RWTH Aachen University
Software Modeling and Verification Group

Bachelor Thesis

Modelling a purely functional subset of ECMAScript 2015

Raoul Schaffranek

Thesis Supervisor: Prof. Dr. Thomas Noll

Second Examiner: Prof. Dr. Jürgen Giesl

Submitted: February 29, 2016

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Objective 1

1.3 Outline of the Thesis 2

2 Preliminaries **3**

2.1 Concrete and Abstract Syntax 3

2.2 Operational Semantics 7

2.3 Functional Purity 10

3 Language Model **12**

3.1 Lambda Calculus 12

3.1.1 Concrete Syntax 12

3.1.2 Abstract Syntax 16

3.1.3 Notions of reduction 18

3.1.4 An abstract syntax machine 21

3.1.5 Call-by-value 23

3.1.6 Call-by-name 24

3.1.7 Call-by-need 24

3.2 Basic Constructors and Constants 28

3.2.1 Syntax 28

3.2.2 Call-by-value 30

3.2.3 Call-by-name 33

3.2.4 Call-by-need 34

3.3 Compound datatypes 34

3.3.1 Syntax 34

3.3.2 Operational Semantics 38

3.4 Recursion 38

3.4.1 Syntax 38

3.4.2 Operational Semantics 45

4 Conclusion **53**

1 Introduction

1.1 Motivation

ECMAScript 2015 is the newest specification of the ECMAScript [ES] programming language[6], which is also commonly known by its unofficial name *JavaScript*. The language can look back on an interesting history which is coined by too many events and environmental changes in order to be repeated here. Instead, only the recent relevant history is depicted to set the thesis in its environmental context, the interested reader may consult [13] for more background information on the language's history.

ES originated as a scripting language for the browser but has outreached to many other domains since then. Today it is an integrated part of various graphical user interface toolkits, including systems from *Windows* and *Linux* alike[7], e.g. *WinJS*, *Gjs* and *Seed*. *Node.js* made ES popular for server-side processing, robotics and real-time systems. Concerning the browser, ES still has its exclusive place, because no other programming language has made it to broad acceptance in this specific environment[7].

This last observation is anticipated with caution by the community. The reasons for this can be roughly split up into sociocultural ones and technical ones. Talking of the first category, programmers often evolve strong preferences and evangelism for different programming constructs and paradigms during their profession. When programming for the browser, however, programmers were forced into ES for a long time. Conversely, this gave raise to various dialects of ES and even to entirely alien languages which can be compiled to ES. For example *TypeScript* is a dialect which extends ES with a static type system, *JSX* extends ES with a domain specific language for virtual operations on the *Document Object Model*. These motions recently spawned a joint effort to develop a language which is specifically suited to serve as a compile-target language which is executed natively by the browser - hence the name: *WebAssembly*. Although these standards are not yet finalised and under heavy alteration, it seems highly probable that some kind of assembly language will see the light of the day.

1.2 Objective

As the previous examples showed ES performs especially well in rather statefull and often asynchronous domains. This combination is often subject to synchronisation problems and bugs, which are hard to track. Common solutions to these problems are often borrowed from functional programming languages, which more often than not enforce immutable state. On a completely different track, ES programs often underlie heavy evolution during their lifetime, which forces programmers to counteract with high abstractions and modularization. Research[10] shows that functional aspects, i.e. higher order functions and lazy evaluation, can contribute to modularization. Now, it is the case that ES is a multi-paradigm programming language with support for a wide range of programming constructs from procedural over object-oriented to functional programming. The purpose of the thesis is to identify the purely functional parts of the language, which arguably do not mutate state nor perform any other side-effects. For this sake, reduction systems are constructed to model the operational semantics of ES. Alternatives and choices in the design-phase of the modelling are also presented. The focus is on *how* to obtain a purely functional subset from the given impure language - the actual appearance is of lower interest. The purpose is to provide a framework to guide the future evolution of the language towards an even better host for functional programming concepts.

1.3 Outline of the Thesis

After these introductionary words, the thesis develops the necessary preliminaries that a reader without prior knowledge of *programming language theory* needs to follow the thoughts of the thesis. Nevertheless, prior knowledge in the fields of formal languages, functional programming and with ES will be highly beneficial. The vocabulary is developed exemplary by investigating the syntax and operational semantics of a small formal language for boolean algebra. The chapter closes with a brief comprehension of the work from Amr Sabry on purely functional programming languages[14] and recapitulates his definition of purity, which states that a purely functional programming language should be independent of certain parameter-passing-strategies.

The main part then successively builds a purely functional model for a subset of ES in four steps. The intermediate languages obtained from each step are called Λ^1, Λ^2,Λ^3 and Λ^4 respectively. The first subsection introduces the traditional λ-calculus and establishes a connection to ES. This is done step-wise: First, the concrete syntax of ES for *variables*, *functions* and *function application* is depicted. Second, the relevant aspects are factored out into an abstract syntax. Finally, (weakly equal) reduction semantics for different parameter-passing-strategies are developed. The language Λ_1 then makes up the starting-point for various extensions, which are developed in the up-following subsections. The scheme is in general the same for each iteration step as shown above. The second subsection extends Λ_1 with basic data-constructors and primitive operations for common primitive data-types, i.e. floating point arithmetic and boolean algebra. The third subsection then introduces compound data-types, i.e. lists and records. Finally, a `letrec`-expression is discussed to enable (mutual) recursion.

The conclusion briefly summarizes the findings. To that end stands a suggestion of purely functional programming features which are not yet available in ES and which inclusion would solidify the functional nature of the language for the future.

2 Preliminaries

2.1 Concrete and Abstract Syntax

Programs are usually written down as ordinary text. Of course, mixing arbitrary symbols together in an uncontrolled manner does not end in meaningful results for the most cases. Just like natural language, programming languages are subject to certain rules which tell the programmer which programs are valid according all possible combinations of symbols. These rules apply at different levels of detail: On a lowest level, a rule might tell the programmer which symbols can be used at all. This set of symbols is called the *alphabet* which is conventionally abbreviated with Σ. On a different level, a rule-set could tell the programmer in which order the symbols might be combined to form reasonable tokens. Rules on this level make up the *lexical* details of the language. On yet another level, rules can specify the order in which tokens might be put together to form words or sentences.

For a very simplistic example take the language of boolean algebra $\mathcal{L}_B$: The alphabet consists of symbols for logical truth (1) and falsehood (0), as well as symbols for the boolean operators ”not” ($\neg$), ”and” ($\wedge$), and ”or” ($\vee$) as well as parenthesis ([,]). Formally: $\Sigma \overset{\text{def}}{=} \{0, 1, \neg, \wedge, \vee, [,]\}$. The words of $\mathcal{L}_B$ are all constants, literals and formulas (without variables) according to the following rules:

- $0 \in \mathcal{L}_B$
- $1 \in \mathcal{L}_B$
- if $B \in \mathcal{L}_B$, then $[\neg B] \in \mathcal{L}_B$
- if B_1, $B_2 \in \mathcal{L}_B$, then $[B_1 \wedge B_2] \in \mathcal{L}_B$
- if B_1, $B_2 \in \mathcal{L}_B$, then $[B_1 \vee B_2] \in \mathcal{L}_B$

Put differently, $\mathcal{L}_B$ is the smallest set which satisfies the previous rules. This definition of $\mathcal{L}_B$ is very concise. The following alternative definition is intentionally more verbose to illustrate some of the choices that language designers have even at this early stage. This time, the language is build up from the alphabet of all small and capital letters from the English alphabet plus a space-character (␣) and parenthesis ([,]). With this alphabet, the following can be used to construct all desired expressions:

- $\texttt{False} \in \mathcal{L}'_B$
- $\texttt{True} \in \mathcal{L}'_B$
- if $B \in \mathcal{L}'_B$, then $[\texttt{Not}\textvisiblespace B] \in \mathcal{L}'_B$
- if B_1, $B_2 \in \mathcal{L}'_B$, then $[\texttt{And}\textvisiblespace B_1\textvisiblespace B_2] \ \in \mathcal{L}'_B$
- if B_1, $B_2 \in \mathcal{L}'_B$, then $[\texttt{Or}\textvisiblespace B_1\textvisiblespace B_2] \in \mathcal{L}'_B$

There are two important differences to spot here: First, even the boolean constants are now made of the concatenation of multiple characters from the alphabet. Second, the boolean operators are now denoted using a prefix- instead of an infix-notation.

The previous rules make it clear how to construct the words of the language inductively. Nevertheless, some additional formalism will be beneficial here, mainly to align the thesis with the ES specification, but also to prevent future confusion when the language model grows. The following definitions are based on [2]:

Definition 1. A context-free grammar $\mathcal{G}$ is a tupel (V, Σ, R, S), where

- V is a non-empty set of non-terminal symbols
- Σ is a non-empty set of terminal symbols, with $\Sigma \cap V = \emptyset$
- $R \subseteq V \times (V \cup \Sigma)^*$ is a set of production-rules, where the asterisk denotes the Kleene closure
- $S \in V$ is the start-symbol

Definition 2. Given a context-free grammar $\mathcal{G} = (V, \Sigma, R, S)$ and two words $w, w' \in (V \cup \Sigma)^*$ with $w = \beta A \gamma, w' = \beta \alpha \gamma$, where $\alpha, \beta, \gamma \in (V \cup \Sigma)^*, A \in \Sigma$. Then w' is derivable from w in one step (written $w \vdash w'$) if there exists a production rule of the form $(A, \alpha) \in R$

w' is derivable from w in n steps (written $w \vdash^n w'$) if there is sequence of words $w_0, w_1, \cdots, w_n$, such that: $w_0 = w$, $w_n = w'$ and $w_i \vdash w_{i+1}$ for all $i : 0 \leq i \leq n-1$. If the number of derivation-steps is dispensable, it also written $w \vdash^* w'$.

Definition 3. The induced language $\mathcal{L}(\mathcal{G})$ of a context-free grammar $\mathcal{G} = (V, \Sigma, R, S)$ is the smallest set, such that: $w \in \mathcal{L}(\mathcal{G})$ if $S \vdash^* w$ and $w \in \Sigma^*$. A language which can be induced by a context-free grammar is called context-free.

For instance, the following figure respectively shows context-free grammars for $\mathcal{L}_B$ and $\mathcal{L}'_B$.

$$
\begin{aligned}
&\mathcal{G}_B = (V, \Sigma, R, S), \text{ where}\\
&V = \{B, C\}\\
&\Sigma = \{0, 1, \neg, \vee, \wedge, [,]\}\\
&R = \{(B, 0), (B, 1), (B, [C]), (C, \neg B), (C, B \vee B), (C, B \wedge B)\}\\
&S = B
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{G}'_B = (V', \Sigma', R', S'), \text{where}\\
&V' = \{B\}\\
&\Sigma' = \begin{matrix}\{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}, \mathtt{F}, \mathtt{G}, \mathtt{H}, \mathtt{I}, \mathtt{J}, \mathtt{K}, \mathtt{L}, \mathtt{M}, \mathtt{N}, \mathtt{O}, \mathtt{P}, \mathtt{Q}, \mathtt{R}, \mathtt{S}, \mathtt{T}, \mathtt{U}, \mathtt{V}, \mathtt{W}, \mathtt{X}, \mathtt{Y}, \mathtt{Z}\\ \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{g}, \mathtt{h}, \mathtt{i}, \mathtt{j}, \mathtt{k}, \mathtt{l}, \mathtt{m}, \mathtt{n}, \mathtt{o}, \mathtt{p}, \mathtt{q}, \mathtt{r}, \mathtt{s}, \mathtt{t}, \mathtt{u}, \mathtt{v}, \mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}, [,], \text{␣}\}\end{matrix}\\
&R' = \{(B, \mathtt{False}), (B, \mathtt{True}), (B, [\mathtt{not}\text{␣}B]), (B, [\mathtt{or}\text{␣}B\text{␣}B]), (B, [\mathtt{and}\text{␣}B\text{␣}B])\}\\
&S' = B
\end{aligned}
$$

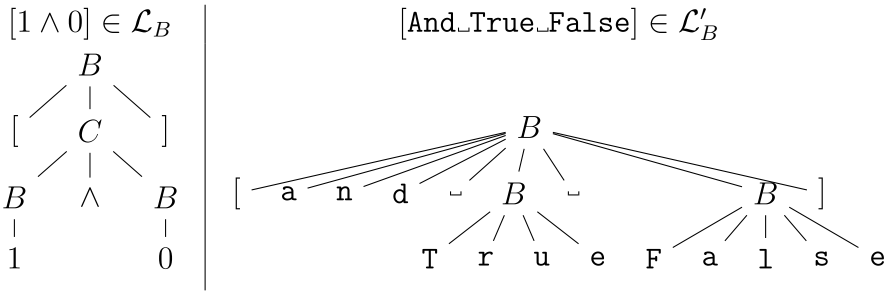
Notation. For readability a production rule $(A, \alpha) \in R$ can also be written as $A ::= \alpha$. Production rules $(A, \alpha_2) \in R$ and $(A, \alpha_1) \in R$ can be collapsed into a single rule of the form: $A ::= \alpha_1 \mid \alpha_2$. For example, the production-rules of $\mathcal{G}_B$ may now be written as:

$$
\begin{aligned}
B &::= 0 \mid 1 \mid [C]\\
C &::= \neg B \mid B \vee B \mid B \wedge B
\end{aligned}
$$

Derivations are reproducible plans to show the containment of word to a language via a given grammar. This is to say, the focus is on *how* to get from the start-symbol to the desired word. Depending on the purpose, the interest is sometimes not on the actual sequence of derivation steps, but more on the structure of a given word. For instance, consider the following word $[1 \wedge 0] \in \mathcal{L}(\mathcal{G})$. It can be derived from B in $\mathcal{G}_B$ by either

- $B \vdash [C] \vdash [B \wedge B] \vdash [1 \wedge B] \vdash [1 \wedge 0]$ or
- $B \vdash [C] \vdash [B \wedge B] \vdash [B \wedge 0] \vdash [1 \wedge 0]$.

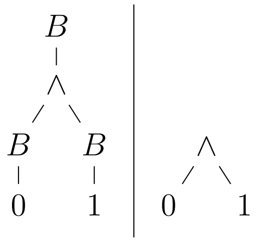
The derivations only differ in the third step. The parse-tree is an alternative perspective on the derivation of a word which hides the actual sequence that is taken and emphasizes the structure of the word. The following table pictures the parse-trees of the running examples, the formulas can be metered from the leaf-nodes in tree-order.


