**UNIVERSITY OF SOUTHAMPTON**
Faculty of Engineering and Physical Sciences
School of Electronics and Computer Science

A project report submitted for the award of
BSc Computer Science

Project supervisor: Dr Julian Rathke
Second examiner: Dr Abdolbaghi Rezazadeh

# Formally verified derivation of an executable and terminating CEK machine from call-by-value $\lambda\hat{p}$-calculus

by Wojciech Krzysztof Rozowski

April 23, 2021

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

A project report submitted for the award of BSc Computer Science

by Wojciech Krzysztof Rozowski

Throughout the development of functional programming languages, one of the most researched topics is their implementation. A common approach is specifying their semantics in terms of abstract machines, first-order deterministic transition systems. Biernacka & Danvy showed the correspondence between abstract machines and reduction calculi. The authors introduced $\lambda\hat{p}$-calculus, a variant of $\lambda$-calculus with explicit substitutions via closures, and derived multiple known abstract machines such as CEK or Krivine machine.

Using Agda, a dependently-typed programming language, and a proof-assistant, we successfully build upon earlier work by Swierstra on the formalisation of the Krivine machine and adapt his approach to obtaining a CEK machine.

The main contributions include a mechanized proof of equivalence of derived CEK machine with the small-step recursive evaluator for $\lambda\hat{p}$-calculus under call-by-value reduction strategy. We successfully introduced and formalised a variant of Strong Normalisation property for $\lambda\hat{p}$-calculus under call-by-value, inspired by Martin-Löf and Tait's proof of Strong Normalisation for Simply Typed $\lambda$-calculus.

The result is a correct-by-specification and executable CEK machine proven to terminate for well-typed terms of the language. This study could be a potential basis for further research, such as formalising executable machines for context-sensitive languages with control operators, such as Parigiot's $\lambda\mu$-calculus.

# Contents

# List of Figures

# Nomenclature

| | |
|---|---|
| ASM | Abstract State Machine |
| $\lambda p$ | Curien Calculus of Closures |
| $\lambda \hat{p}$ | Extended Curien Calculus of Closures |
| Redex | Reducible Expression |
| Applicative order | Leftmost Innermost Redex reduction (Call-by-value) |
| Normal order | Leftmost Outermost Redex reduction (Call-by-name) |
| Evaluation Context | Continuation frame |
| STLC | Simply Typed $\lambda$-Calculus |
| PLFA | Programming Languages Foundations in Agda (Wadler et al. (2020)) |
| BC | Bove-Capretta (Bove (2003)) |
| Closure | $\lambda$ term along its substitution environment |
| Closed term | Closure or closure application |
| Term | $\lambda$ term |
| CEK | Code, Environment and Kontinuation |
| $\alpha$-equivalent | Invariant upon renaming |

# Acknowledgements

First of all, I would like to thank Dr Julian Rathke for supervising this project and being such a great and motivating mentor. I am grateful for his time, patience, and for being my academic inspiration.

I wish to express my gratitude to Dr Wouter Swierstra for his outstanding help. His work on the Krivine machine inspired this project, and I am truly grateful for all discussions we had and the help I received. It was a pleasure to learn from him.

I am grateful to Prof. Thorsten Altenkirch for the discussion around his paper on Normalisation by Evaluation and for inspiring my Strong Normalisation proof for $\lambda\hat{p}$-calculus under call-by-value.

Also, I wish to express my gratitude to Dr Filip Sieczkowski for the discussion and tips about well-foundedness relation on configurations of decomposition function.

I would like to also thank Dr Abdolbaghi Rezazadeh, my second examiner, for his useful academic writing tips.

On a more personal note, I would like to express my gratitude to my parents for their love and support. I wouldn't make it that far if it weren't for you.

To my best friend, Aleksandra M. for her never-ending support and for providing positive distractions.

Last, but not least, to my flatmate Molly S. for her time spent proofreading this report.

**Statement of Originality**

- I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students.
- I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.
- I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

***You must change the statements in the boxes if you do not agree with them.***

We expect you to acknowledge all sources of information (e.g. ideas, algorithms, data) using citations. You must also put quotation marks around any sections of text that you have copied without paraphrasing. If any figures or tables have been taken or modified from another source, you must explain this in the caption and cite the original source.

**I have acknowledged all sources, and identified any content taken from elsewhere.**

If you have used any code (e.g. open-source code), reference designs, or similar resources that have been produced by anyone else, you must list them in the box below. In the report, you must explain what was used and how it relates to the work you have done.

**I have used Agda standard library, PLFA (available under Creative Commons 4.0), as well as Wouter Swierstra Krivine machine formalization (with permission of the author)**

You can consult with module teaching staff/demonstrators, but you should not show anyone else your work (this includes uploading your work to publicly-accessible repositories e.g. Github, unless expressly permitted by the module leader), or help them to do theirs. For individual assignments, we expect you to work on your own. For group assignments, we expect that you work only with your allocated group. You must get permission in writing from the module teaching staff before you seek outside assistance, e.g. a proofreading service, and declare it here.

**I did all the work myself, or with my allocated group, and have not helped anyone else.**

We expect that you have not fabricated, modified or distorted any data, evidence, references, experimental results, or other material used or presented in the report. You must clearly describe your experiments and how the results were obtained, and include all data, source code and/or designs (either in the report, or submitted as a separate file) so that your results could be reproduced.

**The material in the report is genuine, and I have included all my data/code/designs.**

We expect that you have not previously submitted any part of this work for another assessment. You must get permission in writing from the module teaching staff before re-using any of your previously submitted work for this assessment.

**I have not submitted any part of this work for another assessment.**

If your work involved research/studies (including surveys) on human participants, their cells or data, or on animals, you must have been granted ethical approval before the work was carried out, and any experiments must have followed these requirements. You must give details of this in the report, and list the ethical approval reference number(s) in the box below.

**My work did not involve human participants, their cells or data, or animals.**

*ECS Statement of Originality Template, updated August 2018, Alex Weddell aiofficer@ecs.soton.ac.uk*

# Chapter 1

# Introduction

## 1.1 Motivation

Abstract machines are mathematical models used to describe formal semantics of programming languages, as first-order transition systems. Such treatment of semantics have multiple useful properties:

- the ease of modelling complex features of the languages, such as continuations, control operators or threads
- realistic and efficient execution model of the language allowing to easily build an interpreter
- abstract point of view, useful when proving properties of the language or obtaining static analyses (Van Horn and Might (2010))

The formal definition of the semantics of programming languages allows one to reason about programs and prove their correctness properties, which is particularly useful when building safe and trustworthy software systems.

It was shown by Biernacka and Danvy (2007b) that abstract machines, rather than invented, can be mathematically derived from calculi with explicit substitutions. The authors introduced a formalism called $\lambda\hat{p}$ (Extended Calculus of Closures), which upon transformations leads to multiple known abstract machines for $\lambda$-calculus, as well considered its variants with control operators (Biernacka and Danvy (2007a)).

This project concerns the use of Biernacka & Danvy methodology to obtain correct-by-specification abstract machine interpreter for call-by-value Simply Typed $\lambda$-calculus. To do so, we use Agda, a dependently typed language, which thanks to the expressiveness of its type system, can double as a proof assistant to obtain correct-by-specification code, along with constructive proofs of the required properties.

## 1.2 Scope

The target language we focus on is $\lambda$-calculus which is a universal model of computation coming from formal logic, which captures key concepts of a programming language, such as functions, variables, and instantiating the functions with a given argument. $\lambda$-calculus is a backbone for modern functional programming languages, such as Haskell.

We particularly restrict our scope to the Simply Typed $\lambda$-calculus (STLC), which is less expressive than untyped lambda calculus. However, it has one interesting property; it is strongly normalising, that is every well typed program always terminates and never diverges.

As a basis, we consider call-by-value $\lambda\hat{p}$. From it, we derive a well-known CEK $\lambda$-calculus machine (Felleisen and Friedman (1987)) and prove its correctness properties. We build upon earlier work of Swierstra (2012) who considered call-by-name $\lambda\hat{p}$ to derive an executable Krivine machine.

## 1.3 Goals and contributions

The primary goals of this project are as follows:

1. To obtain an executable and correct CEK machine in a dependently typed programming language.
2. To show that such machine always terminates when given well-typed Simply Typed $\lambda$-Calculus terms.

The main contributions of this project are:

1. We extend Swierstra (2012) formalisation of $\lambda\hat{p}$ to call-by-value case, including the properties of head reduction.
2. We provide a proof of a Strong Normalisation property for call-by-value $\lambda\hat{p}$-calculus using Tait-style logical relation.
3. We provide a constructive proof of equivalence of the obtained CEK machine with call-by-value $\lambda\hat{p}$.

## 1.4 Overview and organisation

Chapter 2 sets the scene for this report by introducing the necessary background and discusses previous work in the area.

Chapter 3 shows how we extend Swierstra (2012) formalisation of $\lambda\hat{p}$ to a call-by-value case and obtain a single-step evaluator for it.

Chapter 4 provides lemmas on properties of head reduction in call-by-value $\lambda\hat{p}$ and their proofs.

Chapter 5 discusses our machine-checked proof of Strong Normalisation theorem for call-by-value $\lambda\hat{p}$ inspired by analogous Martin-Löf and Tait's proof for STLC.

Chapter 6 describes our formalisation of Danvy's refocusing transformation of call-by-value $\lambda\hat{p}$ single-step evaluator and the proof of equivalence with the single-step evaluator for $\lambda\hat{p}$. We also show the termination of the refocused version of the evaluator.

Chapter 7 describes our formalisation of the transformation leading to a CEK machine. We show the equivalence of obtained machine with earlier evaluators as well as its termination for the well-typed terms.

Chapter 8 discusses the engineering management aspect of this project.

Finally, Chapter 9 contains a summary of this work.

# Chapter 2

# Literature review and background

## 2.1 $\lambda$-calculus, De Bruijn formalism and intrinsic representation

We encourage the reader to familiarise themselves beforehand with the basics of $\lambda$-calculus. A good resource could be Pierce (2002).

In this project, we consider the variant of the Simple Typed $\lambda$-calculus having only primitive unit type (with no constructors), and an arrow (function type) between any two valid types. In general, any valid type is defined inductively by inference rules (Figure 2.1).

$$\text{Unit type}\ \frac{}{\bullet : Type}$$

$$\text{Arrow type}\ \frac{\begin{array}{c} a : Type \\ b : Type \end{array}}{a \Rightarrow b : Type}$$

FIGURE 2.1: Inference rules for valid types

When denoting the variables, we will use De Bruijn indices, which use numbers for variable identifiers (Figure 2.1). The value of the number tells to which lambda abstraction a variable is bound. Another advantage of such representation is the fact that $\alpha$-equivalent terms have the same representation, so it is easier to reason about substitution, without worrying about the problem of variable capture.

$$\lambda x.\lambda y.yx$$

$$\lambda\lambda 10$$

FIGURE 2.2: Example $\lambda$ term and its De Brujin representation

We will use capital Greek letters $\Gamma$ and $\Delta$ to denote type contexts. As we use De Bruijn representation of variables, we define typing contexts to be lists of types. The position in such list denotes the De Bruijn index of the given variable.

We define the variable lookup from the given type context as an inductive structure with two constructors (Figure 2.3). It is worth noticing that this structure is similar to the inductive definition of natural numbers. For example, lookup of variable with identifier `2` is constructed as S (S (Z))

$$\text{Zero (Z)}\ \frac{}{(\sigma :: \Gamma) \ni \sigma}$$

$$\text{Successor (S)}\ \frac{\Gamma \ni \sigma}{(\tau :: \Gamma) \ni \sigma}$$

FIGURE 2.3: Lookup operation

Instead of separately introducing syntax of terms and typing judgements for them, we can define both at the same time. Such a presentation of the rules is called intrinsic or Church-style (Wadler et al. (2020)). The only elements of our language are lambda abstraction, function application and variables (Figure 2.4). Thanks to the intrinsic representation it is not possible to build an ill typed term.

$$\text{Abstraction } (\lambda)\ \frac{(\sigma :: \Gamma) \vdash \tau}{\Gamma \vdash (\sigma \Rightarrow \tau)}$$

$$\text{Application } (\circ)\ \frac{\Gamma \vdash (\sigma \Rightarrow \tau) \quad \Gamma \vdash \sigma}{\Gamma \vdash \tau}$$

$$\text{Variable } (`)\ \frac{\Gamma \ni \sigma}{\Gamma \vdash \sigma}$$

FIGURE 2.4: Terms of the language

## 2.2 Curry-Howard isomorphism, intuitionistic logic and dependent types

Curry-Howard isomorphism is a well-known correspondence between type systems for $\lambda$-calculus and different kinds of logic. A great resource for the interested reader could be Wadler (2015).

STLC type system corresponds to sentences in first-order propositional logic. Dependent types are a more expressive system in which types can depend on values, and therefore types correspond to quantifiers known from propositional logic. Therefore, a language with such a type system, such as Agda (Bove et al. (2009)) can be used both as a programming language and a theorem prover.

In most cases (when excluding languages with first-class control) kinds of logic corresponding to different type systems are intuitionistic. Such systems exclude rules like double negation elimination or excluded middle, and they restrict the logic in a way that knowing that a given is not true, does not imply that it is true.

Therefore, they only allow constructive proofs and have computational interpretation, known as Brouwer-Heyting-Kolmogorov interpretation (Troelstra et al. (2011)). A paper by Moschovakis (2009) provides a compact overview and history of intuitionistic logic. It is worth noting, that Martin-Löf's intuitionistic type theory, being a backbone of Agda's type system, can be used as an alternative foundational theory for mathematics.

