Author/s:
Chen, Zhuo

Title:
A Verified Cost Model for Call-By-Push-Value

Date:
2025

Persistent Link:
https://hdl.handle.net/11343/366838

THE UNIVERSITY OF MELBOURNE

DOCTORAL THESIS

A Verified Cost Model for Call-By-Push-Value

Author:
Zhuo CHEN
ORCID: 0000-0002-0617-3434

Supervisor:
Christine RIZKALLAH
Johannes ÅMAN POHJOLA

A thesis submitted in total fulfillment for the degree of Doctor of Philosophy

in the

School of Computing and Information Systems
Faculty of Engineering and Information Technology
THE UNIVERSITY OF MELBOURNE

October 2025

Declaration of Authorship

I, Zhuo CHEN, declare that this thesis titled, "A Verified Cost Model for Call-By-Push-Value" and the work presented in it are my own. I confirm that:

- The thesis comprises only my original work towards the degree of Doctor of Philosophy except where indicated in the preface;
- Due acknowledgement has been made in the text to all other material used; and
- The thesis is fewer than the maximum word limit in length, exclusive of tables, maps, bibliographies and appendices as approved by the Research Higher Degrees Committee.

Signed:

Date: February 3, 2026

“我站在今天设想过去又幻想未来，过去和未来在今天随意交叉，因而过去和未来都刮着现在的风。”

史铁生

“I stand in today, envisioning the past and dreaming of the future. The past and future intertwine freely at today, and thus, both the past and future are blowing in the winds of present.”

Tiesheng SHI

(Translated by Zhuo Chen, author of this thesis.)

THE UNIVERSITY OF MELBOURNE

Abstract

Faculty of Engineering and Information Technology
School of Computing and Information Systems

Doctor of Philosophy

A Verified Cost Model for Call-By-Push-Value

by Zhuo CHEN

The call-by-push-value λ-calculus allows for syntactically specifying the order of evaluation as part of the term language. Hence, it serves as a unifying language for embedding various evaluation strategies including call-by-value and call-by-name. Given the impact of call-by-push-value, it is remarkable that its adequacy as a model for computational complexity theory has not yet been studied. In this thesis, I show that the call-by-push-value λ-calculus is *reasonable* for both time and space complexity. A reasonable cost model can encode other reasonable cost models with polynomial overhead in time and constant factor overhead in space. I achieve this by encoding call-by-push-value λ-calculus into Turing machines, following a simulation strategy by Forster et al.; for the converse direction, I prove that Levy's encoding of the call-by-value λ-calculus has reasonable complexity bounds. I also extend the formalisation of the call-by-push-value lambda-calculus with the features necessary to encode a call-by-need evaluation strategy. The main results have been formalised in the HOL4 theorem prover.

Preface

This thesis presents original research conducted during my candidature of Doctor of Philosophy at the University of Melbourne.

Chapter 3 ports formalisation from existing literature Forster, Kunze, and Roth (2020). The ported HOL4 formalisation is assessible online (Chen, 2022a). The formalisation has also been included in the official HOL4 library (HOL, 2022). Chapter 4 is adapted from publication (Chen, Pohjola, and Rizkallah, 2025), where I was the primary contributor. The relevant HOL4 formalisation is published online as an open-source artefact (Chen, 2025). Chapter 5 is unpublished material, it introduces possible extensions opening the door for future work. The relevant HOL4 formalisation is available online (Chen, 2026).

No work in this thesis has been submitted for other qualifications. All the work is carried out after the enrolment in the course.

The work was funded by the Melbourne Research Scholarship and the Faculty of Engineering and Information Technology, University of Melbourne.

Acknowledgements

Fresh from my undergraduate studies, I immersed myself in further research, continuing a path shaped by three years of experience in my undergraduate years. In the blink of an eye, I now find myself at the stage of writing up my doctoral thesis. It has been a remarkable and fulfilling four-year journey, forming one of the most significant chapters of my life. I could not have come this far without the support I have received from so many along the way.

First and foremost, I would like to express my deepest gratitude to my supervisors, Dr. Christine Rizkallah and Dr. Johannes Åman Pohjola. Your patience, insight, unwavering guidance, and deep humanity have been the cornerstone of my doctoral journey. There were some very difficult moments for me in the past four years, and you were always there to support me no matter how busy you were. Thank you for taking me under your wings and guiding me not only in academia, but also in life.

I would also like to thank fellow PhD students in our lab at UniMelb. I got to know each of you at different stages of my journey, and I truly cherish all the moments we shared. I especially want to thank Louis Cheung, Vincent Jackson, and Selene Linares, who started their PhD around the same time as me. I deeply appreciate the peer support you offered. I am also grateful for the feedback provided by all of you for my paper drafts and practice presentations.

I am also grateful for the PhD friends I have in other universities for sharing this journey together. It meant a lot to have friends who could truly resonate with and relate to the challenges we faced along the way. I would especially like to thank Zilin Chen, for help proof-reading my paper drafts and providing extremely detailed and helpful feedback. I would also like to thank Zixian Cai and Yiming Xu for their unwavering peer support and friendship since our undergraduate years. Our online catchups provided a much-needed space to share the challenges of doctoral life; these conversations were both soothing and encouraging, and I am deeply grateful for their companionship across different cities.

This journey would not have started without the support I received during my undergraduate years, thus I would also like to give my special thanks to the academics who kindly helped and/or supervised me at ANU. I would like to thank my undergraduate and honours supervisor, Dr Michael Norrish. Thank you for taking me in as your student and opening the gate of

formal verification to me. I truly would not have been where I am today without this starting point. Thank you for your patience and encouragement from the beginning when I barely know how to write proofs. Your driven attitude towards knowledge and your kindness always inspires me. I would also like to thank Prof. Steve Blackburn and Prof. Antony Hosking, for the invaluable guidance and opportunities they gave me when I worked as a research assistant for them. I must thank Dr. Ekaterina Lebedeva and Dr. Ben Swift for their kind support in helping me transfer into the Bachelor of Philosophy (Science) program at ANU, where I was able to focus on research projects in computer science and math. Before this, I was an actuarial student, unsure of my passion or future direction. Your encouragement gave me the confidence to change paths, and it is the foundation of why I am here today as a computer scientist.

I would also like to thank my wonderful friends in life: Cecilia Lao and Yihong Li. Thank you, Cecilia, for always being there for me during all these years, for listening to my rants, for sharing similar hobbies, and for being such a wonderful companion as we chased our parallel yet beautifully isomorphic goals, and for all the Dopa runs. Thank you, Yihong, for standing by me during a very difficult time last year, for always bringing laughter, and for being such a caring companion.

I would also like to thank the many more friends I made along my path during these years, I cannot name everyone here, but each of you has been a great source of inspiration and courage. I am truly grateful to have become friends with every one of you.

Last but not the least, I would love to thank my family: my parents, my sister, and my extended family. Thank you, mom and dad, for supporting my decision to study abroad 11 years ago, for always standing by me through every bold or reckless choice I made, and for always beliving in me unconditionally. Thank you sister for sharing countless memes that always brightened my days and thank you for the cute Chiikawa plushies. And thank you to my extended family for always being there for me, for sending food and family photos and encouraging voice notes, and for visiting me from time to time.

I feel incredibly fortunate to have met each of you along this journey. Thank you all, once again, for everything. May you achieve what you aspire to and live your lives to the fullest.

Contents

List of Figures

List of Abbreviations

CBV	**C**all **B**y **V**alue
WCBV	**W**eak **C**all **B**y **V**alue
CBPV	**C**all **B**y **P**ush **V**alue
ECBPV	**E**xtended **C**all **B**y **P**ush **V**alue

Dedicated to all the girls who have been told, by culture, tradition, or bias, that they are not meant for STEM, that they are not as capable or intelligent as boys. May your curiosity persist, your voice grow louder, and your path remain yours.

Chapter 1

Introduction

The λ-calculus (Church, 1932) is a fundamental model of computation that represents functions as abstractions over variables. It provides a foundation for computability, mathematical logic, and functional programming. Unlike programs written in an imperative style, functional programs do not specify low-level details describing *how* they should run on a computer. Instead, functional languages provide a convenient way of describing *what* functions should compute *to* in a concise declarative manner. This abstract representation is ideal for proving that a program is correct against its specification.

Besides functional correctness, another property that is of high importance to computer scientists is run-time computational complexity. Computational complexity addresses the vital questions: How fast does my program produce an output? How much space will my program use in order to produce that output?

In complexity analysis, we describe the asymptotic behaviour of *cost functions* that model the cost of the program mathematically. But this leads to a question: Where do cost functions come from? Cost functions are usually constructed in an implicit and ad-hoc manner with no formal connection to the program semantics. *Cost models* (Turing, 1936; Hartmanis and Stearns, 1965) bridge that gap.

Creating cost models for functional programming languages is thus an essential topic. It is also a significant challenge due to the abstract nature of functional programming. As λ-calculus is the basis of functional programming languages, a crucial question is whether we can create cost models for λ-calculus. There has been a large body of research (Lawall and Mairson, 1996; Dal Lago and Martini, 2008; Accattoli and Dal Lago, 2016; Accattoli, 2017; Forster, Kunze, and Roth, 2020; Forster et al., 2021; Accattoli, Dal Lago, and Vanoni, 2022) dedicated to solving this problem.

An important parameter when considering a cost model for λ-calculus is the evaluation strategy being used. An evaluation strategy determines the

order in which function calls are evaluated. An example of such a strategy is *call-by-value*, where function calls are applied once the arguments are fully evaluated. This strategy is used, for instance, by the Standard ML programming language (Milner, Tofte, and Harper, 1990). Another example of an evaluation strategy is *call-by-name*, where the evaluation of arguments is deferred, and the function is applied before the arguments are evaluated. A third example is the *call-by-need* (lazy evaluation) strategy, in which arguments are evaluated only when their values are required by the next immediate computation. The most well-known programming language that uses *call-by-need* is Haskell (Hudak et al., 2007), which is famous for its laziness - this stems from the *call-by-need* evaluation strategy.

Since different evaluation strategies traverse and reduce different parts of a λ-term in different orders, the choice of strategy can have a significant impact on the run-time efficiency of a program. For example, if a computationally expensive subterm is ultimately discarded by the control flow of the program, a strategy like *call-by-value* may still evaluate it eagerly, incurring unnecessary cost. In contrast, *call-by-name* or *call-by-need* may defer or avoid this evaluation altogether, depending on whether the subterm's value is ever required. These differences lead to distinct time and space complexities—and thus, distinct cost models—across evaluation strategies, even for identical programs. As such, evaluation order is not merely an implementation detail, but a fundamental aspect of the operational semantics that can influence both theoretical cost models and practical performance. The question next is: Can we have a cost model that is able to capture the complexity of more than just one evaluation strategy? The call-by-push-value λ-calculus provides a way.

The call-by-push-value λ-calculus (CBPV) (Levy, 1999) is an expressive variant of the λ-calculus that enables programmers to specify when a function call evaluates on a call-by-call basis. In particular, CBPV has syntactic constructs that enable delaying or forcing the evaluation of specific terms. Hence, it provides flexibility and fine-grained control in choosing the order in which to evaluate subterms, without the need for separately considering multiple evaluation strategies. Various evaluation strategies, including call-by-value and call-by-name, can be encoded within this subsuming paradigm.

This high expressive power of CBPV has resulted in a long line of work on the language since its inception. Prior work has studied and extended the calculus (Kesner and Viso, 2021; Torczon et al., 2023; McDermott and Mycroft, 2019), related it to other calculi (Ehrhard and Tasson, 2016; Chouquet and

Tasson, 2020; Garbuzov et al., 2018), and formalised it (Rizkallah, Garbuzov, and Zdancewic, 2018; Forster et al., 2019). The fine-grained control CBPV provides has proved useful to verify compiler optimisations (Rizkallah, Garbuzov, and Zdancewic, 2018). There are foundational results that demonstrate that CBPV aids in recurrence extraction, which can in turn be used for analysing the complexity of functional programs, with various evaluation strategies (Kavvos et al., 2020).

This demonstrates that CBPV is crucial as it can serve as a single foundational basis for further research on analysing the computational complexity of functional languages with various evaluation strategies. As such, it is vital to establish time and space cost models for CBPV.

Naturally, such cost models must maintain some property that demonstrates that they are useful models for analysing computational complexity.

Reasonable is a standard notion in complexity theory for reasoning about complexity of different computational models. This notion is defined as follows in the *invariance thesis* (van Emde Boas, 1990, page 5),

> *"Invariance Thesis: Reasonable machines simulate each other with polynomially bounded overhead in time and constant factor overhead in space".*

What is implicit here is that these reasonable machines must choose some cost models for the simulations between them. Those chosen cost models that allow reasonable simulation between the machines, are called reasonable cost models.

In this thesis, I provide a reasonable cost model for CBPV, which includes a time cost model and space cost model. Using this cost model, I demonstrate that CBPV is a reasonable model of computation, in terms of both time and space. More specifically, I prove that CBPV and Turing machines can reasonably simulate each other by going through one layer of intermediate representation. This intermediate layer consists of two abstract machines, the substitution machine and the heap machine. The simulation between CBPV and the abstract machines is the most theoretically interesting and error-prone part of the whole proof. For this reason, I formally verify the related parts in the HOL4 prover (Slind and Norrish, 2008). This includes formalising (mechanising) CBPV and the abstract machines, and machine-checking the simulation between them.

1.1 Contribution

The specific contributions of this thesis are as follows:

1. To the best of my knowledge, I develop the first reasonable time and space cost models for CBPV.
2. I formalise CBPV and provide a machine-checked proof in HOL4 for the core parts of both the time and space cost models for CBPV.
3. I cross-validate the existing reasonable cost models for weak call-by-value λ-calculus (WCBV) (Forster, Kunze, and Roth, 2020) by formalising them in HOL4 and verifying that their results are consistent across different provers under different foundational systems.
4. I extend my CBPV cost model to obtain verified cost models for WCBV. This is also formally verified in HOL4.
5. I extend my formalisation of CBPV to have call-by-need functionality, called ECBPV (McDermott and Mycroft, 2019), enabling further exploration of the cost models in the context of shared resources.

In future work, my cost models for CBPV can be extended into cost models for λ-calculus variants with different evaluation strategies, thus laying the foundation for a unifying approach of complexity analysis for the λ-calculus. My thesis can also facilitate further research into cost analysis for CBPV with different extensions. While my work presents reasonable time and space cost models, it does not consider sublinear time or space classes. This will be another interesting direction to look into. My formalisation of ECBPV can also be further used for cost model development of ECBPV.

1.2 Formalisation

All the formalisations (formal verifications) in this thesis are done in HOL4. They split into three projects: WCBV, CBPV, ECBPV, with the CBPV project being the main contribution of this thesis.

The machine-checked proof for reasonable time and space cost models for WCBV in HOL4 follows the mechanised proof for WCBV by Forster, Kunze, and Roth (2020) in the Rocq Proof Assistant (2025).

Building on foundational work on formally verified time and space cost models for WCBV (Kunze, Smolka, and Forster, 2018; Forster, Kunze, and

Roth, 2020), I demonstrate that CBPV, a more general λ-calculus, is reasonable. I further show that the embedding of WCBV using CBPV maintains reasonability. The machine-checked proof for CBPV is split into HOL4 theories covering the following aspects:

1. The syntax and semantics of CBPV, including extra big-step semantics that keeps track of the time and space cost during the evaluation. (Section 4.2)

2. The definition of machine-readable programs and compilation functions from CBPV terms to machine-readable programs for substitution machine and heap machine respectively. (Section 4.3)

3. The semantics of substitution machines and proofs for its simulation of CBPV terms with respect to time and space. (Section 4.4.1)

4. The semantics of heap machine and proofs for its simulation of CBPV with respect to time and space. Related helper definitions and functions, such as the heap structure and the unfolding function. (Section 4.4.2)

5. The proofs of Turing machine simulating CBPV through abstract machines. (Section 4.5.1)

6. The proofs of CBPV simulating WCBV, and how this can be extended using existing work to simulate Turing machines. (Section 4.5.2)

The formalisation for ECBPV is built on top of the CBPV formalisation, extending it with the necessary constructs to capture the laziness semantics. This includes the introduction of extra operators for memoisation and the corresponding big-step semantics that accounts for the evaluation of lazily evaluated expressions.

1.3 Thesis Structure

In this thesis, I present a verified reasonable cost model for the call-by-push-value λ-calculus and work I have done surrounding this topic. The assumed audience for this thesis is anyone with undergraduate computer science knowledge. This thesis aims to give a thorough tour of cost analysis for CBPV, starting from the fundamental knowledge. For this purpose, I provide a comprehensive background chapter, which introduces the technical parts that are

omitted in this introduction chapter. Building on these understandings, I then talk about what I did for my research project and in the end, what more could be done in the future.

In this chapter (Chapter 1), I provided the context for my thesis, including motivation and importance of research on cost models for CBPV. I also outlined the main contributions (Section 1.1) of this thesis and the related formalisation projects (Section 1.2). Finally, I presented a roadmap of the thesis in the current section (Section 1.3).

In Chapter 2, I cover the essential background knowledge for this work. That chapter introduces computational models (Section 2.1), complexity (Section 2.2), and formal verification (Section 2.3). For computational models, I give a thorough introduction to λ-calculus (Section 2.1.1), CBPV (Section 2.1.2), and Turing machines (Section 2.1.3). The section on complexity briefly reviews big $\mathcal{O}$ notation (Section 2.2.1), then introduces cost models for computational models and explains what qualifies as a *reasonable* cost model (Section 2.2.2). For formal verification, I discuss the methodology and importance of verifying systems and models. I then introduce one tool used in formal verification - interactive theorem provers (Section 2.3.1). In particular, I describe the HOL4 prover (Section 2.3.2), which I use for all my formalisation.

In Chapter 3, I present the verified time and space cost models for WCBV, originally developed and verified in the Rocq prover by Forster, Kunze, and Roth (2020). I also describe the formalisation I developed in HOL4 for these cost models, further confirming that their results are consistent across different provers and foundational systems.

In Chapter 4, I present the main contribution of this thesis: verified time and space cost models for CBPV in HOL4. I first formalise the CBPV language model. I then develop and formalise reasonable time and space cost models for the CBPV. I describe in detail and provide further pen-and-paper proofs of how the reasonable simulation between CBPV and Turing machines is achieved using abstract machines and the existing cost models for WCBV.

In Chapter 5, I extend the formalised CBPV model to support laziness. By adding the extra operators for memoisation to the syntax, we can express call-by-need λ-calculus using the extended CBPV, following the theory developed by McDermott and Mycroft (2019).

In Chapter 6, I summarise the thesis and its contributions. I also identify the limitations of my work and outline directions for future research.

Chapter 2

Background

In this chapter, I introduce the essential background material needed for the remainder of the thesis. In Section 2.1, I present the computational models that serve as the foundation for my work: the λ-calculus (Section 2.1.1), CBPV (Section 2.1.2), and Turing machines (Section 2.1.3). Following this, I review concepts from computational complexity in Section 2.2. I begin with a brief overview of asymptotic analysis and big $\mathcal{O}$ notation in Section 2.2.1. Next, in Section 2.2.2, I introduce the notion of *reasonable cost models*, which is the standard cost framework used in the thesis for verifying the relevant cost models. Finally, in Section 2.3, I introduce the methodology of formal verification and the main formal verification method I use for constructing and machine checking my formal proofs in this thesis: interactive theorem proving, in particular using the HOL4 prover.

2.1 Computational Models

In this section, I outline the core models of computation used in this thesis. I begin with an in-depth introduction to the λ-calculus in Section 2.1.1, including illustrative examples that highlight its role as a minimal yet expressive model of computation. After establishing the basic ideas of λ-calculus, I turn to CBPV, the main focus of this thesis. CBPV extends the λ-calculus with extra expressive power and capability - turning the choice of evaluation order from the meta (language) level to the program level. Subsequently, I briefly introduce Turing machines, the canonical model of computation widely used in computer science. While Turing machines are not the direct focus of this thesis, they serve as the standard baseline for comparison when we analyze the cost semantics of CBPV.

2.1.1 The λ-Calculus

In this section, I go through the fundamentals of the λ-calculus, including what they are, how to write λ-terms, and how to evaluate λ-terms.

The λ-calculus (Church, 1932), introduced by Alonzo Church in the 1930s, is a standard model of computation that is based on abstractions and applications. The λ-calculus has significant research importance to many fields of study in computer science. It is an elegant language for describing and reasoning about computations and properties of programs on a high level. It is the foundational model for functional programming languages, such as Haskell (Hudak et al., 2007) and Standard ML (Milner, Tofte, and Harper, 1990). It is also a basis for type systems.

Now let's look into λ-calculus in more detail. I will start from its syntax.

Syntax

A λ-term can be a variable, a λ-abstraction (abstraction), or an application, as shown below:

Definition 1 (Syntax for λ-Calculus)**.**

$$\lambda\text{-}term\ s, t := x \mid \lambda x.\, s \mid s\ t$$

Let's have a look at these three constructs in more detail:

1. x: a variable. Variables are typically represented by a character or a string. Examples: x, y, z, f.

2. $(\lambda x.\, s)$: an abstraction, where x indicates the variable that is bound by this abstraction and s is a λ-term representing the body of the abstraction. A λ-abstraction represents a function definition, where the binding parameter x is the placeholder for the function input, and the nested λ-term s is the function body. Examples: $(\lambda x.\, x)$, $(\lambda y.\, (\lambda x.\, x\, y))$. Notation for abstractions can be abbreviated by merging the binding parameters together. For instance, the last example can be written as: $(\lambda y\, x.\, x\, y)$.

3. $(s\ t)$: an application, where s represents a function and t represents an argument. Both s and t are λ-terms. An application corresponds to a function call where s takes in t as the input. Examples: $(f\ x)$, $((\lambda x.\, x)\ f)$,

$(f\ (\lambda x.\, x\, x))$. Applications are *left-associative*. In the case of nested applications, the parentheses can be dropped. For example, $(f\, x\, y)$ is implicitly referring to $((f\, x)\, y)$.

In this thesis, I typically use characters such as x, y, z to refer to λ-variables, use characters like k to refer to integer variables, and use characters like s, t, f to refer to λ-terms.

The λ-abstractions can *bind* the corresponding variables underneath them, the ones that share the same variable name as the binding parameter of the abstraction. Accordingly, one can define variables to be *bound* or *free*.

Definition 2 (Bound and Free Variables)**.** *A variable x is said to be* bound *if it occurs within the scope of a λ-abstraction whose binding parameter is also x. When there are multiple binders with the same variable name, each occurrence of the variable is bound by the closest matching λ to its left. Otherwise, that occurrence of x is called* free.

A variable x occurs free *in an λ-term s if and only if one of the followings hold recursively:*

- *s is a variable and $s = x$;*
- *s is an abstraction of the form $(\lambda y.\, s')$ where $y \neq x$ and x occurs free in s';*
- *s is an application of the form $(s\, t)$ and x occurs free in either s or t.*

Note that it is possible for x to be both bound and free at the same time. For example, x can be bound in s and free in t.

Here are some examples to help with understanding bound and free variables:

- In $(\lambda x.\, x)$, the variable x is bound by $(\lambda x.)$.
- In $(\lambda x.\, x\, y)$, the variable x is bound by the $(\lambda x.)$ at the start, whilst y is free.
- In $(\lambda z.\, x\, y)$, neither x nor y is equal to z, so both are free.
- In $(\lambda x.\, (\lambda x.\, x))$, the variable x is bound by the inner most (right most) binding parameter $(\lambda x.\,)$.
- In $(\lambda y.\, (\lambda x.\, y))$, the variable y is bound by the outer most (left most) binding parameter $(\lambda y.\,)$.

- In $(\lambda x.\, x)\ (\lambda y.\, x)$, the variable x is both bound and free in this application. x is bound in the application body $(\lambda x.\, x)$. x is free in the application argument $(\lambda y.\, x)$.

These examples show that binding is determined entirely by the syntactic structure of the λ-terms.

There is another common terminology in formal reasoning for variables - *fresh* variables.

Definition 3 (Fresh Variable)**.** *A variable name that does not occur in any other term under consideration is considered as being fresh.*

Fresh variables are often used to replace existing variables in expressions to avoid variable capturing or shadowing.

Definition 4 (Closed λ-Terms)**.** *For any λ-term, if it does not contain any free variables, it is called* closed*.*

For example, $(\lambda x.\, x)$ is a closed λ-term, whilst $(\lambda x.\, y)$ is not.

In λ-calculus, the binding parameters are just placeholders, thus replacing the parameters with other fresh variables will not change the meaning of the expression. This replacing process is called α-renaming. For example, $(\lambda x.\, x)$ can be α-renamed to be $(\lambda y.\, y)$ and it still means the same function semantically. This is written as:

$$(\lambda x.\, x) =_\alpha (\lambda y.\, y),$$

where $=_\alpha$ represents α-equivalence.

Note that when replacing the binding parameters, only replace the variables that are bound by that parameter. For example, in the expression $(\lambda x.\, (\lambda x.\, x)\ x)$, by α-renaming the outer (leftmost) $(\lambda x.\,)$ to $(\lambda z.\,)$, we should obtain $(\lambda z.\, (\lambda x.\, x)\ z)$. The innermost variable x is untouched since it is bound by the innermost $(\lambda x.\,)$ instead. α-renaming is often used during reductions in order to avoid capture of free variables.

One thing to note is that in the syntax definition introduced at the beginning of this section, I used variable names such as x and y to notate variables. This notation is concise and simple, and thus widely used. However, it also has its shortcomings. It requires α-renaming to avoid variable capture, and allows many syntactically different but semantically equivalent λ-terms. This makes the manipulation of the λ-terms complicated and makes formalisation more difficult. In this section, I will keep using this notation for ease

of understanding. However, in the section about CBPV (Section 2.1.2) and in the formalisation work presented in later chapters, I will use a different variable representation that avoids these issues: the de Bruijn syntax (de Bruijn, 1972). The de Bruijn syntax eliminates the need for names by representing variables as natural numbers (de Bruijn indices) that indicate the number of binders between a variable occurrence and its corresponding λ-abstraction. For instance, the term $(\lambda x.(\lambda y.\, y\, x))$ can be expressed as $(\lambda.(\lambda.0\,1))$, where 0 represents the variable that is bound to the innermost λ, and 1 refers to the variable bound by the outer λ. Below are some more examples of explicit variable name notation converted into de Bruijn index notation:

- $(\lambda x.\, x)$ becomes $(\lambda.\, 0)$, since the variable refers to the immediately enclosing binder.
- $(\lambda x\, y.\, x)$ becomes $(\lambda\, \lambda.\, 1)$, since x is bound by the first (outer) abstraction.
- $(\lambda x\, y.\, y\, x)$ becomes $(\lambda\, \lambda.\, 0\, 1)$, where 0 refers to y and 1 refers to x.
- $(\lambda x.x)(\lambda y.y)$ becomes $(\lambda.0)(\lambda.0)$, where the first 0 refers to the x in the first abstraction, and the second 0 refers to the y in the second abstraction.
- $(\lambda x.x\,(\lambda y.x))$ becomes $(\lambda.0(\lambda.1))$, where both 0 and 1 refer to the same variable x.

This system avoids issues of α-equivalence and renaming, because all α-equivalent λ-terms have identical de Bruijn representation. This makes formalisation simpler and more uniform. However, indices must be carefully updated whenever a term crosses an abstraction boundary. This operation is called *lifting* (also known as *shifting*). For example, placing $(\lambda.\, 0\, 1)$ under an additional abstraction yields $(\lambda\, \lambda.\, 0\, 2)$. In this case the free variable 1 is lifted to 2 so that it continues to refer to the same variable, which now lies two steps out from the current abstraction layer instead of one. Similarly, lifting in the other direction is required when leaving an abstraction. For instance, after one-step β-reduction, the term $(\lambda.\, 0\, 1)(\lambda.\, 0)$ reduces to $(\lambda.\, 0)\, 0$. The result is not $(\lambda.\, 0)\, 1$ because the free variable now lies 0 step outside the abstraction, whilst it previously was 1 step outside.

Now that we know what λ-terms *are*, we want to know what we can *do* with those terms. In other words, how to *evaluate* (*reduce*) λ-terms to obtain results. Note that the terms *evaluate* (*evaluation*) and *reduce* (*reduction*) are often used interchangeably in λ-calculus.

Prerequisites

Before looking into the reduction rules, we need to first learn some prerequisites: substitution, and rules of inference.