[1 ∧ 0] ∈ $\mathcal{L}_B$
B
[
C
]
B
∧
B
1
0
[And␣True␣False] ∈ $\mathcal{L}'_B$
B
[
a
n
d
␣
B
␣
B
]
T
r
u
e
F
a
l
s
e

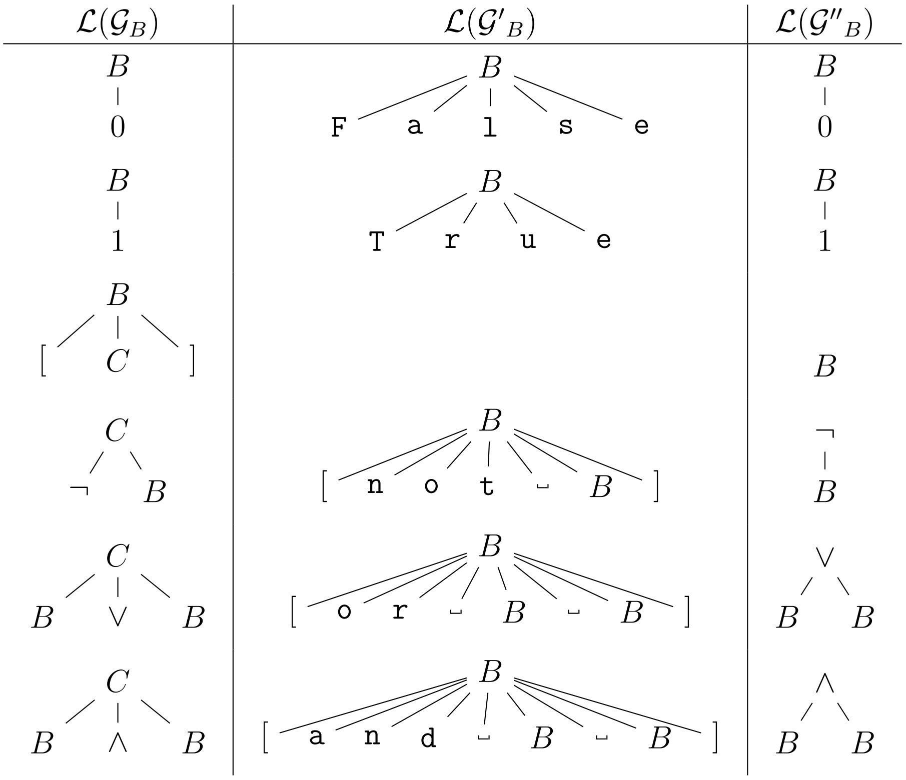
The languages $\mathcal{L}_B$ and $\mathcal{L}'_B$ have a different appearance, but it should be clear that they both encode the same boolean entities. For example $1 \in \mathcal{L}_B$ and `True` $\in \mathcal{L}'_B$ are just two different representations of logical truth. This level of detail is unhandy when developing the semantics of a language. Therefore, the *abstract syntax* of a language factors out its lexical details and further emphasizes the *relevant* structure of the words. Typical neglects during this abstraction-step involve associativity of operators, differences between prefix-, infix- and postfix-notation and low-level lexical details. The abstract syntax, in this context, does not involve new conceptual work, it is simply an organized construction of a grammar, such that the parse-trees are very handy to work with. Take the following grammar as an example:

$$\begin{aligned}
&\mathcal{G}''_B = (V, \Sigma, P, B), \text{ where}\\
&V = \{0, 1\},\\
&\Sigma = \{B, \neg, \vee, \wedge\}\\
&B ::= 0 \mid 1 \mid \neg \mid \vee \mid \wedge\\
&\neg ::= B\\
&\vee ::= BB\\
&\wedge ::= BB
\end{aligned}$$

The parse-tree $0 \wedge 1 \in \mathcal{L}(\mathcal{G}'')$ is indeed extremely concise. Collapsing administrative nodes like the start-symbol with its direct successor contributes even further to the readability as shown on the right-hand side of the following table:



What is still missing is the connection of the concrete syntax and the abstract syntax. It is possible to obtain the abstract parse-tree from a concrete parse-tree through a simple induction over the shape of the concrete parse-tree. If this process leads to a function, one speaks of a *parsing function*. The following table inductively defines the parsing functions from $\mathcal{L}(\mathcal{G}_B)$ and $\mathcal{L}(\mathcal{G}'_B)$ to $\mathcal{L}(\mathcal{G}''_B)$.


$\mathcal{L}(\mathcal{G}_B)$
$\mathcal{L}(\mathcal{G}'_B)$
$\mathcal{L}(\mathcal{G}''_B)$
B
0
F a l s e
T r u e
1
[C]
C
¬ B
[n o t ␣ B]
∨
B ∨ B
[o r ␣ B ␣ B]
∧
B ∧ B
[a n d ␣ B ␣ B]

2.2 Operational Semantics

The last subsection developed two alternative syntactical representations for logical expressions. It further illustrated that with the notion of a *parsing function* it becomes possible to equate expressions from both representations with respect to a common abstract syntax. For example the abstract expressions which result from parsing $[1 \wedge 0] \in \mathcal{L_B}$ and $[\texttt{And␣True␣False}] \in \mathcal{L'}_B$ are provable equal. However, this does not answer the question whether two expressions within a fixed abstract syntax are equal. Intuitively, the above expressions are just two obfuscated ways to encode logical falsehood. One should expect that the evaluation of *any* formula will give either true or false - these are all possible answers of the evaluation process. Put formally, the set of answers is:

$$A ::= 0 \mid 1$$

The *operational semantics* of language should provide all necessary information, that is needed to fully mechanise the process of evaluating an arbitrary expression to an answer. In other words, one is interested in the atomic steps which can be performed to *reduce* an expression to an arguably simpler one. The identification of these atomic steps, their environmental influence and the order in which they must be performed is the primary task of *operational semantics.* Fortunately, purely functional programs do not mutate their environment. This greatly simplifies the work, because no additional data, other than the syntax, must be maintained by the *reduction system.* For example, there is no program-heap to take care of.

For the derivation of sufficient *reduction rules*, it is helpful to first consider the easiest possible expressions and then work backwards against more complicated cases. If the subject expression is already an answer, there is clearly no work to do - the evaluator is done. If the subject expression is the negation of an answer, then there are just two possible cases which must be respected depending on the value of the sub-expression. These thoughts lead to the following rules for logical negation:

$$\neg 1 \to 0 \tag{1}$$
$$\neg 0 \to 1 \tag{2}$$

The left-hand side of such a rule is called a *redex*, the right-hand side is called its *contractum.* The complete rule is often referred to as a *reduction axiom* or simply *axiom.* Similar axioms should be established for the logical conjunction and disjunction. These cases are slightly more complex, because they involve two operands per operation. To this end, simply assume that the first operand is already known. This gives the rules:

$$(0 \vee B) \rightarrow B \tag{3}$$
$$(1 \vee B) \rightarrow 1 \tag{4}$$
$$(0 \wedge B) \rightarrow 0 \tag{5}$$
$$(1 \wedge B) \rightarrow B \tag{6}$$

With the *reduction axioms* from above, it is now possible to reduce, for example, the expression $(0 \vee (1 \wedge (\neg 0)))$ to 1 by the following steps:

$$\begin{aligned}
(0 \vee (1 \wedge (\neg 0))) &\rightarrow (1 \wedge (\neg 0)), && \text{by } 3\\
(1 \wedge (\neg 0)) &\rightarrow (\neg 0), && \text{by } 6\\
(\neg 0) &\rightarrow 1 && \text{by } 2
\end{aligned}$$

Now, consider the very similar expression $(0 \vee ((\neg 0) \wedge 1))$, where only the order of the operands of the logical conjunction has been reversed. The first reduction step can proceed as in the first example, but then a problem appears:

$$\begin{aligned}
(0 \vee ((\neg 0) \wedge 1)) &\rightarrow ((\neg 0) \wedge 1), && \text{by } 3\\
((\neg 0) \wedge 1) &\rightarrow ?
\end{aligned}$$

Which contractum can be put in the place of the question mark? There is no redex which matches $((\neg 0) \wedge 1)$, hence there is no way to go. The reduction system is said to be *stuck*. This comes at no surprise because the assumption that the left-hand side of a conjunction is known, which has been established earlier, is now violated. Intuitively the expression should reduce to 1, but the reduction axioms do not permit this. The axiom system is said to be *incomplete*[8, 11] with respect to the desired semantics. Completeness is an important property that often arises with axiom-systems. Fortunately, for the running example, completeness can be restored by the adoption of the following additional axioms.

$$(B_1 \vee B_2) \rightarrow (B_1' \vee B_2), \qquad \text{if } B_1 \rightarrow B_1' \tag{7}$$
$$(B_1 \vee B_2) \rightarrow (B_1 \vee B_2'), \qquad \text{if } B_2 \rightarrow B_2' \tag{8}$$
$$(B_1 \wedge B_2) \rightarrow (B_1' \wedge B_2), \qquad \text{if } B_1 \rightarrow B_1' \tag{9}$$
$$(B_1 \wedge B_2) \rightarrow (B_1 \wedge B_2'), \qquad \text{if } B_2 \rightarrow B_2' \tag{10}$$

With these new axioms the stuck reduction from above can be continued in the following way:

$$\begin{aligned} ((\neg 0) \wedge 1) &\rightarrow (1 \wedge 1), && \text{by } 2,7 \\ (1 \wedge 1) &\rightarrow 1, && \text{by } 6 \end{aligned}$$

So far, the focus has been on the atomic steps, which an evaluator needs to perform to get to answer. Now, the big picture is taken into account. It has been seen that the expression $(0 \vee ((\neg 0) \wedge 1))$ is related to 1 by a sequence of reduction steps. This observation can be formalised mathematically as the reflexive and transitive closure of the reduction-relation $\rightarrow$. To avoid confusion the reflexive, transitive closure is written with a double-headed arrow:

$$\begin{array}{lll} B \twoheadrightarrow B', & \text{if } B \rightarrow B' & \\ B \twoheadrightarrow B & & (reflexivity) \\ B \twoheadrightarrow B'', & \text{if } B \twoheadrightarrow B' \text{ and } B' \twoheadrightarrow B'' & (transitivity) \end{array}$$

This relation can now be used to define an evaluator which takes an arbitrary expression B and relates it to an answer-expression A. Be aware that the following notation suggests that the evaluation relation is a function. This is in general not the case and requires some form of *confluence*[8, 11] of the reduction system - in case of doubt, it should be proven.

$$\begin{aligned} & eval : \mathcal{L}''_B \rightarrow \{0, 1\} \\ & eval(B) = A, \text{ if } B \twoheadrightarrow A \end{aligned}$$

An equally important question to *confluence* and *completeness* asks for the *correctness*[8, 11] of a reduction system. Does the reduction system reduce expressions only to their desired semantics? For example, a reduction system that reduces $0 \wedge 0$ to 1 should not be considered correct. Correctness must always be related to a normative reference system.

2.3 Functional Purity

ES intentionally serves various programming paradigms, including, but not limited, to procedural programming, object-oriented programming and functional programming. These informal classifications are not sharp. For example, ES's functions are used throughout all of these paradigms: They are used to glue procedural code together, to feature code-reuse. They also serve as the foundation of message-passing between objects and, thereby, contribute to object-orientation. Last but not least, functions can also be used in their mathematical sense. This means they represent pure mappings from arguments to results, without any observable side-effect. Overloading functions to serve all of these aspects definitely has its advantages, so the language and the cognitive load of a programmer can be kept small. On the downside, these paradigms are sometimes in conflict with each other. Side-effects, for instance, make it hard to argue about lazy code[14]. Knowing when a function is guaranteed to be pure, is therefore crucial for a programmer.

The notion of *purity* itself has been subject to many controversial discussions. Amr Sabry reviewed some of the most promising proposals to find a consensus on this topic in 1998[14]. To his own surprise, he found that none of the existing work could adequately capture the intuitive ideas behind a purely functional programming language. Examples of definitions, he ruled out, were based on *confluence*, *soundness of the β-axiom*, *preservation of observational equivalence*, *referential transparency* and *independence of order of evaluation*. This emphasizes the difficulty which the establishment of such a definition implies. He then proposed a new definition based on parameter-passing-independence.

There are some preparations to be met before the chapter can close with the formal definition of Amr Sabry. Parameter-passing-independence means that the language should be insensitive to whether parameters are evaluated before or after entering a functions body and to the count it actually becomes evaluated. There are many different strategies to evaluate parameters, three of which are interesting in this context: The *call-by-value* strategy evaluates parameters before the function's body is entered. All references to the argument then refer to the parameters value that was computed beforehand. This is ES's approach to handle parameters. In contrast, an evaluator that defers the evaluation of arguments to the point where they appear in the function's body and re-evaluates arguments each time they appear, is said to follow the *call-by-name* strategy. *Call-by-need* proceeds like *call-by-name*, but caches the result from the first evaluation and then re-uses the cached value in all subsequent places where the argument is needed. More precise definitions are given throughout the thesis when in demand. As a final preparation, an approximate notion of equivalence between evaluation functions is established:

Definition 4. Given a set of programs P and a set of observable answers A. Let $eval_1$ and $eval_2$ be two partial evaluation functions from p to B. Then $eval_1$ is *weakly equivalent* to $eval_2$, if the following conditions hold:

- If $eval_1(P) = A$, then either $eval_2(P) = A$ or $eval_2(B)$ is undefined
- If $eval_2(P) = A$, then either $eval_1(P) = A$ or $eval_1(A)$ is undefined

Definition 5. A language is *purely functional*, if

1. It is a conservative extension of the simply typed λ-calculus.
2. It has well-defined call-by-value, call-by-need and call-by-name evaluation functions.
3. All three evaluation functions are weakly equivalent.

The reader, who is unfamiliar with the λ-calculus, should work through the next chapter and then revisit the above definition to gain a better understanding of the first constraint.

3 Language Model

3.1 Lambda Calculus

3.1.1 Concrete Syntax

The ES standard specifies a concrete syntax which sets natural bounds to the model for the purely functional subset. Only features which can be expressed by the concrete syntax should be reflected by the language model. This restriction, for example, forbids the inclusion of type-annotations or custom value-constructors into the language. Conversely, if the model makes internal use of non-expressible constructs this information must not leak to the top of the language, in such a way that a programmer could exploit this information to obtain undesired semantics. On the other hand, clearly not all concrete syntactic constructs will have a counterpart in the abstract syntax of the purely functional subset. For example, there is no counterpart for re-assignments for obvious reasons. But even some features, which might turn out purely functional, aren't considered. This is to say, the development of a complete or maximal subset with respect to functional purity is far beyond the scope of the thesis.

ES's concrete syntax is given in two parts: The production-rules of a context-free grammar and the so-called *static semantics*. The *static semantics* impose some additional constraints in natural language over the subject language which cannot be reflected by the production rules. The context-free grammar is suitable to study the connections to the abstract syntax in a formal way - the production rules of ES are not contained with the thesis, because of their enormous size (approximate 21 pages) and must be consulted from [6]. The *static semantics* form a harder case, due to their lack of formalism. These aspects of the grammar are, therefor,e discussed in an as likely informal, yet precise way when needed.