## 2.3 $\lambda p$ and $\lambda\hat{p}$ calculi

Calculus of Closures ($\lambda p$) is a formal system mediating between $\lambda$-calculus and abstract machines, where explicit substitutions are replaced with closures and substitution environments (Curien (1991)). Closures are simply $\lambda$ terms with accompanying substitution environment providing lexically scoped bindings for the free variables in the given term. The size of the substitution environment directly corresponds to the typing context of a given term, as every free variable in the typing context needs to have a corresponding substitution in the substitution environment. The only possible substitutions are other closures. So, a substitution environment is simply a list of closures for each De Bruijn variable of the given term.

The main downside of $\lambda p$ is the fact this calculus is not able to encompass single-step evaluators. To account for that Biernacka and Danvy (2007b) extended $\lambda p$ with the application of two closures and obtained a system known as $\lambda\hat{p}$ (Extended Calculus of Closures). The authors have shown that such a system is non-deterministic, confluent and equivalent to $\lambda$-calculus. Figure 2.5 presents the grammar of terms of $\lambda\hat{p}$, while Figure 2.6 describes the semantics of this language.

$$
\begin{array}{lcl}
\langle \mathrm{t} \rangle & \models & \langle \mathrm{i} \rangle \mid \langle \mathrm{t} \rangle \langle \mathrm{t} \rangle \mid \lambda \langle \mathrm{t} \rangle \\
\langle \mathrm{c} \rangle & \models & \langle \mathrm{t} \rangle [\langle \mathrm{s} \rangle] \mid \langle \mathrm{c} \rangle \langle \mathrm{c} \rangle \\
\langle \mathrm{s} \rangle & \models & \bullet \mid \langle \mathrm{c} \rangle \cdot \langle \mathrm{s} \rangle \\
\langle \mathrm{i} \rangle & \models & \mathit{zero} \mid \mathit{suc} \langle \mathrm{i} \rangle
\end{array}
$$

FIGURE 2.5: Grammar of $\lambda\hat{p}$ (Sourced from Biernacka and Danvy (2007b))

$$
\text{BETA}\ \frac{}{((\lambda t)[s])c \to t[c \cdot s]}
$$

$$
\text{LOOKUP}\ \frac{}{i[c_1 \ldots c_m] \to c_i}
$$

$$
\text{APP}\ \frac{}{(t_0 t_1)[s] \to (t_0[s])(t_1[s])}
$$

$$
\text{LEFT}\ \frac{c_0 \to c_0'}{(c_0 c_1) \to (c_0' c_1)}
$$

$$
\text{RIGHT}\ \frac{c_1 \to c_1'}{(c_0 c_1) \to (c_0 c_1')}
$$

$$
\text{SUB}\ \frac{c_i \to c_i'}{t[(c_1 \ldots c_i \ldots c_n)] \to t[(c_1 \ldots c_i' \ldots c_n)]}
$$

FIGURE 2.6: Semantics of $\lambda\hat{p}$ (Sourced from Biernacka and Danvy (2007b))

`BETA` rule corresponds to closing a lambda term and delaying the $\beta$-reduction by extending the substitution environment. `LOOKUP` simply peels the variable from substitution context, and `APP` takes the application of terms to the closure of application. `LEFT` and `RIGHT` reduce terms in the application.

Call-by-name (Normal order) reduction is obtained by restricting the semantics to the first four rules. To obtain call-by-value semantics (Applicative order), the first five rules are considered and the `RIGHT` rule is restricted to situations where the left-hand side of the application is a closed value. What is more, `BETA` rule is modified and only allows to substitute by closed values.

$$(\text{RIGHT})\frac{c_1 \to c_1'}{(vc_1) \to (vc_1')}$$

$$(\text{BETA})\frac{}{((\lambda t)[s])v \to t[v \cdot s]}$$

FIGURE 2.7: Modified `RIGHT` and `BETA` rules for call-by-value $\lambda\hat{p}$ semantics

## 2.4 Biernacka & Danvy framework

### 2.4.1 Evaluation contexts

The semantics of the language can be divided into two parts: the local part which tells how some closed terms can be atomically reduced (`BETA`, `LOOKUP` and `APP` rules), and the other one which controls where those reductions happen and in what order (`LEFT` and `RIGHT` rules).

The terms corresponding to `BETA`, `LOOKUP` and `APP` are called redexes. The reduction of redexes (also called contraction) happens in some evaluation context. We find an appropriate evaluation context according to `LEFT` and `RIGHT` rules (or only `LEFT` in case of call-by-name reduction).

A great way of looking at evaluation contexts is by analogy to the famous Zipper data structure introduced by Huet (1997). The closed terms are simply an inductive datatype, which can be seen as an abstract syntax tree. Reducing a term can be seen as a depth-first search through that tree until we reach a redex that can be contracted. Evaluation contexts allow us to remember which path we took when traversing the tree allowing us to reconstruct it later. Therefore a current closed term along with accompanying evaluation context forms a data structure similar to the Zipper. An evaluation context is often called continuation and can be seen as stack of an abstract machine. In the case of call-by-value reduction, we

have three constructors for evaluation contexts. `MT` is simply an empty evaluation context, meaning that we are at the root of the tree. We use `ARG`, when we pick the left route downward the tree to store element on the right-hand side of the original tree (the argument) as well as previous context. Figure 2.8 illustrates going to the left-hand side of the application of two closures $f$ and $x$.

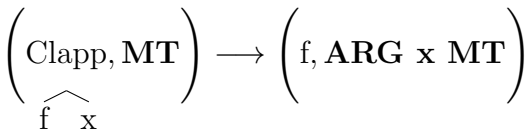


FIGURE 2.8: Traversing the AST to the left of the expression $fx$

Finally, we use `FN` when we pick the right subtree of the expression. We store the left-hand side (the function) and the path that leads us to it. As call-by-value is a left-to-right evaluation strategy, we first visit the left-hand side of the expression and then (if there is no redex, so the left-hand side is a value), we switch to the right side - see Figure 2.9. Single constructors are often called continuation frames


FIGURE 2.9: Visiting the left hand side first and then switching to the right side

(Felleisen and Friedman (1987)). In case of Krivine machine (Swierstra (2012)), we only have two types of continuation frames being `ARG` and `MT`, as call-by-name reduction in $\lambda\hat{\rho}$ has no `RIGHT` reduction rule.

### 2.4.2 Head reduction and refousing transform

Biernacka & Danvy introduced head reduction as a composition of three actions:

- Decompose - Find a redex and accumulate the corresponding evaluation context
- Contract - Reduce a redex in the given evaluation context

- Plug - Take a result of reducing a redex and rebuild the syntax tree according to the evaluation context

Head reduction is repeated until a non-reducible term is obtained. The key observation of Biernacka & Davy was to notice that instead of rebuilding the tree every time using a plug function and then decomposing it, a better idea is to go to the next evaluation contexts according to the given reduction order. Therefore a new function being a composition of decompose and plug is introduced and is called a refocusing function. The authors provided a way of deriving a refocusing function through the code transformation of decomposition function. An evaluator which uses refocusing is called a pre-abstract machine and can be further transformed. To do so, the authors introduced eval/apply and push/enter transforms that convert the pre-abstract machines to the abstract machine. Biernacka and Danvy (2007b) shows the derivation of Krivine machine from call-by-value $\lambda\hat{\rho}$.

What is more, Biernacka and Danvy (2007a) also considered variants of $\lambda\hat{\rho}$ with context-sensitive operators. The authors showed the derivation of the variant of the CEK machine with an abort operator, as well as an abstract machine for $\lambda\mu$ calculus.

## 2.5 Formalisation of abstract machines

Sieczkowski et al. (2011) provided a Coq formalisation of Biernacka & Danvy framework and showed the correctness of the transforms leading to abstract machines. This work was extended in Biernacka (2016), where the authors considered automating the process of derivation of an abstract machine from reduction semantics. Although proven to be correct, a downside of those contributions is the fact those formalisations are not executable, as abstract machines are defined as logical relations between the states. Therefore they cannot be used as a basis for correct-by-specification abstract-machine based interpreter. The authors did not consider proofs of termination for abstract machines for strongly normalising languages.

Completely independent from Biernacka & Danvy framework, Krupicka (2018) has implemented an executable SECD machine in Agda, basing on typed machine code. The author used the Delay monad (Abel and Chapman (2014)), which considers partiality and divergence as a monadic effect. The downside of Krupicka's

formalisation is the fact that an upper bound of maximal amount of reduction steps must be provided to work.

Swierstra (2012) basing on the research of Biernacka & Danvy used Agda to formalise call-by-name $\lambda\hat{p}$ and an executable evaluator for it. The author derived the refocused version of it, as well as the Krivine machine, showing the equivalence of all three evaluators. An interesting contribution of this study is proof of the termination of the executable Krivine machine, which relies on the strong normalisation property for call-by-value $\lambda\hat{p}$ proved using Tait-style logical relation. We rely on this research and adapt the ideas and proofs to call-by-value case. We directly use some part of this formalisation for the parts which are independent of the order of evaluation. All such parts are carefully acknowledged.

## 2.6 Bove-Capretta method

Agda (Bove et al. (2009)) is a total language, which means that each function must terminate. Agda uses a termination checker to verify that. However, the only functions that can pass the termination checker are the structurally recursive ones, that is functions where arguments in recursive calls are strictly smaller. When there are no guarantees about the size of passed arguments, such functions are called general recursive. Abstract machines and evaluators belong to this family.

Bove (2003) provided a solution to the problem of proving termination of generally recursive algorithms in total and dependently typed languages, and it is widely known as the Bove-Capretta method. The idea is to introduce a new datatype (a trace), which can be considered as a predicate that a given function terminates. The original function is transformed to be structurally recursive given the witness of the proof it terminates. The trace carries no computational value but allows to satisfy the requirements of the termination checker. Therefore, proving the termination of the modified version of the function is performed by showing the inhabitance of trace datatype (often referred to as Bove-Capretta datatype).

Altenkirch and Chapman (2009) successfully used this method to formalise Strong Normalisation theorem for System T. This approach directly inspired Swierstra (2012) and his proof of Strong Normalziation for call-by-name $\lambda\hat{p}$. We will rely on both those contributions when introducing our normalisation proof.

# Chapter 3

# Call-by-value $\lambda\hat{p}$ evaluator

This chapter outlines the design and implementation of a single-step evaluator for earlier introduced $\lambda\hat{p}$-calculus under call-by-value reduction, obtained by extending Swierstra (2012) call-by-name $\lambda\hat{p}$ evaluator. We provide definitions of the used datatypes (usually through sequent-style rules) and high-level sketches of the main functions, theorems and lemmas.

## 3.1 Terms, closed terms and substitution environments

As we consider Simply Typed $\lambda$-calculus our formalisation contains the definition of types and intrinsically typed $\lambda$-terms (Figures 2.1 and 2.4) adapted from Wadler et al. (2020).

Both in call-by-name and call-by-value cases we have the same definition of closed terms. Either it is a closure of a term, along with substitution environment or it is an application of closures. Substitution environment (denoted `Env` $\Gamma$, where $\Gamma$ is a type environment) for some term is simply a list of closed terms for each of the free variables of that term. `Nil` stands for empty list and constructor (denoted with mixfix $\cdot$) appends an element to the given list. We use mutually recursive definitions from Swierstra (2012) - Figures 3.1 and 3.2

$$\text{Closure}\ \frac{\Gamma \vdash u \quad \texttt{Env}\ \Gamma}{\texttt{Closed u}}$$

$$\text{Closure application (Clapp)}\ \frac{\texttt{Closed (u} \Rightarrow \texttt{v)} \quad \texttt{Closed u}}{\texttt{Closed v}}$$

FIGURE 3.1: Closed terms of the $\lambda\hat{p}$ (sourced from Swierstra (2012))

$$\text{Nil (Empty substitution environment)}\ \frac{}{\texttt{Env []}}$$

$$\text{Constructor}\ \frac{\texttt{Closed u} \quad \texttt{Env}\ \Gamma}{\texttt{Env (u::}\Gamma\texttt{)}}$$

FIGURE 3.2: Substitution environments (Env) definition (sourced from Swierstra (2012))

## 3.2 Redexes and contraction

In the considered language, closures containing lambda abstractions are the only terms that cannot be reduced further. Value datatype simply takes a closure and a predicate that term in the given closure is a lambda abstraction.

$$\text{Val}\ \frac{\texttt{(c :}\ \ \texttt{Closed}\ \sigma\texttt{)} \quad \texttt{isVal c}}{\texttt{Value}\ \sigma}$$

There are three possible redexes in the $\lambda\hat{p}$, which come from the small-step reduction rules shown in Figure 2.6. The definition of the datatype representing is shown in Figure 3.3. It mostly corresponds to definition from Swierstra (2012), however, the main difference is that we restrict $\beta$-reduction to substitute only by values of the language.

