CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Compiler from CakeML to JavaScript

Master’s thesis in Computer Science - algorithms, language and logic

OSKAR NYBERG

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

A Compiler from CakeML to JavaScript

OSKAR NYBERG

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

A Compiler from CakeML to JavaScript
OSKAR NYBERG

Supervisor: Johannes Åman Pohjola, Department of Computer Science and Engineering & Magnus Myreen, Department of Computer Science and Engineering
Examiner: John Hughes, Department of Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2018

A Compiler from CakeML to JavaScript
OSKAR NYBERG
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis presents a new compiler from CakeML to JavaScript with support for almost the entire CakeML language. In addition to the new compiler, a JavaScript syntax formalization has been defined together with formal semantics for a subset of JavaScript. The semantics include coverage for language features introduced as part of the ECMAScript 2015 standard. The new compiler, syntax formalization and semantics are implemented in the HOL4 theorem prover to allow for future verification of the new compiler.

The new compiler enables CakeML programs to be run in web browsers on both desktop computers and smart phones and other contexts previously not available to CakeML.

Keywords: Compiler, Transpiler, Formal, Semantics, Verification, CakeML, JavaScript, ECMAScript

Acknowledgements

During this thesis I've received support in various ways from several people. I would like to thank my two supervisors, Johannes Åman Pohjola and Magnus Myreen, for their support and feedback throughout the process. Secondly, I would like to thank my examiner, John Hughes. Lastly I would like to thank family and friends for their support.

Oskar Nyberg, Gothenburg, May 2018

Contents

List of Figures

List of Listings

1 Introduction

CakeML is a functional programming language, which is based on a subset of Standard ML, with a formally verified compiler and runtime system [9][12]. The compiler has been proven to produce machine code which is semantically equivalent to the CakeML code, except when the implementation runs out of memory.

According to the Stack Overflow developer survey 2018, JavaScript is currently the most used programming language [2]. JavaScript was initially intended for web development but has seen a great increase in use cases and can now be used in almost any situation, e.g. development of web front end, web back end, desktop applications and mobile applications.

CakeML and JavaScript are two vastly different languages. A big difference is, as between almost all languages, the syntax. More importantly there exists formal semantics for CakeML in contrast to JavaScript which lacks official formal semantics. One important similarity is that both are functional programming languages and therefore support either higher-order functions or first-class functions.

The use cases of JavaScript and the contexts it is used in could be made available to CakeML by compiling CakeML source code to JavaScript source code. The new compiler would preferably also be formally verified. The benefit of a formally verified source to source compiler is that it ensures that the produced code always behaves identically, i.e. is observationally equivalent, to the source code.

To formally verify a compiler, formal semantics for both the source and target languages are needed. The ECMAScript specification, which JavaScript is an implementation of, includes informal semantics for the language. There is a lack of official formal semantics for JavaScript but there exists unofficial formal semantics with varying completeness and varying support for language features in newer ECMAScript versions [8][11].

1.1 Purpose & Objectives

The purpose of this thesis is to explore how to deal with the differences between CakeML and JavaScript when compiling from one to the other, to extend the formal semantics in current literature with features from recent versions of the ECMAScript specification, and to enable CakeML programs to run in new contexts and on new platforms.

This purpose is fulfilled by achieving three objectives:

1. Define abstract syntax for a subset of JavaScript, covering the functionality required to represent all CakeML language features
2. Define big step operational semantics for the subset of JavaScript covered by the abstract syntax
3. Develop a new compiler, from CakeML to JavaScript, supporting almost the entire CakeML language

The abstract syntax, formal semantics and compiler are defined and implemented in the HOL4 theorem prover[1] to simplify future work of verifying the new compiler.

1.1.1 Delimitations

The new compiler covers a subset of the expression level of CakeML and a subset of the declaration level.

The new compiler does not perform any kind of optimization. The effect of using a small subset of JavaScript can result in quite unoptimized JavaScript code.

1.2 Similar compilers

There exist a few compilers with similarities to the new compiler described in Section 1.1, but all differ in either target or source language, or lack a solid connection to formal semantics.

1.2.1 CakeML to machine code

CakeML currently has a formally verified compiler which targets machine code for five different architectures (ARMv6, ARMv8, x86-64, MIPS-64, RISC-V) [12]. Un-

[1]`https://hol-theorem-prover.org/`

like the new compiler described in Section 1.1, the to-machine-code compiler is not a source to source compiler. Since the two compilers share the source language, some parts of the to-machine-code compiler can be reused in the development of the new compiler, namely the parser, lexer, type inferencer and the formal semantics for CakeML.

1.2.2 Standard ML to JavaScript

Elsman [7] has developed a compiler which uses Standard ML as source and JavaScript as target, but in contrast to the CakeML to-machine-code compiler and the compiler described in Section 1.1, it is not proven-correct and does not have any connection to formal semantics. The compiler features interesting solutions to problems like pattern matching, which are described in Section 5.1.2.3.

The compiler uses the MLKit[2] front end which compiles the Standard ML code to an intermediate language before it is compiled into JavaScript. This lowers the abstraction level and performs optimization.

The targeted ECMAScript version is not mentioned but it is stated that ECMAScript 5 (2009) does not include tail call optimization, which indicates that the 5th edition of ECMAScript probably is the targeted version.

An interesting use of the compiler is the web page *SMLtoJsOnline*[3] which is a web IDE. The IDE makes it possible to write, compile and run Standard ML code directly in the browser using a compiled (to JavaScript) version of the compiler.

1.2.3 Haskell to JavaScript

Another relevant compiler is the Haste compiler which is a compiler from Haskell to JavaScript [6]. It differs a bit more than the other mentioned compilers since it compiles from Haskell instead of an ML variant and is not proven correct. A key difference between the languages is that Haskell features lazy evaluation while JavaScript and CakeML feature eager evaluation. To emulate lazy evaluation in JavaScript, the Haste compiler wraps values and computations in thunks. Aside from the differences, a similarity is that it is a compiler from a functional programming language to JavaScript and therefore contains solutions to problems such as how to compile pattern matching.

The author does unfortunately not mention which ECMAScript version the Haste compiler targets but it seems like the targeted version might be ECMAScript 5.1 (2011).

[2] `http://www.elsman.com/mlkit/`
[3] `https://smlserver.org/ide/`

2

Background

This chapter presents the key concepts used in this thesis.

2.1 JavaScript

JavaScript is both an imperative and functional programming language. It supports first-class functions which, for instance, makes it possible to send functions as arguments, however, it does not allow partial application. JavaScript is mostly used in web browsers to create an interactive and dynamic experience and has in the last years started to be used in desktop applications and in mobile applications as well.

JavaScript is implemented according to the ECMAScript standard. Other implementations of the standard exists, such as JScript and ActionScript. The first edition of ECMAScript was introduced in 1997 and has since been succeeded with 7 editions. Up until 2015, new editions were not released on a fixed interval and were named after the version number, e.g. ECMAScript 5.1 (which was released in 2011). Since 2015 new editions have been released annually and named after the year of release, e.g. ECMAScript 2015. In this thesis the ECMAScript editions are referred to according to that naming scheme.

2.2 Compiler

A compiler transforms source code in a programming language to either machine code or source code in another programming language. Compilers perform this transformation in multiple stages where each stage translates the source code, closer to the target code.

The first transformation is performed by a lexer which purpose is to perform lexical analysis. It separates the input sentence into a list of tokens and checks if they are valid tokens of the programming language.

The list of tokens is then passed into a parser which adds structure to the tokens

```
lexer "val five = 5;" =
[(ValT); (AlphaT "five"); (EqualsT);
  (IntT 5); (SemicolonT)]
```

hol

Listing 1: Example of lexical analysis.

to create sentences. Moreover, it checks if the sentences are valid. Together the structured tokens forms an abstract syntax tree (AST), as shown in Listing 2 and Figure 2.1.

```
parser (lexer "val five = 5;") =
[Tdec (Dlet (Pvar "five") (Lit (IntLit 5)))]
```

hol

Listing 2: Example of parsing using simplified[1]CakeML abstract syntax.

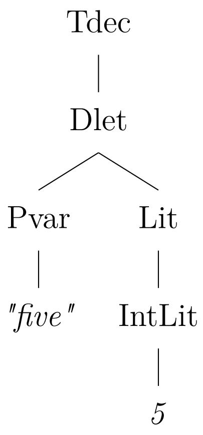

Figure 2.1: Abstract syntax tree in Listing 2 presented as a graph.

A type inferencer then uses the AST to validate if each value is of a legal type. In the CakeML to-machine-code compiler this process does not alter the AST.

Lastly the back end of the compiler transforms the AST into either machine code or another programming language. This is done in multiple phases where each phase either removes abstractions or performs optimization.