The process to relate ES's concrete syntax to the abstract syntax is two-phase: First, a subset of the concrete syntax is constructed. Second, a parser is presented to map expressions from the concrete subset to expressions of abstract syntax.

To easier distinguish between non-terminals from both grammars, they become subscripted with either ES for original non-terminals or Λ for non-terminals of the desired subset. For conciseness, lexical details are deferred to the original syntax. To this end, the following convention applies: If a production rule from the subset grammar refers to a non-terminal of the original grammar, the non-terminal is implicitly imported together with all its production-rules.

Definition 6. The *concrete syntax of the lambda-calculus* Λ_c (also Λ_c^1) with respect to ES syntax is induced by the context-free grammar:

$$\begin{array}{lcl} Expression_\Lambda & ::= & IdentifierName_{ES} \\ & | & (IdentifierName_{ES} \texttt{=>} Expression_\Lambda) \\ & | & (Expression_\Lambda\ (Expression_\Lambda)) \end{array}$$

Theorem 3.1. *Λ_c is a subset of ES.*

To reduce the verbosity of some subsequent derivations, a few common derivations are introduced beforehand.

$Script_{ES} \vdash^* AssignmentExpression_{ES}$ by

$Script_{ES}$

$\vdash ScriptBodyopt_{ES}$

$\vdash StatementList_{ES}$

$\vdash StatementListItem_{ES}$

$\vdash Statement_{ES}$

$\vdash ExpressionStatement_{ES}$

$\vdash Expression_{ES}$

$\vdash AssignmentExpression_{ES}$

$AssignmentExpression_{ES} \vdash^* LeftHandSideExpression_{ES}$ by
$AssignmentExpression_{ES}$
$\vdash ConditionalExpression_{ES}$
$\vdash LogicalORExpression_{ES}$
$\vdash LogicalANDExpression_{ES}$
$\vdash BitwiseORExpression_{ES}$
$\vdash BitwiseXORExpression_{ES}$
$\vdash BitwiseANDExpression_{ES}$
$\vdash EqualityExpression_{ES}$
$\vdash RelationalExpression_{ES}$
$\vdash ShiftExpression_{ES}$
$\vdash AdditiveExpression_{ES}$
$\vdash MultiplicativeExpression_{ES}$
$\vdash UnaryExpression_{ES}$
$\vdash PostfixExpression_{ES}$
$\vdash LeftHandSideExpression_{ES}$

$AssignmentExpression_{ES} \vdash^* PrimaryExpression_{ES}$ by
$AssignmentExpression_{ES}$
$\vdash^* LeftHandSideExpression_{ES}$
$\vdash NewExpression_{ES}$
$\vdash MemberExpression_{ES}$
$\vdash PrimaryExpression_{ES}$

$PrimaryExpression_{ES} \vdash^* (AssignmentExpression_{ES})$ by
$PrimaryExpression_{ES}$
$\vdash CoverParenthesizedExpressionAndArrowParameterList_{ES}$
$\vdash (Expression_{ES})$
$\vdash (AssignmentExpression_{ES})$

Since the proof is very simple and verbose, it is included here only once for illustration and skipped for subsequent syntactical extensions.

Proof. It must be shown, that every word $e \in \mathcal{L}(expression_{\Lambda})$ is in $\mathcal{L}(Script_{ES})$. To proof the containment a derivation in ES's grammar rules for all words $e \in \mathcal{L}(expression_{\Lambda})$ is needed. This can be done by a structural induction over the shape of e. It is however not so trivial to apply the induction hypothesis to the start-symbol $Script_{ES}$. Therefore, the even stronger property is proven that every expression $e \in \mathcal{L}(expression_{\Lambda})$ can be derived from the production-rule $(PrimaryExpression_{ES})$. The original claim then follows from the fact that $Script_{ES} \vdash^{*} PrimaryExpression_{ES}$. By structural induction over the shape of e:

- $e = IdentifierName_{\Lambda}$:

$$\begin{aligned}
&PrimaryExpression_{ES}\\
&\vdash IdentifierReference_{ES}\\
&\vdash Identifier_{ES}\\
&\vdash IdentifierName_{ES}\\
&= IdentifiyerName_{\Delta}
\end{aligned}$$

- $e = (IdentifierName_{\Lambda} => e_1)$:

$$\begin{aligned}
&PrimaryExpression_{ES}\\
&\vdash^{*} (AssignmentExpression_{ES})\\
&\vdash (ArrowFunction_{ES})\\
&\vdash (ArrowParameters_{ES} => ConciseBody_{ES})\\
&\vdash (ArrowParameters_{ES} => AssignmentExpression_{ES})\\
&\vdash^{*} (ArrowParameters_{ES} => PrimaryExpression_{ES})\\
&\vdash^{*} (ArrowParameters_{ES} => e_1) && \text{by i.h.}\\
&\vdash (BindingIdentifier_{ES} => e_1)\\
&\vdash (Identifier_{ES} => e_1)\\
&\vdash (IdentifierName_{ES} => e_1)\\
&= (IdentifierName_{\Delta} => e_1)
\end{aligned}$$

- $e = (e_1\ (e_2))$:

$$\begin{aligned}
&PrimaryExpression_{ES}\\
&\vdash^* (AssignmentExpression_{ES})\\
&\vdash^* (LeftHandSideExpression_{ES})\\
&\vdash (CallExpression_{ES})\\
&\vdash (MemberExpression_{ES}\ Arguments_{ES})\\
&\vdash (PrimaryExpression_{ES}\ Arguments_{ES})\\
&\vdash^* (e_1\ Arguments_{ES}) && \text{by i.h.}\\
&\vdash (e_1\ (ArgumentList_{ES}))\\
&\vdash (e_1\ (AssignmentExpression_{ES}))\\
&\vdash^* (e_1\ (PrimaryExpression_{ES}))\\
&\vdash^* (e_1\ (e_2)) && \text{by i.h.}
\end{aligned}$$

□

3.1.2 Abstract Syntax

The definition of functional purity obligates the subject language to contain the simply typed λ-calculus[14]. The calculus itself, without extensions, must be considered the minimal programming language to obey this property. This notion suggests a starting point for the construction of a more full-fledged language. In contrast to the language of boolean algebra from the preliminaries, the λ-calculus serves as a general computation model. Its abstract syntax is made of three terminals, namely *variables*, *abstraction* and *application*[8].

Definition 7. The abstract syntax of the lambda calculus Λ (also Λ^1) is:

$$\begin{array}{rrl}
\text{Variables} & X, Y, Z & ::= x, y, z, ... \in \mathcal{V}, \ \text{where } \mathcal{V} \text{ is an infinite set}\\
\text{Terms} & M, N, L & ::= X \mid @ \mid \lambda\\
 & @ & ::= M\ N\\
 & \lambda & ::= X\ M
\end{array}$$

For readability the production rules are not always written down in this accurate manner and are usually equipped with new syntactical sugar. To this end, the above reduction rules may be written as:

$$\begin{array}{rl} \text{Variables} & X, Y, Z ::= x, y, z, ... \in \mathcal{V}, \text{ where } \mathcal{V} \text{ is an infinite set} \\ \text{Terms} & M, N, L ::= X \mid (\lambda X.M) \mid (M\ N) \end{array}$$

Definition 8. The parser from concrete words to abstract terms is inductively defined as:

$$\begin{array}{lll} parse(IdentifierName_\Lambda) & \overset{\text{def}}{=} & X_{IdentifierName} \\ parse((IdentifierName_\Lambda \mathrel{=>} e_1)) & \overset{\text{def}}{=} & \lambda\ parse(IdentifierName_\Lambda).parse(e_1) \\ parse(((e_1\ (e_2))) & \overset{\text{def}}{=} & (parse(e_1)\ parse(e_2)) \end{array}$$

Non-terminals M, N, L denote aliases for λ-terms. Non-terminals X, Y, Z represent names for variables, while x, y, z represent the actual variables - for the remainder of the thesis this fine-grained difference in the terminology is not always kept up. Terminals of the form $(\lambda X.M)$ express abstraction of the term M over the variable X, which may or may not appear in M. Intuitively, X (sometimes called the head) holds the formal argument to an anonymous function with body M. $(M\ N)$ is the application of the actual argument N to M.

Examples of λ-terms are:

1. x every variable is a λ-term
2. $(x\ y)$ application of y to the the variable x
3. $(\lambda x.y)$ abstraction of the term y over the variable x
4. $(\lambda x.(\lambda y.(\lambda z.((x\ y)\ z))))$ a compound term

The generous use of parenthesis may sometimes impact the readability of long expressions it is, therefore, conventional to define rules to supply a shorthand syntax on a meta-language level. When leaving out parenthesis it must be ensured that expressions cannot be read in ambiguous ways, this can be accomplished by the adoption of rules for precedence and associativity. For the remainder of the thesis three rules apply:

1. *application* binds stronger than *abstraction*
2. *abstraction* associates to the right
3. *application* associates to the left

For instance, example 4 may now be written as $\lambda x.\lambda y.\lambda z.x\ y\ z$.

The perceived variables suggest a concept where variables act as placeholders for arbitrary terms which may be substituted in their place. However, this is not the case for variables which act as formal arguments of abstractions. Those variable appearances are *bound*[8]. The dual designation are $free$ variables.

Definition 9. The set of free variables $\mathcal{FV}$ is:

$$\begin{aligned}
\mathcal{FV}(X) &\stackrel{\text{def}}{=} \{X\} \\
\mathcal{FV}(M\ N) &\stackrel{\text{def}}{=} \mathcal{FV}(M) \cup \mathcal{FV}(N) \\
\mathcal{FV}(\lambda X.M) &\stackrel{\text{def}}{=} \mathcal{FV}(M) \setminus \{X\}
\end{aligned}$$

Equipped with the idea of *free* and *bound* variables, it is now possible to define capture-free substitution. *Capture-free*, in this context, means free variables inside the replacement-expression remain free after substitution[8]. This can be ensured by renaming formal arguments to fresh variables. It follows a formal definition over the structure of λ-expressions.

Definition 10. The *capture-free substitution* of a variable X in a term N by M, written $N[X \leftarrow M]$, is inductively defined over the shape of N:

$$\begin{aligned}
X[X \leftarrow M] &\stackrel{\text{def}}{=} M \\
Y[X \leftarrow M] &\stackrel{\text{def}}{=} Y,\ \text{if } X \neq Y \\
(NL)[X \leftarrow M] &\stackrel{\text{def}}{=} (N[X \leftarrow M]\ L[X \leftarrow M]) \\
(\lambda X.N)[X \leftarrow M] &\stackrel{\text{def}}{=} (\lambda X.N) \\
(\lambda Y.N)[X \leftarrow M] &\stackrel{\text{def}}{=} (\lambda Z.N[Y \leftarrow Z][X \leftarrow M]), \\
&\quad \text{if } X \neq Y \\
&\qquad Z \notin \mathcal{FV}((\lambda Y.N)) \\
&\qquad Z \notin \mathcal{FV}(M)
\end{aligned}$$

3.1.3 Notions of reduction

Semantics for the λ-calculus are defined as partial functions on abstract syntax trees which are commonly referred to as evaluators. Operational semantics model the mechanics of producing a final answer from expressions. This process may involve several and, possibly, infinite many steps. Each step must be justified by rules which are recognized as notions of reduction or reduction axioms. Simply put, axioms are binary relations on abstract syntax trees. The lambda calculus is conventionally presented with three of these rules, namely α, β and η. However, since the thesis doesn't study the λ-calculus on its own sake, but rather exploits it only as a modelling technique for ES functions, α and η are not included here. The reason for this is that the evaluator which is going to be developed, is only interested in certain kinds of expressions, namely *programs*. Programs are expressions without free variable-appearances[8].

$$((\lambda X.M)\ N)\ \beta\ M[X \leftarrow N]$$

Figure 1: β-axiom of the λ-Calculus

β models the mechanics of applying actual arguments to abstractions. It can be read as follows: If an expression of the form $((\lambda X.M)\ N)$ is encountered, it is then possible to reduce it to an expression of the form $M[X \leftarrow N]$. For example $(\lambda x.x)y$ can be reduced to y.

To be able to reduce sub-expressions in arbitrary positions of complex expressions, this relation can naturally be extended to its compatible closure. This has been seen before in the preliminaries section, although it was not called *compatible closure* there.

$$\begin{array}{rl}
M \xrightarrow{\beta} N, & \text{if } M\ \beta\ N \\
(M\ N) \xrightarrow{\beta} (M'\ N), & \text{if } M \xrightarrow{\beta} M' \\
(M\ N) \xrightarrow{\beta} (M\ N'), & \text{if } N \xrightarrow{\beta} N' \\
(\lambda X.M) \xrightarrow{\beta} (\lambda X.M'), & \text{if } M \xrightarrow{\beta} M'
\end{array}$$

Figure 2: Compatible closure of β

$\xrightarrow{\beta}$ defines an atomic rule for what can be done in single reduction steps. Again, this relation can be extended further to a multi-step reduction by looking at its reflexive and transitive closure.

$$M \overset{\beta}{\twoheadrightarrow} M$$

$$, M \overset{\beta}{\twoheadrightarrow} N, \text{ if } M \xrightarrow{\beta} N$$

$$M \overset{\beta}{\twoheadrightarrow} N, \text{ if } M \overset{\beta}{\twoheadrightarrow} L \text{ and } L \overset{\beta}{\twoheadrightarrow} N \text{ for some } L$$

Figure 3: Reflexive, transitive closure of $\xrightarrow{\beta}$

Subsequent uses of the double headed arrow will always denote the reflexive, transitive closure for the underlying small-step reduction.

One question that arises from this definition is, how deep the order of applying β to different sub-expressions impacts the outcome of the program. It is easily found that the produced expressions diverges even for simple expressions. The following example illustrates this issue. The underlined sub-expression emphasizes the redex which is going to be reduced:

$$(\underline{((\lambda y.y)\ x)}\ ((\lambda x.x)\ y)) \xrightarrow{\beta} (x\ ((\lambda x.x)\ y)) \tag{11}$$

$$(((\lambda y.y)\ x)\ \underline{((\lambda x.x)\ y)}) \xrightarrow{\beta} (((\lambda y.y)\ x)\ y) \tag{12}$$

Despite the divergence of the expressions, it is an interesting property that the expressions will commute, when β-reduction is applied a second time:

$$(x\ \underline{((\lambda x.x)\ y)}) \xrightarrow{\beta} (x\ y) \tag{13}$$

$$(\underline{((\lambda y.y)\ x)}\ y) \xrightarrow{\beta} (x\ y) \tag{14}$$

This question can be generalized to an important property of programming languages: For two expressions of the domain language which are both derived from a common successor expression, are there reduction-sequences, such that both branches commute? This property is known as *confluence* and justifies the perception of evaluators as (partial) functions[8]. The property is known to hold for the λ-calculus[8].