Redexes are simply a different representation of particular closures, therefore each redex can be mapped back to its underlying closure. Function performing this mapping is listed in Figure 3.4.

$$\text{Lookup}\ \frac{\Gamma \ni \sigma \quad \texttt{Env}\ \Gamma}{\texttt{Redex}\ \sigma}$$

$$\text{App}\ \frac{\Gamma \vdash (\sigma \Rightarrow \tau) \quad \Gamma \vdash \sigma \quad \texttt{Env}\ \Gamma}{\texttt{Redex}\ \tau}$$

$$\text{Beta}\ \frac{(\sigma :: \Gamma) \vdash \tau \quad \texttt{Env}\ \Gamma \quad \texttt{Value}\ \sigma}{\texttt{Redex}\ \tau}$$

FIGURE 3.3: Possible redexes in $\lambda\hat{p}$

```
fromRedex : ∀ {u}
              → Redex u
              → Closed u

fromRedex (Lookup x env) = Closure (‘ x) env
fromRedex (App f arg env) = Closure (f ∘ arg) env
fromRedex (Beta body env (Val c _)) = Clapp (Closure (λ body) env) c
```

FIGURE 3.4: Function mapping redex to its underlying closure

Having defined redexes, we can define the contraction function, which maps each redex to closure after single-step reduction according to the rules from Figure 2.6.

```
contract : ∀ {u}
         → Redex u
         → Closed u
contract (Lookup i env) = env ! i
contract (App f x env) = Clapp (Closure f env) (Closure x env)
contract (Beta body env (Val c x)) = Closure body (c · env)
```

FIGURE 3.5: Contraction funtion

Contracting `Lookup` redex uses `!` helper function which returns i-th closure from the given substitution environment. Contracting `Beta` simply outputs the body of the lambda expression, where the substitution environment is extended with an argument given to the stored function. As for `App` redex, the result is closure

application of closures created from the left-hand-side and right-hand-side of the given expression.

## 3.3 Evaluation contexts and hole semantics

We can now introduce the inductive datatype representing evaluation contexts (Figure 3.6) for the call-by-value order. Similarly to Swierstra (2012) the contexts are indexed by the pair of types. The first type represents the type of closure we are currently looking at and the right type represents the original type of the expression we are evaluating.

$$\text{MT}\ \frac{}{\texttt{EvalContext u u}}$$

$$\text{ARG}\ \frac{\texttt{Closed u} \qquad \texttt{EvalContext v w}}{\texttt{EvalContext (u} \Rightarrow \texttt{v) w}}$$

$$\text{FN}\ \frac{\texttt{Value (a} \Rightarrow \texttt{b)} \qquad \texttt{EvalContext b c}}{\texttt{EvalContext a c}}$$

FIGURE 3.6: Inductive definition of typed evaluation contexts

Having defined the evaluation contexts, we can state `plug` function from Biernacka and Danvy (2007b) which takes a closure and corresponding evaluation context are recreates the original term before traversals (Figure 3.7).

```
plug : ∀ {u v}
     → EvalContext u v
     → Closed u
     → Closed v

plug MT f = f
plug (ARG x ctx) f = plug ctx (Clapp f x)
plug (FN (Val closed isval) ctx) x = plug ctx (Clapp closed x)
```

FIGURE 3.7: Listing of the plug function

## 3.4 Decomposition

W are left with introducing the decomposition function. A correct decomposition of any valid closed term of $\lambda\hat{p}$ is either a value (as there are no more redexes to be found) or a redex with accompanying evaluation context that was accumulated before finding the redex. Following Swierstra (2012) we can introduce a dependent datatype describing a decomposition of a given closed term - see Figure 3.8

$$\text{Val}\ \frac{\begin{array}{c}\texttt{(body : (u :: } \Gamma\texttt{)} \vdash \texttt{v)} \\ \texttt{(env : Env } \Gamma\texttt{)}\end{array}}{\texttt{Decomposition (Closure (}\lambda\texttt{ body) env)}}$$

$$\text{Redex}\times\text{Context}\ \frac{\begin{array}{c}\texttt{(r : Redex u)} \\ \texttt{(ctx : EvalContext u v)}\end{array}}{\texttt{Decomposition (plug ctx (fromRedex r))}}$$

FIGURE 3.8: Valid decompositions of a closed term

This datatype directly corresponds to the `Value + (Redex×Context)` type from Biernacka and Danvy (2007b). Below, we restate the definition of the decomposition function from Biernacka and Danvy (2007b) but using Agda syntax. The heart of the decomposition function are two mutually recursive functions `decompose'` and `decompose_aux'` and Figure 3.9 contains a listing of them.
`decompose'` is used to decompose a closed term in the arbitrary context. If it contains an variable lookup or a closure of the application it returns the appropriate redex. If the given closed term is closure application, the function simply calls itself to traverse to the left, while saving the right-hand side in the evaluation context. Finally, if a given closure is a value it calls the `decompose_aux'` function, which decides what to do depending on the accumulated evaluation context.

If it is an empty context, we have found a value of the language. If we came from the left-hand side and reached the value, we should switch to the right-hand side and save the evaluated left-hand side in the `FN` frame. If we came from the right-hand side, it means that we have found the application of two values which forms redex corresponding to $\beta$-reduction.

Finally, the main decomposition function calls `decompose'` with an empty context. It is worth noticing the terminating pragma, which means that Agda termination

```
{-# TERMINATING #-}
mutual
  decompose' : ∀ { u v}
             → (ctx : EvalContext u v)
             → (c : Closed u)
             → Decomposition (plug ctx c)

  decompose' ctx (Closure (‘ i) env) =
    Redex×Context (Lookup i env) ctx

  decompose' ctx (Closure (λ body) env) =
    decompose'_aux ctx (body) env
  decompose' ctx (Closure (f ∘ x) env) =
    Redex×Context (App f x env) ctx
  decompose' ctx (Clapp f x) =
    decompose' (ARG x ctx) f

  decompose'_aux : ∀ { a b w Γ}
                 → (ctx : EvalContext (a ⇒ b) w)
                 → (body : (a :: Γ) ⊢  b)
                 → (env : Env Γ)
                 → Decomposition (plug ctx (Closure (λ body) env))

  decompose'_aux MT body env = Val body env
  decompose'_aux (ARG arg ctx) body env =
      decompose' (FN (Val (Closure (λ body) env) tt) ctx) arg
  decompose'_aux (FN (Val (Closure (λ x) env2) proof) ctx) body env =
    Redex×Context (Beta x env2 (Val (Closure (λ body) env) tt)) ctx

  decompose : ∀ {u}
             → (c : Closed u)
             → Decomposition c

  decompose c = decompose' MT c
```

FIGURE 3.9: Listing of a decomposition function

checker cannot confirm that the function is terminating, due to its non-structurally recursive structure. This topic is discussed more broadly in chapter 8.2.

## 3.5 Small-step evaluator

Having defined decomposition, contraction and plugging we can give a function that takes any closure and performs a reduction of redex at its' head position. We look at the decomposition of a given term. If it is a value, we are done. Otherwise, we contract the found redex and rebuild the closed term, by plugging the contracted redex using earlier found valuation context.

This definition is common to all machines stated using Biernacka & Danvy approach, and therefore we rely on the implementation from Swierstra (2012) - see Figure 3.10.

```
headReduce : ∀ {u}
           → Closed u
           → Closed u

headReduce c with decompose c
headReduce .(Closure (λ body) env) | Val body env =
  Closure (λ body) env
headReduce .(plug ctx (fromRedex redex)) | Redex×Context redex ctx =
  plug ctx (contract redex)
```

FIGURE 3.10: Head reduction function - sourced from Swierstra (2012)

# Chapter 4

# Head reduction properties

After introducing the single-step head reduction evaluator for call-by-value $\lambda\hat{p}$, we can internally verify it, by proving its properties, which will become handy later when proving the termination of the well-typed programs.

In this chapter, we will mostly introduce equational properties relying on inductive proofs by reflection. Proofs by reflection in type theory generally rely on the principle that if two sides of equality evaluate to the same result then they both are indeed equal (Bove et al. (2009)).

## 4.1 Plugging properties

**Lemma 4.1.** *For any types $u$ and $v$, let $r$ denote redex of type $u$ and let ctx denote an EvalContext parametrised by types $u$ and $v$. We have that* `plug ctx (fromRedex r)` *is a closed term, which is not a value and therefore can be further reduced.*

*Proof.* We prove this by relying on two helper lemmas. By case splitting on all possible redexes, we know that for any redex $r$ closed term obtained by `fromRedex r` is not a value. Then, by considering all possible EvalContext constructors we prove that plugging non-value closed term into any EvalContext always yields a non-value. Combining those two lemmas, we prove the needed property. □

## 4.2 Decomposition and plugging properties

Given a closed term and its context, decomposition of the reconstructed term using `plug` is the same as continuing the traversal in a given context using `decompose'`. Therefore, two steps of head reduction can be replaced by a single function, which takes fewer steps. Function with such property could work as refocusing function, simplifying the evaluator.

**Theorem 4.2** (Refocusing theorem)**.** *For any types* `u` *and* `v` *let* `c` *denote closed term of type* `u` *and let* `ctx` *denote an EvalContext parametrised by types* `u` *and* `v`*. We have* `decompose (plug ctx c) ≡ decompose' ctx c`

*Proof.* By induction. We consider three cases of different `EvalContext` constructors. For `MT` proofs goes by reflection. For `ARG` and `MT` cases we rewrite by induction hypothesis and prove the desired equality by reflection. □

We introduce a simple helper lemma from Swierstra (2012) that will become useful later, saying that decomposition a closed term of some redex r in some context is that redex in the original context.

**Lemma 4.3.** *For all* `u v`*, let* `r` *denoted redex t of type* `u`*, and let* `ctx` *denote an EvalContext parametrised by types* `u` *and* `v`*. We have the equality* `decompose' ctx (fromRedex r) ≡ Redex×Context r ctx`

*Proof.* Trivially provable by reflection upon splitting into all cases for EvalContext and all possible redexes. □

We can use that lemma to prove a stronger property adapted from Swierstra (2012). Head reduction of a closed term of a redex in some context corresponds to the closed term of the contracted form of that redex plugged into the original context.

**Lemma 4.4.** *For all* `u v`*, let* `r` *denoted redex t of type* `u`*, and let* `ctx` *denote an EvalContext parametrised by types* `u` *and* `v`*. We have the equality* `headReduce (plug ctx (fromRedex r)) ≡ plug ctx (contract r)`

*Proof.* We rewrite the equation by appealing to the Lemma 4.3 and Theorem 4.2 and then the original statement stands by the reflection. □

## 4.3 Leftmost innermost head reduction properties

The strong normalisation theorem proofs rely on properties of left-to-right call-by-value evaluation of application of closures. We introduce three properties:

**Lemma 4.5** (Left hand side evaluation lemma)**.** *For any types* `u` *and* `v` *let* `f` *denote a closed term of type* `u ⇒ v`*, let* `x` *denote a closed term of type* `u` *and* `fx` *denote a closed term of type* `v` *such that* `Clapp f x ≡ fx` *and* `f` *is not a value. We have the equality* `headReduce fx ≡ Clapp (headReduce f) x`

**Lemma 4.6** (Right hand side evaluation lemma)**.** *For any types* `u` *and* `v` *let* `f` *denote a closed term of type* `u ⇒ v`*, let* `x` *denote a closed term of type* `u` *and* `fx` *denote a closed term of type* `v` *such that* `Clapp f x ≡ fx` *and* `f` *is a value. If x is not a value then* `headReduce fx ≡ Clapp f (headReduce x)`

**Lemma 4.7.** *For any typing contexts* Γ *and* Δ *and types* `a`*,* `b` *and* `v` *let* `body` *denote a term* `((a ⇒ b) :: Γ) ⊢ v)`*. Let* `env` *denote substitution environment for typing context* Γ *and let* `arg` *denote a term* `(a :: Δ) ⊢ b`*. Assume that* `env` *is a substitution environment for typing context* Δ*. We have the equality* `headReduce (Clapp (Closure (λ body) env) (Closure (λ arg) env))` `≡ Closure body ( (Closure (λ arg) env) · env)`

Lemma 4.7 trivially holds by reflection, as it simply describes the case of performing $\beta$-reduction. Lemmas 4.5 and 4.6 are more complex to prove. We extend the idea from Swierstra (2012) and his "backwards view" to a call-by-value case and we show that Lemma 4.5 is a dual of Lemma 4.6.