A compiler targeting another programming language is called a source to source compiler or transpiler. This type of compilation is useful when there is a need to run a program in the context of another language. An example of this is when JavaScript code using features from ECMAScript 6 is compiled to JavaScript ECMAScript 5 to increase support in old web browsers.

[1]Debugging information is removed to make example more readable.

2.3 Formal semantics

Semantics describe the meaning and logic behind programming languages, in contrast to syntax which describes legal code. Informal semantics describe the logic in words in contrast to formal semantics which are defined using formal logic. This formal logic often evaluates the code in regard of the current state and results in a new updated state and a return value, as shown in Listing 3, 4 and Figure 2.2.

```javascript
if (A) {
  B
} else {
  C
}
```

javascript

Listing 3: An if-statement in JavaScript.

$$\frac{\langle A, s\rangle \Downarrow \langle \textbf{true}, s'\rangle \quad \langle B, s'\rangle \Downarrow s''}{\langle \textbf{if } A \textbf{ then } B \textbf{ else } C, s\rangle \Downarrow s''} \qquad \frac{\langle A, s\rangle \Downarrow \langle \textbf{false}, s'\rangle \quad \langle C, s'\rangle \Downarrow s''}{\langle \textbf{if } A \textbf{ then } B \textbf{ else } C, s\rangle \Downarrow s''}$$

Figure 2.2: Formal semantics corresponding to the code Listing 3, where s represents the current state, s' and s'' states after evaluation.

```hol
evaluate_stm s [If condition stm1 stm2] =
  case evaluate_exp s condition of
    | (s', T) => evaluate_stm s' stm1
    | (s', F) => evaluate_stm s' stm2
```

hol

Listing 4: HOL definition of semantics for JavaScript if-statement

There have been multiple attempts at specifying formal semantics for JavaScript. They differ in completeness and correctness and apply to different ECMAScript versions.

2.3.1 JavaScript formalization by Park et al.

Park et al. [11] have implemented formal JavaScript semantics using the K Semantic Framework[2]. The semantics target ECMAScript 5.1 and covers the entire ECMAScript specification. The implementation is executable and has been tested against the ECMAScript conformance test suit, and passes all 2,782 core language tests. The tests it does not pass are related to standard library functions that

[2]https://github.com/kframework/javascript-semantics/tree/d5aca308d12d3838c645e1f787e2abc9257ce43e

could be implemented using features covered in the core language tests, which could therefore be considered unnecessary to spend time and resources on.

The testing against the conformance suit revealed bugs both in production JavaScript engines and in the ECMAScript specification. The fact that their implementation revealed bugs in the specification justifies their sub-objective to encourage language designers to include complete formal semantics in language specifications.

Contrary to multiple other JavaScript formalizations, it supports non-deterministic functionality. An example of non-deterministic functionality in JavaScript is the `for...in`-statement which iterates the elements of an iterable object in a non-deterministic order.

2.3.2 JavaScript formalization by Gardner et al.

Gardner et al. [8] have defined big-step operational semantics for a subset of JavaScript that corresponds to the ECMAScript 3 (1999) standard. The authors explain key concepts and different mechanisms, such as the variable store and function calls, very clearly. It provides a good overview of the semantics and makes the semantics easily understandable. Especially interesting is the abstractions of the semantics regarding the variable store which is used and described in Section 4.2.2.

The semantics are specified in the paper and do not feature an implementation. This enlarges the risk of the semantics being incorrect since they have not been tested.

2.4 Formal verification

Formal semantics can be used to formally verify a compiler. Formal verification is the process of creating logical proofs that show the correctness or incorrectness of algorithms. It is used to ensure that algorithms do what they are intended to do and to show that all corner cases are covered.

2.5 HOL Interactive theorem prover

There are multiple tools available which makes it possible to define formal semantics and to perform formal verification. One such tool is the HOL4 theorem prover [1] which provides an interactive environment for rigorous proofs in higher-order logic.

3

JavaScript Abstract Syntax

Abstract syntax definitions for JavaScript are needed to be able to compile the CakeML abstract syntax tree to a JavaScript abstract syntax tree.

The requirement for the JavaScript abstract syntax is to cover the parts of JavaScript needed to implement the CakeML functionality supported by the new compiler, and to preferably include language features of JavaScript which are as similar as possible to the language features of CakeML.

Each CakeML language feature can be represented in multiple ways in JavaScript and depending on the targeted ECMAScript version, the number of alternatives varies.

3.1 ECMAScript version

The disadvantage with targeting one of the latest ECMAScript versions is that it might not be supported in major JavaScript engines and that there might not exist formal semantics for the language features introduced. The current ECMAScript version, as of writing, is ECMAScript 2018 and as mentioned in Section 2.3 the formal semantics in the literature are defined for the 5th edition of ECMAScript or earlier versions.

An example of features that were introduced in more recent versions of ECMAScript is `let` and `const` keywords for variable declaration which complements the `var` keyword. The differences between the different types of variable declaration are the scope and writability as seen in Figure 3.1 [3][4]. These keywords will be used by the compiler to store variables, as can be seen in Chapter 5.

```javascript
var a = 1
let b = 2
const c = 3
```

javascript

Listing 5: Different variable declaration statements in JavaScript.

Keyword	ECMAScript version	Writable	Scope
var	1 (1997)	Yes	Function scope
let	6 (2015)	Yes	Block scope
const	6 (2015)	No	Block scope

Figure 3.1: Difference between variable declaration keywords.

Another feature that will be used in the new compiler is destructuring of arrays and objects, which was introduced in 2015. The reason why it is interesting when building a compiler from CakeML is that destructuring is an important part of pattern matching.

Even though there is no JavaScript feature able to replicate the behaviour of the pattern matching functionality from CakeML shown in Listing 6, destructuring simplifies a pattern matching implementation.

```javascript
// JavaScript array destructuring
let [a, b] = [1, 2]
```

javascript

```
(* CakeML list destructuring *)
val [a, b] = [1, 2]

(* CakeML pattern matching *)
case [1, 2] of
    [1, b] => b
  | [a, _] => a
```

cakeml

Listing 6: Example of JavaScript and CakeML array destructuring and CakeML pattern matching.

ECMAScript is backwards compatible which makes it possible to use the abstract syntax and semantics defined for ECMAScript 5.1 and if necessary define abstract syntax and semantics for language features in more recent ECMAScript versions such as ECMAScript 2015 [4]. This means that the abstract syntax and semantics defined is a subset of the ECMAScript 2015 specification, and since it is backwards compatible it is also a subset of ECMAScript 2018 etc.

3.2 Definitions

The abstract syntax definitions, in the HOL4 theorem prover, represent a subset of the grammar in the ECMAScript 2015 specification.

3.2.1 Literals

The smallest component of the abstract syntax is literals which in JavaScript consist of *number*, *string*, *boolean* and *null* literals. String and boolean literals behave the same way in both CakeML and JavaScript while integers behave rather differently.

In contrast to CakeML there is no integer type in JavaScript. Instead there is a type called *number* which is actually a floating-point representation of real numbers which is displayed as an integer if there are no decimals [5].

The issue with floating point numbers is that for very large numbers the precision is not enough. The ceiling for correct representation of integers in JavaScript can be found in the constant `Number.MAX_SAFE_INTEGER` which contains the value 9007199254740991 or $2^{53} - 1$ [5]. What makes it challenging for the compiler described in Section 1.1 is that it will not be possible to verify the compiler with CakeML integers represented by JavaScript numbers.

Since there is no integer type, a separate implementation of arbitrary length integers is needed and a few options have been considered. The simplest option is a recursive definition, similar to the simplified implementation only dealing with natural numbers shown in Listing 7 and 8. This implementation could be extended to cover negative integers by wrapping it in a data type keeping track of the sign.

```
datatype integer = Zero | Successor integer
(* Example: 3 => Successor (Successor (Successor Zero)) *)
```

cakeml

Listing 7: A recursive definition of arbitrary length natural numbers in HOL4.

The issue with this approach is that it is very unoptimized and performs poorly (it is both slow and requires great amounts of memory).

Another option is to represent Integers as a list of bits by using an array of booleans where the first "bit" represents the sign, as shown in Listing 9.

The issue with this approach is that the max length of an array, which according to the ECMAScript 5 & 2015 specifications is the maximum value of an unsigned 32 bit integer ($2^{32} - 1$). This is not a valid solution either [3][4].

