UNIVERSITY OF BERGEN
DEPARTMENT OF INFORMATICS

Title of your master thesis

Author: Your name
Supervisors: Name of supervisors

Author: Your name
Supervisors: Name of supervisors

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

April, 2024

Abstract

Lorem ipsum dolor sit amet, his veri singulis necessitatibus ad. Nec insolens periculis ex. Te pro purto eros error, nec alia graeci placerat cu. Hinc volutpat similique no qui, ad labitur mentitum democritum sea. Sale inimicus te eum.

No eros nemore impedit his, per at salutandi eloquentiam, ea semper euismod meliore sea. Mutat scaevola cotidieque cu mel. Eum an convenire tractatos, ei duo nulla molestie, quis hendrerit et vix. In aliquam intellegam philosophia sea. At quo bonorum adipisci. Eros labitur deleniti ius in, sonet congue ius at, pro suas meis habeo no.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Intro goes here

1.1   Background

1.1.1   Figures

Chapter 2

Background

2.1   Proposals

A proposal for EcmaScript is a suggestion for a change to the language. These changes come with a set of problems that if the proposal is included as a part of EcmaScript, those problems should be solved by utilizing the additions contained within the proposal.

Chapter 3

Collecting User Feedback for Syntactic Proposals

The goal for this project is to utilize users familiarity with their own code to gain early and worthwhile user feedback on new syntactic proposals for EcmaScript.

3.1 The core idea

Users of EcmaScript have a familiarity with code they themselves have written. This means they have knowledge of how their own code works and why they might have written it a certain way. This project aims to utilize this pre-existing knowledge to showcase new proposals for EcmaScript. Showcasing proposals this way will allow users to focus on what the proposal actually entails, instead of focusing on the examples written by the proposal author.

Further in this chapter, we will be discussing the *old* and *new* way of programming in EcmaScript. What we are referring to in this case is with set of problems a proposal is trying to solve, if that proposal is allowed into EcmaScript as part of the language, there will be a *new* way of solving said problems. The *old* way is the current status quo when the proposal is not part of EcmaScript, and the *new* way is when the proposal is part of EcmaScript and we are utilizing the new features of said proposal.

The program will allow the users to preview proposals way before they are part of the language. This way the committee will get feedback from users of the language earlier in the proposal process, this will ideally allow for a more efficient process of adding proposals to EcmaScript.

3.1.1 Applying a proposal

The way this project will use the pre-existing knowledge a user has of their own code is to use that code as base for showcasing a proposals features. Using the users own code as base requires the following steps in order to automatically implement the examples that showcase the proposal inside the context of the users own code.

The tool has to identify where the features and additions of a proposal could have been used. This means identifying parts of the users program that use pre-existing EcmaScript features that the proposal is interacting with and trying to solve. This will then identify all the different places in the users program the proposal can be applied. This step is called *matching* in the following chapters

Once the tool has matched all parts of the program that the proposal could be applied, the users code has to be transformed to use the feature/s the proposal is trying to implement. This step also includes keeping the context and functionality of the users program the same, so variables and other context related concepts have to be transferred over to the transformed code.

The output of the previous step is then a set of code pairs, where one a part of the users original code, and the second is the transformed code. The transformed code is then ideally a perfect replacement for the original user code if the proposal is part of EcmaScript. These pairs are used as examples to present to the user, presented together so the user can see their original code together with the transformed code. This allows for a direct comparison and an easier time for the user to understand the proposal.

The steps outlined in this section require some way of defining matching and transforming of code. This has to be done very precisely and accurately in order to avoid bugs. Imprecise definition of the proposal might lead to transformed code not being a direct replacement for the code it was based upon. For this we suggest two different methods, a definition written in a custom DSL JSTQL and a definition written in a self-hosted way only using EcmaScript as a language as definition language. Read more about this in SECTION HERE.

3.2 Applicable proposals

A proposal for EcmaScript is a suggested change for the language, in the case of EcmaScript this comes in the form of an addition to the language, as EcmaScript does not allow for breaking changes. There are many different kinds of proposals, this project focuses exclusively on Syntactic Proposals.