### 4.3.1 View on the last frame

Let `fx` denote `Clapp f x`. The key observation here is that if f is not a value, decomposing `f` yields the same redex as decomposing `fx`. It is also worth noticing that evaluation contexts obtained when decomposing them are similar. Taking the last `MT` of evaluation context obtained when decomposing `f` and replacing it with `ARG x MT` gives the evaluation context of `fx`. Figure 4.1 contains a listing of a function `snoc` (sourced from Swierstra (2012)), that given an evaluation context obtained when decomposing `f` and given `x` produces an evaluation context of decomposing `fx`.

```
snoc : ∀ {u v w}
     → EvalContext u (v ⇒ w)
     → (Closed v)
     → EvalContext u w

snoc MT u = ARG u MT
snoc (FN x ctx) u = FN x (snoc ctx u)
snoc (ARG x ctx) u = ARG x (snoc ctx u)
```

FIGURE 4.1: Snoc function - adapted from Swierstra (2012)

Similarly, in case when `f` is a value, decomposing `fx` and `x` yields the same redex, and replacing last `MT` of evaluation context of decomposition of `x` with `FN (Val f tt) MT` gives the evaluation context of decomposition of `fx`. We introduce `cons` (see Figure 4.2), a dual of `snoc` which given `f` and an evaluation context obtained by decomposing `x`, can produce an evaluation context of decomposition of `fx`.

```
cons : ∀ {a b c}
     → EvalContext a b
     → (Value (b ⇒ c))
     → EvalContext a c
cons MT val = FN val MT
cons (FN x ctx) val = FN x (cons ctx val)
cons (ARG x ctx) val = ARG x ((cons ctx val))
```

FIGURE 4.2: Cons function

Therefore, a key thing to look at when proving the lemmas 4.5 and 4.6 is to have a look at the last non-MT frame. Therefore, we introduce a data structure allowing to store the last non-MT frame of an evaluation context of given decomposition (see Figure 4.3).

$$\text{Nil}\ \frac{}{\texttt{SnocView u u MT}}$$

$$\text{Cons}\ \frac{\begin{array}{c}\texttt{(val : Value (b ⇒ c))}\\ \texttt{(ctx : EvalContext a b)}\end{array}}{\texttt{(SnocView (cons ctx val))}}$$

$$\text{Snoc}\ \frac{\begin{array}{c}\texttt{(x : Closed v)}\\ \texttt{(ctx : EvalContext u (v ⇒ w))}\end{array}}{\texttt{SnocView (snoc ctx x)}}$$

FIGURE 4.3: Snoc view datatype

Having that, we introduce a function (see Figure 4.4) which populates this datatype using any evaluation context. Hence, we can obtain the last non-MT frame of the given evaluation context (if it exists) or obtain `Nil` meaning that we tried to decompose an empty evaluation context.

```
viewSnoc : ∀ {u v}
           → (ctx : EvalContext u v)
           → SnocView ctx
viewSnoc MT = Nil
viewSnoc (FN x ctx) with viewSnoc ctx
viewSnoc (FN x .MT) | Nil = Cons x MT
viewSnoc (FN x .(cons ctx val)) | Cons val ctx = Cons val (FN x ctx)
viewSnoc (FN x .(snoc ctx z)) | Snoc z ctx = Snoc z (FN x ctx)
viewSnoc (ARG x ctx) with viewSnoc ctx
viewSnoc (ARG x .MT) | Nil = Snoc x MT
viewSnoc (ARG x .(cons ctx val)) | Cons val ctx = Cons val (ARG x ctx)
viewSnoc (ARG x .(snoc ctx z)) | Snoc z ctx = Snoc z (ARG x ctx)
```

FIGURE 4.4: Function allowing to populate `SnocView` for any evaluation context

Having defined those functions we can prove following two properties. Lemma 4.8 is adapted from Swierstra (2012), as it is present both in call-by-name and call-by-value cases. We also provide its dual (Lemma 4.9), which is needed for the call-by-value case.

**Lemma 4.8** (Snoc lemma)**.** *For all types* $u$*,* $v$ *and* $w$ *let* `ctx` *denote an evaluation context parametrised by types* $u$ *and* $(v \Rightarrow w)$*. Let* `x` *denote a closed term of type* $v$ *and let* `t` *denote a closed term of type* $u$*. In such case the following equality stands:* `plug (snoc ctx x) t` $\equiv$ `Clapp (plug ctx t) x`

*Proof.* Consider possible evaluation context constructors. For `MT` case, equality trivially stands by reflection. For the other two cases, we appeal to the inductive hypothesis and upon rewriting why arrive at desired equality by reflection. □

**Lemma 4.9** (Cons lemma)**.** *For all types* `a`*,* `b` *and* `c` *let* `ctx` *denote an evaluation context parametrised by types* `a` *and* `b`*. Let* `fn` *denote a closed term of type* `b ⇒ c` *and let* `t` *denote a closed term of type* `a`*. Finally, let* `p` *denote a witness of a proof that* `fn` *is a value. In such case the following equality stands:* `plug (cons ctx (Val fn p) ) t ≡ Clapp fn (plug ctx t)`

*Proof.* Similarly to proof of Lemma 4.8 □

### 4.3.2 Properties of closure application

When proving the head reduction lemmas, we need three simple properties of equalities of applications of closures. We adapt them from Swierstra (2012) as they are independent of the order of reduction.

**Lemma 4.10.** *For all types* `u`*,* `v` *let* `f` *and f' denote closed terms of type* `u ⇒ v`*, and let* `x` *and* `x'` *denote closed terms of type* `u`*. If* `Clapp f x ≡ Clapp f' x'`*, then* `f ≡ f'`

**Lemma 4.11.** *For all types* `u`*,* `v` *let* `f` *and f' denote closed terms of type* `u ⇒ v`*, and let* `x` *and* `x'` *denote closed terms of type* `u`*. If* `Clapp f x ≡ Clapp f' x'`*, then* `x ≡ x'`

**Lemma 4.12.** *For all types* `u`*, u' and* `v` *let* `f` *denote a closed terms of the type* `u ⇒ v` *and let* `f'` *denote a closed term of type* `u' ⇒ v`*. Let* `x` *denote a closed term of the type* `u`*, and let* `x'` *denote a closed term of the type* `u'`*. If* `Clapp f x ≡ Clapp f' x'`*, then* `u ≡ u'`

All those lemmas hold by a trivial reflection.

### 4.3.3 Head reduction lemmas

Finally, we can prove the lemmas 4.5 and 4.6.

*Left-hand side reduction lemma.* Consider all possible outcomes of `viewSnoc` on the evaluation context obtained by decomposing `fx`.

If it's `Nil`, we know that that evaluation context is simply just `MT`. Because `fx` is of arrow type, the redex found by decomposing it must be $\beta$-reduction redex. By appealing to lemmas 4.10 and 4.12 we can show that both closures in this application, that is `f` and `x` are values. But by assumption, `f` is not a value, which leads to falsity. We can infer anything from absurdity, therefore we are done.

If it's a `Cons`, then we know that evaluation context can be created by `cons` function and the last non-MT frame is `FN` which holds some value. Original term can be reconstructed by plugging x into that valuation context, and because last non-MT frame is FN, by appealing to lemmas 4.10 and 4.12 we can show that left hand side of `fx`, that is `f` is a value, which leads to a falsity. Again we use bottom-elimination to provide the witness of the proof.

Finally, if it's `Snoc` then we know that evaluation context can be created by `cons` function and the last non-MT frame is `ARG` which holds some closure. We rewrite the equation by lemmas 4.10, 4.11 and 4.12. By appealing to Lemma 4.8 for both closure form of redex from the decomposition, as well as its contracted form, we show by reflection that reduction of `f` in Clapp f x is indeed a reduction of `fx`. □

Now, we use a similar line of reasoning to prove the dual property. The line of reasoning mirrors the previous proof.

*Right-hand side reduction lemma.* Consider all possible outcomes of `viewSnoc` on the evaluation context obtained by decomposing `fx`.

If it's a `Nil` the case is identical to earlier proof. We have the $\beta$-reduction redex, and because of that, we know that `x` is a value. Therefore, we get to a falsity.

`Snoc` case, which earlier leads to an only non-absurd case, in this lemma leads to absurdity, similarly to `Cons` in the previous proof.

`Cons` case, which leads to the absurd case, now leads to the solution. Instead of appealing to lemma on plugging context obtained by snoc, we use Lemma 4.9 - its' dual, for plugging context obtained by snoc. Besides that, the structure of the proof is identical to the previous one. □

# Chapter 5

# Strong Normalisation property

Relying on the properties introduced in the previous chapter, we prove the strong normalisation property of $\lambda\hat{p}$ under call-by-value. We use it to prove that evaluator from Chapter 3 always terminates with a value, when given a well-typed term.

To do so, we will use the earlier mentioned Bove-Capretta method (Bove (2003)) to mechanise a variant of well-known proof of normalisation by evaluation due to Tait (1967) for call-by-value $\lambda\hat{p}$ calculus. We rely on earlier work on mechanisation of such proofs for System T (Altenkirch and Chapman (2009)) and $\lambda\hat{p}$ under call-by-name (Swierstra (2012))

## 5.1 Bove-Capretta trace

Function performing repeated head reduction until reaching a value is not structurally recursive. To pass the termination checker, we introduce a Bove-Capretta datatype (see Figure 5.1) and make the evaluation function structurally recursive given the trace. This datatype can be seen as a list of recursive calls being made by a function. Such datatype carries no computational value but assists the termination checker to show that every recursive call is made with structurally smaller arguments. More discussion on the collapsibility of Bove-Capretta datatypes is provided in Swierstra (2012).

Figure 5.2 shows the listing of the structurally recursive version of the evaluation function, which takes both the decomposition of the closed term, as well as an corresponding trace datatype to obtain a value of the $\lambda\hat{p}$. Both the trace datatype, as well as structurally recursive evaluation function are adapted from Swierstra

(2012), as they are universal to any order of reduction and apply both to call-by-name and call-by-value cases. The key difference here is the way of obtaining the Bove-Capretta trace for any well-typed closed-term, as call-by-value requires stronger properties to be proved.

$$\text{Done}\ \frac{\texttt{(body : (u :: } \Gamma\texttt{)} \vdash \texttt{v)} \quad \texttt{(env : Env } \Gamma\texttt{)}}{\texttt{Trace (Val body env)}}$$

$$\text{Step}\ \frac{\texttt{\{r : Redex u\}} \quad \texttt{\{ctx : EvalContext u v\}} \quad \texttt{Trace (decompose (plug ctx (contract r)))}}{\texttt{Trace (Redex×Context r ctx)}}$$

FIGURE 5.1: Definition of Bove-Capretta trace for repeated head reduction evaluator - adapted from Swierstra (2012)

```
iterate : ∀ {u : Type}
          → {c : Closed u}
          → (d : Decomposition c)
          → Trace d
          → Value u
iterate (Val body env) (Done .(body) .(env)) = Val (Closure (λ body) env) tt
iterate {c} {u} (Redex×Context r ctx) (Step step) =
    iterate  (decompose ( plug ctx (contract r))) step
```

FIGURE 5.2: Listing of `iterate` function which is structurally recursive - adapted from Swierstra (2012)

## 5.2 Defining a reducibility relation

It was noticed by Tait (1967) that straightforward induction over terms is not enough to prove the strong normalisation of well-typed terms of STLC. To account for that, he introduced the stronger notion of reducibility relation (for more detail see Girard et al. (1989)).

**Definition 5.1.** We define a set `Reducible u` (reducible closed terms of type `u`) by induction on the types.

- For `c` of type •, c belongs to `Reducible u`, if c is strongly normalising

- For `c` of type $a \Rightarrow b$, is reducible, if for any closed term `d` of type `a` which belongs to `Reducible a`, `Clapp c d` belongs to `Reducible b`

In the Agda formalisation of this relation, we use Bove-Capretta trace to express that the given term is strongly normalising. Following Swierstra (2012), in the case of arrow type, we define it as an arrow type, where the first part includes the witness that the left-hand side is strongly normalising.

To account for the fact, that we deal with $\lambda\hat{p}$, which has the notion of closure, we need to extend the approach from Tait (1967). We need to ensure, that every closure stored in the environment of some closure is also reducible. To do so, similarly to Swierstra (2012), we introduce `RedEnv` logical relation.

**Definition 5.2.** We define a set `RedEnv` (reducible environments) by induction on the constructors of substitution environment datatype.

- For the `Nil` constructor, an environment trivially belongs to `RedEnv`
- For the constructor case, an environment is reducible if the closure in the head position belongs to the `Reducible` relation of the appropriate type and the tail of the enviroment belongs to `RedEnv`

We can show that **CR1** property from Girard et al. (1989) trivially holds.

**Lemma 5.3.** *If a closed term* `t` *of type* `u` *belongs to* `Reducible u`*, then it is strongly normalising*

*Proof.* By induction on the type u. If u is a unit type, then by definition of reducibility relation we can obtain the witness (trace datatype) that it is strongly normalising. If `u` is an arrow type, then we project the first component of the cartesian pair to obtain the witness of the strong normalisation. □

We also can state that looking up from a reducible substitution environment, gives reducible closure.

**Lemma 5.4.** *For all typing contexts* $\Gamma$ *and any type* `u`*, let* `env` *denote substitution environment for typing context* $\Gamma$ *and let* `r` *denote a variable lookup* $\Gamma \ni$ `u` *. If* `env` *is reducible environment then closure obtained from that context using r (*`env ! r`*) belongs to* `Reducible u`

*Proof.* By straightforward induction on constructors of lookup type and reducible environment relation. If we have Z constructor, we can easily obtain reducibility proof of closure in the head position. If we have `S` we simply appeal to the induction hypothesis. □

## 5.3 Strong Normalisation Theorem and termination of evaluator

First, we introduce two helper functions (see Figure 5.3) operating on Bove-Capretta traces, which will become handy when proving further lemmas, as they relate trace datatype with head reduction. `step` takes a closed term and the trace built from decomposition of a head reduced version of this term and builds a bigger trace for the decomposition of the term before reduction. `unstep` does the opposite, and peels off one step from the trace, to obtain trace for the head reduced version.

```
step : ∀ {u}
     → (c : Closed u)
     → (t : Trace (decompose (headReduce c)))
     → Trace (decompose c)
step c trace with decompose c
step ._ trace | Val body env =  Done body env
step ._ trace | Redex×Context redex context
  = Step {r = redex} {ctx = context} trace

unstep : ∀ {u}
       → (c : Closed u)
       → (t : Trace (decompose c))
       → (Trace (decompose (headReduce c)))
unstep c trace with decompose ( c)
unstep ._ trace | Val body env =  trace
unstep .(plug context (fromRedex redex)) (Step trace)
  | Redex×Context redex context = trace
```

FIGURE 5.3: Listing of `step` and `unstep` functions

Having those helper functions defined we can prove the following theorem. We use $e \rightsquigarrow e'$ to denote `headReduce e = e'`.

**Theorem 5.5** (Preservation equivalence theorem)**.** *If $e \rightsquigarrow e'$, e is reducible if and only if e' is reducible*

First, we prove one side of the implication.