There are of course more complex methods of representing arbitrary length integers, such as using strings, but that would require greater amounts of code and semantics

```javascript
class Integer {
  constructor(value) {
    this.value = value
  }

  static get zero {
    return new Integer(null)
  }

  get successor {
    return new Integer(this)
  }
}

// Example:
// 3 => Integer.zero.successor.successor.successor
```

javascript

Listing 8: A recursive definition of arbitrary length natural numbers in JavaScript.

```javascript
[false, true, true] => 3
```

javascript

Listing 9: Binary representation of integers implemented in JavaScript.

to be defined. There exists third party libraries that implement arbitrary length integers using such methods. There was also a proposal for built in arbitrary length integer support in ECMAScript 2018 which unfortunately did not make it to the final version.

The abstract syntax implements integers as arbitrary length integers in hopes of it being included in a future version of ECMAScript. Until then, integers are represented using a third party library called BigInteger.js[1] which provides the same functionality and probably the same or very similar semantics.

The introduction of an external library affects the verification of the compiler, since the library is not proven correct and does not have formal semantics. The solution chosen is to trust the non-formal semantics in the *readme*-file associated with the library, and use it as a base for the formal semantics.

The resulting literal definition consists of big integer, string, bool and null literals as shown in Listing 10.

[1]`https://github.com/peterolson/BigInteger.js`

```
js_lit =
  | JSBigInt int
  | JSString string
  | JSBool bool
  | JSNull
```

hol

Listing 10: Literals covered by the abstract syntax.

3.2.2 Binding patterns

During assignment in JavaScript, the right side of the assignment operator is assigned according to a binding pattern on the left side of the operator. The simplest and most commonly used pattern is a variable name, resulting in the value being assigned to a variable with the specified name. More complex patterns include object and array representations which make it possible to destructure a value into the variable names specified in the pattern. It is also possible to assign the rest of an array to a variable using the rest pattern, which mimics the behaviour of the CakeML list constructor `(::)` as shown in Listing 12. Binding patterns are used together with other language features to achieve object and array destructuring to replicate the behaviour of CakeML pattern matching, which is described in Section 5.1.2.3.

```
js_bind_element =
  | JSBVar js_varN
  | JSBObject ((js_varN, js_bind_element option) alist)
  | JSBArray (js_bind_element list)
  | JSBRest js_bind_element
```

hol

Listing 11: Binding patterns covered by abstract syntax.

```
let [a, ...b] = [1, 2, 3]
// b = [2, 3]
```

javascript

Listing 12: JavaScript rest operator as binding pattern.

3.2.3 Expressions

There are many components of the JavaScript expression level that are similar to the CakeML expression level. Some examples of the similarities are CakeML `if...else` expressions and JavaScript conditional expression, and function application in both languages. The expression definition is found in Listing 13.

```
js_exp =
  | Exp_not_compiled exp
  | JSLit js_lit
  | JSArray (js_exp list)
  | JSIndex js_exp js_exp
  | JSObject ((js_varN, js_exp option) alist)
  | JSObjectProp js_exp js_varN
  | JSUop js_unary_op js_exp
  | JSBop js_binary_op js_exp js_exp
  | JSVar js_varN
  | JSAssign js_exp js_exp
  | JSAFun (js_bind_element list) (js_stm list)
  | JSFun js_varN (js_bind_element list) (js_stm list)
  | JSApp js_exp (js_exp list)
  | JSConditional js_exp js_exp js_exp
  | JSClass (js_varN option) (js_varN option)
      ((js_varN # (js_varN list) # js_stm) list);
```

hol

Listing 13: Expressions covered by abstract syntax where `Exp_not_compiled` is used to represent CakeML language features not supported by the new compiler, which is described in Section 5.2.

3.2.4 Operators

As can be seen in Listing 13, the abstract syntax includes both unary (`JSUop`) and binary (`JSBop`) operators. The defined unary operators are negation, new and spread and the binary operators consist of a subset of the mathematical and logical binary operators, as shown in Listing 14.

```
js_unary_op =
  | JSNeg
  | JSNew
  | JSSpread

js_binary_op =
  | JSIntPlus | JSIntMinus | JSIntTimes | JSIntDivide
  | JSIntModulo | JSIntLt | JSIntGt | JSIntLeq | JSIntGeq
  | JSPlus | JSMinus | JSTimes | JSDivide | JSModulo
  | JSLt | JSGt | JSLeq | JSGeq | JSEq | JSNeq
  | JSAnd | JSOr
  | JSComma
```

hol

Listing 14: Operators covered by the abstract syntax.

The spread operator refers to the unary operator (`...`) which expands a list into individual arguments separated by a comma which can be used to represent the CakeML list constructor which is described more in depth in Section 4.2.1.

Most of the binary operators act similarly to the CakeML equivalents with the exception that they can be applied to multiple types and have different semantics depending on the types of the operands. For example, when the plus-operator is applied strings, it concatenates the two strings. In the definition, the operators prefixed with `JSInt` represents operators from the integer library while the remaining operators are the JavaScript operators which can be applied to strings, booleans, numbers etc.

```javascript
let a = [2, 3]
let b = [1, ...a]
// b = [1, 2, 3]
```

javascript

Listing 15: JavaScript spread operator.

3.2.5 Statements

On the declaration level of CakeML there are only a few different components whereas in JavaScript the statement level contains many more components. For example JavaScript has multiple statements for declaring variables, `var`, `let` and `const`, of which the differences are described in Section 3.1. Some other statements are `if...else` statements and exception handling. The statement definition is shown in Listing 16. As can be seen in the definition, there are two statements used to represent features not supported by the compiler, i.e. `Dec_not_compiled` and `Top_not_compiled`. Dec refers to the declaration level of CakeML and top refers to the top level which either contains a declaration or a module definition. These statements contain the parts of the CakeML AST that were not compiled.

```
js_stm =
  | Dec_not_compiled dec
  | Top_not_compiled top
  | JSBlock (js_stm list)
  | JSLet js_bind_element js_exp
  | JSConst js_bind_element js_exp
  | JSVarDecl js_bind_element js_exp
  | JSExp js_exp
  | JSIf js_exp js_stm js_stm
  | JSReturn js_exp
  | JSThrow js_exp
  | JSTryCatch (js_stm list) js_bind_element (js_stm list)
  | JSEmpty;
```

hol

Listing 16: Statements covered by abstract syntax where `Dec_not_compiled` and `Top_not_compiled` represents CakeML language features which are not supported by the new compiler, which is described in Section 5.2.

4 JavaScript Semantics

To formally verify the new compiler, formal semantics for both the source and target language is needed. Since CakeML and JavaScript have different language features, semantics for the whole JavaScript language is not needed. Instead, a subset of the language is used, more specifically the subset defined by the abstract syntax introduced in Chapter 3.

The semantics of the JavaScript subset is defined as a functional big-step operational semantics following the style of semantics used in the to-machine-code compiler proofs [10]. The existing JavaScript formalisations in the literature are not defined in the HOL4 theorem prover and could not be used directly without being translated first. Therefore, previous semantics only served as inspiration when defining the semantics in HOL4. In addition to the translation of formal semantics from literature, formal semantics for some language features in ECMAScript 2015 has been defined.

4.1 Binary operators

An example from the semantics definitions is the semantics for binary operators which is shown in Listing 17. The `js_evaluate_exp` function is the main function in the evaluation of expressions and transforms a state, environment and an expression into a new state, new environment and a result which can be either a value or an error. In the `js_evaluate_exp` function there is a case handling the `JSAnd` operator, which evaluates the first operand and checks if the result is truthy with the help of the function `is_truthy`. If this is the case, then the other operand is evaluated and its result is returned as the result of the evaluation of the operator expression. The evaluation of the `JSOr` operator works similarly to `JSAnd`.

In the case of other binary operators both operands are evaluated before they are sent to a helper function called `js_evaluate_bop` which transforms two values into one value in accordance with the supplied operator.

```
is_truthy v =
  ~ (v = JSUndefined
  \/ v = JSLitv JSNull
  \/ v = JSLitv (JSBool F)
  \/ v = JSLitv (JSBigInt 0)
  \/ v = JSLitv (JSString ""))

js_evaluate_bop JSPlusInt
  (JSLitv (JSBigInt a)) (JSLitv (JSBigInt b)) =
    JSRval [JSLitv (JSBigInt (a + b))]

(js_evaluate_exp st env [JSBop JSAnd exp1 exp2] =
  case js_evaluate_exp st env [exp1] of
    | (st', env', JSRval [v1]) => if is_truthy v1
        then js_evaluate_exp st' env' [exp2]
        else (st', env', JSRval [v1])
    | res => res) /\

(js_evaluate_exp st env [JSBop op exp1 exp2] =
  case js_evaluate_exp st env [exp1] of
    | (st', env', JSRval [v1]) =>
        (case js_evaluate_exp st' env' [exp2] of
          | (st'', env'', JSRval [v2]) =>
              (st'', env'', js_evaluate_bop op v1 v2)
          | res => res)
    | res => res)
```
hol

Listing 17: Semantics of binary operators. The `js_evaluate_bop` function is reduced to only one operator in this definition.