Definition 5 (Substitution)**.** *$s[x := t]$ denotes the substitution of all the free occurrences of variable x in the λ-term s using a λ-term t. Substitution can be recursively defined over the structure of λ-terms:*

- $x[x := s] = s$
- $x[z := s] = x$ *if* $x \neq z$
- $(s\,t)[x := y] = (s[x := y])\,(t[x := y])$
- $(\lambda x.\, t)[x := s] = (\lambda x.\, t)$ *(the bound variable shadows the substitution variable)*
- $(\lambda y.\, t)[x := s] = (\lambda z.\, t[y := z][x := s])$ *if* $x \neq y$ *and* y *is free in* s*, where* z *is a fresh variable.*
- $(\lambda y.\, t)[x := s] = (\lambda y.\, t[x := s])$ *if* $x \neq y$ *and* y *is not free in* s*.*

Below is a more detailed explanation of the substitution process:

1. For variables, substitute it if it is the corresponding variable for the substitution, otherwise the substitution is terminated with no change made:
   - $x[x := s] = s$, in which x is substituted by s.
   - $x[z := s] = x$, in which no substitution happens since $x \neq z$.
2. For applications, it is rather straightforward, just recursively pass the substitution down:

   $$(s\,t)[x := y] = (s[x := y])\,(t[x := y])$$

   A more detailed example will be:

   $$\begin{aligned}&((\lambda x.\, f\,x)\,x)[x := y]\\ &= ((\lambda x.\, f\,x)[x := y]\;x[x := y])\\ &= ((\lambda x.\, f\,x)\,y)\end{aligned}$$

3. For abstractions, it is more complicated. There are a few different scenarios:

(a) The variable to be substituted *away* is the same as the binding parameter of the abstraction. In this case, the substitution is terminated with no change. For example:

$$(\lambda x.\, t)[x := s] = (\lambda x.\, t)$$

(b) The new expression that is *replacing* the variable contains free variables that are the same as the binding parameter. In this case, do an α-renaming for abstraction first using a fresh variable. Then, recursively substitute down in the abstraction body. For example:

$$\begin{aligned}
&(\lambda x.\, f\, z\, x)[z := x] \\
&=_\alpha (\lambda y.\, (f\, z\, x)[x := y])[z := x] \quad (y \text{ is a fresh variable}) \\
&= (\lambda y.\, (f\, z\, y))[z := x] \\
&= (\lambda y.\, (f\, z\, y)[z := x]) \\
&= (\lambda y.\, (f[z := x]\, z[z := x]\, y[z := x])) \\
&= (\lambda y.\, (f\, x\, y))
\end{aligned}$$

The α-renaming here avoids the new variable x being *captured* by the original binding parameter $(\lambda x.)$.

Another, more complicated example with the same logic:

$$\begin{aligned}
&(\lambda x.\, f\, z\, x)[z := (\lambda w.\, w\, x)] \\
&=_\alpha (\lambda y.\, (f\, z\, x)[x := y])[z := (\lambda w.\, w\, x)] \quad (y \text{ is a fresh variable}) \\
&= (\lambda y.\, (f\, z\, y))[z := (\lambda w.\, w\, x)] \\
&= (\lambda y.\, (f\, z\, y)[z := (\lambda w.\, w\, x)]) \\
&= (\lambda y.\, (f\, (\lambda w.\, w\, x)\, y))
\end{aligned}$$

(c) No overlap in variable names - recursively substitute down the abstraction body directly. For example:

$$\begin{aligned}
&(\lambda x.\, f\, z)[z := y] \\
&= (\lambda x.\, (f\, z)[z := y]) \\
&= (\lambda x.\, (f[z := y]\, z[z := y])) \\
&= (\lambda x.\, (f\, y))
\end{aligned}$$

Definition 6 (Rules of Inference)**.** *Rules of inference are a standard way of deriving conclusions from premises in formal logic. There are different formats to represent rules of inference, with a commonly used one as:*

$$\frac{P}{Q} \qquad \frac{P_1\ P_2\ \ldots\ P_n}{Q}$$

where P, P_1, P_2, P_n represent the premises and Q represents the conclusion.

The left rule concludes Q from a single premise P. the right rule concludes Q from (the conjunction of) a set of n premises $P_1, P_2, \ldots, P_n$.

Reduction Rules

Now that we know all the fundamentals, let's have a look at reduction rules.

The primary form of reduction is β-reduction, denoted by $s \to_\beta t$, meaning that s can be β-reduced to t. It is defined in terms of substitution, capturing the idea of function application. The base case of β-reduction is the application of a λ-abstraction to an argument, which results in the substitution of the argument for the bound variable in the body of the abstraction. β-reduction is formally defined as follows:

Definition 7 (One-Step β-reduction)**.**

$$\frac{}{(\lambda x.s)\ t \to_\beta s[x := t]} \qquad \frac{s \to_\beta s'}{s\ t \to_\beta s'\ t} \qquad \frac{t \to_\beta t'}{s\ t \to_\beta s\ t'} \qquad \frac{s \to_\beta s'}{\lambda x.s \to_\beta \lambda x.s'} \quad (\xi\text{-rule})$$

One-step β-reduction only allows a single reduction step at a time. To allow multiple reduction steps, we have multi-step β-reduction, defined as follows:

Definition 8 (Multi-Step β-reduction)**.**

$$\frac{}{s \to_\beta^* s} \qquad \frac{s \to_\beta s'}{s \to_\beta^* s'} \qquad \frac{s \to_\beta^* s' \quad s' \to_\beta^* t}{s \to_\beta^* t}$$

Note that in the rest of this thesis, $\to_\beta$ implicitly refers to $\to_\beta^*$.

Other than β-reduction, there is also *η-conversion*. Although this thesis does not make direct use of η-conversion, it is included here because it is an essential rule of the λ-calculus. η-conversion is a direct expression of extensionality for functions. Extensionality is the idea that two functions are considered equal if and only if they produce the same result for all possible inputs. Formally defined as follows:

Definition 9 (η-conversion)**.**

$$\frac{x \text{ is not free in } f}{(\lambda x.fx) \leftrightarrow_\eta f}$$

After applying the reduction rules, we will eventually reach a point where no more reduction can be applied:

Definition 10 (Normal Form)**.** *When a λ-term can no longer be reduced by any applicable reduction rules, it is said to be in* normal form.

With all the above rules except the ξ-rule in β-reduction, we obtain the weak λ-calculus (Plotkin, 1975), which is a widely studied and applied variant of the λ-calculus. All the variants of λ-calculus that I work with in this thesis are weak. This λ-calculus is called *weak* because it does not include a rule that allows reductions under abstractions. In the weak λ-calculus, normal forms are precisely variables and abstractions, since both are irreducible in this setting. By adding the ξ-rule to the weak λ-calculus, one can obtain the *strong* λ-calculus (also called the *full* λ-calculus). For the rest of the thesis, I will leave out the ξ-rule, since we are working with the weak λ-calculus.

One might naturally ask: why choose the weak λ-calculus when the strong λ-calculus appears to be more general? It is important to emphasise that the terms *weak* and *strong* do not imply that one system is inherently *better* or *worse*. Rather, they reflect different choices of expressivity, each with its own advantages and limitations depending on the intended application. The weak λ-calculus is particularly popular because it provides a simpler and more operationally convenient model of computation. By disallowing reductions under abstractions, it more closely reflects the behaviour of real-world functional programming languages, where evaluation typically occurs at the outermost level and under controlled strategies, rather than arbitrarily deep within function bodies. This makes weak reduction a natural foundation for studying program execution.

Evaluation Strategy

As you might have noticed, for the same λ-term, there are many potential ways we can apply the β-reduction rules. For example, for a λ-term

$$((\lambda x\, y.\, x\, y\, y)\, f)((\lambda x.x)(\lambda y.y)),$$

we can reduce the left subterm $((\lambda x\, y.\, x\, y\, y)\, f)$ first, or we can reduce the right subterm $((\lambda x.x)(\lambda y.y))$ first. If we perform one step β-reduction on the left subterm first and obtain

$$(\lambda y.\, f\, y\, y)((\lambda x.x)(\lambda y.y)),$$

we now have another two choices: to reduce the right subterm $((\lambda x.x)(\lambda y.y))$ first, or to reduce the whole term first by directly substituting the right subterm into the left subterm.

In the above example, regardless of the evaluation order, we still obtain the same results. However, there are also cases where different evaluation order actually result in diverging results. Let's take a look at the following example:

$$(\lambda x.\, f)((\lambda x.\, x\, x)(\lambda x.\, x\, x))$$

If we evaluate the right-hand side first, we will realise that it is an infinite loop. On the other hand, if we evaluate the top level application first, we will obtain f straight away. The infinite loop in fact has a name, the ω combinator.

Having all these possible choices makes β-reduction non-deterministic. At the same time, when we implement a programming language based on λ-calculus, we want the evaluation to be deterministic. Thus, we need to fix the evaluation choices when evaluating in λ-calculus. The different sets of β-reduction rules are called evaluation strategies.

Commonly used evaluation strategies are call-by-value, call-by-name, and call-by-need.

Call by Value The *call-by-value* evaluation strategy requires the function argument to be *fully evaluated* before substituting it into the abstraction body. It is used by, for example, the Standard ML programming language (Milner, Tofte, and Harper, 1990). The inference rules for the call-by-value evaluation strategy are formally defined as:

Definition 11 (Small-Step Semantics for Call-By-Value)**.**

$$\frac{s \to_\beta s'}{s\, t \to_\beta s'\, t} \quad (1)$$

$$\frac{t \to_\beta t'}{(\lambda x.s)\, t \to_\beta (\lambda x.s)\, t'} \quad (2)$$

$$\frac{}{(\lambda x.s)(\lambda x.t) \to_\beta s[x := (\lambda x.t)]} \quad (3)$$

To be more specific, whenever we have an application $(s\,t)$, we first follow rule (1) and reduce its function body s. If s is already in normal form, we then follow rule (2) and reduce its argument t. If both function body and function argument are in normal forms, we then follow rule (3) and evaluate the whole application by applying substitution.

Call by Name In the *call-by-name* strategy, the substitution happens first and the evaluation of the function argument is deferred. It exists more in the theoretical side and has motivated other evaluation strategies, but itself is not commonly adopted for developing programming languages. The inference rules are formally defined as:

Definition 12 (Small-Step Semantics for Call-By-Name)**.**

$$\frac{s \to_\beta s'}{s\,t \to_\beta s'\,t} \quad (1)$$

$$\frac{}{(\lambda x.s)\,t \to_\beta s[x := t]} \quad (2)$$

For a given application $(s\,t)$, the first step is to reduce the function body s to normal form following rule (1). The second step is where it diverges for the call-by-name strategy compared with the call-by-value strategy. The application can be evaluated straight away as long as the function body s is in normal form. It does not require the function argument t to be in normal form already. In this case, rule (2) allows the unreduced t to be substituted into the function body right away.

Call by Need The *call-by-need* strategy is the canonical evaluation strategy for lazy functional programming languages, most notably Haskell (Hudak et al., 2007), which has been highly influential in both research and practical development of functional programming. The formal definition of call-by-need is complicated and not directly related to this thesis, and thus not presented here. Instead, I will provide an intuitive explanation of how call-by-need works.

Call-by-need is closely related to call-by-name, but with two key refinements: expressions are only evaluated when needed, and evaluation results are shared. In the call-by-need strategy, function calls are evaluated only

when their results are actually needed, and these results are stored for future use. Specifically, if a term s is substituted into multiple positions, its evaluation at the first position is stored and reused at the subsequent positions. This sharing avoids repeated evaluations of the same expression, thereby improving efficiency compared with the call-by-name strategy.

Call by Push Value As you might guess, choosing a different evaluation strategy will result in different run-time efficiency of functional programs. Each strategy has its own advantages and shortcomings, and thus is suitable for different scenarios. Is it possible to have a λ-calculus that allows us to choose when to use which strategy for the appropriate scenarios?

The *call-by-push-value* λ-calculus (CBPV) by Levy (1999) is a variant of λ-calculus. It gives the programmers the ability to specify when function calls evaluate on a function-per-function basis. As such, various strategies can be encoded within the same calculus without sacrificing determinism. More specifically, CBPV provides flexibility in choosing when to use which strategy without having to write multiple variants for each strategy. This enables programmers to specify how they want their program to be evaluated within the calculus. We dive deeper into CBPV in the following section.

2.1.2 The Call-By-Push-Value λ-Calculus

The call-by-push-value (CBPV) λ-calculus was first introduced by Levy (1999) in a paper and was subsequently refined in his PhD thesis (Levy, 2001). It is a variant of λ-calculus that is designed to allow evaluation strategies to be encoded as part of the program instead of the default setting in the language. In other words, it is able to alternate evaluation strategies depending on what instructions are encoded in the program. Hence, it serves as a subsuming paradigm that enables studying various common evaluation strategies, and combinations thereof, using a single set of reduction rules. For instance, one can represent CBPV terms that can interleave the call-by-name and the call-by-value reduction strategies. Levy provides semantic-preserving translations from call-by-name and call-by-value into CBPV (Levy, 2001).

Note that I work with an untyped subset of CBPV in this thesis. I take the core part of CBPV while preserving its most essential functionalities. In the rest of the thesis, whenever I mention CBPV, it is referring to the untyped subset of CBPV that I work with, unless otherwise specified.

Syntax In CBPV, terms are divided into two syntactic classes: *values* and *computations*, as shown in Definition 13:

Definition 13 (Syntax for CBPV)**.**

$$
\begin{array}{llllll}
\textit{Values} & \mathtt{V} & := & \mathsf{var}\ x \mid & \mathsf{thunk}\ \mathtt{M} & \\
\textit{Computations} & \mathtt{M}, \mathtt{N} & := & \lambda.\mathtt{M} \mid & \mathsf{app}\ \mathtt{M}\ \mathtt{V} \mid & \mathsf{ret}\ \mathtt{V} \mid \quad \mathsf{force}\ \mathtt{V} \mid \\
 & & & \mathsf{seq}\ \mathtt{M}\ \mathtt{N} \mid & \mathsf{let}\ \mathtt{V}.\ \mathsf{in}\ \mathtt{M} &
\end{array}
$$

As Levy emphasises: "A value *is*, a computation *does*." (Levy, 2001, p. 20), let's look into more detail of CBPV constructors with this in mind:

1. *Variable* ($\mathsf{var}\ x$): This constructor represents a variable value. Here, x is a natural number representing the de Bruijn index of the variable. Example: $(\mathsf{var}\ 0)$ refers to the innermost variable.

2. *Return* ($\mathsf{ret}\ \mathtt{V}$): This constructor is used to create a new computation from a value $\mathtt{V}$. It lifts the value into the computation context, allowing it to be used in further computations. Example:

   $$(\mathsf{ret}\ (\mathsf{var}\ 0))$$

   creates a computation that returns the innermost variable.

3. *Thunk* ($\mathsf{thunk}\ \mathtt{M}$): This constructor is used to delay the evaluation of a computation $\mathtt{M}$ by wrapping it up as a thunk value. It allows for the suspension of computation until it is explicitly forced. It behaves the same as abstraction in λ-calculus. Example:

   $$(\mathsf{thunk}\ (\mathsf{ret}\ (\mathsf{var}\ 0)))$$

   represents a suspended computation that, when forced, will return the innermost variable.

4. *λ-Abstraction* ($\lambda.\mathtt{M}$): This constructor represents a function definition. It takes a computation $\mathtt{M}$ as its body and creates a function that can be applied to arguments. Example:

   $$(\lambda.\ (\mathsf{ret}\ (\mathsf{var}\ 0)))$$

   defines a function that, when applied, will return its argument.

5. *Application* (app M V): This operator is used to apply a computation M (which should evaluate to a function) to a value V. It represents the function call operation. While it mostly works the same as application in λ-calculus, it has the distinction that the function argument in this case is of a value type that can not be "reduced". Example:

$$(\mathsf{app}\ (\lambda.\ (\mathsf{var}\ 0))\ (\mathsf{var}\ 0))$$

applies the identity function to the innermost variable.

6. *Force* (force V): This constructor is used to evaluate a thunked value V. It unwraps the thunk and computes the underlying value, effectively resuming the suspended computation. Example:

$$(\mathsf{force}\ (\mathsf{thunk}\ (\mathsf{ret}\ (\mathsf{var}\ 0))))$$

will evaluate to $(\mathsf{ret}\ (\mathsf{var}\ 0))$.

7. *Sequencing* (seq M N): This operator is used to sequence two computations M and N. The result of the first computation is passed as input to the second computation. Example:

$$(\mathsf{seq}\ (\mathsf{force}\ (\mathsf{thunk}\ (\mathsf{ret}\ (\mathsf{var}\ 0))))\ (\mathsf{app}\ (\lambda.\ (\mathsf{var}\ 0))\ (\mathsf{var}\ 0)))$$

first evaluates the force operation and then substitutes the result to the inner most free variable of the second computation - which is the application in this case.

8. *Let Binding* (let V. in M): This operator is used to bind a value V into a computation M. It allows for the introduction of new variables and the manipulation of values within the computation. Let binding works very similar to the sequencing binding, except that it binds a value to a computation instead of the result of one computation into another. Example:

$$(\mathsf{let}\ (\mathsf{var}\ 0).\ \mathsf{in}\ (\mathsf{app}\ (\lambda.\ (\mathsf{var}\ 0))\ (\mathsf{var}\ 0)))$$

binds the innermost variable to the application computation.

The notations that I use in this thesis differ slightly from Levy's original notations in his thesis (Levy, 2001). The differences are summarised in Table 2.1. The main differences are in variable notation, application notation, and the naming of the return constructor.

Levy's Notation	My Notation
x (explicit name)	var x (de Bruijn index)
thunk M	thunk M
λx.M	λ. M
V'M	app M V
produce V	ret V
force V	force V
M to x.N	seq M N
let x be V . M	let V. in M

TABLE 2.1: Comparison of CBPV Notations

Now that we know what CBPV terms are, let's look into how to reduce them. Substitution in CBPV works the same way as in λ-calculus. In this thesis, since we use de Bruijn indices for variable notation, the detailed substitution process is slightly different from the one described in Definition 5 for explicit variable names. The main difference is that we no longer need to do α-renaming to avoid variable capture. Instead, we need to take care of the shifting of de Bruijn indices when going under an abstraction. I use the notation m_v^n to denote the substitution of the variable with de Bruijn index n in computation m by value v. The detailed definition of substitution with de Bruijn indices will be introduced in the later chapters when it is needed.

In the previous sections, we presented the small-step semantics for call-by-value and call-by-name λ-calculi. Similarly, we can also define the small-step semantics for CBPV. We first define the primitive step relation $m \to m'$ in Figure 2.1, which captures the fundamental reduction steps that can be performed on CBPV terms.

$$\frac{}{\mathsf{force}\,(\mathsf{thunk}\,m) \to m} \qquad \frac{m_v^0 = m'}{\mathsf{app}\,(\lambda.\,m)\,v \to m'} \qquad \frac{m_v^0 = m'}{\mathsf{let}\,v.\,\mathsf{in}\,m \to m'}$$

$$\frac{n_v^0 = n'}{\mathsf{seq}\,(\mathsf{ret}\,v)\,n \to n'}$$

FIGURE 2.1: Primitive Step for CBPV

What makes CBPV special is the *force* and *thunk* pair. With this pair of operators, CBPV is able to postpone computations by converting them into values (thunk) and then later unpack and evaluate these thunked values using the computation constructor force.

Using the primitive step relation, we can now define the small-step semantics for CBPV in Figure 2.2. The judgement $m \rightsquigarrow m'$ means that the computation m reduces to the computation m' in one small step. Note that only computations can be reduced, values are irreducible.

$$\frac{m \to m'}{m \rightsquigarrow m'} \qquad \frac{m \rightsquigarrow m'}{\mathsf{app}\, m\, v \rightsquigarrow \mathsf{app}\, m'\, v} \qquad \frac{m \rightsquigarrow m'}{\mathsf{seq}\, m\, n \rightsquigarrow \mathsf{seq}\, m'\, n}$$

FIGURE 2.2: Small-Step Semantics for CBPV

I introduced the small-step semantics so far for all the λ-calculi. This choice was made for the ease of understanding, since small-step semantics provides a more fine-grained view of the reduction process. In the rest of the thesis, however, I will use big-step semantics for all the λ-calculi that I work with, including CBPV. Big-step semantics focuses more on the overall structure of the computation and initial and final states, rather than the individual steps. It provides a more abstract view of the computation process, which is often more convenient for reasoning about program behavior. This makes big-step semantics easier to work with when defining cost models, which is the main focus of this thesis. I will introduce the relevant big-step semantics for each λ-calculus in the respective chapters when it is needed.

2.1.3 Turing Machines

The *Turing machine*, introduced by Alan Turing in 1936 (Turing, 1936), is the canonical model of computation in a lot areas in computer science, including the theory of computation, complexity theory, and the foundations of mathematics. Turing machines capture the intuitive notion of step-by-step symbolic manipulation on an infinite tape. Importantly, the λ-calculus and Turing machines are equivalent in expressive power, meaning that any computable function defined in one model can also be defined in the other. The implementation details of the Turing machine model is not as relevant in this thesis, but I will do a quick introduction here just to establish a common understanding. A Turing machine can be formally defined as follows from Sipser (2013):

Definition 14 (Turing Machine)**.** *A (deterministic) Turing machine is a tuple*

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$$

where Q, Σ and Γ are finite sets, and

- *Q is the set of states,*
- *Σ is the input alphabet not containing the blank symbol $\sqcup$,*
- *Γ is the tape alphabet, where $\Sigma \subseteq \Gamma$ and $\sqcup \in \Gamma$,*
- *$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,*
- *$q_0 \in Q$ is the start state,*
- *$q_{accept} \in Q$ is the accepting state,*
- *$q_{reject} \in Q$ is the rejecting state, where $q_{reject} \neq q_{accept}$.*

Intuitively, a Turing machine consists of a finite set of states, an infinite tape made of cells where each of them contain a symbol from the tape alphabet Γ, and a head that can read and write symbols and move left or right according to the transition function δ. Computation proceeds step by step until the machine halts in either the accepting or rejecting state.

Turing machines thus provide a rigorous, low-level operational view of computation. For this reason, it is used as a standard model of computation in complexity studies, which we will go into more detail in Section 2.2.2.

2.2 Computational Complexity

In this section, I introduce computational complexity and the related concepts in complexity theory that are used in this thesis, namely the big $\mathcal{O}$ notation and reasonable cost models.

Computational complexity is concerned with classifying computational problems according to the quantifiable resources needed to solve them. The primary resources of interest are time (how long a computation takes) and space (how much memory it uses). These problems are categorised into different *complexity classes* according to the resources they require, such as P (polynomial time), NP (nondeterministic polynomial time), PSPACE (polynomial space), and EXP (exponential time).

To reason about such classes formally, it is essential to express how resource usage grows with input size, which is captured using the big $\mathcal{O}$ notation.

2.2.1 Big $\mathcal{O}$ Notation

I believe that most readers are already familiar with Big $\mathcal{O}$ notation, but for the sake of completeness, I provide a brief recapitulation here.

Big $\mathcal{O}$ notation is a mathematical formalism used to describe the asymptotic behavior of functions, particularly in the context of algorithm analysis. It provides an upper bound on the growth rate of a function, allowing one to reason about the efficiency of algorithms as the size of the input becomes large.

Formally, let $f(n)$ and $g(n)$ be functions mapping natural numbers to non-negative real numbers. We write

$$f(n) \in \mathcal{O}(g(n))$$

if there exist constants $C > 0$ and $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$,

$$f(n) \leq C \cdot g(n).$$

In algorithm analysis, $f(n)$ often represents the running time or space usage of an algorithm as a function of input size n, while $g(n)$ is a simpler, commonly understood function that captures the dominant growth trend (e.g., n, $n \log n$, n^2, 2^n).

Big $\mathcal{O}$ notation abstracts away constant factors and lower-order terms, providing a high-level measure of performance that is independent of specific hardware or implementation details. This makes it a fundamental tool for comparing algorithms and understanding their scalability.

Big $\mathcal{O}$ notation is often used in conjunction with other asymptotic notations, such as big Ω (which provides a lower bound) and big Θ (which provides a tight bound). Together, these notations form a comprehensive framework for analyzing and classifying algorithms based on their resource requirements. In this thesis, only Big $\mathcal{O}$ notation will be used.

Big $\mathcal{O}$ notation describes the asymptotic growth of functions, but it requires a cost model. That is, a definition of what counts as "one" computational step.

2.2.2 Reasonable Cost Models

In order to carry out cost analysis on computational models, we first need appropriate *cost models*. For instance, in the case of the λ-calculus, how should we define a unit of time? What constitutes a unit of space? Once such a cost

model is in place, another important question arises: how do we know that the chosen cost model is in fact useful? This is where the notion of *reasonable* cost models becomes central.

Turing machines are the standard computational model in complexity theory. They operate at a very low level, which makes them particularly suitable for reasoning about time and space costs. Specifically, they provide an obvious cost model: time is measured by the number of machine transitions, and space is measured by the number of cells used on the tape.

In contrast, the λ-calculus is more abstract, which makes it harder to assign concrete costs to computations. At the same time, this very abstraction makes λ-terms far more readable and convenient to use than Turing machines. This naturally raises the question: can cost analysis carried out in one model be meaningfully transferred to the other? Put differently, is there a way to program in λ-calculus but perform cost analysis using equivalent Turing machines?

This question leads to the notion of a *reasonable* model of computation with respect to Turing machines (Slot and van Emde Boas, 1984; van Emde Boas, 1990). The concept of reasonableness serves as a standard criterion in complexity theory for determining whether a computational model is suitable for reasoning about complexity by comparison with Turing machines, which are considered reasonable by definition. Formally, this is captured by the *invariance thesis* (van Emde Boas, 1990, page 5):

> *"Invariance Thesis: Reasonable machines simulate each other with polynomially bounded overhead in time and constant factor overhead in space."*

Note that some literature uses *invariance thesis* to refer to the time invariance thesis, and *strong invariance thesis* to refer to the full invariance thesis. In this thesis, I use both to refer to the full invariance thesis, unless specified otherwise. One can think of the invariance thesis as an extension of the Church–Turing thesis: not only do all reasonable models compute the same class of functions, but they also preserve the same complexity classes. Throughout this thesis, I use the term *reasonable* strictly in this technical sense, without reference to its everyday meaning.

It is worth noting that the term *reasonable* is somewhat overloaded: (1) it can be a property of computational models, (2) it can describe the measures used for cost, and (3) it can apply to cost models themselves. In other words, we speak of reasonable machines when referring to computational

models that can simulate Turing machines in both directions with polynomially bounded time overhead and constant factor space overhead; we speak of reasonable measures when referring to the cost measures that allow a computational model to be reasonable.

Importantly, a reasonable cost model need not be efficient (Accattoli, 2017), nor does it need to resemble practical implementation strategies used in real-world programming languages. Its purpose is instead to preserve asymptotic complexity classes. For example, reasonable cost models guarantee that standard classes such as P, PSPACE, and EXP are independent of the underlying computational substrate. Concretely, we may define a problem to be in P if it can be solved in (a) a polynomial number of Turing machine steps, or (b) a polynomial number of β-reductions in λ-calculus. Without the invariance thesis, there is no guarantee that these two definitions coincide: if simulating λ-calculus on a Turing machine introduced exponential overhead, membership in P(b) would not imply membership in P(a). P(a) and P(b) coincide only if we can show that λ-calculus is reasonable with this measure.

2.3 Formal Verification

In this section, I introduce formal verification, interactive theorem proving and the prover HOL4, which are the main techniques and tool I use in my thesis to verify the cost models for CBPV.

In the modern world, where many activities—industrial, medical, and even everyday ones—rely heavily on computerised devices. The demand for reliability in both software and hardware has grown significantly.

History has shown that malfunctions in such systems can have consequences of varying scales. For instance, the Therac-25 accidents (Leveson and Turner, 1993), where the computerised radiation therapy machine suffered from software faults, led to massive overdoses, resulting in severe harm to patients and even fatalities. Such issues remain pressing in recent years. In 2023, Britain's air traffic control system suffered a meltdown, affecting over 700,000 passengers and causing substantial costs to airlines and airports due to the collapse of the National Air Traffic Services computer system (*Independent Review of NATS (EnRoute) Plc's Flight Planning System Failure (Final Report)* 2024). On the hardware side, Intel reported processor vulnerabilities in 2023 that could potentially expose sensitive data (*CVE-2023-23583 Intel Processor Vulnerability in NetApp Products* 2023).

These examples prompt a critical question: Are there principled approaches we can adopt to enhance the reliability of our software and hardware systems? More specifically, can we design systems whose behaviour is provably correct with respect to safety-critical properties, even under all possible operational scenarios?

Formal methods are techniques for system design in both software and hardware. They provide a rigorous approach to ensuring safety by modelling systems as mathematical entities. While testing is essential for uncovering errors, it can only cover a finite number of cases; in other words, it can only guarantee correctness in the situations tested. Formal methods provide an additional layer of assurance: once a system design is verified, the system must behave as proven, even beyond tested cases. In this framework, system behaviour is formalised using mathematical structures, where states are represented as abstract descriptions, and the entire state space can then be verified.

Similarly, we can also use formal methods to increase our confidence in theoretical results, such as mathematical proofs. For instance, the Flyspeck project (Hales et al., 2015) provides a formal proof of the Kepler conjecture. Despite its significance, the original proof (Hales, 2005) for the Kepler conjecture took years to write, and is famously complicated. The reviewer Lagarias commented that, "The nature of this proof, ..., makes it hard for humans to check every step reliably." (Lagarias, 2011) The formal proof ensured that these proof steps are indeed correct by machine checking them. As we can see, employing formal methods in theoretical research increases the reliability and reusability of the results.

Formal verification can be achieved through techniques such as model checking (Clarke and Emerson, 1981; Queille and Sifakis, 1982; Clarke, Emerson, and Sistla, 1986; Clarke, Grumberg, and Peled, 2001), theorem proving (Milner, 1972; de Bruijn, 1983; Davis, 1957; Gilmore, 1960; Davis and Putnam, 1960; Wang, 1960; Prawitz, Prawitz, and Voghera, 1960; Robinson, 1965; Loveland, 1968), and static analysis (Johnson, 1977; Cousot and Cousot, 1977; Wichmann et al., 1995).

Theorem proving is widely used in verifying theoretical results. It was in fact, the verification technique that was used in the Flyspeck project that we mentioned earlier. In my thesis, I employ interactive theorem proving to verify my results, with the focus on verifying the cost models for call-by-push-value λ-calculus.

