On the Evolution of Genotype-Phenotype Mapping: Exploring Viability in the Avida Artificial Life System

Tomonori Hasegawa, B.Eng.
School of Electronic Engineering, Dublin City University

Supervisor: Prof. Barry McMullin

A thesis submitted for the degree of Ph.D.

January 2015

Declaration.

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Ph.D., is entirely my own work, and that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.
Signed: ____________________________ (Candidate), ID No.: 10117253, Date: ___________

Dedication.

To my mother, to my father, and to my mentor in life, who are always watching over me.

And to the one who opened my mind's I.

Acknowledgments.

The research on this thesis was supported by the European Complexity Network (Complexity-NET) through the Irish Research Council for Science and Technology (IRCSET) under the collaborative project EvoSym, over the course from 2010 to 2013.

First and foremost, I would like to appreciate Prof. Barry McMullin's invaluable and patient supervision on me throughout the years of my Ph.D. research, and Declan Baugh's comradeship as an indispensable colleague of the Artificial Life Laboratory in Dublin City University.

My sincere appreciation is likewise extended to all these professors involved in my Ph.D. examination: Prof. Susan Stepney for being an external viva examiner; and Dr Noel Murphy for being an internal viva examiner and for crucial suggestions and comments on my transfer; and Dr Ronan Scaife for being a viva chairperson.

I hereby express my appreciation to all the Avida founders and developers as well, for releasing such a profound platform which became my research topic/material. I would also like to appreciate all the EvoSym project partners, Bioinformatics Group in Universiteit Utrecht, and Artificial Intelligence Laboratory in Vrije Universiteit Brussel, for great inspiration from a much wider perspective.

On a more personal note, I am truly grateful for, and indebted to, my whole beloved family, especially my two sisters for their long-standing and outstanding support; I also wholeheartedly appreciate constant encouragement and compassionate support from my dear friends met in Japan and in Ireland (especially D. for limitless optimism, and S. and D. for valuable comments to the thesis in embryo); and I deeply thank the faculty of my alma mater, especially Prof. Tatsuo Unemi for guiding me into the field of Artificial Life.

Contents

1 Introduction 1

1.1 Thesis Statement . . . 1

1.2 Motivation and Objectives . . . 1

1.3 Course of Investigation . . . 1

1.4 Contributions . . . 2

1.5 Thesis Outline . . . 2

1.6 Related Publications . . . 3

2 Background and Related Work 5

2.1 Overview . . . 5

2.2 Machine Self-Reproduction and Evolution . . . 5

2.2.1 Architecture of Machine Self-Reproduction . . . 5

2.2.2 Underlying Influences . . . 9

2.2.3 Artificial Life Investigations . . . 11

2.3 Biological Symbol Systems . . . 16

2.3.1 Evolution of Symbol Systems . . . 16

2.3.2 Self-Referential Logic . . . 18

2.4 Closing Remark . . . 21

3 Implementation of a von Neumann Style Self-Reproducer 22

3.1 Overview . . . 22

3.2 The *Avida* System . . . 22

3.2.1 *Avida* Organism Architecture . . . 23

3.2.2 Reproduction Mode . . . 26

3.2.3 *Avida* as a *Coreworld*-type System . . . 30

3.3 Designing a Prototype Ancestor . . . 38

3.3.1 *Genome* and *Phenome* . . . 38

3.3.2 Copying and Decoding . . . 39

3.3.3 Instruction Set . . . 40

3.3.4 Lookup Table . . . 44

3.3.5 Self-Reproduction . . . 46

3.4 Observing the Prototype Behaviour . . . 61

3.4.1 Evolutionary Experiment using *Avida* . . . 62

3.4.2 Observation and Examination . . . 65

3.4.3 Degeneration of the Prototype: the Mechanism . . . 69

3.5 Reflective Remarks . . . 76

3.5.1 Towards Identification of Reproduction Mode . . . 76

3.5.2 Reconsidering the Design of Ancestor 77
3.6 Closing Remark 79

4 Automated Mutational Analysis 80
4.1 Overview 80
4.2 Research Direction and Expositions 80
4.2.1 Exposition 1: Self-Reproduction Revisited 81
4.2.2 Exposition 2: Viability and Viability Analysis 88
4.3 Avida *Analyze Mode* 92
4.3.1 *Analyze Mode* and the `TRACE` command 92
4.3.2 *Analyze Mode*'s Analysis of First-step Mutants 94
4.3.3 *Analyze Mode*'s Assumption Revisited 95
4.4 Enhancement of the *Analyze Mode* 99
4.4.1 Quantifying Viability 99
4.4.2 Modifying the *Analyze Mode* 106
4.4.3 Preparation: Estimating a Tractable Analysis Scale 108
4.4.4 Enhanced Viability Analysis: First-step Mutants 112
4.5 Automation of the Analysis 120
4.5.1 Automation Cycle: Single-step Analysis 121
4.5.2 Multi-step Analysis 123
4.6 Reflections on the Methodology of Mutational Analysis 126
4.6.1 Viability Analysis Results Overview 126
4.6.2 Further Modification and Enhancement of Analysis 128
4.7 Additional Investigation: Redesigning the Prototype 129
4.7.1 Designing `h-copy`-free Prototypes 130
4.7.2 Analyses of the `h-copy`-free Prototypes 131
4.7.3 Reflective Remark: Redesigning the Prototype 139
4.8 Closing Remarks 140

5 Conclusions and Implications 142
5.1 Thesis Summary 142
5.2 Contribution and Concluding Remarks 143
5.2.1 Mutational Pathways of the Prototype 143
5.2.2 Ancestor Design 145
5.3 Future Prospects 145
5.3.1 More Effective Mutation Analysis 146
5.3.2 More Effective Ancestor Design 148
5.3.3 Better Evolutionary Understanding 150
5.3.4 For a General Understanding: Speculative Remarks 151
5.4 Final Words 154

References 155

Appendix A Phenome Design and Mechanism A1

Appendix B Selected Memory Images B1

Appendix C Trace File Formats C1

Appendix D Instruction Library Entries D1

Appendix E Configuration File (Excerpt) E1

Appendix F Analysis Code F1

Appendix G Automation Scripts G1

Appendix H Glossary H1

Tomonori Hasegawa

On the Evolution of Genotype-Phenotype Mapping: Exploring Viability in the Avida Artificial Life System

Abstract

The seminal architecture of machine self-reproduction originally formulated by John von Neumann underpins the mechanism of self-reproduction equipped with genotype and phenotype. In this thesis, initially, a hand-designed prototype von Neumann style self-reproducer as an ancestor is described within the context of the artificial life system *Avida*. The behaviour of the prototype self-reproducer is studied in search of evolvable genotype-phenotype mapping that may potentially give rise to evolvable complexity. A finding of immediate degeneration of the prototype into a self-copying mode of reproduction requires further systematic analysis of mutational pathways. Through demarcating a feasible and plausible characterisation and classification of strains, the notion of *viability* is revisited, which ends up being defined as quantitative potential for exponential population growth. Based on this, a framework of analysis of mutants' evolutionary potential is proposed, and, subsequently, the implementation of an enhanced version of the standard Avida analysis tool for viability analysis as well as the application of it to the prototype self-reproducer strain are demonstrated. Initial results from a one-step single-point-mutation space of the prototype, and further, from a multi-step mutation space, are presented. In the particular case of the analysis of the prototype, the majority of mutants unsurprisingly turn out to be simply infertile, without viability; whereas mutants that prove to be viable are a minority. Nevertheless, by and large, it is pointed out that distinguishing reproduction modes algorithmically is still an open question, much less finer-grained distinction of von Neumann style self-reproducers. Including this issue, specific limitations of the enhanced analysis are discussed for future investigation in this direction.

Chapter 1

Introduction

1.1 Thesis Statement

A hand-designed prototype von Neumann style self-reproducer (von Neumann, 1966) in the context of the artificial life platform known as *Avida* is investigated with respect to evolutionary characteristics peculiar to this reproduction architecture by means of a proposed scheme of mutation analysis. It is hypothesised that mutational pathways to which such a self-reproducer will give rise are distinct from those of a standard self-copying type of self-reproducer, especially in respect of the evolvability of the genotype-phenotype mapping. An analysis method is explored and developed focusing on searching for *viability*, or long-term evolutionary potential, of strains that mutants potentially exhibit within a lineage.

1.2 Motivation and Objectives

The seminal architecture of machine self-reproduction originally formulated by John von Neumann underpins the mechanism of self-reproduction equipped with genotype and phenotype (McMullin, 2000). This architecture is conjectured to play a non-trivial role in evolvable complexity of self-reproducers equipped with it through allowing room for evolvable genotype-phenotype mapping. To date there has been limited understanding of what potential evolvable genotype-phenotype mapping has in effect. The aim of the investigation described in this thesis is to provide a stepping stone to better analysis and understanding of the potential of evolvable genotype-phenotype mapping, grounded in a particular setting of computational agents embedded within the Avida virtual world.

1.3 Course of Investigation

The investigation starts off by implementing a prototype von Neumann style ancestor. The prototype is observed within Avida as a case study to characterise the behaviour of the reproduction architecture under perturbation and mutation.

Following observation of the prototype, further analysis of the evolutionary characterisation of the prototype as an ancestor is attempted for better understanding of the architecture and its potential. In the course of this analysis, the mechanism of an observed degenerative displacement is explained as a step towards reconsidering and redesigning the prototype. Subsequently, the investigation mainly revisits the pre-existing tools for analysis of viability in Avida, and on the basis of that, an idea of enhanced analysis is discussed, developed and evaluated as a step towards a better methodology. Initially, a one-step exhaustive analysis of the prototype's single-point-mutation space is demonstrated, followed by a multi-step selective analysis.

1.4 Contributions

Considering the ultimate objective of observing the evolution of complexity via the evolution of genotype-phenotype mapping, the current investigation is essentially foundational and preliminary. The provided result from the implemented prototype does not, in fact, demonstrate the elaboration of, or any distinctive mutation in, the mapping over evolution: the majority of mutants of the prototype are simply infertile, without long-term viability; whereas mutants selected from the fertile minority prove to give rise to more or less viable individuals.

Nevertheless, the mutational potential of the prototype ancestor is revealed, preliminarily, but systematically and procedurally. Possible mutational pathways of the designed ancestor are classified. On the other hand, specific limitations are found to lie in indeterminacy of lineages faced by the enhanced analysis, and the practical difficulties in classifying reproduction mode. Ideas for further enhancing the analysis are proposed to suggest the future direction of this type of investigation.

The characterisation of an instance of von Neumann style self-reproducer provides a tangible contribution to the field in revealing what tendency and deficiency the particular design can have and how such a self-reproducer can be better explored in Avida. The development of new analysis tools enhances the ability to explore the potential of the reproduction architecture with more practical feasibility.

1.5 Thesis Outline

Chapter 2 provides the background of the current investigation, centred around the theory of machine self-reproduction formulated by John von Neumann. One of the dimensions of the significance of this theory is explained, that is, symbol systems in general, especially those found in molecular biology.

Chapter 3 begins by depicting the architecture of Avida as a platform, and looks at Avida within the context of the developments of such similar artificial life platforms as *Coreworld*. Then the design of a self-reproducer with the von Neumann architecture is laid out. An observation is made about its evolutionary behaviour.

Chapter 4 elaborates on the mutation analysis to be applied on the prototype ancestor and reconsiders the efficacy of the pre-existing analysis tools of Avida. The enhancement of the analysis and the automation of it are described and evaluated. Additionally, the redesigned prototype is considered under the same scheme of analysis.

Chapter 5 summarises the course taken by the investigation and discusses its significance and implications. Future prospects in this line of research within and beyond Avida follow.

1.6 Related Publications

There are several publications associated with the work described in this thesis, contributing to, and formulating, the overall contents and study. These publications are listed chronologically below with brief comments as to how they are related to the current thesis:

- McMullin, B., & Hasegawa, T. (2012). Von Neumann Redux: Revisiting the Self-Referential Logic of Machine Reproduction Using the Avida World. In European Meetings on Cybernetics and Systems Research.
  - Through introducing the degeneration of a particular implementation of the von Neumann architecture of machine self-reproduction in the Avida world (as described in Chapter 3), the distinctiveness of the self-referential logic (or *semantic closure*, as reviewed in Chapter 2) is highlighted in the context of the research programme unifying the subsequent publications, and thus that of the current thesis.
- Hasegawa, T., & McMullin, B. (2012a). Degeneration of a von Neumann Self-Reproducer into a Self-Copier within the Avida World. In T. Ziemke, C. Balkenius, & J. Hallam (Eds.), From Animals to Animats, SAB 2012 (pp. 230–239). Berlin: Springer.
  - The design and a particular behaviour (that is, the degeneration) of a novel, hand-designed von Neumann style self-reproducing ancestor implemented within Avida (the *prototype*) are detailed for the first time, as substantiated in Chapter 3.
- Hasegawa, T., & McMullin, B. (2012b). Revisiting von Neumann's Architecture of Machine Self-Reproduction Using Avida. In European Conference on Complex Systems, ECCS 2012 (pp. 287–293). Berlin: Springer.
  - The motivation of this research programme revisiting the von Neumann architecture of machine self-reproduction is reiterated (i.e., towards evolutionary growth of complexity via some evolution of a genotype-phenotype mapping). Furthermore, the implications of the degeneration of the prototype into a self-copier are discussed, as reflected in Chapter 3.
- McMullin, B., Baugh, D., & Hasegawa, T. (2012). Von Neumann Reproduction: Preliminary Implementation Experience in Coreworlds. In European Conference on Complex Systems, ECCS 2012 (pp. 101–106). Berlin: Springer.
  - From a wider perspective than the previous publications, the research programme investigating von Neumann reproduction is explained in depth. Within this scope, implementations using two of coreworld-type platforms (see Chapter 3), Tierra and Avida, are introduced with preliminary results.

- Hasegawa, T., & McMullin, B. (2012c). Self-Referential Organisation within the Avida World. In Frontiers of Natural Computing Workshop. University of York, UK.
    - From a slightly different perspective of natural computing, or towards non-conventional, alternative computing, the implementation of the prototype von Neumann style self-reproducer in Avida is looked at again, reinforcing the research motivation as in Chapter 2.
- Hasegawa, T., & McMullin, B. (2013a). Analysing the Mutational Pathways of a von Neumann Self-Reproducer within the Avida World. In European Conference on Complex Systems, ECCS 2013.
    - An approach to investigate the mutational pathways of the particular prototype ancestor and an idea of analysis enhancement are first proposed, which was developed into the framework of mutation analysis described in Chapter 4.
- Hasegawa, T., & McMullin, B. (2013b). Exploring the Point-Mutation Space of a von Neumann Self-Reproducer within the Avida World. In Advances in Artificial Life, ECAL 2013 (pp. 316–323). Cambridge, Massachusetts: MIT Press.
    - A first attempt of such an approach as proposed in the above publication is reported. A preliminary examination of the first-step point-mutants of the prototype is presented with results and future directions, which led to the analysis automation demonstrated in Chapter 4.

Chapter 2

Background and Related Work

2.1 Overview

In order to form the background to the research of this thesis, this chapter first introduces the core scheme, the theory of machine self-reproduction by John von Neumann. It is followed by the review of relevant literature to explain the rationale behind the current line of research in the discipline of artificial life. To strengthen the motivation, the next section provides a biological perspective, which is highly relevant to the current context of artificial life, especially in relation to the more general concept of biological symbol systems which underlies self-reproduction, which in turn underlies evolution.

2.2 Machine Self-Reproduction and Evolution

Artificial life is a discipline which seeks to shed light on aspects of biological life such as self-reproduction and evolution. Unlike biology, artificial life focuses more on required or possible processes ("recipes") of life's aspects by in-silico methodology. John von Neumann's seminal architecture of machine self-reproduction is perhaps the earliest theoretical contribution to artificial life, attempting to bridge the gap between the systems of designed physical artefacts or "machines" and the systems of biological organisms (von Neumann, 1951, 1966). The research of this thesis considers questions about evolvability inherent in a particular style of self-reproduction, referred to as von Neumann style self-reproduction throughout the thesis.

2.2.1 Architecture of Machine Self-Reproduction

Machine reproduction was originally analysed theoretically by John von Neumann, largely in the early 1950s. It is recognised as the dawn of the discipline that was formed into artificial life decades later. In brief, his intention was to bridge the gap between artificial life and biological life. His theorisation presents an abstract architecture that models machine reproduction and machine self-reproduction.

As argued by McMullin (2000), one of von Neumann's motives behind this abstract model was to realise the evolutionary growth of complexity in a mechanistic world. Intuitively, one can compare two situations regarding complexity: the engineering situation and the biological situation (see Figure 2.1). In the engineering (or artificial-world) situa-

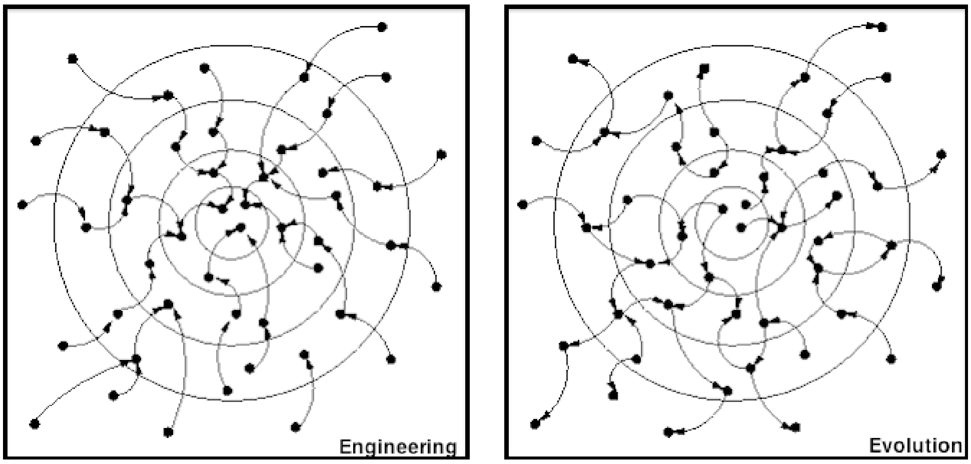


Figure 2.1: Complexity in situations of engineering and evolution (adapted from McMullin, 2000). In the diagram, circles denote the degree of complexity in the abstract sense, with lower complexity inward. Dots and arrows denote machines and construction in the engineering situation on the left, whereas organisms and reproduction in the evolution situation on the right. Note that the engineering situation only allows decreasing complexity denoted by centripetal arrows through construction, while the biological evolutionary situation can also allow maintaining or increasing complexity through reproduction, denoted by neutral and centrifugal arrows.

tion, the complexity does not typically increase or even maintain, but rather only decreases when machines construct other machines. As opposed to this, in the biological (or natural-world) situation, there are not only cases where the complexity decreases, but also where it maintains or even increases. To make such a situation be the case in a mechanistic world, von Neumann set out to formulate an architecture of machine reproduction. He identified that supporting a mechanical genotype-phenotype mapping in such an architecture may allow the evolutionary growth of complexity in the world of self-reproducing machines. Complexity in this context is only roughly and intuitively defined by von Neumann, so likewise the current research does not go beyond it and endeavour to define or measure complexity precisely or formally. There are a number of studies with evolutionary and/or ecological foci that aim at exploring what kind of, and how, complex behaviours can be achieved by evolving agents in an artificial life world. In this framework, a behaviour of a system as a whole can be regarded as complex, if the behaviour is not expected from, or is more than, the "sum" of its components. Although outside the purview of the current research, there are studies that focus on defining and measuring complexity, for example, in terms of *entropy* as in information theory (see Adami et al., 2000). As for the relevant course taken in defining and measuring complexity in *complexity science*, a comprehensive overview with historical aspects is provided by Mitchell (2009).