4.2 ECMAScript 2015 features

The formal semantics for ECMAScript 2015 features were defined according to the ECMAScript specification [5].

4.2.1 Spread operator

The spread operator (`...`) is a unary operator that transforms its operand into a comma separated list. The operator can only be used on an iterable object in an array initializer or when calling a function, as shown in Listing 18.

The formal semantics for the spread operator is defined as part of the semantics for arrays. When an array is evaluated, each element is evaluated and the results are put into a list. The exception for the spread operator is that the array value returned

```javascript
let a = [1, 2, 3]
let b = function() {
  return [...arguments]
}
let c = b(...a)

// c = [1, 2, 3]
```

javascript

Listing 18: JavaScript spread operator.

by the evaluation is not added as an element, instead the items of the returned value are concatenated to the list of elements as shown in Listing 19, which only contains the parts of the array semantics that is affected by the spread operator. The full semantics for the array constructor is shown in Listing 20. When evaluating the arguments in a function call, the arguments are enclosed in an `JSArray` constructor and evaluated as an array. The resulting list is then extracted from the `JSArrayv` value and sent to the function.

```
case exp of
  | JSUop JSSpread exp' => (case vs' of
    | [JSArrayv vs''] => (st2, env2, JSRval (vs ++ vs''))
    | _ => (st2, env2, JSRerr "Not iterable"))
  | _ => (st2, env2, JSRval (vs ++ vs'))
```

hol

Listing 19: Formal semantics for the part of array constructor handling the spread operator, where vs' contains a value in the list constructor.

4.2.2 Let and const

The differences in semantics between `let` and `const`, and `var` keywords are the scope of the variable and writability as described in Section 3.1. The semantic definition of the variable store and assignment was inspired by the semantics in Park et al. [11] and Gardner et al. [8] which describes the variable store as a list of scopes, each containing variables consisting of a name and a value. The semantics described below have been extended to include writability as well.

The HOL definitions containing the semantics for variable declaration using `let` and `const` is shown in Listing 21. The evaluation is done by a hierarchy of functions where the lowest level is the `scopeBind` function which takes a scope and a variable representation as arguments and returns an updated scope containing the variable. The variable representation consists of a tuple containing the variable name and an additional tuple containing the value and a boolean corresponding to the variable's writability, such as the tuples in `envLetDeclare` and `envConstDeclare`.

```
(js_evaluate_exp st env [JSArray exps] = let
    f = FOLDL (\a exp. case a of
      | (st', env', JSRerr err) => (st', env', JSRerr err)
      | (st', env', JSRval vs) =>
        (case js_evaluate_exp st' env' [exp] of
          | (st2, env2, JSRval vs') => (case exp of
            | JSUop JSSpread exp' => (case vs' of
              | [JSArrayv vs''] =>
                  (st2, env2, JSRval (vs ++ vs''))
              | _ => (st2, env2, JSRerr "Not iterable"))
            | _ => (st2, env2, JSRval (vs ++ vs')))
          | res => res)) (st, env, JSRval []) exps
  in case f of
    | (st3, env3, JSRval vs3) =>
        (st3, env3, JSRval [JSArrayv vs3])
    | res => res)
```
hol

Listing 20: Formal semantics for spread operator in array.

```
(OPTION_MAP f NONE = NONE) /\
(OPTION_MAP f (SOME a) = SOME (f a))

scopeBind scope var =
  scope with <| lexEnv := var :: scope.lexEnv |>

scopeLetDeclare scope (name, v) =
  case ALOOKUP scope.lexEnv name of
    | SOME _ => NONE
    | NONE => SOME (scopeBind scope (name, v))

scopesLetDeclare (scope::scopes) var =
  OPTION_MAP (\s. s :: scopes) (scopeLetDeclare scope var)

envLetDeclare env (name, v) = OPTION_MAP
  (\s. env with <| scopes := s |>)
  (scopesLetDeclare env.scopes (name, (v, T)))

envConstDeclare env (name, v) = OPTION_MAP
  (\s. env with <| scopes := s |>)
  (scopesLetDeclare env.scopes (name, (v, F)))
```
hol

Listing 21: HOL definitions containing semantics for `let` and `const` keywords. A definition of `OPTION_MAP` is included to make the other definitions easier to understand.

The function `scopeLetDeclare` checks whether the variable already exists and if it does not, it calls `scopeBind` to declare it. The `scopesLetDeclare` function applies `scopeLetDeclare` to the first scope in the inputed list of scopes. The functions highest in the hierarchy, i.e. `envLetDeclare` and `envConstDeclare`, applies `scopesLetDeclare` to the list of scopes contained in the inputed environment. As can be seen throughout the definitions, the returned value is of type `option` which either consists of the updated environment or `NONE`. The `NONE` result represents that an error occurred, namely that the variable already exists in the scope.

This part of the semantics related to variable assignment is further combined with semantics for destructuring to together represent the `let` and `const` keywords.

4.2.3 Destructuring

The semantics for destructuring is partially defined and covers the code that the new compiler is producing. The definition is shown in Listing 22.

The error handling for when a destructuring-pattern mismatches the content which is supposed to be destructured, is implemented using the *option* type which means that different errors can not be differentiated.

4.3 Exception handling

As seen in Section 3.2.3, the abstract syntax includes `throw` and `try...catch` which enables exceptions and handling of them. This affects the whole semantic definition since when encountering a `throw`-statement, the evaluation stops and steps outwards in the call stack until the exception is caught by a `try...catch`-statement or terminate if there is no `try...catch`-statement. This is currently not included in the semantics and is left as future work. A possible change to add support for exception handling to the semantics could be to add an additional result type, for runtime exceptions.

4.4 Non-terminating programs

An issue with the evaluation of programs is how to evaluate non-terminating programs, such as infinite loops or infinite recursion. This issue is solved, the same way as in the CakeML semantics, by setting a limit for the evaluation, more specifically by incorporating a clock into the evaluate function which is decremented every time an operation which could result in infinite evaluation is performed. An example of an operation which decrements the clock is a function call. The clock could be

```
(js_evaluate_bind (JSBVar name) v = SOME [(name, v)]) /\
(js_evaluate_bind (JSBArray [JSBRest b]) (JSArrayv vs) =
  js_evaluate_bind b (JSArrayv vs)) /\
(js_evaluate_bind (JSBArray []) (JSArrayv _) =
  SOME []) /\
(js_evaluate_bind (JSBArray (b::bs)) (JSArrayv []) =
  OPTION_MAP2 $++
    (js_evaluate_bind b JSUndefined)
    (js_evaluate_bind (JSBArray bs) (JSArrayv []))) /\
(js_evaluate_bind (JSBArray (b::bs)) (JSArrayv (v::vs)) =
  OPTION_MAP2 $++
    (js_evaluate_bind b v)
    (js_evaluate_bind (JSBArray bs) (JSArrayv vs))) /\
(js_evaluate_bind (JSBObject []) (JSObjectv _) =
  SOME []) /\
(js_evaluate_bind
  (JSBObject ((bn, bv)::bs)) (JSObjectv vs) = let
      rest = js_evaluate_bind (JSBObject bs) (JSObjectv vs)
    in case ALOOKUP vs bn of
      | SOME vv => if IS_SOME bv then OPTION_MAP2 $++
            (js_evaluate_bind (THE bv) vv) rest
          else OPTION_MAP (\r. (bn, vv) :: r) rest
      | NONE => if IS_SOME bv then OPTION_MAP2 $++
            (js_evaluate_bind (THE bv) JSUndefined) rest
          else OPTION_MAP (\r. (bn, JSUndefined) :: r)
            rest) /\
(js_evaluate_bind (JSBObject _) JSUndefined = NONE) /\
(js_evaluate_bind (JSBObject _) (JSLitv JSNull) =
  NONE) /\
(js_evaluate_bind _ _ = NONE)
```
hol

Listing 22: Formal semantics for array and object destructuring.

described as fuel for the evaluate function and when it reaches zero, the evaluation terminates without a result.

4.5 Verification

Due to time constraints, verification was deprioritized and never finished. The work on verification reached the point of a well defined goal, shown in Listing 23, and a solution attempt shown in Listing 24.

The goal states that for any CakeML expression in any environment, the semantics of the expression is equal to the semantics of the JavaScript expression produced by the compiler. This is done by asserting that if the compiler succeeds for a CakeML expression and if it can be evaluated, the result of the evaluation corresponds to the result of the evaluation of the produced JavaScript expression. The listing contains three function calls to `compile_exp`, `evaluate` and `js_evaluate_exp` which compiles a CakeML expression to JavaScript, evaluates a CakeML expression and evaluates a JavaScript expression respectively.

