Endringer fra forrige møte

Background

- Skrive litt mer JavaScript background, og forklare hvordan språkets prototype-baserte objekt-orientering fungerer. Section **??**.
- Skrive litt om hva TypeScript er. Section **??**.

Planning the project

Implementation

Does PTS Fulfill the Requirements of PT?

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

Acknowledgements

Part I

Introduction

Chapter 1

Introduction

In this thesis we will be looking at the language mechanism Package Templates and the language TypeScript and how this language mechanism fits into the language.

Package Templates

1.1 What is PT?

1.2 Purpose of Implementing PT in TS

1.3 Structure of Thesis

In part one of this thesis we will focus on getting to know the language mechanism Package Templates, and get a proper understanding of the programming language TypeScript and its ecosystem in chapter **??**.

With a proper understanding of these topics we will move on to the second part of the thesis. This part will focus on the implementation of the project. The part starts off by planning out and designing the language PTS in chapter **??**. PTS, short for Package Template Script, will be a superset of TypeScript with a basic implementation of the language mechanism PT. Chapter **??** will then focus on making a plan for the implementation of our programming language. Here we will perform a requirement analysis of the project and seek out an approach for the implementation. With a plan for approach and execution we will discuss the resulting implementation of the compiler in chapter **??**.

In the final part of this thesis we will discuss the resulting implementation. This discussion starts off in chapter **??** where we will be examining the implementation, looking at whether it fulfills the requirements of PT. After this we will be comparing our implementation of Package Templates to another implementation of PT in Java in chapter **??**, looking at the advantages and disadvantages of having PT in a structurally typed language.

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" [**krogdahl:GP**]. 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 Basics of Package Templates

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 [**jot**], with the extensions of required types as proposed in [**requiredtypes**].

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 **??** 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

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

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

Listing 1: Defining a package P and a template T

instantiating. Instantiation is done inside the body of a package (or a template) with the use of an `inst`-clause. Instantiating a template will make "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 **??** 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 attributes*. Here and henceforth we will be using the term class attributes to describe the union of both the fields and the methods of a class. 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 rename class attributes with a similar arrow syntax, `i -> j`, where the attribute `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)`. On a more technical level the compiler will find the class or attribute declaration that is going to be renamed, and then find all name occurrences bound to this declaration and rename these.

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 **??** 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 getStartingCity()` with the method renaming.

Renaming makes it possible to instantiate templates with conflicting names of classes, or even instantiate the same templates multiple times.

```
// 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.

Listing **??** shows an example of this where we instantiate the same template, `T`, twice without any issues.

Additions to a template

When instantiating a template you can also add attributes to the classes of the template, as well as extending the class' implemented interfaces and this will only apply to the current copy. These additions are written inside an `addto`-clause. Extending the class with additional attributes 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. Listing **??** shows an example of adding attributes to an instantiated class. The resulting class `A` in package `P` would have the field `i` and the methods `someMethod` and `someOtherMethod`.

It is also possible to extend the list of implemented interfaces of a class by suffixing the `addto`-clause with a `implements`-clause containing the list of implementing interfaces. Having the possibility to add implementing interfaces to classes makes working with PT easier and enables the programmer to re-use template classes to a much larger degree. This feature's use is easier explained through an example.

Say we have implemented some class that will deal with logging. This

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

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

Listing 5: Adding new attributes to the instantiated class A in package P

class can log the state of a class given that the class implements some interface `Loggable`. If we want to be able to log the state of our Graph implementation, from **??**, then the Graph class would need to implement the `Loggable` interface. We can't do this at the declaration of the Graph template, as we do not have access to the interface at the time of declaration. By using `addto` however we are able to add the `Loggable` interface and the `log` method to the Graph class. You can also achieve the same functionality through class merging, which we will look at in section **??**.

Merging classes

If two or more classes in the same or in different instantiations in one package share the same name they will be merged into one class. Through this mechanism PT achieves a form of multiple inheritance. This form of inheritance is different from what you would normally find in Java, it acts more like *mixins* (a language feature for injecting code into a class, first introduced in the programming language Jigsaw [**jigsaw**]). The merging of the classes will not lead to a classic superclass-subclass relation, as the merged class is simply a concatenation of textual copies of the merging classes. We call this kind of inheritance *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 **??** we see an example of renaming class `A` from template `T1` to `C` and class `B` from template `T2` to `C`. Renaming these two classes to the same name will force these classes to be merged in package `P`. The result of this merge is a new class `C` with the attributes of 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 `C`, which is a union of both. Any pointers typed with the old `A` or `B` will now be typed as the new merged class `C`.

```
    template Logger {
        interface Loggable {
            String log();
        }

        class Logger {
            void log(Loggable loggable) {
                ...
            }
        }
    }

    package P {
        inst Graph;
        inst Logger;
        addto Graph implements Loggable {
            String log() {
                ...
            }
        }
    }
```

Listing 6: Adding the `Loggable` interface to the Graph class from listing **??**, making it compatible with our logger implementation.

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

template T2 {
    class B {
        ...
    }
}

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

Listing 7: Instantiation with class merging through renaming

Required Types

Required types in PT gives the programmer extra flexibility when declaring templates. They are generic types declared at the template level, which can be substituted at instantiation. If a template instantiates another template with a required type, but does give an actual parameter for the required type, then the required type is propagated to the template it's being instantiated into. When a template with required types is instantiated in a package, then all the required types needs an actual parameter.

Required types can then be used throughout the template, similar to how you would use generics in a Java class. The most basic required type can be seen below.

```
template T { required type R { } }
```

Here `R` is a required type for which any class or interface can be substituted at instantiation. Required types can be constrained using both nominal types, such as classes and interfaces, and structural types, constraining the type to have certain attributes. Below we can see examples of defining different required types with different types of constraints, where the first has a simple nominal type constraint, the second having a similar structural constraint, and the third having both a nominal and structural constraint.

```
template T {
    required type R1 extends Runnable { }
    required type R2 { void f(); }
    required type R3 extends Runnable { void f(); }
}
```

We could then instantiate the template, `T`, giving classes or interfaces as actual parameters for the required types, as seen below.

```
package P {
    class A implements Runnable {
        void run() { ... }
    }

    class B {
         void f() { ... }
    }

    interface C extends Runnable {
        void f();
    }

    inst T with
        R1 <= A,
        R2 <= B,
```

```
        R3 <= C;
}
```

Required types as presented above can not be used as classes or interfaces, that is you cannot create a new object of the type or implement the type as an interface for a class, as they can be substituted with both. They can only be used as type references, like in the simple Tree implementation example below.

