Master Thesis

Tool-supported language extensions for JavaScript

Author:
Matthisk Heimensen
m@tthisk.nl

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

August 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

Master of Science

Tool-supported language extensions for JavaScript

by Matthisk Heimensen

Extending programming languages is an activity undertaken by many programmers, mostly through the use of syntax macros but there exist different manners through which to achieve extensible programming languages. Extension enables any programmer to introduce new language constructs, not just the creators of a language. Language extensions are dedicated modular grammars including a transformation step to the base language. They enable programmers to create their own syntax on top of a base language and embed domain specific knowledge not just in semantics but also in syntax. For implementation of language extensions a tool called a language workbench can be used. Language workbenches promise to provide an environment which improves the efficiency of language oriented programming. To evaluate this promise of the language workbench we implement the latest JavaScript specification (ECMAScript 6) as a set of language extensions on top of current JavaScript and compare our implementation against tools created without the help of a language workbench.

Contents

Chapter 1

Introduction and Motivation

Language extension allow programmers to introduce new language constructs to a base language, with two main purposes. *"First a programmer can define language extensions for language constructs that are missing in the base language"* [1], these *missing* constructs can range from more advanced looping constructs (e.g. *foreach/for-of* loops) to shorthand function notation (e.g. lambda functions). *"Second, extensible programming languages serve as an excellent base for language embedding"* [1] for instance enable the use a markup language (e.g. HTML) inside a programming language. Program transformations is one of the techniques to implement language extensions, they are used to transform language extensions from the source program to base language code. Many systems for program transformation exist but in recent years a specific type of tool has become more popular for this job, the language workbench. This tool aids the (meta-)programmer in creating meta-programs that integrate with modern IDEs. In this thesis we investigate the ability of language workbenches in helping the meta-programmer to create a large set of language extensions.

As an experiment we extend the JavaScript programming language with features introduced by the new specification document of the language, ECMAScript 6 (ES6). The current specification of JavaScript implemented in all major run-times (be it web-browser or dedicated) is ECMAScript 5 (ES5). The language extensions are created with the Rascal [**?**] language workbench and the resulting tool is named RMonia. A second contribution of this thesis is a taxonomy for language extensions. With this taxonomy we try to capture the distinctive characteristics of each language extension in a generic way. With the help of this taxonomy we try to answer the following questions: How can the language extension be implemented, what information does transformation of a language extension require, and what are guarantees can we give of the target program produced by a language extension?

To evaluate the language workbench for the task of extending programming languages through language extensions we implement the latest JavaScript specification (ES6) as a set of language extensions on top of current JavaScript. Because of the popularity of the JavaScript programming language there already exist several implementations of ES6 as language extensions for ES5 JavaScript, implemented without the help of a language workbench, we use these implementations to evaluate the effectiveness of the language workbench. Measures used to evaluate our implementation

are based on coverage of the transformation suite (i.e. amount of ES6 features implemented by the transformer), correctness of the transformations, size (i.e. source lines of code), modularity of the language extension suite, and noise generated by the transformation.

The Rascal language workbench made it possible for us to implement ES6 language extensions in a short time period with fewer lines of code. We cover almost all new language features from ES6 in RMonia, something only other large-scale open-source projects are able to achieve. We deliver editor support for reference hyperlinking, undeclared reference errors, illegal redeclaration errors, and hover documentation preview of target program. The language workbench does however constraint us to one specific IDE and our solution is less portable than other implementations. Syntax definition of the language workbench constrained us from implementing empty element matching in the destructuring (see appendix B.3) language extension of ES6 (see appendix A).

1.1 Outline

This thesis is structured as follows. In chapter 2 we present an analysis of the problem studied in this thesis. Chapter 3 presents background information of program transformations and the language workbench. In chapter 4 we present a taxonomy for language extensions. Chapter 5 discusses the implementation of RMonia. In chapter 6 we evaluate our implementation against other implementations. Finally we conclude in chapter 7.

Chapter 2

Problem Analysis

2.1 Motivation

What are the reasons to create program transformations or extend current programming languages, as opposed to just creating entirely new programming languages? Extensible programming languages allow programmers to introduce custom language features as an extension on a base language. There are two main reasons for programmers to use program transformations. *"First, a programmer can define language extensions for language constructs that are missing in the base language"* [1], for example default parameters, type annotations, or list comprehensions in JavaScript. *"Second, extensible programming languages serve as an excellent base for language embedding"* [1] this refers to the embedding of domain-specific languages inside a host language, for example using HTML markup inside of JavaScript with JSX[1].

The implementation of RMonia focuses on the first motivation for the extension of programming languages, to introduce missing language constructs to a base language, because the ES6 specification only introduces new language constructs and does not embed any domain-specific languages in JavaScript. Similar techniques as to those used to implement RMonia can be used for the embedding of domain-specific languages in a host language, our research thus applies to both motives. We will evaluate the language workbench (see section 3.6) for the task of creating such language extensions. With the help of an evaluation of our language extension set we will measure the effectiveness of the language workbench along several different dimensions:

- Difficulty of implementing language extensions
- Ease with which language extensions integrate with the development environment of the programmer
- Modularity of language extensions, i.e. is it easy or hard to add new language extensions on top of each other and on top of the base language.

JavaScript is an interpreted programming language (see section 3.1) used to run code inside web-browsers to power interactivity of websites. JavaScript programmers are

[1] https://facebook.github.io/jsx/

forced to use the JavaScript version implemented in the majority of their users web-browsers. Because of this they can not benefit from language features introduced by new versions until most of their users have updated their browsers to a version implementing this new standard. Language extensions can solve this issue by transforming the new language constructs to constructs available in the majority of their users web-browsers.

2.2 Research approach

Our research exists out of an experiment and an evaluation. The experiment is an implementation of a subset of the ES6 specification as language extensions on top of ES5 JavaScript as a base language, this implementation is our research's artifact (see appendix A), named *RMonia*. It is implemented in the Rascal programming language (see section 3.6). Second, we evaluate the language workbench for the task of implementing language extensions. For this evaluation we compare our implementation to other transformers that transform ES6 JavaScript to ES5 JavaScript.

2.3 Research questions

The central research questions of this thesis are:

1. How *effective* is the language workbench for the task of implementing language extensions?
2. How can program transformations be categorized and give insight into differences between language extensions?

Where effectiveness is established along several dimensions, ease with which the extensions can be implemented (i.e. time and source lines of code needed for transformation code), modularity of implementation i.e. is it easy or hard to add new language extensions on top of each other and the base language. With the effectiveness established we determine the engineering trade-offs of using the language workbench for implementation of language extensions.

2.4 Results

- A taxonomy for language extensions
- Language extension suite implementing (a subset of) the ES 6 features
- A set of compatibility tests, testing our implementation against the specification document
- Eclipse IDE integration (syntax highlighting & declared at hyperlinks) for new language features
- Set of engineering trade-offs applicable to implementing language extensions inside the language workbench

A set of language extensions for the JavaScript programming language created with the aid of a language workbench. First, we expect that the use of language workbench will reduce the amount of work needed to implement these language extensions. Second, we expect that the use of a language workbench will make our implementation more modular, making it easier to add additional language extensions on top of other extensions or on top of the base language.

The taxonomy of language extensions and the categorization created with the help of this taxonomy will provide us with useful insights in the inner working of these language extensions as program transformations. And will help in establishing a view of the key characteristics of each language extension.

We expect to get insight into the engineering trade-offs of implementing language extensions inside the language workbench. What limitations does the workbench impose, where can we benefit from the power of a language workbench, and what are the generic problems of language extensions.

2.5 Related Work

Many researchers have investigated the possibility to make programming languages extensible [1, 2], be it through language extensions, compiler extensions [3, 4], or syntax macros [5–8]. Often their research also tries to extend the tools with which programmers write their code (i.e. IDEs). Here we discuss several different approaches to programming language extension presented in the past.