Formalisation

The von Neumann architecture for general machine reproduction can be schematically depicted as shown in Figure 2.2. A parent machine reproducer (to the left) reproduces an offspring machine (to the right). At the highest level, the parent machine is comprised of two parts: active and passive parts, labelled P and G, respectively. P consists of a *programmable constructor* (A), a *copier* (B), a *control* (C), and arbitrary *"ancillary"*

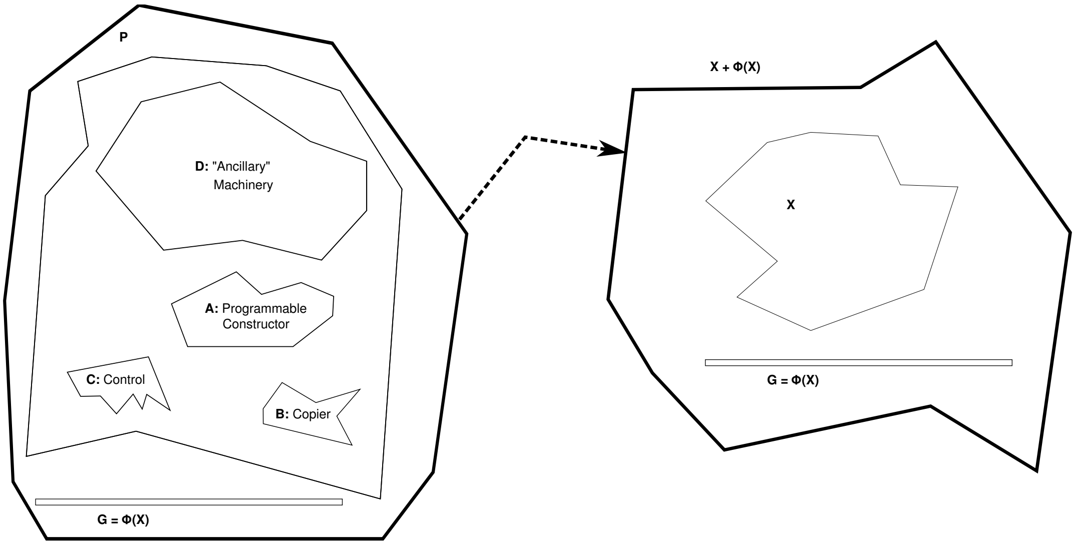


Figure 2.2: The schematic von Neumann style general architecture of machine reproduction (adapted from the slides of McMullin, 2012). For the sake of convenience, the active part is labelled as P and the passive part as G. The parent machine reproduces an arbitrary offspring machine by decoding and copying the description of an arbitrary machine X ($G = \Phi(X)$). This is realised by utilising the programmable constructor (A), the copier (B), and the controller (C); the "ancillary" machinery (D) does not directly engage in this reproductive process.

machinery (D). G is a tape that describes or encodes an arbitrary machine X, via a function $\Phi(X)$, which is coded in a way A can decode. The parent machine produces an arbitrary offspring machine, consisting of X and its description $\Phi(X)$, by decoding and copying G. This reproductive process is realised by the components A, B, and C of P; D is ancillary and here does not directly engage in this process.

As a special case, the architecture for *self*-reproduction can then be schematically depicted as shown in Figure 2.3. A parent machine (to the left) reproduces an offspring machine (to the right). Now G is chosen to be a tape that describes the machine P (i.e. the assembly $A+B+C+D$), again, relative to the specific description language, or "decoding" implemented by A. In operation, A decodes G to produce another instance of P=$A+B+C+D$, B constructs a copy of G, and A and B (and D) are controlled and co-ordinated by C. C ultimately detaches the complete offspring machine instance, $P+G$, identical to the parent, thus realising self-reproduction. As this basic architecture and self-reproducing functionality will be common for any arbitrary D (within the constructive capabilities of A, and assuming that D's operations, whatever they may be, do not interfere with $A+B+C$), this implies the existence of an indefinitely large space of self-reproducing machines, all connected via spontaneous perturbations of the G component (which therefore correspond to heritable mutations). The labels, A, B, C, D, and Φ, accord to von Neumann's original labelling (see von Neumann, 1966). P and G are named after their being analogous to "phenotype" and "genotype", respectively (or, in an individual, instantiated machine, these may be called "phenome" and "genome").

The von Neumann architecture of machine self-reproduction reflects a similar abstract structure to that which is now known to support self-reproduction in biological organ-

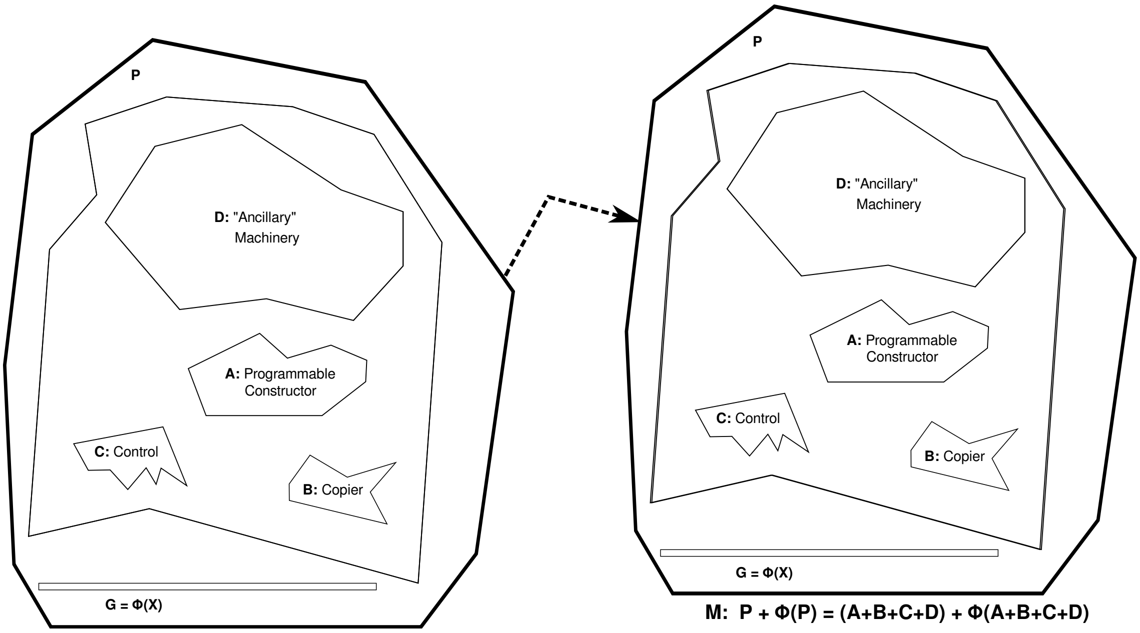


Figure 2.3: The schematic von Neumann style architecture of machine self-reproduction, when the machine $X = P$ (adapted from the slides of McMullin, 2012). The parent machine reproduces its identical offspring machine by decoding and copying its description, by utilising the programmable constructor (A), the copier (B), and the controller (C); again, the "ancillary" machinery (D) does not directly engage in this reproductive process. Note that this architecture supports inheritable changes (i.e., *mutations*) in G.

isms. Based on this understanding, it is reasonable to consider the active machinery as representing the *phenotype* and the passive description tape as representing the *genotype*, even though von Neumann himself did not emphasise (at least) the analogy between his proposed architecture and that in biology.

Inheritable Mutation

In theory, any specific *strain* (more rigorously defined in the context of *Avida* later in Chapters 3 and 4) of such a self-reproducing machine can exhibit exponential population growth, in the absence of resource constraints and perturbation. This potential for exponential growth, combined with mutation (variation) and resource constraints will give rise to variety and competition, and hence the conventional, neo-Darwinian selection and evolution.

It is significant that, in the presence of some perturbation, or inheritable *mutation*, the von Neumann style of machine self-reproduction can theoretically exhibit the evolvability of the genotype-phenotype mapping itself (McMullin, 2000; McMullin et al., 2001); that is, of the "decoding" function implemented by the component sub-machine A. This possibility arises provided A is itself described (in a self-consistent manner) within the genotype, G, and in sufficient detail that there exist potential mutations (or perturbations of G) that do change the decoding function implemented by the (expressed, mutated) A in the following generation. In general, this mutated A may or may not be capable of decoding the inherited genome (or description tape) G in a way that still preserves the self-reproduction functionality, although the *prima facie* likelihood is that such mutational events will fundamentally disrupt self-reproduction. For a change affecting the genotype-phenotype mapping to be truly inheritable, and for a consequent machine to be self-reproducing, the reproduction mechanism must somehow survive through mutational events, sustaining a genotype-phenotype mapping that is still applicable (i.e. "backward compatible") to the mutated description, so as to keep *self*-reproducing.

The current research is concerned precisely with exploring this possibility empirically, at least for one "toy" example of a von Neumann self-reproducer. In other words, the effect of a mutation affecting the genotype-phenotype mapping is of particular interest. This is a much specialised scope, as opposed to the effect of a mutation in general. An affected genotype-phenotype mapping is much more likely to fail to reproduce or self-reproduce. More in general (i.e., regardless of whether or not affecting the genotype-phenotype mapping), it is a fact that von Neumann (p.87, 1966) himself makes a comment touching on the effect of an inheritable mutation in this architecture, closing his lecture on self-reproducing automata in 1949: "So, while this system is exceedingly primitive, it has the trait of an inheritable mutation, even to the point that a mutation made at random is most probably lethal, but may be non-lethal and inheritable." Apparently he did not elaborate on this point any further. Nevertheless, chances are, there may be a distinctive mutational pathway where an organism with a mutation in the constructor is capable of self-reproducing (or, *breeding true*) in a characteristic reproduction mode.

2.2.2 Underlying Influences

Machine reproduction (and machine self-reproduction) with von Neumann's architecture hinges upon: (a) the decomposition into active, constructive machinery and a separate, passive "description tape"; and upon (b) the sub-component A, the programmable constructor, which can construct any arbitrary machine from a description tape fed to it. As von Neumann himself credits, the idea and the design were inspired by (or rather originate with) Turing's abstract elaboration of computing machines that are capable of universal computing (Turing, 1936).

There is a striking resemblance between the architecture of machine self-reproduction and the biological architecture of genotype and phenotype, but the terminology in von Neumann's architecture did not employ the terms genotype and phenotype as such. Chronologically, the terms (and the concepts of) genotype and phenotype already existed, coined and defined by Johannsen in 1911, in the context of genetics; the von Neumann analysis, in around 1950, actually preceded Watson and Crick's discovery of the structure of DNA in 1953 and the subsequent investigation of the mechanism of genotype and phenotype at a molecular level. One possible interpretation is that the concepts of genotype and phenotype were not so prevalent to the extent that these were natural terms to describe the active machine and the passive description when von Neumann proposed his architecture.

Computation and Turing Machines

Turing machines were a first pure mathematical description of which numbers are, and are not, *computable*, and how the computable numbers can be computed. This was a response to one of the questions about mathematics per se posed by Hilbert in 1928, known as the *Entscheidungsproblem*, or the "decision problem". Preceding questions were solved in effect by Gödel in 1930, whose theorem concluded that any minimally useful system of mathematics is either inconsistent or incomplete. It was in 1935 that Turing defined Turing machines that work with definite procedures, or algorithms in a modern term, so as to achieve universal computability; whereas, it was shown that there are certain limits of undecidability to such computation by demonstrating that the halting problem is undecidable by a "universal" Turing machine running on it. The proposition of the halting problem considered by Turing is: "there exists a Turing machine which can algorithmically decide whether any given Turing machine will halt". Turing proved that the negation of this proposition is true, that is, that there exists no such algorithm or Turing machine. The conclusion can be translated into a modern situation where an arbitrary computer program cannot reliably detect by itself whether it may end up in an infinite loop or not.

What was remarkable about the machines pictured by Turing is, first of all, that a Turing machine is formalised as a finite set of states and rules: it is a finite state automaton provided with an infinite tape that numbers are read from and written to, and a transition table that defines possible steps for each state and symbol, such as how to transition a state into another and what to do to the tape at the next time step. By taking steps according to the tape and the transition table, the automaton can carry out a procedure of calculation. These steps are defined to be simple enough so that any machines (within this formalised framework) of different capabilities can perform computation. Second of all, it was a novel idea of Turing that such a machine itself can be described on the tape, so in effect a Turing machine can be an input to another. That means that Turing machines can emulate other machines, and that there exist "universal" Turing machines which can emulate any such automaton, provided with the description of the target automata. The halting classification ("decision") problem, however, cannot be computed and hence is undecidable by any Turing machine. This proof, conversely, ended up defining what it means to be computable.

Thus, a universal Turing machine requires the description tape encoding the target machine in the form that the Turing machine can manipulate (i.e., in natural numbers, or more primitively, sequences of two digits 0 and 1 that can represent natural numbers, states and moves). It is a programmable computer, which was an inspiration for von Neumann's programmable constructor. Thus von Neumann's design is analogous to a universal Turing machine in having the (universal) programmable constructor that can build any arbitrary machine inasmuch as it is encoded on the tape in a form that the constructor can decode. Then it is noticeable that, when the description tape encodes the active component itself which runs on it, the reproduction is tantamount to machine *self*-reproduction. As such, the parent machine can produce an identical offspring machine, by following deterministic, mechanical procedures. The offspring machine is identical unless the description tape undergoes perturbation in its content; if this occurs, the change functions as a mutation in the offspring, whereas perturbations in the active component do not serve as mutation as they are not inheritable.

Regarding the analogy between the two "universalities" of computation (i.e., Turing's) and of construction (i.e., von Neumann's), the interpretation of von Neumann's concept of universal computer-constructor embedded in his self-reproductive architecture can be subtle, as reviewed by McMullin (1993). For the current working purpose, no rigorous stance is going to be adopted over whether those universalities are necessary or sufficient for self-reproduction; but roughly it is assumed that the computation universality and construction universality are basically compatible attributes supported, if implicitly, within the system used (*Avida*, which will be overviewed in Section 3.2 in Chapter 3).

Heredity and Evolution

The understanding of evolution as known today had been markedly shaped by the time Darwin published "On the Origin of Species" in 1859. One of the prevailing views of evolution before Darwin was by Lamarck, which favours inheritance of acquired traits and tendency towards progression. This view is invalidated or at least highly modified today. What Darwin proposed was a mechanism of evolution that is driven by natural selection with inheritable traits which contribute to variation that is subject to selection, in the same vein as the contemporary notion on population growth by Malthus and that of the "invisible hand" by Smith. The exact mechanism of how traits are inherited still remained unclear until after the 1900 re-evaluation of Mendel's work. His work on heredity is another significant pillar of the understanding of evolution that was already established by mid-1860s, which identified that there are some kinds of discrete units behind inheritable traits.

Subsequently, Schrödinger provided his quantum physics point of view on heredity, in his lecture series on "What Is Life" in 1943, well in advance of von Neumann's theory of machine self-reproduction, dating from the late 1940s and early 1950s. In his lecture, Schrödinger (1944) discusses the "code-script", which is a blueprint of an individual organism, and goes so far as to refer to a physical, molecular mechanism of mutation. It may not sound quite precise from a contemporary view of molecular biology that matured after DNA was identified as the substrate carrying *genes* in 1953, in the sense that there is not necessarily exact correspondence between genotype and phenotype; nonetheless, Schrödinger's explanation is reminiscent of heredity as known today, revealing how mutation occurs and gets expressed, inheritable over generations. Even though it is not certain if von Neumann was explicitly inspired by Schrödinger, it is striking that his architecture of machine self-reproduction entails a mechanism of heredity as envisaged by Schrödinger, and that the decomposition into the description tape and the programmable constructor (and the other associated components) in the architecture are parallel to the biological decomposition now generally identified as genotype and phenotype.

2.2.3 Artificial Life Investigations

Self-reproduction and evolution are, among others, central subjects for investigation in the field of artificial life (ALife). Self-reproduction of organisms is an important component to realise open-ended evolution in artificial life platforms (T. Taylor, 2013). There are several classes of abstract machines (automata) that von Neumann used in order to model machine self-reproduction, capable of evolving, for which he is said to be the founder of the discipline of ALife. The ideas he proposed include that of a possible class known as the kinematic automata (KA), and that of a possible class of implementation known as the cellular automata (CA). These are two characteristic types of automata that influenced following artificial life studies into logical aspects of self-reproduction and its dynamics. Although it is not straightforward to define non-trivial self-reproduction, Moore (1962) mentions one triviality of self-reproduction as found in crystal growth. He follows that it is in order to sufficiently overcome such triviality that computational universality was incorporated in self-reproducers in the CA by von Neumann. At least, there is required some mechanism in self-reproduction to retain evolvability in a given framework.

In the post-von-Neumann era, Langton is one of the pioneers who contributed to the establishment of the discipline of ALife (Langton, 1994; Boden, 1996). ALife, on the whole, is now recognised as a field that aims to shed light on the old question of *what is life* by means of artificial, especially computational, systems. By this time, as the computational abilities had been developed, more artificial life studies taking approaches from the perspective of artificial chemistry (AChem) appeared. Although spanning a wide research spectrum, AChem, as the name suggests, generally regards "chemical" reactions in artificial models as important building blocks of simulated "biological" phenomena, including, again, self-reproduction and evolution observed in real biology (see Dittrich et al., 2001, for a general overview).

Automata Investigations: *Kinematic* and *Cellular* Automata

Von Neumann sketched out KA before the CA approach. A KA is a world of robot-like machines that consist of computing elements and primitive action elements, coupled with an environment. Situated in an environment, the machine self-reproduces by constructing a machine from the nearby elements diffusing in the environment. In contrast to CA, however, von Neumann and his successors barely elaborated the KA concept; it lacked theoretical benefit compared to CA since it was then technically too intricate to design self-reproducing KA at a meaningful level of detail, whether virtually or physically.

Nevertheless, there have been endeavours to construct such robot-like creatures similar to KA in a virtual world. In one, Sims (1994) uses an external genetic algorithm (GA) to obtain desired morphologies of creatures, by optimising their control over body to achieve certain behavioural purposes in a virtual environment. It is Holland who undertook the studies towards adaptive systems, which are found among the initial CA investigations as early as Burks (1970). Holland's studies would later lead to the framework of GA (1975). The GA views biological evolution as an optimisation process and enables a form of simulated evolution. The key feature that has had significant influence in relevant studies is the *fitness function*, which is incorporated in the system in a predefined way to calculate how advantageous each agent is within a population. Due to the fitness value, typically calculated relative to the rest of the population, the evaluation of individual agents may be influenced by the population. Genetic operations such as crossover and/or mutations are repeatedly applied to the agents' genomes in a population to produce the next generation population. The Sims' creatures have genotype (represented as directed graphs) and phenotype (embodied as morphologies in the virtual world), and are reproduced externally by means of the genetic operators, which is rather as a part of an external support framework, than through their own operations or activities within their virtual environment. A similar study focusing on self-reproduction in a real-world situation was taken by Zykov et al. (2005). Neither of those agents self-reproduce in the manner of the von Neumann architecture, specifically in that the relationship between genotype and phenotype is fixed or immutable in those universes.