The attempt at a solution is a tactic which first performs recursive induction, by using the tactic `recInduct`, on the goal with the help of the induction theorem corresponding to the `compile_exp` function. Then a few simple cases, e.g. the `Lannot` expression, are proved with the help of a few simplifications. After the tactic has been applied to the goal, the goalstack contains the subgoals corresponding to the specific cases for each expression supported by the new compiler, except for `Lannot` which has been proved.

```
∀exps js_exps vs st st' sem_env js_env.
  env_rel sem_env.v js_env ∧
  compile_exp exps = SOME js_exps ∧
  evaluate st sem_env exps = (st',Rval vs)
    ⇒ ∃js_st js_vs js_st' js_env'.
      js_evaluate_exp js_st js_env js_exps
        = (js_st',js_env',JSRval js_vs)
      LIST_REL v_rel vs js_vs
      env_rel sem_env.v js_env'
```
hol

Listing 23: Goal specified in attempted proof.

```
recInduct compile_exp_ind >> rpt strip_tac
  >> TRY (fs [evaluate_def, js_evaluate_exp_def,
        compile_exp_def]
    >> rveq
    >> fs [evaluate_def, js_evaluate_exp_def,
        compile_exp_def]
    >> NO_TAC)
  >> TRY (rename1 `compile_exp [Lannot _ _] = _`
    >> fs [compile_exp_def, evaluate_def]
    >> rpt (first_x_assum (drule) >> strip_tac)
    >> asm_exists_tac
    >> fs []
    >> NO_TAC)
```
hol

Listing 24: Solution in attempted proof.

5

To-JavaScript Compiler

The first transformations are done by the lexer and parser which together produces a CakeML AST. The lexer and parser are not implemented in the new compiler and are instead imported from the to-machine-code compiler [12].

The next transformation is to compile the CakeML AST to a JavaScript AST which is the main part of the new compiler and is described in this chapter.

The last transformation is from the JavaScript AST to JavaScript concrete syntax.

In the examples in this chapter each example contains two code listings where the second one is the JavaScript code which is produced when compiling the CakeML code in the first listing.

5.1 Implementation

The code produced by the compiler is prefixed by imports of required helper functions and libraries, as in Listing 25. The imports will not be included in later examples of produced JavaScript code.

```javascript
const bigInt = require('./BigInteger.min.js')
const {cmljs_append,cmljs_eq,cmljs_doesmatch} =
  require('./cmljs_utils.js')
```

javascript

Listing 25: Imports in produced JavaScript code.

In addition to removing imports from the produced JavaScript code, it is beautified using *pretty-js*[1] and in some cases formatted manually to make it more comprehensible.

To distinguish variables in the compiled CakeML program from variables in the JavaScript environment, e.g. `undefined`, `window` and `global`, the compiler prefixes

[1]`https://github.com/tests-always-included/pretty-js`

all variables with `cml_`. Non user-specified variables or property names are prefixed with `cmlg_` and variables containing type definitions are prefixed with `cmltype_`.

5.1.1 Literals

As mentioned in Section 3.2.1, the supported literals are *string*, *char* and *integer* literals. Integer literals are compiled to arbitrary length integers using the BigInteger.js library. Since there is no character type in JavaScript, character literals are compiled to an object containing the character as a string.

```cakeml
val a = "foo";
val b = #"b";
val c = 1;
```

cakeml

```javascript
let cml_a = "foo";
let cml_b = { cmlg_char: "b" };
let cml_c = bigInt('5');

/*
cml_c =>
  { [Number: 5] value: 5, sign: false, isSmall: true }
cml_c.toString() => "5"
*/
```

javascript

Listing 26: Compilations of supported CakeML literals.

As can be seen in Listing 26, the integer is represented as an `bigInt`-object which can be converted to a string using its `toString` method.

The aforementioned compilation is performed by the HOL definition shown in Listing 27. In the *char* case there is a call to a function named `addGenPrefix` which prepends the `cmlg_` prefix to the supplied string. The definition returns a value of type *option* which can be either `SOME value` or `NONE` where `NONE` in this case indicates that the supplied literal is of an unsupported data type. Handling of unsupported data types and other unsupported features are explained further in Section 5.2.

5.1.2 Expression level

The new compiler is able to compile all of the CakeML expression level except for some operators. For some CakeML expressions there is a JavaScript equivalent but for some it is more complicated.

```
(compile_lit (IntLit i) = SOME (JSLit (JSBigInt i))) /\
(compile_lit (Char c) = SOME (JSObject
    [(addGenPrefix "char", SOME (JSLit (JSString [c])))]
  )) /\
(compile_lit (StrLit s) = SOME (JSLit (JSString s))) /\
(compile_lit _ = NONE)
```

hol

Listing 27: HOL definition which compiles literals.

5.1.2.1 Constructors

There are multiple types of constructors as part of the expression level. The compilation of some of the constructors (namely boolean, list and tuple constructors) are described below. In addition a description for constructors for user defined types is available in Section 5.1.3.1.

In CakeML there is no boolean type, instead booleans are represented by constructors named `true` and `false`. In contrast to CakeML, JavaScript does have the boolean type which makes it easy to compile these constructors to the corresponding boolean types as shown in Listing 28.

```
val v = true;
```

cakeml

```
let cml_v = true
```

javascript

Listing 28: Compilation of CakeML booleans.

Lists in CakeML can both be defined by enclosing comma separated values in brackets, e.g. `[1, 2, 3]`, and by the list constructor `::`, which is an infix operator which prepends an element to a list. The bracket syntax is just syntactic sugar for the `::` operator with the empty list `[]` as the last argument. `[]` can easily be compiled to the JavaScript equivalent `[]`, but `::` does not have a JavaScript equivalent.

One possible representation of `::` is to recurse through the structure and add each compiled element to a JavaScript array initializer, i.e. `[a, b, c]`. This is a valid solution to the problem but the same method is more difficult to use when destructuring when pattern matching, since the destructuring binding pattern would also need to be recursive.

Another possible representation, which is a more direct translation of the `::` operator, is to use the JavaScript spread operator in combination with an array initializer. Each `::` is represented by an array initializer containing two elements, of which the first is the single element passed to the `::` operator and the second is the spread operator applied to the rest of the list which is passed as the second argument. This is shown in Listing 29. The solution using the spread operator can be performed in reverse using the rest binding pattern to destructure JavaScript arrays, which is described in detail in Section 5.1.2.3.

```
val v = 1 :: (2 :: (3 :: []));
```

cakeml

```
let cml_v = [1, ...[2, ...[3, ...[]]]]
```

javascript

Listing 29: Compilation of list constructors.

In addition to lists, the tuple constructor also shares similarities with JavaScript arrays. Tuples in CakeML are defined as comma separated values enclosed in parentheses, e.g. `(1, "a")`, and is in contrast to lists not defined recursively. There is no tuple type in JavaScript but arrays act the same as CakeML tuples since they can hold values of different types. As previously mentioned there are multiple approaches to array initialization in JavaScript and in this situation the first approach mentioned earlier is more suitable since it is not a recursive definition and since the destructuring is not performed with a recursive binding pattern.

The array representing a tuple is wrapped in an object in the property `cmlg_tuple`. An example of how tuples are compiled is found in Listing 30.

```
val v = (1, "a");
```

cakeml

```
let cml_v = { cmlg_tuple: [1, "a"] }
```

javascript

Listing 30: Compilation of tuples.

The HOL definitions responsible for the compilation of the aforementioned constructors is shown in Listing 31 and corresponds to the previous explanations. The definition takes the contents of the constructor expression, i.e. a type id and arguments, as input and returns the corresponding JavaScript abstract syntax as an *option*. In the definition, the values of the type id are the ones created with the `Short` constructor, which is a reference to a constructor globally available, whereas a constructor that is part of a module is represented by the `Long` constructor. Which constructor each case corresponds to is for most constructors self explanatory, except for the tuple constructor and the data type constructor. In the CakeML abstract syntax, a tuple is represented by a lack of type id, i.e. `NONE`, and the expressions contained in the tuple. The case handling data types is the most generic case that pattern matches the constructor name to the variable `t`. The compilation of data type constructors is described together with data type definitions in Section 5.1.3.1.

```
(compile_con (SOME (Short "true")) _ =
  SOME [JSLit (JSBool T)]) /\
(compile_con (SOME (Short "false")) _ =
  SOME [JSLit (JSBool F)]) /\
(compile_con (SOME (Short "nil")) _ =
  SOME [JSArray []]) /\
(compile_con (SOME (Short "::")) [head; tail] =
  SOME [JSArray [head; JSUop JSSpread tail]]) /\
(compile_con (SOME (Short t)) exps =
  SOME [JSUop JSNew (JSApp (JSVar (addTypePrefix t))
    [JSArray exps])]) /\
(compile_con NONE exps =
  SOME [JSObject
    [addGenPrefix "tuple", SOME (JSArray exps)]]) /\
(compile_con _ _ = NONE)
```

hol

Listing 31: HOL definition which compiles constructors.