```
template Tree {
    required type E { }

    class Tree {
        Node root;
        ...
    }

    class Node {
        E e;
        List<Node> children;
        ...
    }
}
```

It is also possible to declare required classes and interfaces similarly to required types, which can be used as classes and interfaces respectively, however this will not be discussed in this thesis. If the reader wants to get a better understanding for required types, required classes and required interfaces I recommend reading [**requiredtypes**].

2.1.2 Advanced Topics of PT

With a firm understandings of the basics of PT we will now have to dig a bit deeper into some terminology and restrictions of the language mechanism.

Open and Closed Templates

A *closed template* is a template that does not contain any instantiation statements nor any additions to classes, it comprises only classes, interfaces and enums. The body of a closed template in Java is simply a Java program. Closed templates are self-contained units that can be separately type-checked [**Axelsen2012**]. An *open template* on the other hand is a template which has one or more instantiations or `addto`-statements in its body.

Open templates will be closed at compile-time. The task of closing a template is that of performing the contained instantiations and additions to classes. Open templates can instantiate open templates, as long as these instantiations are not cyclic. What this means is that a template A can instantiate a template B if template B does not contain any instantiations of template A. A template B contains an instantiation of template A if it has

an instantiation of template A in its body, or contains a nested instantiation of template A.

Packages can also be open and closed and work in the same manner as with templates, except that they can not be instantiated.

Avoiding Indirect Multiple Inheritance

While PT enables the programmer to merge classes together and giving us some form of static multiple inheritance, it is not intended to actually enable multiple inheritance. However, with class merging, it is not uncommon that a class might end up with two or more different superclasses. To avoid this PT has some restrictions to stop this from happening.

The first restriction is that if an external class is used as a superclass, then it can only be merged with other classes with the same superclass. This restriction is necessary since we can't rename nor merge external classes.

The second restriction is that if two or more classes are merged in an instantiation, then their superclasses must also be merged in the same instantiation. This is to avoid the situation where merging two classes results in two or more different superclasses [**jot**].

2.1.3 PTj

Implementation of PT in Java.

2.2 TypeScript

Before we look at what TypeScript is we first need to understand JavaScript and the JavaScript ecosystem.

2.2.1 JavaScript

Back in the mid 90s web pages could only be static, however there was a desire to remove this limitation and make the web a more interactive platform, as it became increasingly more accessible to non-technical users. In order to remove this limitation, Netscape, with its Netscape Navigator browser, partnered up with Sun in order to bring the Java platform to the browser, and hired Brendan Eich to create a scripting language for the web. Eich was tasked to create a Scheme like language with syntax similar to Java, and the language was intended to be a companion language to Java. The language when it first released was called LiveScript, however it was later renamed to JavaScript. This has been characterized as a marketing ploy by Netscape in order to give the impression that it was a Java spin-off.

Microsoft, with its Internet Explorer, adopted the language and named it JScript. During this time Microsoft and Netscape would both ship new features to the language in order to increase the popularity for their respective browsers. Because of this war between browsers the language was later handed over to ECMA International as a starting point for a

standard specification for the language. This ensured that users would get the same experience across different browsers, making the web more accessible [**jswikipedia**].

A web page generally consists of three layers of technologies. The first layer is HTML, which is the markup language that is used to structure the web page. Second is CSS which gives our structured documents styling such as background colors and positioning. The third and final layer is JavaScript which enables web pages to have dynamic content. Whenever you visit a website that isn't just static information, but instead might have timely content updates, interactive maps, etc., then JavaScript is most likely involved [**whatisjs**].

JavaScript is a programming language conforming to the ECMAScript standard. ECMAScript is a JavaScript standard, created by Ecma International, made to standardize the JavaScript language and ensure interoperability across different browsers. There is no official runtime or compiler for JavaScript as it is up to each browser to implement the languages runtime. When we create a JavaScript program/script for a web page we don't compile it and transfer a binary or bytecode file for the web page to execute, instead the browser takes the raw source code and interprets it[1].

JavaScript is a multi-paradigm language, mainly consisting of object-orientation and functional programming, with a dynamic type system. It is object-oriented in the way that most datastructures are represented through objects, and functional in the way that it has first-class functions, where functions can be free from a class and are treated as values which can be assigned to variables and sent around as parameters.

Where most object-oriented programming languages are class based, like Java, JavaScript is *prototype-based*. What this means is that the objects are not class instances, but are rather "instances" of a prototype. What you would normally think of as a class instance in Java, is an object with a reference to a prototype object. These instances are created through constructor functions, which create an object and sets the prototype for the object.

An object in JavaScript is a "bag" of properties containing values, which are specified in the prototype constructor, and a reference to the prototype object it is an instance of. The prototype object is not special in any way, it is just another object that has contains values that can be commonly used by all objects with the same prototype, and can themselves have prototype objects. This is how inheritance works in JavaScript, chains of prototype objects, where the `Object` prototype is at the end of the chain, similar to how the `Object` class is at the top of the inheritance hierarchy in Java. The `Object` prototype has `null` as its prototype, and `null` does not have any prototype. When trying to access a member of an object the object itself is first checked, then its prototype, and the prototype's prototype and so on, following the chain of prototypes, until a match is found [**prototypechaining**], or until there is no more prototypes to follow. Since prototypes are just objects they can as with any other object be

[1] On a more technical level, JavaScript is generally just-in-time compiled in the browser.

changed at runtime or replaced by other prototypes.

In ECMAScript 2015 there was introduced a class-syntax, however this is just syntactic sugar for creating the prototype object, and the associated constructor. Extending a class with this syntax is as you would expect just defining the prototype for the prototype object.

ECMAScript Versions

ECMAScript versions are generally released on a yearly basis. This release is in the form of a detailed document describing the language, ECMAScript, at the time of release. New versions will most likely include some additions to the language, but never any breaking changes[2]. This is because the developer will not be able to control the environment on which the code will be executed since you can not be sure which ECMAScript version the client browser is using. Because of this lack of control over the runtime environment it is crucial that any pre-existing language features don't have breaking changes between versions.

Backwards Compatibility

With new ECMAScript versions comes new features, and it is up to each browser to implement these changes. As we mentioned earlier, we do not transfer a binary to the client browser, we transfer the source code. So when a JavaScript script uses a new ECMAScript feature it is not guaranteed to work with every client browser, since a lot of users might have older browsers installed, or the team behind the browser has not implemented the language feature yet. To deal with this a common practice in JavaScript development is to first transpile the source code before using it in a production environment. This transpilation step takes the source code and transpiles it into an older ECMAScript version. In doing this you ensure that more browsers will be able to run the script. This will rewrite the new language features, and often replace them with a function, called a *polyfill*. You can think of a polyfill as an implementation of a new language feature that you ship with your code. These polyfills help the developer regain some control over the runtime environment on which the code will be run, and ensure that the code will run on almost any browser as expected.

Some popular transpilers for JS to JS transpilation are Webpack and Babel, but you could also use the TypeScript compiler for this.

Node.js

As of the time of writing there are mainly two ways to execute JavaScript. You can run the program in the browser, as it was intended, or you can use a JavaScript runtime that runs on the backend, outside of the browser.

[2]There has been occasions where there has been minor breaking changes between ECMAScript versions, but these changes happen very rarely and the affected areas are often insignificant.

Node.js (henceforth simply referred to as Node) is the mainstream solution for the latter. Node is a JavaScript runtime built on the JavaScript engine, V8, used by Chrome. It is independent of the browser and can be run through a *CLI* (Command-Line Interface). One major difference from the browser runtimes is that Node also supplies some libraries for IO, such as access to the file system and the possibility to listen to HTTP requests and WebSocket events. This makes Node a good choice for writing networking applications for instance.

We will be using the Node runtime for our compiler since it gives us access to the file system, as well as enabling the compiler to be executed through a CLI, as is the norm for most compilers. The compiler will still also be available as a library.

2.2.2 What is TypeScript?

TypeScript is a superset of JavaScript. The language builds on JavaScript with the additions of static type definitions. TypeScripts type system is structural, which means that the type of an object is not bound to a name, such as with nominal typing, but rather the structure of the object, such as having an attribute `i`, which may be restricted to a number. The type system also offers some more advanced type features such as union types, where you can combine types into a new type. The new union type represents values that can be any one of the combined types. There are also similarly intersection types. These types combine other types into a new type, which is the intersection of the combined types.

All valid JavaScript programs are also valid TypeScript programs. Types in TypeScript can be optional, as the type inference is powerful enough to infer most types without writing extra code. The type-checking can be tailored to be stricter or leaner, where you can for instance disable features such as usage of `any`-types, which are a way for the programmer to bypass the type-check for certain values. TypeScript has full interoperability with JavaScript, so you can adopt the language without refactoring. If you are working with a JavaScript library, but you want the safety of types, there can often be found type declaration files written by the community in the DefinitelyTyped project [**tswebsite**].

Part II

The project

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 attributes (`->` 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 attributes. PTj, for historical purposes, used

two different operators for renaming classes and methods, however in more recent PT implementations, such as [**Isene2018**], 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 [**arrowfunction**], 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 attributes 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 [**with-statement**]. 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 [**tsspec**]. 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 [**estreespec**].

In figure **??** 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⟩ ⟨attribute rename block⟩
⟨attribute rename block⟩ ⊨ `(` ⟨attribute renamings⟩ `)` | λ
⟨attribute renamings⟩ ⊨ ⟨rename⟩ | ⟨rename⟩ , ⟨attribute 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

Listing **??**.