**Lemma 5.6** (Preservation lemma)**.** *If $e \rightsquigarrow e'$ and e' is reducible, then e is reducible*

*Proof.* First, we consider two cases depending on the type of `e`.

If `e` is of unit type, then we can use `step` to obtain the trace of `e` from the trace of `e'`.

If `e` is an arrow type, then we proceed by induction on the term structure. The witness of `e` being reducible will be a cartesian product of witness that `e` is strongly normalising and a function that given a reducible argument for the function gives the proof that application of argument to the function is reducible.

If the term is a lambda abstraction, then we return a cartesian product of `Done` trace and a function appealing to induction hypothesis using the proof that applying any reducible argument to `e'` is also reducible.

In the remaining cases, we use `step` function to obtain the trace, and similarly, we appeal to the induction hypothesis as earlier. In the case of closure application, we also appeal to Lemma 4.5 on the reduction of the left-hand side of closure application. □

Similarly, we prove the converse.

**Lemma 5.7** (Backwards preservation lemma)**.** *If $e \rightsquigarrow e'$ and e is reducible, then e' is reducible*

*Proof.* First, we consider two cases depending on the type of `e`.

If `e` is of unit type, then we can use `unstep` to obtain the trace of `e'`.

If `e` is an arrow type, then we proceed by induction on the term structure. The witness of `e` being reducible will be a cartesian product of witness that `e'` is strongly normalising and a function that given a reducible argument for the function gives the proof that application of argument to the function is reducible.

In each case, we obtain the witness that `e'` is strongly normalising, by the use of `unstep`. To obtain the necessary function to show reducibility of closure application we similarly appeal to the induction hypothesis as in the proof of the earlier lemma.

In the case when `e` is a closure application, we also appeal to Lemma 4.5 on the reduction of the left-hand side of the closure application. □

We arrive at the main property that leads to the desired strong normalisation proof. It is worth noticing that lemmas 5.8 and 5.9 are mutually inductive.

**Lemma 5.8** (Closure reducibility lemma)**.** *Closure of a well-typed term with a reducible environment is always reducible*

*Proof.* Proof by induction. We consider three cases for terms. In each of them, we appeal to preservation lemma (sese Lemma 5.6) to obtain the desired reducibility proof.

In a variable case, we simply use `deref` function to show that a closure obtained by looking up the variable is reducible. If this variable is reducible after looking up, then by Lemma 5.6 we know that one reduction step earlier, when it was a closure of variable lookup, it was also reducible.

In the application case, we use the induction hypothesis to show that closures of terms of the left and right-hand sides of the application are reducible. If closure of the left-hand side is reducible, then because it is a function, we can obtain the witness that application of closures of the left and right-hand side is reducible. Such application of closures is a head reduced version of the closure of application of terms. Therefore, we can again appeal to Lemma 5.6 to show that this closure is reducible.

Finally, let's consider the lambda abstraction case. In all previous cases, we knew what was the previous reduction rule we used (`LOOKUP` and `APP`) and that's why it was straightforward to prove it using Lemma 5.6. Here again, we would like to appeal to Lemma 5.6, but to use it we need to show that closure application of closure containing lambda abstraction and arbitrary closure is reducible. It is trivial in the call-by-name case (Swierstra (2012)), as we always perform a $\beta$-reduction regardless of the form of the closure on the right-hand side. Therefore, to deal with this case, we appeal to helper Lemma 5.9. □

**Lemma 5.9** (Right hand reducibility lemma)**.** *For all* $\Gamma$*,* $\sigma$ *and* $\tau$*, let* `body` *denote a term* `(σ :: Γ) ⊢ τ` *and let* `env` *denote a substitution environment for typing context* $\Gamma$*, which is reducible. Let* `x` *denote a closed term of type* $\sigma$*. Finally let* `trace` *denote a Bove-Capretta trace of decomposition of* `x`*. If* `x` *is reducible, then so is* `(headReduce (Clapp (Closure (λ body) env) x)`

*Proof.* Proof by induction on the structure of decomposition of `x`. If `x` is a value, then we know that the next reduction step would be a $\beta$-reduction. To show the reducibility, we appeal to Lemma 5.8 to show that closure of body of lambda abstraction with an environment extended by x (which is reducible by hypothesis) is also reducible.

If `x` is not a value, then we use the induction hypothesis until we reduce the right-hand side to the value. We show the reduciblity of head reduced version of `x` by the use of Lemma 5.7. We also use right hand side evaluation Lemma 4.6 to obtain the reducibility of `(headReduce (Clapp (Closure (λ body) env) x)` from the reducibility of `headReduce x` □

Finally, we show the desired property.

**Theorem 5.10** (Reducibility Theorem)**.** *Closure of any well-typed term with an empty environment is reducible*

*Proof.* Empty environment is trivially reducible. We use Lemma 5.8 to show that closure of any well typed term in an empty environment is always reducible. □

**Theorem 5.11** (Strong normalisation theorem for $\lambda\hat{p}$ calculus under call-by–value)**.** *Closure of any well-typed term with an empty environment is strongly normalising*

*Proof.* We use Theorem 5.10 to get a witness that the closure of any well-typed term with an empty environment is reducible. Then by Lemma 5.3 we show that this closure is strongly normalising. □

Finally, we can use the Bove-Capretta based iterated head-reduction evaluator and the witness of Theorem 5.11 to define an evaluation function - see Figure 5.4.

```
evaluate : ∀ {u}
         → (t : [] ⊢ u)
         → (Value u)
evaluate t = iterate (decompose (Closure t Nil)) (termination t)
```

FIGURE 5.4: Terminating evaluation function

# Chapter 6

# Refocusing transformation

In this chapter, we combine the results from chapters 4 and 5 to obtain simplified version of the evaluator, which is less expensive computationally. To do so, we rely on the refocusing theorem (see Theorem 4.2), stating that decomposing a closed term created by reconstruction through plugging, is the same as using `decompose'` to continue decomposition in the current context.

We follow Swierstra (2012) and show that refocused version of evaluator is equivalent to the original evaluator from previous chapters. Moreover, we adapt the proofs from the previous chapter to show that simplified evaluator is also terminating.

## 6.1 Modified Bove-Capretta trace

We aim to introduce the refocusing function which replaces the composition of `decompose` and `plug`. In Theorem 4.2 we showed that such function exists and it is `decompose'`. Therefore we define `refocus` (see listing 6.1) to be `decompose'`

```
refocus : ∀ {u v}
        → (ctx : EvalContext u v)
        → (c : Closed u)
        → Decomposition (plug ctx c)

refocus = Redex.decompose'
```

FIGURE 6.1: Definition of `refocus` function

Now, we can trivially show by reflection that following property holds.

**Lemma 6.1.** *For any types* `u` *and* `v` *let* `c` *denote closed term of type* `u` *and let* `ctx` *denote an EvalContext parametrised by types* `u` *and* `v`*. We have* `refocus ctx c ≡ decompose (plug ctx c)`

So, before we define refocused version of evaluator which terminates, we need to create new Bove-Capretta trace datatype which takes `refocus` into account - see listing 6.2. Knowing the refocusing property, we can easily show that given a trace

$$\text{Done}\ \frac{\texttt{(body : (u :: } \Gamma\texttt{)} \vdash \texttt{v)} \quad \texttt{(env : Env } \Gamma\texttt{)}}{\texttt{Trace (Val body env)}}$$

$$\text{Step}\ \frac{\texttt{\{r : Redex u\}} \quad \texttt{\{ctx : EvalContext u v\}} \quad \texttt{Trace (refocus ctx (contract r))}}{\texttt{Trace (Redex×Context r ctx)}}$$

FIGURE 6.2: Definition of Bove-Capretta trace for refocused evaluator - adapted from Swierstra (2012)

of a plugged and decomposed term, we can obtain the trace of the same term upon refocusing.

**Lemma 6.2.** *For any types* `u` *and* `v` *let* `t` *denote closed term of type* `u` *and let* `ctx` *denote an EvalContext parametrised by types* `u` *and* `v`*. Existence of* `Trace (decompose (plug ctx t))`*, implies the existence of* `Trace (refocus ctx t)`

*Proof.* By Lemma 6.1, we can can conclude that `Trace (decompose (plug ctx t))` is equivalent to `Trace (refocus ctx t)`. □

As a consequence of this lemma, we can prove a stronger property.

**Lemma 6.3** (Refocusing trace lemma)**.** *For all* `u`*, let* `t` *denote a closed term of type* `u`*. The existence of a trace of head reduction evaluator for* `decompose t`*, implies the existence of a trace of refocusing evaluator for* `decompose t`

*Proof.* By induction on the head reduction evaluator trace. If `t` is a value then both traces are identical, as they denote the `Done` case. If decomposing `t` yields a redex and a context, we can appeal to Lemma 6.1 to rewrite the head reduction trace to refocusing trace at the first element of trace. Then, we appeal to induction hypothesis for the rest of the trace. □

## 6.2 Modified evaluator

We earlier proved the Strong Normalisation property for $\lambda\hat{p}$, meaning that for every well-typed term we could obtain the head reduction trace. By applying Lemma 6.3 we can now obtain a refocusing trace for any well-typed term. Figure 6.3 shows the listing of the structurally recursive refocused evaluator, which takes a Bove-Capretta trace to run. We also provide a terminating evaluation function, which appeals to Strong Normalisation property (5.11) and trace Refocusing Trace Lemma (6.3) to obtain a needed trace for any well-typed term.

```
iterate : ∀ {u}
        → {c : Closed u}
        → (d : Decomposition c)
        → Trace d
        → Value u

iterate (Val body env) (Done .(body) .(env)) =
  Val (Closure (λ body) env) tt
iterate (Redex×Context r ctx) (Step step) =
  iterate (refocus ctx (contract r)) step

evaluate : ∀ {u}
         → (t : [] ⊢ u)
         → Value u
evaluate t = iterate (refocus MT (Closure t Nil)) (termination t)
```

FIGURE 6.3: Terminating refocused evaluator - adapted from Swierstra (2012)

## 6.3 Correctness guarantees

Finally, we can show that refocused evaluator gives the same result as the original head reduction evaluator. We start with a helper lemma.

**Lemma 6.4.** *For all types* `u`*, let* `t` *denote a closed term of type* `u`*. Given a head reduction evaluator trace and refocused evaluator trace of decomposition of* `t`*,* `iterate` *function of head reduction evaluator produces the same result as* `iterate` *function of the refocused evaluator.*

*Proof.* By induction on the decomposition of `t`. If `t` is a value then desired property trivially holds by reflection. If decomposition of `t` yields a redex and the context, then we appeal to Lemma 6.1 and show equivalence of current step, and then appeal to inductive hypothesis for the rest of the both traces. □

We use this lemma to show a slightly stronger property on the equivalence of results of both evaluators when the refocused evaluator obtains its first configuration by refocusing a given closed term in the empty context.

**Theorem 6.5.** *For all type* `u`*, let* `t` *denote a closed term of type* `u`*. Let* `t1` *denote refocusing trace of configuration obtained by refocusing t in empty evaluation context. Let* `t2` *denote head reduction trace of decomposition of* `t`*. Then* `iterate` *function of refocused evaluator when given a configuration obtained by refocusing* `t` *in the empty context, yields the same result as* `iterate` *of head reduction evaluator when given a decomposition of* `t`*.*

*Proof.* The appeal to Lemma 6.1 to show that configuration obtained by refocusing a term in the empty context is the same as its decomposition. Then we use Lemma 6.4 to show that results obtained by both iterate functions are equivalent. □

And finally, we obtain the central correctness property of the refocused version of the evaluator.

**Corollary 6.6.** *For all types* `u` *let* `t` *denote a well-typed term term* `[] ⊢ u`*. Then* `evaluate` *function of the refocused evaluator yields the same result for* `t` *as* `evaluate` *function of the head reduction evaluator.*

*Proof.* We use Theorem 6.5 to show that repeated `iterate` functions yield the same result. We use Strong Normalisation property (Theorem 5.11) to obtain the trace for head reduction evaluator. Finally, we appeal to Refocusing Trace Lemma 6.3 to obtain the trace for refocused evaluator. □

# Chapter 7

# CEK machine

In this chapter we further simplify the evaluator from Chapter 6 and obtain the CEK machine from Felleisen and Friedman (1987).

Following Danvy and Nielsen (2004) and Biernacka and Danvy (2007b) we combine `refocus` and `contract` functions and factor out the environment from the closures, obtaining a state transition function.

Relying on results from Chapter 5 we introduce a Bove-Capretta trace for the CEK machine, show that it can be obtained from refocusing trace, and finally give correctness and termination guarantees for the obtained machine.