It is finally possible to define an evaluator for lambda programs:

$$eval(M) = closure,\ \text{if } M \overset{\beta}{\twoheadrightarrow} (\lambda.X.N) \text{ for some } N$$

3.1.4 An abstract syntax machine

Modelling semantics based only on syntax as binary relations is mathematically elegant but also verbose. Abstract syntax machines offer a more terse modelling technique. Starting with an indeterministic machine which will allow the reduction of redexes in any place and order, it is then the point of interest to factor out reduction-strategies.

Such a machine requires the decomposition of a particular redex and its surrounding context. A context C is an expression with a hole $[\,]$ in it. The hole can be filled with arbitrary expressions M, which is written as $C[M]$. Unlike substitution, filling a context may capture free variables. For instance the context $(\lambda x.[\,])$ filled with the variable x gives $(\lambda x.x)$. The first machine is going to allow reduction in any place and order, this makes the construction simple:

Syntactic Domains

Variables	x, y, z
Values	$V ::= (\lambda x.M)$
Terms	$L, M, N ::= x \mid (\lambda x.M) \mid (M\ N)$
Contexts	$C ::= [\,] \mid (\lambda x.C) \mid (C\ M) \mid (M\ C)$

Reduction Axioms

(β)	$C[(\lambda x.M)\ N] \to C[M[x \leftarrow N]]$

Figure 4: Indeterminstic ASM for λ-caclulus: λ_i

This yields an alternative approach for an evaluator:

$$eval_i(M) = closure,\ \text{if } \lambda_i \vdash M \to V$$

At this early stage the machine may pick redexes intentionally arbitrary. This behaviour is at best illustrated with an reduction-graph, consider the λ-term $(\lambda x.x\ x)\ ((\lambda y.y)\ (\lambda z.z))$:

$$\begin{tikzcd}
 & (\lambda x.x\ x)\ ((\lambda y.y)\ \lambda z.z) \arrow[d, dashed] \arrow[dr] & \\
 & ((\lambda y.y)\ \lambda z.z)\ ((\lambda y.y)\ \lambda z.z) \arrow[dl, dashed] \arrow[d, dotted] & (\lambda x.x\ x)\ \lambda z.z \arrow[ddl] \\
(\lambda z.z)\ ((\lambda y.y)\ \lambda z.z) \arrow[d, dashed] \arrow[dr, dotted] & ((\lambda y.y)\ \lambda z.z)\ \lambda z.z \arrow[d, dotted] & \\
(\lambda y.y)\ \lambda z.z \arrow[dr, dashed] & (\lambda z.z)\ \lambda z.z \arrow[d] & \\
 & \lambda z.z &
\end{tikzcd}$$

The solid path chooses the context $(\lambda x.x\ x)\ [\ ]$ and redex $(\lambda y.y)\ \lambda z.z$, respectively for the first reduction step. In other words, it chooses to reduce the argument to $\lambda x.x\ x$ before the evaluation of the outermost application. In the next step the expression can only be decomposed into the empty context $[\ ]$ and redex $(\lambda x.x\ x)\ \lambda z.z$ which will contract to $(\lambda z.z)\ \lambda z.z$. Again, this expression can only be decomposed with the empty context. Applying β-reduction a third time produces the final value $\lambda z.z$.

The dashed path first picks the empty context $[\ ]$ together with the redex $((\lambda x.x\ x)\ ((\lambda y.y)\ (\lambda z.z)))$ and reduces the outermost application. This will duplicate the actual argument $((\lambda y.y)\ \lambda z.z)$. It then chooses to reduce the left hand side of the application before the right hand side. Once again, this step yields two forms of contexts to choose from: $[\ ]$ or $(M\ [\ ])$. Choosing $[\ ]$ produces $((\lambda z.z)\ (\lambda z.z))$ after reduction, choosing $((\lambda z.z)\ [\,])$ as the context produces $((\lambda y.y)\ (\lambda z.z))$. Both expressions can then be processed analogously to the last step of the former case. When reducing the right-hand side of the duplicated argument from the first step before the left-hand side, the machine produces $((\lambda z.z)\ (\lambda z.z))$ (dotted path) as an intermediate result. This case then proceeds as in earlier cases. The next goal is to factor out reduction strategies. Subsequent machines will be designed to always choose a particular path without ambiguity.

3.1.5 Call-by-value

The call-by-value evaluator computes arguments before they're substituted inside the function body. This corresponds to the solid path of the previous example. There is still some choice for the evaluator, consider, for instance, the expression $((\lambda x.x)\ y)\ ((\lambda x.x)\ y))$: an evaluator may choose to reduce either side of the outermost application before the other (parallel evaluation is also an option, but not considered here). When such a situation is encountered, the machine will always choose the left-most expression first. This approach forms the eager-reduction strategy, which happens to be the ES strategy as well.

Evaluating arguments before the abstraction body enforces a formal notion of an argument being reduced just enough to become substituted. One way to accomplish this is to look at the shape of a reduced argument: If it is a variable or an abstraction it is in a valid form to be substituted - these kind expressions are commonly referred to as values[14] (hence the name call-by-value). Applications, for contrast, might be reduced further. Hence, they are no candidates for substitution with respect to the call-by-value parameter-passing-strategy. Now that all ambiguities are resolved, it is possible to define an abstract machine for eager-reduction.

Syntactic domains

Variables	x, y, z
Terms	$L, M, N ::= x \mid (\lambda x.M) \mid (M\ N)$
Values	$V ::= \lambda x.M$
Evaluation Contexts	$E ::= [\,] \mid (E\ M) \mid (V\ E)$

Reduction Axioms

(β)	$E[((\lambda x.M)\ V)] \to E[M[x \leftarrow V]]$

Figure 5: Call-by-value λ-calculus: λ^1_{value}

$$eval_v(M) = closure, \text{ if } \lambda^1_{value} \vdash M \to V$$

3.1.6 Call-by-name

Evaluating arguments before their application to a function is cheap in terms of administrative overhead and performance. On the other hand, there are situations where by-value calls do not terminate, even if an argument is never used inside a function. Consider the expression $((\lambda x.x\ x)\ (\lambda x.x\ x)$ (sometimes referred to as Ω): reducing it yields the same expression every time. Now consider the function $(\lambda x.\lambda y.x)$ which consumes two arguments, ignores the second one and returns the first one as its result. When charging a call-by-value abstract machine with the evaluation of $((\lambda x.\lambda y.x)\ \Omega)$ it will not terminate. The call-by-name parameter-passing-strategy in contrast offers the possibility to reduce $((\lambda x.\lambda y.x)\ \Omega)$ to z with a finite sequence of reduction steps. It does so by reducing arguments after their substitution inside the functions body. This corresponds to the dashed path of the indeterministic machine. The following machine is borrowed from [14] with slight modifications to be consistent with the other machines in the thesis.

Syntactic Domains

Variables	x, y, z
Values	$V ::= (\lambda x.M)$
Terms	$L, M, N ::= x \mid (\lambda x.M) \mid (M\ N)$
Evaluation Contexts	$E ::= [\,] \mid (E\ M)$

Reduction Axioms

(β)	$E[((\lambda x.M)\ N)] \to E[M[x \leftarrow N]]$

Figure 6: Call-by-name λ-calculus: λ^1_{name}

$$eval_n(M) = closure,\ \text{if } \lambda^1_{name} \vdash M \to V$$

3.1.7 Call-by-need

One significant drawback of by-name-parameter-passing compared to by-value-calls, is that arguments may be duplicated inside the function's body and become evaluated more than once. This behaviour can be witnessed in the running example of the indeterministic machine: The first reduction step on the left path duplicates the actual argument $((\lambda y.y)\ (\lambda z.z))$. Figure 3.1.7 reconsiders the problematic evaluation step with an alternative representation, chosen to emphasize the structure of the program.

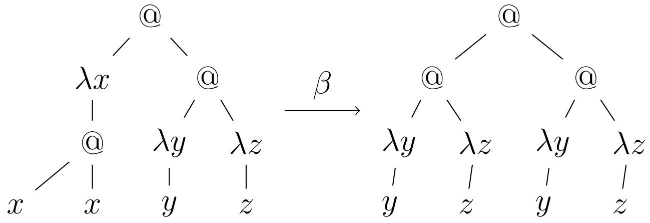


Figure 7: Argument duplication of λ_n

Call-by-need is a variation of call-by-name that unifies the best of both worlds: It conserves the better termination properties of call-by-name, but evaluates parameters at most once similarly to call-by-value. These benefits are traded in for administrative overhead: While the previous machines remained conceptually similar to each other (besides evaluation contexts all syntactic domains were left untouched and even the β-reduction axiom reappeared only with minor changes) the anticipation of the call-by-need strategy demands more complex changes.

The call-by-need λ-calculus λ_{need}[5] from Felleisen and Ariola never actually dispatches function calls. This is to say, it gets along without capture-free-substitution. The key insight is to identify closures, before they become reduced. For instance, consider the program $(\lambda x.\lambda y.y)\ V$: Reducing it with β yields a value which corresponds to an answer in λ_v and λ_n. λ_{need} instead establishes the following recursive definition of answers:

$$\text{Answers } A ::= \lambda x.M \mid (\lambda x.A)\ M$$

λ_{need} transforms programs into a left-linear form, so that answers may be witnessed by a simple shape-analysis of the program: If the program is an abstraction, it is an answer. Otherwise, if the program is an application with an abstraction in the function position, the evaluator may descendent recursively into the functions body and repeat the answer-test.

Syntactic Domains

$$\begin{array}{ll}
\text{Variables} & x, y, z \\
\text{Terms} & L, M, N ::= x \mid (\lambda x.M) \mid (M\ N) \\
\text{Values} & V ::= \lambda x.M \\
\text{Answers} & A ::= V \mid ((\lambda x.A)\ M) \\
\text{Evaluation Contexts} & E ::= [\,] \mid (E\ M) \mid ((\lambda x.E)\ M) \mid ((\lambda x.E[x])\ E)
\end{array}$$

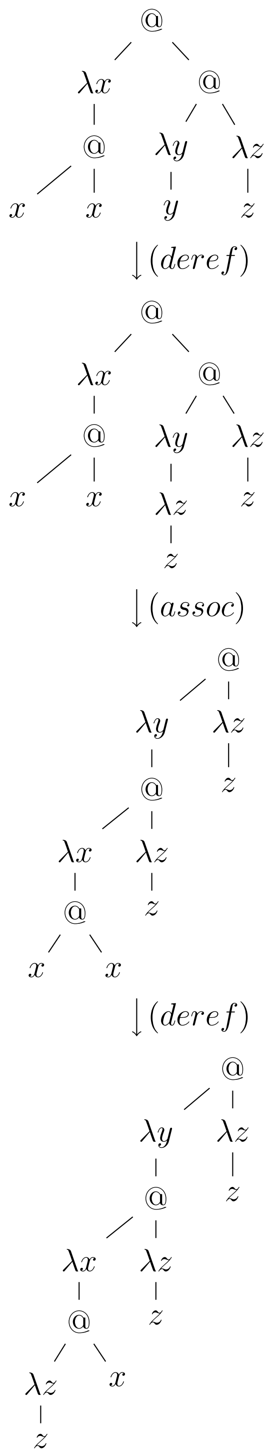
Reduction Axioms

$$\begin{array}{lrcl}
\text{(deref)} & E_1[(\lambda x.E_2[x]\ V)] & \to & E_1[(\lambda x.E_2[V])\ V)] \\
\text{(lift)} & E[(((\lambda x.A)\ M)\ N)] & \to & E[(\lambda x.A\ N)\ M)] \\
\text{(assoc)} & E_1[((\lambda x.E_2[x])\ ((\lambda y.A)\ M))] & \to & E_1[((\lambda y.(\lambda x.E_2[x])\ A)\ M)]
\end{array}$$

Figure 8: Call-by-need λ-calculus: λ^1_{need}

$$eval_{need}(M) = closure, \text{ if } \lambda^1_{need} \vdash M \to A$$

The reduction-sequence that this machine would take on the running example $((\lambda x.x\ x)\ ((\lambda y.y)\ (\lambda z.z))$ is not covered by the indeterministic machine λ_i. The following example demonstrates this evaluation and thereby explains the machine's implementation details.



The program is obviously not an answer yet. The next evaluation step demands the argument of the outermost application. This demand is reflected through the evaluation context $(\lambda x.E[x])\ E$ which matches here. Because the argument is not yet an answer itself, it needs to be reduced with $(deref)$ first.

The actual argument of the outermost application is now an answer. Yet, it is not possible to dereference it inside the function's body. A rearrangement, according to $(assoc)$, is necessary first to respect sharing.

All preliminaries are now met to finally dereference x.

The program has now the shape of an answer. This expression also serves as a good example for the left-linearity of answer-expressions.

3.2 Basic Constructors and Constants

3.2.1 Syntax

The ES specification introduces basic data-constructors and constants for common programming tasks, like string manipulation, arithmetic, boolean algebra and list processing[6]. The sophisticated study[11, 8] of these language-features renders it simple to include many primitive values and corresponding operations into a language. For the conciseness of the thesis, however, support will be added only for floating point arithmetic and boolean algebra. String manipulation could be added analogously, and list processing will be revisited in a subsequent chapter about compound values. This chapter introduces new syntactic constructs and adjusts some of the definitions from the previous chapter accordingly. The language of this chapter will be referred to as Λ^2 and belongs to the ISWIM (If You See What I Mean) family of programming languages.