```
// 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 8: An example program with instantiation, renaming, and addition-classes in PTS vs. PTj

Chapter 4

Planning the Project

Before we start the implementation of our language we first need to do some planning. We know we are going to be creating a programming language, a superset of TypeScript with the addition of Package Templates. However, we might want to look at if creating a superset of TypeScript is the way to go, or if keeping it simple and extending JavaScript is a better call. We might also want to see if it is needed to create a language at all, or if we are able to create a TypeScript library which can achieve the functionality of PT instead. There are a lot of approaches we can take for implementing our language, so we will have to map out the requirements for our desired approach. Lastly we will have to look at all the different approaches we can take, and see which approach is right for this project.

This planning phase is crucial for the success of the project, as starting off on the wrong approach for the wrong language would set us back immensely.

4.1 TypeScript vs. JavaScript

When extending TypeScript you might be asking yourself if it is truly necessary, is it better to keep it simple and just extend JavaScript instead? This is something we need to find out before going any further with the planning of our project.

4.1.1 Type-checking Templates

One of the requirements of PT is that it should be possible to type-check each template separately. There is no easy way to type-check JavaScript code without executing it and looking for runtime errors. Even if some JavaScript program successfully executes without throwing any errors, we can still not conclude that the program does not contain any type errors. 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 safe. Because of this trait TypeScript is the better candidate for our language.

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

4.1.2 Renaming

Renaming is a hard task. In order to perform a rename we will need to find the declaration and all references to this declaration and rename these. Doing this at compile time would mean that we will have to implement a type system of sorts, since this will help us identify references. This is also one of the reasons for why TypeScript is a better candidate than JavaScript, as TypeScript is statically typed, meaning the type of a variable is known at compile-time, while JavaScript is dynamically typed, where the type of a variable is first known at run-time.

4.2 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).

4.3 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:

- Implementing as a library
- Making a preprocessor for the TypeScript compiler
- Making a compiler plugin/transform
- Making a fork of the TypeScript compiler
- Making a custom compiler

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

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

Listing 9: Example of defining a template in a library implementation.

4.3.1 Implementing PT as a TS Library

One of the first approaches we need to check out is if we are able to achieve the functionality of PT, without having to create a compiler. Creating a library would presumably be an easier task than having to create a compiler, and it would also make it easier to use, as the programmer would not have to install a compiler in order to get the PT functionality. 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

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. JavaScript 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 rules out both arrays and objects, as there is no way of referencing other members during definition of the array/object. Templates should therefore be defined as classes, where each member of the template is an attribute of the template class. In listing **??** we see an example of how this could be done.

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 and use the `delete`-op [1] to remove the old field. We can see an example of this in listing **??**.

[1] An operator in JavaScript for removing a property of an object. See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/delete.

```
class T1 {
  static A = class {
    i = 0;
  };
}


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

Listing 10: Example of renaming a template class

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. Listing **??** shows how this can be a problem, where the function `f` of class `X` would fail at run-time due to it not being able to find class `A`.

Attribute renaming would most likely be possible in a similar manner, where we could change the prototype [2] of the class. Seeing as we are not able to fully rename classes by doing this we will not be looking further into this.

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 **??**.

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. We could possibly avoid circumvent this by getting the textual representation of the class, through the class' `toString`, and then use `eval` to evaluate the class declaration.

Merging Classes

For merging of types you would use the built-in declaration merging [**declerationmerging**]. Implementation merging is also possible because JavaScript has open classes. For implementation merging you would create an empty class which has the type of the merged declarations, and

[2] The prototype of a class is an object which objects of the class inherit their methods from. See https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes.

```
    // Type-safe template declaration
    class T1 {
        static A = class {
            i = 0;
        }
        static X = class {
            f() {
                return new A();
            }
        }
    }

   // Renaming
    const classRef = T1.A;
    T1.B = classRef;
    delete T1.A;