3.2.1 Syntactic Proposals

A syntactic proposal, is a proposal that contains only changes to the syntax of a language. This means, the proposal contains either no, or very limited change to functionality, and no changes to semantics. This limits the scope of proposals this project is applicable to, but it also focuses solely on some of the most challenging proposals where the users of the language might have the strongest opinions.

3.2.2 Simple example of a syntactic proposal

Consider a imaginary proposal **optional let to int for declaring numerical literal variables**. This proposal describes adding an optional keyword for declaring numerical variables if the expression of the declaration is a numerical literal.

This proposal will look something like this:

```
1  // Original code
2  let x = 100;
3  let b = "Some String";
4  let c = 200;
5
6  // Code after application of proposal
7  int x = 100;
8  let b = "Some String";
9  let c = 200;
```

Listing 3.1: Example of imaginary proposal **optional let to int for declaring numerical literal variables**

See that in 3.1 the change is optional, and is not applied to the declaration of c, but it is applied to the declaration of x. Since the change is optional to use, and essentially is just *syntax sugar*, this proposal does not make any changes to functionality or semantics, and can therefore be categorized as a syntactic proposal.

3.2.3 [1, Discard Bindings]

The proposal Discard Bindingsis classified as a Syntactic Proposal, as it contains no change to the semantics of EcmaScript. This proposal is created to allow for discarding objects when using the feature of unpacking objects/arrays on the left side of an assignment. The whole idea of this proposal is to avoid declaring unused temporary variables.

Unpacking when doing an assignment refers to assigning internal fields of an object/array directly in the assignment rather than using a temporary variable. See 3.2 for an example of unpacking an object and 3.3.

```
1  // previous
2  let temp = { a:1, b:2, c:3, d:4 };
3  let a = temp.a;
4  let b = temp.b;
5
6  // unpacking
7  let {a,b ...rest} = { a:1, b:2, c:3, d:4 };
8  rest; // { c:3, d:4 }
```

Listing 3.2: Example of unpacking Object

```
1  // previous
2  let tempArr = [ 0, 2, 3, 4 ];
3  let a = tempArr[0]; // 0
4  let b = tempArr[1]  // 2
5
6  //unpacking
7  let [a, b, _1, _2] = [ 0, 2, 3, 4 ]; // a = 0, b = 2, _1 = 3, _2 = 4
```

Listing 3.3: Example of unpacking Array

In EcmaScripts current form, it is required to assign every part of an unpacked object/array to some identifier. The current status quo is to use _ as a sign it is meant to be discarded. This proposal suggests a specific keyword *void* to be used as a signifier whatever is at that location should be discarded.

This feature is present in other languages, such as Rust wildcards, Python wildcards and C# using statement and discards. In most of these other languages, the concept of discard is a single _. In EcmaScript the _ token is a valid identifier, therefore this proposal suggests the use of the keyword *void*. This keyword is already is reserved as part of function definitions where a function is meant to have no return value.

This proposal allows for the *void* keyword to be used in a variety of contexts. Some simpler than others but all following the same pattern of allowing discarding of bindings to an identifier. It is allowed anywhere the *BindingPattern*, *LexicalBinding* or *DestructuringAssignmentTarget* features are used in EcmaScript. This means it can be applied to unpacking of objects/arrays, in callback parameters and class methods.

```
1  using void = new UniqueLock(mutex);
2  // Not allowed on top level of var/let/const declarations
3  const void = bar(); // Illegal
```

Listing 3.4: Example discard binding with variable discard

```
1  let {b:void, ...rest} = {a:1, b:2, c:3, d:4}
2  rest; // {a:1, c:3, d:4};
```

Listing 3.5: Example Object binding and assignment pattern

```
1  function* gen() {
2    for (let i = 0; i < Number.MAX_SAFE_INTEGER; i++) {
3      console.log(i);
4      yield i;
5    }
6  }
7
8  const iter = gen();
9  const [a, , ] = iter;
```

```
10 // prints:
11 //   0
12 //   1
13
14 const [a, void] = iter; // author intends to consume two elements
15 // vs.
16 const [a, void, void] = iter; // author intends to consume three
    ↪ elements
```