2.3.1 Interactive Theorem Proving

Theorem provers are software tools that machine-check logical statements given by users; such statements can include program properties, mathematical statements, etc. Theorem proving has been a body of research since the 1950s (Harrison, Urban, and Wiedijk, 2014), where Davis (1957), Gilmore (1960), Davis and Putnam (1960), Wang (1960), and Prawitz, Prawitz, and Voghera (1960) were looking into computer-assisted proofs and Robinson (1965) and Loveland (1968) looked into automated theorem proving. In interactive theorem proving, a human user and the machine work collaboratively to produce a formal proof. The machine assists the human by checking the detailed formal proofs produced by the human. Thus, interactive theorem provers are also called proof assistants. There are certain levels of automation in interactive theorem provers, where simple proofs can be generated by the machine. However, human guidance is still essential in most of the proof steps. Thus, interactive theorem proving requires extensive human effort. On the other hand, in automated theorem proving, the machine is supposed to fully automatically generate proofs for given assertions. But given a theorem statement, providing a proof is an undecidable problem. The automated theorem provers are only able to generate proofs for theorems at a certain complexity; for more complicated theorems, human intelligence is required to solve the challenging parts of the proofs. Automated theorem provers also tend to require a longer time to run in order to generate the proofs. Thus, interactive theorem provers and automated theorem provers have their own advantages and disadvantages.

You might have a question now: what if the prover has bugs? There are a few different ways to deal with this, one can be from the root of the provers - their design, another one can be from after they are implemented - verifying the provers. LCF (Gordon, Milner, and Wadsworth, 1979; Gordon, 2000; Harrison, 2001) addresses this concern by providing a design framework for constructing reliable theorem provers. The LCF approach assigns a special abstract type `thm` of theorems, only allows derivation of theorems using the inference rules given by the constructors of the `thm` abstract type, further more, it enforces the implementation of the prover to be in a strongly-typed high-level language. This design approach enforces the logical correctness. Another approach is to verify the correctness of the prover itself in provers, which can be done by formalising the prover's implementation in a higher-order logic framework and then proving its properties within that framework (Sozeau et al., 2020; Harrison, 2006).

There exist a number of widely used interactive theorem provers in the area, such as the HOL4 theorem prover (Gordon and Melham, 1993; Slind and Norrish, 2008), the Rocq Proof Assistant (2025) (formerly known as Coq), Isabelle/HOL (Paulson, 1986; Nipkow, Paulson, and Wenzel, 2002; Nipkow and Klein, 2014), and Lean (Moura et al., 2015; Moura and Ullrich, 2021). They are developed based on different axioms and logical frameworks, each suitable for different specific purposes. In the case of my project, I use the HOL4 theorem prover. It allows us to describe the syntax and semantics of λ-calculus with various evaluation strategies, and to define and verify the cost models I develop for CBPV.

2.3.2 HOL4

HOL4 is an implementation of higher order logic, in particular, predicate calculus with terms from simple type theory. HOL4 is implemented in Standard ML (Milner, Tofte, and Harper, 1990), a general-purpose functional programming language which is popular in the development of theorem provers.

There are several reasons why I chose HOL4 as the theorem prover for my project. In comparison to higher-order logic provers, dependently-typed provers are more expressive in ways that can be important when formalising certain mathematical structures, as in category theory. In this work, however, we do not need to go beyond mutually inductive datatypes, and inductive relations and recursive functions over them. HOL4 is sufficiently expressive for that. There are also Boyer-Moore provers (Boyer, Kaufmann, and Moore, 1995), but they might not be expressive enough for the induction and recursion we need because they're first-order logic based. This still leaves many candidates, but I have previous experience with formalising models of computation in HOL4. During my undergraduate studies, I completed individual research projects and my honours thesis using HOL4. I was also involved in group formalisation projects with the TCS reading group at the Australian National University. The topics I worked on include register machines, regular languages, linear temporal logic and topology.

The general approach to work with HOL4 is that we can first create an empty new theory file, then optionally import existing theories and libraries. Theories contain definitions and proofs, and libraries contain arbitrary ML code (tools to help with proof automation, simplification, etc.). Then we can make new definitions and/or define and prove new theorems under this theory. After we are done with this theory, we will be able to export it and use it

as a library when we are making other new theories.

One thing to notice is for the proofs in this thesis and the ones in the HOL4 formalisation, I only use natural numbers (including 0). So when I am talking about numbers, if not specified, they will mean natural numbers. In this thesis, I abstract away the HOL4 syntax and present the definitions and theorems using natural language and mathematical language. For the ones interested to understand the actual code of my formalisation, detailed HOL4 syntax and examples can be found in the description manual from HOL4's official website (HOL, 2025).

2.4 Related Work

Here I survey related work in the development of cost models for variants of the λ-calculus. I also survey the existing research on CBPV.

Over the years, various models of computations (Davis, 1958; Minsky, 1967; Davis, 1982; Kleene, 1981; Cook and Reckhow, 1973; Hartmanis, 1981) have been studied. Among these models, Turing machines and the Random Access Machine (RAM) (Cook and Reckhow, 1973) are the more practical in reasoning about complexity, thus taking the central stage in complexity theory.

Slot and van Emde Boas (1984) formally proposed the invariance thesis, which states that all reasonable machines can simulate each other with polynomially bounded overhead in time and constant factor overhead in space. More specifically, they defined Turing machines to be reasonable by definition, and further showed that RAM satisfies this simulation requirement. This thesis has been widely accepted in the complexity theory community. van Emde Boas (1990) further discussed the invariance thesis and provided more examples of reasonable machines and the simulations. While many sequential (low-level) models of computations are shown to be reasonable, cost models for functional (high-level) languages, is still an active research area.

Frandsen and Sturtivant (1991) provides a time cost model for the full λ-calculus as the combination of the size of the term, the size of the normal form of the term, and the minimum number of parallel β-reduction steps to normal form. Following that, Asperti and Mairson (1998) showed that parallel β-reduction has extremely high worst case cost. Thus it is not an ideal choice for cost measure. Lawall and Mairson (1996) provided a different measure. They proved that the full λ-calculus is reasonable for both time and space using the measures "total ink used" and "maximum ink used". The

time measure of "total ink used" is too general, thus hard to apply to real complexity analysis.

Blelloch and Greiner (1995) and Sands, Gustavsson, and Moran (2002) addressed cost for call-by-value λ-calculus, while studying efficiency for functional languages. Dal Lago and Martini (2008) later provided a time measure for WCBV. They define the time cost by counting the number of β-steps while taking account of the size of β-redexes. They also proved the time invariance of weak call-by-name λ-calculus (Dal Lago and Martini, 2012) following the same idea. This was further strengthed by Accattoli and Dal Lago (2016) in 2016, showing that counting (leftmost-outermost) β-steps makes the full λ-calculus reasonable for time. Continuing on this line, Accattoli, Condoluci, and Coen (2021) showed that the strong call-by-value λ-calculus is reasonable for time. They achieved this by developing an abstract machine, called the strong crumbling abstract machine (SCAM), that encodes useful sharing.

The above work all focuses on time complexity, omitting the space measure, let alone the strong invariance property. Until Forster, Kunze, and Roth (2020) proved that WCBV is reasonable for both time and space with respect to natural measures, accompanied with a partial formalisation. They define a natural measure to be the number of β-reductions for time, and the size of the biggest intermediate term for space. Their research enables the formal verification of complexity-theoretic proofs concerning complexity classes P, NP and PSPACE. They proved that WCBV is reasonable by interleaving two strategies: the substitution-based strategy and heap-based strategy, which I adapt and implement for CBPV in my thesis. Later, Forster et al. (2021) provided a complete formalisation for the time invariance thesis for WCBV. The complete formal verification of the space invariance thesis for the same calculus still remains open. One limitation of this line of work, as well as ours, is that it does not consider sublinear time or space classes.

While space complexity of λ-calculus is not as thoroughly studied as time invariance, there has been work looking into sublinear time complexity for variants of the λ-calculus (Ghica, 2007; Schöpp, 2007; Mazza, 2015; Dal Lago and Schöpp, 2016). Accattoli, Dal Lago, and Vanoni (2022) presented a reasonable space cost model for the pure λ-calculus that works for LOGSPACE. This is achieved by separating space into the *input space* and the *work space*. The input space is represented by λ-terms, and the work space is represented by using a variant over the Krivine abstract machine (KAM) (Krivine, 2007).

Environment-based abstract machines usually exhibit the problem of exceeding space bound requirement due to the poor management of garbage collection and the use of pointers. The Space KAM introduced in their work solves this issue by encoding eager garbage collection and environment unchaining. While this approach is not applicable to time invariance with natural measures, they showed that low-level execution time of the Space KAM is an alternative reasonable time cost model. They further demonstrated that their space cost model is very robust by transporting the result to the call-by-value λ-calculus. It seems promising to adapt their approach to CBPV, however, the nature of Space KAM (giving up environment sharing) makes it unreasonable for time with natural measures. This contradicts with my aim of developing a cost model for CBPV that is reasonable for both time and space with respect to natural measures. Thus while their machine achieves great things on space cost, in this thesis, I do not follow their approach.

CBPV The CBPV developed by Levy (1999) in 1999 has been increasingly popular in the last decade. There are various extensions of CBPV, including with stacks (Levy, 2002), with probability (Ehrhard and Tasson, 2016), with call-by-need (McDermott and Mycroft, 2019), with effect and coeffect tracking (Torczon et al., 2024) and with effects (McDermott, 2025). There is a similar λ-calculus called the Bang-calculus (Ehrhard and Guerrieri, 2016), which can be regarded as an untyped version of CBPV without any side-effects. On the formalisation side, there is a formal equational theory (Rizkallah, Garbuzov, and Zdancewic, 2018) for CBPV; there is another formalisation (Forster et al., 2019) that includes proofs for its operational, equational, and denotational theory. On the applied side, there are projects such as extracting recurrences (Kavvos et al., 2020) using CBPV. On the semantic side, other than some of the already mentioned work (Rizkallah, Garbuzov, and Zdancewic, 2018; Forster et al., 2019; McDermott and Mycroft, 2019), Goncharov, Tsampas, and Urbat (2025) recently developed a theory of program equivalence for CBPV.

As we can see in the increasing amount of papers related to CBPV, it is gaining more and more traction in the programming languages community. Given the high potential of CBPV, however, to the best of my knowledge, there is no existing work on developing cost models for CBPV. My thesis fills this gap by developing a cost model for CBPV that is reasonable for both time and space with respect to natural measures. This work also substantiates

prior research that uses CBPV for cost analysis, such as recurrence extraction for functional programs by Kavvos et al. (2020), where CBPV serves as an intermediate language. Ultimately, this thesis establishes CBPV as a reasonable model of computation for program complexity analysis, providing a foundation for studying complexity classes such as P, NP and PSPACE.

Chapter 3

Weak Call-By-Value is Reasonable for Time and Space in HOL4

In this chapter, I formalise the time and space cost models for the weak call-by-value λ-calculus (WCBV) in HOL4. This formalisation is based on prior work by Forster, Kunze, and Roth (2020), which formalised reasonable cost models for WCBV in the Rocq Proof Assistant (2025).

I first give an overview of the proof strategy in Section 3.1. I then formalise WCBV in Section 3.2. Following that, I then move on to the compiler from WCBV terms into programs in Section 3.3. Next, I develop the intermediate abstract machines in Section 3.4 and verify their simulation to WCBV. In the end, I put everything together in Section 3.5. I recapitulate the proof by Forster, Kunze, and Roth (2020), showing that WCBV is reasonable by an interleaving algorithm, which I also adopt for the reasonable proof in my cost models for CBPV in Chapter 4.

My formalisations for WCBV are all done in HOL4 and can be accessed from my GitHub repo (Chen, 2022b). It is also reviewed and included in the official HOL4 library (HOL, 2022). This port confirms the correctness of the original formalisation by Forster, Kunze, and Roth (2020) in a different theorem prover with a different logical foundation. Importantly, this port also provides a foundation for my subsequent work on cost models for CBPV in Chapter 4.

Note that I have given the pen-and-paper definitions of WCBV in the previous chapter (Section 2.1.1), including its syntax and small-step semantics. In this chapter, I provide definitions and theorems in a syntax that is more consistent with the keywords used in my formalisation. It is still semantically equivalent to the previous definitions, just changing some names or layouts. I also further provide a big-step semantics for WCBV, which is more convenient for reasoning about time and space costs.

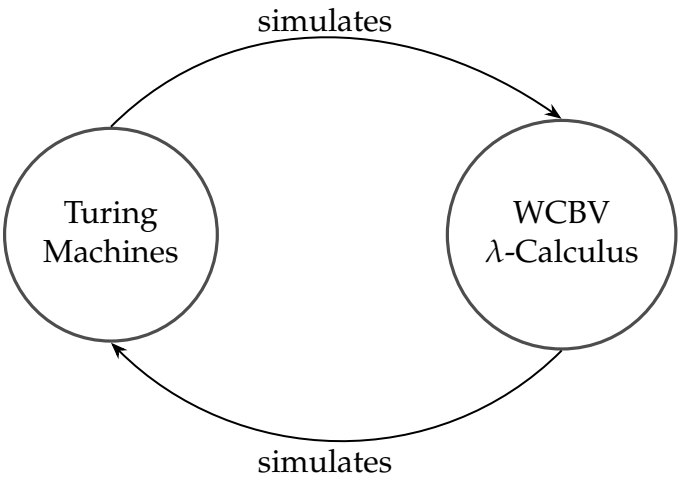


FIGURE 3.1: Simulation between Turing Machines and WCBV

3.1 Proof Strategy

In this section, I present the proof strategy and relevant motivation by Forster, Kunze, and Roth (2020) for showing that WCBV is reasonable.

Recall from the previous chapter (Section 2.2.2), in order to show that WCBV is reasonable, one must prove that the simulation between WCBV and a reasonable machine (here we choose Turing machines) has reasonable overheads in both time and space. This simulation includes two directions, one being Turing machines simulating WCBV, and another one being WCBV simulating Turing machines, as visualised in Figure 3.1. The proof for the latter direction can be obtained by adapting existing work from Dal Lago and Accattoli (2017). The first direction, from Turing machines to WCBV, is the theoretically challenging part.

How would we establish such a simulation? The most obvious approach is using a substitution-based strategy, where everything is eagerly evaluated and substituted in just as how it works in CBPV. However, this leads to problems. It is well-known that λ-calculus has the *size explosion problem*, where linear growth in time can lead to exponential growth in terms of space (Slot and van Emde Boas, 1984). That is, with $\mathcal{O}(n)$ β-reduction steps, the largest intermediate term can be of size $\mathcal{O}(2^n)$. Since Turing machines require at least one unit of time to consume one unit of space, its time cost is going to be at least as much as its space cost. Hence, if one adopts a substitution-based strategy alone, the overhead in time will be exponential, which exceeds the polynomial bound. To solve the size explosion problem, a shared memory

structure is required to store the values. Thus the heap-based strategy is incorporated into the simulation. By adding an extra heap structure and store the intermediate results there, we can solve the problem of size explosion.

The heap-based strategy, however, has a *pointer explosion problem* (Slot and van Emde Boas, 1984), which makes the simulation exceed the constant space overhead. Luckily, the size explosion problem and the pointer explosion problem don't overlap (Forster, Kunze, and Roth, 2020). To be more specific, when the subsitution-based strategy exhibits size explosion problem and is insufficient for time overhead, however, the space bound is guaranteed to be safe for executing the heap-based strategy without pointer explosion. This means that by starting with subsitution-based strategy, and switching to heap-based strategy whenever size explosion happens, we can avoid both size explosion and pointer explosion problems.

These two strategies are achieved by encoding them into two abstract machines, the substitution machine (Section 3.4.1) and the heap machine (Section 3.4.2). The direction of Turing machines simulating WCBV is then proved by going through these two machines (Section 3.5).

3.2 Weak Call-By-Value λ-Calculus

In this section, I formally define the weak call-by-value λ-calculus (WCBV), including its syntax and semantics.

The call-by-value evaluation strategy is widely used in the λ-calculus and functional programming languages. Under the call-by-value strategy, function arguments are evaluated before being passed to the function call as an input. The weak version of call-by-value does not allow reductions under λ-abstractions. Also note that in this thesis, we only work with closed λ-terms.

3.2.1 Syntax

I define the syntax for WCBV formally as follows:

Definition 15 (Syntax for WCBV)**.**

$$\lambda\text{-expression } s, t := \mathsf{var}\, n \mid \lambda.\, s \mid \mathsf{app}\, s\, t$$

In the above definition,

Variable $(\mathsf{var}\, n)$: represents a variable whose de Bruijn index is n.

Abstraction $(\lambda.\, s)$: represents a λ-abstraction, where s is another λ-expression representing the abstraction body.

Application $(\mathsf{app}\, s\, t)$: represents a λ-application, where s represents the function body and t represents the function argument.

Note that in my final version of the formalisation, I replaced my definition of the WCBV language with the existing definition for λ-calculus from HOL4's library. Thus the constructor names that can be found in my formalisation will instead be dV for variables, dAPP for applications and dABS for λ-abstractions. This adaption is done for the sake of consistency in the HOL library, but that syntax is more difficult to read. Thus in this thesis, I will use the original syntax for clarity. Similar adaptions are also done for other definitions and theorems in my formalisation.

3.2.2 Semantics

Recall in the previous chapter, I already provided a small-step semantics (Definition 11) for the call-by-value strategy. I hope that it gave a clear intuition about how the evaluation works. In this section, I introduce a big-step operational semantics for WCBV, which produces the same final results the small-step semantics for converging λ-terms, but easier for inference rule applications for the proof. After that, I introduce two big-step semantics of WCBV with embedded time and space costs respectively.

As introduced in the previous chapter, we need to have substitution (Definition 5) in order to define β-reductions. In particular, we need to define a closed substitution function since we are dealing with only closed terms in this thesis. In other words, v must be closed in this substitution. The following function s_v^i defines a substitution operation that I formalised, in which all the free variables with de Bruijn index of i in s are being replaced by v. This substitution passes down recursively, following the syntactical structure of s:

Definition 16 (Substitution Function for WCBV)**.** *(Forster, Kunze, and Roth, 2020, Def. 2.1)*

$$\begin{array}{lcll} (\mathsf{var}\, x)_v^i & = & v & (\mathit{if}\ x = i) \\ (\mathsf{var}\, x)_v^i & = & \mathsf{var}\, x & (\mathit{if}\ x \neq i) \\ (\lambda.\, s)_v^i & = & \lambda.\, s_v^{i+1} & \\ (\mathsf{app}\, s\, t)_v^i & = & \mathsf{app}\, s_v^i\, t_v^i & \end{array}$$

Note that for the case $(\lambda.\, s)^i_v = \lambda.\, s_v^{i+1}$, the targeted variable index i needs to be increased by one when passed down to the underlying abstraction body s, because this is entering an extra layer of abstraction. Recall that de Bruijn indices are related to the position of the variable in the abstraction.

The big-step operational semantics of WCBV is then defined through the judgement $s \Downarrow s'$ which states that a λ-term s reduces to s'. The rules defining this judgement are presented in Figure 3.2:

$$\frac{}{\lambda.\, s \Downarrow \lambda.\, s} \qquad \frac{s \Downarrow \lambda.\, s' \qquad t \Downarrow \lambda.\, t' \qquad s'^0_{(\lambda.\, t')} \Downarrow u}{\mathsf{app}\, s\, t \Downarrow u}$$

FIGURE 3.2: Big-Step Semantics of WCBV

To make it easier to reason about the time and space cost of WCBV, I formalised two additional versions of big-step semantics for WCBV that carry extra arguments representing time and space cost respectively. Note that in Forster, Kunze, and Roth (2020, Def. 2.2), they show the definition of the small-step semantics instead in the paper. It is equivalent to the big-step semantics and furthermore, the big-step semantics is what ends up being used in the proofs.

For the time cost version, I decorate the big-step semantics judgement with the time cost as a subscript: $s \Downarrow_i s'$. This judgement holds when the computation s reduces to s' with time cost i, where time cost is equivalent to the number of reduction steps for reducing s to s'. Using this syntax, I define the big-step semantic rules of WCBV with time cost in Figure 3.3. Note that in the base case, the time cost for going from $\lambda.\, s$ to itself is 0. This is because we take the number of β-reduction steps as the time cost, and there is no β-reduction performed to go from $\lambda.\, s$ to itself.

$$\frac{}{\lambda.\, s \Downarrow_0 \lambda.\, s} \qquad \frac{s \Downarrow_{i_1} \lambda.\, s' \qquad t \Downarrow_{i_2} \lambda.\, t' \qquad s'^0_{(\lambda.\, t')} \Downarrow_{i_3} u}{\mathsf{app}\, s\, t \Downarrow_{i_1+i_2+i_3+1} u}$$

FIGURE 3.3: Big-Step Semantics of WCBV with Time Cost

For the space cost version, I decorate the big-step semantics judgement with the space cost as a superscript: $s \Downarrow^m s'$. This judgement holds when the computation s reduces to s' with space cost m. This means that the largest intermediate term size during this computation is bounded by m.

In order to quantify the space cost, I define a function $\|s\|$ which computes the size of the WCBV term s. For variables, I count their de Bruijn index x as part of the term size with their face value. Note that we assume the de Bruijn index indices to be represented in unary notation. All constructor counts as 1 unit of space respectively. The size function is defined as follows:

Definition 17 (Size Function for WCBV)**.**

$$\begin{aligned} \|\mathsf{var}\, x\| &= 1 + x \\ \|\lambda.\, s\| &= 1 + \|s\| \\ \|\mathsf{app}\, s\, t\| &= 1 + \|s\| + \|t\| \end{aligned}$$

Using the size function, I then define the big-step semantics rules of WCBV with space cost in Figure 3.4:

$$\frac{}{\lambda.\, s \Downarrow^{\|\lambda.\, s\|} \lambda.\, s} \qquad \frac{s \Downarrow^{m_1} \lambda.\, s' \qquad t \Downarrow^{m_2} \lambda.\, t' \qquad s'^{0}_{(\lambda.\, t')} \Downarrow^{m_3} u}{\mathsf{app}\, s\, t \Downarrow^{\max(m_1 + \|t\| + 1,\, \max(\|(\lambda.\, s')\| + 1 + m_2,\, m_3))} u}$$

FIGURE 3.4: Big-Step Semantics of WCBV with Space Cost

This space-aware big-step semantics keeps track of the biggest intermediate term size by taking the size of the biggest subterm in each transitition step. The interesting case here is the app transition. There are three different stages: (1). when evaluating s; (2). when evaluating t; (3). when substituting the result from (2) into the result from (1). Each stage has a different size due to subterms being at different evaluation stages, and my transition rule picks the biggest size out of the three.

3.3 Compiling Weak Call-By-Value Terms to Programs

As a first step in bridging the gap between WCBV and Turing machines, I define a flat data structure to represent *programs* that correspond to WCBV terms. A program P is of a list of *tokens*, which I name as Tok in the formalisation. They are defined as follows:

$$\text{Tokens t} \in \mathsf{Tok} \;:=\; \mathsf{varT}\, x \mid \mathsf{applyT} \mid \mathsf{lamT} \mid \mathsf{retT}$$

Similarly to WCBV terms, we need to have a size function for the program tokens too. I define the size of tokens and programs as follows:

Definition 18 (Size of Tokens and Programs)**.**

$$\begin{array}{lcll} |\mathsf{varT}\, x| & = & 1 + x & \\ |t| & = & 1 & (\forall x.t \neq \mathsf{varT}\, x) \\ \|P\| & = & 1 + \sum_{t_i \in P} |t_i| & \end{array}$$

I account for the de Bruijn index x of a variable when accounting for token size because larger indices require asymptotically more tape cells to store on Turing machines. Each other token occupies 1 unit of space, as we have a constant number of token constructors. The size of a program is simply the sum of the size of all the tokens in the list representing the program, plus 1 (which is the size of the empty program on a Turing machine).

Intuitively, in order to show a simulation between WCBV and the abstract machines, we need to prove that for the same input, both of them should always return the same output within the required overhead in time and space. But WCBV terms and machine-readable programs are two different data structures. Thus, we need a compiler.

Below is my implementation of the compiler. It takes in a WCBV term and returns a corresponding machine-readable program:

Definition 19 (Compilation Function)**.** *(Forster, Kunze, and Roth, 2020, Def. 2.3)*

$$\begin{array}{lcl} \gamma(\mathsf{var}\, x) & = & \mathsf{varT}\, x \\ \gamma(\mathsf{app}\, s\, t) & = & \gamma(s) +\!\!+ \gamma(t) +\!\!+ [\mathsf{applyT}] \\ \gamma(\lambda.\, s) & = & \mathsf{lamT} :: \gamma(s) +\!\!+ [\mathsf{retT}] \end{array}$$

The compiler turns WCBV terms into programs by converting each subterm into tokens recursively and then concatenating them into a list. The compiler converts variables in place, just map the same index x to the varT token. The compilation for the other two constructors is slightly more complicated, as they carry λ-term(s) as arguments. We need to add delimiters to separate the terms that represents the arguments and the terms that represents the rest of the program. The compilation of app puts the compiled function body at the front, followed by the complied function argument, and ends with the applyT token. The applyT token is used to notate the end of this application. The ordering of the compilation of the subterms $\gamma(s)$ and $\gamma(t)$ is related to their evaluation order, which will be discussed in more detail later

in the abstract machines section. Essentially, the idea is that the abstract machines read the input programs in the left-to-right order, thus we need to put the parts that need to be evaluated first at the front. As for the compilation of λ-abstractions, it just wraps the abstraction body s using the lamT and retT token pair. This token pair enables the abstract machines to distinguish the compiled argument $\gamma(s)$ from the rest of the program. We wrap the whole body rather than just using one delimiter like in the application case because in this case, we do not want the abstraction body to be evaluated first.

The following lemma related to the compiler is useful for the space cost analysis, showing that the size of a compiled program is linearly bounded by the size of its corresponding WCBV term.

Lemma 20 (Program Size Bounds)**.** *(Forster, Kunze, and Roth, 2020, Lem. 2.1)* $1 \leq \|s\| \leq \|\gamma(s)\| \leq 2 * \|s\| - 1$

Proof. By structural induction on s. □

We also need to define the substitution function on programs. In this thesis, I will refer to it as program substitution. Program subsitution works similarly to the substitution in WCBV, except that it is now dealing with a list structure rather than a tree structure:

Definition 21 (Program Substitution)**.** *(Forster, Kunze, and Roth, 2020, Def. 2.6)*

$$
\begin{array}{lcll}
(\mathsf{varT}\ x :: P)^i_Q & = & Q :: P^i_Q & (\textit{if } x = i) \\
(\mathsf{varT}\ x :: P)^i_Q & = & \mathsf{varT}\ x :: P^i_Q & (\textit{if } x \neq i) \\
(\mathsf{lamT}\ :: P)^i_Q & = & \mathsf{lamT}\ :: P^{i+1}_Q & \\
(\mathsf{applyT}\ :: P)^i_Q & = & \mathsf{applyT}\ :: P^i_Q & \\
(\mathsf{retT}\ :: P)^0_Q & = & [\mathsf{retT}\,] & \\
(\mathsf{retT}\ :: P)^{i+1}_Q & = & \mathsf{retT}\ :: P^i_Q & \\
[]^i_Q & = & [] &
\end{array}
$$

One last thing we need to have for the cost analysis is to define the correspondence relation formally. I use $P \gg s$ to notate the correspondence relation between a program P and a term s. This relation only holds if P is the direct counterpart program representing s. I specifically restrict the relation to abstractions, which is s in the form of $(\lambda.\, s')$. This restriction is essential for the cost analysis. It is formally defined as:

Definition 22 (Program-Term Correspondence)**.** *(Forster, Kunze, and Roth, 2020, Def. 2.4) For any WCBV term s and its corresponding program P, we write the correspondence as* $P \gg s$*. This correspondence is defined by the rule* $\gamma t \gg (\lambda.\, t)$*.*

	Task Stack	Value Stack	Assumption
	$(\mathsf{lamT} :: P) :: T$	V	$\varphi P = (M, Q)$
$\triangleright$	$Q ::_{tc} T$	$M :: V$	
	$(\mathsf{applyT} :: P) :: T$	$Q :: R :: V$	
$\triangleright$	$R^0_{\mathsf{lamT} :: Q +\!\!+ [\mathsf{retT}]} ::(P ::_{tc} T)$	V	

FIGURE 3.5: Transition Rules for the WCBV Substitution Machine

3.4 Abstract Machines

In this section, I introduce and formalise the abstract machines that are used as the intermediate models between the Turing machines and WCBV. The abstract machines include the substitution machine, and the heap machine. Let's start with the substitution machine.

3.4.1 Substitution Machine

The substitution machine does exactly what its name suggests - it substitutes function arguments into the function body eagerly under the WCBV evaluation strategy. The reduction rules of the substitution machine are shown in Figure 3.5 (Forster, Kunze, and Roth, 2020, Fig. 1).

The program to be evaluated will be put on the task stack, and the value stack is used to store intermediate results. The task stack is a list of programs, and the value stack is a list of values, where a value is defined as a program that starts with a lamT token and ends with a retT token. The task stack is read from left to right, and the top of the stack is on the left side of the list. The value stack is also read from left to right. Note that in this thesis, I use a special list operator ($::_{tc}$, Definition 23) as well as the default list appending operator ($::$). The special operator $::_{tc}$ performs an empty-check on the element being appended, and skips the appending action if it is empty. This avoids appending empty list onto the stack, which ensures that the size of the machine states is linear to the size of the corresponding WCBV terms.