    // Trying to use the template after renaming
    const x = new T1.X();
    x.f(); // ReferenceError: A is not defined
```

Listing 11: Example showcasing the problems of renaming classes in a template in the library implementation.

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

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

Listing 12: Example of instantiating a template

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 [**mixinjs**].

Conclusion

Since we are not able to support renaming fully we will not be able to implement PT as a library for TypeScript. Because of this we will have to find another approach for the project.

4.3.2 Preprocessor for the TypeScript Compiler

Could we implement the PT specific features in a preprocessor? In order to understand this we need to understand what a preprocessor is. A preprocessor is often used in combination with a compiler, where the preprocessor often prepares the source files for compilation, such as removing comments, expanding macros (such as `#include` in C), and other smaller transformations. One distinct difference between a preprocessor and a compiler is that the preprocessor works on the source code as a piece of text, with no knowledge of the underlying programming language, while the compiler often has knowledge about the language of the program. This means that the compiler is able to perform more advanced tasks, such as semantic analysis.

So the question becomes, can we transform a PTS program to TypeScript by just doing textual transformations, and no semantic analysis. We would most likely be able to implement parts of PT with a preprocessor such as simple instantiation without renaming. However, as we mentioned in section **??** we will need to do some type-checking in order to find the correct references when renaming, we can't just rename everything that is textually equal. This means that we will need something more advanced than a preprocessor to implement the features of PT.

4.3.3 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" [**tscplugin**]. However, there are alternatives that do enable compile time plugins/transformers;

- ts-loader [**tsloadergithub**], for the webpack ecosystem
- Awesome Typescript Loader [**awesometypescriptloadergithub**], for the webpack ecosystem.
- ts-node [**tsnodegithub**], REPL/runtime

Unfortunately all of the above do not support adding custom syntax, as they only work on 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.

4.3.4 Babel plugin

Babel isn't strictly for TypeScript, but for JavaScript, however there does exist a plugin for TypeScript in babel, and we could write a plugin that depend on this TypeScript 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.

4.3.5 TypeScript Compiler Fork

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 could prove a hard task 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 might presumably 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.

4.3.6 Making a Custom Compiler

Making a custom compiler for PTS might seem like a hard task, but let us dig deeper into what this entails. Firstly we need to consider what the target should be. Normally a compiler would output some sort of byte code, like Java byte code in the Java compiler. Many compilers also produce native code. Native code is pretty much out of the image for our implementation as we still want to stay in the same ecosystem, namely the browser. We could possibly also produce WebAssembly byte code, however there are a lot of constructs in TypeScript/JavaScript that do not translate to WebAssembly, such as working with the DOM. Since both of these are out of the picture we could either produce TypeScript or JavaScript. Producing TypeScript is possibly the easiest way to go, as most of PTS is TypeScript. And producing TypeScript also means that

we could run the resulting program through the TypeScript compiler to produce JavaScript.

Having TypeScript as the target for our compiler also means that we can ignore most parts of the language and mainly focus on the PT specifics. The rest of the language can be outputted pretty much as is, since our language will be a superset of TypeScript.

4.4 Conclusion

While creating a Babel plugin might be a good approach, it would seem that creating a custom compiler is the safer bet. Creating a custom compiler will make it more accessible to most, as setting up a babel project is harder than running a compiler.

With the planning done we can jump into the implementation.

Chapter 5

Implementation

In this chapter we are going to look at the implementation for our compiler for PTS, as described in chapter **??**. Before looking at the implementation we will first be discussing the methodology used during development.

5.1 Methodology

When tackling a project of this magnitude it is important to have a proper methodology for development. During the development phase of this project I have had a strong focus on using agile techniques, where I have filled the role as both product owner and developer. This agile software development has aided me in discovering new requirements as the project moves forward, and re-adjusting to these new requirements. I have actively used a Kanban board throughout development to help keep track of tasks and goals.

The compiler was made in an iterative manner. For each iteration I would start off by implementing a new feature, and then put on the product owner hat and test out the compiler. While working as product owner I try to understand how I would like to use the language and what requirements I have for the language. This often leads to re-adjusting the requirements.

I started off by creating a rough MVP (Minimum Viable Product), only implementing the most basic functionality, which comprised declaration of packages/templates and simple instantiation. This MVP made me understand the project and requirements better, and also gave the project some new requirements. After the initial iteration I decided to adopt a test-driven development approach. I made tests for the features I had already implemented and then continued to make tests for the next functionality goal. This was done in order to gain more confidence in the compiler, as well as helping me spot any erroneous code earlier rather than later, which makes fixing it less costly. All of this resulted in a better development cycle, making refactoring and implementation of new features a breeze. When adding new features or refactoring some tests will undoubtedly fail, and before moving on I made sure that all the tests were passing again.

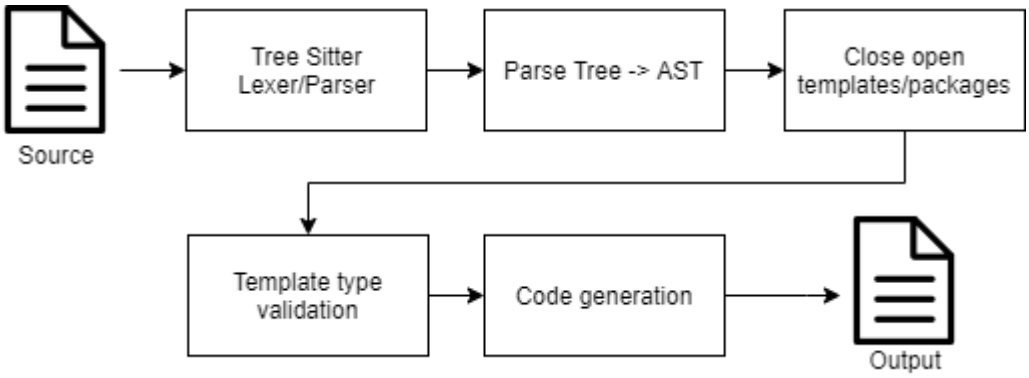

Figure 5.1: Overview of the compiler

5.2 Compiler Architecture

Our compiler consists of the following parts:

- Lexing and parsing
- Parse tree transformation
- Type checking packages/templates
- Code generation

An overview of our architecture can be seen in figure **??**. The first part of the compiler, namely the lexing and parsing will take a source file and transform it into a parse tree. Our compiler will then take this parse tree and transform it into a simpler abstract syntax tree (AST). This AST will then be used to perform the PT transformations. We will then use this AST to close any open packages and templates, and finally use the transformed tree for code generation. After all packages and templates have been closed we can type-check each individual package/template to validate the type-safety of our program. Given a valid type-safe program we can then move on to code generation. The target language for our code generation will mainly be TypeScript, however we will also offer to transpile the TypeScript into JavaScript.

5.3 Lexer and Parser

5.3.1 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 terms of functionality, so that we can more easily find the right tool for the task.

As we talked about in section **??**, 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 be forced to rewrite the entire TypeScript grammar, as well as keeping it up-to-date.

Because our language will be extending TypeScript we would like to utilize the TypeScript compiler as much as we can. The TypeScript compiler will help us perform the type-checking for our compiler, as well as producing JavaScript output. Therefore we need to be able to interact with the TypeScript compiler somehow. The TypeScript compiler has two main interfaces for interaction, through the command-line interface or using the compiler API. Optimally we would like to use the compiler API as this is the easiest way for us to perform type-checking and compilation. The catch is however that the only supported languages for the compiler API is JavaScript and TypeScript. Therefore a desired attribute for our choice of parser generator is that it offers a runtime library in either JavaScript or TypeScript, so that all of our implementation can be done in the same language, and not have to work with command-line interfaces programmatically.

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 [**Terence2012**].

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 [**Terence2012**]. 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 [**bison**].

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 do 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 [**tree-sitter**]. 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.

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

Listing 13: Snippet from the PTS grammar, where we override the `_declaration` rule from the TypeScript grammar, and adding two additional declarations.

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 **??**, where we override the `_declaration` rule from the TypeScript grammar, to include the possibility for package and template declarations.

5.4 Transforming Parse Tree to AST

5.4.1 The AST Nodes

Tree-sitter is a parser-generator written in Rust and C. 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* (Native Abstractions for Node) instead of the newer and more stable *Node-API*. 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

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

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 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.

5.4.2 Transforming

I chose to do the transforming immutable, and in order to do this we have to traverse the parse tree depth first and create nodes postfix. Tree-sitter provides pretty nice functionality for traversing the parse tree through cursors. With a tree cursor we are able to go to the parent, siblings, and children easily. Using this we visit each node and produce an AST node as described in the section above.

5.5 Closing Templates

The task of closing open packages and templates is what most of the implementation is focused around. It is the task of performing the declared instantiations and altering the declared classes through `addto`-statements. This step is cruicial as it will make each package/template a valid TypeScript program and make the program ready for code generation.

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 correctly scoped AST.
2. Find all identifiers, member expressions, class declarations, etc., and replace them with *reference nodes*[2]

[2]Reference nodes are AST nodes that contain a pointer to the class or attribute 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 **??**.

3. Perform the rename.

4. Transform the scoped AST with reference nodes back to the original AST.

5. Merge class declarations and apply `addto` statements.

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.

5.5.1 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 attributes we first need to create correct scopes in which the renaming can be applied to. We start off 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.

Scope is represented through the `Scope` class. 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 attribute or class name, to a reference to either a variable (this covers both class attributes and other variables used throughout the program) or a class. Looking up symbols in the symbol table will always start in the called scope, looking for any references matching the given name. If we don't find any references with the given name we propagate the lookup to the parent scope. Given further misses in the symbol table means that we will eventually reach the root scope, and if the root scope also doesn't contain any references then we fail the compilation and inform the programmer that there is a reference error in the program.

Having several layers of scope enables us to correctly handle shadowed variables, or parallel declarations with equal naming.

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`

We start of with a root scope which is given to the root node of the template body. We then traverse the tree and give every node a reference to the scope. When we reach one of the aforementioned nodes that should have its own scope, we create a new scope, with the current scope as the parent scope. This new scope is then given to all nodes beneath this node, until we reach another node in the list above.

At the end of this traversal we have an AST where every node has a reference to the scope its in.

5.5.2 Transforming Nodes to References

In order to rename a class or an attribute we need to find all references to the class or attribute. That is what this step is supposed to do, find all references and group these together, so that a rename can be performed. A reference in our program can either be a variable or a class reference.

A variable reference can be a class attribute or any other variable declaration. Even though we call it a variable reference this does also cover functions, however both variables and functions share the same namespace, so there is essentially no difference between them, unlike Java where methods and variables can have the same name. These references are represented by a `Variable` class. The class contains the name of the attribute, and optionally what type it is. The type of a variable is set through a very simple type-inference (relative to TypeScripts type system), where we only look at if there is a `new`-expression assigned to the variable at declaration. If the variable is initialized with a class instantiation then the variable is a object of that class. This is not necessarily true, as the variable might have an explicitly declared type, however we currently ignore these in the implementation.

Class references are all references to the class. This can be the class declaration itself, instantiations of the class, etc. The `Class` class is a representation for classes. They are an extension of the `Variable` class, so they can store the name of the class, however they also have references to all the variables that are instances of the class, and optionally a superclass, which is a reference to another `Class` instance.

When a reference is created this is registered in the scope where it occurred. We can then later lookup the reference instance in the symbol table of the scope.

With an understanding of what a reference is, let us look at how we can transform the AST nodes that are references. Transforming nodes into references mainly consists of two steps.

- Transforming the declarations
- Transforming references

Transform Declarations

When transforming declarations we both create the `Variable` or `Class` instance, register them in the scope they were found, and transform the identifier in the declaration to a reference. This reference is a special AST node. It is represented by the `RefNode` class. This class contains the same fields as other scoped AST nodes, `type` (which is always `"variable"` for reference nodes), `text`, `children` and `scope`, as well as two extra fields. The first field is a reference to the `Variable` or `Class` instance that the `RefNode` is a reference to. The second is a field containing the original type of the node, which are used when transforming the `RefNodes` back to the original AST.

The task of transforming declarations requires several passes through the AST. In total we pass through the AST three times when transforming declarations, once for class declarations, once for class heritage, and once for class attribute declarations. This is because in order to transform some declarations we need to have access to others. To understand this we can look at the difference between the first and second pass.

During the first pass through the AST we register and transform all class declarations, creating the `Class` instances. Listing **??** shows an example of how a class declaration will be transformed in this step. In the listing we can see that the `"type_identifier"` node has been transformed to a `RefNode` instance.

For the second pass we register class heritage. The second pass updates the `Class` instances' superclasses, by looking them up in the scope. If we had done these two steps in the same pass through the AST we might have ended up in situations where we can't find the superclass instance, as we might not have reached the declaration yet.

In addition to transforming class declarations, and their heritage, we also transform attribute declarations. This could have been done in the same pass as with class heritage, however it was separated to make it more understandable. During this transformation we also check if there is a `new`-expression in the assignment, in order to possibly register the type of the variable. The current implementation only supports public field definitions. Other declaration types could be supported in the same manner as public field definitions, however due to a lack of time these were not implemented.

Transform References

Transforming references also has several passes, more precisely two passes. The first pass is to transform the `this` keyword. This does not have to be done in a separate pass, however it does simplify the task since we do not have to have extra special cases in the second pass. The `this`-nodes are transformed to a `RefNode` like other references, however here we can set the type of the variable without working out the type of it, as `this` will always have the type of the class it is contained in.

In the second pass is where the rest of the references are transformed.

