Master Thesis

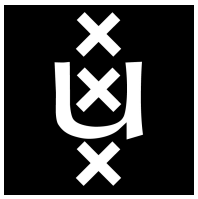
Syntactic language extensions for ECMAScript

Author:
Matthisk Heimensen
m@tthisk.nl

Supervisor:
dr. Tijs van der Storm
storm@cwi.nl

June 2015

Host organization: Centrum Wiskunde & Informatica http://www.cwi.nl



Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

UNIVERSITEIT VAN AMSTERDAM

Abstract

Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Faculteit der Natuurwetenschappen, Wiskunde en Informatica

Master of Science

Syntactic language extensions for ECMAScript

by Matthisk Heimensen

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Acknowledgements

The acknowledgements and the people to thank go here, don't forget to include your project advisor...

Contents

Chapter 1

Introduction and Motivation

1.1 Context

JavaScript is an implementation of the ECMAScript specification. Since the release of the specification there have been five iterations (where version four was skipped). The current version of the specification is ECMAScript 6, which has recently left draft status. With this new specification come new (syntactic) language features, before these features can be used in the JavaScript run-time environment[1] of choice, the vendors of these environments have to implement this new standard. It is possible to use these new features in current JavaScript run-times, and write future proof JavaScript code, through the use of program transformations.

In this work we study the extension of programming languages (through program transformations). The new ECMAScript specification presents an opportunity to research language extensions in context of a real-world problem. Using the RASCAL meta-programming environment[1] we will implement the program transformations. Afterwards we perform an analysis on the resulting transformation suite to uncover the *effectiveness* of our implementation.

"Program transformations find ubiquitous application in compiler construction to realize desugaring, optimizers, and code generators"[2] in our research we will focus on the desugaring of (new) language constructs.

"The aim of program transformation is to increase programmer productivity by automating programming tasks, thus enabling programming at a higher-level of abstraction, and increasing maintainability"[3]

1.2 Problem definition

Program transformations are faced with multiple problems, in what order do we run the transformation rules, how do we represent the program (e.g. abstract syntax) to perform transformations on, what guarantee do we have for the validity of our transformations,

[1] JavaScript code run-times are primarily found in modern browsers, but there also exist run-times that run dedicated inside an operating system

how can a transformation introduce new bindings. Some of these problems reoccur in every transformation that is created. These problems are identified as cross-cutting concerns. The goal of this thesis is to identify these concerns and separate them from the transformation code.

1.3 Scope

1.3.1 Expected results

This thesis will have the following results:

- A taxonomy for language extensions
- Language extension suite implementing (a subset of) the ES6 features
- Identification of cross-cutting concerns of our language extension suite
- A set of compatibility tests, testing our implementation against the specification document
- Eclipse integration (syntax highlighting & declared at hyperlinks) for new language features

1.4 Research questions

The central research questions of this thesis are:

1.
2. Evaluate the language workbench for the task of program transformation suite
3. How can independent program transformations be categorized?
4. What are the cross-cutting concerns of any program transformations, and how can we take care of these concerns outside of the transformations themselves?

1.5 Outline

In chapter 2 the background (program transformations, language workbench) is introduced. In chapter 3 we present a taxonomy for language extensions (categorization according to this taxonomy of ES6 features is found in Appendix B). In Chapter 4 we evaluate our resulting transformation suite (against others). Finally, we conclude in Chapter **??**.

Chapter 2

Background

2.1 ECMAScript

JavaScript[1]is the programming language implemented from the ECMAScript specification[4]. JavaScript is an interpreted, object-oriented, dynamically typed scripting language with functions as first-class citizens. It is mostly used inside the Web browser, but also available for non browser environments (e.g. node.js). In this section we will identify some of the key characteristics of the JavaScript programming language.

Syntax One of the main concerns for program transformations is syntax. The syntax of the programming language is to be extended to support new language features/constructs. ECMAScript falls in the C-type family of syntax definitions (i.e. it uses curly braces to delimit blocks of code). A program exists of a list of *statements*. A *statement* can be one of the language control flow constructs, a declaration, a function, or an *expression*. Functions in JavaScript can be either defined as an *expression* or as a *statement* (see section 2.1).

Inhertiance & Object orientation JavaScript can be classified as an object oriented language without *class* based inheritance model[2]. Instead of such a model JavaScript objects have a prototype chain that determine the inheritance.

In the JavaScript run-time environment everything is an object with properties (even functions 2.1), with the exception of primitive data-types (e.g. numbers). Every object has one special property called the *prototype*, this prototype references another object (with yet another prototype) until finally one object has prototype *null*. This chain of objects is called the *prototype chain*. When performing a lookup on an object for a certain key x, the JavaScript interpreter will look for property x on our object, if x is not found it will look at the *prototype* of our object. This iteration will continue until either prototype *null* is found or an object with property x is found in the prototype chain.

[1]Documentation on the JavaScript programming language can be found at the Mozilla Developer Network

[2]Most popular object-oriented languages do have *class* based inheritance (e.g. Java or C++)

To understand prototypal inheritance in comparison to class based inheritance, we can use an analogy. A class in a class based programming language can be seen as a blueprint from which to construct objects. When calling a function on an object created from such a blueprint the function is retrieved from the blueprint and not from the (constructed) object. So the object is related to the class (or blueprint) with an object-class relation. With prototypal inheritance there is no blueprint everything is an object. This is also how inheritance works, we do not reference a blueprint but we reference a fully build object as our prototype. This creates an object-object relation.

If we have an object Car which inherits from Vehicle we set the Car's object prototype to an instance of the Vehicle object. Where in class based inheritance there is a blueprint for Car which inherits from the Vehicle blueprint, to build a car we initialize from the blueprint.

```
1 var Vehicle = function() {...};
2 var Car = function() {...};
3
4 Car.prototype = new Vehicle();
```

Scoping Most programming languages have some form of variable binding. These bindings are only bound in a certain context, such a context is often called a scope. There are two different ways to handle scope in a programming language, Lexical (or static) scoping and dynamic scoping. Lexical scoping is determined by the placing of a binding within the source program or its *lexical context*. Variables bound in a dynamic scope are resolved through the program state or *run-time context*.

In JavaScript bindings are determined through the use of lexical scoping, on a function level. Variables bound within a function's context are available throughout the entire lexical scope of the function.

One exception on the lexical scoping is the binding of the *this* keyword. Similar to many other object oriented languages JavaScript makes use of a *this* keyword. In most object oriented languages the current object is implicitly bound to *this* inside functions of that object. [3] In JavaScript the value of *this* can change between function calls. For this reason the *this* binding has dynamic scoping (we can not determine the value of *this* before runtime).

Typing JavaScript is a dynamically typed language without type annotations. Each object can be typecast without the use of special operator(s) and will be be converted automatically to correct type during execution. There are 5 primitive types:

- *Boolean*
- *null*
- *undefined*
- *Number*
- *String*

[3] Sometimes *this* is called *self* (e.g. in the Ruby) programming language

Everything is else is an object.

Functions in JavaScript functions are objects. Because of this functions are automatically promoted to first-class citizens (i.e. they can be used anywhere and in anyway normal bindings can be used). This presents the possibility to supply functions as arguments to functions, and calling member functions of the function object. Every function object inherits from the *Function.prototype*[4] which contains functions that allow the overriding of this binding (see section 2.1).

Another common pattern is JavaScript related to functions, is that of the immediately invoking function expression (IIFE). Functions in JavaScript can be defined either in a statement or in an expression. When a function is defined as an expression it is possible to immediately invoke the created function:

```
1 (function() {
2   <Statement* body>
3 })()
```

Listing 2.1: Immediately invoking function expression

Asynchronous JavaScript is a single-threaded language, this means that concurrency through the use of processes, threads, or similar concurrency constructs is impossible.