5.1.2.2 References & Arrays

CakeML contains features such as references and arrays where the value is stored in a value store and the returned value when initializing a reference/array is a pointer to a location in the value store. The use case of this is that multiple variables can refer to the same value and if one is updated, all others are too. JavaScript contains features providing the same functionality, although not labeled as references/pointers, namely objects. When an object is created, the return value is a pointer to the object and can therefore be used to represent references and arrays in CakeML as shown in Listing 32 and 33.

A difference in behaviour is the return value of the assignment operator which in CakeML returns *unit*, i.e. an empty tuple. In JavaScript the assignment operator returns the assigned value, which needs to be suppressed with an empty tuple. One method which achieves this goal is to call an anonymous function containing the assignment in a separate statement and the empty tuple as a return value. Another possible solution is to use the comma operator available in JavaScript which executes both operands but only returns the second one. The first solution is rather intricate in comparison to the latter which is simpler and not as different from the CakeML operator. The use of the second alternative is shown in Listing 32.

```
val a = ref 5;
val b = a
val c = a := 3;
val d = !b
```

cakeml

```
let cml_a = { cmlg_v: bigInt('5') }
let cml_b = cml_a
let cml_c = (cml_a.cmlg_v = bigInt('3'), { cmlg_tuple: [] })
let cml_d = cml_b.cmlg_v
```

javascript

Listing 32: Compilation of references. Line 3 in the JavaScript example uses the Comma operator to return an empty tuple which is the representation of Unit.

```
[Tdec (Dlet unknown_loc (Pvar "a")
  (App Aalloc [Lit (IntLit 5); Lit (StrLit "foo")]));
Tdec (Dlet unknown_loc (Pvar "b")
  (App Aupdate
    [Var (Short "a"); Lit (IntLit 3); Lit (StrLit "bar")]));
Tdec (Dlet unknown_loc (Pvar "b")
  (App Asub [Var (Short "a"); Lit (IntLit 3)]))]
```

hol

```
let cml_a = { cmlg_array:
    new Array(bigInt("5").toJSNumber()).fill("foo") }

let cml_b = (
  cml_a.cmlg_array[bigInt("3").toJSNumber()] = "bar",
  { cmlg_tuple: [] }
)

let cml_b = cml_a.cmlg_array[bigInt("3").toJSNumber()];
```

javascript

Listing 33: Compilation of arrays. The source is represented by the AST containing array operations since these operations are wrapped in functions located in modules, since modules are not supported by the new compiler.

5.1.2.3 Pattern matching

A common feature in functional programming languages, which is also present in CakeML, is pattern matching and as mentioned in Section 3.1, there is no JavaScript equivalent. The solution used in SMLtoJs by Elsman [7] is to generate `if`-statements and `switch`-statements to match a value to a specific pattern and if it is a match, use assignment to destructure the value.

In addition to using multiple assignments to perform destructuring, there is since ECMAScript 2015 an option to destructure objects in a single assignment according to a binding pattern.

Moreover there are also other possible methods to match a value against a pattern. For instance, it is possible to define a function which recursively matches a value against a pattern and returns whether it matches or not. This solution makes the produced code smaller and more human readable and does not require additions to the abstract syntax.

The new compiler implements pattern matching using the latter method and uses a separately implemented helper function called `cmljs_doesmatch`, as shown in Listing 34. The pattern sent to the function is a representation of the CakeML pattern created with JavaScript values, much like the representations of constructors in Section 5.1.2.1. The conditions in both the `cmljs_eq` and `cmljs_doesmatch` functions contains subconditition consisting of only a variable property which can either be `undefined` or a value. If it is `undefined`, then it is interpreted as `false` since a boolean is expected.

The destructuring patterns consist of multiple different binding elements to accommodate multiple data types and constructs. For instance, the CakeML pattern `Pany` which matches any value and throws the value away, is compiled to a binding element containing a JavaScript variable called `cmlg__N`, where N is replaced by an integer to make each variable unique. A literal is destructured the same way as `Pany` since the matching is already made and the value is not saved anywhere. Examples of how pattern matching is compiled is shown in Listing 35, 36 and 37.

```javascript
function every2(as, bs, f) {
  return Array.isArray(as)
    && Array.isArray(bs)
    && as.length === bs.length
    && as.every((a, i) => f(a, bs[i]))
}

function cmljs_eq(a, b) {
  return (a instanceof bigInt && b instanceof bigInt
      && a.equals(b))
    || (a.cmlg_char === b.cmlg_char)
    || (a.cmlg_v && b.cmlg_v && a === b)
    || (a.cmlg_array && b.cmlg_array && a === b)
    || (Array.isArray(a) && Array.isArray(b)
      && every2(a, b, cmljs_eq))
    || (a.tuple && b.tuple
      && every2(a.cmlg_tuple, b.cmlg_tuple, cmljs_eq))
    || (a.constructor === b.constructor
      && every2(a.cmlg_fields, b.cmlg_fields, cmljs_eq))
    || (typeof a === 'function'
      && typeof b === 'function')
    || a === b
}

function cmljs_doesmatch(pat, content) {
  return pat.pany === true
    || pat.pvar === true
    || typeof content !== 'undefined'
      && ((pat.plit && cmljs_eq(pat.plit, content))
        || (pat.pref
          && cmljs_doesmatch(pat.pref, content.cmlg_v))
        || (pat.array && Array.isArray(content)
          && (typeof pat.head === 'undefined'
            ? content.length === 0
            : cmljs_doesmatch(pat.head, content[0])
              && cmljs_doesmatch(pat.tail, content.slice(1))))
        || (pat.tuple
          && every2(pat.tuple, content.cmlg_tuple,
              cmljs_doesmatch))
        || (pat.cls && content instanceof pat.cls
          && every2(pat.fields, content.cmlg_fields,
              cmljs_doesmatch)))
    || false
}
```

javascript

Listing 34: Implementation of `cmljs_doesmatch` function.

```cakeml
val v = case 1 of
  _ => 1;
```

cakeml

```javascript
let cml_v = (function (cmlg_a) {
  return (cmljs_doesmatch(({
    pany: true
  }), cmlg_a) ? (function (cmlg__0) {
    return bigInt("1");
  })(cmlg_a) : (function () {
    throw new Error("Exception- Bind raised");
  })());
})(bigInt("1"));
```

javascript

Listing 35: Compilation of pattern matching on `Pany`.

```cakeml
val v = case Tree Nil 5 Nil of
  Tree t1 v t2 => v;
```

cakeml

```javascript
let cml_v = (function (cml_b) {
  return (function (cml_a) {
    return (function (cml_c) {
      return (function (cmlg_a) {
        return (cmljs_doesmatch(({
          cls: cmltype_Tree, fields: [
            ({ pvar: true }),
            ({ pvar: true }),
            ({ pvar: true })]
        }), cmlg_a) ? (function ({
          cmlg_fields: [ cml_t1, cml_v, cml_t2 ] }) {
          return cml_v;
        })(cmlg_a) : (function () {
          throw new Error("Exception- Bind raised");
        })());
      })(cml_c);
    })(new cmltype_Tree([ cml_a, bigInt("5"), cml_b ]));
  })(new cmltype_Nil([]));
})(new cmltype_Nil([]));
```

javascript

Listing 36: Compilation of pattern matching on `Pcon`.

```cakeml
val v = case 1 of
  n => n;
```

cakeml

```javascript
let cml_v = (function (cmlg_a) {
  return (cmljs_doesmatch(({
    pvar: true
  }), cmlg_a) ? (function (cml_n) {
    return cml_n;
  })(cmlg_a) : (function () {
    throw new Error("Exception- Bind raised");
  })());
})(bigInt("1"));
```

javascript

Listing 37: Compilation of pattern matching on `Pvar`.

5.1.2.4 Exceptions

Both CakeML and JavaScript support exceptions and handling of them, although a bit differently. In CakeML both `raise` and `handle` are expressions while in JavaScript the equivalents `throw` and `try...catch` are statements. In addition to that difference, the `handle`-expression performs pattern matching on the raised exception which is not possible in JavaScript since there is no support for pattern matching as mentioned in Section 5.1.2.3.