Definition 11. The concrete syntax Λ^2_c of this section is given by:

$$\begin{aligned}
Expression_{\Lambda} \quad & ::= \ldots \mid UnaryExpression_{\Lambda} \mid InfixExpression_{\Lambda} \\
& \mid BooleanLiteral_{ES} \mid DecimalLiteral_{ES} \\
UnaryExpression_{\Lambda} \quad & ::= (UnaryOperator_{\Lambda}\ Expression_{\Lambda}) \\
InfixExpression_{\Lambda} \quad & ::= (Expression_{\Lambda}\ InfixOperator_{\Lambda}\ Expression_{\Lambda}) \\
InfixOperator_{\Lambda} \quad & ::= + \mid - \mid * \mid / \mid \&\& \mid ||
\end{aligned}$$

Definition 12. The abstract syntax of Λ^2 is:

$$
\begin{array}{rrl}
\text{Terms} & M, N, L & ::= X \\
 & & \mid\ (\lambda X.M) \\
 & & \mid\ (M\ N) \\
 & & \mid\ b \\
 & & \mid\ (o^n\ M_1 \ldots M_n) \\
\text{Variables} & & x, y, z \\
\text{Signature} & \Sigma & ::= \texttt{True} \mid \texttt{False} \mid \texttt{NaN} \mid \texttt{not} \mid \texttt{uminus} \\
 & & \mid \texttt{plus} \mid \texttt{bminus} \mid \texttt{mult} \mid \texttt{div} \mid \texttt{and} \mid \texttt{or} \mid \texttt{equals} \\
\text{Constructors} & c^0 & ::= \mathit{num} \mid \mathit{bool} \\
\text{Numerals} & \mathit{num} & ::= n \mid \texttt{NaN} \\
 & & \text{for all } n \in \mathbb{FL} \\
\text{Booleans} & \mathit{bool} & ::= \texttt{True} \mid \texttt{False} \\
\text{Operators} & o^1 & ::= \texttt{not} \mid \texttt{uminus} \\
 & o^2 & ::= \texttt{plus} \mid \texttt{bminus} \mid \texttt{mult} \mid \texttt{div} \mid \texttt{and} \mid \texttt{or} \mid \texttt{equals}
\end{array}
$$

Definition 13. The parser $\mathit{parse}^2 : \Lambda^2_c \rightarrow \Lambda^2$ is the smallest closure under the following operations:

`true`	→	`True`
`false`	→	`False`
`NaN`	→	`NaN`
$!e$	→	$(\texttt{not}\ e)$
$-e$	→	$(\texttt{uminus}\ e)$
$(e_1{+}e_2)$	→	$(\texttt{plus}\ e_1\ e_2)$
$(e_1{-}e_2)$	→	$(\texttt{bminus}\ e_1\ e_2)$
$(e_1{*}e_2)$	→	$(\texttt{mult}\ e_1\ e_2)$
(e_1/e_2)	→	$(\texttt{div}\ e_1\ e_2)$
$(e_1\&\&e_2)$	→	$(\texttt{and}\ e_1\ e_2)$
$(e_1\|\|e_2)$	→	$(\texttt{or}\ e_1\ e_2)$
$(e_1{=}{=}{=}e_2)$	→	$(\texttt{equals}\ e_1\ e_2)$

The syntactic domain *num* holds numerals to represent all double precision floating point numbers $\mathbb{FL}$, as defined by IEEE 754-2008[1]. `NaN` is short for *Not a Number* and serves as a representation for undefined operations, as for example division by zero. The members of *bool* encode logical truth and falsehood. o^1 and o^2 comprise unary and binary operations respectively. Constructor terminals conventionally start with a capital letter in contrast to primitive operations.

Definition 14. The set of free variable $\mathcal{FV}$ of Λ^2 is:

$$\begin{aligned}
\mathcal{FV}(X) &\overset{\text{def}}{=} \{X\} \\
\mathcal{FV}(M\ N) &\overset{\text{def}}{=} \mathcal{FV}(M) \cup \mathcal{FV}(N) \\
\mathcal{FV}(\lambda X.M) &\overset{\text{def}}{=} \mathcal{FV}(M) \setminus \{X\} \\
\mathcal{FV}(c^0) &\overset{\text{def}}{=} \emptyset \\
\mathcal{FV}(o^n M_1 \ldots M_n) &\overset{\text{def}}{=} \mathcal{FV}(M_1) \cup \ldots \cup \mathcal{FV}(M_n)
\end{aligned}$$

The set of free variables of a basic constant is trivially empty. Operator application may trap variables under each of its actual arguments whose count may vary depending on the operators arity.

Definition 15. Capture-free substitution for Λ^2 is defined as:

$$\begin{aligned}
X[X \leftarrow M] &\overset{\text{def}}{=} M \\
Y[X \leftarrow M] &\overset{\text{def}}{=} Y, \text{ if } X \neq Y \\
(NL)[X \leftarrow M] &\overset{\text{def}}{=} (N[X \leftarrow M]\ L[X \leftarrow M]) \\
(\lambda X.N)[X \leftarrow M] &\overset{\text{def}}{=} (\lambda X.N) \\
(\lambda Y.N)[X \leftarrow M] &\overset{\text{def}}{=} (\lambda Z.N[Y \leftarrow Z][X \leftarrow M]), \\
&\quad \text{if } X \neq Y \\
&\quad\quad Z \notin \mathcal{FV}((\lambda Y.N)) \\
&\quad\quad Z \notin \mathcal{FV}(M) \\
c^0[X \leftarrow M] &\overset{\text{def}}{=} b \\
(o^n M_1 \ldots M_n)[X \leftarrow M] &\overset{\text{def}}{=} (o^n M_1[X \leftarrow M] \ldots M_n[X \leftarrow M])
\end{aligned}$$

Capture free substitution expands straight forward: Since constants do not trap variables, substitution does not involve any extra work. Operator application may contain free variables under its arguments, as seen earlier, but does not bind any variables itself. Substitution therefore descendants recurively into the operators parameters.

3.2.2 Call-by-value

In contrast to application of abstraction, application of operators does not allow arbitrary terms in the function position. As a decent consequence, evaluation can never descend in the function position of operator application.

The extended language comprises additional syntactic domains for numbers and boolean values. Operators usually work on a subset of these domains. A naive implementation can therefore lead to stuck states when an argument of unexpected domain is applied to an operator. ES instead defines a mechanism to automatically convert between values of different domains to match the appropriate domain of the operator. This feature is commonly known as type coercing, the following definition mimics the rules of type coercing for the intrinsic subset of the thesis:

$$
\begin{aligned}
&ToNumber(num) && \overset{\text{def}}{=} num \\
&ToNumber(bool) && \overset{\text{def}}{=} \begin{cases} 1 & \text{if } bool = \texttt{True} \\ +0 & \text{if } bool = \texttt{False} \end{cases} \\
&ToNumber(\lambda x.M) && \overset{\text{def}}{=} \texttt{NaN} \\
\\
&ToBoolean(num) && \overset{\text{def}}{=} \begin{cases} \texttt{False} & \text{if } num \in \{+0, -0, \texttt{NaN}\} \\ \texttt{True} & \text{otherwise} \end{cases} \\
&ToBoolean(bool) && \overset{\text{def}}{=} bool \\
&ToBoolean(\lambda x.M) && \overset{\text{def}}{=} \texttt{True}
\end{aligned}
$$

Figure 9: Type coercing rules for Λ^2

Strict evaluation enforces arguments to be evaluated to values, regardless of the kind of application. This draws a discrepancy to ES, which specifies strict evaluation only for function application, but not for operator application. For example the term ($\texttt{or true}\ \Omega$) does not terminate according to call-by-value semantics, but reduces to `true` according to ES. The very similar looking expression ($\texttt{or}\ \Omega\ \texttt{true}$), however, still does not terminate due to leftmost evaluation. The small step semantics of primitive operations are defined by the (partial) function δ.

$$\delta(\mathtt{uminus}, V) \overset{\text{def}}{=} \begin{cases} -\mathit{ToNumber}(V) & \text{if } V \neq \mathtt{NaN} \\ \mathtt{NaN} & \mathit{otherwise} \end{cases}$$

$$\delta(\mathtt{bminus}, V, W) \overset{\text{def}}{=} \begin{cases} \mathit{ToNumber}(V) - \mathit{ToNumber}(W) & \text{if } V, W \neq \mathtt{NaN} \\ \mathtt{NaN} & \mathit{otherwise} \end{cases}$$

$$\delta(\mathtt{plus}, V, W) \overset{\text{def}}{=} \begin{cases} \mathit{ToNumber}(V) + \mathit{ToNumber}(W) & \text{if } V, W \neq \mathtt{NaN} \\ \mathtt{NaN} & \mathit{otherwise} \end{cases}$$

$$\delta(\mathtt{mult}, V, W) \overset{\text{def}}{=} \begin{cases} \mathit{ToNumber}(V) \cdot \mathit{ToNumber}(W) & \text{if } V, W \neq \mathtt{NaN} \\ \mathtt{NaN} & \mathit{otherwise} \end{cases}$$

$$\delta(\mathtt{div}, V, W) \overset{\text{def}}{=} \begin{cases} \mathit{ToNumber}(V) / \mathit{ToNumber}(W) & \begin{array}{l} \text{if } V, W \neq \mathtt{NaN}, \\ W \notin \{+0, +0\} \end{array} \\ \mathtt{NaN} & \mathit{otherwise} \end{cases}$$

$$\delta(\mathtt{not}, V) \overset{\text{def}}{=} \neg \mathit{ToBoolean}(V)$$

$$\delta(\mathtt{and}, V, W) \overset{\text{def}}{=} \mathit{ToBoolean}(V) \wedge \mathit{ToBoolean}(W)$$

$$\delta(\mathtt{or}, V, W) \overset{\text{def}}{=} \mathit{ToBoolean}(V) \vee \mathit{ToBoolean}(W)$$

$$\delta(\mathtt{equals}, V, W) \overset{\text{def}}{=} \begin{cases} \mathtt{False} & \text{if } V = \mathtt{NaN} \text{ or } W = \mathtt{NaN} \\ \mathtt{True} & \text{if } V, W \neq \mathtt{NaN} \text{ and } V = W \\ \mathtt{True} & \text{if } V, W \in \{+0, -0\} \end{cases}$$

This definition lacks some precision in terms of differentiation between syntactical and semantic objects. For clarification the result of applying `not` to a value V can be received as follows: Coerce V to its boolean representation. Let V_1 be a label for the intermediate result from this step. Next, apply natural logical negation to the semantic value that is encoded by V_1 and label the result with V_2. Finally, let the result of $\delta(\mathtt{not}, V)$ be the syntactical representation of the semantic value V_2.

Equipped with δ, it is now possible to adopt the semantics for constructors and primitive operators to the call-by-value machine. To allow the reduction of arguments of operator application, the set of evaluation contexts is enriched with $(o^n\ V_1 \dots E \dots M_n)$. This context guarantees that arguments are always evaluated from left to right. The δ-axiom models the dispatching of operator application. This application can only be dispatched, if all arguments are values.

Syntactic domains

Values	$V ::= x \mid \lambda x.M \mid c^0$
Evaluation Contexts	$E ::= [\,] \mid (E\ M) \mid (V\ E) \mid (o^n\ V_1 \ldots E \ldots M_n)$

Reduction Axioms

$$\begin{array}{ll} (\beta) & E[((\lambda x.M)\ V)] \rightarrow E[M[x \leftarrow V]] \\ (\delta) & E[o^n\ V_1 \ldots V_n] \rightarrow E[\delta(o^n, V_1, \ldots, V_n)] \end{array}$$

Figure 10: Call-by-value calculus for Λ^2: λ^2_{value}

$$eval_v(M) = \begin{cases} closure, & \text{if } \lambda^2_{value} \vdash M \rightarrow \lambda x.N \\ c^0, & \text{if } \lambda^2_{value} \vdash M \rightarrow c^0 \end{cases}$$

3.2.3 Call-by-name

Syntactic Domains

Values	$V ::= (\lambda x.M) \mid c^0$
Evaluation Contexts	$E ::= [\,] \mid (E\ M) \mid (o^n\ V_1 \ldots E \ldots M_n)$

Reduction Axioms

$$\begin{array}{ll} (\beta) & E[((\lambda x.M)\ N)] \rightarrow E[M[x \leftarrow N]] \\ (\delta) & E[o^n\ V_1 \ldots V_n] \rightarrow E[\delta(o^n, V_1, \ldots, V_n)] \end{array}$$

Figure 11: Call-by-name calculus for Λ^2: λ^2_{name}

$$eval_n(M) = \begin{cases} closure, & \text{if } \lambda^2_{name} \quad \vdash M \rightarrow \lambda x.N \\ c^0, & \text{if } \lambda^2_{name} \quad \vdash M \rightarrow c^0 \end{cases}$$

3.2.4 Call-by-need

Syntactic Domains

$$\begin{array}{ll} \text{Values} & V ::= \lambda x.M \mid c^0 \\ \text{Answers} & A ::= V \mid ((\lambda x.A)\ M) \\ \text{Evaluation Contexts} & E ::= [\ ] \mid (E\ M) \mid ((\lambda x.E)\ M) \\ & \quad \mid \quad ((\lambda x.E[x])\ E) \mid (o^n\ V_1 \ldots E \ldots M_n) \end{array}$$

Reduction Axioms

$$\begin{array}{lrl} (\text{deref}) & E_1[(\lambda x.E_2[x]\ V)] & \rightarrow E_1[(\lambda x.E_2[V])\ V)] \\ (\text{lift}) & E[(((\lambda x.A)\ M)\ N)] & \rightarrow E[(\lambda x.A\ N)\ M)] \\ (\text{assoc}) & E_1[((\lambda x.E_2[x])\ ((\lambda y.A)\ M))] & \rightarrow E_1[((\lambda y.(\lambda x.E_2[x])\ A)\ M)] \\ (\delta) & E[o^n\ V_1 \ldots V_n] & \rightarrow E[\delta(o^n, V_1, \ldots, V_n)] \end{array}$$

Figure 12: Call-by-need calculus for Λ^2: λ^2_{need}

$$eval_{need}(M) = \begin{cases} closure, & \text{if } \lambda^2_{need} \quad \vdash M \rightarrow \lambda x.N \\ c^0, & \text{if } \lambda^2_{need} \quad \vdash M \rightarrow c^0 \\ eval_{need}(A), & \text{if } \lambda^2_{need} \quad \vdash M \rightarrow (\lambda x.A)\ M \end{cases}$$

3.3 Compound datatypes

3.3.1 Syntax

ES offers concise literal notations for compound datatypes[6]. Two of which, *object literals* and *arrays*, gained mentionable popularity across industry even outside the ES-ecosystem, when Douglas Crockford utilized them for the JSON-data-exchange-format. Roughly speaking, object-literals encode key-value-storages, where the values may vary in type (even within the same storage), but keys are limited to strings. From a semantical point of view, ES's arrays are really just a special form of key-value storages, where the keys also happen to represent string-encoded natural numbers. ES natively comprises much more advanced concepts, as for example computed property-names or symbolic property-keys. These are not covered here. Similarly, abstract datatypes for *maps* and *sets* are not considered. The purpose of the thesis is not to model a rich feature-set, but rather to concentrate on the functional aspects of the language. Objects and arrays are included mainly for their problematic aspects. These involve variadic data-constructors and the concept of object-identities. The former is uncommon to pure functional programming languages, because lists are most often modelled as cons-cells[14]. The latter is problematic, because object-identities naturally conflict with value-equality.

Definition 16. The concrete syntax of Λ_c^3 is given by the following production rules:

$$
\begin{array}{ll}
Expression_{\Lambda} & ::= \ldots \\
 & \mid ArrayLiteral_{\Lambda} \\
 & \mid ObjectLiteral_{\Lambda} \\
ArrayLiteral_{\Lambda} & ::= [\,] \mid [ElementList_{\Lambda}] \\
ElementList_{\Lambda} & ::= Expression_{\Lambda} \\
 & \mid Expression_{\Lambda}, ElementList_{\Lambda} \\
ObjectLiteral_{\Lambda} & ::= \{\} \mid \{PropertyDefinitionList_{\Lambda}\} \\
PropertyDefinitionList_{\Lambda} & ::= PropertyDefinition_{\Lambda} \\
 & \mid PropertyDefinition_{\Lambda}, PropertyDefinitionList_{\Lambda} \\
PropertyDefinition_{\Lambda} & ::= StringLiteral_{ES} : Expression_{\Lambda} \\
CallExpression_{\Lambda} & ::= Expression_{\Lambda}.IdentifierName_{\Lambda}
\end{array}
$$

```
{
   "value" : 42,
   "children" : [
      {
         "value"    : 41,
         "children" : []
      },
      {
         "value"    : 43,
         "children" : []
      }
   ]
}
```

Figure 13: JSON-encoding of a tree with root value 42 and two children with values 41 and 43 respectively.

