possibility to include files, like with any other C/C++ program, in the grammar files prologue, however this will not allow us to include another grammar, as it only inserts the prologue into the generated parser. In order to extend a grammar we would have to change the produced parser to include some extra rules. Although this could possibly be automated by a script, it seems too hacky of a solution to consider.

On top of this Bison does not have a runtime library in JavaScript/TypeScript. There does exist some ports/clones of Bison for JavaScript, such as Jison and Jacob, however to my knowledge these also lack the functionality of extending grammars.

Tree-sitter

Tree-sitter is a fairly new parser generator tool, compared to the others in this list. It aims to be general, fast, robust and dependency-free[3]. The tool has been garnering a lot of traction the last couple of years, and is being used by Github, VS Code and Atom to name a few. It has mainly been used in language servers and syntax highlighting, however it should still work fine for our compiler since it does produce a parse tree.

Although it isn't a documented feature, Tree-sitter does allow for extending grammars. Extending a grammar works much like in ANTLR, where you get almost a superclass relation to the grammar. One difference from ANTLR though is that it does allow for referencing the grammar we are extending during rule overriding. This makes it easier and more robust to extend rules than in ANTLR.

Tree-sitter also has a runtime library for TypeScript, which makes it easier for us to use it in our implementation than the previous candidates.

Another cherry on top is that Tree-sitter is becoming one of the mainstream ways of syntax highlighting in modern editors and IDEs, which means that we could utilize the same grammar to get syntax highlighting for our language.

All this makes Tree-sitter stand out as the best candidate for our project.

Implementing Our Grammar in Tree-sitter

Tree-sitter uses the term rule instead of production, and I will therefore also refer to productions as rules here.

Extending a grammar in Tree-sitter works much like extending a class in an object-oriented language. A "sub grammar" inherits all the rules from the "super grammar", so an empty ruleset would effectively work the same as the super grammar. Just like most object-oriented languages have access to the super class, we also have access to the super grammar in Tree-sitter. All of this enables us to add, override, and extend rules in an existing grammar, all while staying loosely coupled with the super grammar. By extending the grammar, and not forking it, we are able to simply update our dependency on the TypeScript grammar, minimizing the possibility for conflicts.

As mentioned, Tree-sitter allows for referencing the super grammar during rule overriding, effectively making it possible to combine the old rule and the new. A good example of overriding and combining rules can be found in the grammar of PTS, see listing 8 on the following page, where we override the `_declaration` rule from the TypeScript grammar, to include the possibility for package and template declarations.

5.3 Transforming Parse Tree to AST

5.3.1 What Does the Parse Tree Look Like

The tree-sitter API produces a parse tree.

```
    _declaration: ($, previous) =>
        choice(
            previous,
            $.template_declaration,
            $.package_declaration
        )
```

Listing 8: Snippet from the PTS grammar, where we override the `_declaration` rule from the TypeScript grammar, and adding two additional declarations.

5.3.2 The AST Nodes

Tree-sitter is a parser-generator written in Rust. Fortunately for us we can still use tree-sitter in Node, due to Node supporting native addons[1]. Native addons in Node are fairly new, and at the time of writing the tree-sitter Node bindings are still using the older unstable NAN instead of the newer and more stable napi. For the most part it does work, however I did meet some difficulties with the produced parse tree, more specifically the spread operator was not behaving properly on the native produced objects. To get around this we will be walking through the parse tree and produce an AST. What this means in practice is that we are going to be ignoring some parsing specific properties. One of the changes we are going to make is that we are going to ignore if a node is named or unnamed, we will be keeping all nodes. This will help us later in code generation.

For each AST node we picked out the following properties from the parse tree:

- Node type
- Text
- Children

The node type is a string representing the rule which produced the node. An AST node with a node type value of `"class_declaration"` for instance is a class declaration node.

The text field of an AST node contains the textual representation/code for the node and its children. A class declaration node for instance would of course contain the class declaration(`"class A extend B {"`), but also the entire body of the class. This text field is really only useful for leaf nodes, as this would for instance contain the value of a number, string, etc.

Finally, the children field is, as the name would suggest, a list of all the children of the node. For a class declaration node this would contain a leaf node containing the keyword `class`, a type identifier for the class name, and the class body. Optionally it could also contain a class heritage

[1]Node native addons are dynamically-linked shared objects written in C++[14].

node, which again contains either an extends clause, an implements clause or both.

We could have also opted to get the start position and end position of each node, so that we could use this to produce better error messages. This was however not a priority in this thesis.

5.3.3 Transforming

Walk DFS

5.4 Closing Templates

A closed template is a template that comprises only class, interface or enum declarations. An open template on the other hand is a template that contains at least one instantiation statement.

The task of closing open packages and templates is what most of the implementation is focused around. It is the task of performing the instantiations declared in the body of a package/template.

For instantiations without renaming the task is fairly simple. We merely have to find the referenced template, and replace the instantiation statement with the body of said template. Renaming on the other hand requires a bit more work.

In order to perform renaming on an instantiation we will have to perform the following tasks.