2.2 Program Transformations

Eelco Visser defines the aim of a program transformation as follows:

"The aim of program transformation is to increase programmer productivity by automating programming tasks, thus enabling programming at a higher-level of abstraction, and increasing maintainability" ([3])

There are many different types of program transformations. Programs can be transformed from a source to a target language. Where both can be the same or different. One can be a higher level language and one a lower level language. In this thesis we will focus on transformations in the category rephrasing [3]. Here a program in a source language is transformed to a different program in the same language. The use case in our thesis is language extensions which are discussed in the next section.

2.3 Language Extensions

"Language extensions augment a base language with additional language features." [2] With the use of program transformations these extensions can be eliminated an a target program in the base language can be produced. *"Many compilers first desugar a source program to a core language."* [2] For example the Haskell functional programming language defines for many constructs how they can be transformed to a kernel language [5]. These pieces of *syntactic sugar* are examples of language extensions.

[4]For more information see Mozilla Developer Network

Language extensions are closely related to other forms of program transformations. There are (syntax) macros[6], these are language extensions defined within the same source file and are expanded before compilation (examples are SweeterJS [7], Syntax macros [8]). These *"define syntactic abstractions over code fragments"* [9]. Macro systems have several limitations. The transformations need to be present in the source file, macros are often restrained to a fixed invocation (through a macro identifier), and they often lack the expressiveness in their rewriting language[9]. There was an attempt to implement the ES6 specification through the use of syntax macros but the project[5] is abandoned.

Most language extensions can be classified as syntactic sugar. As mentioned before there are several languages that before compilation are transformed to a core language (e.g. Haskell core language). Language features classified as syntactic sugar, are features that can be expressed in the core language. But often benefit from improved human readability when notated with aid of the syntactic sugar. For example the assignment expression `a += b` is syntactic sugar notation for `a = a + b` of frequent occurrence in (c-style) programming languages.

2.3.1 Hygiene

Any program transformations that introduces new bindings in the target program risks the unintended shadow of user identifiers, or variable capture. When variable capture happens as a result of a program transformation, this transformation is unhygienic. Transformations that guarantee that no unintended variable capture will occur are called hygienic transformations. Hygienic program transformations are mostly studied in the context of macro expansions [7, 10, 11]

The problem of variable capture can best be explained through the use of an example. Here we use a language extension called *swap*, that can swap the values of two bindings with the use of a single statement. The transformation eliminates the `swap` statement and introduces an immediately invoking function expression (see section 2.1) to swap out the values. In the body of this function we bind the value of one of the input variables to a temporary binding:

```
1  swap x y;
2
```

```
1  (function() {
2    var tmp = x;
3    x = y;
4    y = tmp;
5  })();
6
```

FIGURE 2.1: A language extension swap is transformed to core JavaScript

This transformation rule introduces a new binding named `tmp` (at line 2). Because of this introduction the transformation could possibly generate variable capture in the target program. When the `swap` statement is performed by the user, where one of the variables to swap is named `tmp` the transformation will shadow the users binding and not produce the expected result (see figure **??**).

[5] https://github.com/jlongster/es6-macros

```
1  swap x tmp;
2
```

```
1  (function() {
2    var tmp = x;
3    x = tmp;
4    tmp = tmp;
5  })();
6
```

FIGURE 2.2: Introduction of variable capture because the transformation binds non-free name

The unhygienic transformation as explained above can often introduce subtle bugs in target programs, which are hard to identify for the user of the program transformation. Especially in the context of real-world applications, because of the complexity of the source program, and the transformation suite. Many transformations suites fail to identify and solve variable capture in their program transformations, in a case study performed by Erdweg et. al. eight out of nine DSL implementations were prone to variable capture [12].

2.4 Program Representation

Before programs can be transformed they need a structured representation. Seldom are program transformations performed on the the textual input. Here we discuss several ways to represent programs.

Parse Trees Parse trees represent a programs syntactic information in a tree. This tree includes layout information of the input program (e.g. white-space, comments). The parse tree is structured according to some context-free grammar, that defines how to parse textual input. The parse tree also deals with disambiguation of input and representation of parentheses. If the parser finds an ambiguity during the parse process the two (or more) alternatives are all inserted in the tree under an ambiguity node.

Abstract Syntax Tree or AST is produced by removing layout information from a parse tree, only the core constructs of the language grammar remain as nodes in the tree. White-space layout, comments, and things like parentheses (these are implicitly implied by the structure of the AST) are removed. An AST is often the input for compiler pipelines, program transformations, and refactorings.

Higher Order Syntax Trees (HOAS) To represent not just a program's sub expression relations but also its variable binding we can use a Higher Order Syntax Tree (HOAS)[13]. In a HOAS the variable bindings are made explicit (just as the sub expression relations are made explicit in an AST). Every variable has a binding-site and possibly uses throughout the rest of the tree. *"In addition to dealing with the problem of variable capture, HOAS provides higher-order matching which synthesizes new function for higher-order variables. One of the problems of higher-order matching is that there can be many matches for a pattern"* [3]

2.5 Parsing

Before a program is represented in on of the previously defined tree formats, it is represented in textual form. To transform the stream of input characters to a tree a program called a parser is used. There are many different types of parser techniques, and a large body of research [14–16]

The parsing of textual input often happens in two stages. First a *scanner* performs the lexical analysis. Which identifies the tokens of our syntax (e.g. keywords, identifiers). With this identification the parser operates on the identified tokens. Usually the lexical syntax of a language is *"specified by regular expression"* [9].

Scanners that do not have access to the parser, are also unaware of the context of lexical tokens. In case of JavaScript this can impose subtle difficulties in the implementation of a scanner. *"due to ambiguities in how regular expression literals (such as /[0-9]*/) and the divide operator (/) should be lexed."* [7]. For this reason traditional JavaScript parsers are often intertwined with their lexer. Disney et. al. avoid this problem by introducing a separate reader stage in the parser. This problem can also be avoided by using scannerless parsing, *"Scannerless parsing is a parsing technique that does not use a scanner to divide a string into lexical tokens. Instead lexical analysis is integrated in the context-free analysis of the entire string."* [14] these parsers preserve the lexical structure of the input characters, and make it possible to compose grammars [14].

There exist different types of parsers, where some can handle more grammars than others. Examples of parser types are LR (left to right), SLR (simple left to right), LALR (look-ahed left to right), or CLR (canonical left to right). Each of these parsers uses a parse table which contains all the knowledge of the grammar that has to be parsed. Now the parser is a loop over the input using the parse table to identify possible next symbols.

Scannerless parsing suffers form conflicts in the parse table [14], there are two causes for these conflicts, either there exists an ambiguity in the grammar, or lack of look-ahead. This second case occurs when a choice from the parse table is incorrect, but this can not be determined by the parser inside its current look-ahead window. Generalized parsers solve this problem by foking new parsers for each conflict found in the parse table, when all forked parsers finish correctly the parser outputs a parse forest (instead of a parse tree). When conflict arose from lack of look-ahead the forked parser for this conflict will fail.

2.6 Language Workbenches

Language workbench is a term popularized by Martin Fowler and we can formulate his definition as follows:

A language workbench makes it easy to build tools that match the best of modern IDEs, where the primary source of information is a persistent abstract representation. Its users can freely design new languages without a semantic barrier. With the result of language oriented programming becoming much more accessible. [17]

> Essentially the promise of language workbenches is that they provide the flexibility of external DSLs without a semantic barrier. Furthermore they make it easy to build tools that match the best of modern IDEs. The result makes language oriented programming much easier to build and support, lowering the barriers that have made language oriented programming so awkward for so many. ([17])

These workbenches improve the efficiency of language oriented programming[18] significantly, by giving programmers the the tools to define programming languages with an abstract representation.