Object oriented programming languages are designed in such a way that libraries created in the language encapsulate knowledge (in semantics) of a specific domain for which the library is created. Programmers of such libraries are not allowed to encapsulate knowledge of the domain in syntax. To allow the extension of programming languages to encapsulate domain knowledge in syntax Eelco Visser created a system for language extension through concrete syntax named MetaBorg [2]. The system is realized through Syntax Definition Formalism [9] and the Stratego [10] transformation language. The system is modular and composable this makes it possible to combine different language extensions. The paper discusses three domains which *"suffer from misalignment between language notation and the domain: code generation, XML document generation, and graphical user-interface construction."* [11] To test MetaBorg, DSLs for these three domains are implemented and integrated with the Java programming language as a host language. The work of Visser and Bravenboer relates closely to RMonia presented in this thesis. Visser ignores the issues related to introduction of new bindings (i.e. variable capture). MetaBorg is not evaluated by quantitative measures against other tools for language extension.

Another system based on SDF and Stratego is presented by Erdweg et. al. [12] called Sugar*. This is a system for language extension agnostic from the base language, the system can extend- syntax, editor support, and static analysis. In contrast to the system presented in this thesis, Sugar* relies on information from a compiler of the base language to operate correctly. To evaluate the Sugar* system measures based on lines of code are applied to language extensions created for five different base languages (Java, Haskell, Prolog, JavaScript, and System F_ω). Sugar* uses Spoofax to create editor support for language extensions, similar to a language workbench. Sugar* is based on SugarJ [13],

a library based language extension tool for the Java programming language. SugarJ is evaluated against other forms of syntactic embedding on five dimensions defined as design goals for SugarJ. They compare against string encoding, pure embedding, macro systems, extensible compilers, program transformations, and dynamic meta-object protocols.

Importing multiple libraries that include new syntax constructs could result in syntactic ambiguities among the new constructs. The Sugar* and SugarJ framework have no way to resolve these ambiguities and clients of the libraries will have to resolve these ambiguities themselves, requiring a reasonable thorough understanding of the underlying parser technology [14]. Omar et. al. [14] sidestep this problem in their research by introducing an alternative parsing strategy. Instead of introducing syntax extensions on top of the entire grammar, extensions are tied to a specific type introducing *type-specific languages* (TSL). TSL's are created with the use of layout sensitive grammars, presenting a new set of challenges not faced in this thesis because we do not rely on layout sensitive grammars.

All the tools discussed above (MetaBorg, Sugar*, and SugarJ) rely for their evaluation on a very small subset of actual implemented language extensions, where most are based on embedding a domain-specific language in a host language. In this thesis we try to evaluate the language workbench for the job of language extensions with the help of a large set of implemented extensions not based on embedding domain-specific languages but introducing new language features.

The extension of the JavaScript programming language is studied by Disney et. al. [5], here JavaScript is extended through the use of syntax macros instead of separate language extensions. Their work focuses more on separating the parser and lexer to avoid ambiguities during parsing (a problem we avert by using a scannerless parser see section 3.4). They give no evaluation of the expressiveness of the resulting macro system. A project trying to implement ES6 features as macros using their macro system exists but has been abandoned[2]

In related work of our taxonomy of language extensions (see chapter 4) we can identify several categorizations of program transformations. The Irvine program transformation catalog [**?**] present a set of useful program transformations for lower-level procedural programming languages (e.g. the C programming language). Visser [15] presents a taxonomy for program transformations, which we partly reuse in our taxonomy of language extensions. This taxonomy can be used for program transformations in general where our taxonomy is used to categorize language extensions, which can be implemented with the use of program transformations but are not limited to them.

Erdweg et. al. [16] present the *name-fix* algorithm, a generic language parametric algorithm that can solve the issue of variable capture that arises after program transformations stand-alone from transformation code. In this thesis we reuse a similar system to ensure no variable capture arises during transformation. Erdweg et. al. prove the correctness of their implementation, something we have not performed for the implementation we reuse. A survey of domain-specific language implementations gives valuable insight into variable capture to be found in program transformations. Transformation hygiene has also received a lot of attention in the domain of macros [5, 6, 17].

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

Chapter 3

Background

3.1 ECMAScript

JavaScript[1]is the programming language specified in the ECMAScript specification[18]. JavaScript is an interpreted, object-oriented, dynamically typed scripting language with functions as first-class citizens. It is mostly used inside Web browsers, 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 3.1). The ES standard specifies automatic semicolon insertion (or ASI), this grammar feature makes it possible for programmers to omit the the trailing semicolon after statements and let the parser insert these automatically.

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 their inheritance.

In the JavaScript run-time environment everything is an object with properties (even functions see section 3.1), 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

[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++).

continue until either prototype *null* is found or an object with property x is found in the prototype chain.

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, independent of their placement within this function (this is called hoisting).

One exception on the lexical scoping is 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 run-time).

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, everything is else is an object:

- *Boolean*
- *null*

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

- *undefined*
- *Number*
- *String*

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 3.1).

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 3.1: 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 not possible.

3.2 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*" ([15])

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 migration [15]. Here a program in a source language is transformed to another language at the same level of abstraction, in our case a different version of a specification of the same language (i.e. ES6 to ES5).

3.3 Language Extensions

Language extensions introduce new syntactic constructs to a base language. They can be implemented with the use of program transformations, the transformations transform

[4]For more information see Mozilla Developer Network.

a target program with language extension to a program executable in the base language. *"Many compilers first desugar a source program to a core language."* [16] For example the Haskell functional programming language defines for many constructs how they can be transformed to a kernel language [19]. These pieces of *syntactic sugar* are examples of language extensions and the transformation used to transform this syntactic sugar to a base language is called a *desugaring*.

Language extensions are closely related to other forms of program transformations. There are (syntax) macros[7], these are language extensions defined within the same source file and are expanded before compilation (examples are SweeterJS [5], Syntax macros [8]). These *"define syntactic abstractions over code fragments"* [2]. Macro systems have several limitations. The transformations need to be present in the source file, macros are often restrained to a fixed invocation syntax (through a macro identifier), and macros often lack expressiveness in their rewriting language[2].

Some language extensions classify 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 with the aid of syntactic sugar often benefit from improved human readability. For example the assignment expression `a += b` is syntactic sugar notation for `a = a + b` this notation is often found in C-style programming languages.

3.3.1 Hygiene

Any program transformation that introduces new bindings in the target program risks the unintended capture of user identifiers. When variable capture happens as a result of a program transformation, this transformation is unhygienic [16]. 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 [5, 17, 20]

Variable capture can arise in two different forms, one originates from bindings introduced by a transformation, the other is introduced because a transformation relies on bindings from the source program or the programming languages standard library. Here we will illustrate both forms with the help of an example.

Lets introduce a language extension called *swap*, this extension swap can swap the values of two variables with the use of a single statement. The transformation transforms the `swap` statement to an immediately invoking function expression (see section 3.1) 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 3.1: 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 one of the arguments of the `swap` statement is named `tmp` the transformed code will capture the users binding and not produce the expected result (see figure 3.2).

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

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

FIGURE 3.2: Introduction of variable capture because the transformation binds non-free name

The second type of variable capture that can be introduced during a transformation happens during transformations from which the target program depends on binding from either the source program or the global scope. The `log` language extension transforms a `log` statement to a call of log on the global console object:

```
1  log "message";
2
```

```
1  console.log("message");
2
```

FIGURE 3.3: A log language extension

If the console object is redefined in the source program the reference to the original global object is captured and the target program will not behave as expected:

```
1  var console = ...;
2  log "message";
3
```

```
1  var console = ...;
2  console.log("message");
3
```

FIGURE 3.4: Variable capture with the log language extension

The unhygienic transformations 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 implementations of a domain-specific language were prone to unintended variable capture [1].

3.4 Parsing

Before a program is represented in a tree format, 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[21–23] here we will discuss a few relevant for our thesis.

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 grammar.

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."* [5]. 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."* [21] these parsers preserve the lexical structure of the input characters, and make it possible to compose grammars [21].

There exist different types of parsers, where some can handle more grammars than others. Examples of parser classes are LR (left to right), SLR (simple left to right), LALR (look-ahead 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. This parse table is used during execution of the parser to identify the possible next symbols in the current context.

Scannerless parsing suffers form conflicts in the parse table [21]. 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 forking 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.

3.5 Program Representation

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

Parse Trees Parse trees represent a program's 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.

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.

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)[24]. 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"* [15]

3.6 Language Workbenches

Language workbench is a term popularized by Martin Fowler and we can summarize 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 [25].