Strings	str	an infnite set
Properties	$\mathit{prop} ::= (\texttt{Prop}\ \mathit{str}\ M)$	
Hashmaps	$\mathit{map}^n ::= (\texttt{Map}\ \mathit{prop}_1\ \mathit{prop}_2\ \ldots\ \mathit{prop}_n)$	

Figure 14: Strings, Properties and Objects

One idea to admit variadic data-constructors, is to extend the language with constructors of arity n for each possible size. As a consequence, the set of language terminals grows to a countable, infinite set. While this is not a problem from a mathematical stance, it still has the disadvantage that this kind of modelling does not reflect real language implementations. Functional languages instead tend to model inductive data-structures and provide syntactical sugar to cope with them. Figure 15 demonstrates this technique for arrays.

Lists	$\mathit{list} ::= \texttt{Empty} \mid (\texttt{Cons}\ M\ \mathit{list})$

Figure 15: Inductive List

The migration into the language facilitates a reuse of ES native array syntax and yields an interesting alternative semantic model. This opens the argument for optimized implementations of inductive lists in future language specifications.

Objects and lists bring their own operations, covering the full set of the ES specification is beyond the scope of the thesis. The modelling process is indeed very similar to the one for primitive operators, therefore, only a few of which are modelled for exemplary purpose. Figure 16 shows the exact list.

$$\begin{array}{rl}
\text{Signature} & \Sigma ::= \ldots \mid \texttt{Empty} \mid \texttt{Cons} \mid \texttt{Prop} \mid \texttt{Map} \mid \texttt{length} \mid \texttt{get} \mid \texttt{concat} \\
\text{Constructors} & c^0 ::= \cdots \mid \texttt{Empty} \\
 & c^2 ::= \cdots \\
 & \quad \mid (\texttt{Cons}\ M\ \mathit{list}) \\
 & \quad \mid (\texttt{Prop}\ \mathit{str}\ M) \\
 & c^n ::= \cdots \mid \mathit{str}^n \mid \mathit{map}^n \\
 & \quad \text{for every } n \in \mathbb{N} \\
\text{Operators} & o^1 ::= \cdots \mid (\texttt{length}\ M) \\
 & o^2 ::= \cdots \mid (\texttt{get}\ M\ \mathit{string}) \\
 & \quad \mid (\texttt{concat}\ M\ N)
\end{array}$$

Figure 16: Operations for compound datatypes

An important point here is that the abstract syntax uses typing information to construct maps and lists inductively. It falls into the responsibility of the parser to respect this information. To this end, the following parser collects this information from the concrete syntax and propagates it to the abstract syntax. For ease of use, the types discussed here collapse with the symbols from the concrete syntax. For example, if the parser finds an expression of the form $[e_1]$ it may infer that the type of the expression is *ArrayLiteral* and the type of the subexpression is *ElementList*. The parser has then the possibility to handle the case for the *ElementList* and *PropertyList* differently. To this end, the following parser implicitly defines two sub-parsers $\xrightarrow{el}$ and $\xrightarrow{pl}$.

Definition 17. The parser $\mathit{parse}^3 : \Lambda^3_c \to \Lambda^3$ is the smallest closure under the following operations:

$$\begin{array}{lll}
[\,] & \to \texttt{Empty} & \\
\{\mathsf{e}\} & \to (\texttt{Map}\ e'), & \text{if } e \xrightarrow{pl} e' \\
[\mathsf{e}] & \to (e'), & \text{if } e \xrightarrow{el} e' \\
e_1 : e_2 & \to (\texttt{Prop}\ e_1\ e_2) & \\
e_1, e_2 & \xrightarrow{pl} e_1\ e_2' & \text{if } e_2 \xrightarrow{pl} e_2' \\
e_1, e_2 & \xrightarrow{el} (\texttt{Cons}\ e_1\ e_2') & \text{if } e_2 \xrightarrow{el} e_2' \\
e_1.e_2 & \to (\texttt{get}\ e_1\ e_2) & \\
(\texttt{get}\ e_1\ \texttt{length}) & \to (\texttt{length}\ e_1) & \\
(\texttt{get}\ e_1\ \texttt{concat})(e_2) & \to (\texttt{concat}\ e_1\ e_2) &
\end{array}$$

3.3.2 Operational Semantics

The semantics of data-structures are determined by the operators they offer. The purpose of the list-operators is very much self-explanatory: `concat` concatenates two lists and *length* shall compute a list's length. `get` accepts an object and a key-name and returns the associated value from the object. Note that these operators are less generic compared to the operators for primitive datatypes.

$$\begin{array}{ll}
\delta(\texttt{length}, \texttt{Empty}) & \overset{\text{def}}{=} 0 \\
\delta(\texttt{length}, (\texttt{Cons}\ M\ \mathit{list})) & \overset{\text{def}}{=} \delta(1 + \delta(\texttt{length}, \mathit{list})) \\
\delta(\texttt{concat}, \texttt{Empty}, \mathit{list}) & \overset{\text{def}}{=} \mathit{list} \\
\delta(\texttt{concat}, (\texttt{Cons}\ M\ \mathit{list}_1), \mathit{list}_2)) & \overset{\text{def}}{=} (\texttt{Cons}\ M\ \delta(\mathit{concat}, \mathit{list}_1, \mathit{list}_2)) \\
\delta(\texttt{get}, (\texttt{Cons}\ M\ \mathit{list}), 0) & \overset{\text{def}}{=} M \\
\delta(\texttt{get}, (\texttt{Cons}\ M\ \mathit{list}), n) & \overset{\text{def}}{=} (\texttt{get}\ \mathit{list}\ \delta(n-1))
\end{array}$$

$$\delta(\texttt{get}, (\texttt{Map}\ (\texttt{Prop}\ \mathit{str}_1 M_1)\ \cdots\ (\texttt{Prop}\ \mathit{str}_n\ M_n)), \mathit{str}_0) \overset{\text{def}}{=} M_i$$
$$\text{if } \mathit{str}_i = \mathit{str}_0 \text{ and } \mathit{str}_j \neq \mathit{str}_0 \text{ for all } j \in \mathbb{N}, j < i$$

3.4 Recursion

3.4.1 Syntax

Recursion allows a programmer to refer back to the function he is currently defining inside the definition itself. Thus, a recursive solution breaks down a problem into a partial solution and a reasonable simpler instance of the same problem. The λ-calculus already supports recursion with fix-point-combinators, e.g the Y-combinator $\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (xx)))$ [8]. While this kind of recursion, lets call it *implicit* recursion, is interesting on its own sake, for example for complexity theoretic reasoning, it does not yield a suggestive programming model and, even worse, it cannot be typed. Most mainstream programming languages therefore have a construct for *explicit* recursion.

Explicit recursion lets the programmer define a *recursive equation system*, the search for a solution of the system is then delegated to the evaluator. As a first example for a *recursive equation system* consider the one from figure 17: A solution for y should be the result from computing the faculty of 3, i.e. 6. The recursion appears in the second equation, where the recursion-variable fac appears an both sides of the equation. Figure 18 shows a *recursive equation system* for a simple test, on whether 2 is an even number. The recursion here is somewhat obfuscated, because it is spread over multiple equations: Equations 18 and 19 both comprise the recursion-variables *even* and *odd*, but on mutually different sides of the equations. This kind of recursion is called *mutual recursion* [3].

$$y = fac\ 3 \tag{15}$$
$$fac = \lambda x.\texttt{if}\ (\texttt{strictEqual}\ x\ 0)\ 1\ (\texttt{mult}\ x\ (fac\ (\texttt{bminus}\ x\ 1)) \tag{16}$$

Figure 17: RES to compute faculty of 3

$$z = even\ 2 \tag{17}$$
$$even = \lambda x.\texttt{if}\ (\texttt{strictEqual}\ x\ 0)\ \texttt{True}\ (odd\ (\texttt{bminus}\ x\ 1)) \tag{18}$$
$$odd = \lambda x.\texttt{if}\ (\texttt{strictEqual}\ x\ 0)\ \texttt{False}\ (even\ (\texttt{bminus}\ x\ 1) \tag{19}$$

Figure 18: Mutual RES to compute the parity of 2

The leading equation, in such systems for recursion, is conventionally the one for which a search for a solution should actually be performed. The remaining equations *may* be used during the computation to find sub-solutions. Equations that are not enrolled in the computation at all, are considered *garbage*[4]. Figure 18 extends the current abstract syntax with the so called *letrec*-construct for *recursive equation systems.* In a *letrec*-term $\langle M|D\rangle$, M is called the *external* part, while D is sometimes referred to as the *internal part.* The syntax of *letrec* is more liberal than in the previous examples in the sense that it avoids the necessity to equate the leading term with a *recursion-variable*. Figure 19 transfers figure 18 to the given abstract syntax. Notice that the example stepped aside from earlier conventions and gave expressive names to *recursion variables*. This new convention is now kept throughout the rest of the thesis.

Definition 18. The abstract syntax of Λ^4 extendeds Λ^3 with a *letrec*-expression:

$$M, N, L \overset{\text{def}}{=} \ldots \mid \langle M | D \rangle$$
$$D \overset{\text{def}}{=} x_1 = M_1, x_2 = M_2, \ldots, x_n = M_n \qquad \text{where } x_i \neq x_j \text{ forall } i \neq j$$

ES does not have a trivial candidate for the concrete syntax of a *letrec*-expression. There are a few caveats here: First, the definition-list inside the internal part is scoped to the *letrec*-expression and must not leak bindings to outer expressions. This forbids the use of sequenced expression like the following:

```
(even = (x===0) ? true : odd(x-1),
 odd  = (x===0) ? false : even(x-1),
 even(2))
```

In this case, every binding would introduce a global variable. There are a few syntax-constructs which permit scoped bindings, i.e `var`-, `let`-, `const`-statements, function-declarations and formal-parameter lists. All these alternatives are subject to syntactical overhead: Statements neither produce return-values nor are they allowed to appear under arbitrary expressions. It is, therefore, necessary to wrap statements into an immediately invoked function-expressions, thereby, solving two problems at once: Applications may appear under every expression and an explicit return-value can be provided:

```
((()=> {
  const  even = (x => (x === 0) ? true  : (odd(x-1))),
         odd  = (x => (x === 0) ? false : (even(x-1)));
  return even(2)
})())
```

Building upon the formal parameter-list with default-values avoids most of the overhead, only the outermost application is kept:

```
((even = x => (x===0) ? true  : odd(x-1),
  odd  = x => (x===0) ? false : even(x-1)
) => even(2))()
```

There is however another problem with *letrec*-expressions: The definition list is intentionally unordered, thus it is allowed for a later equation to depend upon an earlier equation and vice-versa. This amount of horizontal sharing is not reflected by any of the solutions so far, as it can be witnessed by the following example, where the former external-part is explicitly named:

```
((
  dump = even(2),
  even = x => (x===0) ? true  : odd(x-1),
  odd  = x => (x===0) ? false : even(x-1)
) => dump)()
```

This example will produce an error because the function *even* is undefined, when the evaluator reaches the second line, according to ES semantics. To this end, it is possible to defer the execution with a thunk:

```
((
  dump = () => even()(2),
  even = () => x => (x===0) ? true  : odd()(x-1),
  odd  = () => x => (x===0) ? false : even()(x-1)
) => dump())()
```

This leads to the following concrete syntax for *letrec*-expression:

Definition 19. The concrete syntax Λ_c^4 is given by:

$$\begin{array}{lcl}
\mathit{Expression}_\Lambda & ::= & \cdots \\
 & | & (((\mathit{BindingList}_\Lambda) \Rightarrow \mathit{Expression}_\Lambda)()) \\
 & | & \mathit{ThunkApplication} \\
\mathit{BindingList}_\Lambda & ::= & \mathit{LexicalBinding}_\Lambda \\
 & | & \mathit{BindingList}_\Lambda, \mathit{LexicalBinding}_\Lambda \\
\mathit{LexicalBinding}_\Lambda & ::= & \mathit{IdentifierName} = () \Rightarrow \mathit{Expression}_\Lambda \\
\mathit{ThunkApplication}_\Lambda & ::= & \mathit{IdentifierName}_\Lambda()
\end{array}$$

The parser emphasizes that the use of the thunk here is only syntactical noise, from the perspective of the abstract syntax. Notice, how the parser equates the identifier from the left-hand side of an equation with thunk-applications from the right-hand side of the equation. At this point the difference between a variable-name and a real variable is important.

Definition 20. The parser $parse^4 : \Lambda_c^4 \to \Lambda^4$ is the smallest closure under the following operations:

$$\begin{array}{lll}
e & \to e', & \text{if } parse^3(e) = e' \\
e_1 = ((() \texttt{=>} e_2)()) & \to e_1() = e_2 & \\
e_1() & \to X_{e1()} & \\
(((e_1) \texttt{=>} e_2)()) & \to \langle\ e_2 \mid e_1\ \rangle &
\end{array}$$

$$\begin{aligned}
&\langle even\ 2| \\
&\ even = \lambda x.\texttt{if}\ (\texttt{strictEqual}\ x\ 0)\ \texttt{True}\ (odd\ (\texttt{bminus}\ x\ 1)), \\
&\ odd = \lambda x.\texttt{if}\ (\texttt{strictEqual}\ x\ 0)\ \texttt{False}\ (even\ (\texttt{bminus}\ x\ 1))\rangle
\end{aligned}$$

Figure 19: This program demonstrates the use of the *letrec*-construct in abstract syntax. The program recursively computes whether 2 is an even number.

The theory behind recursive term rewriting systems is tightly related to *abstract graph rewriting systems*, like described in [4, 3]. Although, the additional generality that *abstract graph rewriting systems* offer over *abstract term rewriting systems* is not needed here, there are still plenty of concepts to benefit from, when building an intuitive mathematical model. Most noticeable are the graphical representations, graphs offer for cycles that tree-structures commonly lack. The next partial goal is, therefore, to encode terms as graphs. Consider figure 20 as an motivating example. It shows the graph-encoding of the program from figure 19.