Using `throw` and `try...catch` as an expression is done by wrapping it in an anonymous function without arguments.

```javascript
(function() { throw new Error() })()
```

javascript

Listing 38: Using `throw` as an expression.

Wrapping `throw` statements and `try...catch` as shown in Listing 38 combined with the implementation of pattern matching described in Section 5.1.2.3, it is possible to replicate the semantics of CakeML exceptions and handling of them as shown in Listing 39.

```
val v = (raise Error) handle
    Error => 0;
```
cakeml

```
let cml_v = (function () {
  try {
    (function (cml_a) {
      return (function () {
        throw cml_a;
      })();
    })(new cmltype_Error([]));
  } catch (cmlg_error) {
    return (function (cmlg_a) {
      return (cmljs_doesmatch(({
        cls: cmltype_Error,
        fields: []
      }), cmlg_a) ? (function ({
        cmlg_fields: []
      }) {
        return bigInt("0");
      })(cmlg_a) : (function () {
        throw new Error("Exception- Bind raised");
      })());
    })(cmlg_error);
  }
})();
```
javascript

Listing 39: Compilation of exceptions and handling.

5.1.2.5 Foreign function interface

CakeML includes support for a foreign function interface (FFI) which is a way to use functionality outside the CakeML language, e.g. I/O operations. The new compiler implements support for the FFI by calling the specified foreign function with the supplied arguments and returning the function call's return value the same way as non-foreign functions. This is shown in Listing 40.

```
val v = #(sum) 1 2 3;
```
cakeml

```
let cmlg_v = sum(bigInt('1'), bigInt('2'), bigInt('3'));
```
javascript

Listing 40: Compilation of foreign function calls.

5.1.3 Declaration level

The new compiler supports almost the entire declaration level and a few interesting examples are described in this section.

5.1.3.1 Data types

JavaScript does not allow new types to be defined, it only contains the predefined types *boolean*, *number*, *string*, *null*, *undefined*, *object* and *symbol* (symbol was introduced in ECMAScript 2015). A JavaScript feature sharing similarities with datatype definitions is class declarations. In comparison to for example Java, an instance of a class does not have the class as type, instead it is of type *object* and inherits the class' prototype object.

Although defining classes does not define a new type, classes can be used to replicate the semantics of CakeML data type definitions. This is done by defining a class with a constructor which takes a list of fields as arguments and saves the list. Value constructors are represented by classes which extend the class representing the type. This is shown in Listing 41.

CakeML data types can be specified using type variables to depict a generic type. Since JavaScript is dynamically typed, nothing needs to be done to replicate this behaviour.

The HOL definitions used to perform the aforementioned compilation is shown in Listing 42. The list of class declarations is produced by the function named `compile_type_def`, which uses the function `type_class_def` to create class definitions for both the main class and the sub-classes. As can be seen in the definition, the types are prefixed with `cmltype_` by the function `addTypePrefix`.

5.1.3.2 Exceptions

Defining exception types is much like defining data types in CakeML and the compiled JavaScript code looks much like the one for data types. The main difference from data types is that exceptions do not allow multiple value constructors which results in JavaScript code only consisting of one class without any sub-classes, as shown in Listing 43.

```
datatype 'a tree = Nil | Tree ('a tree) 'a ('a tree);
val a = Tree Nil 5 (Tree Nil 3 Nil);
```

cakeml

```
let cmltype_tree = class {
  constructor(cmlg_fields) {
    this.cmlg_fields = cmlg_fields
  }
};
let cmltype_Nil = class extends cmltype_tree {};
let cmltype_Tree = class extends cmltype_tree {};

let cml_a = new cmltype_Tree([
  new cmltype_Nil([]),
  bigInt('5'),
  new cmltype_Tree([
    new cmltype_Nil([]),
    bigInt('3'),
    new cmltype_Nil([])
  ])
])
```

javascript

Listing 41: Compilation of data types.

```
type_class_def extends name = let
    constructor = ("constructor", [addGenPrefix "fields"],
      JSExp (JSAssign (JSObjectProp (JSVar "this")
          (addGenPrefix "fields"))
        (JSVar (addGenPrefix "fields"))))
  in JSLet (JSBVar name) (JSClass NONE extends
      (if IS_NONE extends then [constructor] else []))

compile_type_def (_, name, fields) =
  type_class_def NONE (addTypePrefix name) ::
    MAP (type_class_def (SOME (addTypePrefix name))
      o addTypePrefix o FST) fields

compile_dec (Dtype _ type_defs) =
  SOME (FLAT (MAP compile_type_def type_defs))
```

hol

Listing 42: The HOL definitions responsible for compilation of data type definitions.

```
exception Error int;
```

cakeml

```
let cmltype_Error = class {
  constructor(cmlg_fields) {
    this.cmlg_fields = cmlg_fields
  }
}
```

javascript

Listing 43: Compilation of exception definitions.

5.1.3.3 Function declarations

A simple function expression in CakeML works similar to how a function expression in JavaScript works. However, a key difference is that JavaScript functions can have multiple parameters while CakeML functions can only have one. The compilation can therefore be done by transforming a CakeML function to a JavaScript function with only one parameter, as can be seen in Listing 44.

```
compile_exp [Fun par exp] = OPTION_MAP
  (toList o (JSAFun [JSBVar (addVarPrefix par)])
    o toList o JSReturn o HD)
  (compile_exp [exp])
```

hol

Listing 44: HOL definition responsible for compilation of a function definition.

In CakeML a function is represented by a closure consisting of the environment (which the function was declared in and is used to resolve free variables), parameter name and function body. This representation does not allow mutual recursion since the function defined first is not aware of functions defined after. To solve this a recursive closure is used, consisting of a list of functions where each function is represented by its name, parameter and body.

In JavaScript, the functionality provided by recursive closures are available in all function definitions. That is due to the fact that the environment tied to the closure continues to be mutated with later definitions and assignments after the closure has been created. The free variables do not need to exist in the environment when the closure is created, instead they need to exist when the function is executed. The difference between how closures work introduces issues regarding variable shadowing, which is discussed in more detail in Section 6.1.1.

5.2 Unsupported CakeML features

If a CakeML program contains language features not supported by the new compiler, the compiler still compiles the parts which are supported and leaves the other parts, as CakeML AST, in a special expression or statement. This is shown in Listing 45. Handling unsupported features this way makes it possible to use the parts of a program that was compiled correctly. Furthermore, it makes it easy to see which features are unsupported when continuing development of the compiler in the future.

```
[Tdec (Dlet unknown_loc (Pvar "v") (Lit (Word8 1w)))]
```

hol

```
[JSVarDecl (JSBVar "cml_v")
  (Exp_not_compiled (Lit (Word8 5w)))]
```

hol

```
let cml_v = "Expression not supported by compiler";
```

javascript

Listing 45: Compilation of unsupported CakeML features from CakeML AST to JavaScript AST to JavaScript concrete syntax

5.3 Web page example

In order to demonstrate that the compiler produces usable code that integrates seamlessly with external JavaScript code, we have developed an example application where compiler output is used in a simple calculator web page which calculates the sum of a list of integers. The web page includes two JavaScript functions that interface with the DOM and the CakeML program, which are shown in Listing 46. The CakeML program, shown in Listing 47, consists of two functions which calculate the sum of a list of integers and calls the foreign function `print` with the result. The CakeML program is compiled to JavaScript and imported in the web page to make the sum functionality work.

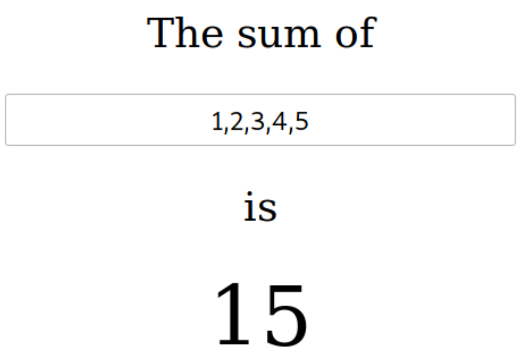

Figure 5.1: Screenshot of demo application using a compiled CakeML program.

```javascript
function handleInput() {
  let el = document.getElementsByTagName("input")[0]
  let ints = el.value.split(',')
    .filter(v => v)
    .map(v => bigInt(v))
  cml_printSum(ints)
}

function print(s) {
  document.getElementById('sum').textContent = s
}
```

javascript

Listing 46: JavaScript source code in demo application.

```
fun sum l = case l of
    [] => 0
  | (n :: ns) => n + (sum ns);

fun printSum l = let
    val sum = sum l;
  in #(print) sum end;
```

cakeml

Listing 47: CakeML program used to calculate sum of list of integers in CakeML program.