These workbenches improve the efficiency of language oriented programming [26] significantly, by giving programmers the the tools to define and extend programming languages while integrating with their developer tools.

The Rascal [**?**] meta-programming environment is a language workbench. It allows programmers to define context-free grammars, and 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 3.4). Rascal uses generalized parsing to avoid lack of look-ahead (see section 3.4).

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 readability of transformation code

A concrete-syntax pattern may contain variables in the form of a grammar's non-terminals, these variables are bound in the context of the current pattern if the pattern is matched successfully. A quoted concrete-syntax pattern starts with a non-terminal to define the desired syntactic type (e.g. Expression), and they ignore layout. A concrete-syntax pattern can contain tokens, 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. Here we provide an example of a concrete syntax pattern matching the `swap` statement.

```
1 (Statement)`swap <Id x> <Id y>` := pt
```

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

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

Chapter 4

Taxonomy

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, such a taxonomy can be used for several purposes. The taxonomy helps identifying the following aspects of language extensions, (1) implementation intricacies of language extensions, (2) Implementation type (e.g. through the use of a program transformation), (3) What is the relation between language extensions. In appendix B we categorize ES6 language features according to this taxonomy.

4.1 Structure

The several dimensions of the taxonomy give insight in the answers to three questions specific to a language extension.

How can the language extension be implemented?
There are multiple ways to implement a language extension (e.g. a program transformation or a compiler extension). Sometimes one solution is preferable over another, the following dimensions help in selecting an implementation type for a language extensions:

- Category
- Abstraction level
- Extension or modification
- Dependencies
- Decomposable

What information does transformation of a language extension require?
Transformations rely on information in the source program to be able to perform transformations. Some require more information than others, this additional information required can make implementation of the language extension more complex. The following dimensions help in identifying the amount of context needed for a language extension to be transformed:

- Category
- Compositionality
- Analysis of subterms
- Scope

What are guarantees can we give of the target program produced by a language extension?
Transformation of a language extension produces a target program different from the source program. The following dimensions help identifying to what extend target and source program differ from each other, giving insight into the inner working of a transformation:

- Syntactically type preserving
- Introduction of bindings
- Depending on bindings
- Backwards compatible

4.2 Dimensions

The following dimensions are identified and used to categorize every language extension. Each of the following paragraphs investigates a relevant question to language extensions and provides insight into answering these question with the use of examples.

Category One of the rephrasing categories defined by Eelco Visser [15], rephrasings are program transformations where source and target program language are the same. Here we discuss each category.

Normalization The reduction of a source program to a target program in a sublanguage 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 an 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 running time of the program.

Other

Refactoring	is a disciplined technique for restructuring an existing body of code, alteringits internal structure without changing its external behaviour.[1]
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 [15]

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 3.5). In addition to the syntactic representation semantic transformations also have access to the static semantics of the input program (e.g. variable binding).

Extension or Modification Rephrasings try to say the same thing (i.e. no change in semantics) but using different words[15]. 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* category (see table **??**) 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.10).

[1] http://www.refactoring.com

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:

1. When a program transformation matches on a sub-tree of the parse-tree and only transforms this matched sub-tree it is a *local-to-local* transformation.
2. If the transformation needs information outside the context of the matched sub-tree, but only transforms the matched sub-tree it is *global-to-local*.
3. When a transformation has no additional context from its local sub-tree but does alter the entire parse-tree it is called *local-to-global*.
4. If the transformation transforms the input program in its entirety it is *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 3.1). If a transformation matches on a Expression node but returns a Statement it is not syntactically type preserving.

Introduction of bindings Transformations that do not introduce bindings are guaranteed to be hygienic (see section 3.3.1), where binding introducing transformations can cause variable capture from synthesized bindings to source bindings.

Depending on bindings (i.e. runtime system) 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.

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 terms used inside of the language extension's non-terminals. For example we can only use identifiers in the sub-terms of our `swap` language extension (see section 3.3.1).

Analysis of sub-terms Are the non-terminals of our language extension analyzed by the transformation rule. This is related to compositionality but differs because non-compositional transformations analyze *and transform* sub-terms.

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 all valid base language code also valid inside an environment where the base language is extended with this language extension. For example the *let-const* (see appendix B.12) language extension introduces a new keyword `let`, because of this code written in the base language could break when parsed with the *let-const* language extension enabled. For instance `let[x] = 10;` would be correct ES5 JavaScript code in ES6 this is however not allowed because `let` is a reserved keyword.

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

TABLE 4.1: 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.	-	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, **S**: simplification, **Opt**: Optimization, **CfS**: Context-free-syntax, **L2L**: local-to-local, **E**: Extension, **M**: Modification

Chapter 5

Implementation RMonia

Each transformation of a language extension is defined as one or more rewrite rules. 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. ES 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.

5.1 Basics

To get a better understanding of the inner working of RMoniawe discuss the implementation of the `swap` language extension (see section 3.3.1). The language extension introduces a new statement

```
1 module core::Syntax
2
3 syntax Statement
4   = ...
5   | ...
6   ...;
```

Listing 5.1: Core syntax

To extend the list of possible statements we create a new Rascal module that extends the core syntax definition and defines a new syntax rule for *Statement* named *swap*. With this new syntax rule our parser is able to correctly parse `swap` statements.

```
1 extends core::Syntax;
2
3 syntax Statement = swap: "swap" Id Id ";";
```

Listing 5.2: Swap statement syntax

Before we create a function that transforms the `swap` statement to JavaScript code we introduce a simple visitor and default desugar function. A default desugar function for statements is defined which is the id function on a statement. To desugar an entire parse

FIGURE 5.1

```
Statement desugar( (Statement)`swap <Id x> <Id y>` )
    = (Statement)
            `(function() {
             var tmp = <Id x>;
             <Id x> = <Id y>;
             <Id y> = tmp;
            })();`;
```

tree we only have to visit each node (bottom up) and invoke the desugar function for each statement that is found by the visitor.

```
1 default Statement desugar( Statement s ) = s;
2
3 Source runDesugar( Source pt ) {
4   return visit(pt) {
5     case Statement s => desugar(s)
6   }
7 }
```

LISTING 5.3: Desugar visitor

To desugar the new syntax construct to JavaScript code we overload the previously defined desugar function with a pattern match on *Statement* to match the use of the `swap` statement and translate it to an immediately invoking function expression (IIFE) in which we rebind both supplied identifiers to each other.

The argument of desugar is a pattern match on parse-tree nodes of the type `Statement` matching on `swap` statements, if a match is found the identifiers supplied to the statement are bound respectively to `x` and `y`. The function returns a piece of concrete syntax which in turn is of the type `Statement` where the supplied identifiers are rebound to each other. Remember that the supplied identifiers to the `swap` statement could have any name, what would happen if one of the identifiers was name `tmp`?

5.2 RMonia

In *RMonia* we implement a large subset of ES6 features as language extensions on top of a core syntax describing ES5. Here we discuss several parts of interest from our implementation. Everything with the exception of let-const is based on the simple example described in the previous section.