More recently, there has been a study using *reified quines* (Williams, 2011), which can be thought of as another example (virtual) implementation of KA. This work is clearly inspired by von Neumann in its design: agents interact in an environment where there is the diffusion of primitive components; and they affect each other to engender construction. *Quines* are a type of machine self-reproducer coded in a higher-level language. They are virtually reified in the model, equipped with the mechanism of genotype and phenotype, thus self-reproducing and undergoing neo-Darwinian evolution. According to Williams, it is unlike *worms*, which are defined as a simpler type that is only capable of "copying the phenotype directly"; worms are rather subject to Lamarckian evolution even if they are programmed to self-reproduce (in a loose sense that perturbations in the phenotype would be copied and thus inherited). One would notice that this work's result is that such populations of KA exhibit exponential growth as seen in the world of biological cells, until they reach a saturation. Generally self-reproduction is associated with exponential population growth, as long as there is no limit to the supply of energy, components, time, space, and so forth. In his subsequent work, Williams (2013, 2014) further presents a developed class of self-reproducers of this type in a KA-like universe (*distributed virtual machines*), to exploit the potential of evolving from such an ancestor to a more complex and more efficient style of self-reproduction.

Following the conceptualisation of KA, what von Neumann himself elaborated in most detail was CA (von Neumann, 1966). A CA is a potentially infinite network of "cells", typically organised in a two dimensional grid, each cell of which is a finite state machine. (Here, "cells" are not intended as analogues of biological cells, but as abstract "atomic" components defining this particular mechanistic universe.) A CA can be one-dimensional (as extensively investigated by Wolfram decades later from mid-1980s to late 1990s, for example); or logically, three- or higher- dimensional CA are conceivable as well. However, the discussion here will be mainly concerned with two-dimensional CA. Each cell of the grid in CA is spatially fixed in a two-dimensional world and changes its state according to a transition rule. The rule normally concerns the states of the cell itself and its immediately adjacent cells. The interaction of cells is local, in that sense. A pattern of states (or class of dynamically linked patterns) across a collection of cells can be interpreted as a machine embedded in the CA. In suitable conditions, such an embedded machine may exhibit self-reproduction within the world represented by the CA as a whole.

Von Neumann's original manuscripts on this theme were reviewed by researchers including Burks (1970) and complementary works in the same vein have been carried out. Burks chiefly edited the manuscripts and collected the relevant works to deepen and widen the scope of its potential. The original design of the 29-state CA by von Neumann had not been implemented until around four decades later (Nobili et al., 1994; Pesavento, 1995) when computers became much more capable. In the early CA investigations, Codd (1968) searched for a possible simplification of von Neumann's original design, which was later reviewed and implemented by Hutton (2010). Similarly, alternative CA designs were sought after and refined over time; most CA used in such studies are typically homogenous and symmetrical, spatially fixed, having small neighbourhood, with a relatively small number of states per cell (see Sipper, 1998, for a general overview of this line of research).

In particular, Langton (1984) used CA as a framework for the implementation of another form of self-reproducers, so-called “Langton Loops”.[1] This work proposes simpler, but supposedly still non-trivial, self-reproducing machines without a universal constructor (or a universal computer), in distinction to von Neumann’s original design (or Codd’s version of it). From one point of view, one can see a spectrum of simple to complex self-reproducers within CA worlds, with Langton’s design and von Neumann design at the respective ends. In the self-reproduction of the Langton loops, it is implied that construction universality (or computation universality) is not necessary, but that adding it might bring auxiliary functionality to self-reproducers through evolution. Unlike Burks, whose emphasis seems to be the universal computation and construction for self-reproduction, Langton looked at the von Neumann style architecture in connection with biology: the importance was highlighted of the separation and the mapping between genotype and phenotype. Langton seems to have been cognisant of the salience of the distinction between the interpreter and the “interpretee”, and the mapping between them, but he appears to have held that the self-replicating loop with an explicit but impoverished genotype-phenotype mapping would be enough for the purpose of artificial life modelling or at least qualified as “non-trivial”. Langton reinforced the idea that inheritable mutation is essential, both computationally and biologically; and it is thus implied that evolvability (or, further, evolution’s “open-endedness”) is one of the key aspects of life.

Artificial Chemistry and *The RNA World*

The perspective of artificial chemistry (AChem) has been an inspiration for automata studies, including the KA-like reified quines mentioned above, both in the aim and approach. Some typical artificial chemistry platforms deal with string-like molecules which can replicate through interaction (such as binding and alignment of strings, or chemical reactions) with each other: *Typogenetics* (Hofstadter, 1979) (later revisited by Morris, 1987 and Bobrik et al., 2008); and *Stringmol* (S. J. Hickinbotham et al., 2010; S. Hickinbotham, Clark, et al., 2011; S. Hickinbotham, Stepney, et al., 2011) are specific examples. There are molecular biological entities and phenomena which these platforms are to simulate. For instance, Typogenetics simulates molecular components such as DNA strands, enzymes, and amino acids, whereas Stringmol simulates proteins and small RNA molecules (relatively shorter RNAs produced by bacteria). Neither of these necessarily aims to imitate the molecular world in physico-chemical detail, but to construct a reasonably abstracted version of such a self-contained biological symbol system. Biological molecules, including DNA, RNA, and proteins, and their coding relationship will be overviewed here and in the next Section 2.3 from the perspective of the current research. In addition to the platforms mentioned above, *coreworld* type systems may also be categorised as artificial chemistry platforms. These will be introduced in the context of *Avida*, the target system in the current research, in Section 3.2 in Chapter 3.

To a certain extent, the modelling of artificial chemistry platforms, particularly in

[1] With regards to the population growth brought about by self-reproduction, what Langton Loops are able to observe is at most polynomial population growth (according to Mange et al., 2004), seemingly unlike the real biological self-reproduction. However, it is rather because of the spatial effect of the two-dimensional CA world: each self-reproducing loop is fixed in space and when one comes across another, the former is to be “dead” and stop self-reproducing. Self-reproduction will, naturally, lead to exponential growth of population if there are no such spatial or other resource restraints.

relation to self-reproduction, is motivated by the origin-of-life hypothesis called the RNA world. One of the central questions in the field is: why does life as known today operate on the foundation of DNA, RNA and protein? Among other explanations[2], the RNA world hypothesis also counts as a realistic scenario. The discovery of the double helix structure of DNA was a landmark in molecular biology, on which enormous investigations have been predicated, revealing DNA as a form of genetic information storage.

The RNA world hypothesis (see Gilbert, 1986, for the original idea of the RNA world hypothesis; and Gollihar et al., 2014, for the latest interpretation in this line of research of the origin of life) proposes that the prebiotic world started as a situation where only RNA molecules existed, and through interaction over evolutionary time, they came to specialise ("division of labour") into either information storage or catalytic and/or enzymic functions, which are now roles played by DNA and protein, respectively. The evolution of the division of labour between *templates* and *catalysts* has been further investigated (Takeuchi et al., 2011). Templates store information while catalysts realise different biochemical functions or transformations. It is suggested that DNA's lack of catalytic ability serves as selective pressure for DNA to emerge in the RNA world. Along the same line, it is suggested that *RNA-adapters* may help overcome the so-called information threshold (de Boer & Hogeweg, 2012). The information threshold means the trade-off between the necessary error-correction mechanisms of a genetic code and the decreasing size of it due to mutational pressure. In their simulated world of RNA molecules, RNA-adapters, which are conceived as distinctly shorter molecules compared to typical RNA functional molecules, increase in number with a higher mutation rate, and help diversify functional (or regulatory) structures of the RNA sequences. In other words, the information threshold shapes the emergence of coding between genotype and phenotype.

Noticeably, not only at the RNA level, but also at the protein level, there exists the folded ("secondary" and "tertiary") structure of proteins which give rise to enzymatic function, as opposed to that of the ("primary") sequence of amino acids. As such, many artificial chemistry models, more or less, are concerned with higher-order structures (e.g. folding, binding) of molecules. On the other hand, in the Avida system in question, although sometimes categorised as an artificial chemistry model, genotype and phenotype can exist one-dimensionally, linearly on the circular memory as described later; there is apparently no precise analogue for structural functionality like RNA or protein folding. However, addressing labels can give higher level structure, by redirecting the execution of programs on the linear memory, which otherwise proceeds just successively from the starting location. The current research, however, mainly looks into the linear sequences (*strains*) of organisms in Avida.

[2] For example, Kauffman (1993) argued abstractly that there must have been some autocatalytic chemical system in the prebiotic world which possibly and likely led to life forms as known today. Although outside of the current research scope, it is noteworthy that ALife and complexity science witnessed the appearance and use of random boolean networks (RBNs) by Kauffman et al., the relatively simple structure of which is reminiscent of CA. If a CA had cells that have different rules and that are not fixed spatially but connected arbitrarily, it would be a RBN. RBNs have been used and developed by Kauffman and his followers since the 1960s, to model genes interacting with and regulating one another. It is a line of research which deals with evolutionary diversity and interaction of genes, rather than concerning self-reproduction in evolution like the current research.

2.3 Biological Symbol Systems

Natural languages, computer programs, and genetic codes are some instances of symbol systems observable in various levels in the world from micro to macro. In such systems, generally, "materials" of some substrate become "symbols" that signify something somehow at a certain point as long as there is a defined rule for interpretation. To be valid, such a system must comprise a collection of relationships between symbols and what they "symbolise" (or signify) based on certain materials. Especially, natural languages and genetic codes, which are complex symbol systems shaped and formed by nature, have been evolved into what are known today over evolutionary time. Such symbol systems can be regarded as complex coding systems just like computer programs (where there underlies a certain encoding/decoding mechanism). Likewise, computer programs can be compared to natural languages or genetic codes, even though artificially shaped and formed. Most of the above-mentioned are, primarily, symbol systems that may roughly be categorised as digital, as in communication theory, which involves "discretisation" in one way or another. Due to the arbitrariness of digital coding behind relationships between symbols and the symbolised, and between the encoded and the decoded, it is difficult to elucidate how symbol systems emerge and evolve adequately. That being said, to contrast, the interpretation in biological symbol systems is mutable and evolvable, whereas the interpretation in conventional computer programs is fixed and immutable by some external entity. This might be relevant to the distinction between evolution and engineering as abstractly depicted in Figure 2.1 in Section 2.2. In this section, biological "counterparts" of computing systems are detailed.

2.3.1 Evolution of Symbol Systems

The current research was conducted originally as a part of a collaborative project, *EvoSym*, involving a collaboration between three research groups[3] from three different perspectives upon the evolution of "biological symbol systems". The three perspectives are: molecular biology, artificial life, and biosemiotics. It is expected, by extension, that the more the emergence of symbol systems can be understood, the more insights into the evolution of symbol systems will be illuminated, and vice versa.

In this project, the methodology of molecular biology is to look into the world of biomolecules by modelling the RNA world (similar to that of artificial chemistry) in order to understand how the molecular biological world as we know it has shaped up over evolutionary time. The methodology of artificial life is to deal with symbol systems in a pure computational world such as *Avida* and *Tierra* (refer to Section 3.2 in Chapter 3 for the profiles of those systems), focusing on a more abstract architecture of symbol systems. The methodology of biosemiotics, on the other hand, aims to deepen the understanding of the nature of meaning that arises (or evolutionarily emerges) in higher-order communication. That may or may not be used to analyse both the biological symbol systems and those of artificial life.

[3] The EvoSym project consists of three work packages conducted by laboratories from three universities: Universiteit Utrecht, Netherlands (UU); Dublin City University, Ireland (DCU); and Vrije Universiteit Brussel, Belgium (VUB). The project ran from November 2010 for three years.

As explained, the current line of artificial life research pays particular attention to the theory of machine self-reproduction by von Neumann. The architecture provides a framework characterised by processes of copying and decoding, and the decoding process literally entails the coding mechanism of a certain symbol system. The von Neumann architecture might particularly allow complexity to grow evolutionarily in a computational world with a mutable genotype-phenotype mapping. Such a symbol system may or may not turn out to be a substantial advantage for self-reproducers that employ it in an evolutionary sense. Strictly speaking, such mutability is not necessarily guaranteed in the sense that any mutation can destroy the reproductive cycle; but there is at least the potential that a mutation, even one altering the genotype-phenotype mapping, can breed true without harming the reproductive cycle: through mutation, a genotype-phenotype mapping at a certain generation can therefore change into a different one in the next generation. Theoretically, there can be a variety of genotype-phenotype mappings within mutational reach. The rationale is that both genotype and phenotype can be enumerated or labelled with numerical values, and that a genotype-phenotype mapping amounts in that sense to a mapping between numbers. In this light, what the current research investigates is how one can (and how far one has to) tread mutational pathways within a certain universe so as to reach a functionally or behaviourally distinctive, “interesting” genotype-phenotype mapping employed by a self-reproducing machine which is otherwise unseen or less accessible among others.

The potential, that is, the evolvability of the genotype-phenotype mapping and possibly the effect it has upon the growth of complexity, is here investigated within the purely computational world of an artificial life system. In one platform of Avida,[4] a von Neumann style self-reproducer was invented by design in order to study its characteristics in computational situations. The artificial life platforms like Avida have major concepts in common with artificial chemistry platforms, in that agents interact on the basis of given rules and constraints in computer simulation, exhibiting emerging, self-organising, and evolving entities in populations. Since such artificial life platforms have been used for evolutionary studies, it is a particular interest to explore the evolutionary potential which such a genotype-phenotype architecture exhibits. For example, characteristics that are long-term, open-ended, and dynamical might be attributed to self-reproduction that incorporates some genotype-phenotype architecture.

In particular, the artificiality of artificial life platforms becomes more apparent especially when demonstrating abstract characteristics of life, because phenomena observed in artificial life platforms do not necessarily correspond to particular physical, chemical, or biological phenomena. In this sense, the focus of this thesis is centred on demonstrating the rather abstract model, that is, a model of machine self-reproduction proposed by von Neumann within a purely computational situation. That being said, it would also be reasonable and appropriate to draw some analogy from biology or from different domains, in order to better compare and contrast understandings of evolutionary potential of a genotype-phenotype architecture (and especially evolvability of a genotype-phenotype mapping). The following subsection overviews a kind of genotype-phenotype relationship

[4] The Tierra system is the other platform chosen by DCU within the EvoSym project. These were chosen as predominantly used in the relevant field with relatively abundant literatures. Also, von Neumann style self-reproducers had not been previously investigated within these platforms before this project.

at a certain level of self-organising system found in molecular biology, and provides a certain biosemiotic perspective to better understand the significance of such a relationship.

2.3.2 Self-Referential Logic

There underlies a self-referential logic in the world of biomolecules. Hofstadter (1979, 1985), for example, has pointed out and depicted that there is this intriguing relationship between biomolecules regarding self-reference in relation to self-reproduction. In the same light, Pattee (1982, 1995, 2005) coined the term *semantic closure* to refer to the concept of the intertwined self-referential relationship, with the emphasis on the embodiment of such a relationship with physical requirements.

Interrelationship of DNA, RNA, and Protein

DNA, passive in itself, not only copies itself through *replication*, but also undergoes the processes called *transcription* and *translation* so that proteins that DNA codes are produced. A strand of DNA, to begin with, is transcribed into a distinct form called messenger RNA (mRNA). Transcription is guided by particular enzymes, which are themselves proteins produced beforehand. The DNA strands are complementary, but transcription itself is not simply a copying process as it involves specific alteration. A strand of mRNA, then, is translated into an amino acid sequence, which self-organises (folds) into a functional protein (see Figure 2.4). Translation, effectively a decoding process, proceeds with the help of ribosomes and transfer RNAs (tRNAs). Ribosomes are comprised of proteins and ribosomal RNAs (rRNAs). Molecules of tRNA are floating around in the molecular environment and have an anticodon on one end and an amino acid on another end. A ribosome reads codons sequentially from the strand of mRNA, and, for each codon read, capture a matching anti-codon that a tRNA molecule has. Then the tRNA molecule releases the amino acid on the other end and this is appended to the amino acid sequence (protein) which is under construction. The translation rule, or the combination of a codon and an amino-acid, is actually implemented by the aminoacyl-tRNA synthetases (AARSs): those enzymatic protein molecules help load tRNAs with proper amino-acids. The definition of the code hinges on the work done by AARSs.

On the whole, the translation rule (represented by tRNAs and underpinned by AARSs) and the translation procedure (represented by ribosomes) are described in the DNA (and in the mRNA) in the first place, and once they are activated, they behave as encoded. It is a closed relationship in the sense that DNA, and also mRNA, cannot work in isolation but with some proteins and ribosomes; however, without ribosomes, which contain proteins, mRNA cannot translate into proteins. On top of that, the coding relationship between these molecular components must be self-consistent. Proteins fold differently so as to have functionality accordingly. Proteins have structures depending on the dimension: namely, primary, secondary and tertiary structures. Primary and tertiary structures are noteworthy as they denote the sequence of amino acids and the three-dimensional and functional shape, respectively.

Hofstadter also points out that these components (i.e., DNA, RNA, and protein) can have multiple roles comparable to components in a modern computer: program, data, language processor, interpreter, and so forth. It is not easy to identify the origin of

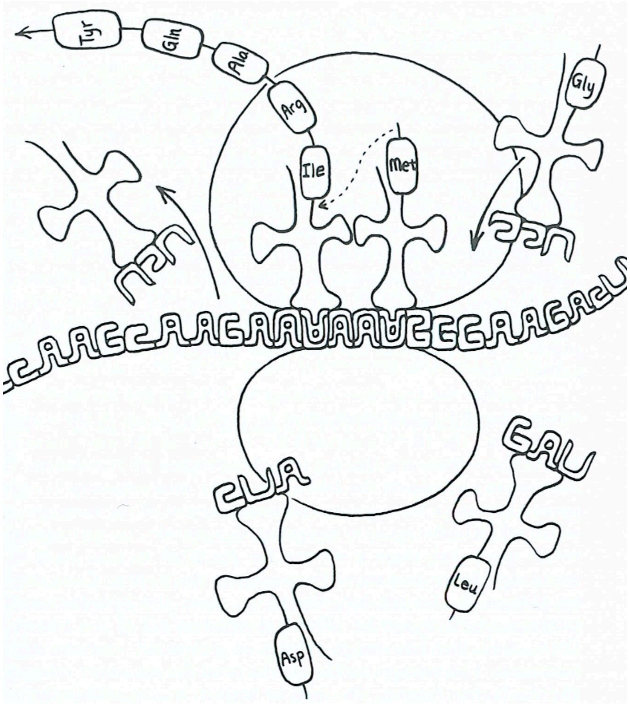


Figure 2.4: Schematic process of *Translation* from a section of mRNA into an amino acid sequence (adapted from Hofstadter, 1979, chapter 16, figure 96). An mRNA molecule, drawn as a sequence of letters in the middle, can be delimited into three-letter-long codons, each of which waits to match an anticodon held by the clover-shaped tRNA molecules floating around. Ribosomes (drawn as two circles in the centre) facilitate this process, creating proteins by appending amino acids, which the tRNA molecules hold on the other end.

such an interrelationship, or imagine how the mechanism came into existence in a way capable of “self-bootstrapping”. Nevertheless, this interrelationship analogous to program execution possesses an intriguing genotype-phenotype mapping in the sense that it can be interpreted as the translation from an encoded version of information to a decoded version of information.