Definition 23 (Appending Function)**.** *(Forster, Kunze, and Roth, 2020, Fig. 1)*

$$\begin{aligned} [\,] ::_{tc} C &= C \\ c ::_{tc} C &= c :: C \quad (c \neq [\,]) \end{aligned}$$

The machine transits from states to states, where each state is a tuple containing a task stack and a value stack, written as (T, V). The one-step transition from one state (T, V) to state (T', V') is annotated as $(T, V) \triangleright (T', V')$. Multiple transitions are written as $(T, V) \triangleright_k^\sigma (T', V')$. We obtain a new state (T', V') after applying the transition rules k times on the current state (T, V), with the size of the biggest intermediate state being σ. We elide σ or k when irrelevant. We write $\triangleright_*$ to represent 0 or more transition steps.

The machine starts with an initial configuration where the input program P is on the task stack and the value stack is empty: $([P], [])$. The machine stops when there are no more tasks to compute, i.e., when the task stack is empty: $([], [V])$. The final result will be on top of the value stack.

There should not be a case where a naked variable is present in the task stack, thus the reduction rules only include two cases: one for abstraction, and one for application. These cases are sufficient to capture the behavior of the WCBV evaluation strategy.

In the abstraction case, the machine reads the lamT token from the task stack, and it uses the extraction function φ to find the body of the abstraction. The extraction function ($\varphi P = (M, Q)$) takes in a program P and returns a pair of programs (M, Q), where M is the body of the abstraction, and Q is the rest of the program after the abstraction. The machine then pushes Q back onto the task stack and pushes the whole abstraction body (wrapped with lamT and retT tokens) onto the value stack. Note that the extraction function is used by both the substitution machine and the heap machine. The extraction function φ is defined as follows:

Definition 24 (Extraction Function)**.** *(Forster, Kunze, and Roth, 2020, Def. 2.5)* $\varphi P = \varphi^0_{[]} P$ *where:*

$$
\begin{aligned}
\varphi^0_M(\mathsf{retT} :: Q) &= (M, Q) \\
\varphi^{k+1}_M(\mathsf{retT} :: Q) &= \varphi^k_{M +\!\!+ [\mathsf{retT}]} Q \\
\varphi^k_M(\mathsf{lamT} :: Q) &= \varphi^{k+1}_{M +\!\!+ [\mathsf{lamT}]} Q \\
\varphi^k_M(t :: Q) &= \varphi^k_{M +\!\!+ [t]} Q \quad \textit{if } t = \mathsf{var}\ n \textit{ or } \mathsf{app} \\
\varphi^k_M[] &= \textit{undefined}
\end{aligned}
$$

In the application case, the machine reads the applyT token from the task stack, and it pops the top two values from the value stack. The first value is

the function argument Q, and the second value is the function body wrapped with lamT and retT tokens. Note that these two things are on the value stack before the machine reaches the applyT token because of the order of compilation we define in Definition 19. It means that the machine will evaluate the application argument and application body first and push them onto the value stack before "knowing" it is part of an application. The machine then performs a substitution of Q into the body M using the substitution function we defined earlier, and pushes the result back onto the task stack for further evaluation.

I then formalise the proofs stating the time and space overhead of the simulation between the substitution machine and WCBV in Lemma 25 and Lemma 26 respectively.

Lemma 25 (Substitution Machine Time Simulation)**.** *(Forster, Kunze, and Roth, 2020, Lem. 3.1) If* $m \Downarrow_k n$*, then there exists* k' *such that* $([P], []) \triangleright_{k'} ([], [P'])$ *where* $k' = 3 * k + 1$ *and* $P = \gamma(m)$ *and* $P' \gg n$*.*

Proof. The proof proceeds by induction on the structure of the big-step semantics of WCBV. □

Lemma 26 (Substitution Machine Space Simulation)**.** *(Forster, Kunze, and Roth, 2020, Lem. 3.2) If* $m \Downarrow^s n$ *then there exists* σ *such that* $([P], []) \triangleright_*^\sigma ([], [P'])$ *, where* $s \leq \sigma \leq 2 * s$ *and* $P = \gamma(m)$ *and* $P' \gg n$*.*

Proof. Similar to the time simulation proof, we induct on the structure of the big-step semantics of WCBV and show that this theorem is true for all the transition rules. □

3.4.2 Heap Machine

The heap machine works by maintaining a heap structure that allows for efficient storage and retrieval of values. It extends the capabilities of the substitution machine by incorporating a heap for managing closures and their environments. Instead of substituting terms directly, the heap machine allocates closures on the heap and uses references to these closures in the program.

Below are the formal definitions of the constructs that are used in the heap machine definition.

Definition 27 (Closure)**.** *A closure* $\langle P, b \rangle$ *is a pair consisting of a program* P *and pointer* b*. The pointer* b *represents the environment under which all the free variables in the program* P *can be binded to the values in the heap.*

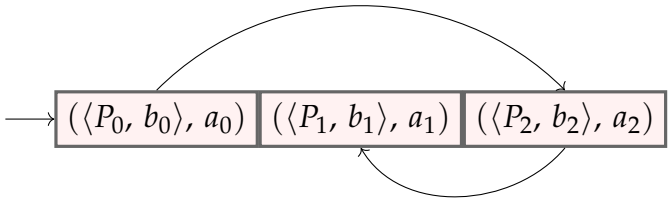


FIGURE 3.6: Heap and Closure

Definition 28 (Heap). *A heap is defined as a list of heap cells (heap entries) where each cell* (C, a) *consists of a closure* C *and an additional pointer* a*. The pointer* a *points to the previous cell in the heap, providing a linked list representation of the heap.*

I also draw the heap structure out in Figure 3.6 for visual demonstration. In this figure, I show a heap with three heap entries. Each heap entry consists of a closure $\langle P', b_n \rangle$ and a pointer a_n. It is in a linked list structure, thus the order of elements in this heap goes from cell 0 $(\langle P_0, b_0 \rangle, a_0)$ to cell 2 $(\langle P_2, b_2 \rangle, a_2)$ and finally to cell 1 $(\langle P_1, b_1 \rangle, a_1)$, rather than from cell 0 to cell 1 to cell 2.

I further define functions *get* and *extended* to extract information from the heap. I also define the functions *put* and *lookup* to help manipulate the heap.

Definition 29 (Heap Indexing Function). *get H a = H[a]*

Definition 30 (Extension Relation). *H' is H extended if and only if, for any heapentry m in H:*

$$(\forall \alpha \in \mathbb{N}.\, (get\ H\, \alpha\ =\ m) \rightarrow (get\ H'\, \alpha\ =\ m)$$

Definition 31 (Append Function). *Given a heap H, and a new cell e to add to the heap.*

$$put\ H\, e\ =\ (H +\!\!+ [e], len(H))$$

where $+\!\!+$ is list concatenation, and len is the standard length function for lists.

The append function returns a tuple containing the updated heap, and the updated top pointer represented by the length of the original heap.

Definition 32 (Heap Lookup Function). *Given a heap H, two numbers a and x, and $H[a] = (C, a')$.*

$$\begin{aligned} lookup\ H\, a\, 0\ &=\ C \\ lookup\ H\, a\, x\ &=\ lookup\ H\ a'\ (x-1) \qquad (if\ x\ \neq\ 0) \end{aligned}$$

	Task Stack	Value Stack	Heap	Assumption
	$\langle \mathsf{varT}\ x :: P, a\rangle :: T$	V	H	$\mathsf{lookup}\ H\ a\ x = g$
$\triangleright$	$\langle P, a\rangle :: T$	$g :: V$	H	
	$\langle \mathsf{lamT}\ :: P, a\rangle :: T$	V	H	$\varphi P = (M, Q)$
$\triangleright$	$\langle Q, a\rangle :: T$	$\langle \mathsf{lamT}\ :: M +\!\!+ [\mathsf{retT}\,], a\rangle :: V$	H	
	$\langle \mathsf{applyT}\ :: P, a\rangle :: T$	$Q :: \langle \mathsf{lamT}\ :: M +\!\!+ [\mathsf{retT}\,], b\rangle :: V$	H	$\mathsf{put}\ H(Q, b) = (H', c)$
$\triangleright$	$\langle M, c\rangle :: \langle P, a\rangle :: T$	V	H'	
	$\langle [\,], a\rangle :: T$	V	H	
$\triangleright$	T	V	H	

FIGURE 3.7: Transition Rules for the WCBV Heap Machine

The lookup function goes through the linked heap structure via pointers. Starting with pointer a on the heap, we can get the heap entry C at the corresponding location. The heap entry that is required is at index x. If index is not 0 yet, we need to keep going down the heap using the next pointer a' while counting the heap index down by 1.

With all the above definitions and helper functions, and the extraction function from earlier (Definition 24), I then define and formalise the heap machine as follows in Figure 3.7 (Forster, Kunze, and Roth, 2020, Fig. 2).

The heap machine has a few more reduction rules than the substitution machine, as it needs to handle heap operations. The task stack and value stack are now lists of closures instead of programs and values respectively. The heap is an additional component in the machine state. The initial configuration of the heap machine is $(\langle P, 0\rangle, [], [])$, where the input program P is wrapped in a closure with pointer 0 (indicating an empty environment), the value stack is empty, and the heap is also empty. The machine stops when there are no more tasks to compute, i.e., when the task stack is empty: $([], V, H)$. The final result will be on top of the value stack.

In the heap machine, the final result is stored in the form of heap-dependent programs. But WCBV is heap-independent. The existing compilation function (Definition 19) only converts heap-independent WCBV terms to heap-independent programs. In order to define the simulation between the heap machine and WCBV, we need to further define a correspondence relation between heap-dependent WCBV terms and heap-independent WCBV terms. This can be achieved by defining a function that maps a heap-dependent WCBV term to a corresponding heap-independent WCBV term by mapping

$$\frac{x < k}{(H,\, a|\, (\mathsf{var}\, x)) \rightsquigarrow_k (\mathsf{var}\, x)} \qquad \frac{(H,\, a|\, m) \rightsquigarrow_{k+1} m'}{(H,\, a|\, (\lambda.\, m)) \rightsquigarrow_k (\lambda.\, m)'}$$

$$\frac{(H,\, a|\, m) \rightsquigarrow_k m' \qquad (H,\, a|\, v) \rightsquigarrow_k v'}{(H,\, a|\, (\mathsf{app}\, m\, v)) \rightsquigarrow_k (\mathsf{app}\, m'\,)v'}$$

$$\frac{x \geq k \qquad \mathsf{lookup}\, H\, a\, (x-k) = \langle P,\, b\rangle \qquad P \gg s \qquad (H,\, b|\, s) \rightsquigarrow_0 s'}{(H,\, a|\, (\mathsf{var}\, x)) \rightsquigarrow_k s'}$$

FIGURE 3.8: Unfolding Rules for WCBV

the variables in the term to their actual values in the heap. I define such a function, which I name as *unfolding*, as follows (Forster, Kunze, and Roth, 2020, Def. 2.7). The unfolding function is defined as a relation $(H,\, a|\, m) \rightsquigarrow_k m'$, which states that under heap H and pointer a, the WCBV term m can be unfolded to m' with de Bruijn index offset k. The offset k is used to account for the change in variable indices when entering an abstraction. The unfolding relation is defined inductively by the rules presented in Figure 3.8. Note that the variable rule is exactly where we unfold a bound variable to itself.

With the above unfolding function, I then am able to define a heap-aware correspondence for WCBV terms as follows:

Definition 33 (Closure-Term Correspondence with Heap)**.** *(Forster, Kunze, and Roth, 2020, Definition. 2.8) For any closure $\langle P,\, a\rangle$ with heap H and WCBV term n, $\langle P,\, a\rangle$ is the corresponding closure for n (written as $\langle P,\, a\rangle \gg_H n$) if and only if there exists a WCBV term m such that $(H,\, a|\, m) \rightsquigarrow_0 n$ and $P \gg m$.*

I then formalise the simulation between the heap machine and WCBV in terms of both time and space respectively in Lemma 34 and Lemma 35 as follows:

Lemma 34 (Heap Machine Time Simulation)**.** *(Forster, Kunze, and Roth, 2020, Lem. 3.3) If $m \Downarrow_k n$ then there exists k' such that $([P], [\,], [\,]) \triangleright_{k'} ([\,], [P'], H)$, where $k' = 4 * k + 2$ and $P = \langle \gamma(m),\, 0\rangle$ and $P' \gg_H n$.*

Proof. The proof proceeds by induction on the big-step semantics of WCBV. □

The space cost of the heap machine is unrelated to the space cost of the WCBV term: the size of the k-th state is bounded by the size of the original WCBV term, and the number of transition steps:

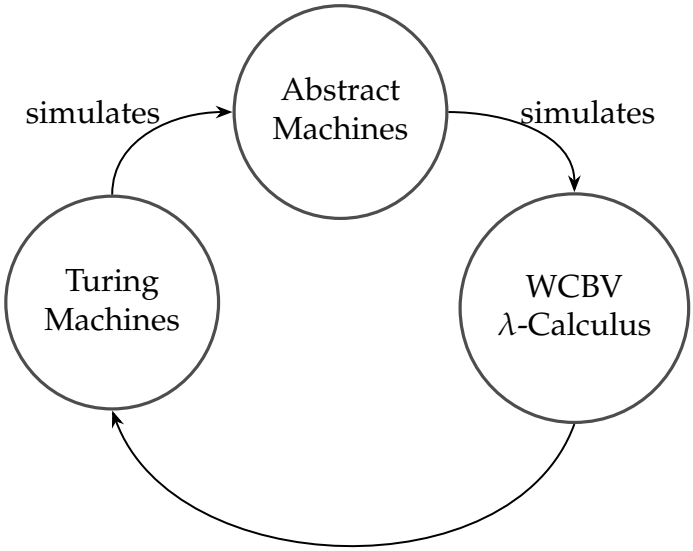


FIGURE 3.9: Simulation between Turing Machines and WCBV with Intermediate Models

Lemma 35 (Heap Machine Space Simulation)**.** *(Forster, Kunze, and Roth, 2020, Lem. 3.4) If* $(P, [\,], [\,]) \triangleright_k (T, N, H)$ *where* $P = \langle \gamma(m), 0 \rangle$ *then* $\|T +\!\!+ N +\!\!+ H\| \leq (k+1) * (3 * k + 4 * \|m\|)$.

Proof. The proof proceeds by showing each stack's length and that all elements in each stack are bounded by the size of the original term m and the number of transitions taken k. □

3.5 Weak Call-By-Value is Reasonable

In order to show that WCBV is reasonable, we must show that WCBV and Turing machines can simulate each other within polynomial time overhead and constant space overhead. This involves proofs for both directions.

The detailed full proof can be found in the original work by Forster, Kunze, and Roth (2020, Section 4-5). In this section, I will give a high-level explanation of their strategy, especially the parts that are related to my formalisation, while omitting in-depth proof details.

The figure Figure 3.9 illustrates the simulation and the related intermediate models or proofs used. For the direction from Turing machines to WCBV, we go through the abstract machines. In the previous sections, we have obtained the simulation from abstract machines to WCBV. To be more specific, we proved that the substition machine can simulate WCBV (Lemma 53,

Lemma 54) in desired time and space bounds. We also proved that the simulation from the heap machine to WCBV (Lemma 56, Lemma 57) is within the time and space bounds. Thus what is left to do is show that the Turing machines can simulate the abstract machines. This can be done by first implementing Turing machines $M_1, M_2, M_n, \ldots$ which simulate each individual transition step of the abstract machines. Then, by looping these small Turing machines, we can obtain bigger Turing machines M_{sub} and M_{heap} which simulate the entire computation of the substitution machine and the heap machine respectively. By interleaving these two Turing machines M_{sub} and M_{heap}, we can ultimately simulate WCBV. The interleaving strategy by default always starts with M_{sub}, until it encounters exploding terms (size explosion). It then switches to use M_{heap} for the computation. In this way, size explosion is avoided, and as explained earlier in Section 3.1, pointer explosion will not happen in this case.

As for the other direction, which is from WCBV to Turing machines, it can be done by adapting the prior work from Dal Lago and Accattoli (2017).

Together, these results provide a complete proof that WCBV is reasonable for both time and space, as elaborated before in Figure 3.9.

Chapter 4

Call-By-Push-Value is Reasonable for Time and Space

This chapter demonstrates the main contribution of my thesis, proving that CBPV is reasonable for both time and space. In this chapter, I first give an overview of the proof strategy and the formalisation layout in Section 4.1. Following the ideas explained in the proof strategy, I then go through the detailed implementation and proofs in the proceeding sections. In Section 4.2, I introduce my formalisation of CBPV in HOL4 and explain my design choices. I then give relevant formalised functions and theorems necessary for compiling the CBPV terms into machine readable programs in Section 4.3. Following that, I develop and formalise the abstract machines in Section 4.4, including the substitution machine and the heap machine. In the same section, I formally prove that the simulation between the abstract machines and CBPV is reasonable. Finally, in Section 4.5, I put the formalisation results together and use them to prove on pen and paper that CBPV is reasonable for both time and space. This is done by showing that the simulation between CBPV and Turing machines in both directions has polynomial time overhead and constant space overhead.

4.1 Overview

The cost model I choose is a natural measure for both time and space. Recall from Section 2.4, the natural measure for time is the number of β-reductions, and the natural measure for space is the size of the biggest intermediate term. Next step is to show that this measure gives CBPV the ability to define standard complexity classes like P, NP, PSPACE, or EXP. Recall that in the background chapter Section 2.2.2, we introduced a standard criterion for computational complexity analysis: a *reasonable* model of computation.

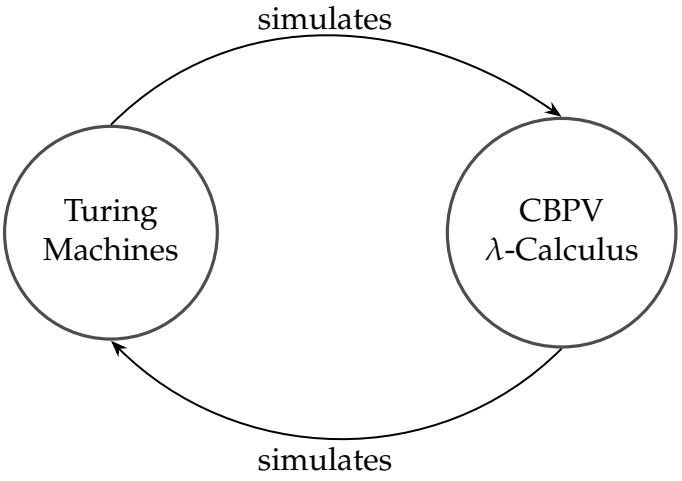


FIGURE 4.1: Simulation between Turing Machines and CBPV

More specifically, Turing machines are reasonable by definition, and reasonable machines can simulate each other with polynomial time overhead and constant factor space overhead. Thus, the goal becomes to show that with my measures, CBPV and Turing machines can simulate each other with polynomial overhead in time, and constant factor overhead in space. A high-level overview of this simulation is shown below in Figure 4.1, a more detailed diagram of this simulation is given in the later section (Section 4.5) as Figure 4.8. This proof is an immediate corollary of the following two theorems:

Theorem 36 (Turing machines simulating CBPV)**.** *Let $T, S \in \Omega(n)$. For a CBPV term s, if s reduces to a normal form t in time n and space m, then one can construct a Turing machine P_s simulating s that halts with output P_t simulating t, in time $\mathcal{O}(poly(T(n)))$ and space $\mathcal{O}(S(m))$.*

Theorem 37 (CBPV simulating Turing machines)**.** *Let $T, S \in \Omega(n)$. For a Turing machine P_s that halts with output P_t in time n and space m, one can construct a CBPV term s simulating P_s which can be reduced to a normal form t simulating P_t in time $\mathcal{O}(poly(T(n)))$ and space $\mathcal{O}(S(n))$.*

That is, we must model the cost of CBPV using Turing machines and prove that the resulting simulation is reasonable (Theorem 36). Moreover, we must show that CBPV provides sufficient expressivity to reasonably simulate any Turing machine (Theorem 37). With these two theorems, we can then have that CBPV is reasonable for both time and space (Theorem 38).

Theorem 38. *CBPV is reasonable for time and space, i.e.*

1. *Turing machines can simulate CBPV with polynomial time overhead and constant factor space overhead (Theorem 36).*

2. *CBPV can reasonably simulate Turing machines with polynomial time overhead and constant factor space overhead (Theorem 37).*

4.1.1 Proof Strategy

In this section, I give an overview of the proof strategy for showing that CBPV is reasonable. This includes the simulation from Turing machines to CBPV, and the simulation from CBPV to Turing machines. I later apply this proof strategy to obtain the main result in Section 4.5.

Turing machines simulating CBPV

Inspired by the strategy used for proving that WCBV is reasonable (Forster, Kunze, and Roth, 2020), I verify that Turing machines can simulate CBPV within the required bounds on overhead with the help of two intermediate abstract machines. Namely, the *substitution machine* (Section 4.4.1) and the *heap machine* (Section 4.4.2).

As discussed in the previous chapter (Section 3.1), both of these abstract machines are needed for simulating WCBV due to the size explosion and pointer explosion problems. The same problems apply to CBPV, thus both machines are used for this simulation too.

I formalise the simulation from each of the two abstract machines to CBPV and verify that it respects the desired bounds in terms of time and space overhead. The costs for CBPV turn out to be asymptotically similar to those for WCBV, enabling the adoption of an existing algorithm (Forster, Kunze, and Roth, 2020) to obtain a Turing machine simulation by interleaving the evaluation of the substitution and heap machines.

CBPV simulating Turing machines

For this direction, I use WCBV as an intermediate model. I first formalise the translation from WCBV to CBPV provided by Levy (1999). I then show that CBPV can simulate WCBV with reasonable time and space overheads. We know that WCBV can simulate Turing machines with reasonable time and space overheads (Forster, Kunze, and Roth, 2020, Theorem 5.1); as a corollary, we obtain that the simulation from CBPV to Turing machines is reasonable.

4.1.2 Formalisation

I formally verify in HOL4 that my time and space cost models for CBPV exhibit the desired overheads.My formalisation is structured into a collection of HOL4 theory files, each addressing the following aspects:

1. The syntax and semantics of CBPV, including extra big-step semantics that keeps track of the time and space cost during the evaluation. (Section 4.2)
2. The definition of machine-readable programs and a compilation function from CBPV terms to machine-readable programs. (Section 4.3)
3. The semantics of substitution machines and proofs for its simulation of CBPV terms with respect to time and space. (Section 4.4.1)
4. The semantics of heap machine and proofs for its simulation of CBPV with respect to time and space. Related helper definitions and functions, such as the heap structure and the unfolding function. (Section 4.4.2)
5. The proofs of CBPV simulating WCBV. (Section 4.5.2)

The above formalisation provides all the core proofs for CBPV being reasonable. These formalised proofs are then brought together to reach the final proof goal by using the interleaving strategy via pen-and-paper proofs, adapted from the literature and detailed in Section 4.5.

4.2 Call-By-Push-Value λ-Calculus

I have introduced CBPV in the background chapter (Section 2.1.2) on a more general level and provided detailed explanation and examples. In this section, I focus more on my formalisation of CBPV and provide insights on my design decisions. I choose my notation to closely follow the formalisation. This is both to ease the reading of this thesis by itself and to bridge to the formalisation as smoothly as possible.

For simplicity, I choose to work on a core fragment of untyped CBPV that is sufficient for demonstrating reasonability. I consider types an orthogonal concern: the well-typed CBPV terms are a strict subset of the untyped CBPV terms, so a cost model for the former immediately suggests a cost model for the latter. Note that the untyped variant of CBPV itself is used in λ-calculus research, particularly in the context of compilation. For instance,

it was linked to control flow graphs (Rizkallah, Garbuzov, and Zdancewic, 2018), and subsequently used to verify real-world compiler optimisations.

The set of core untyped CBPV terms is defined below in Definition 39 through two mutually recursively defined sets; a set of *values* V and a set of *computations* M. The mutually recursive definition of CBPV added technical difficulties in my formalisation as all the other relevant functions need to be defined in a mutually recursive manner as well. For instance, the substitution function for CBPV, and the compilation function for compiling CBPV terms into programs are both defined mutually recursively, which adds formalisation challenges.

Definition 39 (Syntax for CBPV)**.**

Values	V	:=	var x |	thunk M		
Computations	M	:=	λ. M |	app M V |	force V |	ret V |
			seq M M |	pseq M M M |	let V. in M	

As one might notice, the above definition has an extra pseq compared to the standard syntax of untyped CBPV without pairs defined in Definition 13. This is because without the expressive power of pairs, the space overhead for simulating Turing machines using CBPV in Section 4.5.2 exceeds the desired threshold. In order to fix this problem, I add an extra constructor pseq, an implementation of a pair type designed specifically for seq. pseq works similar to seq, except that it takes three computations instead of two. pseq then substitutes the return results from both of the two computations into the third computation.

In my formalisation for CBPV syntax, terms are defined as two separate but mutually recursive datatypes. Hence, all the definitions and proofs that use CBPV terms in the formalisation are written in the mutual recursive style. For instance, the compilation function is formalised in terms of two functions compileVal and compileComp instead of a single compile function. For simplicity, in this thesis, I omit this distinction when referring to other functions.

I then formalise the big-step semantics of closed CBPV provided by Levy (2001); that is, I only consider terms with no free variables at the top level. Similar to the change I made in the syntax, I also add pseq as a special case of the pair type into my semantics.

In order to define the semantics for closed CBPV, I need to first provide a closed substitution function for β-reductions. The following function m_v^i

substitutes all the variables with de Bruijn index i by v in m by recursively visiting all the inner terms of m:

Definition 40 (Substitution function)**.**

$$
\begin{array}{lcllcl}
(\lambda.\, m)^i_u & = & \lambda.\, (m^{i+1}_u) & (\mathsf{var}\ x)^i_u & = & u\ (\mathit{if}\ x = i) \\
(\mathsf{app}\ m\ v)^i_u & = & \mathsf{app}\ (m^i_u)\ (v^i_u) & (\mathsf{var}\ x)^i_u & = & \mathsf{var}\ x\ (\mathit{if}\ x \neq i) \\
(\mathsf{ret}\ v)^i_u & = & \mathsf{ret}\ (v^i_u) & (\mathsf{thunk}\ m)^i_u & = & \mathsf{thunk}\ (m^i_u) \\
(\mathsf{seq}\ m\ n)^i_u & = & \mathsf{seq}\ (m^i_u)\ (n^{i+1}_u) & (\mathsf{force}\ v)^i_u & = & \mathsf{force}\ (v^i_u) \\
(\mathsf{pseq}\ m_2\ m_1\ n)^i_u & = & \mathsf{pseq}\ (m_{2u}{}^i)\ (m_{1u}{}^i)\ (n^{i+2}_u) & (\mathsf{let}\ v.\ \mathsf{in}\ m)^i_u & = & \mathsf{let}\ (v^i_u).\ \mathsf{in}\ (m^{i+1}_u)
\end{array}
$$

Note the special cases such as $(\lambda.\, m)^i_u$, where we need to increment the targeted variable index i accordingly because we are entering extra layers of abstractions. A more special case is $(\mathsf{pseq}\ m_2\ m_1\ n)^i_u$, we increment i by two for this substitution because n needs to leave two free variable names for the two computations m_1 and m_2.

The big-step operational semantics of CBPV is then defined through the judgement $m \Downarrow n$ which states that a computation m reduces to n. The rules defining this judgement are presented in Figure 4.2:

$$
\frac{}{\lambda.\, m \Downarrow \lambda.\, m} \qquad \frac{}{\mathsf{ret}\ v \Downarrow \mathsf{ret}\ v} \qquad \frac{m \Downarrow m'}{\mathsf{force}\ (\mathsf{thunk}\ m) \Downarrow m'} \qquad \frac{m^0_v \Downarrow m'}{\mathsf{let}\ v.\ \mathsf{in}\ m \Downarrow m'}
$$

$$
\frac{m \Downarrow \lambda.\, n \qquad n^0_v \Downarrow n'}{\mathsf{app}\ m\ v \Downarrow n'} \qquad \frac{m \Downarrow \mathsf{ret}\ v \qquad n^0_v \Downarrow n'}{\mathsf{seq}\ m\ n \Downarrow n'}
$$

$$
\frac{m_1 \Downarrow \mathsf{ret}\ v_1 \qquad m_2 \Downarrow \mathsf{ret}\ v_2 \qquad (n^0_{v_1})^1_{v_2} \Downarrow n'}{\mathsf{pseq}\ m_2\ m_1\ n \Downarrow n'}
$$

FIGURE 4.2: Big-Step Semantics of CBPV

I then further provide two other versions of big-step semantics of CBPV with regard to time cost (Figure 4.3) and space cost (Figure 4.4). They share the same notation as the ones for WCBV (Section 3.2), hence I will elide repeated explanation.