5.2.1 Visitor

In our basic example we used a function `runDesugar`, this is the visitor of the parse-tree and is in principle how the visitor of RMonia works. Differences are we also match on expressions, functions, and the source node (root node) this final match can be used by

transformations that are performed global-to-local or global-to-global. If we have multiple language extensions performing a global-to-local or global-to-global transformation or if a language extension's transformation introduces a language construct that should be desugared either by itself or another language extension, one pass over the parse tree is not enough. For extensions with these kind of transformations we need to revisit the tree until no more changes are applied by desugar functions. For example the use of a destructuring assignment pattern inside a *for-of* loop's binding will be desugared to a variable declaration inside the loop's body (see section 5.2.3), relying on a second pass over the parse tree to desugar the assignment pattern inside the variable declaration:

```
for(var [a,b] of arr) {
  <Statement* body>
}
```

```
for(var i = 0; i <= arr.length; i++) {
  var [a,b] = arr[i];
  <Statement* body>
}
```

```
for(var i = 0; i <= arr.length; i++) {
  var _ref = arr[i],
      a = _ref[0],
      b = _ref[1];
  <Statement* body>
}
```

The final desugar visitor and default desugar functions are as follows, where *solve* will ensure we revisit the tree until no more desugarings are performed:

```
default Source desugar( Source s ) = s;
default Function desugar( Function s ) = s;
default Statement desugar( Statement s ) = s;
default Expression desugar( Expression s ) = s;
```

Listing 5.4: Identity desugar functions

```
Source runDesugar( Source pt ) {
  return solve(pt) {
    pt = visit(pt) {
      case Source src => desugar(src)
      case Function f => desugar(f)
      case Statement s => desugar(s)
      case Expression e => desugar(e)
    }
  }
}
```

Listing 5.5: Final desugar visitor

An example of a language extension that relies on a global-to-local transformation is the arrow function. Arrow function's that reside directly in the global scope, are desugared differently from locally defined arrow functions. See appendix B.1

5.2.2 Introducing bindings

Some language extensions need to introduce new bindings in the target program to work properly. There are multiple ways for a transformation to introduce bindings in

JavaScript. When a transformation is performed on an expression an IIFE can be used to introduce new bindings only accesible in the scope of this expression.

```
1 <Expression e>
```

```
1 (function(x, y, ...) {
2   return <Expression e>;
3 })(xd, yd, ...)
```

When a transformation is performed on a statement a block statement can be used in combination with variables declared through the *let* declarator, to introduce variables only accessible in the scope of the statement. *Let* declarations are however not part of ES5 JavaScript but introduced by ES6, this assumes there is a language extension for *let* present in the current set of enabled language extensions (see appendix B.12).

```
1 <Statement s>
```

```
1 {
2   let x = xd,
3     y = yd
4     ...;
5   <Statement s>
6 }
```

In the case of a statement an IIFE can not be used to introduce new bindings because the control flow of the target program would break if the desugared statement contains a non-local control flow construct, such as `return` or `break`.

5.2.3 Mutually dependent language extensions

Language extensions can depend upon other language extensions to work correctly, this dependency makes a collection of language extensions less modular because the dependent extension always needs to be used in combination with its dependency. It would be preferable if the two extensions could be used in separation where functionality which relies on both extensions working together is only enabled in a situation where the two language extensions are enabled.

An example of language extension with mutually dependent functionality in the ES 6 specification is that of destructuring assignment (see appendix B.3) inside for-of loop's binding (see appendix B.5).

```
1 for(var [a,b] of [ [1,2], [3,4] ]) {
2   // ... do something with a and b
3 }
```

The for-of loop needs a dependency on the destructuring assignment syntax definition, to be able to create the statement syntax to allow an assignment pattern (i.e. either an array or object destructuring) inside the for-of loop:

```
1 syntax Statement
2   = "for" "(" "var" AssignmentPattern "of" Expression ")" Statement;
```

Shared language extensions need to have a non-terminal defined in the base-language which can be extended in both language extensions. In case of the for-of loop we can

FIGURE 5.2: default *declareBinding* function from *for-of* language extension

```
VariableDeclaration declareBinding( (ForBinding)`<Id id>`, Expression e )
    = (VariableDeclaration)`<Id id> = <Expression e>`;
```

FIGURE 5.3: overloaded *declareBinding* function from *destructuring* language extension

```
VariableDeclaration declareBinding( (ForBinding)`<Id id>`, Expression e )
    = (VariableDeclaration)`<Id id> = <Expression e>`;
```

introduce a new non-terminal *ForBinding* which can be extended in the destructuring assignment language extension.

```
1  // Core syntax definition
2  syntax ForBinding
3    = Id;
4
5  // For−of loop language extension
6  syntax Statement
7    = "for" "(" "var" ForBinding "of" Expression ")" Statement;
8
9  // Destructuring language extension
10 syntax ForBinding
11   = AssignmentPattern;
```

How are the transformations of both extensions affected by the fact that they are mutually dependent on each other? Before this question can be answered we need insight in how a *for-of* loop is desugared, this is explained in appendix B.5. For this example we just assume a *for-of* loop is desugared to a traditional *for* loop over the length property of the input array (ignoring some of the intricacies of *for-of* loop language extension). The following example presents the wanted desugaring where the assignment pattern binding is moved to the body of the loop.

```
1  for(var [a,b] of arr) {
2    <Statement* body>
3  }
```

```
1  for(var i = 0; i < arr.length; i++)
2  {
3    var [a,b] = arr[i];
4    <Statement* body>
5  }
```

The `ForBinding` non-terminal can consist either of an `Identifier` or an `AssignmentPattern` non-terminal. The *for-of* loop language extension is however unaware of the existence of the second option, because it is created by the *destructuring* language extension, and thus unable to correctly desugar in this case. To solve this issue we introduce a function that can be overloaded in the *destructuring* language extension to help transform the `ForBinding` to a `VariableDeclaration`, this function is named *declareBinding*.

Designing language extensions in this way makes the entire suite of language extensions more modular. Language extensions can be mutually dependent instead of having explicit dependencies on one another. This presents the possibility of selecting only the needed language extensions without being required to depend on unneeded language extensions. And at the same time it reduces the amount of code needed for the implementation of language extensions that are mutually dependent. In case of our example above if we had an explicit dependency from *destructuring* to *for-of* the desugaring function for *for-of* loops had to be overridden in the *destructuring* language extension.

5.2.4 Variable capture

In section 3.3.1 we have described the problem of variable capture in the context of program transformations. Here we discuss the algorithm reused to solve variable capture in RMonia, the algorithm reused is named *v-fix*. The algorithm relies on a binding graph to identify variable capture, and uses string origins [27] to distinguish between synthesized identifiers (i.e. those identifiers introduced by the transformation) and identifiers originating from the source program.

v-fix analyses the *name graph* of generated 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 *v-fix* algorithm uses origin tracking [27] to identify the origin of identifier (either from the source program or introduced by a transformation).

In figure 5.4 we use the target program from our `swap` language extension (see section 3.3.1) to illustrate how these name graphs are constructed, and how *v-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  })();
```

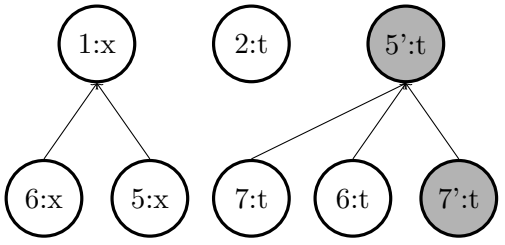


FIGURE 5.4: Target program generated by swap language extension and corresponding name graph; where *t* stands for *tmp*

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

are not renamed. The resulting name graph and target program are presented in figure 5.5

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

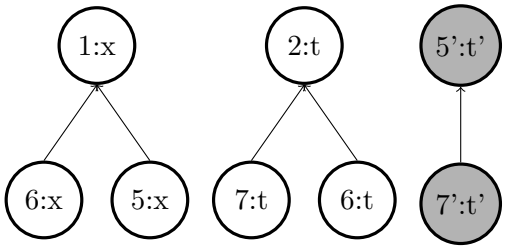


FIGURE 5.5: Target program and name graph after *v-fix* algorithm; where *t* stands for *tmp* and *t'* for *tmp$0*

v-fix does apply a *"closed-world assumption to infer that all unbound variables are indeed free, and thus can be renamed at will."* [16]. 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 program, could possibly be shadowed by the renamed bindings. However there is no way for *v-fix* to know what names will be taken by other bindings than those bound in the source program.

A second form of variable capture is introduced when the user shadows a global variable on which a target program relies. *v-fix* is also able to solve this form of variable capture, to illustrate this we use the example of the `log` language extension. The name-graph for the target program of our example (see section 3.3.1) shows a reference from an identifier introduced by our transformation to a source binding (see figure 5.6).

```
1  var console = <Expression e>;
2  console.log("message");
3  console;
```

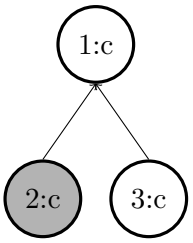


FIGURE 5.6: Target program generated by log language extension and corresponding name-graph; where *c* stands for *console*

This reference is identified as variable capture and declaration `console` is marked to be renamed (see figure 5.6.