Semantic Closure

The semantic closure is a closed system as found in the von Neumann style self-reproduction, which entails symbols that are used to logically represent a whole “self” capable of reproducing and materials that are required to physically implement such self-reproduction. In discussing this notion, Pattee explicitly makes reference to von Neumann in these respects: firstly, von Neumann was aware that the programmable constructor is a requirement to reach (and exceed) a threshold of complexity in evolution; and secondly, that von Neumann’s models exemplify the self-reproduction that requires primitive components having both symbolic and material functions (e.g., “logic” functions in KA is symbolic, and operations such as cutting and moving in KA are material). Here, Pattee reiterates the “matter-symbol” problem, taking a stance that “all symbolic behaviour must have a material embodiment”, which is clearly different from a view that “symbols are ‘nothing but’ matter”.

The concept of semantic closure is worth considering as it arguably arises in different levels from logic and language to molecules and machines to cognition. It is concerned with the symbolisation that counts as one pervasive characteristic in a biological system, where the distinction between genotype and phenotype matters (De Beule et al., 2010; De Beule, 2011). From this point of view, manipulation of genetic symbols can be regarded as communication between different levels of entities by means of a language. Recent works in the field have endeavoured to introduce von Neumann’s insights and dynamics into the discipline. Thus, although rather crudely, it resonates with the language-game (Wittgenstein, 1953) that argues that meaning emerges nowhere but within a certain framework or context where players (whether explicitly or implicitly) agree to act in accord with the given, governing rules.

Although the molecular biological or biosemiotic aspects neither fall in the immediate scope of the current research nor propose experimental approaches, it is highly relevant in terms of the general framing of the current research. It is relevant firstly because there is the separation between genotype and phenotype which a self-reproducer with von Neumann architecture has. In the typical or traditional study of *coreworld* systems, basically there is no such reflexive symbolic relationship between genotype and phenotype, which might potentially allow complexity growth via evolution of genotype-phenotype mapping. This is why the current research deals with implementation of a von Neumann style self-reproducer in the first place. Secondly, it is because there is the distinction between a program on a memory space and a sequence of numbers (or uninterpreted symbols) on a memory. For example, as described in the following chapters, to code an ancestor in Avida usually means to compose a program by arranging instructions, but each of the instructions can be treated as a number, too. This is also about symbolisation, in that the numbers can be interpreted as instructions in an arbitrary way, either premeditated or spontaneous, in the way that materials (in a broad sense) can be interpreted as symbols. Again, in a typical computing system, the interpretation (the coding from numbers to instructions) may be arbitrary, but not mutable within the system itself. Other layers of interpretation, however, may be mutable, including genotype-phenotype mappings. This kind of mutability, or evolvability, is worth exploring. It might be key to realising open-ended complexity growth, or the "open-endedness" of evolution, in an artificial system. The genotype-phenotype mapping introduced in this section and that introduced in Chapter 3 may be limited examples, as there are generally a wide variety of genotype-phenotype relationships that can be defined in many different levels and domains, with potentially vast space of evolutionary pathways. However, it is still worthwhile to frame particular cases (such as the DNA-RNA-protein world or von Neumann style self-reproduction in Avida) and investigate empirically and analytically.

2.4 Closing Remark

This chapter introduced the core theory by von Neumann and relevant concepts of symbol systems, self-reproduction, and evolution, through which the context of the current research on evolvability of genotype-phenotype mapping using an artificial life platform was provided. On the basis of this, the next two chapters elaborate the investigations on a von Neumann style self-reproducer within the Avida platform. The first investigation (Chapter 3) is from designing to implementing a prototype von Neumann style self-reproducing ancestor in Avida. The second investigation (Chapter 4) is to explore a methodology for analysing evolvability through considering developing a mutation analysis tool.

Chapter 3

Implementation of a von Neumann Style Self-Reproducer

3.1 Overview

This chapter opens by introducing the *Avida* world as a modelling and experimental platform. Then the design of a novel von Neumann style self-reproducing ancestor to be embedded within Avida (the *prototype*) is described. The observation of the prototype behaviour follows as a preliminary step of evolutionary characterisation of such a self-reproducer.

3.2 The *Avida* System

The intention behind the design of *Avida* is to realise an artificial life system where self-reproducing organisms can evolve, achieving some complex features. The Avida system can be conceptually likened to a modern cluster computer, or a collective of micro-controllers, in design. In short, Avida is a cluster of (virtual) micro-controllers, each of which (when executing a suitable program) is analogous to an individual biological organism. The execution of programs on such virtual micro-controllers running in parallel thus emulates a population of organisms. Distinct strains of organism may grow in number and may compete with each other for resources such as CPU time and memory space; and accordingly, the population in the Avida system can exhibit neo-Darwinian evolution. For this reason, Avida has been widely utilised as an evolutionary, adaptive platform (see Bedau, 2003; McKinley et al., 2008, for general reviews). There have been a number of evolutionary, ecological studies using Avida to date, such as Lenski et al. (1999); Ofria et al. (1999); Wilke et al. (2001); Ofria et al. (2002); Adami (2002); Lenski et al. (2003); Misevic et al. (2006, 2010), and more recently: Goldsby et al. (2012); Covert et al. (2013); Hessel & Goings (2013); Knoester et al. (2013); Anderson & Harmon (2014). Some of these use the platform in order to substitute for evolutionary biological experiments which could otherwise be expensive or non-repeatable. The platform can also be seen as a virtual universe which has an ecosystem in and of itself, where some analogies to biology might be drawn. In general, these prior studies are predominantly characterised by the existence of externally specified fitness and the *self-copying* mode of self-reproduction.

Around the time of the inception of Avida (in the early 1990s), one of the goals specified for Avidian organisms was to evolutionarily obtain solutions to particular mathematical problems. It is to achieve this goal that a feature of external fitness function (the concept of which is briefly mentioned in Subsection 2.2.2 in Chapter 2) was originally introduced: organisms are given bonus CPU time from the operating system environment, depending on their achievement of these external mathematical tasks. Every organism has an input and an output port, via which the interaction with the external task environment is performed. If an organism harvests abundant CPU time by succeeding in computation of tasks, then it is able to spend more time to reproduce itself than its competitors can. Completion of the external tasks thus facilitates the reproduction of the organism (by increasing available CPU time), but it is not otherwise relevant to the reproduction process itself. This function of computation of external tasks could be considered analogous to the functionality of the so-called ancillary machinery of a von Neumann style self-reproducer.

Regarding the version of Avida used, the research of this thesis is based on a fork of version 2.10.0, released in 2010.[1] There is another variation of Avida intended for educational purpose, called Avida-ED, with a more intuitive GUI. This is optimised for visualising relatively short organisms and their reproduction process, but with somewhat limited configurability (and without open source code). Avida-ED was not used in the current project.

The following subsections describe the Avida system further.

3.2.1 *Avida* Organism Architecture

An organism in Avida has an architecture of a virtual, abstract computer similar to a micro-controller (or, even to a GPU utilised for general purpose computing), which consists of a combination of a CPU and a memory. See Figure 3.1 for the schematic representation of the organism architecture in the system.

The Avida world is normally configured as a two-dimensional toroidal grid. Nodes of the grid are interconnected, and each node has 8-neighbours typically (Moore neighbourhood).[2] Each node can be either empty or occupied by an organism. The virtual micro-controllers running concurrently are the population of organisms in Avida.

CPU

The CPU comes with registers and stacks, along with control heads, and executes instructions stored in the memory. Program execution thus represents the metabolism of the organism.

[1] The source codes and precompiled binaries of existing versions including 2.10.0 can be found at `http://sourceforge.net/projects/avida/files/`. The current research has been based on this version, mostly because it was the latest stable version when the research within the EvoSym project (mentioned in Subsection 2.3.1 in Chapter 2) commenced, and also because it was not certain how exactly the platform was going to be further developed and modified in subsequent versions. As of 2014, the latest version of Avida is 2.12.4. The fork on which the current research is based consists of some enhanced functionality of Avida, as later described and demonstrated in Chapter 4. It would not be particularly difficult to merge the fork back to the newer versions, as the changes made to the source code were relatively minimal and local (see Section 4.4 in Chapter 4); but extensive debugging and testing would also be required, as there may be unexpected bugs or collisions while doing so.

[2] The neighbourhood can be configured so that each node is connected to all other nodes; in that case, the world topology is changed.

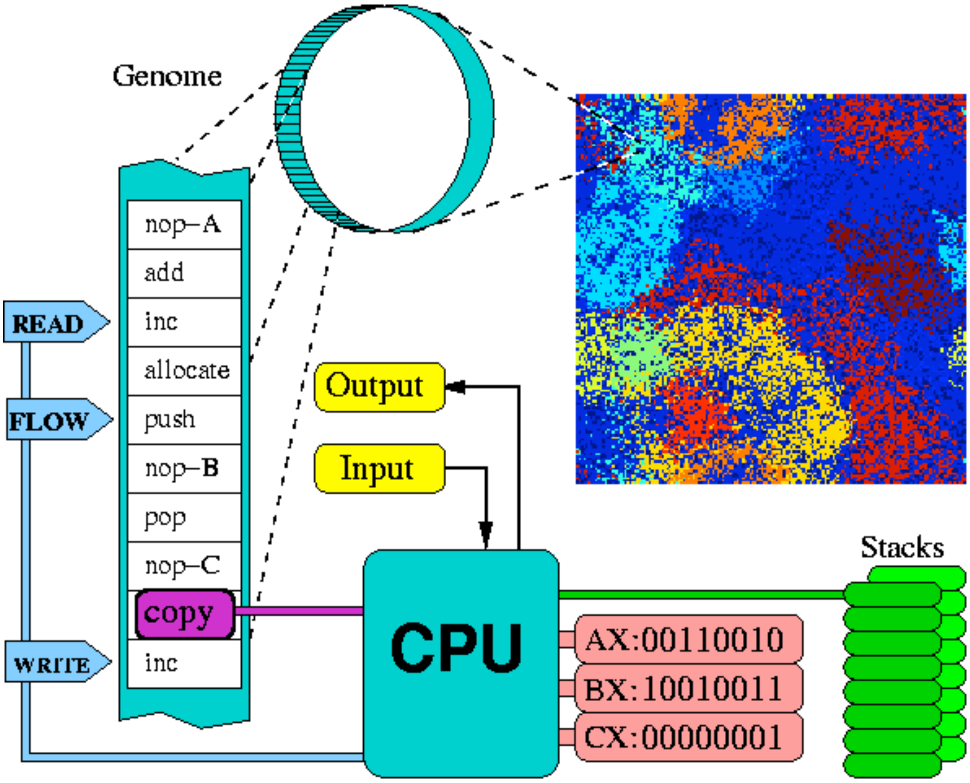


Figure 3.1: Schematic architecture of the organism in the Avida world (as it appears in the documentation distributed along with the version 2.10.0 *Avida* used, the same graphic as the latest one used in `https://github.com/devosoft/avida/wiki/Default-Ancestor-Guided-Tour`). The Input/Output ports are used for interaction with the external environment, but are not used in the current study. “Genome” in this diagram rather refers to memory or memory image in the current context.

There are three registers, namely AX, BX, and CX, which mainly store numbers that are currently manipulated by the CPU, or are arguments for instructions. Some instructions (so-called `nop` instructions such as `nop-A`, `nop-B` and `nop-C`; see Subsection 3.3.3) can serve as modifiers of other instructions (effectively act as *operands*) and vary which registers to use (e.g., to read operand values from or write a result value in).

Along with registers are two complementary stacks (Stack-0 and Stack-1) for storing numbers that are not being immediately manipulated by the CPU. Either of the two is activated at a time (therefore in practice, these work as one stack). Stack-0 is activated first. The active stack is switched to the other by executing the special `swap-stk` instruction (see Subsection 3.3.3). Unlike stacks in the usual sense, the stack in Avida is not so much a component of memory as a storage space inside CPU. The stack in Avida is finite. One stack is fixed and 10 deep, and is not extensible. It is noteworthy that the Avida stack does not have a mechanism to prohibit seemingly abusive manipulations, such as “pop off from an empty stack”, or “push on when the stack is full”. When such manipulations occur, a 0 is popped off from an empty stack, in the first case, or the last word is discarded, in the latter case. (The fact the storage space for the Avida stack is “filled” with 0s is important in the mechanism of a hand-designed self-reproducer described in Section 3.3.)

The control heads include the following: (a) The instruction pointer (IP) points at the location of the instruction to be executed by the CPU; (b) The flow head is used as a reference point to move another head to; (c) The read head and the write head indicate the source of and the destination of memory content that are being handled, respectively; moreover, the read head and the write head specify the range of an offspring’s memory image to be divided off on execution of the special `h-divide` instruction (see Subsection 3.3.3).

Those stacks, registers, and control heads share a basic function: they store numerical values, but for different purposes. While both registers and stacks are used for numbers handled in the process of execution of programs, registers are for direct handling (e.g., providing arguments to instructions or holding a return value from instructions) and stacks are for indirect handling (i.e., to store values for later use). So-called control heads are effectively registers specifically for storing address values.

Memory

The design of the memory in Avida is peculiar (relative to a typical hardware microcontroller) in two respects: (a) the memory can be extended dynamically, without explicit limits, by executing a particular instruction for memory allocation; (b) the memory is circular, so that every possible address maps validly into it. The feature (a) stems from the fact that Avida is not built as hardware in a physical world, even though it virtualises it. A design in a physical setting would require memory to be dynamically added in the hardware. Commonly, the physical memory is sized relative to the address space, so the memory falls within the range of the address space, not the other way around. The feature (b) may not be a necessary one, except that one could argue that such a design would improve robustness against perturbations.

The memory is a crucial entity for an Avidian organism in that it stores a sequence of words (or more primitively, numerical values) which constitutes a *program.* The memory image, or the sequence of words on the memory, is composed of certain Avida *instructions* (and, possibly, interspersed numerical data). The CPU runs such a program, executing the instructions accordingly. We will consider the combination of a memory image and an initial state of the CPU (i.e., a set of values stored in stacks, registers, and working heads) as defining a specific type or class of organism, or a *strain*, as this essentially defines the subsequent dynamic behaviour.

The virtual memory that is dynamically added on demand is reminiscent of the Turing Machine's indefinite tape (which was outlined in Subsection 2.2.2 in Chapter 2). Likewise, the heads working on the memory are reminiscent of the Turing Machine's head, which is supposed to be "attached" to a particular location of the infinitely long (i.e. indefinitely extensible) tape, as opposed to "address registers" which are implicitly fixed in width, or range of possible addressing. This similarity does not actually imply that an Avidian organism has the comparable capability of an ideal Turing Machine (e.g., there will always be an upper limit to the extensible memory in an Avida node, programmatically or physically imposed). Even for a Turing Machine, there is no knowing algorithmically in advance how much tape may be required (related to the halting problem in the Turing Machine context).

In the Avida literature, an initial memory image of an organism (together with an implied default initial CPU state) is referred to as a *genome* or *genotype*. If two organisms have identical memory images (i.e. identical genomes), they belong to the same genotype. On the other hand, in the current study, the term *strain* is used to mean the notion of a memory image representing a particular sequence of words. If two organisms are of the same strain, those organisms have identical memory images (and it is *not* that the organisms share the same memory). The current work regards only a certain delimited segment of a memory image as constituting a genome, in general. So, in this sense, the concepts "genotype" and "genome" are not equivalent in the current work. With respect to the execution of an organism's program, Avida excludes any possibility of syntactic error in program execution: in other words, any sequence of words can be validly interpreted as a sequence of instructions and executed as a program.

3.2.2 Reproduction Mode

When programmed appropriately, an Avidian organism is able to reproduce itself. By default, the standard mode[3] of self-reproduction of organisms is by self-copying (see Figure 3.2 and Figure 3.3). A self-copier will self-reproduce by inspecting its entire memory, one memory location after another, and copying each memory content into its (prospective) offspring's memory in making. When a self-reproducer is a seed program in the Avida world, it is called the *ancestor* (relative to a particular experimental run). Typically, an experimenter would design an ancestor based on the standard self-copier for

[3] The term *reproduction mode* in this thesis is used to mean the way in which an organism reproduces. There can be another term *production mode*, meaning some more general production conducted by an organism, but the former term reproduction mode will be used throughout the thesis, as it is more central to the interest of the research. In the context of the current research, the word *reproduction* connotes, but does not strictly imply, self-reproduction. The concept of self-reproduction can be subtle and vague, so will be revisited in Section 4.2 in Chapter 4. Relevant to this, the word *style* as in von Neumann style or Ray style appears throughout the thesis, and the context in which it is used is basically about self-reproduction, and especially the established architecture of self-reproduction.

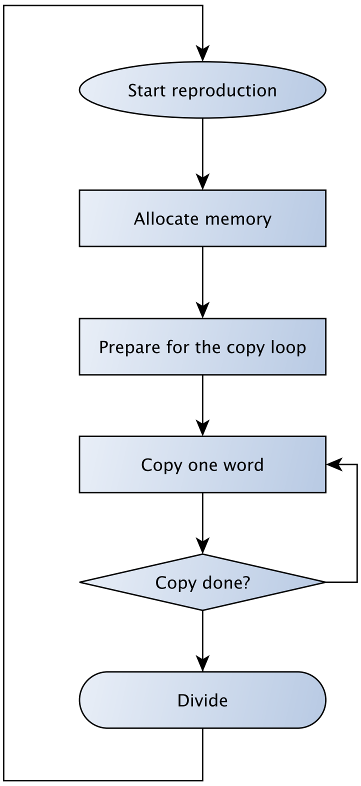


Figure 3.2: The flowchart of how a self-copier (or, a Ray style self-reproducer, after the Tierra creator) self-reproduces.

varied purposes.

With the self-copying reproduction mode, every memory content of a strain (or "genome" in traditional Avida terminology) is copied, thus carrying over any perturbation. That is, any change that has occurred in a self-copier's memory image is inevitably copied into the offspring and thus the changes are inheritable. Such inheritable changes correspond to *mutations* in Avida.

Contrary to the usage in Avida, in this thesis, the term "genome" is used for the most part to mean the component which is complementary to "phenome", both of which correspond only to *parts* of the memory image of a self-reproducing program with the von Neumann architecture. In brief, genome, as a description tape, corresponds to a passive part of the memory image, whereas phenome corresponds to an active part of the memory image, which works on the genome. It follows, in the current case, that a change on a memory image can mean either an inheritable one (i.e., a change in the genome, or in the genotype a particular individual may belong to) or a non-inheritable one (i.e., a change in the phenome, or in the phenotype a particular individual may belong to). In the current context, therefore, only changes that affect the genome (and that are potentially expressed in the phenome) should be counted as *mutations*, for only they are naturally inheritable.