Before defining rules for the big-step semantics with space cost, we need to have a size function:

$$\frac{}{\lambda.\, m \;\Downarrow_0\; \lambda.\, m} \qquad \frac{}{\mathsf{ret}\, v \;\Downarrow_0\; \mathsf{ret}\, v} \qquad \frac{m \;\Downarrow_k\; m'}{\mathsf{force}\,(\mathsf{thunk}\, m) \;\Downarrow_{k+2}\; m'}$$

$$\frac{m_v^0 \;\Downarrow_k\; m'}{\mathsf{let}\, v.\, \mathsf{in}\, m \;\Downarrow_{k+1}\; m'} \qquad \frac{m \;\Downarrow_{k_1}\; \lambda.\, n \qquad n_v^0 \;\Downarrow_{k_2}\; n'}{\mathsf{app}\, m\, v \;\Downarrow_{k_1+k_2+1}\; n'} \qquad \frac{m \;\Downarrow_{k_1}\; \mathsf{ret}\, v \qquad n_v^0 \;\Downarrow_{k_2}\; n'}{\mathsf{seq}\, m\, n \;\Downarrow_{k_1+k_2+1}\; n'}$$

$$\frac{m_1 \;\Downarrow_{k_1}\; \mathsf{ret}\, v_1 \qquad m_2 \;\Downarrow_{k_2}\; \mathsf{ret}\, v_2 \qquad (n_{v_1}^0)_{v_2}^1 \;\Downarrow_{k_3}\; n'}{\mathsf{pseq}\, m_2\, m_1\, n \;\Downarrow_{k_1+k_2+k_3+1}\; n'}$$

FIGURE 4.3: Big-Step Semantics of CBPV with Time Cost

Definition 41 (Size of CBPV Term)**.**

$$\begin{aligned}
\|\mathsf{var}\, x\| &= 1 + x \\
\|\mathsf{thunk}\, m\| &= 1 + \|m\| \\
\|\mathsf{force}\, v\| &= 1 + \|v\| \\
\|\mathsf{let}\, v.\, \mathsf{in}\, m\| &= 1 + \|v\| + \|m\| \\
\|\lambda.\, m\| &= 1 + \|m\| \\
\|\mathsf{app}\, m\, v\| &= 1 + \|m\| + \|v\| \\
\|\mathsf{ret}\, v\| &= 1 + \|v\| \\
\|\mathsf{seq}\, m\, n\| &= 1 + \|m\| + \|n\| \\
\|\mathsf{pseq}\, m_2\, m_1\, n\| &= 1 + \|m_2\| + \|m_1\| + \|n\|
\end{aligned}$$

We can then define the space-aware semantics:

$$\frac{}{\lambda.\, m \;\Downarrow^{\|m\|+1}\; \lambda.\, m} \qquad \frac{}{\mathsf{ret}\, v \;\Downarrow^{\|v\|+1}\; \mathsf{ret}\, v}$$

$$\frac{m \;\Downarrow^{s}\; m'}{\mathsf{force}\,(\mathsf{thunk}\, m) \;\Downarrow^{\max(s, \|m\|+2)}\; m'} \qquad \frac{m_v^0 \;\Downarrow^{s}\; m'}{\mathsf{let}\, v.\, \mathsf{in}\, m \;\Downarrow^{\max(s, \|v\|+\|m\|+1)}\; m'}$$

$$\frac{m \;\Downarrow^{s_1}\; \lambda.\, n \qquad n_v^0 \;\Downarrow^{s_2}\; n'}{\mathsf{app}\, m\, v \;\Downarrow^{\max(s_1+\|v\|+1, s_2)}\; n'} \qquad \frac{m \;\Downarrow^{s_1}\; \mathsf{ret}\, v \qquad n_v^0 \;\Downarrow^{s_2}\; n'}{\mathsf{seq}\, m\, n \;\Downarrow^{\max(s_1+\|n\|+1, s_2)}\; n'}$$

$$\frac{m_1 \;\Downarrow^{s_1}\; \mathsf{ret}\, v_1 \qquad m_2 \;\Downarrow^{s_2}\; \mathsf{ret}\, v_2 \qquad (n_{v_1}^0)_{v2}^1 \;\Downarrow^{s_3}\; n'}{\mathsf{pseq}\, m_2\, m_1\, n \;\Downarrow^{\max(s_1+\|m_2\|+\|n\|+1,\, \|v_1\|+s_2+\|n\|+1,\, s_3)}\; n'}$$

FIGURE 4.4: Big-Step Semantics of CBPV Terms with Space Cost

I will go through the pseq case as an example. There are three different stages in this reduction: (1). when evaluating m_1; (2). when evaluating m_2 (3). when substituting results v_1 and v_2 into n. Each stage incurs different costs, and only the maximum is taken as the cost for the whole reduction of pseq.

4.3 Compiling Call-By-Push-Value Terms to Programs

For the same reason as the case for WCBV in Section 3.3, I develop relevant definitions and functions for compiling CBPV terms to machine-readable programs. Note that there are two sets of program tokens and two compilation functions, which are used for the substitution machine and the heap machine respectively.

I define the programs to be the same flat list structure as in the previous chapter, with the difference of some extra CBPV specific tokens:

Definition 42 (Program Tokens for Substitution Machine)**.**

$$\begin{aligned} \mathsf{t} \in \mathsf{Tok} \quad := \quad & \mathsf{varT}\, x \mid \mathsf{thunkT} \mid \mathsf{endThunkT} \mid \mathsf{forceT} \mid \mathsf{applyT} \mid \\ & \mathsf{lamT} \mid \mathsf{endLamT} \mid \mathsf{retT} \mid \mathsf{endRetT} \mid \mathsf{seqT} \mid \mathsf{endSeqT} \mid \\ & \mathsf{pseqT} \mid \mathsf{endPseqT} \mid \mathsf{letinT} \mid \mathsf{endLetinT} \end{aligned}$$

Definition 43 (Size of Tokens)**.**

$$\begin{aligned} |\mathsf{varT}\, x| &= 1 + x \\ |t| &= 1 \qquad (\forall x.t \neq \mathsf{varT}\, x) \end{aligned}$$

Definition 44 (Size of Programs)**.**

$$\|P\| = 1 + \textstyle\sum_{t_i \in P} |t_i|$$

The program tokens for the heap machine (Definition 45) are mostly the same as the ones for the substitution machine, except that it removed endRetT . The size functions work similarly, we will not repeat them again.

Definition 45 (Program Tokens for Heap Machine)**.**

$$\begin{aligned} \mathsf{t} \in \mathsf{Tok} \quad := \quad & \mathsf{varT}\, x \mid \quad \mathsf{thunkT} \mid \quad \mathsf{endThunkT} \mid \quad \mathsf{forceT} \mid \quad \mathsf{applyT} \mid \\ & \mathsf{lamT} \mid \quad \mathsf{endLamT} \mid \quad \mathsf{retT} \mid \quad \mathsf{seqT} \mid \quad \mathsf{endSeqT} \mid \\ & \mathsf{pseqT} \mid \quad \mathsf{endPseqT} \mid \quad \mathsf{letinT} \mid \quad \mathsf{endLetinT} \end{aligned}$$

We can then define the compilers. Below are my implementations of two compilers. Each compiler takes in a CBPV term and returns a corresponding machine-readable program. The first compiler in Definition 46 is designed for the substitution machine. The second compiler in Definition 47 is designed for the heap machine and is modified based on the first compiler.

Definition 46 (Compilation Function for Substitution Machine)**.**

$$\begin{aligned} \gamma(\mathsf{var}\, x) &= \mathsf{varT}\, x \\ \gamma(\mathsf{thunk}\, m) &= \mathsf{thunkT} :: \gamma(m) +\!\!+ [\mathsf{endThunkT}] \\ \gamma(\mathsf{force}\, v) &= \gamma(v) +\!\!+ [\mathsf{forceT}] \\ \gamma(\mathsf{ret}\, v) &= \mathsf{retT} :: \gamma(v) +\!\!+ [\mathsf{endRetT}] \\ \gamma(\lambda.\, m) &= \mathsf{lamT} :: \gamma(m) +\!\!+ [\mathsf{endLamT}] \\ \gamma(\mathsf{app}\, m\, v) &= \gamma(m) +\!\!+ \gamma(v) +\!\!+ [\mathsf{applyT}] \\ \gamma(\mathsf{seq}\, m\, n) &= \gamma(m) +\!\!+ [\mathsf{seqT}] +\!\!+ \gamma(n) +\!\!+ [\mathsf{endSeqT}] \\ \gamma(\mathsf{pseq}\, m_1\, m_2\, n) &= \gamma(m_1) +\!\!+ \gamma(m_2) +\!\!+ [\mathsf{pseqT}] +\!\!+ \gamma(n) +\!\!+ [\mathsf{endPseqT}] \\ \gamma(\mathsf{let}\, v.\, \mathsf{in}\, m) &= \gamma(v) +\!\!+ [\mathsf{letinT}] +\!\!+ \gamma(m) +\!\!+ [\mathsf{endLetinT}] \end{aligned}$$

The compiler turns CBPV terms into programs by converting each subterm into tokens recursively and then merging them into a list. For simple terms like $(\mathsf{var}\, x)$, we convert them in place. For more complicated terms, we add delimiters when necessary to preserve the syntactic structure of the original CBPV term.

The development of this compiler is challenging as different constructors require their fields to be treated differently in order for the transition rules to work. For instance, the compilation of $(\lambda.\, m)$ adds an extra $\mathsf{endLamT}$ token to delimit the computation m, so it's easier for the abstract machines to distinguish $\gamma(m)$ from the rest of the programs.

Another example is $(\mathsf{seq}\, m\, n)$. In this case, we have two computations to take care of. Thus extra care needs to be taken. I designed the compiler in a way such that it moves $\gamma(m)$ to the front and delimits $\gamma(n)$ by using extra token pairs instead of putting both of the computations inside, like the case for

$(\lambda.\, m)$. This is done so that the machine can recognise when the computation of $\gamma(m)$ is done and then perform the substitution on $\gamma(n)$ using its result.

Note that having $\gamma(m)$ at the front instead of $\gamma(n)$ is necessary for $\gamma(m)$ to be evaluated automatically first. Another thing to pay atttention to is how $\gamma(n)$ is delimited by seqT and $\mathsf{endSeqT}$ instead of being by itself like $\gamma(m)$. This is because we do not wish to evaluate $\gamma(n)$ in the case of seqT. Similar to CBPV's semantics, in the corresponding machines for CBPV, *seqT* should not evaluate $\gamma(n)$ before substituting the value of $\gamma(m)$ into $\gamma(n)$ accordingly. Thus we use seqT and $\mathsf{endSeqT}$ to prevent $\gamma(n)$ from being evaluated automatically in the machine rules.

Other computation rules work similarly, I omit the explanation of the remaining rules.

The following function (Definition 47) for the heap machine differs from Definition 46 only in the compilation of $(\mathsf{ret}\, v)$. It is necessary for the heap machine to compile retT in this way instead of delimiting the body with a retT and $\mathsf{endRetT}$ pair like the substitution machine. In Section 4.4.2 I provide more explanation of this design.

Definition 47 (Compilation Function for Heap Machine)**.**

$$
\begin{array}{lcl}
\gamma'(\mathsf{var}\, x) & = & \mathsf{varT}\, x \\
\gamma'(\mathsf{thunk}\ m) & = & \mathsf{thunkT}\ ::\ \gamma'(m) +\!\!+ [\mathsf{endThunkT}] \\
\gamma'(\mathsf{force}\, v) & = & \gamma'(v) +\!\!+ [\mathsf{forceT}] \\
\gamma'(\mathsf{ret}\, v) & = & \gamma'(v) +\!\!+ [\mathsf{retT}] \\
\gamma'(\lambda.\, m) & = & \mathsf{lamT}\ ::\ \gamma'(m) +\!\!+ [\mathsf{endLamT}] \\
\gamma'(\mathsf{app}\, m\, v) & = & \gamma'(m) +\!\!+ \gamma'(v) +\!\!+ [\mathsf{applyT}] \\
\gamma'(\mathsf{seq}\, m\, n) & = & \gamma'(m) +\!\!+ [\mathsf{seqT}] +\!\!+ \gamma'(n) +\!\!+ [\mathsf{endSeqT}] \\
\gamma'(\mathsf{pseq}\, m_1\, m_2\, n) & = & \gamma'(m_1) +\!\!+ \gamma'(m_2) +\!\!+ [\mathsf{pseqT}] +\!\!+ \gamma'(n) +\!\!+ [\mathsf{endPseqT}] \\
\gamma'(\mathsf{let}\, v.\, \mathsf{in}\, m) & = & \gamma'(v) +\!\!+ [\mathsf{letinT}] +\!\!+ \gamma'(m) +\!\!+ [\mathsf{endLetinT}]
\end{array}
$$

The reason to have two compilations is convenience. On the one hand, having start-end token pairs is technically convenient when reasoning about the machines. On the other hand, having one less token gives us more slack when chasing inequalities in proofs of space bounds. I could modify the substitution machine to use γ', but there would be little benefit to this: the simulation strategy does not require the machines to use the same syntax.
The following lemma is useful for the space cost analysis, showing that the size of a compiled program is linearly bounded by the size of its corresponding CBPV term. (Note that γ' also has a similar lemma.)

Lemma 48 (Program Size Bounds). $1 \leq \|m\| \leq \|\gamma(m)\| \leq 2 * \|m\| - 1$

Proof. By structural induction on m. □

We also need to define program substitution functions (Definition 49,Definition 50) to use in the abstract machines. Note that the only difference of the two program substitution functions is the rule for endRetT , as the compilation for heap machine programs removed this construct completely.

Definition 49 (Program Substitution for Substitution Machine).

$$
\begin{array}{lcll}
(\mathsf{varT}\ x :: P)^i_Q & = & Q :: P^i_Q & (if\ x = i) \\
(\mathsf{varT}\ x :: P)^i_Q & = & \mathsf{varT}\ x :: P^i_Q & (if\ x \neq i) \\
(\mathsf{lamT}\ :: P)^i_Q & = & \mathsf{lamT}\ :: P^{i+1}_Q & \\
(\mathsf{endLamT}\ :: P)^0_Q & = & [\mathsf{endLamT}\,] & \\
(\mathsf{endLamT}\ :: P)^{i+1}_Q & = & \mathsf{endLamT}\ :: P^i_Q & \\
(\mathsf{seqT}\ :: P)^i_Q & = & \mathsf{seqT}\ :: P^{i+1}_Q & \\
(\mathsf{endSeqT}\ :: P)^0_Q & = & [\mathsf{endSeqT}\,] & \\
(\mathsf{endSeqT}\ :: P)^{i+1}_Q & = & \mathsf{endSeqT}\ :: P^i_Q & \\
(\mathsf{pseqT}\ :: P)^i_Q & = & \mathsf{pseqT}\ :: P^{i+2}_Q & \\
(\mathsf{endPseqT}\ :: P)^i_Q & = & [\mathsf{endPseqT}\,] & (if\ i \leq 1) \\
(\mathsf{endPseqT}\ :: P)^{i+2}_Q & = & \mathsf{endPseqT}\ :: P^i_Q & \\
(\mathsf{letinT}\ :: P)^i_Q & = & \mathsf{letinT}\ :: P^{i+1}_Q & \\
(\mathsf{endLetinT}\ :: P)^0_Q & = & [\mathsf{endLetinT}\,] & \\
(\mathsf{endLetinT}\ :: P)^{i+1}_Q & = & \mathsf{endLetinT}\ :: P^i_Q & \\
(\mathsf{applyT}\ :: P)^i_Q & = & \mathsf{applyT}\ :: P^i_Q & \\
(\mathsf{retT}\ :: P)^i_Q & = & \mathsf{retT}\ :: P^i_Q & \\
(\mathsf{endRetT}\ :: P)^i_Q & = & \mathsf{endRetT}\ :: P^i_Q & \\
(\mathsf{forceT}\ :: P)^i_Q & = & \mathsf{forceT}\ :: P^i_Q & \\
(\mathsf{thunkT}\ :: P)^i_Q & = & \mathsf{thunkT}\ :: P^i_Q & \\
[]^i_Q & = & [] &
\end{array}
$$

Definition 50 (Program Substitution for Heap Machine).

$$
\begin{array}{lcll}
(\mathsf{varT}\ x :: P)^{i}_{Q} & = & Q :: P^{i}_{Q} & (if\ x = i) \\
(\mathsf{varT}\ x :: P)^{i}_{Q} & = & \mathsf{varT}\ x :: P^{i}_{Q} & (if\ x \neq i) \\
(\mathsf{lamT}\ :: P)^{i}_{Q} & = & \mathsf{lamT}\ :: P^{i+1}_{Q} & \\
(\mathsf{endLamT}\ :: P)^{0}_{Q} & = & [\mathsf{endLamT}\,] & \\
(\mathsf{endLamT}\ :: P)^{i+1}_{Q} & = & \mathsf{endLamT}\ :: P^{i}_{Q} & \\
(\mathsf{seqT}\ :: P)^{i}_{Q} & = & \mathsf{seqT}\ :: P^{i+1}_{Q} & \\
(\mathsf{endSeqT}\ :: P)^{0}_{Q} & = & [\mathsf{endSeqT}\,] & \\
(\mathsf{endSeqT}\ :: P)^{i+1}_{Q} & = & \mathsf{endSeqT}\ :: P^{i}_{Q} & \\
(\mathsf{pseqT}\ :: P)^{i}_{Q} & = & \mathsf{pseqT}\ :: P^{i+2}_{Q} & \\
(\mathsf{endPseqT}\ :: P)^{i}_{Q} & = & [\mathsf{endPseqT}\,] & (if\ i \leq 1) \\
(\mathsf{endPseqT}\ :: P)^{i+2}_{Q} & = & \mathsf{endPseqT}\ :: P^{i}_{Q} & \\
(\mathsf{letinT}\ :: P)^{i}_{Q} & = & \mathsf{letinT}\ :: P^{i+1}_{Q} & \\
(\mathsf{endLetinT}\ :: P)^{0}_{Q} & = & [\mathsf{endLetinT}\,] & \\
(\mathsf{endLetinT}\ :: P)^{i+1}_{Q} & = & \mathsf{endLetinT}\ :: P^{i}_{Q} & \\
(\mathsf{applyT}\ :: P)^{i}_{Q} & = & \mathsf{applyT}\ :: P^{i}_{Q} & \\
(\mathsf{retT}\ :: P)^{i}_{Q} & = & \mathsf{retT}\ :: P^{i}_{Q} & \\
(\mathsf{forceT}\ :: P)^{i}_{Q} & = & \mathsf{forceT}\ :: P^{i}_{Q} & \\
(\mathsf{thunkT}\ :: P)^{i}_{Q} & = & \mathsf{thunkT}\ :: P^{i}_{Q} & \\
[]^{i}_{Q} & = & [] &
\end{array}
$$

Moreover, my cost analysis relies on the correspondence relation $P \gg m$ between a program P and a term m that holds if P is the direct counterpart program representing m. Formally:

Definition 51 (Program-Term Correspondence). *For any program P and CBPV term m, P is the corresponding program for m (written as $P \gg m$) if and only if $\gamma(m) = P$.*

You might notice that this correspondence definition is slightly different from the one defined in the last chapter (Definition 22). In this case, we no longer need to restrict it to be abstractions because our machine rules do not eagerly remove the abstraction constructor from the term before placing it on the value stack.

4.4 Abstract Machines

In this section, I first introduce an auxiliary extraction function that is used by both abstract machines. I then introduce the two intermediate abstract machines, the substitution machine (Section 4.4.1) and the heap machine (Section 4.4.2), and investigate their cost in relation to CBPV.

For each pair of delimiter tokens, I define a φ function to scan a program until the corresponding end delimiter. That includes: thunkT - endThunkT , lamT - endLamT , seqT - endSeqT , pseqT - endPseqT , letinT - endLetinT for both abstract machines, and an extra pair, retT - endRetT , for the substitution machine. This difference is due to the different compilation function for the substitution machine and the heap machine.

The extraction functions for all these pairs work in the exact same way, and they serve the same purpose. The idea is that $\varphi P = (M, Q)$ strips the argument body out of P and returns it as M, where Q is the rest of the program. We show the extraction function ${}_l\varphi$ for lamT - endLamT in the substitution machine below as an example. The intuition is similar to finding matching parentheses pairs. When ${}_l\varphi$ is applied to a wellformed P, we have ${}_l\varphi P = {}_l\varphi(M :: [\mathsf{endLamT}] +\!\!+ Q) = (M, Q)$, where M has balanced lamT - endLamT pairs.

Definition 52 (Extraction Function for lamT - endLamT). ${}_l\varphi P = {}_l\varphi^0_{[]} P$ *where:*

$$
\begin{array}{lcllcl}
{}_l\varphi^0_M(\mathsf{endLamT} :: Q) & = & (M, Q) & {}_l\varphi^k_M(t :: Q) & = & {}_l\varphi^k_{M +\!\!+ [t]} Q \\
{}_l\varphi^k_M(\mathsf{lamT} :: Q) & = & {}_l\varphi^{k+1}_{M +\!\!+ [\mathsf{lamT}]} Q & {}_l\varphi^k_M[] & = & \textit{undefined} \\
{}_l\varphi^{k+1}_M(\mathsf{endLamT} :: Q) & = & {}_l\varphi^k_{M +\!\!+ [\mathsf{endLamT}]} Q & & &
\end{array}
$$

The extraction functions for the rest of the delimiter pairs (${}_t\varphi$ for thunkT - endThunkT , ${}_s\varphi$ for seqT - endSeqT , ${}_p\varphi$ for pseqT - endPseqT , ${}_i\varphi$ for letinT - endLetinT and ${}_r\varphi$ for retT - endRetT) are defined in a similar manner, I will elide their definition here.

4.4.1 Substitution Machine

In this section, I develop a substitution machine that implements a substitution-based evaluation strategy for programs corresponding to CBPV terms. I use the helper function ($::_{tc}$, Definition 23) defined in the previous chapter for both my substitution machine and my heap machine in this chapter. This

helper function removes empty inputs instead of appending everything onto a given list of lists. This prevents empty lists from accumulating on the task stack.

Similar to CBPV's small-step semantics, the substitution machine performs substitutions immediately as they appear at the top of the current stacks. The machine state consists of two stacks: the task stack and the value stack. In the initial state, the value stack is empty and the task stack contains the $\gamma(m)$ for a CBPV computation m. On successful termination, a final value is produced on the value stack. Note that the value stack (despite its name) will sometimes contain computations in non-final states. An alternative presentation would be to add an extra stack for computations, but I found no need for this.

The substitution function for programs, written, P^i_Q is similar to that for CBPV terms (Section 4.2), so its definition is elided. Figure 4.5 describes the transition rules for the substitution machine in a tabular format. The three columns represent the task stack, the value stack, and the assumptions. Each $\triangleright$ represents one transition step, where the row above $\triangleright$ represents the current state of the machine, and the row on the same level as $\triangleright$ represents the next state of the machine after the transition. While it may seem similar to the subsitution machine for WCBV (Figure 3.5), I carefully decide the evaluation ordering for all the additional constructors, as they exhibit different behaviours. Such decision was already reflected in the prior definition of the compilation functions (Definition 19, Definition 47), where the ordering of the operands after compiling implicitly suggests which of them will be evaluated or otherwise acted on first. Similar design process applies to the heap machine in the later section as well.

Let's look at the transition for thunkT as an example. It basically strips one layer of $\mathsf{thunkT} :: \cdots :: [\mathsf{endThunkT}]$ off the task stack and places it on the value stack. In the current state, we have $(\mathsf{thunkT} :: P) :: T$ in the task stack, V in the value stack. We further have the assumption that extracting P (using the extraction function φ) gives the body of the thunkT operation as M and stores the rest of the program in P as Q. Using the transition rule, it updates the task stack to be Q (the part of P that is not included in the thunkT operation) appended to T using the appending function from Definition 23; it also adds the whole thunkT operation on the existing value stack.

Note that the transition rules for seqT and pseqT can strip retT components from the value stack directly without the extraction function φ. For instance, in the seqT rule, $\mathsf{retT} :: U +\!\!+ [\mathsf{endRetT}]$ is just the first element on the

value stack. We can strip it with a simple list operation. Furthermore, since there is no extra tailing programs after $\mathsf{endRetT}$ in $\mathsf{retT} :: U +\!\!+ [\mathsf{endRetT}]$, we can obtain U by removing retT and $\mathsf{endRetT}$ using simple list operations.

The transition rule for varT is not strictly necessary: we only consider closed terms, so the rule will never be exercised when running the compilation output. Nonetheless, including it appears to make the proofs more ergonomic, by making it unnecessary to carry around a closedness side condition. For example, consider the useful technical lemma

$$((\gamma(v) :: P) :: T, V) \triangleright (P ::_{tc} T, \gamma(v) :: V)$$

which holds unconditionally when the varT rule is present in the substitution machine semantics. If we remove the rule, it only holds when v is a thunk, which means we could not apply this lemma in the proof of this section's main theorems without carrying around an explicit closedness assumption.

Multiple transitions are written as $(T, V) \triangleright_k^\sigma (T', V')$. We obtain a new state (T', V') after applying the transition rules k times on the current state (T, V), with the size of the biggest intermediate state being σ. We elide σ or k when irrelevant. We write $\triangleright_*$ to represent 0 or more transition steps.

I prove that the substitution machine can simulate CBPV with constant time overhead. In particular, I formally verify that the following time bounds hold.

Lemma 53 (Substitution Machine Time Simulation)**.** *If $m \Downarrow_k n$, then there exists k' such that $([P], []) \triangleright_{k'} ([], [P'])$ where $k' \leq 3 * k + 1$ and $P \gg m$ and $P' \gg n$.*

Proof. The proof proceeds by induction on the structure of the big-step semantics of CBPV. We exhibit each reduction rule and show that their corresponding transition rule in the substitution machine will have the time cost in the required bound. □

Moreover, I prove that the space cost of the substitution machine is similar to that of CBPV. More specifically, I formally verify the following constant space bound hold.

Lemma 54 (Substitution Machine Space Simulation)**.** *If $m \Downarrow^s n$ then there exists σ such that $([P], []) \triangleright_*^\sigma ([], [P'])$, where $s \leq \sigma \leq 9 * s$ and $P \gg m$ and $P' \gg n$.*

	Task Stack	Value Stack	Assumption
	$(\mathsf{varT}\ n :: P) :: T$	V	
$\triangleright$	$P ::_{tc} T$	$\mathsf{varT}\ n :: V$	
	$(\mathsf{thunkT} :: P) :: T$	V	${}_t\varphi P = (M, Q)$
$\triangleright$	$Q ::_{tc} T$	$\mathsf{thunkT} :: M +\!\!+ [\mathsf{endThunkT}\,] :: V$	
	$(\mathsf{forceT} :: P) :: T$	$(\mathsf{thunkT} :: K) :: V$	${}_t\varphi K = (M, [\,])$
$\triangleright$	$(M +\!\!+ P) ::_{tc} T$	V	
	$(\mathsf{lamT} :: P) :: T$	V	${}_l\varphi P = (M, Q)$
$\triangleright$	$Q ::_{tc} T$	$(\mathsf{lamT} :: M +\!\!+ [\mathsf{endLamT}\,]) :: V$	
	$(\mathsf{applyT} :: P) :: T$	$Q :: (\mathsf{lamT} :: M +\!\!+ [\mathsf{endLamT}\,]) :: V$	
$\triangleright$	$M^0_Q :: (P ::_{tc} T)$	V	
	$(\mathsf{retT} :: P) :: T$	V	${}_r\varphi P = (U, Q)$
$\triangleright$	$Q ::_{tc} T$	$(\mathsf{retT} :: U +\!\!+ [\mathsf{endRetT}\,]) :: V$	
	$(\mathsf{seqT} :: P) :: T$	$(\mathsf{retT} :: U +\!\!+ [\mathsf{endRetT}\,]) :: V$	${}_s\varphi P = (N, Q)$
$\triangleright$	$N^0_U :: (Q ::_{tc} T)$	V	
	$(\mathsf{pseqT} :: P) :: T$	$(\mathsf{retT} :: U_2 +\!\!+ [\mathsf{endRetT}\,]) :: (\mathsf{retT} :: U_1 +\!\!+ [\mathsf{endRetT}\,]) :: V$	${}_p\varphi P = (N, Q)$
$\triangleright$	$(N^0_{U_1})^1_{U_2} :: (Q ::_{tc} T)$	V	
	$(\mathsf{letinT} :: P) :: T$	$K :: V$	${}_i\varphi P = (M, Q)$
$\triangleright$	$M^0_K :: (Q ::_{tc} T)$	V	

FIGURE 4.5: Transition Rules for the CBPV Substitution Machine

Proof. Similar to the time simulation proof, we induct on the structure of the big-step semantics of CBPV and show that this theorem is true for all the transition rules. □

4.4.2 Heap Machine

In this section, I introduce the heap machine, an environment-based abstract machine used for simulating CBPV. Free variables in programs are interpreted as heap pointers and the machine uses a heap to store the values of such variables. I adapt the relevant definitions from the previous chapter (Section 3.4.2), including closure type, heap type, and other helper functions (Definition 27, Definition 28, Figure 3.6, Definition 29, Definition 30, Definition 31, and Definition 32).

With these definitions in place, we can now construct our heap machine. The heap structure for the heap machine works similarly as the previous chapter, where the states of the heap machine is defined as triplets consisting of a task stack, a value stack and a heap. The heap machine deals with variables and substitution-related operations very differently to substitution machines. It stores values in the heap and substitutes variables with pointers instead of directly substituting all the values in-place.

The transition rules for the heap machine are shown in Figure 4.6. They read similarly to the rules for the CBPV substitution machine (Figure 4.5) and the rules for the WCBV heap machine (Figure 3.7).

Comparing to the CBPV substitution machine, there are some minor differences that are not directly related to substitution, but are convenient in the proofs; for example, I spell out the $\mathsf{thunk}\mathsf{T}$ structure for the $\mathsf{force}\mathsf{T}$ case, so this rule does not have to use the φ function. Other than that, I also made a major change to the $\mathsf{ret}\mathsf{T}$ case. The observant reader may recall that the compilation function for substitution machines defined in Definition 46 had an $\mathsf{endRet}\mathsf{T}$, which I drop in the case of heap machine. This allows the heap machine to evaluate the value inside $\mathsf{ret}\mathsf{T}$ first. In the substitution machine, I simply put the whole value on the value stack directly. This works for the substitution machine because in CBPV the values do not require further reduction. However, this becomes an issue in the heap machine because the value inside $\mathsf{ret}\mathsf{T}$ can now be a free variable, which needs to be mapped to its value in the heap.