```
1  var console$0 = <Expression e>;
2  console.log("message");
3  console$0;
```

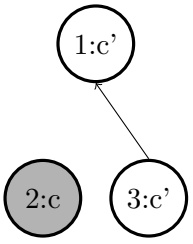
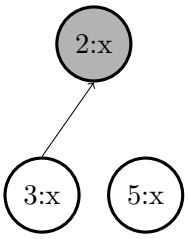
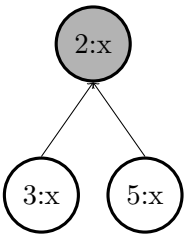


FIGURE 5.7: Target program and name-graph after application of *v-fix* algorithm; where *c* stands for *console* and *c'* for *console$0*

Notice that the *v-fix* algorithm in the example of the `swap` language extension (see figure 5.5) renames a synthesized binding. Whereas in case of the `log` language extension (see



(A) Name graph (block scope binding)



(B) Name graph (function scope binding)

figure 5.7) the *v-fix* algorithm renames an identifier originating from the source program. When identifiers from the source program are renamed external scripts relying on these bindings will no longer function correctly.

5.2.5 Block scoping

As described in section 3.1 JavaScript's bindings are function scoped, i.e. they are accessible throughout the entire lexical scope of the function they are defined in. ES6 introduces a new binding mechanism through which identifiers can be bound in a block scope (see appendix B.12). Block bound identifiers are only available after declaration within the lexical enclosing block scope they are defined in. Where normal declarations can be redeclared without errors, redeclration of block bound bindings is illegal and results in a parse error. To desugar the new block binding introduced in ES6 there are two approaches. First, replace all block binding by a *try catch* block where we throw the binding inside of the try block and catch the binding in the catch block.

```
let x = 0;
... statements
```

```
try {
  throw 0;
} catch(x) {
  ... statements
}
```

The problem with this implementation is that *try catch* blocks are not optimized by interpreters[1] and thus the performance of transformed code would be much worse than that of the source program. Second, the block binding could be achieved through the renaming of bindings to avoid reference of *let* or *const* defined bindings outside of their block scope.

```
{
  let x = 0;
  x;
}
x;
```

```
{
  var x_0 = 0;
  x_0;
}
x;
```

To resolve the naming needed for the emulation of block scope in ES5 we also use a *name-graph* as described in the previous section. To identify illegal references in the name graph we compare a name graph with block binding against a name graph without block binding, the two graphs for the example are depicted below.

The first name graph is the one created using a block binding mechanism where *const* and *let* declarations are block bound. In the second name graph all declarations are

[1] http://ryanmorr.com/reinventing-the-try-catch-block/

bound in their lexical enclosing function's scope. Now we identify the reference from *5:x* to the declaration *2:x* as illegal, because it exists in our function scoped name graph but does not occur in the block bound name graph. Declaration *2:x* is marked to be renamed. After the entire name graph is build and all illegal references are identified, all declarations and their references according to the block bound name graph are renamed.

Implementation For the implementation of the block binding resolver we use a special type of scope graph, consisting of three types of nodes. First, a root node created for the root (or global) scope of an input parse tree. This node contains an environment consisting of names with the location of their declaration. Second, each function scope is identified by a closure node. This node consists of an `Env` containing all function scoped definitions (i.e. declarations of functions or using `var` declarator). It also contains a `LEnv` which is the name-graph created where all bindings are function scoped (i.e. also declarations using `let` or `const` declarators). Third, a node block is created for each block of statements, containing all block-scoped declarations in an environment.

```
1 alias LEnv = lrel[str name, loc def];
2 alias Env = map[str name,loc def];
3
4 data Scope
5   = block( Env environment, Scope parent )
6   | closure( Env environment, LEnv closureEnvironment, Scope parent )
7   | root( Env environment );
```

Listing 5.6: Data structure for scope graph

A resolver visits the parse-tree while building a scope trees for the current path. All declaration in a function's scope are hoisted, because these declarations can be referenced throughout the entire lexical scope of the function, even before their actual declaration. Usage of an identifier by reference are resolved through a function named `lookup`:

```
1 set[loc] lookup(str name, loc use, Scope scope);
```

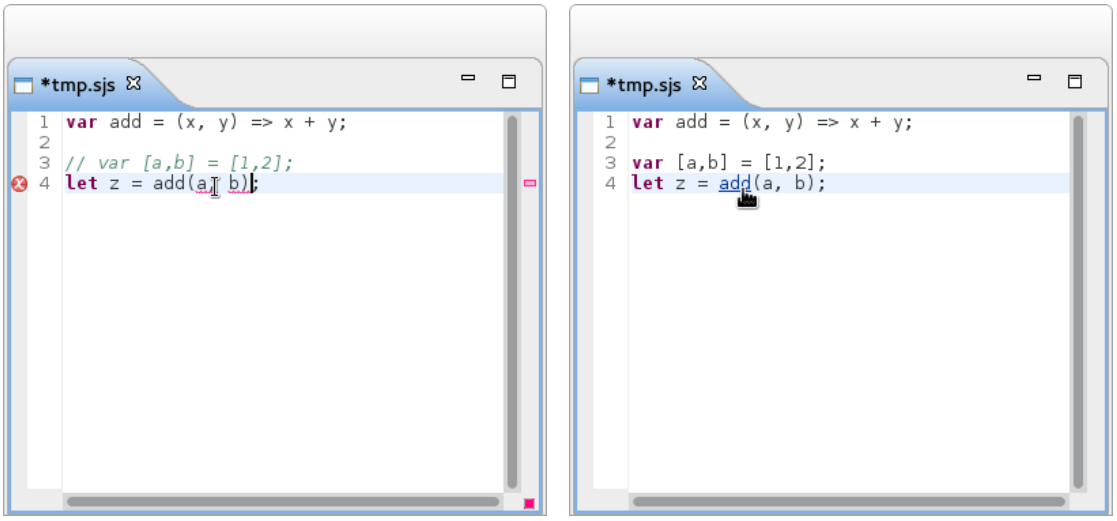
This function visits the scope tree (top-down, the root of the tree is the current scope). For block and root nodes the location of declaration of the binding is returned if it is present in the environment of that scope. If no declaration is found in block scopes and we visit a closure node we first identify whether any *illegal* references are possible to declarations in other block scopes not accesible from our scope. Then we lookup whether a declaration exists in the current closure and return this declaration if it exists.

To detect illegal redeclarations a function name `declare` is called on each declaration. It checks whether the name is already declared in the current block-scope, in this case it sets an error message. If the binding is not yet declared in the block-scope it checks whether it is set in any other block-scopes of the current closure, in this case the declaration is marked for renaming.