```
// Before transformation
{
    type: 'class_declaration',
    text: 'class A ...',
    scope: <Scope 1>,
    children: [
        {
            type: 'class',
            text: 'class',
            scope: <Scope 1>,
            children: []
        }, {
            type: 'type_identifier',
            text: 'A',
            scope: <Scope 1>,
            children: [],
        }
        ...
    ]
}

// After transformation
{
    type: 'class_declaration',
    text: 'class A ...'
    scope: <Scope 1>,
    children: [
        {
            type: 'class',
            text: 'class',
            scope: <Scope 1>,
            children: []
        }, {
            type: 'variable',
            text: '',
            scope: <Scope 1>,
            children: [],
            origType: 'type_identifier',
            ref: <Class 1>
        },
        ...
    ]
}
```

Listing 14: AST of a class declaration of class A before and after transforming the references. The values surrounded by angle brackets are references to `Scope`/`Class` instances.

Currently the implementation will transform `new` expressions, member expressions, and type annotations.

`new` expressions are pretty simple to transform, similar to class declarations, we can just find the identifier node and replace this by the referenced class. We find the class by looking it up in the scope. It is worth noting that this will not rename any generic type parameters.

Member expression are a bit more troublesome. A member expression are expressions such as `a.i` or `this.b.x`. This is the main reason for why we transformed `this`-expression separately, since they can, and often do, occur in member expressions, and this saves us from checking another special case. A member expression consists of the object and the property. The property is always an identifier, while the object on the other can be one of three things, a `RefNode`, a member of an identifier, or another member expression.

The most common member expression we can transform is a member of a `RefNode`. The `RefNode` here could be something like `this` or a variable, `a`. Here we use the type of the `Variable` instance in order to find the attribute that the member expression is referencing. If the attribute does not exist we throw an error, informing the programmer that there is a reference error. Else we can create a `RefNode` for the property part of the member expression.

The second scenario for member expressions are when there is a nested member expression. When this is the case we can recursively call the function on the nested member expression. After the nested member expression has been transformed we can get the property of the nested member expression, and use this to transform the property of the initial member expression, similar to how we did it for members of `RefNode` above.

The final member expression is a member expression of an identifier. This can happen if we are dealing with something that can't be renamed, such as `console.log`, or if we stumble upon an identifier that has not been picked up during previous transformations. We will in both cases try to see if we are able to find the object by looking it up in the scope. If we can't find it we know that its something that can't be renamed. If we do find it we treat it like a member of a `RefNode`.

Type annotations consists of the following:

- Type identifiers
- Generic types
- Predefiend types
- Type predicates

TODO: Skriv om transformation av type annotations

5.5.3 Performing the Rename

With all declarations and references transformed into reference nodes we can now perform the rename. For each class rename we can simply lookup the old class name in the root scope and change the reference to the new class name. For class attribute renames we do the same just for the enclosing classes scope.

5.5.4 Going back to the original AST

reference -> original with new names remove scope

5.5.5 Merging Class Declarations

Check if can be merged, non overlapping members. Override and merge addto Preserve hierarchy

5.6 Type-checking of Templates

After the previous step we have a program where all packages and templates are closed, meaning that the bodies of these should contain plain TypeScript. Because of this we can relatively easily type-check each package/template individually by using the TypeScript compiler and its compiler API. In order for us to type-check our pacakges/templates we will have to transform the bodies of the packages/templates into a textual format. This will be done by applying our code generation implementation, which we will discuss in section **??**, to the body of the package/template we are currently working on. Running code generation on the package/template body will give us a TypeScript program. This program can then be passed on to the TypeScript compiler for type-checking. We will make the TypeScript compiler transpile the program to JavaScript without emitting any output. This will effectively type-check the program.

If the TypeScript compiler throws any errors we can log this for the user of our compiler to fix, and inform in which package/template this error occurred. If no errors were thrown we have a type-safe package/template, and we can then proceed to the next step in our compilation.

5.7 Code Generation

After performing these steps we can finally produce the output. Producing TypeScript output is a pretty simple task. By traversing the AST we can concatenate the text from each leaf node with whitespace between each leaf node's resulting textual representation. This will produce quite ugly, unformatted code, but as long as the contents of the closed packages and templates are valid typescript programs this will also produce a valid typescript program. In order to make it more readable we perform an

extra step before writing the output to the specified file, a formatting step. There are a lot of TypeScript formatters out there, but we will be using *Prettier*[3] for our implementation as it is relatively simple to use. Running our produced source code through the Prettier formatter produces a nicely formatted, readable output.

The TypeScript output is probably the best target for understanding what the PT mechanism does, however it might not be the best output for production use. Since the only officially supported language for the web is JavaScript we will also be implementing this as a target for code generation. This is fortunately also a relatively simple task, as we already depend on the TypeScript compiler, and since we are able to produce TypeScript source code, we can use this to produce JavaScript output.

5.8 Notes on Performance

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

5.9 Testing

Testing has been an essential part throughout the development of the compiler. After the initial prototype of the compiler was running I continued onward with test driven development. This allows me to write up tests for all the features of the language, and run them concurrently as I make the changes to the implementation.

The cycle

5.9.1 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*[4].

5.9.2 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] A code formatter for the web ecosystem. See https://prettier.io/.

[4] S-expressions are textual representations for tree-structured data. See **[sexprs]** for additional information and examples

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

Part III

Result

Chapter 6

Does PTS Fulfill The Requirements of PT?

This thesis is concerned about implementing Package Templates in TypeScript. However, in order to determine to what degree we have actually implemented PT or just created something that looks like it, we have to understand what the requirements of PT are, and if we are meeting those requirements. We will therefore in this chapter look at the requirements as described in [**jot**]. After getting an understanding of the requirements we are going to look at how our implementation holds up to them.

6.1 The Requirements of PT

In [**jot**] the authors discuss requirements of a desired language mechanism for re-use and adaptation through collections of classes. They then present a proposal for Package Templates, which to a large extent fulfills all the desired requirements. These requirements can therefore be used to evaluate our implementation and determine whether our implementation can be classified as a valid implementation of Package Templates.

The requirements presented in the paper were the following:

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

In order to get a better understanding of what these requirements entail we will have to dive a bit deeper into each requirement.

6.1.1 Parallel Extension

The parallel extension requirement is about making additions to classes in a package/template, and being able to make use of them in the same collection. What this means is that if we are making additions to a class,