The Avida system works with a number of configuration files. One such configuration file is for the *instruction set*. This set is configured by selecting and enabling a subset as necessary from a wider library of possible instructions which the system incorporates. The instruction set plays an important role as it defines what instructions can be used to code a program of an organism, or can appear throughout a particular experimental run

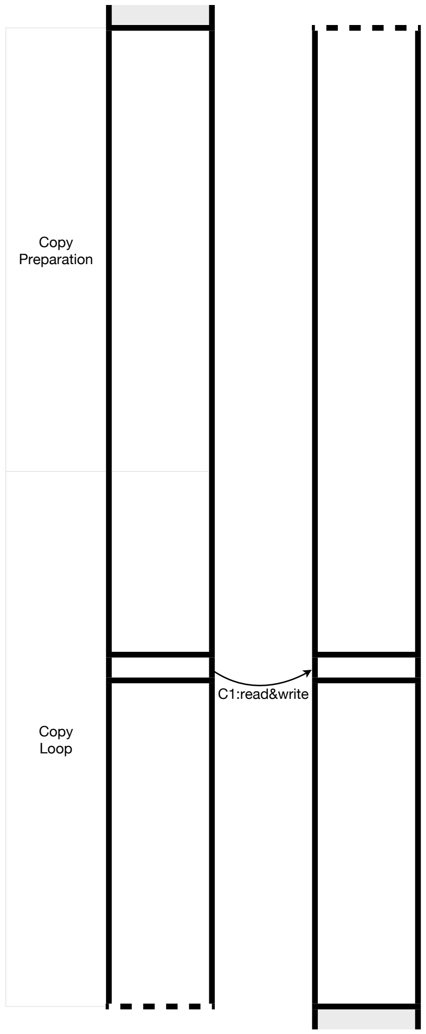


Figure 3.3: Schematic self-reproduction as observed in the standard Avida self-copier. A parent organism has a program on a memory coded with a copy loop. It can construct the memory image of the offspring organism by copying its own memory contents one by one. In the diagram, the arrow in the middle with the label "C1:read&write" ("C" denoting "Copy") signifies that a word at the location the arrow leaves from (in the parent on the left) is read, and written to the location the arrow goes to (in the offspring on the right). The offspring will replicate likewise. In this scheme, any occasional perturbation in the memory image of a parent program (such as a "cosmic-ray" type of error or a carried over change occurred in a previous copy process) will be copied exactly, or *inherited*, hence serving as mutation.

with Avida. As implied, in the face of perturbation events, it is from the range of numbers defined by this set that an number replacing a particular memory location is chosen.

The default instruction set contains 26 instructions. Some instructions are not essential for reproduction, including mathematical operations used for computation done in the interaction with the external task environment. It is not documented whether this instruction set is, or what minimal Avida instruction set is, formally necessary and sufficient for Turing completeness (or, Turing's computation universality), although the system is claimed to support it.[4] Note, in particular, that the default 26-instruction set lacks separate instructions for reading and writing words from and into memory locations, which makes general data manipulation using this instruction set very cumbersome at best. In designing a novel ancestor, this led to additionally enabling the `read` instruction and the `write` instruction. This inclusion of `read` and `write` applies to all instruction sets that are experimentally investigated in the current thesis.

Reproduction Cycle

The most general reproduction process of an organism in the Avida world proceeds as below. The memory allocation and division take place at the beginning and the end of the process, respectively, as conceptually illustrated in Figure 3.4.

1. Firstly, the parent allocates to itself an additional block of memory where the memory image of its (prospective) offspring is constructed;[5]

2. When ready, the parent divides off the offspring's memory image typically on the appended part, with any leftover memory space being discarded;

3. A selected neighbouring node is replaced by the offspring, replacing the organism that was there previously, if any. The location of the node to place the offspring is typically selected based on the age of the organisms currently occupying each neighbouring node, with the oldest being replaced.

To execute the reproduction cycle, each organism needs CPU time, which means that the longer the reproductive program is, the more CPU time the organism needs to self-reproduce. A CPU time slice is allocated to each organism at each update in an Avida run, the value of which is configurable. In the current setting, it is 30 instructions on average per organism per update. With no external fitness function (i.e., no external task environment) imposed, shorter organisms (i.e., with higher net rate of reproduction) are intrinsically favoured by the Avida system.

[4] See, for example, Ofria et al. (2002) or the document at: `https://github.com/devosoft/avida/wiki/Default-Ancestor-Guided-Tour`.

[5] More technically, the system by default extends the memory so that the whole memory length becomes three times as long as the existing memory image. This allows leeway for an offspring's memory image to become up to the double of the parent's. Generally, one can conceive parents whose reproduction gives rise to offspring of varied size. Where there are size-changing perturbation events, such as insertions or deletions, an offspring can also have an increased or decreased size compared to its parent. In the investigation of this chapter, such perturbations have been omitted. Therefore, if any size change is observed, it is because of the way of reproduction that somehow deterministically leads to a different-sized offspring, not because of perturbation. In the main configuration file (namely, `avida.cfg`), there are two variables to set minimum and maximum memory image lengths (namely, `MIN_GENOME_SIZE` and `MAX_GENOME_SIZE`) as shown in Appendix E. In the current configuration, no restraint on the size of legitimate organisms is explicitly set.

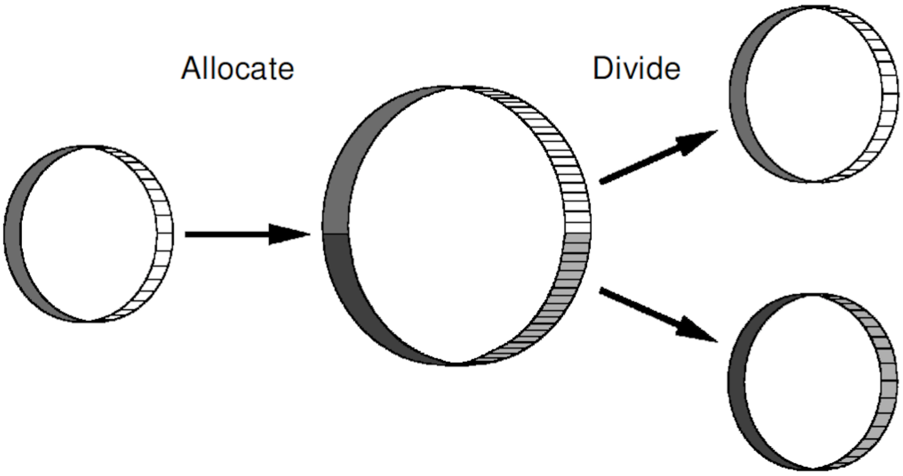


Figure 3.4: Schematic memory allocation and division in Avida (adopted from the Avida introductory paper by Ofria & Wilke, 2004). A circular memory of a parent triples (the multiplier of which is a configurable value; see `CHILD_SIZE_RANGE` in the configuration file shown in Appendix E) as the parent allocates a memory space to create a memory image of its offspring. Once the memory image of the offspring is ready, the parent divides off the offspring. Each memory becomes likewise a part of each organism and each (re-)starts execution accordingly. In general there will be some unused space in allocated memory, which is discarded upon division. Although not shown explicitly, it is possible that the parent becomes shorter or longer, and that the offspring is not the same size as its parent.

3.2.3 *Avida* as a *Coreworld*-type System

It is on the *Avida* system that a von Neumann self-reproducer is implemented as described in the current chapter. The purpose is to study the evolvability of a mutable genotype-phenotype mapping (which may potentially open up greater potential for evolutionary growth of complexity). First, the motivation and origin of the Avida system is briefly provided as an additional background context.

The *Avida* (Adami, 1997; Ofria & Wilke, 2004) artificial life system has been developed since 1993, and is widely used as an experimental platform for evolutionary studies. Avida can allow exponential population growth as part of the neo-Darwinian evolution (Ofria & Wilke, 2004).

Many of the important features of Avida derive indirectly from the system called *Coreworld* (Rasmussen et al., 1990). The Coreworld system itself was inspired by *Core War* (Dewdney, 1984), which was released as a computer game mainly oriented for programmers. The basic idea of the game is that programmers write computer programs that would beat others in the Core War battle. In the game, programs on a single shared memory are regarded as interacting and competing (as opposed to programs on multiple nodes like Avida, where each node has separate, dedicated, memory). These organisms compete with one another for limited resources such as memory space and CPU time. Subsequently, Rasmussen et al. made modifications to the Core War framework to invent the new Coreworld system. They were aware that self-reproduction was crucial for organisms to survive the competition, and that copy errors serve as mutations in the system with the aim of giving rise to evolution. Its objective was to replace the human programmers in Core War by spontaneous natural (or the neo-Darwinian) evolution within the core environment. Contrary to the expectation, in fact, what the system exhibited was anything but an evolving population. This is firstly because there is no memory protection and organisms can easily overwrite one another, and secondly because when there are perturbations, they are likely to make organisms sterile.[6]

The *Tierra* system developed by Ray in the early 1990s in effect overcame the above stumbling block that the Coreworld had hit. Like the Coreworld, Tierra has a single shared memory, in which organisms compete for virtual computational resources. In designing Tierra, however, Ray (1994) circumscribed carefully the system to prevent the fragility of organisms by introducing memory protection, which is done by allocating memory as if each creature has "cellularity". To prevent the population from simply saturating the finite available memory, the *reaper* was introduced to automatically kill organisms so as to accommodate newly born organisms. Not only did the system enable evolving populations, it demonstrated diverse evolutionary phenomena including parasitism. Parasites would successfully keep themselves short and hence be advantageously quick in self-reproducing, by using some or all of the hosts' reproductive code. Once they emerge, "arm races" can begin: the hosts will evolve their defences, whereas the parasites in turn evolve the way to overcome them, and so forth. Chronologically, the Tierra system was to directly inspire the development of the Avida system. The way of labelling memory locations in Avida is, among others, reminiscent of the "template addressing" in Tierra.

Aside from the above systems, the *Amoeba* system (Pargellis, 1996, 2001, 2003) is another more recent Coreworld type system, which made its appearance slightly later than Avida was first released. The Amoeba system is a model specifically designed to study how self-replicating, evolving programs can emerge from a world of non-replicating programs (unlike Tierra and Avida, where the main expectations are to observe the diversity that evolutionarily emerges or the evolution of self-replicators when seeded with an initially designed self-reproducing organism). The initialisation of the Amoeba world is done not by seeding an ancestor but organisms are randomly generated in the soup, where variations emerge and interact. Reproduction is the only task Amoeba organisms have to carry out and there is no external fitness function as can be seen in Avida. Interestingly, the organisms in Amoeba do not necessarily share the same instruction set; whereas in Avida, it is basically fixed for all organisms during an experimental run (though it is configurable per run). Amoeba, with those features, succeeds in facilitating evolution from non-self-reproducers to self-reproducers and in forming colonies of them, although the population does not exhibit further evolutionary dynamics as Tierra does.

Even without introducing the von Neumann architecture of machine self-reproduction, there can be evolution of complexity to some extent, as seen in various artificial life studies using such systems overviewed above. Take Tierra for example: it succeeds in giving rise to diversity in organism functionality through evolution. In Avida's case, it can allow interactions between organisms and the external environment so that they can gain fitness by developing ancillary machinery (e.g., computing ability to gain a more quota of CPU time, acquired unrelated to self-reproduction itself). Those organisms, however,

[6] In relation to such fragility of machines in a computational world, a notion of *autopoiesis* has been proposed as a crucial feature for continuous biological self-reconstruction along with reproduction and evolution. The notion refers to some resilience to preempt the fragility that organisms are subject to, but designing it is not a simple task, as pointed out by McMullin (2004).

do not allow room for the mutation of genotype-phenotype mapping, hence the evolution of genotype-phenotype mapping, since they lack the genotype-phenotype architecture as described by von Neumann.

In such Coreworld type systems, the shorter the reproduction cycle takes, the faster the organism will self-reproduce, and possibly, the more likely that organism will increase in number. In general, the fast reproduction rate is likely to be attributed to the shortness of the size. Suppose that a reproductive cycle is realised by some fixed size of segment of a memory image, and that there is a hypothetical self-reproducing organism of this size. Then, organisms longer than that organism may have non-reproductive or non-functional segments. If that is the case, reproduction cycles of longer organisms would not be affected (hence the organisms would survive) if such non-reproductive or nonfunctional segments are perturbed. In this sense, segments that are not involved in the reproduction cycle may allow room for mutations, and may help give rise to different, possibly more complex, traits. This, naturally, would depend on the kind of the perturbation, and the composition of the memory image and the CPU state that runs on it.

One may argue that any self-reproducers could be regarded as embodying a genotype-phenotype character in a narrower sense. Apparently, their codes can play a twofold role as either the program (which is itself data) or the data input/output by the program (e.g., a genome can be seen as input data, and a memory image of an offspring can be seen as output data). However, not all genotype-phenotype mappings of self-reproducers exist in a mutable way (e.g., genotype-phenotype mappings of self-replicators or self-copiers are hard-wired, even though its usage of memory contents is dual). In this light, the current research is rather concerned with using an ancestor that is purely and explicitly designed with von Neumann's architecture in a manner where the genotype-phenotype mapping is theoretically mutable, and looking at the evolution of such a self-reproducer. Particularly, it is to investigate the minimal effect that such an architecture (or a style of self-reproduction) has on the evolutionary behaviour in Avida.

***Embeddedness*, Individuality, and Interaction**

One of the characteristics of Avida, among other Coreworld systems, is the relationship between the world structure and individuals which reside in the world. When virtually represented, it resembles a typical cellular automaton (CA) as it is a two-dimensional, spatially distributed grid space. The difference is that in Avida each node of the grid represents a location for an organism to occupy, and that once occupied, the node has an extended memory component allocated to it. In other words, the grid space is a collection of pointers that can point to individual organisms. In the space, individual organisms can interact in the sense they compete locally with each other for nodes (or, another memory space as resource in the more general sense). Simply put, Avida is a less "embedded" system, compared to CA (T. J. Taylor, 1999); in Avida, an individual organism is represented as, and occupies, a single node, unlike typical CA where an individual organism (or "embedded machine") is represented as multiple cells working together to achieve some larger scale function. Multiple nodes in Avida rather correspond simply to a population of organisms. An Avida node points to a virtual CPU with a virtual memory representing an organism, which operates according to the relatively complex Avida CPU dynamics;

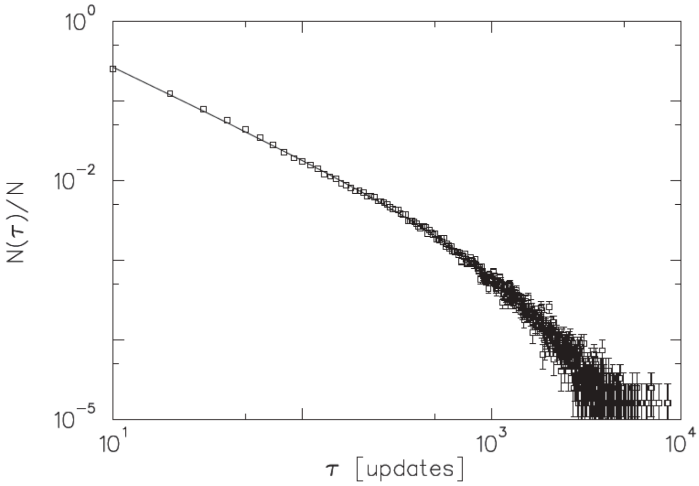


Figure 3.5: Frequency ($N(\tau)/N$) distribution of genotype ages (τ) (adopted from Adami & Brown, 1994). Total of 121,703 “genotypes” have been yielded through 20 runs. The world size is 40x40 and the mutation rate is 0.002.

whereas a cell in CA operates according to a certain relatively simple set of transition rules. Qualitatively speaking, in CA, a relatively simple cell suffices with a small state space compared to a node in Avida; whereas a node in Avida—an individual organism—is not necessarily made to be relatively simple with a small state space. In Avida, a node's memory is extensive (and dynamically extensible), which makes its state space generally much larger than with a fixed small memory; moreover, the size of a single memory location can be larger depending on the data type that is defined in the implementation as the size of a single memory location.

As a contrast to the CA cells again, which can change the states of neighbouring cells according to transition rules, the Avida nodes typically cannot read or write neighbouring nodes,[7] except for when a node is replaced and overwritten by a new-born offspring following division. After division and replacement, the offspring (and likewise, the parent, too) starts executing its program with a reset CPU: register and stack values are all set back to zeros and control heads are positioned back to the beginning address of the memory.

A Re-Creation: Earliest Avida Experiment

One of the earliest experiments in the Avida literature where the system is introduced was by Adami & Brown (1994). The experiment is concerned with the diversity which Avida can bring about from a simple ancestor through the spontaneous evolutionary process. The result of the frequency distribution of genotype ages presented in the original paper is shown in Figure 3.5.

This particular experiment was re-created by the author as a first attempt towards implementing a hand-designed, novel ancestor. It was, especially, in order to have a better grasp as to what a typical Avida experiment looks like, as to how a standard ancestor

[7] Though not particularly relevant in the current research, there are other features that allow interaction of organisms in Avida. One is parasitism as in Tierra, which is configurable in Avida. Also, there is an instruction called `inject` in the library (see Appendix D). Typical division divides off a prospective offspring's memory image and replaces the entire memory image of an adjacent organism with it; whereas *injection*, literally, injects a prospective offspring's memory image into somewhere in the memory image of an adjacent organism (Ofria & Wilke, 2004).

<table>
<tr><th colspan="2">Instruction Sets</th></tr>
<tr><th>Default (New)</th><th>Classic (Old)</th></tr>
<tr><td colspan="2">nop-A</td></tr>
<tr><td colspan="2">nop-B</td></tr>
<tr><td colspan="2">nop-C</td></tr>
<tr><td colspan="2">if-n-equ</td></tr>
<tr><td colspan="2">if-less</td></tr>
<tr><td colspan="2">pop</td></tr>
<tr><td colspan="2">push</td></tr>
<tr><td colspan="2">swap-stk</td></tr>
<tr><td colspan="2">swap</td></tr>
<tr><td colspan="2">shift-r</td></tr>
<tr><td colspan="2">shift-l</td></tr>
<tr><td colspan="2">inc</td></tr>
<tr><td colspan="2">dec</td></tr>
<tr><td colspan="2">add</td></tr>
<tr><td colspan="2">sub</td></tr>
<tr><td colspan="2">nand</td></tr>
<tr><td>h-alloc</td><td>if-bit-1</td></tr>
<tr><td>h-divide</td><td>jump-f</td></tr>
<tr><td>IO</td><td>jump-b</td></tr>
<tr><td>h-copy</td><td>call</td></tr>
<tr><td>h-search</td><td>return</td></tr>
<tr><td>mov-head</td><td>copy</td></tr>
<tr><td>jmp-head</td><td>allocate</td></tr>
<tr><td>get-head</td><td>divide</td></tr>
<tr><td>set-flow</td><td>get</td></tr>
<tr><td>if-label</td><td>put</td></tr>
<tr><td></td><td>search-f</td></tr>
<tr><td></td><td>search-b</td></tr>
</table>

Table 3.1: “Default” (new) and “classic” (old) instruction sets (see Appendix 1 of Ofria & Wilke, 2004, for a more detailed description). 16 instructions are common.

behaves, and as to what instruction set to start designing a new ancestor with.

For this experiment, two minimal, standard ancestors were prepared based on two different instruction sets and they were run. They were expected to yield diverse organisms (or *genotypes*, in the Avida literature) through evolution. The re-creation was made as faithfully to the original setting as possible, except for the instruction set: it was unknown from the description in (Adami & Brown, 1994) precisely which instruction set had been used, the *default* set (new) or the *classic* (old) set, both of which were available in the versions of Avida released around that time.

As shown in Table 3.1, these instruction sets have 16 instructions in common. The codes of simple, standard ancestors coded in these instruction sets (referred to as the “default” ancestor and the “classic” ancestor here, respectively) are listed in Listings 3.1 and 3.2 (codes and comments adopted from Appendix 1 of Ofria & Wilke, 2004). Characteristically, the default instruction set and the “default” ancestor assume the use of control heads, while the classic set and the “classic” ancestor do not. (The instruction set in the context of the current work will be explained in more detail in Section 3.3.)