```
1 bool declare(loc decl, str name, loc def, Scope scope);
```

5.2.6 IDE integration

Because the features described in the previous two sections need name graph analysis we can perform some static analysis on these graphs and integrate the resulting data



(A) Reference of undeclared bindings (B) Declared at hyperlink

FIGURE 5.9: IDE support

with the IDE. First, the name graph gives us information of all references in the source file, with this information we create hyperlinks from references to declarations. Second, we can also identify a reference to an undefined binding and mark this reference with an error inside the IDE. Finally, bindings created by let (or const) are not allowed to be re-declared in the same scope, illegal redeclarations can be identified and marked with an error in the IDE.

Chapter 6

Results and Comparison

In this chapter we will evaluate our implementation of ES6 specification as a transformation suite implemented using a language workbench. We will validate against several open-source projects that also implement such a transformation suite but instead of using a language workbench implement the transformations using JavaScript.

6.1 Evaluation

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

Coverage
: How many language features are implemented

Correctness
: A language extension is performed correct if the target program is semantically similar to that of the language feature defined in the ES 6 specification [18]. To test the correctness we use a test-suite created especially for this purpose[1].

Size
: How many lines of code are used to implement the transformation suite, compared on a feature basis.

Modularity
: Is it possible separate and recombine the different language extensions.

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 the 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.

Performance
: What is the run time of the transformation code.

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

To evaluate our implementation of the transformation suite we validate against three open-source implementation of ES 6 language as a transformation suite (to ECMAScript 5). We selected the following *transpilers* (these transformation suites categories themselves as transpilers a combination of compiler and transformer) to evaluate against:

TABLE 6.1: Open-source ES6 transpilers

	Description	**Cont.**	**Com.**
Babel JS[1]	The most used ES 6 to 5 transpiler. 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 on a run-time environment.	55	1584
ES6-Transpiler[3]	Apart form Traceur and Babel JS this is the most feature complete transpiler. It relies on the ESPrima[5] JavaScript parser.	5	250

Coverage In appendix A we give an overview of ES6 features implemented by our transformation suite, with argumentation why several features are not implemented. Babel JS is them most feature complete with the exception of *new.target* property (not implemented by any of the transpilers). Generators are transformed by Babel JS but the needed transformation is not implemented by the Babel JS developers (they use the *regenerator* transformation suite to transform generators). Traceur lacks support for proper tail-calls and the *new.target* property they do have their own implementation of generator transformations. ES6 Transpiler implements *for of* loops in a variant too basic to categorize it as implemented (i.e. *for of* loops can only iterate over lists, not strings, or iterables). ES6 transpiler also lacks support for generators, the *new.target* property, and proper tail-calls.

Correctness In table 6.3 results of the compatibility tests. Four categories are of interest because they provide tests for the new language features of ES 6, and not tests for new standard-library functionality introduced by ES 6. Syntax depicts the category testing all language features that introduce new syntactical elements. Tests regarding the new binding mechanisms fall under the category Bindings. Tests for the new function types (i.e. generators, arrow functions, and classes) fall under the Function category.

Not all tests can be satisfied by a transformation suite alone, some tests use the *eval* function to evaluate a string as JavaScript in the current run-time. A transformation suite only has access to the static semantics during transformation it can not resolve the *string* that is going to be evaluated during run-time. Because other transformation suites face the same issue we do not remove these tests from the set of tests.

[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/

TABLE 6.2: ES6 features implemented

	Babel JS	Traceur	ES6 Transpiler	RMonia
Function extensions				
Arrow functions	●	●	●	●
Class declaration	●	●	●	●
Super	●	●	●	●
Generators	○	●	○	○
New.target	○	○	○	○
Syntax extensions				
Destructuring	●	●	●	◐
For of loops	●	●	○	●
Binary & Octal literals	●	●	●	●
Object literal	●	●	●	●
Rest Parameters	●	●	●	●
Default	●	●	●	●
Spread operator	●	●	●	●
Template Literals	●	●	●	●
Unicode Escape Points	●	●	●	○
RegExp "y" and "u" flag	●	●	●	○
Binding extensions				
Let	●	●	●	●
Const	●	●	●	●
Modules	●	●	○	○
Optimization				
Proper tail-calls	◐	○	○	○

●: Implemented, ◐: Partially implemented, ○: Not implemented

Our implementation does not score as high as the big open-source projects, mainly because we do not implement generators and automatically fail twenty-two tests related to generators. ES6-transpiler scores lower than our implementation. The compatibility score indicates that our artifact could be used in a *real-world* environment, as long as unimplemented features are avoided.

As discussed in section 3.3.1 hygiene is an important aspect of the correctness of program transformations. We will present one example where we try to shadow an introduced binding by the transpiler and detect whether or not the transpiler resolves the shadowed binding. For variable capture to happen we need to test a language extension that either introduces bindings or depends on source bindings. In this example we use the arrow function extension and try to introduce variable capture because this extension has to bind *this* from the lexical scope of the parent function (see appendix B.1).

```
1 function f() {
2   var <Id> = null;
3   var x = () => this;
4 }
```

LISTING 6.1: Example input to Traceur[2]

TABLE 6.3: Compatibility tests

		Transpiler			
	Total	**Babel**	**Traceur**	**ES6 Transpiler**	**RMonia**
Syntax	76	76	60	46	64
default function parameters	6	5	3		
rest parameters	5	4	4		
spread (...) operator	12	12	12		
object literal extension	6	6	6		
for..of loops	8	8	8		
octal and binary literals	4	4	2		
template strings	3	3	3		
RegExp "y" and "u" flags	2	1	1		
destructuring	32	29	26		
Unicode code point escapes	2	1	1		
new.target	2	0	0		
Bindings	19	15	15	10	16
const	8	6	6		
let	10	8	8		
block-level function declaration	1	1	1		
Functions	62	54	50	43	35
arrow functions	12	9	10		
class	23	19	16		
super	7	6	6		
generators	22	21	20		
Total	157	136 (87%)	125 (80%)	99 (63%)	115 (73%)

we replace `<Id>` with the variable name *this* is bound to by the transpiler:

Both ES6 Transpiler and Babel JS correctly resolve the shadowed variable (see appendix C for examples). Traceur fails to correctly identify variable capture in this example (see listing 6.2). Because `this` is still bound to `$__0` and then we redeclare `$__0` and bind `null` to it, the resulting target program will not behave as expected.

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

LISTING 6.2: Variable capture in Traceur transpiler

Size In table 6.4 we compare the size of all projects in lines of code, where our own artifact is named Rascal. Lines of code are measured using the *cloc* tool[3], because *cloc* has no built in support to measure lines of code of Rascal programs we forced *cloc* to measure Rascal programs as if they where *Java* code (Rascal comments are similar to those of Java).

[3] http://cloc.sourceforge.net/

To compare the total size of all transpilers based on lines of code used we should normalize according to the amount of features implemented by each transpiler. There are a total of 19 features that an ES6 to ES5 transformer can implement (see section 6.1), since our project is time-constrained we implemented 12 features completely and one feature partially, making our project about 70% feature complete. The lines of code used to calculate percentages from are presented in table 6.4.

Babel fully implements 16 features and partially implements 1 feature. Resulting in a feature completeness of 87% while using 378% more lines of code. Traceur fully implements 17 features and non partially. Making their implementations 90% feature complete and using 622% more lines of code. ES6 Transpiler fully implements 14 features and non partially. Their implementation is thus 74% feature complete while using 290% more lines of code.

Our artifact relies on the Rascal parser generator and a Rascal syntax definition to define the ES 5 grammar, and ES 6 language extensions. Other implementations rely on open-source ES parsers implemented in JavaScript, or implement their own parser. The syntax definitions are about eight to ten times smaller in size than any of the JavaScript parsers. Rascal syntax definitions do impose some limitations that made it difficult or impossible to implement certain ES grammar features, in appendix A we elaborate on this.

TABLE 6.4: Lines of code — transformations & parser

	Files	**SLOC**
Babel	76	6547
Traceur	19	9881
ES6 Transpiler	22	5341
RMonia	62	1368

	Files	**SLOC**
Acorn	22	3583
Traceur	15	6681
ESprima	1	4323
RMonia	12	555

In table **??** we analyze the size of each transpiler on a feature basis, where only the size of actual transformation code for an extension is measured. RMoniaand that of Babel JS have simlilar sizes for transformations. Traceur and ES6 Transpiler differ in size per feature compared to RMonia, the differences are inline with the size differences of projects as a whole.

Traceur and ES6 transpiler's size difference as compared to Rascal's implementation are similar to that of the size different of the projects as a whole.

The main reason why Babel JS is so small is because of the architecture of the project. Many concerns are extracted from transformation code and solved separately in different modules/files. Some examples: creating new bindings (*src/babel/traversal/scope/binding.js*), pattern matching on parse tree nodes (*src/babel/traversal* SLOC: 2456), creating new abstract syntax (*src/babel/transformations/templates* SLOC: 456).

Modularity Modularity can be measured in several different ways. First, how easy (if at all possible) is it to extend any of the language extensions present, and to add new extensions to the base language. Second, is it possible to run the transformer with a subset of the available extensions. And finally, how do extensions depend on each-other (i.e. can you combine extensions).

Table 6.5: Source lines of code per transformation

	Transpiler			
	Babel	**Traceur**	**ES6 Transpiler**	**RMonia**
Syntax				
default function parameters	92	66	277[1]	58
rest parameters	146	41	❙	✖
spread (...) operator	93	117	460	57
object literal extension	74	248	204	74
for..of loops	127	90	529	159
octal and binary literals	❙	31	22	✖
template strings	64	203	179	89
RegExp “y” and “u” flags	¿	45	300	✖
destructuring	349	403	303	261
Unicode code point escapes	✖	41	82	❙
Bindings	513	651	219	221
Functions				
arrow functions	58	144	277[1]	40
class	390	286	498	215
super	246	287	❙	❙
generators	¿	203	✖	✖

✖ – extension not implemented by transpiler.
❙ – extension not implemented in separate source file.
¿ – feature implemented using heavy dependency on external modules.

The extending of language extensions or creating new extensions on top of the base language is hard for all other implementations discussed in this section. The issue lies in the fact that these implementations rely on a separate parser for parsing source programs. Thus to create a new language extension the parser code needs to be extended. In RMoniaa language extension is capable of defining extended syntax on top of the base language, thus making it possible to create language extensions without knowledge of the underlying parser technology.

Babel JS supports selection of enabled extensions through the *–exclude* command line parameter through which language extensions can be disabled. Traceur allows enabling/disabling of extensions through a command line parameter per feature (i.e. *–arrow-functions=false* disables the arrow functions language extension). ES6 Transpiler does not support the enabling/disabling of extensions, the transformer always enables all language extensions.

In our implementation all new language features are fully modularized, making it possible to only enable certain language extensions. Even when extensions rely on each other it is possible to use them in separation. For instance the destructuring language extension can use computed property names (see appendix B.4 and listing 6.3) to destructure an objects property. The computed property names are part of the object literal language extension, if both extensions are enabled then it is possible to use computed property names inside object destructurings, if only destructurings are enabled computed property names are not enabled (for explanation of mutually dependent language extension see section 5.2.3).

Table 6.6: Size (in characters) of transformed JavaScript per transpiler (source files in appendix C)

		Transpiler			
	Input	**Babel JS**	**Traceur**	**ES6 transpiler**	**RMonia**
Arrow functions	409	580	688	498	651
Destructuring	188	455	1134	294	–
Object literals	116	193	607	251	213