```
template T {
    class A {
        ...
    }

    class B {
        A a = new A();
        ...
    }
}

package P {
    inst T;

    addto A {
        void someMethod() {
            ...
        }
    }

    addto B {
        void someOtherMethod() {
            a.someMethod();
        }
    }
}
```

Listing 16: Example of parallel extension in PTj. Here we make additions to both `A` and `B` in our instantiation in package `P`, and we are able to reference the additions done to `A` in our addition to `B`. This is done without the need to cast `A`, as if the additions were present at the time of declaration.

then we should be able to reference these additions in a declaration or in an addition to a separate class within the same package/template, without needing to cast it. We can see an example of this in listing **??**, where an addition to class `B` is referencing the added method of class `A`. The order of additions does not affect the parallel extension. We could just as easily switched the positions of the additions around in this example.

6.1.2 Hierarchy Preservation

PT should never break the inheritance hierarchy of its contents. If we have a template with classes `A` and `B`, and class `B` is a subclass of class `A`, then this relation should not be affected by any additions or merges done to either of the classes. That is if we make additions to class `B` it should still be a subclass of class `A`, and any additions made to class `A` should be inherited to class `B`. Even if we make additions to both class `A` and `B`, then `B` with

additions should still be a subclass of class `A` with additions.

6.1.3 Renaming

The renaming requirement states that PT should enable us to rename the names of any class, and its attributes, so that they better fit their use case.

6.1.4 Multiple Uses

PT should be allow us to use packages/templates multiple times for different purposes in the same program, and any additions or renamings should not affect any of the other uses. Each use should be kept independent of each other. This is an important requirement of PT as when we create a package or a template it is often designed to be reused. An example of this is the graph template we created in listing **??**. Here we bundled the minimal needed classes in order to have a working implementation for graphs. We then used this graph implementation to model a road systems, however we might later also want to reuse the graph implementation for modelling the sewer systems of each city, and this should not be affected by any changes we made to the graph template for our road system.

6.1.5 Type Parameterization

The requirement of type parameterization of templates works similar to how type parameterization for classes works in Java. Type parameterization in Java enables the programmer to assume the existence of a type during declaration of a class, and the actual type can be given when a new object of the class is created. Type parameters in Java can have constraints where they must extend another class or interface. Similarly, type parameterization in PT also enables the programmer to assume the existence of a type, however here the type parameter is accessible to the whole template. PT type parameters can also be constrained similar to Java, by giving a nominal type it should extend, however PT also allows for type constraining through structural types, by giving a structure the type should conform to.

In listing **??** we see an example of how type parameterization can be used to implement a list. Type parameterization in PT might feel similar to how you would use it in regular Java, however having the type parameter at the template level, where Java has it at class level, does have some advantages. One advantage of having the type parameter at template level is that you don't need to specify the actual parameter again after instantiation. At instantiation of the `ListsOf` template we can give i.e. a `Person` class, containing some information about a persons name, date of birth, etc., as the actual parameter, and then we would not have to keep specifying the actual parameter for every reference. Another advantage of using type parameters at the template level is that the type parameter can be used by all classes in the template. If we wanted to implement this in

```
template ListsOf {
    required type E { }
    class List {
        AuxElem first, last;
        void insertAsLast(E e) { ... }
        E removeFirst() { ... }
    }
    class AuxElem {
        AuxElem next;
        E e; // Reference to the real element
    }
}
```

Listing 17: Modified example from **[jot]** where type parameterization is used to create a list implementation.

Java we would either have to have type parameters for both classes, or the `AuxElem` class would need to be an inner class of the `List` class.

6.1.6 Class Merging

PT should allow for merging two or more classes. When merging classes the result should be a union of their attributes. If we merge two classes `A` and `B`, it should be possible to reach all of `B`'s attributes from an `A`-variable, and vice versa.

6.1.7 Collection-Level Type-Checking

The final requirement is collection-level type-checking. This requirement is there to ensure that each separate package/template can be independently type-checked. By having the possibility to type-check each pacakge/template we can also verify that the produced program is also type-safe, as long as the instantiation is conflict-free.

6.2 PTS' Implementation of the Requirements

With a proper understanding of the requirements of PT we can examine our implementation and see whether our implementation fulfills these requirements. For each requirement we will be looking at a program in PTS which showcases the requirements, and the resulting program after compilation.

6.2.1 Parallel Extension

To understand how this requirement can be fulfilled it is important to understand how the requirement could fail to be fulfilled. A failure to fulfill the requirement would be that making additions in parallel would

fail to compile, or create an otherwise incorrect program. Failure to compile might of course not always be a bad thing, there are certain scenarios where we do want the compiler to throw an error. There are mainly two scenarios where we would like the compilation to fail for additions, trying to make an addition to a non-existent class, and trying to reference non-existent attributes in a class.

The first scenario where our compiler should fail is when we are making additions to a non-existent class. This will be caught in the class merging part of our compiler. In the class merging part of the implementation the compiler will group all class declarations and additions by the class name. If there is a group containing only additions then it will fail, as we have no class to make additions to.

The second scenario is when we are trying to reference non-existent attributes in a class. An example of this can be seen in listing **??**. This example will fail during the type-checking of our pacakges/templates, as discussed in **??**. Our approach for dealing with this is pretty much by not dealing with it, and instead assume that everything is okay at this stage of the compilation. We will then instead discover any inconsistencies in the type-checking stage of the compiler. In the aforementioned listing it is of course pretty easy to examine class `A` to see if it contains an attribute `h`, however it might not always be this easy. In a more complicated example where we are in the process of merging several classes and additions it might prove a tougher task to see if the addition would result in a type-safe class. So as long as we are able to perform the addition we can instead assume that it is working as intended and instead let the TypeScript compiler check if it is type-safe, after the addition has been performed.

Now that we understand when we want compilation to fail let us look at where we do not want it to fail, when we have a valid parallel extension. One such way it could fail is if we tried to check if the addition contains any invalid references or type errors. This could commonly happen if we are trying to check the addition's references to the declared class. However, as discussed above, checking if a reference to an attribute is valid is quite tricky, and in our implementation instead leave this up to the TypeScript compiler. By doing this we will not incorrectly throw any false-negatives when it comes to parallel extensions. This approach does unfortunately come with some downsides. By not addressing the issue at the addition stage it makes it harder to give informative error messages when invalid references do occur, however this was a tradeoff that was beneficial for this project.

6.2.2 Hierarchy Preservation

In order to fulfill the hierarchy preservation requirement we have to preserve all super-/subclass relations after additions and merges have been applied. Listing **??** shows a program, and the resulting TypeScript program after compilation, which fulfills the requirement of hierarchy preservation. This one example fulfills the requirement as class `B` is still a subclass to class `A` after both a merge and an addition is made to `B`. As