Listing 3.6: Example Array binding and assignment pattern. It is not clear to the reader that in line 8 we are consuming 2 or 3 elements of the iterator. In the example on line 13 we see that is it more explicit how many elements of the iterator is consumed

```
1  // project an array values into an array of indices
2  const indices = array.map((void, i) => i);
3
4  // passing a callback to 'Map.prototype.forEach' that only cares about
5  // keys
6  map.forEach((void, key) => { });
7
8  // watching a specific known file for events
9  fs.watchFile(fileName, (void, kind) => { });
10
11 // ignoring unused parameters in an overridden method
12 class Logger {
13   log(timestamp, message) {
14     console.log('${timestamp}: ${message}');
15   }
16 }
17
18 class CustomLogger extends Logger {
19   log(void, message) {
20     // this logger doesn't use the timestamp...
21   }
22 }
23
24 // Can also be utilized for more trivial examples where _ becomes
25 // cumbersome due to multiple discarded parameters.
26 doWork((_, a, _1, _2, b) => {});
27 // vs.
28 doWork((void, a, void, void, b) => {
29 });
```

Listing 3.7: Example discard binding with function parameters. This avoids needlessly naming parameters of a callback function that will remain unused.

The grammar of this proposal is precicely specified in the specification found in the proposal definition on github.

```
1  var [void] = x;            // via: BindingPattern :: 'void'
2  var {x:void};              // via: BindingPattern :: 'void'
3
4  let [void] = x;            // via: BindingPattern :: 'void'
5  let {x:void};              // via: BindingPattern :: 'void'
6
7  const [void] = x;          // via: BindingPattern :: 'void'
8  const {x:void} = x;        // via: BindingPattern :: 'void'
9
10 function f(void) {}        // via: BindingPattern :: 'void'
11 function f([void]) {}      // via: BindingPattern :: 'void'
12 function f({x:void}) {}    // via: BindingPattern :: 'void'
13
14 ((void) => {});            // via: BindingPattern :: 'void'
15 (([void]) => {});          // via: BindingPattern :: 'void'
16 (({x:void}) => {});        // via: BindingPattern :: 'void'
```

```
17
18  using void = x;         // via: LexicalBinding : 'void' Initializer
19  await using void = x;   // via: LexicalBinding : 'void' Initializer
20
21  [void] = x;             // via: DestructuringAssignmentTarget : 'void'
22  ({x:void} = x);         // via: DestructuringAssignmentTarget : 'void'
```

Listing 3.8: Grammar of Discard Binding

3.2.4 Pipeline Proposal

The pipeline proposal is a Syntactic proposal with no change to functionality of EcmaScript, it focuses solely on solving problems related to nesting of function calls and other expressions that allow for a topic reference.

The pipeline proposal aims to solve two problems with performing consecutive operations on a value. In EcmaScript there are two main styles of achieving this functionality currently. Nesting calls and chaining calls, these two come with a differing set of challenges when used.

Nesting calls is mainly an issue related to function calls with one or more arguments. When doing many calls in sequence the result will be a *deeply nested* call expression. See in 3.9.

Challenges with nested calls

- The order of calls go from right to left, which is opposite of the natural reading direction users of EcmaScript are used to
- When introduction functions with multiple arguments in the middle of the nested call, it is not intuitive to see what call it belongs to.

Benefits of nested calls

- Does not require special design thought to be used

```
1      // Deeply nested call with single arguments
2      function1(function2(function3(function4(value))));
3
4      // Deeply nested call with multi argument functions
5      function1(function2(function3(value2, function4)), value1);
```

Listing 3.9: Example of deeply nested call

Nesting solves some of the issues relating to nesting, as it allows for a more natural reading direction left to right when identifying the sequence of call. However, solving consecutive operations using chaining has its own set of challenges when used

3.2.5 Description of Pipeline proposal

Challenges with chaining calls

- APIs has to be specifically designed with chaining in mind
- Might not even be possible due to external libraries
- Does not support other concepts such as arithmetic operations, array/object literals, await, yield, etc...