The RASCAL[1] meta-programming environment is a language workbench. It allows programmers to define context-free grammars, generate parsers for these grammars. The workbench integrates with the Eclipse development environment, which allows meta-programmers to integrate interactive language-specific IDE features and syntax highlighting. All language extensions presented in this thesis are implemented using the RASCAL meta-programming environment.

Syntax definition Syntax definitions in Rascal allows the definition of parsers for programming languages or domain-specific languages. Rascal generates scannerless parsers from the context-free syntax definition (see section 2.5). Rascal uses generalized parsing to avoid lack of look-ahead, when the parser detects an ambiguity in its parse table, two parsers are forked from the main parser. In case of ambiguity a parse forest is returned, in case of lack of look-ahead error this parser fails (see section 2.5).

Concrete syntax A Rascal generated parser returns a parse tree (or forest), which is an ordered, rooted tree that represents the syntactic structure of a string according to the formal grammar used to specify the parser. Most transformation systems would implode this concrete syntax tree to an AST and perform program transformations on this tree. Rascal gives the possibility to perform program transformation directly on the concrete syntax tree through the use of concrete-syntax patterns. This has multiple advantages over an AST based solution:

- Preserving layout information encoded in the concrete-syntax tree
- Avoiding the use of a pretty printer to output the transformed AST to a textual representation
- Rewrite rules can use concrete syntax patterns for their definition, improving human readability

A concrete-syntax pattern may contain variables in the form of a grammars non-terminals, these variables are bound in the context of the current patterns use (e.g. functions body when used as a formal parameter) if the pattern is matched successful. A quoted concrete-syntax pattern starts with a non-terminal to define the desired syntactic type (e.g. Expression), and they ignore layout.

```
1 (Symbol)‘Token1 Token2 ... Token3‘ := pt
```

Where a token can be a lexical token, typed variable pattern: `<Type Var>`, or a bound variable pattern `<Var>` (where `Var` is already bound in the current context, resolved and used as a pattern). And `Symbol` is a non-terminal.

Rascal supports a shorthand notation to parse a string starting from a non-terminal: (where `"..."` is any string)

```
1 [Symbol]"..."
```

Chapter 3

Transforming ECMAScript 6

3.1 Motivating Examples

The Babeljs compiler visits each AST node, functions inside of a specific transformation can announce themselves for the callback stack of a node visit. The callback to this function receives several arguments (current AST node, parent AST node, current Scope, and the file being transformed). The Scope class has functions presented in 3.1, if a node transformation wants to introduce a new variable say ref it calls $generateUidIdentifier$ with String ref as an argument on Scope and receives a name which is not bound in the current Scope. For this to work the visitor needs to have information of all the variables bound in the current scope. This makes the transformation more a full compiler than a set of transformation rules. For starters the transformation code needs to be aware of the Scope class and its functions.

```
1  generateUidIdentifier(name: string) {
2    return t.identifier(this.generateUid(name));
3  }
4
5  generateUid(name: string) {
6    name = t.toIdentifier(name).replace(/^_+/, "");
7
8    var uid;
9    var i = 0;
10   do {
11     uid = this._generateUid(name, i);
12     i++;
13   } while (this.hasBinding(uid) || this.hasGlobal(uid) || this.
     hasReference(uid));
14
15
16   var program = this.getProgramParent();
17   program.references[uid] = true;
18   program.uids[uid] = true;
19
20   return uid;
21 }
22
23 _generateUid(name, i) {
24   var id = name;
25   if (i > 1) id += i;
26   return `_${id}`;
```

```
27   }
```

Listing 3.1: Variable capture avoidance code from babeljs source[1]

The Traceur compiler totally ignores the problem of variable capture. In conformance with the case study conducted by Erdeweg et. al. [2] we were able to identify a transpiler that fails to address variable capture. With a simple example we can demonstrate the introduction of capture by Traceur 3.2

```
1 function f() {
2   var $__0 = 0;
3   var x = () => this;
4 }
```

Listing 3.2: Example input to Traceur[2]

```
1 function f() {
2     var $__0 = this;
3     var $__0 = 0;
4     var x = (function() {
5       return $__0;
6     });
7   }
```

Listing 3.3: Variable capture

3.2 Taxonomy of language features

Every language extension has several properties which can be identified and categorized along certain dimensions. In this chapter we present a taxonomy for language extensions. In appendix B we categorize ES6 language features according to this model.

3.2.1 Dimensions

The following dimensions are identified and used to categorize every language extension.

Category One of the rephrasing categories defined by Eelco Visser[3], rephrasings are program transformations where source and target program language are the same. Each category is explained in table 3.1.

Abstraction level Program transformations can be categorized by their abstraction level. There are four levels of abstraction (similar to those of macro expansions [8]), character-, token-, syntax-, or semantic-based. Character and token based transformations work on a program in textual representation. Syntactical transformations work on a program in its parsed representation (either as an AST or as a parse tree, see section 2.4). Next to the syntactic representation semantic transformations also have access to the static semantics of the input program (e.g. variable binding).

[1] https://goo.gl/BZKIvV
[2] https://goo.gl/Ds3xUn
[3] http://www.refactoring.com

Extension or Modification Rephrasings (identified above) try to say the same thing (i.e. no change in semantics) but using different words[3]. Sometimes these different words are an extension on the core language, in this case we call the transformation a *program extension*. In other cases the transformation uses only the words available in the core language, then we call the transformation a *program modification*. Transformations that fall in the *optimization* (see table 3.1) category are program modifications. An example is tail call optimization in which a recursive function call in the *return* statement is reduced to a loop to avoid a call-stack overflow error (see appendix B.9).

Scope Program transformations performed on the abstraction level of context-free syntax (or semantics) receive the parse tree of the source program as their input. A transformation searches the parse tree for a specific type of node. The type of node to match on is defined by the transformation and can be any syntactical type defined in the source program's grammar. The node matched by a transformation and whether or not information from outside this node's scope is used during transformation determine the scope of a program transformation, there are four different scopes:

When a program transformation matches on a sub-tree of the parse-tree and only transforms this matched sub-tree it is a *(1) local-to-local* transformation. If the transformation needs information outside the context of the matched sub-tree, but only transforms the matched sub-tree it is *(2) global-to-local*. When a transformation has no additional context from its local sub-tree but does alter the entire parse-tree it is called *(3) local-to-global*. If the transformation transforms the input program in its entirety it is *(4) global-to-global*.

Syntactically type preserving Program transformations performed on syntax elements can preserve the syntactical type of their input element or alter it. Two main syntactical types in JavaScript are Statement and Expression (see section 2.1). If a transformation matches on a Expression node but returns a Statement it is non syntactical type preserving.

Introduction of bindings Does the transformation introduce new bindings. Transformations that do not introduce bindings are guaranteed to be hygienic (see section 2.3.1), where binding introducing transformations can cause variable capture from synthesized bindings to source bindings.

Depending on bindings (i.e. run-time code) Will the target program produced by the transformation depend on context not introduced by the transformation (e.g. global variables, external libraries).

Compositional When a program transformation does not alter the containing context of the matched parse-tree node, it is said to be compositional. The main concern of compositionality of program transformations is if the transformation can be reversed or not.
A parse-tree node with context C and children context $C[x]$ is transformed to context C' with children context $C'[x]$ where x is unchanged.

Preconditions What are the preconditions that have to be met before execution of a transformation rule, to ensure validity of the transformation (e.g. all sub-terms have to be analyzed and transformed)

Restrictions on sub-terms Does the language extensions impose restrictions on the sub-terms used inside of the language extension.