```
template T {
    class A {
        function f() {
            return 1;
        }
    }
}

package P {
    inst T;
    addto A {
        function g() {
            this.h();
        }
    }
}
```

Listing 18: An example showing a program that should fail during compilation, where we are trying to reference a non-existent attribute, `h`, in an addition to class `A`.

we talked about briefly in section **??** when we merge classes we make sure to also merge their class heritage, combining the extending classes and implementing interfaces of the different classes. This means that we might end up with instances where we are extending multiple different classes, however this will then be picked up in the type-checking stage of the compiler. If we had not merged class heritage, we could have ended up breaking the inheritance hierarchy in the aforementioned listing, as we could have for example ended up with class `C`'s heritage, which does not have a superclass. Because of the heritage merging we can with confidence say that we have fulfilled the requirement of hierarchy preservation, as we always preserve all super-/subclass relations.

6.2.3 Renaming

In order to be able to fulfill the renaming requirement our implementation should be able to rename classes and their attributes. This renaming should result in a program where not only the declarations have been renamed, but also all references. Listing **??** shows an example program of renaming in PTS, where we are renaming a class, `A`, and the class' attribute, `i`. We can see in the resulting program that the identifier in the declaration of both the class and the attribute has changed, but so has the references to these in the constructor of the class and references in another class, `B`. The renaming has also not wrongly renamed other references that are similar in naming, such as the parameter of the constructor of class `A`. This simple example works as expected, however there are also scenarios where the renaming does not work as expected.

```
// PTS
template T1 {
    class A {
        i = 0;
    }

    class B extends A {
        f() {
            return this.i;
        }
    }
}

template T2 {
    class C {
        j = 0;
    }
}

package P {
    inst T1;
    inst T2 { C -> B };
    addto B {
        k = 0;
    }
}

// Resulting program
class A {
    i = 0;
}

class B extends A {
    f() {
        return this.i;
    }
    j = 0;
    k = 0;
}
```

Listing 19: Example showcasing a program where the super-/subclass relation between classes `A` and `B` is preserved after additions and class merging have been applied. We can see the resulting TypeScript program at the bottom of the listing, where the semantics are as expected.

```
// PTS
template T {
    class A {
        i = 0;
        constructor(i: number) {
            this.i = i;
        }
    }

    class B {
        a = new A();
        function f() {
            return a.i;
        }
    }
}

pack P {
    inst T { A -> X (i -> j) };
}

// Resulting program
class X {
    j = 0;
    constructor(i: number) {
        this.j = i;
    }
}

class B {
    a = new X();
    function f() {
        return a.j;
    }
}
```

Listing 20: Example of renaming in PTS.

Since TypeScript is a structurally typed language we can run into scenarios where a rename could and should result in an invalid program. Listing **??** showcases this problem. The problem arises in the `a` attribute of class `B` in template `T`. This has been declared to be a variable expecting an object where there exists an attribute `i`. `a` is initialised with an object of the class `A`. This is fine in template `T`, however when `T` is instantiated in package `P`, and `A`'s attribute `i` is renamed to `j`, this is no longer the case. Since an object of `A` no longer contains an attribute `i`, this is no longer valid.

The aforementioned listing shows how we would like the compilation result to look like, however this is not the result the current implementation produces. TypeScript's type system can be quite complicated, and due to a lack of time I chose to ignore most of the type declarations. The current implementation would have treated the attribute `a` as an `A`-variable, since it is being initialized with an object of `A`, and therefore have renamed later references to `a.i` to `a.j`. It was more important to get a working prototype, than support all scenarios with different type signatures. This is something I would of course have liked to take into consideration if I had more time to spend on the implementation. Deciding the type of variables is something that possibly would have come for cheaper if I had opted for a fork of the TypeScript compiler as my approach. This is something we will come back to in **??**.

6.2.4 Multiple Uses

In order for this requirement to be fulfilled we should be able to re-use a template several times, with different renamings and additions while the different instantiations stay independent of each other. This was something I paid extra attention to during implementation, not just to fulfill the requirement, but to avoid bugs. I solved this by making sure that while transforming the AST this would be done in an immutable fashion. In order to test this we will be creating a simple program where we instantiate the same template more than once and see if the resulting program is as expected. The program can be seen in listing **??**. The program comprises a template `T` with a single class, `A`, with an attribute `i`. This template will then be instantiated three times, where we first will be renaming the class and field, then instantiate without renaming, and finally instantiate it with just an attribute renaming. The expected program should have two classes, one class `B`, with an attribute `j`, and a class `A` where the two bottom instantiations should have created a merged class with attributes `i` and `x`. We can see from the resulting program after a successful compilation that this is as expected.

6.2.5 Type Parameterization

The type parameterization requirement is something the implementation does not fulfill. This was not implemented due to it not being prioritized. There is only so much time available during the span of a master thesis, and I chose to look at how the core of PT would fit into a structurally

```
// PTS
template T {
    class A {
        i = 0;
    }

    class B {
        a : { i : number } = new A();
        i = a.i;
    }
}

pack P {
    inst T { A -> A (i -> j) };
}

// Expected result
class A {
    j = 0;
}

class B {
    a : { i : number } = new A();
    i = a.i;
}
```

Listing 21: Example showcasing the problem of having renaming in a structural language. In class `B` we have an attribute, `a`, that expects an object that contains an attribute `i`. The attribute is initialized with an `A` object. This is fine in template `T` as `A` contains an attribute `i`, however when class `A`'s attribute is renamed in the instantiation in package `P` then an object of `A` is no longer valid as a value, since it no longer contains an attribute `i`. This is an instance where we can't just rename the references to `i`, since this reference isn't explicitly related to `A`.

```
// PTS
template T {
    class A {
        i = 0;
    }
}

pack P {
    inst T { A -> B (i -> j) };
    inst T;
    inst T { A -> A (i -> x) };
}

// Resulting program
class B {
    j = 0;
}

class A {
    i = 0;
    x = 0;
}
```

Listing 22: A program showcasing multiple uses in PTS, and the resulting program in TypeScript at the bottom.

```
template ListsOf {
    class E { }
    class List {
        AuxElem first, last;
        void insertAsLast(E e) { ... }
        E removeFirst() { ... }
    }
    class AuxElem {
        AuxElem next;
        E e;
    }
}
```

Listing 23: Example of a similar list implementation as in listing **??**, without the use of required types. Instead of giving a type for the required type we will have to merge the class `E` with the "actual parameter".

language like TypeScript, rather than on making sure it would be a fully fleshed out implementation of PT. Another reason for avoiding this is that much of type parameterization can be achieved through merging and renaming. Listing **??** shows an example of how you can use an empty class as a generic type implementation of lists, similar to the list implementation with required types in listing **??**. Required types do of course have a lot of advantages such as making it possible to constrain the type, and forcing the programmer to give an actual parameter for the type, which we are unable to do.

6.2.6 Class Merging

6.2.7 Collection-level Type-checking

6.3 Conclusion

While we do not fulfill every requirement, we do fulfill most of them. The current implementation might not be a full implementation of PT, but we can confidently say we have at least made an implementation of the core of PT for TypeScript. Not having a full implementation does mean that we might not be able to examine all the differences between our implementation and PTj, however we will be able to examine the common elements, which covers the most interesting parts. This allows us to explore how a mechanism like PT fits with the TypeScript language, and its potential utility.

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 are 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 are 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 is often defined as a record, and could for example be `{ name: string }`.

In listing **??** we can see an example of a nominally typed program in a Java-like language. Here `B` is a subtype of `A`, while `C` is not. This is due to nominally typed programs having the requirement of explicitly naming its subtype relations, through e.g. a subclass-relation. Because of this we can see that at the bottom of the listing the first two statements pass, since both `A` and `B` are of type `A`, while the last statement fails (typically at compile time), as `C` is not of type `A`.

In listing **??** we see a structurally typed program. This program also has the exact same declarations as in listing **??**, that is classes `A`, `B`, and `C` and the function `g`. In this program both type `B` and type `C` are a subtype of type `A`, since they both contain all members of type `A`. Not necessarily the same

```
// 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 24: Example of a nominally typed program in a Java-like language

implementation as in class `A`, but the same types as in type `A`. This is one of the sole differences between nominal and structural typing, types can conform to other types without having to explicitly state that they should. Type `C` is an example of this, while it does not have a subclass relation to class `A`, nor implement any common interface, it still conforms to the type of `A`. The result of this is that all three usages of function `g` are valid in a structural type system, while consuming `C` was illegal in the nominal example.