Benefits of chaining calls

- More natural direction of call order
- Arguments of functions are grouped with function name
- Untangles deep nesting

```
1    // Chaining calls
2    function1().function2().function3();
3
4    // Chaining calls with multiple arguments
5    function1(value1).function2().function3(value2).function4();
```

Listing 3.10: Example of chaining calls

The pipeline proposal aims to combine the benefits of these two styles without all the challenges each method faces.

The main benefit of pipeline is to allow for a similar style to chaining when chaining has not been specifically designed to be applicable. The idea uses syntactic sugar to change the order of writing the calls without influencing the API of the functions.

```
1  // Status quo
2  var minLoc = Object.keys( grunt.config( "uglify.all.files" ) )[ 0 ];
3
4  // With pipes
5  var minLoc = grunt.config('uglify.all.files') |> Object.keys(%)[0];
```

Listing 3.11: Example from jquery

```
1    // Status quo
2    const json = await npmFetch.json(npa(pkgs[0]).escapedName, opts);
3
4    // With pipes
5    const json = pkgs[0] |> npa(%).escapedName |> await
        ↪ npmFetch.json(%, opts);
```

Listing 3.12: Example from unpublish

```
1    // Status quo
2    return filter(obj, negate(cb(predicate)), context);
3
4    // With pipes
5    return cb(predicate) |> _.negate(%) |> _.filter(obj, %, context);
```

Listing 3.13: Example from underscore.js

```
1      // Status quo
2      return
       ↪ xf['@@transducer/result'](obj[methodName](bind(xf['@@transducer/step'],
       ↪ xf), acc));
3
4      // With pipes
5      return xf
6        |> bind(%['@@transducer/step'], %)
7        |> obj[methodName](%, acc)
8        |> xf['@@transducer/result'](%);
```

Listing 3.14: Example from ramda.js

3.2.6 Do proposal

This is where a description of the Do proposal might land even though actually defining the do proposal will be quite difficult, as it is hard to see exactly where one can apply this LOOOOL

3.2.7 Await to Promise

This section contains a description of a use-case of this DSL for rewriting code using the *await* keyword to use a Promise in stead.

3.3 Searching user code for applicable snippets

In order to identify snippets of code in the users codebase where a proposal is applicable we need some way to define patterns of code where we can apply the proposal. To do this, a DSL titled JSTQL is used.

3.3.1 JSTQL

In order to allow for the utilization of the users code. We have to identify snippets of the users code that some proposal is applicable to. In order to do this, we have designed a DSL called JSTQL JavaScript Template Query Language. This DSL will contain the entire definition used to identify and transform user code in order to showcase a proposal.

3.3.2 Matching

In order to identify snippets of code a proposal is applicable to, we use templates of JavaScript. These templates allow for *wildcard* sections where it can match against specific AST nodes. These *wildcard* sections are also used to transfer the context of the code matched into the transformation.

A template containing none of these *wildcards* is matched exactly. This essentially means the match will be a direct code search for snippets where the AST of the users code match the template exactly.

The *wildcards* are written inside a block denoted by ¡¡ WILDCARD ¿¿. Each wildcard has to have a DSL identifier, a way of referring to that wildcard in the definition of the transformation.

Each wildcard has to have some form of type. These types can be node-types inherited from Babels AST definition. This means if you want a wildcard to match any *CallExpression* then that wildcard should be of type CallExpression. In order to allow for multiple node-types to match against a single wildcard, JSTQL allows for sum types for wildcards, allowing multiple AST node-types to be allowed to a single wildcard definition.

This type definition is also used to define specific behavior the program using this DSL should perform. One example of this can be found in 3.22, where the DSL function *anyRest* is used to allow for any amount of child nodes found together with the wildcard. This means it is feasible to match against any number of function parameters for example.

```
1 let variableName = << expr1: CallExpression | Identifier >>;
```
Listing 3.15: Example of a wildcard

In 3.15 a wildcard section is defined on the right hand side of an assignment statement. This wildcard will match against any AST node classified as a CallExpression or an Identifier.