Analysis of sub-terms Are the non-terminals of our language extension analyzed and possibly transformed by the transformation rule.

Dependency on other extensions Can the language extensions be performed stand-alone or is there a dependency on one of the other extensions.

Backwards compatible Is the API of the transformed code compatible with the ECMAScript 6 specification (i.e. can we import a transformed module in ECMAScript 6 and use it properly).

Decomposable Is it possible to identify smaller transformation rules inside this language extension, that can be performed independently from one another.

3.2.2 On the expressive power of ECMAScript 6

Matthias Felleisen presents a rigorous mathematical framework to define the expressiveness of programming languages (as compared to each other) in his paper "On the expressive power of programming languages"[19].

The formal specification presented by Felleisen falls outside of the scope of this thesis. But we can summaries his findings informally: If a construct in language A can be expressed in a construct in language B by only performing local transformations (*local-to-local*), then language A is no more expressive than language B. And *"By the definition of an expressible construct, programs in a less expressive language generally have a globally different structure from functionally equivalent programs in a more expressive language."*[19]

This implies that every language extension transformed through a *local-to-local* scoped transformation (see section 3.2), creates a language that is no more expressive than the original (core) language. In table 3.2 the categorization of ECMAScript 6 language extensions is displayed. Ten of the twelve extensions can be transformed with the use of a *local-to-local* transformation, the two exceptions are, the extension of binding constructs (see appendix B.11). This extension requires a *global-to-global* transformation. And the arrow function (see appendix B.0.0.1) which requires a *global-to-local* transformation, because the transformation depends on the context in which the arrow function is used (i.e. either the global scope or inside a function).

"Intuitively, a programming language is less expressive than another if the latter can express all the facilities the former can express in the language universe." [19]. Thus ES6 is more expressive than ES5.

TABLE 3.1: Rephrasing categories

Normalization	The reduction of a source program to a target program in a sub-language of the source program language.
Desugaring	a language construct (called syntactic sugar) is reduced to a core language.
Simplification	this is a more generic form of normalization in which parts of the program are transformed to a standard form, with the goal of simplifying the program without changing the semantics.
Weaving	this transformation injects functionality in a source program, without modifying the code. It is used in aspect-oriented programming, where cross-cutting concerns are separated from the main code and later 'weaved' with the main program through a aspect weaver.)
Optimization	These transformations help improve the run-time and/or space performance of a program
Specialization	Code specialization deals with code that runs with a fixed set of parameters. When it is known that some function will run with some parameters fixed the function can be optimized for these values before run-time. (e.g. compiling a regular expression before execution).
Inlining	Transform code to inline a certain (standard) function within your function body, instead of calling the function from the (standard) library. This produces a slight performance increase because we avoid an additional function call. (this technique is more common in lower level programming languages e.g. C or C++)
Fusion	Fusion merges two bodies of loops (or recursive code) that do not share references and loop over the same range, with the goal to reduce runtime of the program.
Other	
Refactoring	"is a disciplined technique for restructuring an existing body of code, alteringits internal structure without changing its external behaviour."[3]
Obfuscation	is a transformation that makes output code less (human) readable, while not changing any of the semantics.
Renovation	is a special form of refactoring, "*to repair an error or to bring it up to date with respect to changed requirements*" [3]

Table 3.2: ES6 features transformation dimensions

	Arrow Functions	Classes	Destructuring	Object literals	For of loop	Spread operator
Category	D.	D.	D.	D.	D.	D.
Abstraction level	CfS.	CfS.	CfS.	CfS.	CfS.	CfS.
Scope	G2L	L2L	L2L	L2L	L2L	L2L
Extension or Modification	E.	E.	E.	E.	E.	E.
Syntactically type preserving	●	●	○	●	●	●
Introducing bindings	●	○	●	○	○	●
Depending on bindings	○	●	●	○	○	●
Compositional	●	●	●	●	●	●
Analysis of subterms	●	●	●	○	○	●
Constraints on subterms	○	●	○	○	○	○
Preconditions	●	●	○	○	○	○
Dependencies	○	○	●	○	●	○
Backwards compatible	●	●	●	●	●	●
Decomposable	○	○	●	●	●	●

	Default parameters	Rest parameters	Template Literals	Generators	Let Const	Tail call
Category	D.	D.	D.	D.	U.	Opt.
Abstraction level	CfS.	CfS.	CfS.	CfS.	S.	CfS.
Scope	L2L	L2L	L2L	L2L	G2G	L2L
Extension or Modification	E.	E.	E.	E.	E.	M.
Syntactically type preserving	●	●	●	●	●	●
Introducing bindings	○	○	○	○	●	○
Depending on bindings	○	○	○	●	○	○
Compositional	●	●	●	●	○	○
Analysis of subterms	○	○	○	●	●	●
Constraints on subterms	○	○	○	○	○	○
Preconditions	○	○	○	●	●	○
Dependencies	○	○	○	○	○	○
Backwards compatible	●	●	●	●	○	●
Decomposable	○	○	○	○	●	○

●: Yes, ○: No, **D**: Desugaring, **U**: Undefined, **Opt**: Optimization, **CfS**: Context-free-syntax, **L2L**: local-to-local, **E**: Extension, **M**: Modification

3.3 Implementation

Each transformation is defined as a rewrite rule. It has a concrete syntax pattern which matches part of a parse-tree. The result is a concrete piece of syntax, using only constructs from the core syntax definition (i.e. ECMAScript 5). The rewrite rules are exhaustively applied on the input parse-tree until no more rewrite rules match any sub-trees of the input. Application of rewrite rules to the parse-tree is done bottom-up, because several rewrite rules (e.g. arrow function) demand their sub-terms to be transformed to guarantee successful completion.

3.3.1 cross-cutting concerns

Each transformation rule has to deal with similar issues, can we identify these issues and solve them in a standalone (language agnostic) fashion? Many problems with transformations originate with name binding in source and target language.

Variable capture In section 2.3.1 we have described the problem of variable capture in the context of program transformations. To solve this problem for our transformation suite we reuse the *name-fix* algorithm presented by Erdweg et. al. [2]. The algorithm relies on a binding graph to identify variable capture, and uses string origins [20] to distinguish between synthesized identifiers (i.e. those identifiers introduced by the transformation) and identifiers originating from the source program.

name-fix analyses the *name graph* of source and target program to identify variable capture. The name graph contains nodes for all identifiers in the program, a directed edge indicates a reference to a declaration. Because our source and target language are both JavaScript and thus have the same binding mechanism, we only generate the name graph for our target program. In the graph a distinction is made between synthesized and user bindings. To identify whether a binding originates from the source program or was synthesized during transformation, the *name-fix* algorithm uses string origins [20].