6 Conclusion

This chapter presents my thoughts about the result, compares the new compiler and semantics to similar projects and proposes future work.

6.1 Discussion

The semantics described in Chapter 4 is based on and translated from the existing JavaScript formalizations by Gardner et al., Park et al. and the official ECMAScript specification. Our definitions in HOL4 have been manually tested but lacks testing with the ECMAScript conformance suite. This means that there is no guarantee that the semantics are completely correct.

The same applies to the new compiler, which has also been tested manually and has not been formally verified.

6.1.1 Limitations

The new compiler implementation contains two known limitations. The first one is that a result of using a big integer library to represent arbitrary length integers is that when interacting with the JavaScript standard library, integers cannot be sent as arguments. This makes it necessary for FFI-functions to be wrapped in a function which calls the `toJSNumber` on all integers sent as arguments. In addition to complicating the process of using the standard library, it also reintroduces the possibility of lost precision.

There is currently no solution to this issue which meets all criteria. A possible solution is to use a built-in integer type when a future ECMAScript edition includes support for arbitrary length integers. The implementation will hopefully make conversion between numbers and integers seamless.

The other known limitation is that since environments are handled differently in CakeML and JavaScript, as described in Section 5.1.3.3, compilation of variable definitions and variable lookup does not preserve semantics when shadowing occurs in the same scope. More specifically, the JavaScript environment is updated in already existing closures, whereas CakeML environments are not. An example of this is the CakeML program in Listing 48 which would be compiled to a JavaScript program resulting in an error when the second definition of `a` is executed.

```
val a = 1;
val a = 2;
```

cakeml

```
let a = bigInt('1');
let a = bigInt('2');
```

javascript

Listing 48: Example of a program containing shadowing

A solution to this issue could be to pass a state, containing a map from CakeML variable name to a generated JavaScript variable name, to the `compile_exp` function. The JavaScript variable name could be the CakeML variable name suffixed by an integer which is incremented when shadowing occurs. This would ensure that variable lookups in closures uses the contents of the variable before the shadowing definition.

6.1.2 Target language

A question that emerged early on during the thesis work, was if JavaScript is the best choice of target language for the new compiler. The choice was between JavaScript, asm.js[1] and Web Assembly[2].

The benefits of using asm.js would be that it consists of a small subset of JavaScript which results in much more optimized code, less complex abstract syntax and semantics. The disadvantages are that it is only a work in progress with no stable release, there do not exist any formal semantics previously defined and it is not an official web standard.

Web Assembly would have been a better choice if optimization and if less complex abstract syntax and semantics would have been desirable. There exists small step semantics implemented in Isabelle[3] and work in progress semantics implemented using the K Semantic Framework, that could be used as inspiration. The disadvantages are that it was not fully supported in all major browsers in the beginning of this thesis, and since it is on another level of abstraction it would make the compilation and verification process much more complex.

[1]`http://asmjs.org/`
[2]`http://webassembly.org/`
[3]`https://isabelle.in.tum.de/`

The advantages with using JavaScript is that there exists formal semantics for substantial parts of the language, that it is fully implemented in all major browsers and that it is on a similar level of abstraction as CakeML which makes the compilation process less complex. A disadvantage with JavaScript is that it's semantics is complex and contains a lot of edge cases.

6.1.3 Related work

The most similar currently existing compiler is SMLtoJs by Elsman [7] which compiles Standard ML to JavaScript and is implemented in Standard ML. In contrast to the SMLtoJs, the new compiler explores how to deal with the differences between ML and JavaScript when compiling from one to the other, without passing through intermediate languages. As a result of not passing through any intermediate languages, each statement and expression in the produced JavaScript code better corresponds to the CakeML input in most cases.

The previously existing formal semantics for JavaScript by Gardner et al. [8] and Park et al. [11] covers ECMAScript 5. The semantics presented in this thesis extends the semantics in literature with coverage of the new variable declaration keywords, the spread operator and a subset of the binding elements from ECMAScript 2015.

6.1.4 Future work

This introduction of the new compiler opens up for future work in the areas of expanding the compiler, verifying it and to use it to compile itself and the to-machine-code compiler to be run in a web browser.

6.1.4.1 Semantics

A direction the work done in this thesis could be further developed is to expand the coverage of the semantics. The current definitions contain semantics for a subset of the features supported by the abstract syntax and used in the new compiler. Further work would reasonably start by expanding the semantics to all features covered by the abstract syntax, e.g. arrays, objects, some of the operators, the assignment expression, the conditional expression, class definitions and the statements not related to variable definition.

To ensure that the semantics are correct they could be tested against the ECMAScript conformance suite. This would either require a parser and lexer from JavaScript source code to the abstract syntax specified in this thesis, or someone to manually parse the tests to an AST. However it is performed, it would indicate whether or not the semantics is correct.

Since this thesis did not reach the initial goals of verifying the new compiler, verification is an important future step.

6.1.4.2 Expand compiler

The new compiler covers a substantial subset of the CakeML language and could be expanded to cover the complete language. Some language features would require less work than others and some are less useful than others, resulting in multiple different paths for future implementation.

There are three major parts of the language which are not currently supported. The first one is words, i.e. the types *word8* and *word64*. The second one is a subset of the operators, mainly for words but also some operators on literals.

The third major part of the language which is not currently supported by the new compiler is modules. The advantage with support for modules would be that great parts of the *basis*-library will be possible to compile, which will enable programs to have access to the standard CakeML environment. This would make it possible to compile existing CakeML programs to JavaScript.

There are multiple JavaScript features that could be used to reproduce the behaviour of CakeML modules. A solution could include classes with static methods and/or properties representing values and functions in the module. Another solution could be to create a file for each module and use the export and import features of JavaScript, although it would be better to produce a single file/string containing the entire compiled code.

6.1.4.3 Compile compilers to JavaScript

A very interesting use case of a CakeML to JavaScript compiler is to be able to compile both the new compiler and the to-machine-code compiler, to JavaScript. This would make it possible to develop an interactive web page enabling users to run examples and develop simple programs which can be compiled and run directly in the web browser, similar to the SMLtoJsOnline website[4]. This would also open up for the possibility to develop an interactive look into the compilation process that allows users to investigate what a compiler is doing in each step, which would make a great tool for understanding the compilers and how the language works.

[4]`https://smlserver.org/ide/`

6.2 Conclusion

The new compiler described in this thesis is able to compile a subset of CakeML to JavaScript. The new compiler is built as an additional back end to the to-machine-code compiler and uses the already existing parser and lexer. The work done also includes big step operational semantics for a subset of JavaScript including some language features from ECMAScript 2015 and more recent versions.

This thesis explores how to deal with the differences between CakeML and JavaScript and shows that it is possible to compile language features of CakeML to JavaScript, and extends the formal semantics for JavaScript, thus reaching one step closer to a formally verified source to source compiler from CakeML to JavaScript.

The implementation was made in the HOL4 theorem prover to make future verification of the compiler as seamless as possible. The implementation can be found at `https://github.com/raksooo/cakeml/tree/javascript/javascript`.

Bibliography

[1] Hol interactive theorem prover. `https://hol-theorem-prover.org/`.

[2] Stack overflow developer survey 2018. `https://insights.stackoverflow.com/survey/2018/`, 2018.

[3] ECMA Ecmascript. Language specification. `https://www.ecma-international.org/ecma-262/5.1/`, 2011.

[4] ECMA Ecmascript. Language specification. `https://www.ecma-international.org/ecma-262/6.0/`, 2017.

[5] ECMA Ecmascript. Language specification. `https://www.ecma-international.org/ecma-262/8.0/`, 2017.

[6] Anton Ekblad. A distributed haskell for the modern web. 2015.

[7] Martin Elsman. Smltojs: Hosting a standard ml compiler in a web browser. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients*, PLASTIC '11, pages 39–48, New York, NY, USA, 2011. ACM.

[8] Philippa Anne Gardner, Sergio Maffeis, and Gareth David Smith. Towards a program logic for javascript. *SIGPLAN Not.*, 47(1):31–44, January 2012.

[9] Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. Cakeml: a verified implementation of ml. In *ACM SIGPLAN Notices*, volume 49, pages 179–191. ACM, 2014.

[10] Magnus O. Myreen and Scott Owens. Proof-producing synthesis of ml from higher-order logic. *SIGPLAN Not.*, 47(9):115–126, September 2012.

[11] Daejun Park, Andrei Stefănescu, and Grigore Roşu. Kjs: A complete formal semantics of javascript. In *ACM SIGPLAN Notices*, volume 50, pages 346–356. ACM, 2015.

[12] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. A new verified compiler backend for cakeml. *SIGPLAN Not.*, 51(9):60–73, September 2016.