Listing 3.1 shows how the “default” ancestor self-reproduces with a copy loop. This ancestor starts by allocating memory for a prospective offspring by executing the `h-alloc` instruction (at line 1). It then locates the end of the organism, by executing the `h-search` instruction (at line 2), making use of *labels*. One label is `nop-C` and `nop-A` (located at lines 3 and 4), and the complementary one is `nop-A` and `nop-B` (located at lines 14 and 15). Then, this ancestor places the write head to the start address of the offspring segment, by executing the `mov-head` instruction (at line 5). This instruction has a *modifier*, the `nop-C` instruction (at line 6), which specifies which head to move (in this case, the write head). Next, the `h-search` (at line 7) is executed to mark the start of the copy loop. The `h-copy` instruction (at line 8) is the main instruction in the copy loop: it reads one word from where the read head is located at (by default, at the start of this ancestor's memory image) and writes to where the write head is located at (now at the start of the prospective offspring segment) and moves both of the heads forwards. The copy loop branches by the `if-label` instruction (at line 9). This copy loop is supposed to finish after copying the instructions that constitute the label (`nop-A` and `nop-B` at lines 14 and 15) that marks the end of the segment of this ancestor as a parent. This label is complementary of the label (`nop-C` and `nop-A` at lines 10 and 11) that follows the `if-label` instruction. This is how the `if-label` instruction works: if this is followed by a label corresponding to the most recently copied label, then it executes the next instruction; but otherwise skips it. In this case, if the last two words of this organism are certainly copied, the division takes place by executing the `h-divide` instruction (at line 12); before that, this organism keeps copying words using the copy loop, by executing the `mov-head` instruction (at line 13). For more details about instructions, labels and modifiers, see Subsection 3.3.3 in Section 3.3.

Listing 3.1: “Default” ancestor

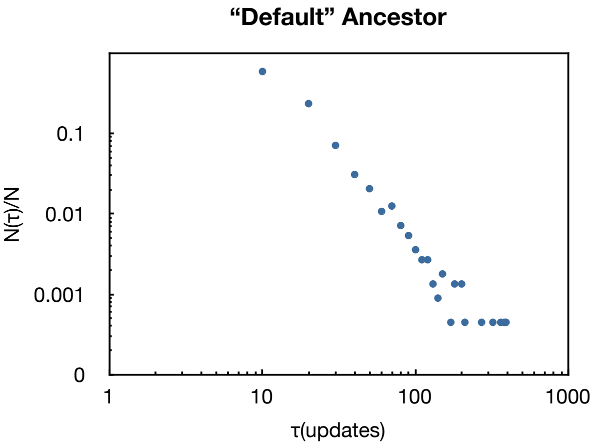
```
 1 h-alloc  # Allocate space for child
 2 h-search # Locate the end of the organism
 3 nop-C # Label alpha
 4 nop-A #
 5 mov-head # Place write head at the beginning of the offspring
 6 nop-C # Nop modifier for mov-head
 7 h-search # Mark the beginning of the copy loop
 8 h-copy   # Do the copy
 9 if-label # If we're done copying...
10 nop-C # Label alpha
11 nop-A #
12 h-divide # ... divide!
13 mov-head # Otherwise, loop back to the beginning of the copy loop
14 nop-A # Label complementary alpha
15 nop-B #
```

Listing 3.2 shows how the “classic” ancestor self-reproduces similarly with a copy loop. This ancestor starts by measuring the distance to the end of its memory image, by executing the `search-f` instruction (at line 1), making use of labels (i.e., one is `nop-A` and `nop-A` at lines 2 and 3, and the complementary one is `nop-B` and `nop-B` at lines 21 and 22). The distance measured does not include the length of the `search-f` instruction and the label, so the `add` and `inc` instructions (at lines 4 and 5) the `search-f` instruction are executed for this organism to obtain the size of its whole segment (22 in this case). Then, the `allocate` instruction (at line 6) allocates the memory for the prospective offspring, setting the *offset* (or the relative start address of the offspring segment, 22 in this case) in AX. The `push` and `nop-C` instructions (at lines 7 and 8), and the `pop` and `nop-C` instructions (at lines 9 and 10) are executed to move the value of the size to CX, signifying how many words to be copied, or the end address of the memory image of this ancestor as a parent. Then, the `pop` instruction sets the value 0 to BX, signifying the "source" location where the (initial) word is copied from. The start and end of the copy loop of this ancestor are marked by another pair of labels (i.e., `nop-B` and `nop-C` at lines 12 and 13, and `nop-A` and `nop-B` at lines 18 and 19). Once the copy loop is entered, the `copy` instruction (at line 14) is executed and copies one word (initially, from the start of the parent segment, at the relative address 0, to the offspring segment, at the relative address 22). The `inc` instruction (at line 15) is executed to move to the next word. The offset is used to specify the location to where a word is copied relative to this source location, so in effect, words are sequentially copied by executing this loop. This copy loop branches using the `in-n-equ` instruction (at line 16); in this particular case, before the current source location is reached at the end of the parent segment, then the loop continues and the next `jump-b` instruction (at line 17) is executed; otherwise it is skipped, and (`nop-A` and `nop-B` at lines 18 and 19 are executed, doing nothing in particular, then) the `divide` instruction (at line 20) is executed. The memory image from the point specified by the offset turns into the offspring's memory image.

Listing 3.2: "Classic" ancestor

```
1  search-f # Find distance to the end label
2  nop-A # Label alpha
3  nop-A #
4  add    # Account for label alpha's size
5  inc    # Account for the initial search-f
6  allocate # Allocate space for the daughter
7  push   # Push size from BX onto the stack
8  nop-B # Nop modifier for push
9  pop    # Pop size off of the stack into CX
10 nop-C # Nop modifier for pop
11 pop    # Since the stack is empty, pop 0 into BX
12 nop-B # Label beta complementary (copy loop start)
13 nop-C #
14 copy   # Copy the current line
15 inc    # Move on to the next line
16 if-n-equ # If we aren't done copying...
17 jump-b   # ... jump back to the loop's beginning
18 nop-A # Label beta
19 nop-B #
20 divide   # Done copying; separate the daughter!
21 nop-B # Label complementary alpha
22 nop-B #
```

As a result of the experiment, almost the same trend was exhibited, except for a peak in the graph for the "classic" ancestor, which the "default" ancestor does not have (see Figure 3.6). Strangely, the classic ancestor, which the original experiment seems to have used, did not produce a closely comparable result, while the default (or newer) ancestor did. Although the cause was not identified, as the Avida setting and configuration are not fully provided in the original literature, it is clear that the choice of the instruction set may have a significant impact on such evolutionary dynamics.



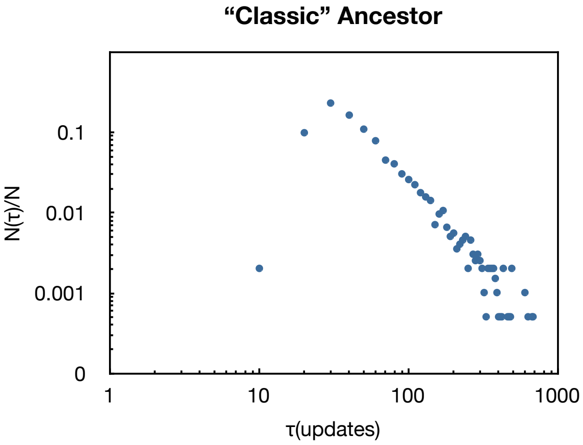


Figure 3.6: Frequency distribution of genotype ages re-created with the ancestors coded with the "default" instruction set (top) and the "classic" instruction set (bottom). Totals of 2,242 and 1,971 "genotypes" have been yielded through 5 runs, respectively. The world size is set to 40 × 40 and the mutation rate 0.002, in the same manner as the original (Adami & Brown, 1994).

3.3 Designing a Prototype Ancestor

As a toy model, a prototype ancestor of the von Neumann style architecture is designed to seed the Avida world. This section describes the design and implementation of this prototype.

3.3.1 *Genome* and *Phenome*

To design a prototype ancestor, the description by McMullin et al. (2001) was followed as a guideline. This is as opposed to the standard self-copying architecture in Avida, which lacks genotype-phenotype distinction, and which presumably leaves no opportunity for a mutable or evolvable genotype-phenotype mapping.

The present hand-designed prototype ancestor has a top-level decomposition into a *genome* and a *phenome*, corresponding to the passive machinery and the active machinery of von Neumann's self-reproductive architecture, respectively. The order in which genotype and phenotype are placed is not essential (hence the order of genome and phenome, the instantiated versions of genotype and phenotype). The decomposition or the order might potentially be re-structured into a different one through evolution. In the present ancestor design, the phenome is placed first, followed by the genome, as execution naturally starts from the beginning of the memory image of an Avida organism by default. It could be the reverse order, with the genome followed by the phenome. Similarly, although procedures for decoding are conducted before those for copying in the present design, the order of sequential activities during reproduction could be different.[8]

The phenome supports the active process of *copying* and *decoding* of the description tape (the genome): from extending the parent memory by allocating memory for creating the memory image of a prospective offspring, to dividing off that created memory image as an offspring's memory image to replace a neighbouring node. In the design of the prototype ancestor, the genome is a part treated exclusively as "data", so it is never to be executed (although it is not enforced by anything in the system); whereas, the phenome not only necessarily contains executable codes, but also incorporates "data" segments as necessary.

One may notice that no explicit "ancillary machinery" is included in this design of the prototype. Having no ancillary machinery means that all the program contributes to the reproductive process directly, each section from one way to another. By doing so, unnecessary complication in the design process was avoided, and one can focus on more essential effects purely from the self-reproductive functionality of the von Neumann style architecture. If any part of the program that does not necessarily engage in the reproductive process emerges through evolution, then that part may be counted as ancillary machinery in von Neumann's architecture.

[8] In biology, the ways genotype and phenotype are organised can vary significantly and be much more complex. For example, it may be the offspring's genotype that is decoded into the offspring's genotype, rather than the parent's genotype. In other words, the genotype is copied first and then, from it, decoded into the phenotype. Buckley (2008) discusses the equivalent idea of whether to start from the daughter or the mother. The current thesis adopts the standpoint that, in terms of von Neumann's design, the programmable constructor can start decoding either from the parent's genotype or the offspring's genotype after copying.

The design of the prototype ancestor's self-reproduction is further explained in the following subsections.

3.3.2 Copying and Decoding

Copying is a part of the self-reproductive process, in which a putative parent organism inspects its genome and copies one memory location after another to a memory location of the memory segment allocated for its prospective offspring. The copy source is read, and the content directly goes to the copy destination, unchanged. The copied part is to be the offspring's genome after division.

Decoding, on the other hand, means the translation process of mapping a genome to a phenome. In the design phase, it was decided to adopt a mapping where the source memory contents (of the genome) are translated sequentially word by word into the destination memory contents of the phenome. Also it was decided to implement the mapping as a *lookup table*. As explained later, the lookup table is embedded in the phenome. This type of mapping was chosen for relative ease of the implementation and the reverse translation of the phenome. More specifically, decoding was designed so that a putative parent reads the content of a memory location, translates it using the lookup table, and writes the resulting content into a memory location of its prospective offspring. The decoded part is to be the offspring's phenome after division.

In the parent, only the relevant parts of the memory image are treated exclusively as data (i.e., the genome is not executed; in the phenome, the lookup table, labels and modifiers are not executed, as explained later). That is, a content of a memory location (or, a word) may be interpreted as an instruction, which is predefined in the form of an *instruction set* in the system configuration setting; whereas, alternatively, it may not be interpreted as an instruction but as uninterpreted data, typically represented as a distinct integer number. In copying, the relevant code in the phenome is executed and copies the genome (treating this segment as data); whereas in decoding, the relevant code in the phenome is executed and decodes the genome accessing the lookup table (treating these segments as data). Only after division does the decoded segment get re-interpreted as a phenome, and thus, some part of it as instructions. Since both the complete genome and phenome can be viewed as (indefinitely large, multi-digit) integers, the prototype's decoding of the genome into the offspring phenome is abstractly a mapping from an integer to another. The initial genotype-phenotype mapping was chosen in a way such that the genome is decoded into the particular phenome which, in turn, is capable of decoding the same genome into the same phenome.

The designed way that the prototype ancestor conducts the self-reproduction can be summed up as follows:

1. allocate the memory for creating the offspring memory image;

2. execute various instructions in the *copy loop* to copy the target genome;

3. execute various instructions in the *decode loop* to decode the genome into the target phenome, referring to the embedded lookup table; and

4. divide off the created memory image as the offspring's memory image and replace some neighbouring node with it.

To realise self-reproduction, the parent's memory image and the offspring's memory image must be identical. In order to achieve this, the designed phenome was reverse-translated (or encoded) into the genome. Thus, the organism with the phenome and the genome, as a parent, is able to self-reproduce an offspring. They are instantiated as two individual organisms of the same strain after division.

3.3.3 Instruction Set

Avida supports an assembly-like language for coding organisms' programs (see Appendix D for the complete instruction library found in the Avida source code). The instruction set is configured arbitrarily by an experimenter run by run, in order to determine a list of instructions that can be used in a single run. It is from the range of numbers defined by this set that resulting words of mutational events are chosen. The instruction set does not vary within a single run.

As each line of the instruction set configuration file sequentially corresponds to a particular natural number (i.e. an index), the instruction set file implicitly provides an association between instructions and numbers (i.e., 0, 1,... N-1; with N being the fixed size of the set). In other words, a specific instruction set implicitly defines the set of word values that are legal on execution (i.e. opcodes). A memory location containing a number larger than the instruction set size is interpreted (on execution) as the remainder of the number when divided by the size. Because of this, the finite instruction set file is capable of associating any memory location value, even beyond the size of the set, to an instruction.

Biological Operations and Added Instructions

For the purpose of implementing the prototype, a subset of the whole collection of instructions defined in the system was chosen and enabled. The 26 instructions included in the default instruction set were enabled as a starting point.[9] Additionally, two further instructions, namely `read` and `write`, were enabled in order to code the decoding process, as mentioned in Subsection 3.2.2. See Table 3.2 for the instruction set used.

Among the instructions in the default instruction set, the `h-copy` instruction plays the memory-to-memory copying role. The prefix `h-` implies that this instruction relies on the control heads; what the `h-copy` instruction does is first to read the value at the memory location which the read head points at and then to write the same value to the memory location which the write head points at. It then moves both of the heads forward (i.e., increment the addresses) so that they point at the (putative) next copy

[9] The classic instruction set (from the first generation of the Avida system) is not generally used by default in later distributed versions, and is not used in the current investigation based on the 2.10.0 version of Avida either. Apparently, the classic set had an impact on the default set design. While some instructions of the classic set are retained in the default instruction set, there are some newly introduced instructions, mostly related to the reproduction that relies on the control heads. With the classic set, reproduction process would use somewhat more primitive instructions, which do not rely on the control heads.

Word Content	Opcode	Operation
0	`nop-A`	No-operations
1	`nop-B`	
2	`nop-C`	
3	`if-n-equ`	Flow Control Operations
4	`if-less`	
5	`if-label`	
6	`mov-head`	
7	`jmp-head`	
8	`get-head`	
9	`set-flow`	
10	`shift-r`	Single Argument Math
11	`shift-l`	
12	`inc`	
13	`dec`	
14	`push`	
15	`pop`	
16	`swap-stk`	
17	`swap`	
18	`add`	Double Argument Math
19	`sub`	
20	`nand`	
21	`h-copy`	Biological Operations
22	`h-alloc`	
23	`h-divide`	
24	`IO`	Input/Output and Sensory
25	`h-search`	
26	`read`	(Additionally Enabled Operations)
27	`write`	

Table 3.2: The instruction set used for the implementation of the prototype. “Operation” depends on the categories in the actual instruction set configuration file. Along with the default standard 26 instructions, two more instructions, namely, the `read` instruction and the `write` instruction, are enabled for the purpose of implementing decoding. These 28 instructions are indexed with numbers from 0 to 27 (the corresponding opcodes).

source and destination locations, respectively. The conventional, standard Avida self-copying ancestor, relies almost exclusively on this instruction to self-reproduce by copying.

Unlike the `h-copy` instruction used for copying, the `read` instruction and the `write` instruction were not enabled in the default instruction set. Technically, the `read` instruction reads a value from a memory location pointed to by a certain register and stores the value in a specific CPU register (e.g., reads a value at the location specified by BX into CX, which can be modified); whereas the `write` instruction writes a value stored in a particular register into a memory location pointed to by the combination of the other two registers (e.g., writes the value in CX into the location specified by AX + BX, the operand of which can be modified).

The `h-divide` instruction is used at the end of the reproduction to divide off the offspring (whose memory image has been created as a part of the parent's memory), overwriting and resetting an adjacent node. The memory image between the read head and the write head at the point when the `h-divide` instruction is executed is regarded as the offspring's memory image.

A self-copying organism using the `h-copy` instruction (like the default ancestor) starts self-replication by copying from the beginning of the (putative) parent memory image, where the read head is initially at, to the beginning of the (prospective) offspring memory image, where the write head is at (set by the `h-search` instruction), which is right after the end of the parent memory image. As it moves the read head and the write head forward step by step, it sequentially copies the original segment of the parent memory image, until the end. By this time, the read head and the write head end up being at the beginning and the end of the offspring memory image, respectively. Then division takes place using the `h-divide` instruction, taking off the image between the read head and the write head, and discarding/de-allocating memory beyond the write head (if there is any).

In terms of implementing decoding, the default 26-instruction set poses two problems with respect to the `h-copy` instruction: (a) the `h-copy` instruction does not allow to access *and* process a read value via general purpose arithmetic or logical instructions, which can operate on register contents or memory contents; (b) the `h-copy` instruction depends on the read head and the write head to specify addresses.

The reason the `h-copy` instruction is not suitable for the implementation of decoding can be more specifically explained as below:

- On the one hand, by using the `mov-head` instruction, it is possible to position the read head, the write head, or the instruction head at a destination memory location where the flow head is set;

- However, in order to set the flow head somewhere, the `set-flow` instruction has to be executed, which uses the value from a certain register, and there is no way to access and process an arbitrary register value;

- On the other hand, by using the `get-head` instruction, it is possible to get the address of the read head, the write head, or the instruction head, and to move the flow-head to that obtained address;

- However, there is no way to position the read head, the write head, or the instruction head at an arbitrary address, for above-mentioned reasons.

Instead of forcibly using the `h-copy` instruction even for the purpose of decoding in a novel ancestor, the `read` instruction and the `write` instruction were introduced to alleviate the problems[10] posed by the restricted functionality of the `h-copy` instruction. The use of these instructions allows to decouple the two more primitive operations that the `h-copy` instruction performs. At the cost of taking some extra steps to perform equivalent operations, now a program can access and process the read content, which is a numerical value that is originally not loaded into the registers or the stacks. While the `h-copy` instruction was not removed from the instruction set, the `read` instruction and the `write` instruction were added to compensate for the difficulty of implementing decoding. Also, enabling these instructions allows some freedom of pointing at arbitrary addresses without the help of the read head and the write head. Now the read and the write heads are used for the operations performed by the `h-copy` instruction and the `h-divide` instruction, but not used for reading a value from, or writing a value in, a certain memory location while decoding. Those heads are still used for copying and for dividing off the offspring at the end of the reproduction. The combination use of the `h-copy` instruction and the `h-divide` instruction for copying can be found in the self-reproduction of the "default" ancestor.

The implemented self-reproduction which requires allocation of memory, copying of a memory content, and division of an offspring organism is implemented based on values loaded in registers such as important addresses and sizes, not necessarily using the control heads. Particularly, one crucial thing incorporated in the design of the novel prototype ancestor is to be able to handle values read in decoding (which influences how the constructor decodes and is therefore important), while also minimising the change that had to be made to the default instruction set.[11] Simply put, using the `h-copy` instruction, the source and destination values are hardwired, whereas with the use of the `read` instruction and the `write` instruction, they are not.