## 7.1 Correct environments, closures and lookup

Machine state consisting only of terms, environments and evaluation contexts can only store closures, and there is no way of representing the closure application. Therefore, in this section, we strictly follow Swierstra (2012) to introduce logical predicates which ensure a lack of closure applications in any parts of the state of the machine. What is more, we introduce properties of manipulating the correct closures and environments. In general, a closed term is valid only if it is a closure and has a valid environment. Valid environments are those which are empty or only store valid closures. Finally, valid evaluation contexts are those that only store closure applications in their frames or are the empty context. Listing 7.1 (adapted from Swierstra (2012)) shows the Agda definitions of valid closures, environments and evaluation contexts.

```
mutual
  isValidClosure : ∀ {u}
                 → Closed u
                 → Set
  isValidClosure (Closure x env) = isValidEnv env
  isValidClosure (Clapp closure closure) = ⊥

  isValidEnv : ∀ {Γ}
             → Env Γ
             → Set
  isValidEnv Nil = ⊤
  isValidEnv (x · env) = (isValidClosure x × isValidEnv env )

isValidContext : ∀ {u v}
               → EvalContext u v
               → Set
isValidContext MT = ⊤
isValidContext (FN (Val (Closure (λ body) env) proof) context)
  = ( isValidEnv env × isValidContext context )
isValidContext (ARG (Closure x env) context) =
  ( isValidEnv env × isValidContext context )
isValidContext (ARG (Clapp _ _) context) = ⊥
```

FIGURE 7.1: Valid closures and environments - sourced from Swierstra (2012)

Having those predicates defined, we can follow Swierstra (2012) further and introduce functions for extraction of typing contexts, environments and terms from valid closures - see Figure 7.2.

```
getContext : ∀ {u}
           → Σ (Closed u) (isValidClosure)
           → Context
getContext (Closure {Γ } _ _ , _) = Γ

getEnv : ∀ {u}
       → (c : Σ (Closed u) (isValidClosure))
       → Env (getContext c)
getEnv (Closure _ env , _) = env

getTerm : ∀ {u}
        → (c : Σ (Closed u) isValidClosure)
        → (getContext c) ⊢ u
getTerm (Closure x _ , _) = x
```

FIGURE 7.2: Type safe deconstruction of closures - sourced from Swierstra (2012)

Figure 7.3 introduces a type-safe lookup, which for a valid environment is guaranteed to return a valid closure.

```
lookup : ∀ {u Γ}
       → Γ ∋ u
       → (env : Env Γ)
       → isValidEnv env
       → Σ (Closed u) isValidClosure
lookup Z (Closure x env · _) (fst , _) = (Closure x env) , fst
lookup (S ref) (x · env) (_ , snd) = lookup ref env snd
```

FIGURE 7.3: Type safe lookup - sourced from Swierstra (2012)

Such lookup satisfies the two properties, which can be found below.

**Lemma 7.1.** *For all type environents* Γ *and any type* `u` *let* `env` *denote a substitution environment for typing context* Γ*. Let* `p` *denote a witness that* `env` *satisfies* `isValidEnv` *predicate. Let* `i` *denote a variable lookup* Γ ∋ `u`*. Then, the result of closure lookup using* `env ! i` *is the same as* `Closure (getTerm (lookup i env p)) (getEnv (lookup i env p))`

*Proof.* Proof by induction. In the base case, when `i` is a lookup from the first position, the statement holds by reflection. In the inductive case, when `i` is a successor, we appeal to the induction hypothesis. □

**Lemma 7.2.** *For all type environments* Γ *and any type* `u` *let* `env` *denote a substitution environment for typing context* Γ*. Let* `p` *denote a witness that* `env` *satisfies*

*`isValidEnv` predicate. Let `i` denote a variable lookup* $\Gamma \ni$ *`u`. Then an environment obtained by `getEnv (lookup i env p)` satisfies `isValidEnv` predicate.*

*Proof.* By induction. In the base case, when `i` is a lookup from the first position, then we use the first component of cartesian pair constituting i, to obtain witness that the environment of the looked up closure is valid.

In the inductive case, we appeal to the induction hypothesis using the second component of the cartesian pair constituting `i`. □

## 7.2 Bove-Capretta trace and state transition function for CEK machine

As earlier, we can introduce a Bove-Capretta trace for the next version of the evaluator. The main difference, between previous traces and CEK machine trace, is the fact that because of inlining contraction with refocusing, we have different kinds of reduction steps rather than having just `Step`.

Therefore each constructor of the new trace corresponds to a different state transition of the CEK machine. The only common thing for all trace datatypes is `Done` constructor, which corresponds to the final state of each of the evaluators. A full Bove-Capretta trace for a given term will be simply a call graph of CEK machine transitions when evaluating the given term. Figure 7.4 shows the definition of the Bove-Capretta trace for the CEK machine.

Obtained machine corresponds with the CEK machine presented in Felleisen and Friedman (1987) and Van Horn and Might (2010), but has one interesting difference. Our CEK machine does not have a closure making step and its control language are simply well-typed terms of STLC. Rules performing transition upon having closure in the control part of the state are composed with the closure making step. What is more, our variant of the CEK machine uses De Bruijn indices, as opposed to having named variables, like in Felleisen and Friedman (1987). Therefore obtained machine corresponds to the presentation known from Biernacka and Danvy (2007b) and Biernacka and Danvy (2007a).

$$\text{Done}\ \frac{\{\texttt{env : Env } \Gamma\}\ \texttt{(body : (v :: } \Gamma\texttt{)} \vdash \texttt{u)}}{\texttt{Trace } (\lambda\ \texttt{body) env MT}}$$

$$\text{Lookup}\ \frac{\{\texttt{ctx : EvalContext u v}\}\{\texttt{env : Env } \Gamma\ \}\ \texttt{(i : } \Gamma \ni \texttt{u)(p : isValidEnv env)}\quad \texttt{Trace (getTerm (lookup i env p)) (getEnv (lookup i env p)) ctx}}{\texttt{Trace (`` i) env ctx}}$$

$$\text{Left}\ \frac{\{\texttt{env : Env } \Gamma\}\{\texttt{ctx : EvalContext v w}\}\ \texttt{(f : } \Gamma \vdash \texttt{(u} \Rightarrow \texttt{v) )(x : } \Gamma \vdash \texttt{u)}\quad \texttt{Trace f env (ARG (Closure x env) ctx)}}{\texttt{Trace (f} \circ \texttt{x) env ctx}}$$

$$\text{Right}\ \frac{\{\texttt{env : Env } \Gamma\}\{\texttt{ctx : EvalContext v w}\}\ \texttt{(env2 : Env } \Delta\texttt{)(body : (u :: } \Delta\texttt{)} \vdash \texttt{v)(x : } \Gamma \vdash \texttt{u)}\quad \texttt{Trace x env (FN (Val (Closure (}\lambda\ \texttt{body) env2) tt) ctx)}}{\texttt{Trace (}\lambda\ \texttt{body) env2 (ARG (Closure x env) ctx)}}$$

$$\text{Beta}\ \frac{\{\texttt{env : Env } \Gamma\}\texttt{(ctx : EvalContext u w)(argBody : (a :: } \Delta\texttt{)} \vdash \texttt{b)}\ \texttt{(argEnv : Env } \Delta\texttt{)(body : ( (a} \Rightarrow \texttt{b) :: } \Gamma\texttt{)} \vdash \texttt{u)}\quad \texttt{Trace body (Closure (}\lambda\ \texttt{argBody) argEnv} \cdot \texttt{env) ctx}}{\texttt{Trace (}\lambda\ \texttt{argBody) argEnv (FN (Val (Closure (}\lambda\ \texttt{body) env) tt) ctx)}}$$

FIGURE 7.4: CEK machine trace datatype

Having defined the trace, we can build a structurally recursive function taking a well-typed term and corresponding trace to compute a result of the term. Figure 7.6 shows listing of the CEK transition function. For readability, Figure 7.5 provides a simplified presentation of the obtained transitions rules.

$$\texttt{(f} \circ \texttt{x , env , kont)} \rightsquigarrow \texttt{(f , env , ARG <x , env> kont)}$$

$$\texttt{(}\lambda\ \texttt{body , env2 , ARG <x , env> kont)} \rightsquigarrow \texttt{(x , env , FN <}\lambda\ \texttt{body , env2> kont)}$$

$$\texttt{(}\lambda\ \texttt{arg , env2 , FN <}\lambda\ \texttt{body , env> kont))} \rightsquigarrow \texttt{(body , <}\lambda\ \texttt{arg , env2>} \cdot \texttt{env , kont)}$$

$$\texttt{(`` i , env , kont)} \rightsquigarrow \texttt{(env[i] , env , kont)}$$

$$\texttt{(}\lambda\ \texttt{body , env , MT)} \rightsquigarrow \texttt{Done}$$

FIGURE 7.5: Obtained CEK transition rules

```
refocus : ∀ {Γ u v}
          → (ctx : EvalContext u v)
            (t :  Γ ⊢ u)
            (env : Env Γ)
          → Trace t env ctx
          → Value v

  refocus kont .(‘ i) env (Lookup i p trace) =
    let c = (lookup i env p) in
        refocus kont (getTerm c) (getEnv c) trace
  refocus .MT .(λ body) env (Done body) =
    Val (Closure (λ body) env) tt
  refocus kont .(f ∘ x) env (Left f x trace) =
    refocus (ARG (Closure x env) kont) f env trace
  refocus (ARG (Closure x argEnv) kont)
  .(λ body) env (Right .env body x trace) =
    refocus (FN (Val (Closure (λ body) env) tt) kont) x argEnv trace
  refocus (FN (Val (Closure (λ body) env2) tt) ctx)
  .(λ argBody) env (Beta ctx argBody .env body trace) =
    refocus ctx body (Closure (λ argBody) env · env2) trace
```

FIGURE 7.6: CEK transition function

## 7.3 Correctness guarantees

Finally, having both the new trace datatype as well as the evaluation function, we can look into the correctness and termination of the obtained CEK evaluator. We first introduce property similar to Lemma 6.4, but relating CEK evaluator with the refocused evaluator.

**Lemma 7.3.** *For all types* `u`, `v` *and any typing context* Γ, *let* `ctx` *denote an evaluation context parametrised by types* `u` *and* `v`. *Let* `t` *denote a term* Γ ⊢ `u`. *Let* `env` *denote a substitution environment for a typing context* Γ. *Given a CEK machine trace for* `t`, `env`, `ctx` *and refocusing evaluator trace for a decomposition obtained by refocusing closure of t with environment* `env` *in a context* `ctx`, *CEK* `refocus` *function provides the same result, as* `iterate` *function of the refocusing evaluator.*

*Proof.* Proof by induction on CEK machine case. In case of `Done` trace, trivially holds by reflection. In a case of `Lookup` we appeal to Lemma 7.1 before using induction hypothesis. In the case of `Beta`, Left and `Right` we appeal to induction hypothesis for a next CEK machine state. □

Following Swierstra (2012) we also introduce a predicate that must be satisfied by the CEK machine at each step of the evaluation - that both current environment and current evaluation context are valid - see Figure 7.7

```
invariant : ∀ {Γ u v}
          → EvalContext u v
          → Env Γ
          → Set

invariant ctx env = isValidEnv env × isValidContext ctx
```

FIGURE 7.7: Invariant predicate which needs to be satisfied by CEK machine at every stage of evaluation - sourced from Swierstra (2012)

**Lemma 7.4.** *For all types* $u$, $v$ *and any typing context* $\Gamma$, *let* $ctx$ *denote an evaluation context parametrised by types* $u$ *and* $v$. *Let* $t$ *denote a term* $\Gamma \vdash u$. *Let* $env$ *denote a substitution environment for a typing context* $\Gamma$. *Additionally,* $ctx$ *and* $env$ *satisfy environment and correctness predicates. In such case, the existence of refocusing evaluator trace for a decomposition obtained by refocusing a closure of* $t$ *in environment* $env$ *in a context* $ctx$ *implies the existence of CEK machine trace for* $t$, $env$, $ctx$.

*Proof.* By straightforward induction on evaluation contexts and term structure. `Done` trace of refocusing evaluator, corresponds to `Done` trace of CEK machine. In cases when the term is a variable lookup, we appeal to Lemma 7.1 and use `getEnv` and `getTerm` before relying on induction hypothesis. Additionally, we appeal to Lemma 7.1 to get witness that environment obtained by looking up is correct.

In all remaining cases, we trivially populate CEK machine trace, by appealing to the induction hypothesis for the next CEK state, given the appropriate cases of evaluation contexts and structure of terms. □

We can obtain CEK trace from refocusing evaluator trace if evaluation context and environment are correct. Trace of refocusing evaluator can be obtained from the trace of head reduction evaluator (by Lemma 6.3). We can obtain a head reduction evaluator trace for the closure of any well-typed term in an empty environment. Empty environment and empty context trivially satisfy correctness properties. Therefore, we can obtain a CEK trace for any well-typed term, starting from an empty environment and context.

Knowing that we can introduce a CEK evaluation function that is proven to terminate for any well-typed term. Finally, we state the most important correctness

```
termination : ∀ {u}
            → (t : [] ⊢ u)
            → Trace t Nil MT
termination t = traceLemma MT t Nil (tt , tt) (Refocusing.termination t)

evaluate : ∀ {u}
         → [] ⊢ u
         → Value u
evaluate t = refocus MT t Nil ((termination t))
```

FIGURE 7.8: Terminating CEK evaluation function

property of the obtained CEK machine, that it always reaches the same result as refocusing evaluator for any well-typed term

**Corollary 7.5.** *For all types* `u`*, let* `t` *denote a term* `[] ⊢ u`*. When given* `t`*, CEK* `evaluate` *function yields the same result as* `evaluate` *function of the refocusing evaluator.*

*Proof.* We appeal to Lemma 7.3 and provide witnesses that both refocusing evaluator and CEK evaluator have a trace for any well-typed term by using the consequences of Theorem 5.11 and lemmas 6.3 and 7.4. □

# Chapter 8

# Testing, critical evaluation and project management

## 8.1 Testing and verifiction

The usual part of the report on a programming project is a section about testing the obtained code, using unit, integration or component tests. However, in a project like this, which was developed using dependently-typed Agda, there is no need to do so. Dependent types allow internal verification of the code, that is performing formal verification while developing the code. Therefore, the obtained program is formally proven to be correct with the specification.