3.3.3 Transforming

Observe that once the a matching template has been defined, a definition of transformation has to be created. This transformation has to transfer over the code matched to a wildcard. This means a way to refer to the wildcard is needed. We do this in a very similar manner as defining the wildcard, since we have an internal DSL identifier previously defined in the definition of the matching, all that is needed is to refer to that identifier. This is done with a similar block definition ¡¡ ¿¿ containing the identifier.

```
1 const variableName = <<expr1>>;
```
Listing 3.16: See 3.15 contains identifier expr1, and we refer to the same in this example, the only transformation happening here is rewriting let to const.

3.3.4   Structure of JSTQL

JSTQL is designed to mimic the examples already provided by a proposal champion in the proposals README. These examples can be seen in each of the proposals described in 3.2.

Define proposal

The first part of JSTQL is defining the proposal, this is done by creating a named block containing all definitions of templates used for matching alongside their respective transformation. This section is used to contain everything relating to a specific proposal and is meant for easy proposal identification by tooling.

```
1  proposal Pipeline_Proposal{
2
3  }
```
Listing 3.17: Example of section containing the pipeline proposal

Defining a pair of template and transformation

Each proposal will have 1 or more definitions of a template for code to identify in the users codebase, and its corresponding transformation definition. These are grouped together in order to have a simple way of identifying the corresponding pairs. This section of the proposal is defined by the keyword *pair* and a block to contain its related fields. A proposal will contain 1 or more of this section. This allows for matching many different code snippets and showcasing more of the proposal than a single concept the proposal has to offer.

```
1      pair PAIR_NAME {
2
3      }
```
Listing 3.18: Example of pair section

Template used for matching

In order to define the template used to match, we have another section defined by the keyword *applicable to*. This section will contain the template defined using JavaScript with specific DSL keywords defined inside the template.

```
1  applicable to {
2
3  }
```
Listing 3.19: Example of applicable to section

Defining the transformation

In order to define the transformation that is applied to a specific matched code snippet, the keyword *transform to* is used. This section is similar to the template section, however it uses the specific DSL keywords to transfer the context of the matched user code, this allows us to keep parts of the users code important to the original context it was written in.

```
transform to{

}
```

Listing 3.20: Example of transform to section

All sections together

Taking all these parts of JSTQL structure, defining a proposal in JSTQL will look as follows.

```
proposal PROPOSAL_NAME {
    pair PAIR_NAME {
        applicable to {

        }
        transform to {

        }
    }
    pair PAIR_NAME {
        applicable to .....
    }

    pair ....
}
```

Listing 3.21: JSTQL definition of a proposal

3.4 Using the JSTQL with an actual syntactic proposal

3.4.1 Pipeline Proposal

The Pipeline Proposal is the easiest to define of the proposals presented in 3.2. This is due to the proposal being applicable to a very wide array of expressions, and the main problem this proposal is trying to solve is deep nesting of function calls.

```
 1 proposal Pipeline{
 2     pair 1 {
 3         applicable to {
 4             <<someFunctionIdent >>(<<someFunctionParam: Expression |
                 ↪ Identifier >>);
 5         }
 6
 7         transform to {
 8             <<someFunctionParam >> |> <<someFunctionIdent >>(%);
 9         }
10     }
11
12     pair 2 {
13         applicable to {
14             <<someFunctionIdent >>(
15                 <<firstFunctionParam : Expression | Identifier >>,
16                 <<restOfFunctionParams: anyRest >>
17                 );
18         }
19
20         transform to {
21             <<firstFunctionParam >> |> <<someFunctionIdent >>(%,
                 ↪ <<restOfFunctionParams >>);
22         }
23     }
24 }
```

Listing 3.22: Example of Pipeline Proposal definition in JSTQL

This first pair definition of the Pipeline proposal will apply to any *CallExpression* with a single parameter. And it will be applied to each of the deeply nested callExpressions.

As can be seen in 3.22

Bibliography

[1] proposal-discard-binding, April 2024.
   URL: `https://github.com/tc39/proposal-discard-binding?tab=readme-ov-file#object-binding-and-assignment-patterns`. [Online; accessed 25. Apr. 2024].

Appendix A

Generated code from Protocol buffers

```
1 System.out.println("Hello Mars");
```
Listing A.1: Source code of something