In figure 3.1 we use the target program from our `swap` language extension (see section 2.3.1) to illustrate how these name graphs are constructed, and how *name-fix* identifies variable capture. Each node contains a line number and the identifier it resembles from that line, a directed edge is a reference to a declaration. Nodes with a white background originate from the source program, nodes with a dark background are synthesized during transformation. Line numbers from synthesized identifiers are appended with a tick.

```
1  var x = 0,
2    tmp = 1;
3
4  (function() {
5    var tmp = x;
6    x = tmp;
7    tmp = tmp;
8  })();
```

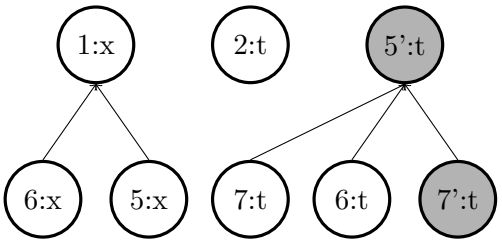


FIGURE 3.1: Target program generated by swap language extension and corresponding name graph

Node *7:t* and *6:t* originate from the source program but reference a synthesized declaration. Node *5:t* shadows the original declaration *2:t*. *name-fix* adds node *5:t* to the list of declarations to be renamed. After the entire name graph is analyzed/generated all declarations from this list are rebound under free names (i.e. names not yet bound in the target program). Since *7:t* and *6:t* are not correct references of *5:t* they are not renamed. The resulting name graph and target program are presented in figure 3.2

```
1  var x = 0,
2    tmp = 1;
3
4  (function() {
5    var tmp$0 = x;
6    x = tmp;
7    tmp$0 = tmp;
8  })();
```

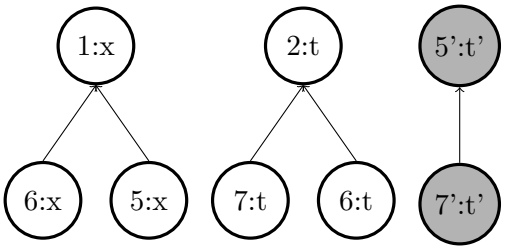


FIGURE 3.2: Target program and name graph after *name-fix* algorithm

name-fix does apply a *"a closed-world assumption to infer that all unbound variables are indeed free, and thus can be renamed at will."* [2]. This means that global variables defined by the user (or a third-party library) that are included within the same run-time as our target problem, could possibly be shadowed by the renamed bindings. However there is no way for *name-fix* to know what bindings will be taken by other bindings than those bound in the source program.

Introducing (multiple) bindings As described above various desugarings have to introduce bindings for correct transformation. In most cases new bindings will be bound to some temporary or reference variable name (e.g. _ref or _tmp). If a language extension is used multiple times within the same scope the same identifier will be declared multiple times. This is a similar problem to that of variable capture, but the capture now happens with binding all introduced by transformation code and not from a transformation to source binding. Similar to the variable capture problem this problem could be solved uniquely for every transformation, or can be extracted and solved in a generic manner (i.e. agnostic from transformation code).

Supply each *desugar* function with a function called *generateUId* (generate unique identifier). When a transformation wants to generate an identifier, it calls this function (which has knowledge of all previously generated identifiers) and generates a unique identifier through the use of *gensym***??** algorithm.

Chapter 4

Results and Comparison

4.1 Evaluation

There are several criteria as to which we evaluate the resulting transformation suite. These criteria are described in the following table.

Table 4.1: Evaluation criteria

Correctness	A language extension is performed correct if the target program is semantically similar to that of the language feature defined in the ECMAScript 6 specification[4]. To test the correctness we use a test-suite created especially for this purpose[1]
Modularity	Is it possible separate and recombine the different language extensions.
Size	How many lines of code are used to implement the transformation suite, if possible compared on a feature basis
Output "noise"	One of the challenges programmers face when using program transformations is the debugging of the generated program. During transformation the language extensions are eliminated from their code (and replaced by core language constructs). The more "noise" a transformation (suite) generates the harder it becomes for a programmer to debug generated code. An informal definition of noise is code added by a transformation that removes structure of the target program further away from the source program.
Coverage	How many language features are implemented
Performance	What is the run time of the transformation code

To evaluate our implementation of the transformation suite we validate against three open-source implementation of ECMAScript 6 language as a transformation suite (to

[1] https://kangax.github.io/compat-table/es6/

ECMAScript 5). We selected the following *transpilers* (these transformation suites categories themselves as transpilers a combination of compiler and transformer) to evaluate against:

Table 4.2: Open-source ES6 transpilers

	Description	Cont.	Com.
Babel JS[1]	The most used ECMAScript 6 to 5 transpiler (was named 6to5 before). It is created using JavaScript and relies on the Acorn[4] JavaScript parser.	105	4582
Traceur[2]	This transpiler is produced by a team from Google. Written in JavaScript and includes its own parser. This transpiler relies more heavily on a run-time environment than the other transpilers.	55	1584
ES6-Transpiler[3]	Apart form Traceur and Babel JS there only exist small individual projects that implement ES6 to 5 transpilers. This is the most feature complete of those transpilers. It relies on the ESPrima[5] JavaScript parser.	5	250

minipage0.45

4.2 Related Work

Eelco Visser, name-fix algo, Irvine program transformation catalog, Maerds

[1] http://www.babeljs.io
[2] https://github.com/google/traceur-compiler
[3] https://github.com/termi/es6-transpiler
[4] https://github.com/marijnh/acorn
[5] http://esprima.org/

[2] ES6 transpiler does not have separate files for arrow function and paramater transformation, these both reside in *functions.js*. This number is the size (in lines of code) of this file.

[3] Not implemented by ES6 transpiler

Table 4.4: Lines of code - parser

	Files	Lines of code
Babel	22	3583
Traceur	15	6681
Rascal	12	555

Table 4.6: Compatibility tests

	Total	**Babel**	**Traceur**	**Rascal**
Syntax	76	76	60	58
Bindings	19	15	15	12
Functions	62	54	50	34
Total	157	136 (87%)	125 (80%)	98 (62%)

Appendix A

Artifact Description

Summary We provide implementation of ECMAScript 6 language features syntax, and transformation rules in the Rascal meta-programming language (http://rascal-mpl.org). We use rascal's built-in support for syntax definition and parsing.

Content The code for language extensions is stored in *src/extensions* here we will discuss implementation status for each language feature:

Core Syntax (*src/core/Syntax.rsc*)
The core JavaScript syntax definition. There are two differences with the specification, semicolons are required (the ECMAScript 5 defines a functionality called automatic semicolon insertion or asi, with asi semicolons are not required and can be inferred by the parser). Because we use the Rascal syntax definition (and not a custom parser) we do not support asi. For the same reason there is no support for comma expression, the introduction of this expression causes ambiguities in the grammar.

Function extensions
Arrow functions (*src/extensions/arrow*)
Full functionality* (i.e. basic functionality and correct *this*,*arguments* binding)

Class declaration (*src/extensions/classes*)
Full functionality.* (i.e. basic support, support for extension, getter/setter/static methods, methods are non-enumerable)

Super (*src/extensions/classes*)
Full functionality.*

Generators (*src/extensions/generators*)
No support. Due to the time constraint nature of the project this feature has not been implemented.

Syntax extensions
Destructuring (*src/extensions/destructuring*)
Near full functionality. There is no support for unbound match:

```
1 var [,b] = [1,2]; // b == 2
```

This feature is not implemented because syntax definition for empty elements in a comma separated list is complex. Resulting in very unclear/cluttered/complicated transformation rules/code. To avoid this complexity we decided to refrain from implementation of this feature.

For of loops (*src/extensions/forof*)
Full functionality.

Binary & Octal literals (*src/extensions/literal*)
Full functionality.

Object literal (*src/extensions/object*)
Full functionality.

Rest Parameters (*src/extensions/parameters*)
Near full functionality. (i.e. no support for arguments object interaction outside of strict mode).

Default (*src/extensions/parameters*)
Basic functionality.

Spread operator (*src/extensions/spread*)
Near full functionality. (i.e. no support for spread operator with generators, because we do not implement generators)

Template Literals (*src/extensions/template*) Full functionality (i.e. tagged template literals and normal template literals)

Binding extensions
Let (*src/extensions/letconst*)
Full functionality. (i.e. support for renaming of function scoped clashes of block scoped names, no reference possible before definition, and possibility to throw run-time errors on redeclaration)

Const (*src/extensions/letconst*)
Full functionality. (i.e. same as let)

Modules (*src/extensions/modules*) No support.