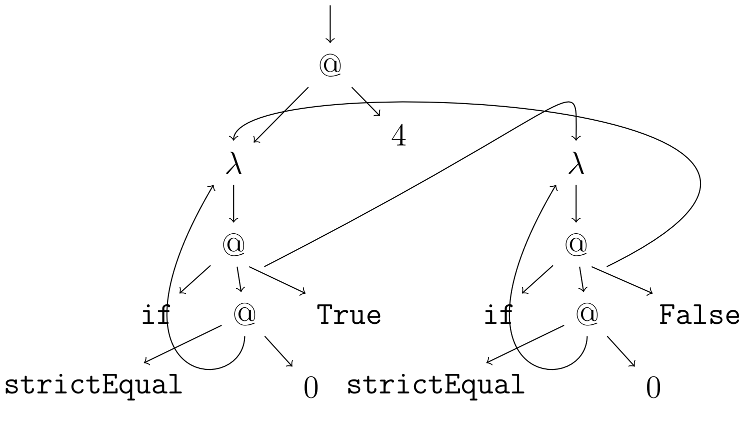
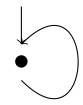


Figure 20: λ-graph for a program to compute a natural numbers parity

λ-graphs classify between nodes for abstraction, application, primitive operators, free variables and so-called *black holes*. In contrast to earlier illustrations, bound variable-occurrences are not considered as nodes anymore, but as bows pointing to the λ-node which binds the variable. When beneficial, the bow will be labelled with the variable-identifier. Similarly, recursion-variables are represented as bows, pointing to the right-hand side of the equation. To distinguish these bows from the former ones, bows that represent recursion-variables enter their target-node from above, while bows that represent normal variables enter their binding λ-node from below. The latter ones are referred to as *backpointers* in contrast to *normal pointers* or just *pointers*. Concerning *black holes*: For a first impression, consider the recursive program $\langle x \mid x = x\rangle$. If it was the task to encode this program as a graph, it would be impossible to define the set of nodes if only abstractions, applications, primitive operators, free variables and constructors were about to become nodes. This is exactly where black holes come into play. The program can be encoded as:



The following definition extends the one from [3] with labels the labels of the operators and data-constructors from Λ^4.

Definition 21. A λ-*graph* is a 4-tuple (V, L, A, r), where

- V is a set of nodes
- $L : V \to \{\lambda, @, \bullet\} \cup \Sigma$ is a labelling function
- $A : V \to (V \oplus \{\overline{v}|v \in V, L(v) = \lambda\} \oplus \mathcal{V})^*$ is a successor function, such that
$$|A| = \begin{cases} 0, & \text{if } L(v) = \bullet \\ 1, & \text{if } L(v) = \lambda \\ 2, & \text{if } L(v) = @ \end{cases}$$
- $r \in (V \oplus \{\overline{v}|v \in V, L(v) = \lambda\} \oplus \mathcal{V})$ is a root node

This definition should remind the reader of a parse-tree. Indeed, λ-graphs can bee seen as higher-order abstract syntax [12]. Intuitively speaking, a higher-order abstract syntax abstracts over variable-names in addition to the usual abstractions.

The λ-graph from figure 20 and the program from figure 19 are two encodings of the same underlying program. Yet, it might not be clear how the former can be constructed from the latter. Indeed, this process requires some more machinery. The point here is that there exist λ-graphs without term-representations. The overhang is however uninteresting for the thesis. Conversely speaking, only λ-graphs which have a corresponding λ-term are considered here; those are commonly referred to as *well-formed λ-graphs*. Ariola and Blom classified the *well-formed λ-graphs* as λ-graphs which have an associated *scoped* λ-graph [4]. Scoped λ-graphs can directly be derived from λ-terms.

Definition 22. A *scoping function* $S : \{v \in V | L(v) = \lambda\} \to \mathcal{P}(V)$ for a λ-graph (V, L, A, r) is a function, such that for every $v \in V$ with $L(v) = \lambda$ the following axioms hold:

$$\begin{array}{ll}
\text{(auto)} & v \in S(v) \\
\text{(bind)} & \text{forall } w : \overline{v} \text{ is an argument of } w \text{ implies } w \in S(w) \\
\text{(upwards-closure)} & w_1 \in S(w), \text{ if } \begin{array}{l} w_2 \neq v \text{ and} \\ w_2 \text{ is an argument of } w_1 \text{ and} \\ w_2 \in S(w) \end{array} \\
\text{(nesting)} & w \in S(v), \text{ forall } w \text{ with } \begin{array}{l} L(w) = \lambda : S(w) \cap S(v) = \emptyset \text{ or} \\ S(w) \subseteq S(v) \setminus \{v\} \text{ or} \\ S(v) \subseteq S(w) \setminus \{w\} \end{array}
\end{array}$$

Definition 23. A *scoped λ-graph* is a 5-tuple (V, L, A, S, r), where

- (V, L, A, r) is a λ-graph
- S is a scoping function
- root condition: $r \in \mathcal{V}$ or $r \in V$ such that: forall $v \in V : r \notin S(v) \setminus \{v\}$

The construction of a scoped graph from a given expression proceeds in two phases on the structure of the expression: First, a scoped pre-graph is constructed which is a λ-graph-like structure, but is more liberal in the sense that *black holes* might have zero or one successor. Second, this liberalism is resolved to produce a conventional λ-graph.

Notation. The following notational conventions apply for the remainder of the thesis. If it is clear from the context, then the set of nodes V of a scoped pre-graph for an expression M is referred to as V_M. If the expression is indexed with i, like in M_i, the set of nodes can also be referred to as V_i. Respectively, L_M,L_i, A_M, A_i, ...can be written to refer to other elements of the pre-graph. Furthermore, if two functions $f : X \to Y$, $g : V \to W$ have point-wise disjoint domains and images, then $f \cup g$ denotes the function $f \cup g : X \cup V \to Y \cup W$ with $x \to f(x)$, if $x \in X$ and $x \to g(x)$ if $x \in V$. An iterated version for functions $f_1, \cdots, f_n$ is available as $\bigcup_{i=1}^{n} f_i$. Functions with an empty domain might be written as $\emptyset$. For tupels a notation for replacements is introduced: Given a tupel $(e_0, \cdots, e_n)$, let $(e_0, \cdots, e_n)[e_i \leftarrow e_k]$ denote the tupel, where all occurrences of e_i have been replaced with e_k.

Definition 24. The *scoped pre-graph* of an expression is

$$
\begin{aligned}
&\rho_{pre}(x) && \overset{\text{def}}{=} (\{v_0\}, v_0 \to \bullet, v_0 \to x, \emptyset, v_0)\\
&\rho_{pre}(M\ N) && \overset{\text{def}}{=} (\{v_0\} \cup V_M \cup V_N, L, A, S_M \cup S_N, v_0),\ \text{where}\\
& && L = \{v_0 \to @\} \cup L_M \cup L_N\\
& && A = \{v_0 \to (r_0, r_1)\} \cup A_M \cup A_N\\
&\rho_{pre}(\lambda x.M) && \overset{\text{def}}{=} (\{v_0\} \cup V_M, L, A, v_0 \to V, v_0),\ \text{where}\\
& && L = \{v_0 \to \lambda\} \cup L_M\\
& && A = \{v_0 \to (r_0)\} \cup A_M[x \leftarrow \overline{v_0}]\\
&\rho_{pre}(\langle M_0 \mid x_1 = M_1, \cdots, x_n = M_n\rangle) && \overset{\text{def}}{=} (V, L, A, S, r_0),\ \text{where,}\\
& && V = \bigcup_{i=0}^{n} V_i,\ L = \bigcup_{i=0}^{n} L_i,\ S = \bigcup_{i=0}^{n} S_i\\
& && A(v) = (\bigcup_{i=0}^{n} A_i(v)[x_i \leftarrow r_j]),\ \text{for all } 1 \leq j \leq n\\
&\rho_{pre}(\sigma) && \overset{\text{def}}{=} (\{v_0\}, v_0 \to \sigma, \emptyset, \emptyset, v_0)
\end{aligned}
$$

Figure 21 shows the scoped pre-graph for the running example. The rectangles around the λ-nodes draw frames around the corresponding scopes. It is easy to see, that if the black holes are replaced with backpointers leading directly from the application to the binding λ-node, then the desired scoped λ-graph is obtained. In general, the scoped λ-graph can be derived from the scoped pre-graph in two steps: First, all pointers from black holes leading to themselves are removed. Second, sequences of pointers that first lead into a black hole and then proceed to some other node are replaced by a direct pointer the latter.

3.4.2 Operational Semantics

The introduction of a *letrec*-expression to the language has unexpected implications on the operational semantics. Probably the most crucial one is that confluence is lost for unrestricted reductions [4]. Ariola and Blom developed an approximate

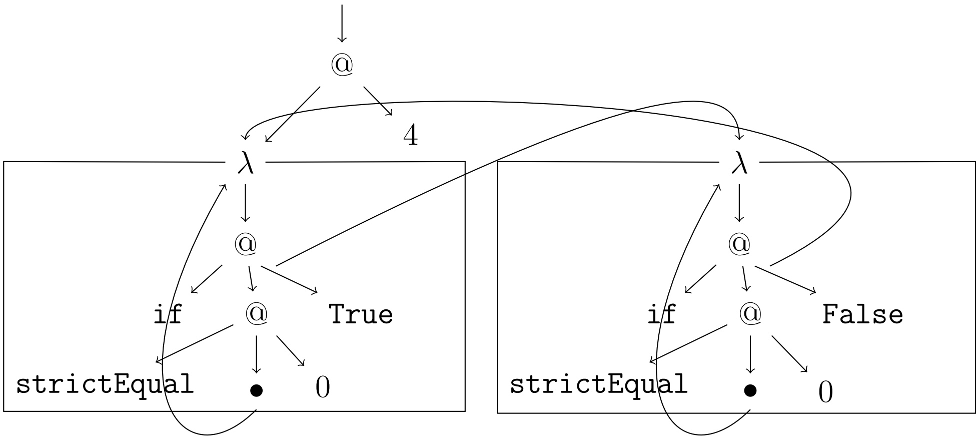


Figure 21: Scoped pre-graph for a program to compute a natural number's parity

notion of confluence which they call *confluence up to a quasi order*[4]. Building on that, they showed that it is possible to formulate reduction systems for cyclic λ-terms which are confluent up to *information content*. Going further, they also showed that standard reduction is confluent, but at the cost that only the top-most information content can be reached.

Definition 25. An *ordered abstract rewriting system* is a structure $(A, \rightarrow, \preceq)$, where $(A, \rightarrow)$ is an abstract rewriting system and $(A, \preceq)$ is a quasi order.

Given an ARS $(A, \rightarrow, \preceq)$. Then $\rightarrow$ is *confluent up to* $\preceq$, if
$\forall a, b, c \in A : a \twoheadrightarrow b, a \twoheadrightarrow c \implies \exists d \in A : b \twoheadrightarrow d, c \preceq d$

Intuitively, *information content* quantifies the amount of simplicity that is associated with a cyclic term. This corresponds to the idea that at the end of rewriting, there should be a most simple expression. This also suggests the following strategy to derive qualified reduction rules. First, complex and simple terms are identified. Then reduction rules are introduced to reduce the complexity with respect to *information content*.

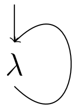
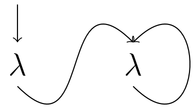
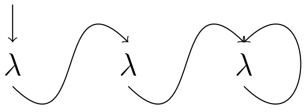
Definition 26. The *information content* of M is given by the function ω, which given M, returns the normal form of M with respect to the following rules:

$$
\begin{array}{ll}
(\lambda x.M)\ N & \xrightarrow{\omega} \Omega \\
(o^n\ M_1 \cdots M_n) & \xrightarrow{\omega} \Omega \\
\langle C[x] \mid x = M, D \rangle & \xrightarrow{\omega} \langle C[\Omega] \mid x = M, D \rangle \\
\Omega\ M & \xrightarrow{\omega} \Omega \\
\langle M \mid D \rangle & \xrightarrow{\omega} M, \text{if } D \perp M
\end{array}
$$

Definition 27. The relation $\preceq_\Omega$ defines a pre-order on terms in the following way:

$$
\begin{array}{ll}
M \preceq_\Omega N, & \text{if } |\omega(M)|_\Omega \leq |\omega(N)|_\Omega \\
 & \text{where } |\cdot|_\Omega \text{ counts the } \Omega\text{-symbols}
\end{array}
$$

The primary challenge taken here is to rewrite cyclic λ-terms. Recall, that terms represent λ-graphs. For graphs it is possible to resolve cycles, hence transforming the graph to an acyclic tree structure, while preserving its semantics. This transformation does, obviously, not always end in a finite tree, this is to say, the resulting graph can not be represented by a term. This issue is picked up later in the chapter, when termination is discussed. For now, the focus is on the operational semantics of the *tree unwinding* transformation, i.e. on the small-step semantics. The following table shows two steps of *tree unwinding* of a cyclic graph.

λ-term	$\langle x \mid x = \lambda y.x \rangle$		$\langle \lambda y.x \mid x = \lambda y.x \rangle$		$\langle \lambda y.\lambda y.x \mid x = \lambda y.x \rangle$
λ-graph					
Information Content	Ω	$\preceq_\Omega$	$\lambda y.\Omega$	$\preceq_\Omega$	$\lambda y.\lambda y.\Omega$

The key idea to spot here is that tree-unwinding substitutes a variable inside the external part of *letrec*-expression by a named expression from the internal part. This motivates the external substitution reduction rule:

$$\langle C[x] \mid x = M, D \rangle \rightarrow \langle C[M] \mid x = M, D \rangle$$

Another observation here is that as long as the number of substitution steps is finite, it is possible to represent the λ-graph by a λ-expression. This attempt fails, however, once infinite many reduction steps are considered. In this case, the existence of an infinite reduction-sequence is justified by the fact that the recursion is not working against a simpler base case, but instead produces a equally complex successor in each step. Notice the duality here, while external substitution increases information content, it does not reduce the complexity of the term.

Building on external substitution, it is now natural to define a variant of β-reduction.

$$(\lambda x.M)\ N \to \langle M \mid x = N \rangle$$

Recall, that the term on the left-hand side represents an obfuscated closure. This gives rise to the idea that that *letrec*-expressions can evenly be seen as closures which directly leads to a modified version of the *lift*-reduction-rule:

$$\langle M \mid D \rangle\ N \to \langle M\ N | \ D \rangle$$

Together with δ-reduction-rule these small-step reductions are sufficient to define standard call-by-name reduction. To this end, appropriate evaluation contexts must be selected. The machines for the call-by-value and call-by-need reduction strategies require some additional administrative axioms. The following machines are based on [4, 9].