```
1 function key() { return "k"; }
2
3 var { [key()] : value } = obj;
```

Listing 6.3: Example of computed property names inside object destructurings

Output "noise" To measure the amount of noise a transpiler produces we will compare the size of the input program (in characters) against the size of the output program. We use several example files given in Appendix C the results are presented in table 6.6. The table reveals that there are large differences in output size, measured in amount of characters, between transpilers when using the same input file. Traceur uses more characters then any of the other transpilers. This is the result of Traceur's relience on a large runtime system.

Apart from a quantative measure for the amount of noise introduced, we also inspect target programs generated by the transpilers and notice if we can detect large differences from the source program.

Traceur gives every binding introduced by a transformation non descriptive name (e.g. $__0) to avoid variable capture. This additional noise makes it harder for the programmer to debug the target program. Babel JS will bind to a descriptive name (e.g. _this to bind the this keyword for later use). In figure 6.1 we present an example program where an arrow function that uses its lexical enclosing `this` binding has to be desugared. Babel, ES6 Transpiler, and RMoniabind to descriptive variable names, where Traceur uses non descriptive identifiers.

Performance This dimension is not measured in our comparison of RMoniato other transpilers. There are major differences between the technology on which RMoniais implemented (i.e. Rascal) and all of the other transpilers. First, all other transpilers are implemented in JavaScript where RMoniais implemented in Rascal which has to be integrated with an IDE. Second, RMoniadoes not implement ES6 modules which are needed to transform a large ES6 project. To transform modules Babel can integrate with programs that package front-end dependencies (e.g. Browserify or Webpack), because Rascal is integrated in an IDE it is not as trivial to integrate with such packages as when you implement your solution in JavaScript.

FIGURE 6.1: Synthesized binding descriptive naming example

```
function getScope() {
  return () => this;
}
```

```
var _this = this;

return function () {
  return _this;
};
```

LISTING 6.4: Babel JS

```
var $__0 = this;
return (function() {
  return $__0;
});
```

LISTING 6.5: Traceur

```
function getScope() {
  var this$0 = this;
    return function()  {
      return this$0
    };
}
```

LISTING 6.6: ES6 Transpiler

```
return
(function(_this, _arguments) {
  return function() {
    return _this;
  };
})(this, arguments);
```

LISTING 6.7: RMonia

6.2 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"[28].

The formal specification presented by Felleisen falls outside of the scope of this thesis. But we can summarize 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."*[28]

This implies that every language extension transformed through a *local-to-local* scoped transformation (see section 4), creates a language that is no more expressive than the original (core) language. In table 4.1 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 let/const binding constructs (see appendix B.12). This extension requires a *global-to-global* transformation. And the arrow function (see appendix B.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).

6.3 Reflecting on the taxonomy

In chapter 4 we have presented a taxonomy for language extensions and in appendix B we have created a categorization of ES6 language extensions according to this taxonomy.

Together with the taxonomy we presented several questions underlying the structure of the taxonomy. With an implementation of a subset of ES6 language extensions and a categorization of these language extensions we can now reflect on the usefulness of this taxonomy to make predictions about implementation details of language extensions.

How can the language extension be implemented?

Category	All language extensions implemented in this thesis are categorized as desugarings. Drawing conclusions for other categories is thus not possible. We can conclude that program transformations are well suited for implementing language extensions that categorize as desugarings.
Dependencies	This dimension can give insight into the possibility of mutual dependent language extension (see section 5.2.3) in a set of language extensions.

What information does transformation of a language extension require?

Scope	The scope dimension gives insight into the element matched by the parse tree visitor (see section 5.2.1) for program transformation of a language extension. For instance a *global-to-local* or a *global-to-global* transformation will have to match on the root element of the parse tree to have the correct context information for their program transformation.
Analysis of subterms	Language extensions that have to analyze sub-terms are more complex to implement then transformations that do not have to perform such an analysis.

What are guarantees can we give of the target program produced by a language extension?

Introduction of bindings	Any transformation that introduces new bindings is exposed to the possibility of introducing variable capture. This implies that any transformation not introducing new bindings is also safe from variable capture.
Depending on bindings	As shown in section 3.3.1 there are two forms of variable capture one occuring because of introduced bindings by the transformation. The second because the target program depends on bindings from the source program or global scope. This dimension helps identifying whether a transformation can introduce the second form of variable capture.

6.4 Engineering trade-offs

With an implementation of ES6 inside a language workbench and a comparison to other implementations we identify the engineering trade-offs of implementing language extensions with the help of a language workbench.

6.4.1 Benefits

Increased productivity With the data we gathered it seems that implementation of the transformation suite inside the language workbench made the meta-programmer more productive. We needed fewer lines of code to express transformations that are evenly compatible with the specification as the other transpilers.

IDE integration Building language extensions inside the language workbench allows meta-programmers to immediately implement IDE features to help programmers while creating language extensions. Transpilers implemented outside of the language workbench will always need additional programming to integrate their solution with IDEs and deliver additional support to programmers using their transpiler. The language workbench made it possible to integrate, hyperlink referenced bindings, displaying errors for use of undeclared variables and illegal redeclarations, and previewing parts of target program.

Expressive transformations Because the language workbench allows the use of concrete syntax fragments inside transformation code the resulting transformations are more readable than code constructing abstract syntax trees. The JavaScript programmer can easily understand the concrete syntax fragments inside of the transformation code.

Modular language extensions Because language extensions in Rascal are nothing more than an extension of a module defining a base syntax, it is possible to extend and compose language extensions. By good design it is possible to implement all ES6 language extensions without direct dependencies on each-other, only on the base grammar. But it can be seen as a serious limitation of other implementations that it is not possible to extend any of the defined language extensions.

Grammar extension The language workbench allows us to define a base grammar and extend this grammar with language extensions. Other implementations of ES6 described above take a different approach, they rely on a full parser capable of parsing ES6 source programs. To introduce new features they have to extend this parser, separating the parsing implementation from the rest of their language extension's transformation code.

6.4.2 Limitations

Syntax definition The syntax definition of a language workbench can make the implementation of language extensions significantly easier/faster. It also imposes a limitation on our expressiveness, a meta-programmer can only work with the tools for solving ambiguity, defining syntax, supplied by the syntax definition of the language workbench. Some grammars do not lend themselves very well to be defined within these constraints (e.g. the automatic inference of semi-colons of a JavaScript grammar).

Portability By using a language workbench RMoniais bound to be used inside the IDE of the language workbench. Where on one hand this is an advantage for easier integration with the tools of a developer. It can be a limitation in the build step of a project using language extensions. Other transpilers can integrate with known build-pipelines for front-end technology. But RMoniais tied to a Rascal runtime, which is currently integrated with the Eclipse IDE.

6.4.3 Problems

Hygiene of program transformations Hygiene is identified as a problem of program transformations and a solution to solve this issue is reused. The case study reveals that other transformation suites do not always solve such issues (see listing 6.2).

Chapter 7

Conclusion

In this thesis we have made two contributions. First, a taxonomy for language extensions. Second, we have tried to show the *effectiveness* of a language workbench for the task of implementing language extensions. We implemented ES6 language extensions in RMoniaas an experiment to guide evaluation of the language workbench. And evaluated the implementation against other transpilers capable of transforming ES6 to ES5.

The taxonomy can be used to create a categorization of language extensions and estimate the work needed to implement transformation code for a specific extension. The categorization created for the ES6 language extensions proved valuable in predicting problems with the implementation of language extensions (e.g. can variable capture arise during transformation of the language extension).

The language workbench has been a valuable tool for the implementation of language extensions. However it is inconclusive if it really improves on all aspects as compared to traditional implementations. The issue of portability does impose a large restriction on our implementation, advantages of the language workbench are integration with developer tools, expressive transformation code using concrete syntax patterns, and modular language extensions. If a fixed set of language extensions has to be implemented to be distributed to a large group of developers, it is arguably better to take a traditional approach, because parser code will not change much overtime and portability issues of the language workbench can be avoided. If however the set of language extensions is not fixed and the language extensions are used by a small team the language workbench can present a good option for implementation of the language extensions.

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, which are difficult to remove with the disambiguation tools present in Rascal's syntax definition.

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 constraints set for this thesis the 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 code.

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)

Unicode Escape Points Not implemented, due to time constraints.

RegExp "y" and "u" flag Not implemented, due to time constraints.

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 4).

The compatibility tests as discussed in section 6.1 are stored in *input/compatibility* and can be invoked from the Rascal module at *src/test/Compatibility.rsc*

Our implementation of the *name-fix*[1] algorithm resides in *src/core/resolver*.

Getting the artifact The latest version of the artifact is available at `https://github.com/matthisk/ESarmonia` 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 chapter 4

B.1 Arrow Functions

Arrow functions[18, 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 3.1) but instead uses the binding of its parent lexical scope.

Category The arrow function is eliminated through transformation to a normal JavaScript function expression wrapped in a IIFE to bind *this* and *arguments* from the enclosing lexical scope. Arrow functions thus categorize as syntactic sugar.

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 the placement of this arrow function, thus the transformations scope is global-to-local.

Analysis of sub-terms References to the *arguments* 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." ([18, 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

[1] http://coffeescript.org/

with the **this**, and **arguments** bindings of the lexical enclosing scope bound to **_this** 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 appendix B.11) 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 an 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.2 Classes

Class definitions [18, 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 3.1).

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.3 Destructuring

Destructuring [18, 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.4):