Modifiers and Labels

For the current purpose of implementing the prototype ancestor, notice that there are some instructions that input values from or output to registers, and the register to be used can be changed using a *modifier*.

As a general rule, the registers AX, BX and CX and can be accessed with the no-

[10] As mentioned, computational completeness is assumed in Avida. In the case of Tierra, on which Avida was originally based, a similar question arises, as to whether the instruction set without a `read` or `write` instruction can posit a significant theoretical problem (Maley, 1994). It was found that though the absence of `read` and `write` instructions may cause very significant clumsiness in coding in the language, it does not affect the Turing completeness; but that adding instructions for reading and writing may be beneficial to evolution in Tierra. Although there seems not to be an equivalent study for Avida, presumably the Tierra case is analogous and applicable to some extent. The current investigation relies basically upon the default instruction set consisting of 26 instructions which the Avida developers claim yields computational completeness (see Subsection 3.2.2). In any case, regardless of the extent that computational completeness depends on the choice of the instruction set, from the current point of view it is important that adding the `read` and `write` instructions adds substantial flexibility to the language.

[11] It is a possibility to consider including, for example, the plain `copy` instruction of the classic instruction set. It copies content from a certain memory location to another location specified by values in the registers, and there is a possibility of copy perturbation. In the current instruction set, it was not included, mainly because it was not included in the default set and because a minimal change to the default set was considered. In general, it was not sufficiently clear how to design an instruction set (not to mention how to design it *from scratch*) in order to suit the current purpose of implementing a von Neumann style self-reproducer.

operation (i.e. so-called `nop`) instructions, `nop-A`, `nop-B` and `nop-C`, respectively, when used as modifiers for instructions such as `push` and `pop`. When specified with no modifier, the register BX is used by default. Also, these instructions, `nop-A`, `nop-B` and `nop-C`, have a complementary relationship: `nop-A` is complemented by `nop-B`, `nop-B` by `nop-C`, and `nop-C` by `nop-A`. As for the registers, AX, BX, and CX are complementary, likewise: AX is complemented by BX, BX by CX, and CX by AX. This relationship of registers is relevant when an instruction accepts two arguments; when one register is used for a first argument, its complementary register is used for the second; for example, the `swap` instruction swaps the values in BX and CX by default (or with a modifier `nop-B`), but when modified by `nop-A`, it is between the values in AX and BX, when modified by `nop-C`, it is between CX and AX. In addition, the modifiers `nop-A`, `nop-B` and `nop-C` are associated to heads, IP, the read head and the write head, respectively, and are used with instructions which make use of heads (such as the `mov-head` instruction and the `set-head` instruction).

Labels serve as markers in an Avidian program, representing particular addresses in memory. The usage of labels is supported by the complementary relationship of the `nop` instructions `nop-A`, `nop-B` and `nop-C`. For example, a pair of `nop-A` and `nop-B` in this order makes a label which is complemented by a label of `nop-B` and `nop-C`. As regards coding labels, technically, a designer has to be careful so that no other same sequence as a complementary label should appear between a label and an intended complementary label.

3.3.4 Lookup Table

To implement the decoding process according to the selected initial genotype-phenotype mapping, a *lookup table* approach was adopted so that the genome is decoded sequentially into the phenome on a word-by-word basis, like a substitution cipher in cryptography. The lookup table was designed as a part of the phenome (incorporated towards the end of the segment). The word-by-word mapping as expressed in the lookup table provides building blocks for the translation from genotype into phenotype. Thus, the genotype-phenotype mapping is underpinned by those sequential decoding steps. Though this approach may be intuitively simplistic, it at least provides a convenient starting point that can in principle be extended into a more complex one.

In the decoding process using a lookup table, a read word is treated as a number itself (or a numerical value) as opposed to an instruction; and then the number is interpreted as a relative address (or an index) in the lookup table to look up a target number. Although the choice of the mapping defined by the lookup table is arbitrary, the lookup table in the current case was created as a simple list of all allowed word values, listed in reverse or decreasing numerical order. This order is an arbitrary one of the possible permutations obtained from the set of allowed word values. Each number listed in this lookup table signifies a target (or destination) number for the source number specified by the relative address: the 0th entry in the lookup table is the number that a source number 0 will be translated into, the 1st entry is the number that a source number 1 will be translated into, and so forth.

As there are 28 instructions indexed from 0 to 27, let a list 27..0 (i.e., descending order) be the lookup table. If the word read from a location of the parent genome in the decoding process is 0, the process first looks up what word is at the relative address of 0 (which is the 0th word) in the lookup table, which is 27, and then goes on to write the looked-up word into the corresponding location of the offspring phenome. This means that the value 0 is decoded into the value 27. Likewise, 1 will be decoded as 26, 2 as 25, ..., 26 as 1, and 27 as 0.

Due to the design using a lookup table, there is more arbitrariness to the mapping than the genetic code in real biology. For example, codons are translated into amino acid chains (and further, amino acid chains fold into proteins, according to some physical or chemical constraints and laws as overviewed in Subsection 2.3.2 in Chapter 2). Given codons are defined by a triplet of four kinds of nucleotide, there could be more than sixty logically possible combinations ($64 = 4^3$), although there are actually fewer kinds of amino acids (which is commonly 20, in animals including humans). The current lookup table is loosely inspired by the biological genetic code, but is designed as a one-to-one list of numbers, the same size as the instruction set.

It is important that the word-by-word mapping of a lookup table has invertible and one-to-one correspondence, so that it is possible to "encode" an arbitrary ancestor phenome given a lookup table.[12] The prototype phenome is encoded with the inverse of the lookup table. In practice, the phenome segment was designed first (i.e., the mechanism of copying and decoding, attached with the lookup table), with the genome segment of the same length being a "black box". Then, the fully designed phenome segment was reverse-translated into the corresponding genome segment using the lookup table, so that the phenome and genome pair of a parent can produce the identical organism as an offspring. Simply put, the phenome is the decoded version of the genome, and the genome is the encoded version of the phenome. Of course, encoding only ever takes place *outside* the system, to produce the ancestor; whereas the ancestor itself performs decoding.

In the current design, the lookup table implements an invertible, one-to-one correspondence between all possible genotype words and phenotype words. It follows that the maximum translation repertoire is available in the mapping. Accordingly, there is no possibility for mutation to enhance the translation repertoire beyond that. However, it is possible for mutation to reduce the translation repertoire. If that happens, then the mapping will also necessarily include redundancy in the coding of one or more phenotype words. From this point of view, mutants with mutations expressed in a lookup table are of particular interest. It is noteworthy that in the context of the current study, only one-point mutation is made available, and mutations such as insertion or deletion are excluded, as explained in Section 3.4. This means that mutations could not directly bring about duplication of a mapping.Subsection 4.4.4 in Chapter 4 provides a relevant discussion based on a result from some empirical analysis on such mutants of the prototype introduced in this chapter.

Aside from the specification of the lookup table, it is noticeable that the segment of the genome that encodes the lookup table will always be precisely the sequence of numbers

[12] Though numbers that can be interpreted as instructions are limited by the range determined by an instruction set, this fact does not restrict what numeric value a memory word can contain. Suppose, hypothetically, an ancestor whose memory word contains a numeric value outside the range defined by the instruction set, which is used purely as data (e.g., used functionally, not interpreted as an instruction), then such an ancestor would not be "encodable". That is, not every possible phenome can be encoded. As the design of the prototype does not call for such a data word, this is not *prima facie* obvious.

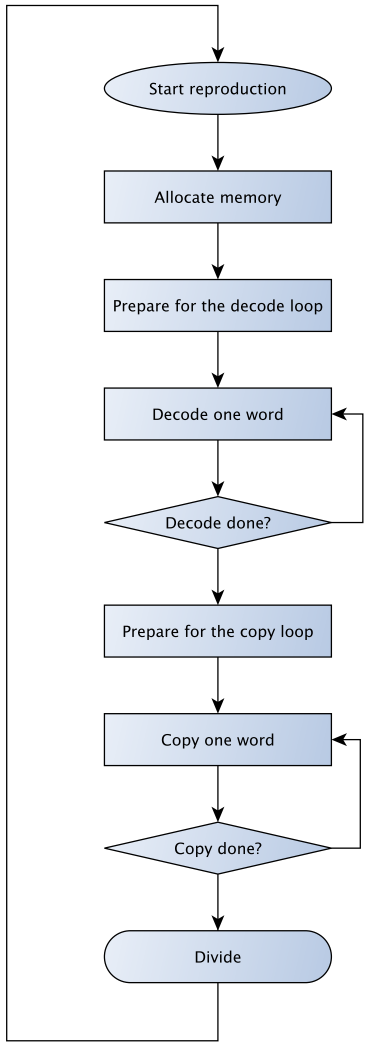


Figure 3.7: Flowchart of how the von Neumann style prototype self-reproducer operates.

from 0 up to N-1 (where N is the size of the lookup table, or the size of the instruction set, in the current case N = 28), with the algorithm of the designed decoding process using a lookup table, regardless of the order of values in the lookup table itself. This is because each number must denote its own relative address in the lookup table.

3.3.5 Self-Reproduction

Equipped with the von Neumann style architecture of self-reproduction, the designed prototype decomposes into the phenome, the first half, and the genome, the second half of the complete memory image (see Figure 3.7 for the flowchart of the self-reproduction by the prototype ancestor, and Figure 3.8 for the prototype's schematic layout in memory).

The phenome can be seen as having five functionally separate segments, namely Decode Preparation, Decode Loop, Copy Preparation, Copy Loop, and Lookup Table (see Table 3.3 for its segment allocation and correspondence, and Appendix B.1 for the entire phenome memory image). In terms of the generic von Neumann architecture introduced earlier, the Decode Loop along with the Lookup Table corresponds to the programmable constructor A, the Copy Loop corresponds to the copier B, and Decode Preparation and Copy Preparation correspond to the control C. The prototype does not incorporate the

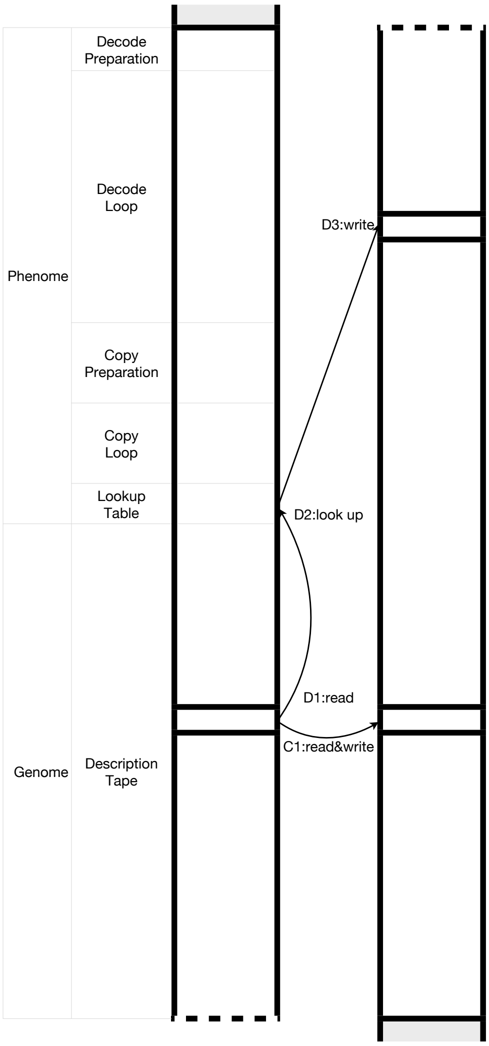


Figure 3.8: Designed self-reproduction of the novel prototype ancestor. To obtain an identical offspring self-reproducer, the parent's phenome (the active part) decodes and copies its genome (the passive part), as the program is executed from the start. The lookup table, a section at the end of the phenome, is referred to when decoding in order to create the offspring's phenome. In the diagram, "D1", "D2" and "D3" denote (high-level) decode steps, and "C1" denotes a (high-level) copy step. The arrow with the labels "D1:read" and "D2:look up" signifies that a word at the location this arrow leaves from is read, and that a word corresponding to the read word is looked up from the lookup table at the location this arrow goes to (both in the parent on the left). The arrow with the labels "D2:look up" and "D3:write" signifies that a resulting word (from the steps above) at the location this arrow leaves from (in the parent on the left) is written to the location this arrow goes to (in the offspring on the right). The arrow with the label "C1:read&write" signifies that a word at the location the arrow leaves from (in the parent on the left) is read, and written to the location the arrow goes to (in the offspring on the right).

Segment	**Address (in Phenome)**	**Code Address (in Genome)**
Decode Preparation	0–27	322–349
Decode Loop	28–193	350–515
Copy Preparation	194–245	516–567
Copy Loop	246–293	568–615
Lookup Table	294–321	616–643

Table 3.3: The five phenotypic functional segments of the prototype and the corresponding genotypic segments (that are encoding these phenotypic functional segments). The Lookup Table segment is only used in the decoding process, not executed like the other four phenotypic segments.

(arbitrary) ancillary machinery D deliberately, presuming it is not essential for reproduction per se (i.e., D can be null, without violating the abstract architecture) and it is not relevant to the immediate investigation of mutations affecting the genotype-phenotype mapping (i.e. the programmable constructor A). It may not necessarily hold that ancillary machinery D can be null, especially if one assumes that there is some interdependency or even inseparability among components (e.g., between $A + B + C$ and D). If such is the case, having no ancillary machinery might affect the reproductive function of $A + B + C$. The stance adopted in the current study, however, is that the components for reproduction, $A + B + C$, and ancillary machinery D are separable enough to the extent that D can be set null. (This assumption may be interpreted as a "reproduction first" model, where only a reproduction mechanism with $A + B + C$ precedes "metabolism", which can be regarded as part of ancillary machinery D.[13])

Once the prototype ancestor is seeded in the Avida world, Decode Preparation and Decode Loop are initially executed and decode the genome to create the offspring's phenome. One step of decoding is as follows:

1. a source word is read from the genome (Decode1),
2. and a target word is looked up via Lookup Table (Decode2)
3. and is written in a corresponding location in the prospective offspring's phenome (Decode3).

Subsequently, Copy Preparation and Copy Loop are executed and copy the genome to create the offspring's genome as follows:

1. a word is read and written at one step (Copy1).

[13] Note that this "reproduction first" concept refers to a discussion in the literature on autopoiesis. The latter generally argues for self-sustaining metabolism as a necessary precondition for reproduction. This is related to, but not identical to, the debate between "replicator first" and "metabolism first" discussions in the broader molecular biology literature.

Then, the final step is division, which is achieved by the `h-divide` instruction. For a complete self-reproduction, it takes 52218 Avidian instruction cycles, or steps. This is the prototype's *gestation time* (i.e. reciprocal of reproduction rate).

The four phenotypic segments except for Lookup Table are the executed part, which is supposed to be executed as pointed at by the IP. The Lookup Table segment is not executed, but is used as a data structure to support decoding. This division of the segments is not a strict one, but rather for the sake of convenience of understanding the structure designed; some instructions do not necessarily belong precisely to one segment or the other.

The subprocess conducted by the active segments of the decode preparation and the decode loop can be summarised as follows (for a line-by-line description, refer to Appendix A).

Decode Phase

The program starts with allocating memory for the creation of the offspring's memory image, which is triggered by the `h-alloc` instruction (see Listing 3.3 for this segment code, and Table 3.4 for the execution trace). The size of the allocation is double the size of the prototype itself. At this point, the whole length of the prototype (= 644) and the label size (= 2 in this case) are collected. The program then starts collecting necessary values for the process, namely the start address of the lookup table. It is located by the `h-search` instruction with the help of the label that follows it, together with the complementary label located immediately before the lookup table. The stack is used as a store to save necessary values throughout. The memory space used for the stack is filled with zeros by default, and here two of these are designed to be used as necessary values, one as a constant zero for comparison which is to be executed later at the end of the decode loop, the other as a counter which is to be incremented to point to a relative destination address in the offspring. Subsequently, the whole length and the label size obtained earlier are stacked (although the label size could have been omitted as it is not going to be used). The read head is positioned at the start address of the lookup table obtained earlier. Next, the start address of the genome is located (again using the `h-search` instruction), and the read head is positioned at the start address of the genome. Then the process calculates the remaining (genome) length to decode (or, a counter to be decremented, to be stored in the stack) by subtracting the value of the genome start address from the value of the whole length. At the end of the decode preparation, the flow head is set to mark the beginning of the decode loop.

Listing 3.3: Decode Preparation

```
h-alloc  # Allocate space for child. # AX:Whole Length (WL)
h-search # Locate the start of Lookup Table (LT). # BX:distance to LT,CX:labelsize
    (1)
nop-A # Label Alpha.Looks for nop-B,nop-C.
nop-B #
push   # Push AX:WL.(Stack-0:WL,c,0,..)(c==0:for comparison)
nop-A #
push   # Push CX:1.(Stack-0:1,WL,c,0,..)
nop-C #
mov-head # Move Read Head to LT.
```

```
nop-B # (Read Head)
get-head # Get CX:LT
nop-B # (of Read Head)
push  # Push CX:LT.(Stack-0:LT,l,WL,c,0,..)
nop-C #
h-search # Locate the start of Genome. # BX:d to G,CX:labelsize
nop-A # Label Beta.Looks for nop-B,nop-A.
nop-C #
mov-head # Move Read Head to G.
nop-B #
get-head # Get CX:G.AX:WL,BX:d to G.
nop-B # (of Read Head)
swap  # AX:d to G,BX:WL,CX:G
nop-A # between AX and BX
sub   # BX:GL (=BX-CX=WL-G) (="Length to Go before Genome")
push  # Push BX:GL.(Stack-0:GL,LT,l,WL,c,0,..)
swap  # AX:d to G,BX:G,CX:GL
push  # Push BX:G.(Stack-0:G,GL,LT,l,WL,c,0,..)
h-search # Set Flow Head to mark the start of the loop. # BX:0,CX:0
```

By now values stored in the stack are ordered as: [genome start address, remaining length, lookup table start address, label size, whole length, constant zero for comparison, 0 (filler), ...] (listed from top). The other side of the stack is not active and not used yet for the first time, but later it is to become: [relative source address in parent genome, relative destination address in offspring, 0 (filler), ...] (listed from top).

The decode loop segment starts by retrieving the genome start address and a relative distance from it (see Listings 3.4 and 3.5 for this segment code and the lookup table, respectively). It is from the source specified by those values that a word (the first word of the genome to be decoded, namely `5`) is read, using the `read` instruction, into the register CX. Now another zero, a filler from the stack, is obtained to be used as a counter to be incremented to point at a relative source address in the genome. At this point this counter is incremented and stacked. Then the program goes on to get the lookup table start address as well as a target word (`22`, meaning the first word of the prospective offspring memory image, decoded from the word `5` read earlier) into the register (BX and CX, respectively). Then, the relative destination address in the prospective offspring memory segment as well as the whole length are retrieved into the registers BX and AX, respectively. It is in the destination pointed at by those values that the decoded word is written. The relative destination address in the offspring here gets incremented and stacked. Then, the constant zero for comparison and the remaining length to decode are retrieved into the registers CX and BX, respectively. Before the comparison and branch, the stack is set back to the starting position for the next loop. Lastly, the comparison is done by executing the `if-n-equ` instruction. It compares the remaining length to zero and if it does not match, the execution is looped back (the IP is moved back to where the flow head is located, by the `mov-head` instruction) to the beginning of the decode loop (see Table 3.5); otherwise (when the remaining length is decremented to zero, after repeating the same decode process to cover the whole genome), it skips and enters the next phase of the self-reproduction process (see Table 3.6). Within the decode loop, the complete parent genome is thus decoded word by word to the offspring phenome.