Syntactic Domains

$$
\begin{array}{lrl}
\text{Values} & V & ::= \lambda x.M \mid c^n\ V_1 \cdots V_n \\
\text{Answers} & A & ::= V \mid \langle A \mid D \rangle \\
\text{Evaluation Contexts} & E & ::= \lambda x.E \mid \langle E \mid D \rangle \\
 & & \quad \mid \quad \mathit{App}[y, M_1, \cdots, M_{i-1}, E, M_{i+1}, M_n] \\
 & & \quad \mid \quad \mathit{App}[[\ ], M_1, \cdots, M_n] \\
 & \mathit{App} & ::= [\ ] \mid \mathit{App}\ [\ ] \mid \langle \mathit{App} \mid D \rangle \\
 & & \quad \mid \quad (o^n\ V_1\ V_{i-1}\ \cdots E\ M_{i+1}, \cdots\ M_n)
\end{array}
$$

Reduction Axioms

$$
\begin{array}{ll}
(\beta \circ\,) & E[(\lambda x.M)\ N] \\
\to & E[\langle M \mid x = N \rangle] \\
 & \\
(\text{external substitution}) & E[\langle E[x] \mid x = V, D \rangle] \\
\to & E[\langle E[V] \mid x = V, D \rangle] \\
 & \\
(\text{lift}) & E[\langle A \mid D \rangle\ N] \\
\to & E[\langle A\ N \mid D \rangle] \\
 & \\
(\delta) & E[o^n\ V_1 \ldots V_n] \\
\to & E[\delta(o^n, V_1, \ldots, V_n)]
\end{array}
$$

Figure 22: Call-by-name rewriting system for Λ^4: λ^4_{name}

Syntactic Domains

$$\begin{array}{llcl}
\text{Values} & V & ::= & \lambda x.M \mid c^n\ V_1 \cdots V_n \\
\text{Answers} & A & ::= & V \mid \langle\ A \mid D\rangle \\
\text{Dependencies} & D[x, x_n] & ::= & x = E[x_1], \cdots, x_{n-1} = E[x_n], D \\
\text{Evaluation Contexts} & E & ::= & [\ ] \mid (E\ M) \\
 & & \mid & \langle E \mid D\rangle \\
 & & \mid & \langle E[x] \mid x = E, D\rangle \\
 & & \mid & \langle E[x] \mid D[x, x_n], x_n = E\rangle \\
 & & \mid & (o^n\ V_1\ V_{i-1}\ \cdots E\ M_{i+1} \cdots\ M_n)
\end{array}$$

Reduction Axioms

$$\begin{array}{ll}
(\beta \circ\,) & E[(\lambda x.M)\ N] \\
\rightarrow & E[\langle M \mid x = N\rangle] \\
\text{(external substitution)} & E[\langle E[x] \mid x = V, D\rangle] \\
\rightarrow & E[\langle E[V] \mid x = V, D\rangle] \\
\text{(internal substitution)} & E[\langle E[x] \mid D[x, x_n], x_n = V\rangle] \\
\rightarrow & E[\langle E[x] \mid D[x, V], x_n = V\rangle] \\
\text{(lift)} & E[\langle A \mid D\rangle\ N] \\
\rightarrow & E[\langle A\ N \mid D\rangle] \\
\text{(internal merge)}_1 & E[\langle E[x] \mid x = \langle A \mid D\rangle, D_1\rangle] \\
\rightarrow & E[\langle E[x] \mid x = A, D, D_1\rangle] \\
\text{(internal merge)}_2 & E[\langle E[x] \mid D[x, x_n], x_n = \langle A \mid D\rangle\rangle] \\
\rightarrow & E[\langle E[x] \mid D[x, x_n], x_n = A, D\rangle] \\
(\delta) & E[o^n\ V_1 \ldots V_n] \\
\rightarrow & E[\delta(o^n, V_1, \ldots, V_n)]
\end{array}$$

Figure 23: Call-by-need calculus for Λ^4: λ^4_{need}

Syntactic Domains

Values	$V ::= x \mid \lambda x.M \mid c^n\ V_1 \cdots V_n$
Answers	$A ::= V \mid \langle A \mid D \rangle$
Lift Contexts	$L ::= [\,] \mid L\ M \mid V \mid L$
Binding Contexts	$B ::= x = [\,], D$
Evaluation Contexts	$E ::= L \mid B[L]$
	$\mid o^n\ V_1\ V_{i-1}\ \cdots E\ M_{i+1} \cdots\ M_n$

Reduction Axioms

$(\beta \circ\,)$	$E[(\lambda x.M)\ N]$
$\rightarrow$	$E[\langle M \mid x = N \rangle]$
(external substitution)	$E[\langle E[x] \mid x = V, D \rangle]$
$\rightarrow$	$E[\langle E[V] \mid x = V, D \rangle]$
(internal substitution)	$E[\langle M \mid x = C[x_1], x_1 = V, D \rangle]$
$\rightarrow$	$E[\langle M \mid x = C[V], x_1 = V, D \rangle]$
(lift)	$E[M\ \langle N \mid D \rangle]$
$\rightarrow$	$E[\langle M\ N \mid D \rangle]$
(external merge)	$E[\langle \langle M \mid D_1 \rangle \mid D_2 \rangle]$
$\rightarrow$	$E[\langle M \mid D_1, D_2]$
(internal merge)	$E[\langle M \mid x = \langle N \mid D_1 \rangle \mid D_2 \rangle]$
$\rightarrow$	$E[\langle M \mid x = N, D_1, D_2]$
(δ)	$E[o^n\ V_1 \ldots V_n]$
$\rightarrow$	$E[\delta(o^n, V_1, \ldots, V_n)]$

Figure 24: Call-by-value calculus for Λ^4: λ^4_{value}

$$\begin{array}{l}
\langle \mathit{even}\ 2 \mid \\
\qquad \mathit{even} = \lambda x.\texttt{if}\ (\texttt{strictEqual}\ x\ 0)\ \texttt{True}\ (\mathit{odd}\ (\mathit{uminus}\ x)), \\
\qquad \mathit{odd} = \lambda x.\texttt{if}\ (\texttt{strictEqual}\ x\ 0)\ \texttt{False}\ (\mathit{even}\ (\mathit{uminus}\ x))\rangle
\end{array}$$

$$\qquad\qquad \downarrow \text{(external substitution)}$$

$$\langle(\lambda x.\texttt{if}\ (\texttt{strictEqual}\ x\ 0)\ \texttt{True}\ \mathit{odd}\ (\mathit{uminus}\ x))\ 2 \mid \cdots\rangle$$

$$\qquad\qquad \downarrow (\beta\circ)$$

$$\langle\langle\texttt{if}\ (\texttt{strictEqual}\ x\ 0)\ \texttt{True}\ \mathit{odd}\ (\mathit{uminus}\ x) \mid x = 2\rangle \mid \cdots\rangle$$

$$\qquad\qquad \downarrow \text{(external substitution)}$$

$$\langle\langle\texttt{if}\ (\texttt{strictEqual}\ 2\ 0)\ \texttt{True}\ \mathit{odd}\ (\mathit{uminus}\ 2) \mid x = 2\rangle \mid \cdots\rangle$$

$$\qquad\qquad \downarrow (\delta)$$

$$\langle\langle\texttt{if}\ \texttt{False}\ \texttt{True}\ \mathit{odd}\ (\mathit{uminus}\ 2) \mid x = 2\rangle \mid \cdots\rangle$$

$$\qquad\qquad \downarrow (\delta)$$

$$\langle\langle\mathit{odd}\ (\mathit{uminus}\ 2) \mid x = 2\rangle \mid \cdots\rangle$$

$$\qquad\qquad \downarrow \text{(external substitution)}$$

$$\langle\langle(\lambda x.\texttt{if}\ (\texttt{strictEqual}\ x\ 0)\ \texttt{False}\ (\mathit{even}\ (\mathit{uminus}\ x))\ (\mathit{uminus}\ 2) \mid x = 2\rangle \mid \cdots\rangle$$

$$\qquad\qquad \downarrow (\beta\circ)$$

$$\langle\langle\langle\texttt{if}\ (\texttt{strictEqual}\ x\ 0)\ \texttt{False}\ (\mathit{even}\ (\mathit{uminus}\ x) \mid\ x = (\mathit{uminus}\ 2)\rangle \mid x = 2\rangle \mid \cdots\rangle$$

$$\qquad\qquad \downarrow (\delta)$$

$$\langle\langle\langle\texttt{if}\ (\texttt{strictEqual}\ x\ 0)\ \texttt{False}\ (\mathit{even}\ (\mathit{uminus}\ x) \mid\ x = 1\rangle \mid x = 2\rangle \mid \cdots\rangle$$

$$\qquad\qquad \downarrow \text{(external substitution)}$$

$$\langle\langle\langle\texttt{if}\ (\texttt{strictEqual}\ 1\ 0)\ \texttt{False}\ (\mathit{even}\ (\mathit{uminus}\ 1) \mid\ x = 1\rangle \mid x = 2\rangle \mid \cdots\rangle$$

$$\qquad\qquad \downarrow (\delta)$$

$$\langle\langle\langle\texttt{if}\ \texttt{False}\ \texttt{False}\ (\mathit{even}\ (\mathit{uminus}\ 1) \mid\ x = 1\rangle \mid x = 2\rangle \mid \cdots\rangle$$

$$\qquad\qquad \downarrow (\delta)$$

$$\langle\langle\langle(\mathit{even}\ (\mathit{uminus}\ 1) \mid\ x = 1\rangle \mid x = 2\rangle \mid \cdots\rangle$$

$$\qquad\qquad \downarrow \text{(external substitution)}$$

$$\langle\langle\langle(\lambda x.\texttt{if}\ (\texttt{strictEqual}\ x\ 0)\ \texttt{True}\ \mathit{odd}\ (\mathit{uminus}\ x))\ (\mathit{uminus}\ 1) \mid\ x = 1\rangle \mid x = 2\rangle \mid \cdots\rangle$$

$$\qquad\qquad \downarrow (\beta\circ)$$

$$\langle\langle\langle\langle\texttt{if}\ (\texttt{strictEqual}\ x\ 0)\ \texttt{True}\ \mathit{odd}\ (\mathit{uminus}\ x)\mid x = (\mathit{uminus}\ 1)\rangle \mid\ x = 1\rangle \mid x = 2\rangle \mid \cdots\rangle$$

$$\qquad\qquad \downarrow (\delta)$$

$$\langle\langle\langle\langle\texttt{if}\ (\texttt{strictEqual}\ x\ 0)\ \texttt{True}\ \mathit{odd}\ (\mathit{uminus}\ x)\mid x = 0\rangle \mid\ x = 1\rangle \mid x = 2\rangle \mid \cdots\rangle$$

$$\downarrow \text{(external substitution)}$$

$$\langle\langle\langle\langle \texttt{if (strictEqual } 0\ 0\texttt{) True } \mathit{odd}\ (\mathit{uminus}\ 0) | \ x = 0\rangle \ | \ \ x = 1\rangle \ | \ x = 2\rangle \ | \cdots \rangle$$

$$\downarrow (\delta)$$

$$\langle\langle\langle\langle \texttt{if True True } \mathit{odd}\ (\mathit{uminus}\ 0) | \ x = 0\rangle \ | \ \ x = 1\rangle \ | \ x = 2\rangle \ | \cdots \rangle$$

$$\downarrow (\delta)$$

$$\langle\langle\langle\langle \texttt{True} \ | \ x = 0\rangle \ | \ \ x = 1\rangle \ | \ x = 2\rangle \ | \cdots \rangle$$

Figure 25: Example reduction with λ^4_{need}

4 Conclusion

ES takes a unique position around mainstream programming languages - technically, socially and historically. It has been well-known for its preference of prototypical inheritance over classical inheritance since its first appearance. Recent events have also strengthened the functional aspects of the language. The reasons for this shift are complex. The thesis identified two driving factors: One is its success in rather stateful and reactive domains. In this constellation concurrency and synchronisation become difficult problems and often require increased attention and cognition of the programmer. The absence of side-effects can greatly contribute to the predictability of a program and decreases the cognitive load. Another reason is the high influence that other programming languages take on ES. Due to its exclusive settlement in the browser, programmers have been forced to work with the language for a long time, since there has been no alternatives around. These programmers were/are characterised with different backgrounds and experiences from other programming languages, sometimes coupled with strong preferences for certain programming styles. This gave raise to new programming languages which use ES as a compile-target. Two mentionable languages with influence on ES and strong functional background are *Elm* and *PureScript*.

The thesis demonstrates techniques to identify purely functional aspects of the language and walks through different alternatives to construct a *programming language theoretical* model of the desired subset. The first iteration step establishes a connection between ES and the λ-calculus. This is then the staring-point for various extensions. The language of the second iteration adds basic constants and operators for boolean algebra and floating point arithmetic to the language. At this point the language can be seen as member from the ISWIM family of programming languages[1]. From here, is is only a small to step to integrate data-structures to the language. The last iteration step enriches the language with a *letrec*-construct for explicit, mutual recursion.

The subset of the thesis is not maximal with respect to functional purity. In fact, there are other features which likely preserve this exceptional property. Conversely, there are some features missing in the language which would be beneficial for functional programming. The following suggestions are not justified by the thesis, but are highly subjective. Furthermore, a suggestion should not be mistaken for absolution, but rather as an endorsement for further discussion. With these thoughts in mind, ES lacks a low-level mechanism which would enable a programmer to define custom data-constructors. Second, there is no native facility to build function definitions inductively from simple base-cases towards more complex cases. Although, it is possible to simulate this behaviour by means of recursion.

With *WebAssembly* in expectation browsers open up their doors for new competition between programming languages. The thesis provides guidance in the form of operational semantics for language-implementors and -designers who either directly work on ES or on foreign languages that compile to ES.

[1] Interestingly Scheme is sometimes referred to as the closest living relative of ISWIM. Brendan Eich, the creator of JavaScript, once stated that when he began his work on JavaScript he was charged with the task to bring Scheme to the browser. In some sense, the thesis closes this gap between the triangular relation of ISWIM, Scheme and ES

References

[1] Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

[2] M. S. Anil Maheshwari. *Introduction to Theory of Computation.* 2014.

[3] Z. Ariola and J. W. Klop. Cyclic lambda graph rewriting. In *Logic in Computer Science, 1994. LICS '94. Proceedings., Symposium on*, pages 416–425, Jul 1994.

[4] Z. M. Ariola and S. Blom. Skew confluence and the lambda calculus with letrec. *Annals of Pure and Applied Logic*, 117(1–3):95 – 168, 2002.

[5] Z. M. Ariola, J. Maraist, M. Odersky, M. Felleisen, and P. Wadler. A call-by-need lambda calculus. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 233–246, New York, NY, USA, 1995. ACM.

[6] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification.* 6 edition, June 2015.

[7] E. Elliott. *Programming JavaScript Applications.* O'Reilly Media, 2014.

[8] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex.* The MIT Press, 1st edition, 2009.

[9] T. Hirschowitz, X. Leroy, and J. B. Wells. Compilation of extended recursion in call-by-value functional languages. *Higher-Order and Symbolic Computation*, 22(1):3–66, 2009.

[10] J. Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, Apr. 1989.

[11] J. C. Mitchell. *Foundations for Programming Languages.* 1996.

[12] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 199–208, New York, NY, USA, 1988. ACM.

[13] A. Rauschmayer. *Speaking JavaScript.* O'Reilly Media, 2014.

[14] A. Sabry. What is a purely functional language? *J. Funct. Program.*, 8(1):1–22, Jan. 1998.