In comparison to the WCBV heap machine, while the overlapping rules behave the same, the CBPV heap machine has more extra constructors and

thus many different reduction cases that need to be taken care of in the proof, which created a new level of difficulty in the formalisation process.

In the proofs, I write heap machine transitions as $(T, V, H) \triangleright_k^\sigma (T', V', H')$, where (T, V, H) is a triple representing the current task stack, value stack, and heap. We obtain a new state (T', V', H') after applying the transition rules k times on the current state, with the size of the biggest intermediate state being σ. We leave the σ or k out when they are not relevant to the proof.

Similar to the WCBV case, before verifying the simulation from heap machine to CBPV, we need to first define the correspondence between programs with heap and CBPV terms. The correspondence between programs and CBPV terms in the presence of an environment is more complicated compared to the prior correspondence we defined for regular programs and CBPV terms (Definition 51). In order to reason about CBPV terms with the heap, I define an *unfolding* judgement to relate a given heap-dependent CBPV term to a heap-independent CBPV term that is semantically equivalent.

The unfolding judgement $(H, a | m_1) \rightsquigarrow_k m_2$ (Figure 4.7) takes as inputs a heap H, a pointer a, a variable bound number k, and two CBPV terms m_1 and m_2. It returns true if and only if m_2 is identical to m_1 with all of its free variables replaced by their values in the heap H.

The two variable rules are the interesting ones. The first variable rule is also present in the unfolding rules for WCBV (Figure 3.8), representing the case where the variable mapping actually happens. The second variable rule (at the bottom of Figure 4.7) is for the case when the variable is free, so it substitutes the variable away using the value that this variable represents on the heap.

I then define a heap-aware correspondence using the unfolding function as follows:

Definition 55 (Closure-Term Correspondence with Heap)**.** *: For any closure $\langle P, a\rangle$ with heap H and CBPV term n, $\langle P, a\rangle$ is the corresponding closure for n (written as $\langle P, a\rangle \gg_H n$) if and only if there exists a CBPV term m such that $(H, a | m) \rightsquigarrow_0 n$ and $\gamma'(m) = P$.*

I prove in HOL4 that the heap machine can simulate CBPV with linear overhead in time:

	Task Stack	Value Stack	Heap	Assumption
	$\langle \mathsf{varT}\ x :: P,\ a\rangle :: T$	V	H	$\mathsf{lookup}\ H\ a\ x = g$
$\triangleright$	$\langle P,\ a\rangle :: T$	$g :: V$	H	
	$\langle \mathsf{thunkT} :: P,\ a\rangle :: T$	V	H	${}_t\varphi P = (M, Q)$
$\triangleright$	$\langle Q,\ a\rangle :: T$	$\langle \mathsf{thunkT} :: M +\!\!+ [\mathsf{endThunkT}],\ a\rangle :: V$	H	
	$\langle \mathsf{forceT} :: P,\ a\rangle :: T$	$\langle \mathsf{thunkT} :: M +\!\!+ [\mathsf{endThunkT}],\ b\rangle :: V$	H	
$\triangleright$	$\langle M,\ b\rangle :: \langle P,\ a\rangle :: T$	V	H	
	$\langle \mathsf{lamT} :: P,\ a\rangle :: T$	V	H	${}_l\varphi P = (M, Q)$
$\triangleright$	$\langle Q,\ a\rangle :: T$	$\langle \mathsf{lamT} :: M +\!\!+ [\mathsf{endLamT}],\ a\rangle :: V$	H	
	$\langle \mathsf{applyT} :: P,\ a\rangle :: T$	$Q :: \langle \mathsf{lamT} :: M +\!\!+ [\mathsf{endLamT}],\ b\rangle :: V$	H	$\mathsf{put}\ H(Q,\ b)\ = (H', c)$
$\triangleright$	$\langle M,\ c\rangle :: \langle P,\ a\rangle :: T$	V	H'	
	$\langle \mathsf{retT} :: P,\ a\rangle :: T$	$\langle M,\ b\rangle :: V$	H	
$\triangleright$	$\langle P,\ a\rangle :: T$	$\langle M,\ b\rangle :: V$	H	
	$\langle \mathsf{seqT} :: P,\ a\rangle :: T$	$\langle M,\ b\rangle :: V$	H	${}_s\varphi P = (N, Q)$ $\mathsf{put}\ H(\langle M,\ b\rangle,\ a)\ = (H', c)$
$\triangleright$	$\langle N,\ c\rangle :: \langle Q,\ a\rangle :: T$	V	H'	
	$\langle \mathsf{pseqT} :: P,\ a\rangle :: T$	$\langle M_2,\ b_2\rangle :: \langle M_1,\ b_1\rangle :: V$	H	${}_p\varphi P = (N, Q)$ $\mathsf{put}\ H(\langle M_2,\ b_2\rangle,\ a)\ = (H_1, c_1)$ $\mathsf{put}\ H_1(\langle M_1,\ b_1\rangle,\ a)\ = (H_2, c_2)$
$\triangleright$	$\langle N,\ c_2\rangle :: \langle Q,\ a\rangle :: T$	V	H_2	
	$\langle \mathsf{letinT} :: P,\ a\rangle :: T$	$K :: V$	H	${}_i\varphi P = (M, Q)$ $\mathsf{put}\ H(K,\ a)\ = (H', b)$
$\triangleright$	$\langle M,\ b\rangle :: \langle Q,\ a\rangle :: T$	V	H'	
	$\langle [\,],\ a\rangle :: T$	V	H	
$\triangleright$	T	V	H	

FIGURE 4.6: Transition Rules for the CBPV Heap Machine

$$\frac{x < k}{(H,\, a|\, (\mathsf{var}\, x)) \rightsquigarrow_k (\mathsf{var}\, x)} \qquad \frac{(H,\, a|\, m) \rightsquigarrow_{k+1} m'}{(H,\, a|\, (\lambda.\, m)) \rightsquigarrow_k (\lambda.\, m)'}$$

$$\frac{(H,\, a|\, v) \rightsquigarrow_k v'}{(H,\, a|\, (\mathsf{ret}\, v)) \rightsquigarrow_k (\mathsf{ret}\, v)'} \qquad \frac{(H,\, a|\, m) \rightsquigarrow_k m'}{(H,\, a|\, (\mathsf{thunk}\, m)) \rightsquigarrow_k (\mathsf{thunk}\, m)'}$$

$$\frac{(H,\, a|\, v) \rightsquigarrow_k v'}{(H,\, a|\, (\mathsf{force}\, v)) \rightsquigarrow_k (\mathsf{force}\, v)'} \qquad \frac{(H,\, a|\, m) \rightsquigarrow_k m' \qquad (H,\, a|\, v) \rightsquigarrow_k v'}{(H,\, a|\, (\mathsf{app}\, m\, v)) \rightsquigarrow_k (\mathsf{app}\, m')v'}$$

$$\frac{(H,\, a|\, m) \rightsquigarrow_k m' \qquad (H,\, a|\, n) \rightsquigarrow_{k+1} n'}{(H,\, a|\, (\mathsf{seq}\, m\, n)) \rightsquigarrow_k (\mathsf{seq}\, m'\, n')}$$

$$\frac{(H,\, a|\, m_1) \rightsquigarrow_k m_1' \qquad (H,\, a|\, m_2) \rightsquigarrow_k m_2' \qquad (H,\, a|\, n) \rightsquigarrow_{k+2} n'}{(H,\, a|\, (\mathsf{pseq}\, m_1\, m_2\, n)) \rightsquigarrow_k (\mathsf{pseq}\, m_1'\, m_2'\, n')}$$

$$\frac{(H,\, a|\, v) \rightsquigarrow_k v' \qquad (H,\, a|\, m) \rightsquigarrow_{k+1} m'}{(H,\, a|\, (\mathsf{let}\, v.\, \mathsf{in}\, m)) \rightsquigarrow_k (\mathsf{let}\, v'.\, \mathsf{in}\, m')}$$

$$\frac{k \leq x \qquad \mathsf{lookup}\, H\, a\, (x - k) = \langle \gamma((\mathsf{thunk}\, m)),\, b\rangle \qquad (H,\, b|\, (\mathsf{thunk}\, m)) \rightsquigarrow_0 v}{(H,\, a|\, (\mathsf{var}\, x)) \rightsquigarrow_k v}$$

FIGURE 4.7: Unfolding Rules for CBPV

Lemma 56 (Heap Machine Time Simulation). *If $m \Downarrow_k n$ then there exists k' such that $([\langle P, 0\rangle], [\,], [\,]) \triangleright_{k'} ([\,], [\langle P', a\rangle], H)$, where $k' \leq 7 * k + 5$ and $\langle P, 0\rangle \gg m$ and $\langle P', a\rangle \gg_H n$.*

Proof. The proof proceeds by induction on the big-step semantics of CBPV. □

The space cost of the heap machine is unrelated to the space cost of the CBPV term: the size of the k^{th} state is bounded by the size of the original CBPV term, and the number of transition steps:

Lemma 57 (Heap Machine Space Simulation). *If $(P, [\,], [\,]) \triangleright_k (T, N, H)$ and $P \gg m$ then $\|T +\!\!+ N +\!\!+ H\| \leq (k+1) * (3 * k + 4 * \|m\|)$.*

Proof. The proof proceeds by showing each stack's length and that all elements in each stack are bounded by the size of the original term m and the number of transitions taken k. □

4.5 Call-By-Push-Value is Reasonable for Time and Space

In the previous section, I first constructed an abstract substitution machine and an abstract heap machine. I then proved that there exists simulation relations between these two abstract machines and CBPV with specific bounds.

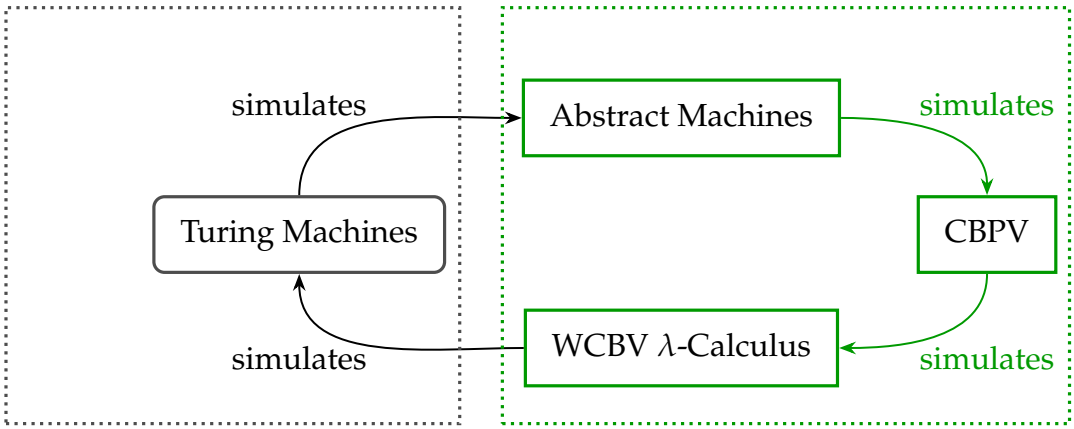
In this section, I use the results from Section 4.4 to prove that CBPV is reasonable. I achieve this by providing two simulations between CBPV and Turing machines, as elaborated in Figure 4.8.

Section 4.5.1 describes how Turing machines can simulate CBPV with polynomial time overhead and constant factor space overhead, fulfilling Theorem 36 in Section 4.1. Note that this simulation goes through the abstract machines we implemented and formalised in Section 4.4.

Section 4.5.2 describes how CBPV can reasonably simulate WCBV. I adapt the result that WCBV can reasonably simulate Turing machines from existing literature (Forster, Kunze, and Roth, 2020). Together, we have that CBPV can simulate Turing machines with polynomial time overhead and constant factor space overhead, fulfilling Theorem 37 in Section 4.1.

4.5.1 Turing Machines Simulating CBPV

In this section I show that it is always possible to construct a Turing machine M_{CBPV} that simulates CBPV with polynomially bounded time overhead and



- Left side:
  Pen-and-paper proof
  based on the proof technique
  by Forster, Kunze, and Roth (2020).
- Right side:
  Mechanised in HOL4 (Chen, 2025).

FIGURE 4.8: Simulation between Turing Machines and CBPV with Intermediate Models

constant factor space overhead. The results from Section 4.4, specifically, Lemmas 53, 54, 56 and 57, will be used here.

My proof is very similar to a proof from the literature Forster, Kunze, and Roth (2020), which I explained briefly in the previous chapter (Section 3.5). In their work, similar heap and substitution machines are interleaved to obtain a simulation of WCBV. Their proof relies on formalised results similar to my formalised results, and proves that the interleaving strategy always obeys the required time and space bounds. I demonstrate that their argument extends to the more general case of CBPV. While my abstract machines are much more involved, they have similar time and space bounds, which simplifies adaptation.

In the previous chapter, I explained the proof in a bottom-up manner. In this section, I will do the reverse, and explain from a top-down perspective, adding more intuitions of how we reached to that machine construction step.

In order to show that Turing machines can simulate CBPV, we must prove that one can always construct a Turing machine M_{CBPV} that simulates the given CBPV term within the required bounds. We obtain such M_{CBPV} by constructing and interleaving Turing machines M_{subst} and M_{heap} that simulate the respective abstract machines. Provided an input s which has a normal form t, the algorithm starts by applying M_{subst} over (the program equivalent of) s to reduce it. If a size explosion occurs during this reduction, execution then

switches to use M_{heap} before the explosion happens. In this case, the heap-based strategy is guaranteed not to encounter the pointer explosion problem. That is because the terms that causes size explosions result in a space cost of $\mathcal{O}(n^2)$, which easily accommodates the $\log n$ space cost for pointer storage in the heap machine. This algorithm guarantees termination and reasonable time and space overhead.

The substitution-based Turing machine M_{subst} is constructed by iterating over smaller Turing machines each simulating an individual transition rule from Figure 4.5. Similarly, the heap-based Turing machine M_{heap} is constructed by iterating over Turing machines simulating the rules of Figure 4.6. In addition to the original CBPV term s, each of these machines takes as input a number of steps k that they are meant to simulate. The M_{subst} takes an additional input σ and aborts if the amount of space used is larger than σ. The encoding of the machine transition rules in Turing machines is straightforward, where we use 13 symbols to represent the 13 constructors(tokens) in the substitution machine. Similarly, we use 12 symbols to represent the 12 constructors(tokens) in the heap machine.

Now let's construct the Turing machines. In the following theorems, I write $\|s\|_T$ for the number of transition steps and $\|s\|_S$ for size of the biggest intermediate term.

We first consider the substitution machine. M_{subst} takes as inputs a term s, two numbers k and σ. s is the term to be reduced, k represents the number of transition steps to perform, and σ represents the space threshold over which the machine should abort. We show that this machine must satisfy one of the following three conditions: (1) it returns a desired value t within k steps; (2) it reaches the space bound and halts; (3) it finishes k reductions and halts (within both space and time bounds). This is formally stated below as Theorem 58.

Theorem 58. *There exists a Turing machine M_{subst} that takes as inputs k, σ and a term s. It halts in time $\mathcal{O}(k \cdot poly(min\ (\sigma, \|s\|_S)))$ and space $\mathcal{O}(min\ (\sigma, \|s\|_S) + log\ \sigma + log\ k)$ while one of the following statements holds:*

- *The machine outputs a term t, then s has normal form t and $\sigma \geq \|s\|_S$ and $k \geq 3 \cdot \|s\|_T + 1$.*
- *The machine halts in a state named space bound not reached and $k \leq 3 \cdot \|s\|_T + 1$ holds.*
- *The machine halts in a state named space bound reached and $\sigma \leq 9 \cdot \|s\|_S$ holds.*

Proof. We can construct a Turing machine M_{subst} by looping Turing machines that implement the individual steps of the abstract substitution machine. We add an extra rule where this machine has to halt if it were to reach a state with size larger than σ. Note that the extra rule requires M_{subst} to halt even if it is in the middle of execution, in order to avoid the size explosion problem. With an initialisation function τ that converts s from a λ-term into a program $\tau(s)$, we now have the desired Turing machine M_{subst}.

The time and space cost from the substitution machine mostly carry over directly, but the extraction functions φ cannot be implemented in constant time. For instance, $\varphi^k_Q P$ takes time and space $\mathcal{O}(\|Q\| + \|P\| + k)$.

From Lemma 54, we know that if $s \Downarrow^{\|s\|_S} t$ then there exists σ such that $(P_s, []) \triangleright^{\sigma}_{*} ([], P_t)$, where $\|s\|_S \leq \sigma \leq 9 * \|s\|_S$ and $P_s \gg s$ and $P_t \gg t$. Thus the size of all intermediate states and the overall space consumption lie within $9 * \|s\|_S$.

From Lemma 53, we know that if $s \Downarrow_{\|s\|_T} t$, then there exists k such that $(P_s, []) \triangleright_k ([], P_t)$ where $k \leq 3 * \|s\|_T + 1$ and $P_s \gg s$ and $P_t \gg t$. Thus there must exist a k that is large enough to simulate the reduction while still lying within the bound $3 * \|s\|_T + 1$. □

The next step is to construct the heap-based Turing machine M_{heap}:

Theorem 59. *There exists a Turing machine M_{heap} that, given a number k and a closed term s, halts in time $\mathcal{O}(poly(\|s\|,k))$ and space $\mathcal{O}(\|s\| \cdot poly(k))$. If s has a normal form t and $k \geq 10 \cdot \|s\|_T + 3$, it computes a heap H and a closure g such that $g \gg_H t$. Otherwise, it halts in a distinguished final state (denoting 'failure').*

Proof. We can implement the abstract substitution machine from Section 4.4.2 by looping Turing machines that implement the individual steps of the abstract machine.

Time and space cost of the φ functions is as in Theorem 58. We also have to consider the lookup function. lookup iterates through the heap for n indices using the heap headers (a in this case) as pointers, resulting in at most $\mathcal{O}(n)$ time cost.

Thus each abstract step $(T, V, H) \triangleright (T', V', H')$ can be implemented in $\mathcal{O}(poly(\|(T, V, H)\|))$ time and $\mathcal{O}(max(\|(T, V, H)\|, \|(T', V', H')\|))$ space. The space consumption of all involved operations in Figure 4.6 is bounded by their input or output. Using Lemma 57, the size of all intermediate (T, V, H) can be bound by k and $\|s\|$ to derive the claimed resource bounds. The successful computation of g and H for large enough k follows with Lemma 56.

□

Before constructing M_{CBPV}, we need a lemma to say that unfolding only changes de Bruijn indices starting at k. In particular, closed terms are invariant under unfolding.

Lemma 60 (Unfold Bounded). *If s is bounded by k, then $(s, a|\ H) \rightsquigarrow_k s$.*

Proof. Induction on $s \leq k$. □

We now combine everything together to form our final theorem.

Theorem 61. *There is a Turing machine M_{CBPV} that, given a closed term s that has a normal form t, computes a heap H and a closure g such that $g \gg_H t$ in time $\mathcal{O}(poly(\|s\|, \|s\|_T))$ and space $\mathcal{O}(\|s\|_S)$.*

Proof. By using the interleaving algorithm (Algorithm 1) adapted from WCBV (Forster, Kunze, and Roth, 2020):

Algorithm 1 Interleaving Strategy Algorithm

Let p be the polynomial such that the machine from Theorem 59 runs in space $\mathcal{O}(\|s\| \cdot p(k))$.

1. Initialise $k := 0$ (in binary)
2. Compute $\sigma := \|s\| \cdot p(k)$ (in binary)
3. Run M_{subst} on s, k and σ.
   (a) If M_{subst} computes the normal form t, output $(\gamma(t), 0)$ and an empty heap [ ] and halt.
   (b) If M_{subst} halts with space bound not reached, set $k := k + 1$ and go to 2
   (c) If M_{subst} halts with space bound reached, continue at 4.
4. Run M_{heap} on s and k.
   (a) If this computed a closure g and a heap H representing t, output H and g and halt.
   (b) Otherwise, set $k := k + 1$ and go to 2.

There are four things to prove about the algorithm. We will go through them one by one.

Halting states. M_{CBPV} has only two halting states: when M_{subst} returns (state 3(a) in Algorithm 1), and when M_{heap} returns (state 4(a)). In both cases,

M_{CBPV} returns a closure-heap pair representing the normal form t of s. In the first case, Theorem 58 shows that M_{subst} will return a normal form t of s and Lemma 60 shows that the closure-heap pair we constructed in state 3(a) indeed represents t. The second case is immediate Theorem 59.

Termination. The machine will terminate for terminating terms s and diverge on non-terminating CBPV terms. For the terminating case, we need to show two things: (1). for all k, each iteration eventually finishes and goes to the next iteration; (2). there exists a k such that the machine halts and returns a closure-heap pair. We consider (1) first, and fix k. Time cost in step 1 is constant. Binary computation in Turing machines have polynomial cost, thus the time cost in step 2 is $\mathcal{O}(poly(\|s\|,(k)))$. Using Theorem 58, step 3 takes time

$$\begin{array}{lll} \mathcal{O}(k \cdot poly(min(\sigma, \|s\|_S))) & \subseteq & \mathcal{O}(k \cdot poly(\sigma)) \\ & = & \mathcal{O}(k \cdot poly(\|s\| \cdot p(k))) \qquad \text{from step 2} \\ & \subseteq & \mathcal{O}(k \cdot poly(\|s\|, k)) \qquad p \text{ is a polynomial} \\ & \subseteq & \mathcal{O}(poly(\|s\|, k)) \end{array}$$

If Step 4 is executed, this takes time $\mathcal{O}(poly(\|s\|,k))$ by Theorem 59. Since each of the four steps has at most time complexity $\mathcal{O}(poly(\|s\|,k))$, one iteration has time cost $\mathcal{O}(poly(\|s\|,k))$, which suffices for (1). For (2), consider $k = 10\|s\|_T + 3$, which is larger than the two values required in Theorem 58 and Theorem 59. By Theorem 58, the machine does halt during Step 3, unless $\sigma \leq \|s\|_S$. In the latter case, 4 is tried. Then, by Theorem 59, as k is large enough, we have that M_{heap} indeed halts with a closure-heap pair.

Time Complexity. We have proved the time cost for each iteration, so we just need to sum up all the iterations for the overall time complexity for M_{CBPV}:

$$\mathcal{O}\left(\sum_{k=0}^{10\|s\|_T+3} (poly(\|s\|,k))\right) \subseteq \mathcal{O}(\|s\|_T \cdot poly(\|s\|, \|s\|_T)) \subseteq \mathcal{O}(poly(\|s\|, \|s\|_T))$$

Space Complexity. We first analyse the space cost for one iteration with an arbitrary k. Step 1 is constant. Step 2 takes $\mathcal{O}(log(\sigma))$ space since the computation is in binary. By Theorem 58, Step 3 takes space $\mathcal{O}(min(\sigma, \|s\|_S) + \log\sigma + \log k) \subseteq \mathcal{O}(\|s\|_S + \log\sigma + \log k)$. If step 3(c) reaches the space bound ($\sigma \leq 3 \cdot \|s\|_S$), step 4 is tried. Together with Theorem 59 and definition of m, the space cost for step 4 is $\mathcal{O}(\|s\| \cdot p(k)) \subseteq \mathcal{O}(\sigma) \subseteq \mathcal{O}(\|s\|_S)$.

Thus we have the space cost of one iteration:

$$\begin{aligned}
& \mathcal{O}(\|s\|_S + \log\sigma + \log k) \\
= \ & \mathcal{O}(\|s\|_S + \log(\|s\| \cdot p(k)) + \log k) && \text{(definition of } \sigma) \\
\subseteq \ & \mathcal{O}(\|s\|_S + \log\|s\| + \log(p(k)) + \log k) \\
= \ & \mathcal{O}(\|s\|_S + \log(p(k)) + \log k) && \text{(as } \|s\| \leq \|s\|_S) \\
= \ & \mathcal{O}(\|s\|_S + \log(k^c) + \log k) && (c \text{ const., } p \text{ poly.}) \\
= \ & \mathcal{O}(\|s\|_S + c\log(k) + \log k) \\
\subseteq \ & \mathcal{O}(\|s\|_S + \log k)
\end{aligned}$$

The space cost for all iterations is as follows, where the last equation is by Lemma 62:

$$\mathcal{O}(\max_{0\leq k\leq 10\|s\|_T+3}(\|s\|_S + \log k)) \subseteq \mathcal{O}(\|s\|_S + \log\|s\|_T) = \mathcal{O}(\|s\|_S)$$

□

By combining our formalisation with results above, we obtain Theorem 36. For terms with $\|s\|_T \notin \mathcal{O}(\|s\|_S)$ it is crucial that the machine tracks the step number k in binary, because it would need $\Omega(\|s\|_T)$ space otherwise. This suffices because of lemma 62:

Lemma 62. $\log\|s\|_T \in \mathcal{O}(\|s\|_S)$.

Proof. As the vocabulary is finite, there are at most exponentially many terms for a given size. A reduction from s cannot visit the same term twice since reduction is deterministic. $\|s\|_S$ is the biggest intermediate term, which means that all the terms in the reduction for s will be smaller than $\|s\|_S$. This implies that $\|s\|_T$ will be at most equal to the total number of terms smaller than $\|s\|_S$. Formally: $\|s\|_T \leq c^{\|s\|_S}$ for a constant c.

To see that the number of terms smaller than a given size σ is at most exponential, note that $\#\{t|\ \|t\| \leq \sigma\} = \#\{\|t\| \mid \|\gamma(t)\| \leq 2\cdot\sigma\}$ by Lemma 48. Because γ is injective we have $\#\{\|t\| \mid \|\gamma(t)\| \leq 2\cdot\sigma\} = \#\{P|\ \|P\| \leq 2\cdot\sigma\}$, which is $\leq \#\{P|\ \|P\| \leq 2\cdot\sigma\}$. Finally, $\#\{P|\ \|P\| \leq 2\cdot\sigma\} \leq 5n-1$ follows by induction on n. The 5 is because there are four different symbols a program can start with, and variables indices use a fifth symbol. □

4.5.2 CBPV Simulating Turing Machines

In this section, I prove that CBPV can simulate Turing machines with reasonable time and space overhead. I achieve this by using WCBV as an intermediate model. There is existing work showing WCBV can simulate Turing machines with reasonable time and space overhead (Forster, Kunze, and Roth, 2020). Thus what remains to be shown in this section is a reasonable simulation of WCBV using CBPV. With prior work, this suffices to prove Theorem 37.

Recall the syntax (Definition 15), time cost semantics (Figure 3.3), and space cost semantics (Figure 3.4) of WCBV from Chapter 3.
The compilation function $c(t)$ is then defined as follows:

Definition 63 (Compilation Function (WCBV to CBPV))**.**

$$\begin{array}{lcllcl} c(\mathsf{var}\, x) & = & \mathsf{var}\, x & c(\lambda.\, t) & = & \mathsf{ret}\,\mathsf{thunk}\, \lambda.\, c(t) \\ c(\mathsf{app}\, t\, u) & = & \mathsf{pseq}\, (c(u))\, (c(t))\, (\mathsf{app}\, (\mathsf{force}\,\mathsf{var}\, 0)\, (\mathsf{var}\, 1)) & & & \end{array}$$

I prove that CBPV can simulate WCBV with constant overheads, by a routine induction:

Theorem 64. *For each closed WCBV term t, we have:*

1. *(Time) If $t \Downarrow_k u$, then there exists k' such that $c(t) \Downarrow_{k'} c(u)$ and $k' \leq 5 * k$; and*
2. *(Space) If $t \dot{\Downarrow}^s u$, then there exists s' such that $c(t) \Downarrow^{s'} c(u)$ and $s' \leq 6 * s$.*

This proves Theorem 37, which together with Theorem 36 from the previous section, immediately shows that CBPV is indeed reasonable for both time and space (Theorem 38).
Note that the standard translation uses seq twice instead of pseq once, like this:

$$\begin{array}{lcllcl} c'_\eta(\mathsf{var}\, x) & = & \mathsf{var}\, \eta(x) & c'_\eta(\lambda.\, t) & = & \mathsf{ret}\,\mathsf{thunk}\, \lambda.\, c'_{\eta\Uparrow[0\mapsto 0]}(t) \\ c'_\eta(\mathsf{app}\, t\, u) & = & \mathsf{seq}\, (c'_\eta(u))\, (\mathsf{seq}\, c'_{\eta\uparrow}(t)\, (\mathsf{app}\, (\mathsf{force}\,\mathsf{var}\, 0)\, (\mathsf{var}\, 1))) & & & \end{array}$$

Some de Bruijn arithmetic is needed to account for the extra binder in the (app) case. *Environments* η are (partial) functions from $\mathbb{N}$ to $\mathbb{N}$. We define lifting operators as follows:

Definition 65. *Lifting*

$$\eta^{\uparrow} \equiv x \mapsto \eta(x) + 1 \qquad \eta^{\Uparrow} \equiv x \mapsto \eta(x+1) + 1$$
$$\eta[x \mapsto y](z) \equiv \begin{cases} y & \text{if } x = z \\ \eta(z) & \text{otherwise} \end{cases}$$

However, it turns out that c' has linear space overhead, because there exists terms t such that $\|c'(t)\| = \Omega(\|t\|^2)$, as shown by the following example. Consider a term t_n consisting of n right-associated applications, followed by n occurrences of the variable 0. For example, we'd have the following, where I denotes the identity function $(\lambda.\,(\mathsf{var}\,0))$:

$$\begin{aligned} t_1 &\equiv \mathsf{app}\,\mathsf{I}\,(\mathsf{var}\,0) \\ t_2 &\equiv \mathsf{app}\,\mathsf{I}\,(\mathsf{app}\,\mathsf{I}\,(\mathsf{app}\,(\mathsf{var}\,0)\,(\mathsf{var}\,0))) \\ t_3 &\equiv \mathsf{app}\,\mathsf{I}\,(\mathsf{app}\,\mathsf{I}\,(\mathsf{app}\,\mathsf{I}\,(\mathsf{app}\,(\mathsf{app}\,(\mathsf{var}\,0)\,(\mathsf{var}\,0))\,(\mathsf{var}\,0)))) \\ &\dots \end{aligned}$$

We have $size(t_n) = \mathcal{O}(n)$. But if we consider $c'(t_n)$, the n occurrences of $(\mathsf{var}\,0)$ in t_n become n occurrences of $(\mathsf{var}\,n)$, and hence $size(t_n) = \Omega(n^2)$.