1. Create a scope for the template body, and give all nodes of the body access to this scope
2. Find all identifiers, member expressions, class declarations, etc., and replace them with *reference nodes*[2]
3. Perform the rename.
4. Transform the scoped AST with reference nodes back to the original AST.

If the template body is also open we would have to close it as well. We want to close the nested templates before closing the upper templates, as renaming at the top level should affect all members from the nested instantiations.

Finally, once all templates have been closed we will have to perform class merging and apply any additions to classes.

In order to get a better understanding of this we will go through each step of closing a template in more detail.

[2]Reference nodes are AST nodes that contain a pointer to the class or field they are supposed to represent. This makes the task of renaming easier as we only have to worry about changing the name in one place. We will go into more detail about this in section 5.4.2 on the next page.

5.4.1 Scoping

This step of closing templates works on the body of a copy of the instantiated template. In order to be able to rename classes and class fields we first need to create correct scopes in which the renaming can be applied to. We start of with a list of normal AST nodes and will transform these nodes into nodes that has a reference to the scope they are part of.

Scopes in the compiler mainly consists of three different classes. The `Scope`, `Variable`, and `Class` classes.

The `Scope` class is essentially a symbol table that optionally extends a parent scope. A scope without a reference to a parent scope is the root scope. The symbol table is implemented as a map from the original field or class name, to a reference to either `Variable` or `Class` instance. Looking up symbols in the symbol table will always start in the called scope, and work its way upwards until we reach the root scope. This allows us to correctly handle shadowed variables.

The `Variable` class is a representation for class fields. The class contains the name of the field.

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`

5.4.2 Transforming Nodes to References

5.4.3 Performing the rename

5.4.4 Going back to the original AST

5.5 Verification of Templates

ts api

5.6 Code Generation

generate ts and compile ts to js through ts api.

5.7 Notes on Performance

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

```
============================
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 9: Example of tree-sitter grammar test

5.8 Testing

5.8.1 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*[3].

5.8.2 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.

[3]S-expressions are textual representations for tree-structured data. See [18] for additional information and examples

Chapter 6

Does PTS Fulfill The Requirements of PT?

6.1 The Requirements of PT

What are the requirements of PT?
As described in [8]

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

6.1.1 Parallel Extension

6.1.2 Hierarchy Preservation

6.1.3 Renaming

6.1.4 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.

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

Listing 10: Example of hierarchy preservation

6.1.5 Type Parameterization

6.1.6 Class Merging

6.1.7 Collection-Level Type-Checking

6.2 Verifying Templates

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

Chapter 7

Difference between PTS and PTj

7.1 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 `Tree`, 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` .

7.1.1 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 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 11: Example of subtype relations in nominal and structural typing, in a Java-like language.

```
interface Tree<T> {
    getData(): T;
    getChildren(): Tree<T>[];
}
```

Listing 12: Usage of a recursive type, `Tree`, in TypeScript

through some clever representation techniques[17]. We can see an example of subtype relations in both nominal and structural type systems, in a Java-like language, in listing 11. 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. An 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[17]. Listings 12 and 13 on the facing 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

```
interface Tree<T> {
    T getData();
    List<Tree<T>> getChildren();
}
```

Listing 13: Usage of a recursive type, `Tree`, in Java

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

7.1.2 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 [9].

7.1.3 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.

7.1.4 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 14: Example of a nominally typed program in a Java-like language

```
// Given the same class definitions and
// the same consumer as in the example above.
// 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 15: Example of a structurally typed program in a Java-like language

Chapter 8

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] Max Brunsfeld. *Tree-sitter*. URL: https://tree-sitter.github.io/tree-sitter/ (visited on 19/02/2021).

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

[5] Free Software Foundation. *Bison 3.7.1*. URL: https://www.gnu.org/software/bison/manual/bison.html (visited on 24/02/2021).

[6] Eirik Isene. *PT#-Package Templates in C# Extending the Roslyn Full-Scale Production Compiler with New Language Features*. Tech. rep. 2018.

[7] Stein Krogdahl. 'Generic Packages and Expandable Classes'. In: (2001).

[8] Stein Krogdahl, Birger Møller-Pedersen and Fredrik Sørensen. 'Exploring the use of package templates for flexible re-use of collections of related classes'. In: *Journal of Object Technology* 8.7 (2009), pp. 59–85. ISSN: 16601769. DOI: 10.5381/jot.2009.8.7.a1.

[9] 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.

[10] MDN Web Docs. *with - JavaScript*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/with (visited on 24/02/2021).

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

[12] Microsoft Corporation. *Typescript Language Specification Version 1.8*. Tech. rep. Oct. 2012. URL: https://web.archive.org/web/20200808173225if%7B%5C_%7D/https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md.

[13] Mozilla Contributors and ESTree Contributors. *The ESTree Spec*. Tech. rep. URL: https://github.com/estree/estree.

[14] Node. *C++ Addons | Node.js v11.3.0 Documentation*. URL: https://nodejs.org/api/addons.html (visited on 04/03/2021).

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

[16] Terence (Terence John) Parr. *The definitive ANTLR 4 reference*. Dallas, Texas, 2012.

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

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

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

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