## 8.2 Termination of decomposition problem

A careful reader should notice that our mutually recursive definition of `decompose'` and `decompose'_aux` functions are annotated with termination pragma, that is Agda termination checker cannot confirm that a function is terminating. The reason why the Agda termination checker cannot prove the termination of those functions is the fact that they are not structurally recursive. The call-by-value evaluation requires evaluating the left-hand side of the application to a value before looking into the right-hand side. When switching from evaluating the left-hand to the right-hand side, the argument to a next call is not structurally smaller, which triggers the Agda termination checker warning. Locally it looks like the size of the

argument is not decreasing, even though when looking globally, from the point of view of an application of two expressions, progress can be seen.

`decompose'` and `decompose'_aux` were proven to terminate in Danvy and Nielsen (2004). What is more, formalisation of Sieczkowski et al. (2011), which models abstract machines as logical relations, also provides a machine checked proof that `decompose'` and `decompose'_aux` are terminating. Therefore, there is no reason to suspect that asserting a termination of those functions is inconsistent with the rest of theory.

As termination of those functions was explicitly proven in the mentioned literature, we recognise that there is little value in mechanising such proof. Therefore, due to the complexity and little value, proving the termination of those functions was left out as an optional task in this project. Due to a short time frame, we were able to partially complete this task. Therefore, the appendix B contains a sketch of a potential way of solving this problem, which could be the basis for future developments in the project.

## 8.3 Reflection on time management and project planning

The project successfully reached its main objectives, while producing a valuable deliverable and new contributions to the field. It is worth mentioning, that this project was research-oriented and therefore had a high risk associated with it. The engineering management of this project allowed to mitigate those risks.

The project progress was discussed in the weekly meetings with the supervisor. Those meetings involved both theoretical and conceptual discussions, as well as project planning aspects. All tasks in the project were managed and tracked with the Kanban board, using Trello service. Before the first term a Gantt chart was created with the road map of project development (see Figure 8.1). When learning more about the problem domain, and upon seeing the increased complexity of producing a proof of termination a Gantt chart was appropriately adjusted (Figure 8.2).

Two meetings with the secondary examiner were conducted. The main objective was to brief the examiner on the progress of the project, as well as to discuss the final write-up of the project. All received remarks were appropriately addressed.

The mitigation of the risks is described in the Figure 8.3. The project planning involved the potential impact of data loss and hardware failure and was appropriate by backing up the Agda code on University's GitLab service. Moreover, the potential effect of Covid-19 was considered. Finally, to mitigate a high risk related to pursuing a research project contact with researchers from the field, including Dr Swierstra, Prof. Altenkirch and Dr Sieczkowski, was established. Discussions with them were parallel to the discussions with the main supervisor and introduced great value to the project.

Finally, it is worth mentioning that the overall project topic has changed compared to the original project brief (appendix A). The original plan was to deliver Agda formalisation of Douence and Fradet (1998) combinator based framework for abstract machines. The type system considered by the authors contained one unsound rule, which highly increased the complexity of the necessary work, as it would need developing a new type system from scratch. Instead of building some theory from scratch, the focus of the project was shifted towards contributing to existing research in dependently typed formalisations of abstract machines and extending it to the broader case.

## 8.4 Self-evaluation

The project was somewhat challenging, however it was an outstanding opportunity for me to gain a greater understanding of programming language theory and formal methods. I am incredibly pleased with the final outcome of the project. I was able to efficiently and competently adapt to teach myself Agda to a level that has allowed me to contribute to the field I was working in. I consider establishing contact with the authors of the papers I was relying on a great success. Exchanging ideas with Dr Swierstra was a great experience, which allowed me to pick up practical research skills. I completed all the main objectives of the project, however only partially completed the low-priority optional task. Having been given more time, I would choose to focus on proving termination of decomposition, as well as consider languages with control operators. I am extremely grateful to my supervisor and second examiner for their guidance and support.

## 8.5 Gantt charts

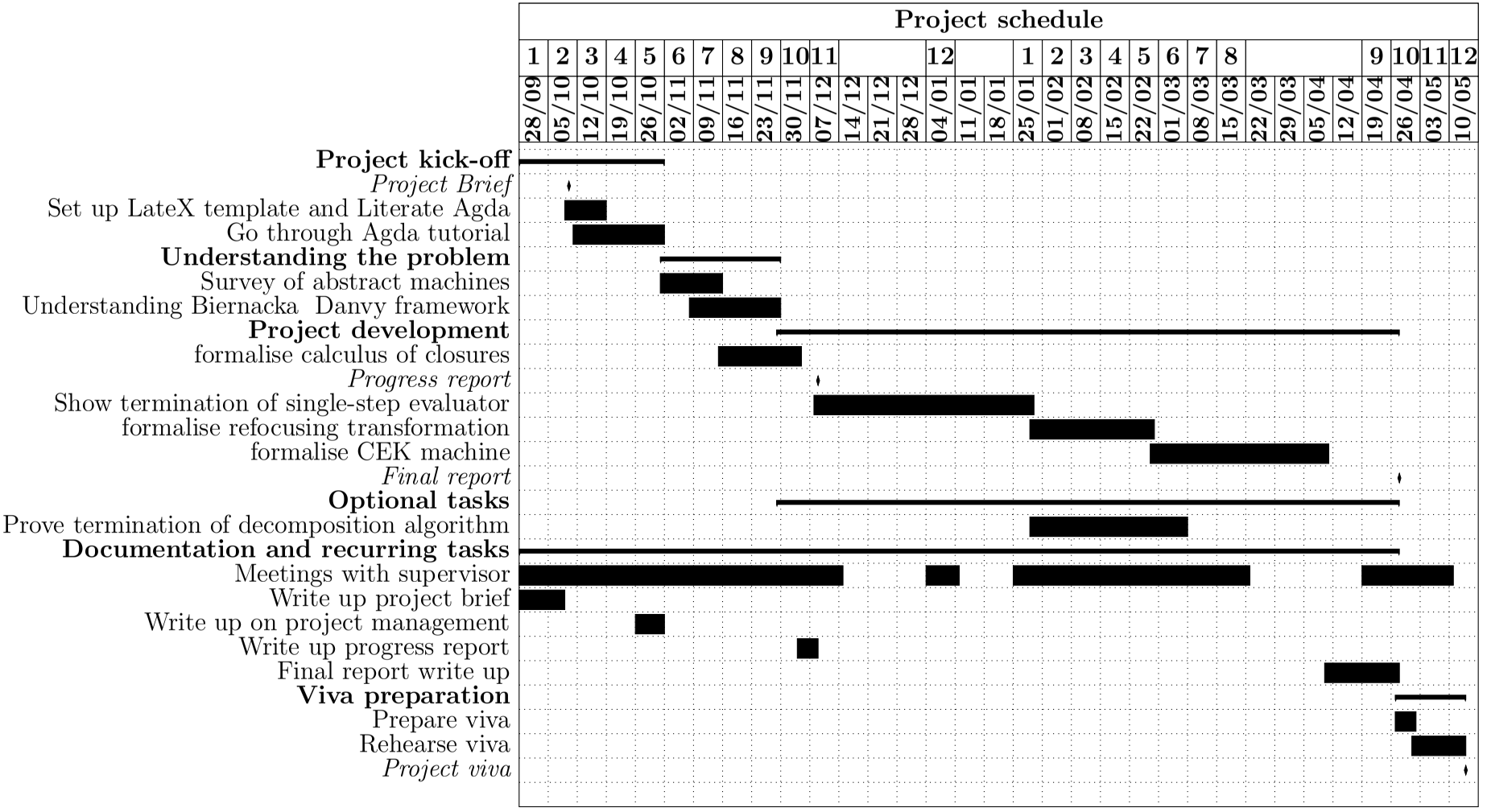


FIGURE 8.1: Planned Gantt chart

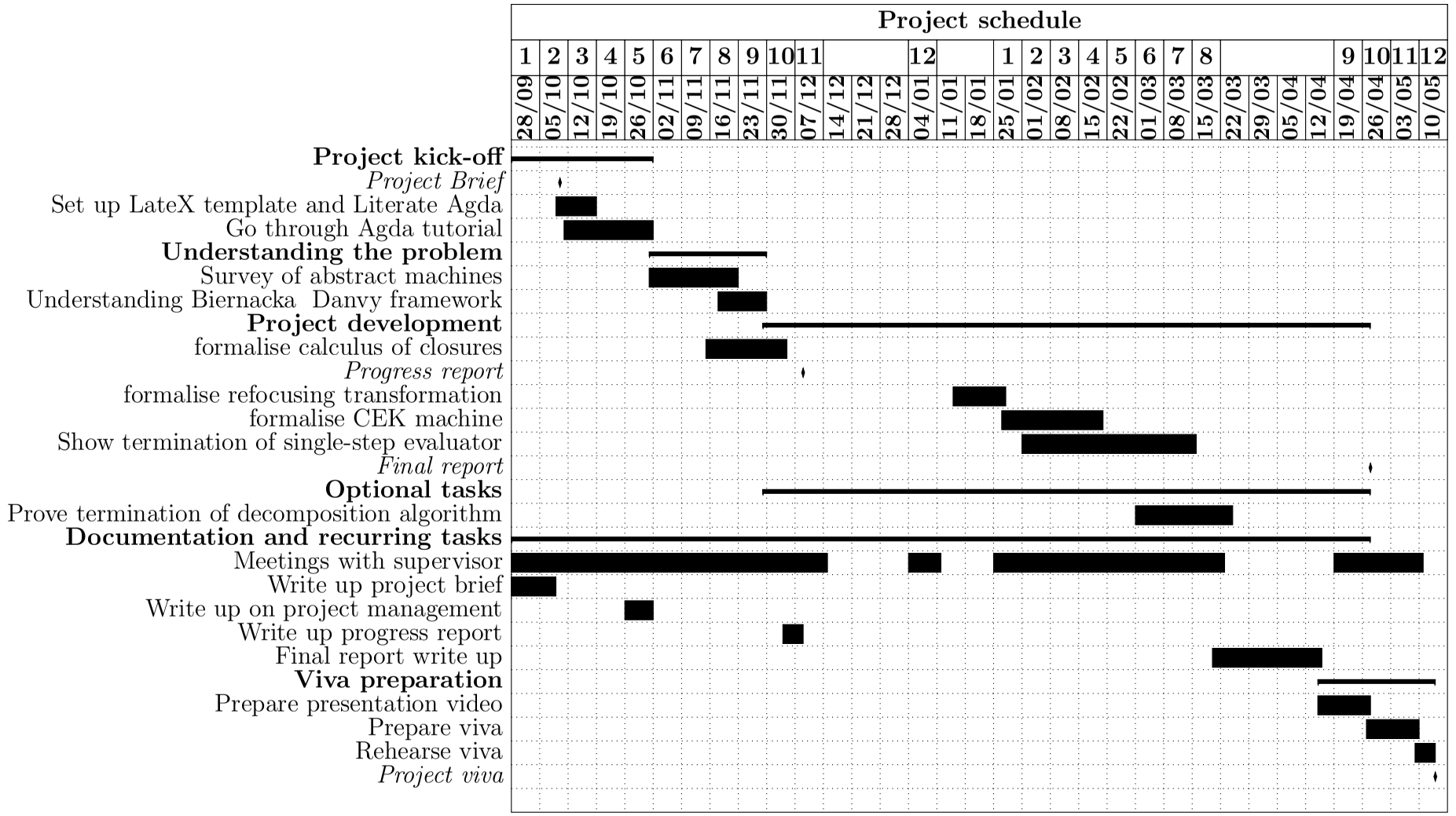


FIGURE 8.2: Actual Gantt chart

## 8.6 Risk assesment

| Risk | Probability (1 - 5) | Severity (1 - 5) | Risk exposure (1-25) | Mitigation |
|---|---|---|---|---|
| Developing severe symptoms of unexpected illness (such as Covid-19) | 2 | 5 | 10 | Follow social distancing, to avoid catching Covid-19.. Have a flexible plan, which could allow 2-3 week break in work. |
| Supervisor developing symptoms of unexpected severe illness (such as Covid-19) | 2 | 4 | 8 | Be independent from the supervisor. Prepare for the possibility of working for a longer time without supervision. |
| Data loss | 1 | 4 | 4 | Use University GitLab service, backup the data |
| Closure of University | 2 | 3 | 6 | Rely on remote work, store data in cloud, so changing location wouldn't affect the project work |
| Unexpected complexity of the topic related to the fact that the project is on a research level and contains topics not widely studied before | 2 | 5 | 10 | Be prepared to slightly change the topic of the project to make it more applied. Carefully study and follow Agda tutorials.Become familiar with Agda forums |
| Physical failure of the hardware used, while developing the project | 1 | 2 | 2 | Backup the project data using University GitLab service, sign in to building 16 bookable desks system, if the computer used for the development fails |

FIGURE 8.3: Risk assessment of the project

# Chapter 9

# Conclusions and future work

The main deliverable of our project is an Agda formalisation of a correct-by-specification CEK machine equivalent to a call-by-value iterated head reduction evaluator for $\lambda\hat{p}$. To our best knowledge, it is first proven to terminate and executable the formalisation of the CEK machine in a dependently typed language.

The main project objectives were met and associated tasks were completed. The only optional task was partially completed, however, it was not critical to the main aims of the project.

We successfully extended Swierstra (2012) formalisation of the Krivine machine to a call-by-value case. Our main contributions include providing Strong Normalisation theorem for $\lambda\hat{p}$ under call-by-value in a style of Tait and Martin-Löf, as well as providing machine-checked proof of CEK equivalence with head reduction evaluator.