*These features make use of the ES6 *new.target* property. This newly introduced property is an implicit parameters. Implicit parameters are those parameters that are set during every function execution but not explicitly passed as an argument by the caller. It is used to refer to the constructor function when a function is invoked with the use of the *new* keyword. Our implementation does not support the *new.target* property. It is possible to parse a new.target references but we have not implemented any transformation code to transform the property reference to ES5 code. The dynamic behavior[1]of *new.target* makes the transformation complex to perform during a static (i.e. non-runtime) phase. To transform *new.target* a global analysis of the program would

be needed and a global transformation (each function needs to be analyzed and each function call), thus a global-to-global transformation (see Section 3.2).

The compatibility tests as discussed in **??** are stored in *input/compatibility* and can be invoked from the Rascal module at *src/test/Compatibility.rsc*

Our implementation of the *name-fix*[12] algorithm resides in *src/core/resolver*.

Getting the artifact The latest version of the artifact is available at `https://github.com/matthisk/rascal-sweetjs` as a git repository. The repository page includes information on how to run the artifact inside the Eclipse IDE[2].

[1] *new.target* refers to the constructor function called with the new keyword, even inside the paren constructor function invoked through *super*. It is undefined in functions that are called without new keyword. The behavior can be overridden with the use of *Reflect.construct*

[2] `http://eclipse.org`

Appendix B

Language feature categorization

The following appendix contains a categorization of ECMAScript 6 language features according to the taxonomy established in section 3.2

B.0.0.1 Arrow Functions

Arrow functions[4, 14.2] are the new lambda-like function definitions inspired by the Coffeescript[1] and C# notation. The functions body knows no lexical this binding (see section 2.1 for more on JavaScript's this binding) but instead uses the binding of its parent lexical scope.

Category The arrow function construct is eliminated by translating them into the core function construct. Thus we speak of a syntactic sugar and a desugaring.

Scope To avoid a *ReferenceError* when an arrow function is used outside of the lexical scope of a function, the *_arguments* identifier should reference *undefined*. This is not the case when an arrow function is used inside the lexical scope of a function, then *_arguments* references the value of *arguments* in its containing lexical function. This implies that the transformation of an arrow function requires context information of this arrow function, thus the transformations scope is global-to-local.

Analysis of sub-terms References to the *arguments*, *super* and *this* keyword have to be identified in the body of an arrow function and renamed.

"An ArrowFunction does not define local bindings for **arguments**, **super**, **this**, or new.target. Any reference to arguments, super, or this within an ArrowFunction must resolve to a binding in a lexically enclosing environment." ([4, 14.2.16])

An arrow function is transformed to an ES5 function. This function is wrapped in a immediately invoking function expression, this IIFE is used to create a new closure with the **this**, and **arguments** bindings of the lexical enclosing scope bound to **_this**

[1] http://coffeescript.org/

and **_arguments** respectively. The enclosing lexical scope is not polluted with new variable declarations. All references to *this* and *arguments* inside the body of arrow function are renamed to the underscored names (only references in the current scope (i.e. no deeper scope) are renamed).

```
var f = <Arrow Function>;
```

```
var f = (function(_arguments,_this) {
  return <Desugared Arrow Function>
})(arguments,this);
```

Constraints on sub-terms Use of the **yield** keyword is not allowed in an arrow functions body. Arrow functions can not be generators (see section **??**) and deep continuations are avoided (i.e. yielding from inside an arrow function to the lexical enclosing generator).

Preconditions Before an arrow function can be transformed its body should have been transformed. This to prevent the incorrect renaming of sub-terms (**super**,**arguments**,**this**), because they still fall in the same scope depth. When visiting the body of an arrow function to rename sub-terms, the visitor breaks a path once a ES5 function is found, it does not break a path once it finds an arrow function. The following example illustrates how a desugaring would fail to correctly rename sub-terms if the body is not already desugared.

```
var f = () => {
  () => this;
};
```

```
var f = (function(_this,_arguments) {
  return function() {
    () => _this;
  };
})(this,arguments);
```

Listing B.1: Incorrect renaming of this in nested arrow function

B.1 Classes

Class definitions[4, 14.5] are introduced in ECMAScript 6 as a new feature to standardize the inheritance model. This new style of inheritance is still based on the prototypal inheritance (see section 2.1).

Category The class construct is syntactic sugar for simulation of class based inheritance through the core prototypal inheritance system of the ECMAScript language.

Syntactically type preserving There are two types of class declarations, a direct declaration as a Statement, or an Expression declaration. They are both transformed to a construct of the same syntactical type.

Depending on bindings Several features of the class declaration as described in the specification use some run-time functions. This to avoid unnecessary "noise" in the generated program. The following features use a run-time function

- *(_classCallCheck)*: Check if constructor was called with *new*
- *(_createClass)*: Add method as non-enumerable property to prototype of class.
- *(_inherits)*: Inherit from the prototype of the parent class.
- *(_get)*: Retrieve properties or constructor of the parent class.

Analysis of sub-terms References of **super** and a **super** call have to be identified inside the constructor or class methods. These are transformed to preserve the correct binding of **this**.

Constraints on sub-terms No.

Preconditions Bodies of constructor and class methods should have been transformed before the class declaration itself is transformed. This to prevent incorrect transformation of the sub-term **super**. (See preconditions of arrow functions, this is the same problem)

B.2 Destructuring

Destructuring[4, 12.14.5] is a new language construct to extract values from object or arrays with a single assignment. It can be used in multiple places among which parameters, variable declaration, and expression assignment.

Category The destructuring language feature is eliminated by translating it into core language concepts such as member access on objects and array member selection.

Syntactically type preserving Yes

Dependencies Object destructuring supports the computed property notation of extended object literals (as discussed in Appendix B.3):

```
1 var qux = "key";
2 var { [qux] : a } = obj;
```

Destructuring thus has a dependency on extended object literal notation for this feature to work properly.

Decomposable This language features consists of two types of destructuring, object destructurings, and array destructurings. These two different features can be transformed separately.

B.3 Extended object literals

Literal object notation receives three new features in the ECMAScript 6 standard[4, 12.2.5]. Shorthand property notation, shorthand method notation, and computed property names.

Category The extended object literals are syntactic sugar in the purest form. There are direct rules to eliminate the introduced concepts to core concepts of the ECMAScript 5 language. (e.g. shorthand property notation translate to *property: value* notation)

Introducing bindings One run-time function is used by the transformation suite, to set properties with computed property names. This method is a wrapper for *Object.defineProperty*[2]method. By using this run-time function instead of *Object.defineProperty* directly we avoid unnecessary "noise".

Decomposable All three extensions of the object literal can be transformed separately.

B.4 For of loop

The ECMAScript 6 standard introduces iterators, and a shorthand *for-loop* notation to loop over these iterators. This loop is called the *for of* loop[4, 13.6.4]. Previous versions of the ECMAScript standard had default for loops, and *for in* loops (which iterate over all enumerable properties of an object and those inherited from its constructor's prototype).

Category This construct can be eliminated by transformation to a *for-loop* using an iterator variable, and selecting the bound variable for the current index from the array using this iterator variable.

Constraints on sub-terms No constraints any < *Statement* > can be used within the body of a for-of loop.

```
1 syntax Statement
2   = "for" "(" Declarator ForBinding "of" Expression ")" Statement;
```

[2]For more information on this function see the Mozilla Developer Network documentation

Dependencies The for-of loop binding can be declared with the let declarator (see Appendix B.11).

```
1 for( let x of arr ) <Statement>
```

B.5 Spread operator

ECMAScript 6 introduces a new unary operator named spread[4, 12.3.6.1]. This operator is used to expand an expression in places where multiple arguments (i.e. function calls) or multiple elements (i.e. array literals) are expected.