This technicality arises because de Bruijn indices count towards term size. We must count like so because numbers are not representable in constant space on Turing machines.

Fortunately, the problematic additional bindings introduced by applications are vacuous over the operand. Translating WCBV to CBPV requires introducing intermediary bindings, and we cannot solve the issue by shuffling arguments: it is necessary for either the operator or operand of an application to be in the scope of a vacuous binding. Based on this observation, while other alternative approaches could be possible, we found pseq to be the most intuitive and does not introduce extra problems.

Do we need to go to Turing machines?

Since WCBV is known to be reasonable, the reader may wonder if we must go all the way to Turing machines to prove Theorem 37. Wouldn't going to WCBV be simpler? This turns out to be straightforward for time cost, but unfortunately the natural encoding of CBPV in WCBV has linear space

overhead (cf. Section 4.5.2). Consider this compilation function:

$$
\begin{array}{lcllcl}
d(\mathsf{var}\, x) & = & \mathsf{var}\, x & d(\mathsf{thunk}\, m) & = & \lambda.\, d(m)^{\uparrow} \\
d(\mathsf{force}\, v) & = & \mathsf{app}\, (d(v))\, \lambda.\, \mathsf{var}\, 0 & d(\mathsf{ret}\, v) & = & d(v) \\
d(\lambda.\, m) & = & \lambda.\, d(m) & d(\mathsf{app}\, m\, v) & = & \mathsf{app}\, (d(m))\, (d(v)) \\
d(\mathsf{seq}\, m\, n) & = & \mathsf{app}\, (\lambda.\, d(m))\, (d(n)) & d(\mathsf{let}\, v.\, \mathsf{in}\, m) & = & \mathsf{app}\, (\lambda.\, d(m))\, (d(v)) \\
d(\mathsf{pseq}\, m_2\, m_1\, n) & = & \mathsf{app}\, (\mathsf{app}\, (\lambda.\, \lambda.\, d(n))\, (d(m_2)))\, (d(m_1)) & & &
\end{array}
$$

We flatten the distinction between values and computations, and to suspend computations we use the one mechanism on offer: λ-abstraction. We have proved that this compilation strategy is reasonable for time cost. Unfortunately, it does not yield a reasonable space cost model. To see why, consider the following variation on the example from the previous section.

$$
\begin{array}{lcl}
t_1 & \equiv & \mathsf{app}\, (\mathsf{force}\, \mathsf{var}\, 0)\, (\mathsf{var}\, 0) \\
t_2 & \equiv & \mathsf{force}\, \mathsf{thunk}\, \mathsf{app}\, (\mathsf{app}\, (\mathsf{force}\, \mathsf{var}\, 0)\, (\mathsf{var}\, 0))\, (\mathsf{var}\, 0) \\
t_3 & \equiv & \mathsf{force}\, \mathsf{thunk}\, \mathsf{force}\, \mathsf{thunk}\, \mathsf{app}\, (\mathsf{app}\, (\mathsf{app}\, (\mathsf{force}\, \mathsf{var}\, 0)\, (\mathsf{var}\, 0))\, (\mathsf{var}\, 0))\, (\mathsf{var}\, 0) \\
\ldots & &
\end{array}
$$

That is, term t_n contains $\mathcal{O}(n)$ mentions of $(\mathsf{var}\, 0)$ under $\mathcal{O}(n)$ layers of thunks. We have $size(t_n) = \mathcal{O}(n)$. But when we consider $d(t_n)$, the n occurrences of $(\mathsf{var}\, 0)$ in t_n become n occurrences of $(\mathsf{var}\, n)$, and hence $size(d(t_n)) = \Omega(n^2)$.

Clearly a reasonable encoding in this direction is possible: the detour via Turing machines would yield one. But the choice of encoding function would not be obvious.

Chapter 5

Extending Call-By-Push-Value with Call-By-Need

In this chapter, I demonstrate how to extend my formalisation of CBPV with call-by-need. There has been pen-and-paper work on this topic by McDermott and Mycroft (2019). They extend Levy's CBPV using shared reductions, obtaining a call-by-need version of CBPV, called *extended call-by-push-value* (ECBPV). I follow their naming convention and refer to my extended CBPV as ECBPV too. I take inspiration from their shared reduction to develop and formalise my ECBPV in HOL4.

This work can lead to the development of cost models for ECBPV by combining the formalisation of ECBPV language in this chapter, and the formalisation of cost models for CBPV in Chapter 4.

5.1 Syntax

The original CBPV does not suffice for simulating call-by-need strategy. This is because CBPV itself does not have any functionality that supports memory sharing, which is a crucial requirement for applying call-by-need strategy. In order to embed call-by-need into CBPV, we need to have a way to enable memory sharing. That is, we need to be able to share the results of computations that have already been evaluated and know where to share these results to. To achieve this, I introduce new syntactic constructs called need and needvar. The rest of the syntax for ECBPV follows the syntax of CBPV:

Definition 66 (Syntax for ECBPV)**.**

Values	V	:=	var x |	thunk M			
Computations	M	:=	λ. M |	app M V |	force V |	ret V |	let V. in M |
			seq M M |	pseq M M M |	need M M |	needvar x	

Note that the need construct takes two computations as arguments, rather than one computation and one value like in application. need $m\,n$ is similar to seq $m\,n$ in the sense that m is the first computation in the sequence, and n takes in (the result of) m as an input. The difference is that in seq $m\,n$, the result of m is computed first and passed in n as an argument. In the case for need $m\,n$, m is only computed if it is *needed* by n. Another new construct is needvar x, which reprents a *need* variable at index x.

The size function for ECBPV works similarly as the ones for WCBV and CBPV, where de Bruijn indices are taken into account:

Definition 67 (Size functions)**.**

$$
\begin{aligned}
\|\mathsf{var}\,n\| &= 1+n \\
\|\mathsf{thunk}\,c\| &= 1+\|c\| \\
\|\mathsf{force}\,v\| &= 1+\|v\| \\
\|\lambda.\,m\| &= 1+\|m\| \\
\|\mathsf{app}\,m\,v\| &= 1+\|m\|+\|v\| \\
\|\mathsf{ret}\,v\| &= 1+\|v\|
\end{aligned}
\qquad
\begin{aligned}
\|\mathsf{seq}\,m\,n\| &= 1+\|m\|+\|n\| \\
\|\mathsf{let}\,v.\,\mathsf{in}\,m\| &= 1+\|v\|+\|m\| \\
\|\mathsf{pseq}\,m_2\,m_1\,n\| &= 1+\|m_2\|+\|m_1\|+\|n\| \\
\|\mathsf{need}\,m\,n\| &= 1+\|m\|+\|n\| \\
\|\mathsf{needvar}\,n\| &= 1+n
\end{aligned}
$$

5.2 Semantics

The substitution process for ECBPV is more complicated than the ones for WCBV and CBPV, because we need to take care of variable sharing. In order to distinguish the substitution for regular variables from the substitution for *need* variables (needvar), I define two separate substitution functions serving these two purposes respectively in Definition 68 and Definition 69.

The notation for the regular substitution is m^0_v, following the ones used for WCBV and CBPV. The regular substitution is defined as follows:

Definition 68 (Substitution function)**.**

$$
\begin{aligned}
(\mathsf{var}\,n)^k_u &= \begin{cases} u & \text{if } n=k \\ \mathsf{var}\,n & \text{if } n\neq k \end{cases} \\
(\mathsf{thunk}\,m)^k_u &= (\mathsf{thunk}\,m^k_u) \\
(\mathsf{force}\,v)^k_u &= \mathsf{force}\,(v^k_u) \\
(\lambda.\,m)^k_u &= \lambda.\,(m^{k+1}_u) \\
(\mathsf{app}\,m\,v)^k_u &= \mathsf{app}\,(m^k_u)\,(v^k_u) \\
(\mathsf{ret}\,v)^k_u &= \mathsf{ret}\,(v^k_u)
\end{aligned}
\qquad
\begin{aligned}
(\mathsf{seq}\,m\,n)^k_u &= \mathsf{seq}\,(m^k_u)\,(n^{k+1}_u) \\
(\mathsf{let}\,v.\,\mathsf{in}\,m)^k_u &= \mathsf{let}\,(v^k_u).\,\mathsf{in}\,(m^{k+1}_u) \\
(\mathsf{pseq}\,m_2\,m_1\,n)^k_u &= \mathsf{pseq}\,(m_{2u}^k)\,(m_{1u}^k)\,(n^{k+2}_u) \\
(\mathsf{need}\,m\,n)^k_u &= \mathsf{need}\,(m^k_u)\,(n^k_u) \\
(\mathsf{needvar}\,x)^k_u &= \mathsf{needvar}\,x
\end{aligned}
$$

In order to distinguish the needvar substitution from the regular one, I use a slightly modified notation $m_v^{\nabla 0}$, where an extra symbol ∇ is added. Note that in the substitution function for needvar (Definition 69), the de Bruijn index is only lifted when we enter a need layer, rather than whenever we enter a λ-abstraction.

Definition 69 (Substitution function for needvar)**.**

$$
\begin{array}{lcl}
(\mathsf{var}\, n)_u^{\nabla k} & = & \mathsf{var}\, n \\
(\mathsf{thunk}\, m)_u^{\nabla k} & = & \mathsf{thunk}\, (m_u^{\nabla k}) \\
(\mathsf{force}\, v)_u^{\nabla k} & = & \mathsf{force}\, (v_u^{\nabla k}) \\
(\lambda. m)_u^{\nabla k} & = & \lambda. (m_u^{\nabla k}) \\
(\mathsf{app}\, m\, v)_u^{\nabla k} & = & \mathsf{app}\, (m_u^{\nabla k})\, (v_u^{\nabla k}) \\
(\mathsf{ret}\, v)_u^{\nabla k} & = & \mathsf{ret}\, (v_u^{\nabla k})
\end{array}
\qquad
\begin{array}{lcl}
(\mathsf{seq}\, m\, n)_u^{\nabla k} & = & \mathsf{seq}\, (m_u^{\nabla k})\, (n_u^{\nabla k}) \\
(\mathsf{let}\, v.\, \mathsf{in}\, m)_u^{\nabla k} & = & \mathsf{let}\, (v_u^{\nabla k}).\, \mathsf{in}\, (m_u^{\nabla k}) \\
(\mathsf{pseq}\, m_2\, m_1\, n)_u^{\nabla k} & = & \mathsf{pseq}\, (m_{2u}^{\nabla k})\, (m_{1u}^{\nabla k})\, (n_u^{\nabla k}) \\
(\mathsf{need}\, m\, n)_u^{\nabla k} & = & \mathsf{need}\, (m_u^{\nabla k})\, (n_u^{\nabla k+1}) \\
(\mathsf{needvar}\, n)_u^{\nabla k} & = & \begin{cases} u & \text{if } n = k \\ \mathsf{needvar}\, n & \text{if } n \neq k \end{cases}
\end{array}
$$

With substitutions defined, we can now move onto the semantics. Starting with small-step semantics. In order to define small-step semantics for ECBPV, we need to first define a primitive step relation $m \to m'$ similar to the one for CBPV:

$$
\frac{}{\mathsf{force}\, (\mathsf{thunk}\, m) \to m}
\qquad
\frac{m_v^0 = m'}{\mathsf{app}\, (\lambda. m)\, v \to m'}
\qquad
\frac{m_v^0 = m'}{\mathsf{let}\, v.\, \mathsf{in}\, m \to m'}
$$

$$
\frac{n_v^0 = n'}{\mathsf{seq}\, (\mathsf{ret}\, v)\, n \to n'}
\qquad
\frac{(n_{v_1}^0)_{v_2}^1 = n'}{\mathsf{pseq}\, (\mathsf{ret}\, v_2)\, (\mathsf{ret}\, v_1)\, n \to n'}
\qquad
\frac{n_{(\mathsf{ret}\, v)}^{\nabla 0} = n'}{(\mathsf{need}\, (\mathsf{ret}\, v)\, n) \to n'}
$$

FIGURE 5.1: Primitive Step for ECBPV

In order to encode the lazy property of ECBPV, which is only evaluate computation when it is *needed*, I define a helper function *demands* $\mathsf{dms}\, m\, k$ (Definition 70) to denote when such occasion rises, where m is an ECBPV computation whose result is *demanded* (needed) and k is a number that represents the de Bruijn index. Note that in this thesis I use the words *need* and *demand* interchangeably to refer to the same concept.

Definition 70 (Demands Function)**.**

$$\frac{}{\mathsf{dms}\,(\mathsf{needvar}\,k)\,k} \qquad \frac{\mathsf{dms}\,m\,k}{\mathsf{dms}\,(\mathsf{app}\,m\,v)\,k} \qquad \frac{\mathsf{dms}\,m\,k}{\mathsf{dms}\,(\mathsf{seq}\,m\,n)\,k}$$

$$\frac{\mathsf{dms}\,m1\,k}{\mathsf{dms}\,(\mathsf{pseq}\,m2\,m1\,n)\,k} \qquad \frac{\mathsf{dms}\,m2\,k}{\mathsf{dms}\,(\mathsf{pseq}\,m2\,(\mathsf{ret}\,v)\,n)\,k} \qquad \frac{\mathsf{dms}\,m\,k \qquad \mathsf{dms}\,n\,0}{\mathsf{dms}\,(\mathsf{need}\,m\,n)\,k}$$

$$\frac{\mathsf{dms}\,n\,(k+1)}{\mathsf{dms}\,(\mathsf{need}\,m\,n)\,k}$$

I also define a helper function to check whether the given input is a return expression:

Definition 71 (Return Predicate)**.** *For any ECBPV term x,* $\mathsf{is\_ret}(x)$ *if and only if there exists an ECBPV term x' such that $x = (\mathsf{ret}\,x')$.*

Putting everything together, we now have the small-step semantics for ECBPV as follows:

$$\frac{m \to m'}{m \rightsquigarrow m'} \qquad \frac{m \rightsquigarrow m'}{\mathsf{app}\,m\,v \rightsquigarrow \mathsf{app}\,m'\,v} \qquad \frac{m \rightsquigarrow m'}{\mathsf{seq}\,m\,n \rightsquigarrow \mathsf{seq}\,m'\,n}$$

$$\frac{m_1 \rightsquigarrow m_1'}{\mathsf{pseq}\,m_2\,m_1\,n \rightsquigarrow \mathsf{pseq}\,m_2\,m_1'\,n} \qquad \frac{m_2 \rightsquigarrow m_2'}{\mathsf{pseq}\,m_2\,(\mathsf{ret}\,v)\,n \rightsquigarrow \mathsf{pseq}\,m_2'\,(\mathsf{ret}\,v)\,n}$$

$$\frac{\mathsf{dms}\,n\,0 \qquad m \rightsquigarrow m'}{(\mathsf{need}\,m\,n) \rightsquigarrow (\mathsf{need}\,m'\,n)} \qquad \frac{n \rightsquigarrow n' \qquad \neg\mathsf{is\_ret}(m)}{(\mathsf{need}\,m\,n) \rightsquigarrow (\mathsf{need}\,m\,n')}$$

FIGURE 5.2: Small-Step Semantics for ECBPV

The last two rules encode the idea that during evaluation of a need term (e.g. need $m\,n$), the reduction of the second computation n happens before the reduction of the first computation m. Furthermore, the first computation m is only evaluated if it is *needed* in n. Note that the result of m is only needed if it will not be thrown away during the evaluation of n.

The above small-step semantics gives an intuitive understanding of the reduction steps of my ECBPV. For it to be more useful for reasoning about cost models, I further define big-step semantics 8or ECBPV, decorated with time cost and space cost respectively in Figure 5.3 and Figure 5.4. The strategy to count time and space cost in the big-step semantics follows from the

ones in the previous chapters (Chapter 3, Chapter 4). To be more specific, the time cost is measured by the number of β-reductions, and the space cost is measured by the size of the biggest intermediate term. The choice of time cost measure is why there are a few rules with 0 time cost in Figure 5.3. In those rules, no β-reduction is performed, thus with our time measure, it has the time cost of 0.

$$\frac{}{\lambda.\, m \Downarrow_0 \lambda.\, m} \qquad \frac{}{\mathsf{ret}\, v \Downarrow_0 \mathsf{ret}\, v} \qquad \frac{}{\mathsf{needvar}\, v \Downarrow_0 \mathsf{needvar}\, v}$$

$$\frac{m \Downarrow_k m'}{\mathsf{force}\,(\mathsf{thunk}\, m) \Downarrow_{k+2} m'} \qquad \frac{m_v^0 \Downarrow_k m'}{\mathsf{let}\, v.\, \mathsf{in}\, m \Downarrow_{k+1} m'} \qquad \frac{m \Downarrow_{k_1} \lambda.\, n \qquad n_v^0 \Downarrow_{k_2} n'}{\mathsf{app}\, m\, v \Downarrow_{k_1+k_2+1} n'}$$

$$\frac{m \Downarrow_k n \qquad \mathsf{dms}\, n\, x}{\mathsf{app}\, m\, v \Downarrow_k \mathsf{app}\, n\, v} \qquad \frac{m \Downarrow_{k_1} \mathsf{ret}\, v \qquad n_v^0 \Downarrow_{k_2} n'}{\mathsf{seq}\, m\, n \Downarrow_{k_1+k_2+1} n'} \qquad \frac{m \Downarrow_k m' \qquad \mathsf{dms}\, m'\, x}{\mathsf{seq}\, m\, n \Downarrow_k \mathsf{seq}\, m'\, n}$$

$$\frac{m_1 \Downarrow_{k_1} \mathsf{ret}\, v_1 \qquad m_2 \Downarrow_{k_2} \mathsf{ret}\, v_2 \qquad (n_{v_1}^0)_{v_2}^1 \Downarrow_{k_3} n'}{\mathsf{pseq}\, m_2\, m_1\, n \Downarrow_{k_1+k_2+k_3+1} n'}$$

$$\frac{m_1 \Downarrow_{k_1} m_1' \qquad \mathsf{dms}\, m_1'\, x_1}{\mathsf{pseq}\, m_2\, m_1\, n \Downarrow_{k_1} \mathsf{pseq}\, m_2\, m_1'\, n} \qquad \frac{m_1 \Downarrow_{k_1} \mathsf{ret}\, v_1 \qquad m_2 \Downarrow_{k_2} m_2' \qquad \mathsf{dms}\, m_2'\, x_2}{\mathsf{pseq}\, m_2\, m_1\, n \Downarrow_{k_1+k_2} \mathsf{pseq}\, m_2'\, (\mathsf{ret}\, v_1)\, n}$$

$$\frac{n \Downarrow_{k_1} n' \qquad \mathsf{dms}\, n'\, 0 \qquad m \Downarrow_{k_2} \mathsf{ret}\, v \qquad n'^{\triangledown 0}_{\mathsf{ret}\, v} \Downarrow_{k_3} u}{\mathsf{need}\, m\, n \Downarrow_{k_1+k_2+k_3+1} u}$$

$$\frac{n \Downarrow_{k_1} n' \qquad \neg \mathsf{is\_ret}(n') \qquad \neg \mathsf{dms}\, n'\, 0}{\mathsf{need}\, m\, n \Downarrow_{k_1} \mathsf{need}\, m\, n'} \qquad \frac{n \Downarrow_{k_1} \mathsf{ret}\, v}{\mathsf{need}\, m\, n \Downarrow_{k_1} \mathsf{ret}\, v}$$

FIGURE 5.3: Big-Step Semantics of ECBPV with Time Cost

$$\frac{}{\lambda.\, m \;\Downarrow^{\|m\|+1}\; \lambda.\, m} \qquad \frac{}{\mathsf{ret}\, v \;\Downarrow^{\|v\|+1}\; \mathsf{ret}\, v}$$

$$\frac{m \;\Downarrow^{s}\; m'}{\mathsf{force}\,(\mathsf{thunk}\; m) \;\Downarrow^{\max(s,\,\|m\|+2)}\; m'} \qquad \frac{m_v^0 \;\Downarrow^{s}\; m'}{\mathsf{let}\, v.\,\mathsf{in}\, m \;\Downarrow^{\max(s,\,\|v\|+\|m\|+1)}\; m'}$$

$$\frac{m \;\Downarrow^{s_1}\; \lambda.\, n \qquad n_v^0 \;\Downarrow^{s_2}\; n'}{\mathsf{app}\, m\, v \;\Downarrow^{\max(s_1+\|v\|+1,\,s_2)}\; n'} \qquad \frac{m \;\Downarrow^{s_1}\; m' \qquad \mathsf{dms}\, m'\, x}{\mathsf{app}\, m\, v \;\Downarrow^{s_1+\|v\|+1}\; \mathsf{app}\, m'\, v}$$

$$\frac{m \;\Downarrow^{s_1}\; \mathsf{ret}\, v \qquad n_v^0 \;\Downarrow^{s_2}\; n'}{\mathsf{seq}\, m\, n \;\Downarrow^{\max(s_1+\|n\|+1,\,s_2)}\; n'} \qquad \frac{m \;\Downarrow^{s_1}\; m' \qquad \mathsf{dms}\, m'\, x}{\mathsf{seq}\, m\, n \;\Downarrow^{s_1+\|n\|+1}\; \mathsf{seq}\, m'\, n}$$

$$\frac{m_1 \;\Downarrow^{s_1}\; \mathsf{ret}\, v_1 \qquad m_2 \;\Downarrow^{s_2}\; \mathsf{ret}\, v_2 \qquad (n_{v_1}^0)_{v2}^1 \;\Downarrow^{s_3}\; n'}{\mathsf{pseq}\, m_2\, m_1\, n \;\Downarrow^{\max(s_1+\|m_2\|+\|n\|+1,\,\|v_1\|+s_2+\|n\|+1,\,s_3)}\; n'}$$

$$\frac{m_1 \;\Downarrow^{s_1}\; m_1'}{\mathsf{pseq}\, m_2\, m_1\, n \;\Downarrow^{s_1+\|m_2\|+\|n\|+1}\; \mathsf{pseq}\, m_2\, m_1'\, n}$$

$$\frac{m_1 \;\Downarrow^{s_1}\; \mathsf{ret}\, v_1 \qquad m_2 \;\Downarrow^{s_2}\; m_2'}{\mathsf{pseq}\, m_2\, m_1\, n \;\Downarrow^{\max(s_1+\|m_2\|+\|n\|+1,\,\|v_1\|+s_2+\|n\|+1)}\; \mathsf{pseq}\, m_2'\, (\mathsf{ret}\, v_1)\, n}$$

$$\frac{m \;\Downarrow^{s_1}\; \mathsf{ret}\, v \qquad n \;\Downarrow^{s_2}\; n' \qquad \mathsf{dms}\, n'\, 0 \qquad n'^{\,\triangledown 0}_{\mathsf{ret}\, v} \;\Downarrow^{s_3}\; w}{\mathsf{need}\, m\, n \;\Downarrow^{\max(1+s_1+\|n\|,\,1+\|v\|+s_2,\,s_3)}\; w}$$

$$\frac{n \;\Downarrow^{s}\; n' \qquad \neg\mathsf{is\_ret}(n') \qquad \neg\mathsf{dms}\, n'\, 0}{\mathsf{need}\, m\, n \;\Downarrow^{1+\|m\|+s}\; \mathsf{need}\, m\, n'}$$

$$\frac{n \;\Downarrow^{s}\; \mathsf{ret}\, v}{\mathsf{need}\, m\, n \;\Downarrow^{\max(1+\|m\|+1+\|n\|,\,1+\|m\|+s)}\; \mathsf{ret}\, v}$$

FIGURE 5.4: Big-Step Semantics of ECBPV with Space Cost

As you can see, there are a few more rules in ECBPV comparing to the ones in CBPV. The additional rules that corresponds to the existing CBPV constructors are to deal with the situation where one of the underlying computations (e.g. m) has a needvar that is pending evaluation, thus it stops midway through reduction (at e.g. m') and can not be reduced to the normal form (e.g. $\mathsf{ret}\, v$). In this case, we simply update the computation m in the term with m'.

The ones for need and needvar are the core ones for ECBPV. All three of them form a decision chain for reducing need terms, for example, for need $m\ n$:

1. If n's reduction gets blocked at n' before reaching normal form because m is needed, evaluate m and substitute the result in n';
2. If n's reduction gets blocked at n' before reaching normal form because of other needvar, update n with n';
3. If n can be reduced to normal form ret n' successfully, simply return that.

Chapter 6

Conclusions and Future Directions

This thesis contributed to the development and formal verification of reasonable cost models for functional languages within the λ-calculus family, with a particular focus on CBPV. It presented three closely related projects that together advance the formal study of complexity of functional programming.

6.1 Summary of Work

The first project formalised the existing reasonable time and space cost models for WCBV. The original work formalised their models in the Rocq prover, while this thesis formalises them in the HOL4 prover. This further verified their correctness and consistency by performing formalisation in a prover with a different logical foundation. This work is also included in the HOL4 library, enabling future developments to build upon it. This also provided a foundation for subsequent developments in this thesis by ensuring a mechanised understanding of how existing cost models with natural measure behave.

The second project, which forms the core of this thesis, introduced and verified the first reasonable time and space cost models for CBPV. This project provided both theoretical and formal contributions. The theoretical work established that CBPV satisfies the invariance thesis for both time and space. The formalisation component mechanised the core parts of this proof: the CBPV language model, the machine models for the intermediate abstract machines (the substitution and heap machines), and the simulations between CBPV and these machines. As part of this proof, we demonstrated that Levy's call-by-value embedding into CBPV preserves reasonable time and space bounds.

The third project extended the theoretical framework by developing and formalising ECBPV that supports memory sharing. Although no cost model was proposed for ECBPV in this thesis, the mechanised language model lays

the groundwork for future exploration of cost models for λ-calculus with memory sharing.

These projects collectively establish a systematic framework connecting language semantics, abstract machines, and complexity. In particular, the interleaving of the substitution and heap machines preserves desired time and space bounds with respect to CBPV and Turing machines, extending known results for WCBV. The resulting cost-aware simulations provide the first formal proof that CBPV constitutes a reasonable model of computation. Following this, the development of ECBPV further extends the landscape of λ-calculus that can be studied under this framework.

6.2 Main Contributions

The central achievement of this thesis is the development and proof of reasonable time and space cost models for CBPV, establishing CBPV as a reasonable model of computation in the sense of the invariance thesis.

CBPV has become increasingly influential in the study of λ-calculus and the theory of computation. Showing that CBPV is reasonable provides a crucial step toward understanding its computational complexity and solidifies its role as a unifying model between call-by-value and call-by-name paradigms.

The proposed cost models adopt natural and intuitive measures: time is quantified by the number of reduction steps rather than concrete execution time, and space is measured as the size of the largest intermediate term rather than total accumulated memory usage. These definitions make the models conceptually clear and theoretically tractable.

Because CBPV enables explicit control of evaluation order within the language itself, the results of this thesis support future research into the cost and complexity of other λ-calculus variants, with CBPV serving as a unifying intermediate model.

The formalisation work complements and strengthens the theoretical results by providing mechanised, machine-checked proofs in a theorem prover.

The mechanisation process revealed and corrected an issue in an earlier version of the theoretical CBPV model I developed, leading to a more accurate formulation. The formalisation confirms the soundness of the time and space cost models and provides certified simulations between CBPV and its abstract machines. The mechanised models are reusable: researchers can build upon these machine-checked components without re-verifying their

correctness, thereby enabling further development of cost models and verified compilers.

By combining theoretical innovation with formal verification, this thesis advances both the conceptual and practical understanding of computational cost in functional programming languages. It establishes CBPV as a solid foundation for reasoning about time and space complexity, linking the abstract semantics of functional languages to concrete machine-based complexity measures.

Beyond CBPV, this work contributes to the broader effort of integrating complexity theory into mechanised reasoning. The verified models and simulations developed here can serve as a reference for future work on reasoning about program efficiency in proof assistants, and for studying other λ-calculus variants under a unified framework.

6.3 Limitations and Future Work

While this thesis establishes the first reasonable cost models for CBPV and formally verifies their correctness, several limitations remain open for future research.