```
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.4 Extended object literals

Literal object notation receives three new features in the ECMAScript 6 standard [18, 12.2.5]. Shorthand property notation, shorthand method notation, and computed property names.

Category The extended object literals are syntactic sugar. 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.5 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 [18, 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.12).

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

Destructuring assignment can be used as a for-of loop binding (see Appendix B.3):

```
1 for( [a,b] of arr ) <Statement>
```

B.6 Spread operator

ECMAScript 6 introduces a new unary operator named spread[18, 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 3.1). 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:

[3]For more information see Mozilla Developer Network documentation

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

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

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 `<Expression e>(<{ArgExpression ","}* args >)` is matched `e` has to be analyzed to detect whether it is a function, or object property function. (See Appendix B.6)

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

B.7 Default parameter

The ECMAScript 6 spec defines a way to give parameters default values [18, 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.8 Rest parameter

The *BindingRestElement* defines a special parameter which binds to the remainder of the arguments supplied by the caller of the function.[18, 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.9 Template Literal

Standard string literals in JavaScript have some limitations. The ECMAScript 6 specification introduces a template literal to overcome some of these limitations [18, 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.10 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 [18, 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.

Program modification This extension is purely semantic, no new syntax is introduced, thus a program modification and not extension. 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.

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

Listing B.3: Function with tail recursion

```
function fib(arg0,arg1,arg2) {
  while (true) {
    var n = arg0, nn = arg1, res = arg2;

    if(nn > 100) return res;
    arg0 = nn;
    arg1 = n + nn;
    arg2 = res + [n,nn];
  }
}
```

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.11 Generators

The ECMAScript 6 standard introduces a new function type, generators [18, 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 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.12 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`).

A second way to emulate block binding in ECMAscript 6 is with the help of a *try catch* statement, the introduced binding can be thrown in the try block and caught by the catch block. Making the binding available only to the remaining statements enclosed in the catch block. The issue with this solution is performance overhead imposed by *try catch* blocks, because these are not optimized by the interpreter of the JavaScript code.

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

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

```
{
  let x = 0;
}
{
  let x = 1;
}
```

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 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:

```
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.

Appendix C

ES6 Test Examples

,

```
var evens = [2,4,6,8,10],
    fives = [];

var odds = evens.map(v => v + 1);
var nums = evens.map((v, i) => v + i);

console.log(odds);
console.log(nums);

nums.forEach(v => {
  if (v % 5 === 0) fives.push(v);
});

function thisBinding() {
  return () => console.log(this.string);
}

thisBinding.call({ string: 'bound' })();

function args() {
  return () => console.log(arguments);
}

args('arg1','arg2')();
```

Listing C.1: Arrow function

,

```
var [a,b,c] = [1,2,3];

function fun( [a,b] ) {
  return a + b;
}

var qux = "corge";
var { [qux] : corge } = { corge: "graphly" };

var { d = '100' } = {b:0};
var [e,f,g = '100'] = [0,1];
```

Listing C.2: Destructuring

,

```
var handler;
var obj = {
    handler,
    toString() {
     return "d";
    },
    [ 'prop_' + (() => 42)() ]: 42
};
```

Listing C.3: Object literals

```
function f() {
  var _this2 = this;
  var _this = null;

  var a = function a() {
    return _this2;
  };
}
```

Listing C.4: Babel JS

```
function f() {
  var this$1 = this;
    var this$0 = null;
    var a = function()  {return this$1};
}
```

Listing C.5: ES6 Transpiler

Bibliography

[1] 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.

[2] 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.

[3] Matthias Zenger and Martin Odersky. Implementing Extensible Compilers. *In ECOOP Workshop on Multiparadigm Programming with Object-Oriented Languages*, pages 61–80, 2001.

[4] N. Nystrom, M.R. Clarkson, and a.C. Myers. Polyglot: An extensible compiler framework for Java. *Proceedings of the 12th International Conference on Compiler Construction*, (April):138–152, 2003. ISSN 0302-9743. doi: 10.1007/3-540-36579-6_11.

[5] 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.

[6] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. *Proceedings of the 1986 ACM conference on LISP and functional programming - LFP '86*, pages 151–161, 1986. doi: 10.1145/319838.319859. URL http://dl.acm.org/citation.cfm?id=319838.319859.

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

[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] Jan Heering, P R H Hendriks, Paul Klint, and Jan Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/71605.71607.

[10] Eelco Visser. Stratego: A Language for Program Transformation based on Rewriting Strategies. *Computer*, 2051(May):357–361, 2001.

[11] Eelco Visser. Program transformation with Stratego/XT. rules, strategies, tools, and systems in Stratego. *Domain-Specific Program Generation*, (February):315–349, 20024. doi: 10.1007/b98156. URL http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Program+Transformation+with+Stratego+/+XT+:+Rules+,+Strategies+,+Tools+,+and+Systems#6.

[12] Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D P Konat, Pedro J Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, and Riccardo Solmi. The State of the Art in Language Workbenches Conclusions from the Language Workbench Challenge. pages 197–217.

[13] Sebastian Erdweg and Tillmann Rendel. SugarJ: Library-based syntactic language extensibility. *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, (iii):391–406, 2011. ISSN 0362-1340. doi: 10.1145/2048066.2048099. URL http://dl.acm.org/citation.cfm?id=2048099.

[14] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Safely Composable Type-Specific Languages. pages 105–130, 2014.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.