Category This unary operator can be expressed using core concepts of the ECMAScript 5 language. In case the operator appears in a place where multiple arguments are expected, a call of member function *apply*[3]on our function can be used to supply arguments as an array. In case it appears in a place where multiple elements are expected the prototype function *concat* of arrays can be used to interleave the supplied argument to the operator with the rest of the array.

Every function in JavaScript is an object, Function.prototype has a method called *apply* which can be used to invoke a function, the first argument of *apply* is the object to bind to *this* during function execution (see Section 2.1). And the second argument is an array whose values are passed as arguments to the function:

```
1 f(...arr);
```

```
1 f.apply(f, arr);
```

```
1 [x, ...arr];
```

```
1 [x].concat(arr);
```

Introducing bindings As explained above a call to *apply* on a Function's prototype requires the callee to set the object that is bound to *this* during execution of the function. When a function is called as a property of an object, this object should be bound to *this* during the function's execution (and not the function as shown above). The following example illustrates this behavior:

```
1 a.f(...args);
```

```
1 a.f.apply(a,args);
```

Listing B.2: apply function with correct *this* scope

If however the object (*a* in this case) is itself a property of a different object we need to bind this value so it can be used as the first argument of *apply*. The following example illustrates this special case:

```
1 a.b.f(...args);
```

```
1 var _b$a;
2 (_b$a = b.a).f.apply(_b$a, args);
```

[3]For more information see Mozilla Developer Network documentation

If there exists a function call of function f on object a where spread operator is used in arguments list. This function call has to be transformed to a call to function *apply* on function f on object a. Where the first parameter of *apply* is a and second is the argument of spread. The sub-terms of function calls on objects with spread operators thus have to be analyzed to determine the call scope of the called function.

This introduces a new binding ($\_b\$a$).

Depending on bindings For correct specification compliance the argument of the spread operator has to be transformed to a consumable array. This behavior can be simulated through a run-time function.

Analysis of sub-terms When the pattern $< Expressione > (< ArgExpression", " * args >)$ is matched e has to be analyzed to detect whether it is a function, or object property function. (See Appendix B.5)

Decomposable The operator can be used in two places (places of multiple arguments, and multiple elements), which can be transformed separately.

B.6 Default parameter

The ECMAScript 6 spec defines a way to give parameters default values[4, 9.2.12]. These values are used if the caller does not supply any value on the position of this argument. Any default value is evaluated in the scope of the function (i.e. *this* will resolve to the run-time context of the function the default parameter value is defined on).

Category This language feature can be eliminated by transformation to a core concept of the ECMAScript 5 language. By binding the variable in the function body to either the default value (in case the parameter equals $undefined$) or to the value supplied by the parameter.

Introducing bindings This transformation does not introduce any bindings because the bindings already exist as formal parameters.

B.7 Rest parameter

The *BindingRestElement* defines a special parameter which binds to the remainder of the arguments supplied by the caller of the function.[4, 14.1]

Category The rest element can be retrieved in the function's body through the use of a *for* loop. All arguments of a function are stored inside the implicit variable *arguments*[4](*arguments* is an array like object). The rest argument is a slice of the *arguments* object, where the start index is the number of parameters (minus the rest parameter) of the function, and until the size of *arguments*. This slice can be obtained by a *for* loop.

Rest parameter is eliminated to core ECMAScript 5 language construct, this feature is syntactic sugar.

Syntactically type preserving $< Function >$ to $< Function >$

B.8 Template Literal

Standard string literals in JavaScript have some limitations. The ECMAScript 6 specification introduces a template literal to overcome some of these limitations[4, 12.2.8]. Template literals are delimited by the ‘ quotation mark, can span multiple lines and be interpolated by expressions: `$<Expression>`

Category Template strings are syntactic sugar for Strings concatenated with expressions and new-line characters.

Syntactically type preserving
`<Expression <TemplateLiteral>>` to `<Expression <StringLiteral>>`

B.9 Tail call optimization

JavaScript knows no optimization for recursive calls in the tail of a function, this results in the stack increasing with each new recursive call overflowing the stack as a result. The ECMAScript 6 specification introduces tail call optimization to overcome this problem[4, 14.6]

Category Optimization, this transformation improves the space performance of a program.

Abstraction level Tail call optimization is a semantic based transformation. To identify if the expression of a return statement is a recursive call to the current function we need name binding information.

Pure semantics This extension is purely semantic, no new syntax is introduced. To transform this extension to work in older JavaScript interpreters the tail call has to be removed and replaced by a `while` loop, to prevent the call stack from overflowing.

```
1 function fib(n,nn,res) {
2   if( nn > 100 ) return res;
3   return fib(nn,n + nn,res + [n,nn]);
4 }
```

Listing B.3: Function with tail recursion

```
1  function fib(arg0,arg1,arg2) {
2    while (true) {
3      var n = arg0, nn = arg1, res = arg2;
4
5      if(nn > 100) return res;
6      arg0 = nn;
7      arg1 = n + nn;
8      arg2 = res + [n,nn];
9    }
10 }
```

Listing B.4: Semantically identical function, without tail recursion

Figure B.4 illustrates how a function with tail recursion can be transformed to a function with an infinite loop. Not filling the call stack with new return positions and thus executing correctly even when our upper limit (100) is increased.

B.10 Generators

The ECMAScript 6 standard introduces a new function type, generators[4, 14.4]. These are functions that can be exited and later re-entered, when leaving the function through *yielding* their context (i.e. variable bindings) is saved and restored upon re-entering the function. A generator function is declared through `function*` and returns a Generator object (which inherits from *Iterator*)

Category A generator can be implemented as a state machine function wrapped in a little run-time returning a Generator object. It does categorize as syntactic sugar, however the transformation is complex (because of identification of control flow of body statement, see Analysis of sub-terms).

Scope Including a run-time the transformation can be managed local-to-local.

Syntactically type preserving

`<Statement>` to `<Statement>`

`<Expression>` to `<Expression>`

Analysis of sub-terms The control flow of the functions body has to be identified. The control flow graph can be interpreted as a state machine with several leaps, entry points, and exit points that result from the use of different control-flow constructs (e.g. *yielding*, *returning*, *breaking*, etc.). After transformation this state machine can be represented in a `switch` statement. All bindings of the function body have to be hoisted outside of this state machine, and only to be assigned through an assignment expression in the state machine.

Preconditions The `<GeneratorBody>` has to be transformed before transformation of the Generator function starts.

Depending on bindings The state machine function is wrapped in a run-time function to initialize the Generator object returned from a generator function. The run-time makes sure the state-machine has correct scoping bindings on each entry[5]

B.11 Let and Const declarators

JavaScript's scoping happens according to the lexical scope of functions. Variable declarations are accessible in the entire scope of the executing function, this means no block scoping. The `let` and `const` declarator of the ECMAScript 6 specification change this. Variables declared with either one of these is lexically scoped to its block and not its executing function. The variables are not hoisted to the top of their block, they can only be used after they are declared. Code before the declaration of a `let` variable is called the temporal dead-zone. `const` declaration are identical to `let` declarations with the exception that reassignment to a `const` variable results in a syntax error. This aids developers in reasoning about the (re)binding of their const variables. Do not confuse this with immutability, members of a `const` declared object can still be altered by code with access to the binding.

Category It is possible to eliminate `let` declarations without losing correct block scoping of bindings. We can illustrate this with the help of a simple example. 0 is bound to `x`, now we bind 1 to `x` in a nested block. Outside of this block 1 is still bound to x.

```
1 let x = 1;
2
3 {
4   let x = 0;
5   x == 0; // true
6 }
7
8 x == 0; // false
```

This behavior can be emulated in ECMAScript 6 with the use of immediately invoking function expressions (or IIFE), the function expression introduces a new function scope. And the rebinding of x is not 'visible' outside of the new IIFE.