Sublinear Time and Space While the verified cost model provided in this thesis enables CBPV to be used to define standard complexity classes like P, PSPACE, however, it does not capture sublinear time or space behaviour, leaving room for further investigation into more refined space analyses. The environment in the heap machine is a linked list, thus caps the time at constant time at best. Improvement can be done by developing a more efficient heap structure and corresponding lookup function. As for space, it is a well-known problem for λ-calculus complexity analysis, that is the size of the input term itself is counted towards its space complexity. This is because in λ-calculus, there is no separation between input space and work space. As discussed in the related work section (Section 2.4), the space KAM from (Schöpp, 2007) is not directly applicable to the CBPV if we want to maintain natural measures, however, it provides a potential direction to look into: intermediate representations that separates input space and work space.

Extend to Call-By-Name In this thesis, I demonstrated that the framework can extend naturally to (weak) call-by-value. The extension to call-by-name still remains open. From my observation, Levy's encoding of call-by-name

from CBPV does not lie within the desired cost bounds. While more investigation is required to determine whether his encoding is indeed not reasonable under any circumstances, it will be interesting to look into whether there exist other encodings that are natural and reasonable.

Alternative Simulation Strategies Another limitation concerns the interleaving algorithm between the substitution and heap machines. Although the current design is effective and formally verified, it is not intuitively straightforward; a simplified or more general formulation could improve both understanding and extensibility. Selecting a different model of computation as the baseline model instead of Turing machines could also potentially result in a more intuitive simulation.

Extension with Memory Sharing The cost model for CBPV does not directly accommodate λ-calculus with evaluation strategies involving memory sharing, such as call-by-need. For such extensions to work, we have to strengthen CBPV first. The ECBPV developed in this thesis (Chapter 5) provides a promising foundation for addressing this limitation and developing such extensions. To develop reasonable cost models for ECBPV, intermediate machine(s) with environment is essential. The CBPV heap machine can be used as the starting point. There also exist improvements that could be done for ECBPV. For one, the implementation of the demand function has linear time complexity, which means that the usage of this function will incur a big time cost. This should be taken into account during the cost model development. Another improvement is about space cost. Currently, there is no rule in the ECBPV semantics to ever get rid of a need binding. Implementation of efficient garbage collection for need bindings can help reduce the unnecessary space cost. These improvements will ultimately contribute to the future development of ECBPV cost models.

Untyped v.s. Typed The present work focuses on an untyped subset of CBPV. Extending the cost model and its correctness proofs to typed variants or full variants would strengthen its theoretical guarantees and enhance its relevance to real-world functional programming languages. The gap remaining is adding a few missing constructs from Levy's original presentation: sums, products, and a fixpoint combinator.

6.4 Concluding Remarks

In conclusion, this thesis establishes a mechanised and theoretically sound foundation for reasoning about the time and space complexity of CBPV. By showing that CBPV is a reasonable model of computation and providing reusable mechanised artefacts, it bridges the gap between abstract complexity theory and formal verification. These results open the way toward a unified, mechanised theory of computational cost for functional programming languages, and provide a basis for future exploration of cost models in richer and more realistic language settings.

Bibliography

Accattoli, Beniamino (2017). "(In)Efficiency and Reasonable Cost Models". In: *12th Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2017, Brasília, Brazil, September 23-24, 2017*. Ed. by Sandra Alves and Renata Wasserman. Vol. 338. Electronic Notes in Theoretical Computer Science. Elsevier, pp. 23–43. DOI: 10.1016/j.entcs.2018.10.003. URL: https://doi.org/10.1016/j.entcs.2018.10.003.

Accattoli, Beniamino, Andrea Condoluci, and Claudio Sacerdoti Coen (2021). "Strong Call-by-Value is Reasonable, Implosively". In: *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*. IEEE, pp. 1–14. DOI: 10.1109/LICS52264.2021.9470630. URL: https://doi.org/10.1109/LICS52264.2021.9470630.

Accattoli, Beniamino and Ugo Dal Lago (2016). "(Leftmost-Outermost) Beta Reduction is Invariant, Indeed". In: *Log. Methods Comput. Sci.* 12.1. DOI: 10.2168/LMCS-12(1:4)2016. URL: https://doi.org/10.2168/LMCS-12(1:4)2016.

Accattoli, Beniamino, Ugo Dal Lago, and Gabriele Vanoni (2022). "Reasonable Space for the λ-Calculus, Logarithmically". In: *LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022*. Ed. by Christel Baier and Dana Fisman. ACM, 47:1–47:13. DOI: 10.1145/3531130.3533362. URL: https://doi.org/10.1145/3531130.3533362.

Asperti, Andrea and Harry G. Mairson (1998). "Parallel Beta Reduction is not Elementary Recursive". In: *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. Ed. by David B. MacQueen and Luca Cardelli. ACM, pp. 303–315. DOI: 10.1145/268946.268971. URL: https://doi.org/10.1145/268946.268971.

Blelloch, Guy E. and John Greiner (1995). "Parallelism in Sequential Functional Languages". In: *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*. Ed. by John Williams. ACM,

pp. 226–237. DOI: 10.1145/224164.224210. URL: https://doi.org/10.1145/224164.224210.

Boyer, R.S., M. Kaufmann, and J.S. Moore (1995). "The Boyer-Moore theorem prover and its interactive enhancement". In: *Computers & Mathematics with Applications* 29.2, pp. 27–62. ISSN: 0898-1221. DOI: 10.1016/0898-1221(94)00215-7. URL: https://www.sciencedirect.com/science/article/pii/0898122194002157.

Chen, Zhuo (Mar. 2022a). *HOL-wcbv-reasonable*. URL: https://github.com/ZhuoZoeyChen/HOL-cbvlcm.

— (Mar. 2022b). *HOL-wcbv-reasonable*. URL: https://github.com/ZhuoZoeyChen/HOL-wcbv-reasonable.

— (June 2025). *cbpv-reasonable-HOL*. URL: https://github.com/ZhuoZoeyChen/cbpv-reasonable-HOL.

— (Jan. 2026). *ecbpv-reasonable-HOL*. URL: https://github.com/ZhuoZoeyChen/ecbpv-reasonable-HOL.

Chen, Zhuo Zoey, Johannes Åman Pohjola, and Christine Rizkallah (2025). "A Verified Cost Model for Call-By-Push-Value". In: *16th International Conference on Interactive Theorem Proving, ITP 2025, September 28 to October 1, 2025, Reykjavik, Iceland*. Ed. by Yannick Forster and Chantal Keller. Vol. 352. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:19. DOI: 10.4230/LIPICS.ITP.2025.7. URL: https://doi.org/10.4230/LIPIcs.ITP.2025.7.

Chouquet, Jules and Christine Tasson (2020). "Taylor expansion for Call-By-Push-Value". In: *28th EACSL Annual Conference on Computer Science Logic, CSL 2020, January 13-16, 2020, Barcelona, Spain*. Ed. by Maribel Fernández and Anca Muscholl. Vol. 152. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 16:1–16:16. DOI: 10.4230/LIPICS.CSL.2020.16. URL: https://doi.org/10.4230/LIPIcs.CSL.2020.16.

Church, Alonzo (1932). "A Set of Postulates for the Foundation of Logic". In: *Annals of Mathematics* 33.2, pp. 346–366. ISSN: 0003486X. URL: http://www.jstor.org/stable/1968337.

Clarke, E. M., E. A. Emerson, and A. P. Sistla (Apr. 1986). "Automatic verification of finite-state concurrent systems using temporal logic specifications". In: *ACM Trans. Program. Lang. Syst.* 8.2, 244–263. ISSN: 0164-0925. DOI: 10.1145/5397.5399. URL: https://doi.org/10.1145/5397.5399.

Clarke, Edmund M. and E. Allen Emerson (1981). "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic". In: *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*.

Ed. by Dexter Kozen. Vol. 131. Lecture Notes in Computer Science. Springer, pp. 52–71. DOI: 10.1007/BFB0025774. URL: https://doi.org/10.1007/BFb0025774.

Clarke, Edmund M., Orna Grumberg, and Doron A. Peled (2001). *Model checking, 1st Edition*. MIT Press. ISBN: 978-0-262-03270-4. URL: http://books.google.de/books?id=Nmc4wEaLXFEC.

Cook, Stephen A. and Robert A. Reckhow (1973). "Time Bounded Random Access Machines". In: *J. Comput. Syst. Sci.* 7.4, pp. 354–375. DOI: 10.1016/S0022-0000(73)80029-7. URL: https://doi.org/10.1016/S0022-0000(73)80029-7.

Cousot, Patrick and Radhia Cousot (1977). "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. Ed. by Robert M. Graham, Michael A. Harrison, and Ravi Sethi. ACM, pp. 238–252. DOI: 10.1145/512950.512973. URL: https://doi.org/10.1145/512950.512973.

CVE-2023-23583 Intel Processor Vulnerability in NetApp Products (2023). URL: https://security.netapp.com/advisory/ntap-20231116-0015/.

Dal Lago, Ugo and Beniamino Accattoli (2017). "Encoding Turing Machines into the Deterministic Lambda-Calculus". In: *CoRR* abs/1711.10078. arXiv: 1711.10078. URL: http://arxiv.org/abs/1711.10078.

Dal Lago, Ugo and Simone Martini (2008). "The Weak Lambda Calculus as a Reasonable Machine". In: *Theor. Comput. Sci.* 398.1-3, pp. 32–50. DOI: 10.1016/j.tcs.2008.01.044. URL: https://doi.org/10.1016/j.tcs.2008.01.044.

— (2012). "On Constructor Rewrite Systems and the Lambda Calculus". In: *Log. Methods Comput. Sci.* 8.3. DOI: 10.2168/LMCS-8(3:12)2012. URL: https://doi.org/10.2168/LMCS-8(3:12)2012.

Dal Lago, Ugo and Ulrich Schöpp (2016). "Computation by interaction for space-bounded functional programming". In: *Inf. Comput.* 248, pp. 150–194. DOI: 10.1016/J.IC.2015.04.006. URL: https://doi.org/10.1016/j.ic.2015.04.006.

Davis, M. (1957). "A computer program for Presburger's algorithm". In: *Proving Theorems (as Done by Man, Logician, or Machine)*. Proc. Summer Institute for Symbolic Logic. Second edition; publication date is 1960.

Davis, Martin (1982). "Why Gödel Didn't Have Church's Thesis". In: *Inf. Control.* 54.1/2, pp. 3–24. DOI: 10.1016/S0019-9958(82)91226-8. URL: https://doi.org/10.1016/S0019-9958(82)91226-8.

Davis, Martin and Hilary Putnam (1960). "A Computing Procedure for Quantification Theory". In: *J. ACM* 7.3, pp. 201–215. DOI: 10.1145/321033.321034. URL: https://doi.org/10.1145/321033.321034.

Davis, Martin D. (1958). *Computability and Unsolvability*. McGraw-Hill Series in Information Processing and Computers. McGraw-Hill.

de Bruijn, N. G. (1983). "AUTOMATH, a Language for Mathematics". In: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Ed. by Jörg H. Siekmann and Graham Wrightson. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 159–200. ISBN: 978-3-642-81955-1. DOI: 10.1007/978-3-642-81955-1_11.

de Bruijn, N.G (1972). "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)* 75.5, pp. 381–392. ISSN: 1385-7258. DOI: https://doi.org/10.1016/1385-7258(72)90034-0. URL: https://www.sciencedirect.com/science/article/pii/1385725872900340.

Ehrhard, Thomas and Giulio Guerrieri (2016). "The Bang Calculus: an Untyped Lambda-Calculus Generalizing Call-by-Name and Call-by-Value". In: *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*. Ed. by James Cheney and Germán Vidal. ACM, pp. 174–187. DOI: 10.1145/2967973.2968608. URL: https://doi.org/10.1145/2967973.2968608.

Ehrhard, Thomas and Christine Tasson (2016). "Probabilistic Call by Push Value". In: *CoRR* abs/1607.04690. arXiv: 1607.04690. URL: http://arxiv.org/abs/1607.04690.

Forster, Yannick, Fabian Kunze, and Marc Roth (2020). "The Weak Call-by-Value λ-Calculus is Reasonable for both Time and Space". In: *Proc. ACM Program. Lang.* 4.POPL, 27:1–27:23. DOI: 10.1145/3371095. URL: https://doi.org/10.1145/3371095.

Forster, Yannick et al. (2019). "Call-by-Push-Value in Coq: Operational, Equational, and Denotational Theory". In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*. Ed. by Assia Mahboubi and Magnus

O. Myreen. ACM, pp. 118–131. DOI: 10.1145/3293880.3294097. URL: https://doi.org/10.1145/3293880.3294097.

Forster, Yannick et al. (2021). "A Mechanised Proof of the Time Invariance Thesis for the Weak Call-By-Value λ-Calculus". In: *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*. Ed. by Liron Cohen and Cezary Kaliszyk. Vol. 193. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 19:1–19:20. DOI: 10.4230/LIPIcs.ITP.2021.19. URL: https://doi.org/10.4230/LIPIcs.ITP.2021.19.

Frandsen, Gudmund Skovbjerg and Carl Sturtivant (1991). "What is an Efficient Implementation of the \lambda-calculus?" In: *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*. Ed. by John Hughes. Vol. 523. Lecture Notes in Computer Science. Springer, pp. 289–312. DOI: 10.1007/3540543961_14. URL: https://doi.org/10.1007/3540543961_14.

Garbuzov, Dmitri et al. (2018). "Structural Operational Semantics for Control Flow Graph Machines". In: *CoRR* abs/1805.05400. arXiv: 1805.05400. URL: http://arxiv.org/abs/1805.05400.

Ghica, Dan R. (2007). "Geometry of synthesis: a structured approach to VLSI design". In: *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*. Ed. by Martin Hofmann and Matthias Felleisen. ACM, pp. 363–375. DOI: 10.1145/1190216.1190269. URL: https://doi.org/10.1145/1190216.1190269.

Gilmore, Paul C. (1960). "A Proof Method for Quantification Theory: Its Justification and Realization". In: *IBM J. Res. Dev.* 4.1, pp. 28–35. DOI: 10.1147/RD.41.0028. URL: https://doi.org/10.1147/rd.41.0028.

Goncharov, Sergey, Stelios Tsampas, and Henning Urbat (2025). "Abstract Operational Methods for Call-by-Push-Value". In: *Proc. ACM Program. Lang.* 9.POPL, pp. 1013–1039. DOI: 10.1145/3704871. URL: https://doi.org/10.1145/3704871.

Gordon, Michael J. C., Robin Milner, and Christopher P. Wadsworth (1979). *Edinburgh LCF*. Vol. 78. Lecture Notes in Computer Science. Springer. ISBN: 3-540-09724-4. DOI: 10.1007/3-540-09724-4. URL: https://doi.org/10.1007/3-540-09724-4.

Gordon, Michael John Caldwell and Thomas F. Melham (1993). *Introduction to HOL, a theorem proving environment for higher order logic*. Cambridge University Press.

Gordon, Mike (2000). "From LCF to HOL: a short history". In: *Proof, Language, and Interaction, Essays in Honour of Robin Milner*. Ed. by Gordon D. Plotkin, Colin Stirling, and Mads Tofte. The MIT Press, pp. 169–186.

Hales, Thomas C. (2005). "A proof of the Kepler conjecture". In: *Annals of Mathematics* 162.3, pp. 1065–1185. DOI: 10.4007/annals.2005.162.1065.

Hales, Thomas C. et al. (2015). "A formal proof of the Kepler conjecture". In: *CoRR* abs/1501.02155. arXiv: 1501.02155. URL: http://arxiv.org/abs/1501.02155.

Harrison, John (2001). *The LCF Approach to Theorem Proving (Slides)*. Presented at the University of Manchester, 12 September 2001. URL: https://www.cl.cam.ac.uk/~jrh13/slides/manchester-12sep01/slides.pdf.

— (2006). "Towards self-verification of HOL light". In: *Proceedings of the Third International Joint Conference on Automated Reasoning*. IJCAR'06. Seattle, WA: Springer-Verlag, 177–191. ISBN: 3540371877. DOI: 10.1007/11814771_17. URL: https://doi.org/10.1007/11814771_17.

Harrison, John, Josef Urban, and Freek Wiedijk (2014). "History of Interactive Theorem Proving". In: *Computational Logic*. Ed. by Jörg H. Siekmann. Vol. 9. Handbook of the History of Logic. Elsevier, pp. 135–214. DOI: 10.1016/B978-0-444-51624-4.50004-6. URL: https://doi.org/10.1016/B978-0-444-51624-4.50004-6.

Hartmanis, Juris (1981). "Observations About the Development of Theoretical Computer Science". In: *IEEE Ann. Hist. Comput.* 3.1, pp. 42–51. DOI: 10.1109/MAHC.1981.10005. URL: https://doi.org/10.1109/MAHC.1981.10005.

Hartmanis, Juris and Richard Edwin Stearns (1965). "On the Computational Complexity of Algorithms". In: *Transactions of the American Mathematical Society* 117, pp. 285–306. URL: https://api.semanticscholar.org/CorpusID:40185800.

HOL (June 2022). *HOL-example-wcbv-reasonable*. URL: https://github.com/HOL-Theorem-Prover/HOL/tree/develop/examples/lambda/wcbv-reasonable.

HOL (2025). *HOL Website*. URL: https://hol-theorem-prover.org/#doc (visited on 08/12/2025).

Hudak, Paul et al. (2007). "A history of Haskell: being lazy with class". In: *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*. Ed. by Barbara G. Ryder and Brent Hailpern. ACM, pp. 1–55. DOI: 10.1145/1238844.1238856. URL: https://doi.org/10.1145/1238844.1238856.

Independent Review of NATS (EnRoute) Plc's Flight Planning System Failure (Final Report) (2024). URL: https://www.caa.co.uk/publication/download/23337 (visited on 05/31/2024).

Johnson, S.C. (1977). *Lint, a C Program Checker*. Computing science technical report. Bell Laboratories. URL: https://books.google.com.au/books?id=b8nWGwAACAAJ.

Kavvos, G. A. et al. (2020). "Recurrence Extraction for Functional Programs through Call-by-Push-Value". In: *Proc. ACM Program. Lang.* 4.POPL, 15:1–15:31. DOI: 10.1145/3371083. URL: https://doi.org/10.1145/3371083.

Kesner, Delia and Andrés Viso (2021). "The Power of Tightness for Call-By-Push-Value". In: *CoRR* abs/2105.00564. arXiv: 2105.00564. URL: https://arxiv.org/abs/2105.00564.

Kleene, Stephen Cole (1981). "Origins of Recursive Function Theory". In: *IEEE Ann. Hist. Comput.* 3.1, pp. 52–67. DOI: 10.1109/MAHC.1981.10004. URL: https://doi.org/10.1109/MAHC.1981.10004.

Krivine, Jean-Louis (2007). "A call-by-name lambda-calculus machine". In: *Higher-Order and Symbolic Computation* 20.3, pp. 199–207. DOI: 10.1007/s10990-007-9016-y. URL: https://doi.org/10.1007/s10990-007-9016-y.

Kunze, Fabian, Gert Smolka, and Yannick Forster (2018). "Formal Small-Step Verification of a Call-by-Value Lambda Calculus Machine". In: *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*. Ed. by Sukyoung Ryu. Vol. 11275. Lecture Notes in Computer Science. Springer, pp. 264–283. DOI: 10.1007/978-3-030-02768-1_15. URL: https://doi.org/10.1007/978-3-030-02768-1_15.

Lagarias, Jeffrey C. (2011). "The Kepler Conjecture and Its Proof". In: *The Kepler Conjecture: The Hales-Ferguson Proof*. Ed. by Jeffrey C. Lagarias. New York, NY: Springer New York, pp. 3–26. ISBN: 978-1-4614-1129-1. DOI: 10.1007/978-1-4614-1129-1_1. URL: https://doi.org/10.1007/978-1-4614-1129-1_1.

Lawall, Julia L. and Harry G. Mairson (1996). "Optimality and Inefficiency: What Isn't a Cost Model of the Lambda Calculus?" In: *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, USA, May 24-26, 1996*. Ed. by Robert Harper and Richard L. Wexelblat. ACM, pp. 92–101. DOI: 10.1145/232627.232639. URL: https://doi.org/10.1145/232627.232639.

Leveson, Nancy G. and Clark Savage Turner (1993). "Investigation of the Therac-25 Accidents". In: *Computer* 26.7, pp. 18–41. DOI: 10.1109/MC.1993.274940. URL: https://doi.org/10.1109/MC.1993.274940.

Levy, Paul Blain (1999). "Call-by-Push-Value: A Subsuming Paradigm". In: *Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings*. Ed. by Jean-Yves Girard. Vol. 1581. Lecture Notes in Computer Science. Springer, pp. 228–242. DOI: 10.1007/3-540-48959-2_17. URL: https://doi.org/10.1007/3-540-48959-2_17.

— (2001). "Call-by-Push-Value". PhD thesis. Queen Mary University of London, UK. URL: https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.369233.

— (2002). "Adjunction Models For Call-By-Push-Value With Stacks". In: *Category Theory and Computer Science, CTCS 2002, Ottawa, Canada, August 15-17, 2002*. Ed. by Richard Blute and Peter Selinger. Vol. 69. Electronic Notes in Theoretical Computer Science. Elsevier, pp. 248–271. DOI: 10.1016/S1571-0661(04)80568-1. URL: https://doi.org/10.1016/S1571-0661(04)80568-1.

Loveland, Donald W. (1968). "Mechanical Theorem-Proving by Model Elimination". In: *J. ACM* 15.2, pp. 236–251. DOI: 10.1145/321450.321456. URL: https://doi.org/10.1145/321450.321456.

Mazza, Damiano (2015). "Simple Parsimonious Types and Logarithmic Space". In: *24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7-10, 2015, Berlin, Germany*. Ed. by Stephan Kreutzer. Vol. 41. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 24–40. DOI: 10.4230/LIPICS.CSL.2015.24. URL: https://doi.org/10.4230/LIPIcs.CSL.2015.24.

McDermott, Dylan (2025). "Grading Call-By-Push-Value, Explicitly and Implicitly". In: *10th International Conference on Formal Structures for Computation and Deduction, FSCD 2025, July 14-20, 2025, Birmingham, UK*. Ed. by Maribel Fernández. Vol. 337. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 28:1–28:19. DOI: 10.4230/LIPICS.FSCD.2025.28. URL: https://doi.org/10.4230/LIPIcs.FSCD.2025.28.

McDermott, Dylan and Alan Mycroft (2019). "Extended Call-by-Push-Value: Reasoning About Effectful Programs and Evaluation Order". In: *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*. Ed. by Luís Caires. Vol. 11423. Lecture Notes in Computer

Science. Springer, pp. 235–262. DOI: 10.1007/978-3-030-17184-1_9. URL: https://doi.org/10.1007/978-3-030-17184-1_9.

Milner, Robin (1972). *Logic for Computable Functions: description of a machine implementation*. Tech. rep. Stanford, CA, USA.

Milner, Robin, Mads Tofte, and Robert Harper (1990). *Definition of standard ML*. MIT Press. ISBN: 978-0-262-63132-7.

Minsky, Marvin L. (1967). *Computation: finite and infinite machines*. USA: Prentice-Hall, Inc. ISBN: 0131655639.

Moura, Leonardo de and Sebastian Ullrich (2021). "The Lean 4 Theorem Prover and Programming Language". In: *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*. Ed. by André Platzer and Geoff Sutcliffe. Vol. 12699. Lecture Notes in Computer Science. Springer, pp. 625–635. DOI: 10.1007/978-3-030-79876-5_37. URL: https://doi.org/10.1007/978-3-030-79876-5_37.

Moura, Leonardo Mendonça de et al. (2015). "The Lean Theorem Prover (System Description)". In: *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*. Ed. by Amy P. Felty and Aart Middeldorp. Vol. 9195. Lecture Notes in Computer Science. Springer, pp. 378–388. DOI: 10.1007/978-3-319-21401-6_26. URL: https://doi.org/10.1007/978-3-319-21401-6_26.

Nipkow, Tobias and Gerwin Klein (2014). *Concrete Semantics - With Isabelle/HOL*. Springer. ISBN: 978-3-319-10541-3. DOI: 10.1007/978-3-319-10542-0. URL: https://doi.org/10.1007/978-3-319-10542-0.

Nipkow, Tobias, Lawrence C. Paulson, and Markus Wenzel (2002). *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer. ISBN: 3-540-43376-7. DOI: 10.1007/3-540-45949-9. URL: https://doi.org/10.1007/3-540-45949-9.

Paulson, Lawrence C. (1986). "Natural Deduction as Higher-Order Resolution". In: *J. Log. Program.* 3.3, pp. 237–258. DOI: 10.1016/0743-1066(86)90015-4. URL: https://doi.org/10.1016/0743-1066(86)90015-4.

Plotkin, G.D. (1975). "Call-by-name, call-by-value and the lambda-calculus". In: *Theoretical Computer Science* 1.2, pp. 125–159. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(75)90017-1. URL: https://www.sciencedirect.com/science/article/pii/0304397575900171.

Prawitz, Dag, Håkan Prawitz, and Neri Voghera (1960). "A Mechanical Proof Procedure and its Realization in an Electronic Computer". In: *J. ACM* 7.2,

pp. 102–128. DOI: 10.1145/321021.321023. URL: https://doi.org/10.1145/321021.321023.

Queille, Jean-Pierre and Joseph Sifakis (1982). "Specification and verification of concurrent systems in CESAR". In: *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*. Ed. by Mariangiola Dezani-Ciancaglini and Ugo Montanari. Vol. 137. Lecture Notes in Computer Science. Springer, pp. 337–351. DOI: 10.1007/3-540-11494-7_22. URL: https://doi.org/10.1007/3-540-11494-7_22.

Rizkallah, Christine, Dmitri Garbuzov, and Steve Zdancewic (2018). "A Formal Equational Theory for Call-By-Push-Value". In: *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*. Ed. by Jeremy Avigad and Assia Mahboubi. Vol. 10895. Lecture Notes in Computer Science. Springer, pp. 523–541. DOI: 10.1007/978-3-319-94821-8_31. URL: https://doi.org/10.1007/978-3-319-94821-8_31.

Robinson, John Alan (1965). "A Machine-Oriented Logic Based on the Resolution Principle". In: *J. ACM* 12.1, pp. 23–41. DOI: 10.1145/321250.321253. URL: https://doi.org/10.1145/321250.321253.

Rocq Proof Assistant (2025). URL: https://rocq-prover.org.

Sands, David, Jörgen Gustavsson, and Andrew Moran (2002). "Lambda Calculi and Linear Speedups". In: *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th birthday]*. Ed. by Torben Æ. Mogensen, David A. Schmidt, and Ivan Hal Sudborough. Vol. 2566. Lecture Notes in Computer Science. Springer, pp. 60–84. DOI: 10.1007/3-540-36377-7_4. URL: https://doi.org/10.1007/3-540-36377-7_4.

Schöpp, Ulrich (2007). "Stratified Bounded Affine Logic for Logarithmic Space". In: *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*. IEEE Computer Society, pp. 411–420. DOI: 10.1109/LICS.2007.45. URL: https://doi.org/10.1109/LICS.2007.45.

Sipser, Michael (2013). *Introduction to the Theory of Computation*. Boston, MA: Cengage Learning.

Slind, Konrad and Michael Norrish (2008). "A Brief Overview of HOL4". In: *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*. Ed. by Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar. Vol. 5170. Lecture Notes in Computer Science. Springer, pp. 28–32. DOI: 10.1007/978-

3-540-71067-7_6. URL: https://doi.org/10.1007/978-3-540-71067-7_6.

Slot, Cees F. and Peter van Emde Boas (1984). "On Tape Versus Core; An Application of Space Efficient Perfect Hash Functions to the Invariance of Space". In: *Proceedings of the 16th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1984, Washington, DC, USA*. Ed. by Richard A. DeMillo. ACM, pp. 391–400. DOI: 10.1145/800057.808705. URL: https://doi.org/10.1145/800057.808705.

Sozeau, Matthieu et al. (2020). "The MetaCoq Project". In: *J. Autom. Reason.* 64.5, pp. 947–999. DOI: 10.1007/S10817-019-09540-0. URL: https://doi.org/10.1007/s10817-019-09540-0.

Torczon, Cassia et al. (2023). "Effects and Coeffects in Call-By-Push-Value (Extended Version)". In: *CoRR* abs/2311.11795. DOI: 10.48550/ARXIV.2311.11795. arXiv: 2311.11795. URL: https://doi.org/10.48550/arXiv.2311.11795.

— (2024). "Effects and Coeffects in Call-by-Push-Value". In: *Proc. ACM Program. Lang.* 8.OOPSLA2, pp. 1108–1134. DOI: 10.1145/3689750. URL: https://doi.org/10.1145/3689750.

Turing, Alan Mathison (1936). "On computable numbers, with an application to the Entscheidungsproblem". In: *J. of Math* 58.345-363, p. 5.

van Emde Boas, Peter (1990). "CHAPTER 1 - Machine Models and Simulations". In: *Algorithms and Complexity*. Ed. by Jan Van Leeuwen. Handbook of Theoretical Computer Science. Amsterdam: Elsevier, pp. 1–66. ISBN: 978-0-444-88071-0. DOI: https://doi.org/10.1016/B978-0-444-88071-0.50006-0. URL: https://www.sciencedirect.com/science/article/pii/B9780444880710500060.

Wang, Hao (1960). "Toward Mechanical Mathematics". In: *IBM J. Res. Dev.* 4.1, pp. 2–22. DOI: 10.1147/RD.41.0002. URL: https://doi.org/10.1147/rd.41.0002.

Wichmann, Brian A. et al. (1995). "Industrial perspective on static analysis". In: *Softw. Eng. J.* 10.2, pp. 69–75. DOI: 10.1049/SEJ.1995.0010. URL: https://doi.org/10.1049/sej.1995.0010.