A natural extension of the mentioned project would be to consider the derivation of abstract machines for context-sensitive calculi involving control operators. For example lambda calculus with an abort operator, or $\lambda\mu$-calculus (Parigot (1992)). Languages with first-class continuations allow having a computational version of some elements of classic logic through Curry-Howard isomorphism.

There have been already some work on variants of $\lambda\hat{p}$ with control operators and deriving context-sensitive machines through refocusing, with the most important contribution being Biernacka and Danvy (2007a). It is interesting that closures in their work mixed namespace of $\lambda$ and $\alpha$ variables. When deriving a machine for $\lambda\mu$-calculus, one could consider variants of $\lambda\hat{p}$ where closure has two substitution environments - one for $\lambda$ variables and the other one for $\alpha$-variables.

When extending our project to formalising such machines the interesting problem would be proving termination. Biernacka and Biernacki (2009) introduced a variant of Tait-style logical relation taking evaluation contexts into account. The authors also introduced a typing for evaluation contexts, however different from the one from Swierstra (2012) and our project. The authors proved the termination of $\lambda$-calculus with various control operators including callcc, abort and Felleisen's C.

# Bibliography


Andreas Abel and James Chapman. Normalization by evaluation in the delay monad: A case study for coinduction via copatterns and sized types. *Electronic Proceedings in Theoretical Computer Science*, 153, 06 2014.

Thorsten Altenkirch and James Chapman. Big-step normalisation. *Journal of Functional Programming*, 19(3-4):311–333, 2009.

Biernacka. Generalized refocusing: a formalization in coq. 2016.

Malgorzata Biernacka and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theor. Comput. Sci.*, 375:76–108, 05 2007a.

Małgorzata Biernacka and Dariusz Biernacki. A context-based approach to proving termination of evaluation. *Electronic Notes in Theoretical Computer Science*, 249:169–192, 2009. ISSN 1571-0661. Proceedings of the 25th Conference on Mathematical Foundations of Programming Semantics (MFPS 2009).

Małgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Trans. Comput. Logic*, 9(1):6–es, December 2007b. ISSN 1529-3785.

Ana Bove. General recursion in type theory. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs*, pages 39–58, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-39185-2.

Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda – a functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 73–78, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-03359-9.

P. Curien. An abstract framework for environment machines. *Theor. Comput. Sci.*, 82:389–402, 1991.

Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics, 2004.

Rémi Douence and Pascal Fradet. A systematic study of functional language implementations. 20(2), 1998. ISSN 0164-0925.

M. Felleisen and D. Friedman. Control operators, the secd-machine, and the $\lambda$-calculus. In *Formal Description of Programming Concepts*, 1987.

Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, USA, 1989. ISBN 0521371813.

Gérard Huet. The zipper. *J. Funct. Program.*, 7:549–554, 09 1997.

Adam Krupicka. Coinductive formalization of secd machine in agda, 2018.

Joan Rand Moschovakis. The logic of brouwer and heyting. In Dov M. Gabbay and John Woods, editors, *Logic from Russell to Church*, volume 5 of *Handbook of the History of Logic*, pages 77 – 125. North-Holland, 2009.

Michel Parigot. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In Andrei Voronkov, editor, *Logic Programming and Automated Reasoning*, pages 190–201, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg. ISBN 978-3-540-47279-7.

Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

Filip Sieczkowski, Małgorzata Biernacka, and Dariusz Biernacki. Automating derivations of abstract machines from reduction semantics:. In Jurriaan Hage and Marco T. Morazán, editors, *Implementation and Application of Functional Languages*, pages 72–88, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-24276-2.

Wouter Swierstra. From mathematics to abstract machine: A formal derivation of an executable krivine machine. In James Chapman and Paul Blain Levy, editors, Proceedings Fourth Workshop on *Mathematically Structured Functional Programming,* Tallinn, Estonia, 25 March 2012, volume 76 of *Electronic Proceedings in Theoretical Computer Science*, pages 163–177. Open Publishing Association, 2012.

W. W. Tait. Intensional interpretations of functionals of finite type i. *Journal of Symbolic Logic*, 32(2):198–212, 1967.

Anne Sjerp Troelstra et al. History of constructivism in the 20th century. 2011.

David Van Horn and Matthew Might. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, page 51–62, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605587943.

Philip Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, November 2015. ISSN 0001-0782.

Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*. July 2020.

# Appendix A

# Original project brief

# Formally verified, compositional framework for abstract $\lambda$-calculus machines

Authored by **Wojciech Rozowski**
Supervised by **Dr Julian Rathke**

## Problem

Throughout the development of functional programming languages, one of the most researched topics is their implementation. A common approach to describe programming languages semantics is using abstract state machines (ASMs), a first-order transition systems based on the abstract syntax trees. Several abstract machines have been developed and formally described, with most notable examples of SECD (Landin [1964]), Krvine-machine (Krivine [2007]), CE(S)K (Felleisen and Friedman [1987]) and STG (Jones et al. [1992]).

Despite the usefulness of abstract state machines and high amount of research papers about them, there have been relatively few efforts to come up with way of unified global way of studying, reasoning and comparing them. Biernacka and Danvy [2007] did a notable contribution to this field, by providing a derivational taxonomy of ASMs from one-step reduction functions.

A slightly less known, yet highly interesting concept was introduced by Douence and Fradet [1998]. The authors descibed a unified abstract framework to describe, relate, compare and classify functional language ASMs implementations, by expressing compilation process as series of compositional program transformations, between combinator-based intermediate languages. Each intermediate transform can be seen as different fundamental choice for the evaluation strategy and correctness of each of the steps can be treated distincly from the rest of program transforms.

Using this theory one could build a formally verified framework for ASM-based interpreters of functional languages in a manner similar to formally verified compilers (eg. CompCert by Leroy [2009]), however no such works have been done before in this area, and this leaves an interesting area for investigation.

## Goals

A primary goal of this project is to implement Dounce and Fradet's framework in a formally verified way using Agda (Bove et al. [2009]) - a dependently typed programming language based on Martin-Löf's intuitionistic type theory, which extends it with mutliple programming language features. By Curry-Howard isomorphism, well-typed programs in a language with such type system, can be also seen as constructive proof of inhabitance of their corresponding types.

Upon formalizing this framework in Agda, the secondary goal is to study what properties of the program transformations can be formally guaranteed. Possible routes could include looking into guarantees of normalization properties, semantic preservation or type preservation.

The tertiary and optional goal is to consider coinductive formalization of the target language of the transforms and study whether one could prove the termination of the well-typed input programs, by using delay monad, similarly to master thesis by Krupicka [2018]

## Scope

Instead of thinking about more complicated functional programming language, the scope of the project will be restricted to a language with fewer amount of constructs. As a starting point, simply typed $\lambda$-calculus will be considered, as it can be seen as natural basis for any programming language. Then, more sophisticated elements will be studied and considered accordingly to the time left. Also, given a rather short-time frame for the project, the scope of the project might consider only a subset of intermediate languages transformations and therefore a smaller subset of covered ASMs (eg. omitting graph reduction-based machines), focusing on providing functional and verified framework that can be gradually extended. However, the goal is to cover as many transforms as possible.

# Appendix B

# Sketch of termination proof of a decomposition function

Even though the arguments to the `decompose` and `decompose'_aux` are not structurally recursive, we can attempt to order the configurations (that is pair of current closed term and substitution environment) by the order they are being called. We use a principle, which is supported in Agda and is known as well-founded recursion. If we can show that the non-structurally recursive function calls argument smaller with respect to a given well-founded relation, then it passes the termination checking. Safe configurations to recurse through, are those which are accessible in a given relation. If for some `x` for all `y`, such that `y < x` (with respect to that relation), then `x` is accessible.

As `decompose'` and `decompose'_aux` take multiple arguments, we introduce a configuration (see figure B.1) datatype, which describes valid states of traversing through some closed term. Confuguration of a closed term `closure` is some current closed term along a corresponding substitution context and witness that plugging a current closed term into given evaluation context recreates the original `closure`.
Now, inspired by $<_f$ and $<_t$ relations from Sieczkowski et al. (2011) we introduce an ordering on configurations (see figure B.2). We restrict the relation to compare the possible configurations of only one underlying closed term, rather than arbitrary configurations. Broadly speaking, `arg-lt-clapp` says that traversing to the left-hand side is a configuration closer to the end than the closed-term of application of two sides. `fn-lt-arg` describes situation when switching the evaluation contexts. The rules are not transitive by definition, so we impose the transitivity by definition.

```
data Configuration (v : Type) (closure : Closed v) : Set where
  [_-_-_]   : ∀ {u}
        → ( ctx : EvalContext u v )
        → ( c : Closed u )
        → ( plug ctx c ≡ closure )
        → ( Configuration v closure )
```

FIGURE B.1: Configuration datatype definition

$$\text{arg-lt-clapp}\ \frac{\begin{array}{c}\texttt{(ctx : EvalContext u v)(x : Closed a)}\\ \texttt{(f : Closed (a ⇒ u))(closure : Closed v)}\\ \texttt{(p1 : plug ctx (Clapp f x) ≡ closure)}\\ \texttt{(p2 : plug (ARG x ctx) f ≡ closure)}\end{array}}{\texttt{[[ closure ]] [ ARG x ctx - f - p2 ] < [ ctx - Clapp f x - p2 ]}}$$

$$\text{fn-lt-arg}\ \frac{\begin{array}{c}\texttt{(ctx : EvalContext u v)(x : Closed a)}\\ \texttt{(f : Closed (a ⇒ u) )(p : isVal f)}\\ \texttt{(closure : Closed v)}\\ \texttt{(p1 : plug ctx (Clapp f x) ≡ closure)}\end{array}}{\texttt{[[ closure ]] [ FN (Val f p) ctx - x - p1 ] < [ ARG x ctx - f - p1 ]}}$$

$$\text{transitivity}\ \frac{\begin{array}{c}\texttt{(c : Closed v)(e1 : Configuration v c)}\\ \texttt{(e2 : Configuration v c)(e3 : Configuration v c)}\\ \texttt{[[ c ]] e1 < e2}\\ \texttt{[[ c ]] e2 < e3}\end{array}}{\texttt{[[ c ]] e1 < e3}}$$

FIGURE B.2: Ordering on configurations

Having such a relation, we can rewrite our `decompose'` and `decompose'_aux` to pass termination checking, given the accessibility predicate - see figure B.3. The open problem is giving an accessibility predicate for any valid configuration, that is proving the well-foundedness of the relation. It is possible, that the given definition of the relation is non-canonical and transforming it into an equivalent definition that is canonical would allow creating a straightforward well-foundedness proof. Also, the concerning thing is the fact that multiple properties of decomposition rely on proofs by reflection, which could be problematic when rewriting the equations involving the variants of functions taking accessibility predicate.

```
mutual
  dec5a : ∀ { u v}
         → (ctx : EvalContext u v)
         → (c : Closed u)
         → (a : Acc [[ plug ctx c ]]_<_ ([ ctx - c - refl ]))
         → Decomposition (plug ctx c)
  dec5a ctx (Closure (λ x) env) a = dec5b ctx x env a
  dec5a ctx (Closure (x ∘ x) env) a = Redex×Context (App x x env) ctx
  dec5a ctx (Closure (‘ x) env) a = Redex×Context (Lookup x env) ctx
  dec5a ctx (Clapp f arg) (acc rs) =
      dec5a (ARG arg ctx) f
      (rs [ ARG arg ctx - f - refl ]
          (arg-lt-clapp ctx arg f (plug ctx (Clapp f arg)) refl refl))

  dec5b : ∀ { a b w Γ}
        → (ctx : EvalContext (a ⇒ b) w)
        → (body : (a :: Γ) ⊢  b)
        → (env : Env Γ)
        → (a : Acc [[ plug ctx (Closure (λ body) env) ]]_<_
          ([ ctx - (Closure (λ body) env) - refl ]))
        → Decomposition (plug ctx (Closure (λ body) env))
  dec5b MT body env rs = Val body env
  dec5b (FN (Val (Closure (λ x) env) p) ctx) body env rs =
      Redex×Context (Beta x env (Val (Closure (λ body) env) tt)) ctx
  dec5b (ARG x ctx) body env (acc rs) =
      dec5a (FN (Val (Closure (λ body) env) tt) ctx) x
          (rs [ FN (Val (Closure (λ body) env) tt) ctx - x - refl ]
              (fn-lt-arg ctx x (Closure (λ body) env) tt
              (plug ctx (Clapp (Closure (λ body) env) x)) refl))
```

FIGURE B.3: Well-founded `decompose'` and `decompose'_aux`

# Appendix C

# Desciption of project archive

The whole formalisation is contained in the `cek.agda` file. It requires agda 2.6.1, as well as compatible agda standard library to sucessfully pass the checks. The formalisation root module is `cek` and it contains several submodules:

- `Terms` - contains the definitions of closd terms of $\lambda\hat{p}$-calculus (see chapter 3)
- `Redex` - contains the decomposition function, definition of single-step head reduction as well as lemmas on head reduction properties (see chapters 3 and 4)
- `IteratedHeadReduction` - Strong Normalization theorem and corresponding lemmas (see chapter **??**)
- `Refocusing` - Refocusing transformation and corresponding equivalence and termination proofs (see chapter 6)
- `Machine` - CEK machine and corresponding equivalence and termination proofs (see chapter 7)

We also include `wf-decomposition.agda` which contains our unfinished developments in proving termination of `decompose'` and `decompose'_aux` functions. It only includes `Terms` and `Redex` modules.

# Appendix D

# Total word count

The total word count in the body of the report was counted using TeXcount web service (https://app.uio.no/ifi/texcount/online.php) and is **9975**