[5] http://github.com/facebook/regenerator/blob/master/runtime.js

```
1 var x = 1;
2
3 (function() {
4   var x = 0;
5   x == 0; // true
6 })();
7
8 x == 0; // false
```

Thus *let* declarations can be desugared if every block is transformed to a `closure` with the help of IIFE's. This does however imply a serious performance overhead on the transformed program (instead of normal block execution the interpreter has to create a closure for every block). This also breaks the standard control-flow of our program when the block uses any control-flow statement keywords (e.g. `return`).

There is another way to transform `let` declarations, for this implementation we need full name analysis of a program. The previous example can also be transformed by renaming the rebinding of `x`:

```
1 var x = 1;
2
3 {
4   var x_1 = 0;
5   x_1 == 0; // true
6 }
7
8 x == 0; // false
```

The analysis has access to both the binding graph of the program with function-scoping and `var` declarations, and the binding graph with block scoping and `let` declarations. After the creation of these graphs an illegal reference can be defined as a reference that does appear in the function-scope binding graph, but does not appear in the block-scope binding graph. In the case of our example this means that `x` references 0 bound to `x` in block (at line 4) when analyzing function-scope bindings. But `x` does not reference the same binding when analyzing the block-scope binding graph. [6]

Scope The transformation needs to analyse the entire program (global) and make transformations to its entirety (global). Thus the scope of the transformation is global-to-global.

Compositional The transformation of `let` (or `const`) bindings is not compositional because bindings of these types can not be captured by names in sibling scopes

```
1 {
2   let x = 0;
3 }
4 {
5   let x = 1;
6 }
```

[6] More examples: http://raganwald.com/2015/05/30/de-stijl.html

Once these `let` bindings are transformed to `var` bindings without renaming they would be declared in the same lexical scope. Whereas they are not at the moment. For a transformation to `var` bindings to work one of these bindings and its uses has to be renamed.

Introducing bindings The transformation introduces no new bindings, it converts `let` and `const` bindings to `var` bindings, and renames these bindings (and their references) where necessary.

Preconditions Other transformations need to be performed before the let const extension is transformed. To prevent this extension to have dependencies on other extensions. For example the for-of extension has a dependency on the let-const extension:

```
1 for( let x of arr ) <Statement>
```

If the for-of loop is transformed to a normal loop let-const needs not to be aware of the for-of loop as an extension, but just of the ECMAScript 5 manner to declare variables inside loops.

Dividable Variables declared through let and const can be transformed to ECMAScript 5 code separately.

Bibliography

[1] Paul Klint, Tijs van der Storm, and Jurgen Vinju. RASCAL : a Domain Specific Language for Source Code Analysis and Manipulation.

[2] Sebastian Erdweg, Tijs van der Storm, and Yi Dai. Capture-Avoiding and Hygienic Program Transformations (incl. Proofs). 8586:1–29, April 2014. URL http://link.springer.com/10.1007/978-3-662-44202-9http://arxiv.org/abs/1404.5770.

[3] Eelco Visser. A Survey of Rewriting Strategies in Program Transformation Systems. *Electronic Notes in Theoretical Computer Science*, 57:109–143, December 2001. ISSN 15710661. doi: 10.1016/S1571-0661(04)00270-1. URL http://linkinghub.elsevier.com/retrieve/pii/S1571066104002701.

[4] Ecmascript 2015 language specification. final draft 38, Ecma International, apr 2015.

[5] SL Peyton Jones, RJM Hughes, L Augustsson, D Barton, B Boutel, W Burton, J Fasel, K Hammond, R Hinze, P Hudak, T Johnsson, MP Jones, J Launchbury, E Meijer, J Peterson, A Reid, C Runciman, and PL Wadler. Report on the programming language haskell 98, February 1999. URL http://research.microsoft.com/apps/pubs/default.aspx?id=67041.

[6] B. M. Leavenworth. Syntax macros and extended translation. *Communications of the ACM*, 9(11):790–793, 1966. ISSN 00010782. doi: 10.1145/365876.365879.

[7] Tim Disney, Nathan Faubion, David Herman, and Cormac Flanagan. Sweeten your JavaScript. In *Proceedings of the 10th ACM Symposium on Dynamic languages - DLS '14*, pages 35–44, New York, New York, USA, 2014. ACM Press. ISBN 9781450332118. doi: 10.1145/2661088.2661097. URL http://dl.acm.org/citation.cfm?doid=2661088.2661097.

[8] Daniel Weise and Roger Crew. Programmable syntax macros. *ACM SIGPLAN Notices*, 28(6):156–165, 1993. ISSN 03621340. doi: 10.1145/173262.155105.

[9] Martin Bravenboer and Eelco Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04*, pages 365–383, 2004. ISSN 0362-1340. doi: 10.1145/1035292.1029007.

[10] David Herman and Mitchell Wand. A Theory of Typed Hygienic Macros. *Proceedings of the 17th European Symposium on Programming*, 4960:48, 2010. URL http://www.springerlink.com/index/V03K0734Q7217R35.pdf.

[11] David Herman and Mitchell Wand. A theory of hygienic macros. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 4960 LNCS, pages 48–62, 2008. ISBN 3540787380. doi: 10.1007/978-3-540-78739-6_4. URL http://www.springerlink.com/index/V03K0734Q7217R35.pdf.

[12] Sebastian Erdweg and Felix Rieger. A framework for extensible languages. *ACM SIGPLAN Notices*, 49(3):3–12, March 2014. ISSN 03621340. doi: 10.1145/2637365.2517210. URL http://dl.acm.org/citation.cfm?doid=2637365.2517210.

[13] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation - PLDI '88*, pages 199–208, 1988. ISBN 0897912691. doi: 10.1145/53990.54010. URL http://portal.acm.org/citation.cfm?doid=53990.54010.

[14] Eelco Visser. Scannerless Generalized-LR Parsing. (P9707), 1997.

[15] M van den Brand, Jeroen Scheerder, J Vinju, Eelco Visser, and M Van Den Brand. Disambiguation filters for scannerless generalized LR parsers. *Compiler Construction*, pages 143–158, 2002. doi: http://link.springer.de/link/service/series/0558/bibs/2304/23040143.htm. URL http://www.springerlink.com/index/03359K0CERUPFTFH.pdf.

[16] D. J. Salomon and G. V. Cormack. Scannerless NSLR(1) parsing of programming languages. *ACM SIGPLAN Notices*, 24(7):170–178, 1989. ISSN 03621340. doi: 10.1145/74818.74833.

[17] Martin Fowler. Language workbenches: The killer-app for domain specific languages. *Accessed online from: http://www. martinfowler. com/articles/languageWorkbench. html*, pages 1–27, 2005. doi: http://www.martinfowler.com/articles/languageWorkbench.html. URL http://www.issi.uned.es/doctorado/generative/Bibliografia/Fowler.pdf.

[18] M P Ward. Language Oriented Programming. *Software-Concepts and Tools*, 15(4):1–22, 1994.

[19] Matthias Felleisen. On the Expressive Power of Programming Languages. *Science of Computer Programming*, pages 131 – 151, 1990. ISSN 01676423. doi: 10.1016/0167-6423(91)90036-W.

[20] Pablo Inostroza, Tijs van der Storm, and Sebastian Erdweg. Tracing Program Transformations with String Origins. pages 154–169. 2014. doi: 10.1007/978-3-319-08789-4_12. URL http://link.springer.com/10.1007/978-3-319-08789-4_12.