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

```
// 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 25: Example of a structurally typed program in a Java-like language

```
// 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 26: Example of subtype relations in nominal and structural typing, in a Java-like language. In the example of the nominal subtype we have to explicitly state the subtype relation, while in the structural subtype example the subtype relation is inferred from the common attributes.

languages through some clever representation techniques [**tapl**]. We can see an example of subtype relations in both nominal and structural type systems, in a Java-like language, in listing **??**. 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`.

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. It is also easier to check sub-type relations in nominal type systems, even though you might still have to do a structural comparison, you only have to perform this once per type [**tapl**].

7.1.2 Advantages of Structural Types

Tidier and More Elegant

Structural types carry with it all the information needed to understand its meaning. This is often seen as an advantage over nominal typing as the programmer arguably only has to look at the type to understand it, while in nominal typing you would often have to look at the implementation or documentation to understand the type, as the type itself is part of a global collection of names [**tapl**].

More General Functions/Classes

Malayeri and Aldrich performed a study (see [**malayeri**]) on the usefulness of structural subtyping. The study was mainly focused around two characteristics of nominally-typed programs that would indicate that they would benefit from a structurally typed program. The first characteristic was that a program is systematically making use of a subset of methods of a type, in which there is no nominal type corresponding to the subset. The second characteristic was that two different classes might have methods which are equal in name and perform the same operation, but are not contained in a common nominal supertype. 29 open-source Java projects were examined for these characteristics.

For the first characteristic the authors ran structural type inference over the projects and found that on average the inferred structural type consisted of 3.5 methods, while the nominal types consisted of 37.8 methods. While for the second characteristic the authors looked for types with more than one common methods and found that every 2.9 classes would have a common method without a common nominal supertype. We can see that from both of these characteristics that the projects could have benefited from a structural type system, as this would make the programs more generalized, and could therefore support easier re-use of code.

7.1.3 Disadvantage of Structural Type Systems

It is worth noting that the advantage of types conforming to each other without explicitly stating it in structural type systems can also be a disadvantage. Structurally written programs can be prone to *spurious subsumption*, that is consuming a structurally equal type where it should not be consumed. An example of this can be seen in listing **??**.

7.1.4 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 nominal and structural type systems. 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.

```
function double(o: {calculate: () => number}) {
    return o.calculate() * 2
}

const vector = {
    x: 2,
    y: 3,
    calculate: () => 4
}

const unintended = {
    calculate: () => {
        doSomeSideEffect();
        return 1;
    }
}
```

Listing 27: Example of spurious subsumption in TypeScript. Here the function `double` will consume an object that has a `calculate` attribute. The intended use is to consume something that does a calculation on the object and returns a number which will be doubled, while the unintended use example in this case does some unexpected side-effect and returns a number as a status code. The unintended object can be consumed by the `double` function as it is conforms to the signature of the function, while in a nominally typed system this can be avoided to a much larger extent.

7.1.5 Which Better Fits PT?

With the addition of required types in PT the language mechanism now has to utilize structural typing, independent of the underlying language's type system. Using structural typing was seen as a necessity for required types as this would give the mechanism the its required flexibility. One could therefore argue that a structural type system is a better fit for Package Templates as it would remove the confusion of dealing with two different styles of typing in a single program, and make the language mechanism feel more like a first class citizen of the host language.

Another advantage for having structural typing for PT is that it can help strengthen one of the main concepts of Package Templates, namely re-use. As we learned from the study of Malayeri and Aldrich, structural typing can make programs more general which makes them more prone to re-use.

However, there are also some quite significant problems with having PT in a structural language. Renaming is especially something that might not fit nicely into a structurally typed language. In listing **??** we see an example of a program that breaks after renaming an attribute. The renaming resulted in class `Consumable` no longer conforming to the signature of function `f` in class `Consumer`. PT does not support changing the signature of functions so there is no way for us to be able to make the `Consumable` class conform. In order to avoid running into this problem we might consider disallowing inline type declaration. This would force listing **??** to give an interface as the type for the formal parameter, `consumable`. We could then also rename the members of the interface in order to once again make the `Consumable` class conform to the signature of function `f`.

It is worth noting that the problem of renaming causing programs to break is not something unique to structural typing, this can also occur in nominally typed programs. Listing **??** showcases a program that breaks after renaming. In this listing we see that after renaming method `f` of class `A` the class no longer fulfill the requirements of the implementing interfaces `I`, as `I` expects a method `f` to be present, which it no longer is. We could resolve this by also performing an equal renaming to interface `I`. Although it is a problem in nominal PT as well, it is less so than with structural PT, since it can be resolved with just an additional renaming, and due to the relation being explicitly stated we can also give an error message during compilation notifying the programmer of this inconsistency. In structural PT we would have to disallow inline-types in order to make the problem more solvable for the programmer, but due to the relation between the class and interface not necessarily being explicitly stated it would be harder to give a sensible warning for the programmer.

With the discussed general advantages and disadvantages of structural and nominal type systems, and the points brought forward in this section we can see that both styles of typing have strong use cases with PT. A nominal type system in PT would seemingly lead to less problematic renaming scenarios, while a structural type system in PT would arguably fit better with the overall theme of PT, flexible re-use.

```
template T {
    class Consumable {
        i = 0;
    }

    class Consumer {
        function f(consumable : {i: number}) {
            ...
        }
    }
}

pack P {
    inst T { Consumable -> Consumable (i -> j) };
}
```

Listing 28: Example of how using renaming in PTS might break a program. After renaming the field `i` to `j` the class `Consumable` is no longer consumable by function `f` in class `Consumer`.

```
template T {
    interface I {
        void f();
    }

    class A implements I {
        void f() { ... }
    }
}

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

Listing 29: Example of how using renaming in PTj might break a program. After renaming the method `f` to `g` the class, `A`, no longer conform to the implementing interface `I`.

7.2 Attribute Renaming

Don't need full signature in PTS, because no overloading, etc. Might be easier to understand PTS version?

Chapter 8

Results

8.1 Conclusion

8.1.1 Approach

Did I choose the right approach? TypeScript compiler fork might have been easier.

8.2 Future Work

8.2.1 Diamond Problem

essay

8.2.2 Improve the Compilers Error Messages

As I mentioned shortly in subsection **??** tree-sitter does have support for giving position of a syntax node, and this could be utilized to produce better error messages.

```
template FetchJson {
    class FetchJson extends Component {
        componentDidMount() {
            fetch(this.props.url)
                .then(response => response.json())
                .then(data =>
                    this.setState(state => ({...state, data})))
        }
    }
}

pack Pokemon {
    inst FetchJson { FetchJson -> Pokemon };
    addto Pokemon extends Component {
        render() {
            if(this.state.data === undefined)
                return 'Loading...';

            const name = this.state.data.name;
            const image = this.state.data.sprites.front_default;
            return (
                <div>
                    <img src={image} />
                    <h1>{name}</h1>
                </div>
            )
        }
    }

    class App extends Component {
        render() {
            <Pokemon
                url="https://pokeapi.co/api/v2/pokemon/ditto" />
        }
    }
}
```

Listing 30: Real world example of PTS being used in React. In this example we create a component for fetching JSON when the component has been mounted, and then we re-use this functionality in our Pokemon-component.