Listing 3.4: Decode Loop

```
pop    # Get G and d,to read word into CX. Pop BX:G.(Stack-0:GL,LT,l,WL,c,0,..)
push   # Push BX:G.(Stack-0:G,GL,LT,l,WL,c,0,..)
swap-stk # Stack-1 acitive.(Stack-1:d,D,0,..) (D="Destination in daughter" used later)
pop    # Pop CX:d.(Stack-1:D,0,..) (=distance from G)
nop-C #
push   # Push CX:d.(Stack-1:d,D,0,..)
nop-C #
add    # BX:G+d,CX:d
read   # Read one word at BX:G+d,then CX:word.
pop    # BX:d.(Stack-1:D,0,..)
inc    # BX:d++
push   # BX:d.(Stack-1:d,D,0,..)
swap-stk # Get LT at BX so as to read word' (word translated) into CX # Stack-0 active
pop    # Pop BX:G.(Stack-0:GL,LT,l,WL,c,0,..)
swap-stk # Stack-1 active
push   # Push BX:G.(Stack-1:G,d,D,0,..)
swap-stk # Stack-0 active
pop    # BX:GL.(Stack-0:LT,l,WL,c,0,..)
swap-stk # Stack-1 active
push   # Push BX:GL.(Stack-1:GL,G,d,D,0 ..)
swap-stk # Stack-0 active
pop    # Pop BX:LT.(Stack-0:l,WL,c,0,..)
push   # Push BX:LT.(Stack-0:LT,l,WL,c,0,..)
add    # BX=LT+word,CX:word
read   # Read one word at BX:LT+word,then CX:word'
swap-stk # Get D at BX. (Preprocess for write) # Stack-1 active
pop    # Pop BX:GL.(Stack-1:G,d,D,0,..)
swap-stk # Stack-0 active
push   # Push BX:GL.(Stack-0:GL,LT,l,WL,c,0,..)
swap-stk # Stack-1 active
pop    # Pop BX:G.(Stack-1:d,D,0,..)
swap-stk # Stack-0 active
push   # Push BX:G.(Stack-0:G,GL,LT,l,WL,c,0,..)
swap-stk # Stack-1 active
pop    # Pop BX:d.(Stack-1:D,0,..)
swap-stk # Stack-0 active
push   # Push BX:d.(Stack-0:d,G,GL,LT,l,WL,c,0,..)
swap-stk # Stack-1 active
pop    # Pop BX:D.(Stack-1:0,..)
swap-stk # Get WL at AX. (Preprocess for write). # Stack-0 active
pop    # Pop AX:d.(Stack-0:G,GL,LT,l,WL,c,0,..)
nop-A #
swap-stk # Stack-1 active
push   # Push AX:d.(Stack-1:d,0,..)
nop-A #
swap-stk # Stack-0 active
pop    # Pop AX:G.(Stack-0:GL,LT,l,WL,c,0,..)
nop-A #
swap-stk # Stack-1 active
push   # Push AX:G.(Stack-1:G,d,0,..)
nop-A #
swap-stk # Stack-0 active
pop    # Pop AX:GL.(Stack-0:LT,l,WL,c,0,..)
nop-A #
swap-stk # Stack-1 active
push   # Push AX:GL.(Stack-1:GL,G,d,0,..)
nop-A #
swap-stk # Stack-0 active
pop    # Pop AX:LT.(Stack-0:l,WL,c,0,..)
nop-A #
```

```
swap-stk # Stack-1 active
push   # Push AX:LT.(Stack-1:LT,GL,G,d,0,..)
nop-A #
swap-stk # Stack-0 active
pop    # Pop AX:l.(Stack-0:WL,c,0,..)
nop-A #
swap-stk # Stack-1 active
push   # Push AX:l.(Stack-1:l,LT,GL,G,d,0,..)
nop-A #
swap-stk # Stack-0 active
pop    # Pop AX:WL.(Stack-0:c,0,..)
nop-A #
push   # Push AX:WL.(Stack-0:WL,c,0,..)
nop-A #
write # Write CX:word' at AX+BX:WL+D.
inc    # D++.(D="Destination in daughter")
swap-stk # Stack-1 active
pop    # Pop CX:l.(Stack-1:LT,GL,G,d,0,..)
nop-C #
swap-stk # Stack-0 active
push   # Push CX:l.(Stack-0:l,WL,c,0,..)
nop-C #
swap-stk # Stack-1 active
pop    # Pop CX:LT.(Stack-1:GL,G,d,0,..)
nop-C #
swap-stk # Stack-0 active
push   # Push CX:LT.(Stack-0:LT,l,WL,c,0,..)
nop-C #
swap-stk # Stack-1 active
pop    # Pop CX:GL.(Stack-1:G,d,0,..)
nop-C #
swap-stk # Stack-0 active
push   # Push CX:GL.(Stack-0:GL,LT,l,WL,c,0,..)
nop-C #
swap-stk # Stack-1 active
pop    # Pop CX:G.(Stack-1:d,0,..)
nop-C #
swap-stk # Stack-0 active
push   # Push CX:G.(Stack-0:G,GL,LT,l,WL,c,0,..)
nop-C #
swap-stk # Stack-1 active
pop    # Pop CX:d.(Stack-1:0,..)
nop-C #
push   # Push BX:D.(Stack-1:D,0,..)
push   # Push CX:d.(Stack-1:d,D,0,..)
nop-C #
swap-stk # Get c at CX. (Preprocess for comparison) # Stack-0 active
pop    # Pop CX:G.(Stack-0:GL,LT,l,WL,c,0,..)
nop-C #
swap-stk # Stack-1 active
push   # Push CX:G.(Stack-1:G,d,D,0,..)
nop-C #
swap-stk # Stack-0 active
pop    # Pop CX:GL.(Stack-0:LT,l,WL,c,0,..)
nop-C #
swap-stk # Stack-1 active
push   # Push CX:GL.(Stack-1:GL,G,d,D,0,..)
nop-C #
swap-stk # Stack-0 active
pop    # Pop CX:LT.(Stack-0:l,WL,c,0,..)
nop-C #
swap-stk # Stack-1 active
push   # Push CX:LT.(Stack-1:LT,GL,G,d,D,0,..)
```

```
nop-C #
swap-stk # Stack-0 active
pop   # Pop CX:l.(Stack-0:WL,c,0,..)
nop-C #
swap-stk # Stack-1 active
push  # Push CX:l.(Stack-1:l,LT,GL,G,d,D,0,..)
nop-C #
swap-stk # Stack-0 active
pop   # Pop CX:WL.(Stack-0:c,0,..)
nop-C #
swap-stk # Stack-1 active
push  # Push CX:WL.(Stack-1:WL,l,LT,GL,G,d,D,0,..)
nop-C #
swap-stk # Stack-0 active
pop   # Pop CX:c.(Stack-0:0,..)
nop-C #
push  # Push CX:c.(Stack-0:c,0,..)
nop-C #
swap-stk # Get GL at BX (Preprocess for comparison) # Stack-1 active
pop   # Pop BX:WL.(Stack-1:l,LT,GL,G,d,D,0,..)
swap-stk # Stack-0 active
push  # Push BX:WL.(Stack-0:WL,c,0,..)
swap-stk # Stack-1 active
pop   # Pop BX:l.(Stack-1:LT,GL,G,d,D,0,..)
swap-stk # Stack-0 active
push  # Push BX:l.(Stack-0:l,WL,c,0,..)
swap-stk # Stack-1 active
pop   # Pop BX:LT.(Stack-1:GL,G,d,D,0,..)
swap-stk # Stack-0 active
push  # Push BX:LT.(Stack-0:LT,l,WL,c,0,..)
swap-stk # Stack-1 active
pop   # Pop BX:GL.(Stack-1:G,d,D,0,..)
dec   # BX:GL--.(Decrement as one word is read and written)
swap-stk # Stack-0 active
push  # Push BX:GL.(Stack-0:GL,LT,l,WL,c,0,..)
swap-stk # Prepare for the next loop # Stack-1 active
pop   # Pop AX:G.(Stack-1:d,D,0,..)
nop-A #
swap-stk # Stack-0 active
push  # Push AX:GL.(Stack-0:G,GL,LT,l,WL,c,0,..)
nop-A #
if-n-equ # Branch. # Compare BX:GL to CX:c.Do the next and loop back if BX not= CX
    .(while GL>0).
mov-head # If BX=CX.(when GL=0),onto the next phase. # AX:GL,BX:GL,CX:c.
```

Listing 3.5: Lookup Table

```
27 # 0 to 27/write
26 # 1 to 26/read
25 # 2 to 25/h-search
24 # 3 to 24/IO
23 # 4 to 23/h-divide
22 # 5 to 22/h-alloc
21 # 6 to 21/h-copy
20 # 7 to 20/nand
19 # 8 to 19/sub
18 # 9 to 18/add
17 # 10 to 17/swap
16 # 11 to 16/swap-stk
15 # 12 to 15/pop
14 # 13 to 14/push
13 # 14 to 13/dec
12 # 15 to 12/inc
```

```
11 # 16 to 11/shift-l
10 # 17 to 10/shift-r
9  # 18 to 9/set-flow
8  # 19 to 8/get-head
7  # 20 to 7/jmp-head
6  # 21 to 6/mov-head
5  # 22 to 5/if-label
4  # 23 to 4/if-less
3  # 24 to 3/if-n-equ
2  # 25 to 2/nop-C
1  # 26 to 1/nop-B # No other nop-B - nop-A label must exist before this.
0  # 27 to 0/nop-A # Also used as Label complemental Beta.
```

Copy Phase

The copy process is similarly twofold, consisting of a preparation and a loop. The process is summarised as follows.

The copy preparation starts off by newly getting the remaining (genome) length to copy (again as a counter) by subtracting the value of the genome start address from the value of the whole length (see Listing 3.6 for this segment code, and Table 3.7 for the execution trace). Then the process sets the read head at the genome start address, and then the write head at the destination (of the address specified by the sum of the value of the whole length and the value of the genome start address, meaning the offspring's genome start address).

By now values stored in the active side of the stack are ordered as: [whole length, constant zero for comparison, 0 (filler), ...] (listed from top). The other side of the stack should store: [label size, lookup table start address, remaining length, genome start address, relative source address in parent genome, relative destination address in offspring, 0 (filler), ...] (listed from top).

The copy loop starts next (see Listing 3.7 for this segment code). The beginning of the loop is marked, and a word is copied using the `h-copy` instruction. This instruction, as executed, automatically moves the read head and the write head forward by one. Following it, the remaining length to copy as well as the constant zero for comparison are obtained into the registers BX and CX, respectively. Before the comparison and branch, the stack is likewise set back to the starting position for the next loop. The comparison is done by again executing the `if-n-equ` instruction, comparing the remaining length to zero. If it does not match, the execution is looped back to the beginning of the copy loop (see Table 3.8); otherwise (when the remaining length is decremented to zero, after repeating the same copy process to cover the whole genome), it skips and executes the `h-divide` instruction, thereby the offspring memory image is divided off and instantiated as a separate individual organism (see Table 3.9). After division, the parent now has the remaining memory image which is the same as what it started with, and as the offspring has, thus having realised self-reproduction. Again, the CPU states are reset in both organisms after the execution of the `h-copy` instruction, so both will start executing again at the start of their respective memory images. In that sense, it is understood that the starting organism produced two individuals through division, one of which replaces the node occupied by the starting organism, the other of which replaced one of the neighbouring nodes. Within the copy loop, the complete parent genome is copied word by word to the offspring genome.

Time	IP	Content	Registers	Heads	Stack
1	0	h-alloc	[0,0,0]	[0,0,0]	*Stk0[0,...]
2	1	h-search	[644,0,0]	[0,0,0]	*Stk0[0,...]
		nop-A			
		nop-B			
3	4	push	[644,290,2]	[0,0,294]	*Stk0[0,...]
		nop-A			
4	6	push	[644,290,2]	[0,0,294]	*Stk0[644,...]
		nop-C			
5	8	mov-head	[644,290,2]	[0,0,294]	*Stk0[2,...]
		nop-B			
6	10	get-head	[644,290,2]	[294,0,294]	*Stk0[2,...]
		nop-B			
7	12	push	[644,290,294]	[294,0,294]	*Stk0[2,...]
		nop-C			
8	14	h-search	[644,290,294]	[294,0,294]	*Stk0[294,...]
		nop-A			
		nop-C			
9	17	mov-head	[644,305,2]	[294,0,322]	*Stk0[294,...]
		nop-B			
10	19	get-head	[644,305,2]	[322,0,322]	*Stk0[294,...]
		nop-B			
11	21	swap	[644,305,322]	[322,0,322]	*Stk0[294,...]
		nop-A			
12	23	sub	[305,644,322]	[322,0,322]	*Stk0[294,...]
13	24	push	[305,322,322]	[322,0,322]	*Stk0[294,...]
14	25	swap	[305,322,322]	[322,0,322]	*Stk0[322,...]
15	26	push	[305,322,322]	[322,0,322]	*Stk0[322,...]
16	27	h-search	[305,322,322]	[322,0,322]	*Stk0[322,...]
17	28	pop	[305,0,0]	[322,0,28]	*Stk0[322,...]
...	...	...	...	...	...
77	102	write	[644,0,22]	[322,0,28]	*Stk0[644,...]
...	...	...	...	...	...
143	192	if-n-equ	[322,321,0]	[322,0,28]	*Stk0[322,...]
144	193	mov-head	[322,321,0]	[322,0,28]	*Stk0[322,...]
	194	pop			

Table 3.4: Execution trace for the decode preparation until entering the decode loop. The summary of the snapshots of the states (each right before the execution of the instruction of the address pointed at by IP at the time step) is shown on the right. Registers are in the form: [AX value, BX value, CX value]. Heads are in the form: [Read Head, Write Head, Flow Head]. Stack only shows the top value of the stack active at the time step; *Stk0 is the active Stack 0, and *Stk1, the active Stack 1.

Time	IP	Content
145, 273, ..., 40977	28	pop
......	...	...
205, 333, ..., 41037	102	write
......	...	...
271, ..., 40975, 41103	192	if-n-equ
272, ..., 40976, 41104	193	mov-head
	194	pop

Table 3.5: Execution trace of the body of the decoding loop (after the first, before the last execution).

Time	IP	Content	Registers	Heads	Stack
41105	28	pop	[322,1,0]	[322,0,28]	*Stk0[322,...]
...	...	...	...	...	...
41165	102	write			
...	...	...	...	...	...
41231	192	if-n-equ	[322,0,0]	[322,0,28]	*Stk0[322,...]
	193	mov-head			
41232	194	pop	[322,0,0]	[322,0,28]	*Stk0[322,...]

Table 3.6: Execution trace of the end of the decode loop, about to enter the copying process.

Listing 3.6: Copy Preparation

```
pop    # Get a new GL by doing WL-G. # CX:G.(Stack-0:GL,LT,l,WL,c,0,..)
nop-C #
swap-stk # Stack-1 active
push   # CX:G. (Stack-1:G,d,D,0,..)
nop-C #
swap-stk # Stack-0 active
pop    # BX:GL.(Stack-0:LT,l,WL,c,0,..)
pop    # AX:LT.(Stack-0:l,WL,c,0,..)
nop-A #
swap-stk # Stack-1 active
push   # AX:LT.(Stack-1:LT,G,d,D,0,..)
nop-A #
swap-stk # Stack-0 active
pop    # AX:l.(Stack-0:WL,c,0,..)
nop-A #
swap-stk # Stack-1 active
push   # AX:l.(Stack-1:l,LT,G,d,D,0,..)
nop-A #
swap-stk # Stack-0 active
pop    # AX:WL.(Stack-0:c,0,..)
nop-A #
push   # AX:WL.(Stack-0:WL,c,0,..)
nop-A #
swap   # AX:GL,BX:WL,CX:G
nop-A #
sub    # AX:WL-G (=GL)
nop-A #
swap-stk # Stack-1 active
pop    # BX:l.(Stack-1:LT,G,d,D,0,..)
```

```
swap-stk # Stack-0 active
push   # BX:l.(Stack-0:l,WL,c,0,..)
swap-stk # Stack-1 active
pop    # BX:LT.(Stack-1:G,d,D,0,..)
push   # AX:GL. (Stack-1:GL,G,d,D,0,..)
nop-A #
push   # BX:LT.(Stack-1:LT,GL,G,d,D,0,..)
set-flow # Set Read Head at G.AX:GL,BX:LT,CX:G # Set the Flow Head at CX:G.
mov-head # Move the Read Head to G.
nop-B # Read Head
swap-stk # Set Write Head at WL+G. # Stack-0 active
pop    # BX:l.(Stack-0:WL,c,0,..)
swap-stk # Stack-1 active
push   # BX:l.(Stack-1:l,LT,GL,G,d,D,0,..)
swap-stk # Stack-0 active
pop    # BX:WL.(Stack-0:c,0,..)
push   # BX:WL.(Stack-0:WL,c,0,..)
swap   # AX:GL,BX:G,CX:WL
add    # CX:WL+G
nop-C # (Sum into CX)
set-flow # Set the Flow Head at CX:WL+G.
mov-head # Move the Write Head to WL+G where Flow Head is at.
nop-C # (Write Head)
```

Listing 3.7: Copy Loop

```
h-search # Mark the start of the loop.AX:GL(debris),BX:0,CX:0. # *(Stack-0:WL,c,0,..)(Stack-1:l,LT,GL,G,d,D,0,..)
h-copy   # Copy a word from Read Head to Write Head; inc both.
swap-stk # Get GL into BX and decrement. # Stack-1 active
pop    # BX:l.(Stack-1:LT,GL,G,d,D,0,..)
swap-stk # Stack-0 active
push   # BX:l.(Stack-0:l,WL,c,0,..)
swap-stk # Stack-1 active
pop    # BX:LT.(Stack-1:GL,G,d,D,0,..)
swap-stk # Stack-0 active
push   # BX:LT.(Stack-0:LT,l,WL,c,0,..)
swap-stk # Stack-1 active
pop    # BX:GL.(Stack-1:G,d,D,0,..)
dec    # BX:GL-- as a counter
push   # BX:GL.(Stack-1:GL,G,d,D,0,..)
swap-stk # Get c into CX. # Stack-0 active
pop    # CX:LT.(Stack-0:l,WL,c,0,..)
nop-C #
swap-stk # Stack-1 active
push   # CX:LT.(Stack-1:LT,GL,G,d,D,0,..)
nop-C
swap-stk # Stack-0 active
pop    # CX:l.(Stack-0:WL,c,0,..)
nop-C #
swap-stk # Stack-1 active
push   # CX:l.(Stack-1:l,LT,GL,G,d,D,0,..)
nop-C #
swap-stk # Stack-0 active
pop    # CX:WL.(Stack-0:c,0,..)
nop-C #
swap-stk # Stack-1 active
push   # CX:WL.(Stack-1:WL,l,LT,GL,G,d,D,0,..)
nop-C #
swap-stk # Stack-0 active
pop    # CX:c.(Stack-0:0,..)
nop-C #
push   # CX:c.(Stack-0:c,0,..)
```

```
nop-C #
swap-stk # Prepare for the next loop. # Stack-1 active
pop    # AX:WL.(Stack-1:1,LT,GL,G,d,D,0,..)
nop-A #
swap-stk #
push  # AX:WL.(Stack-0:WL,c,0,..)
nop-A #
if-n-equ # Branch. # Compare BX:GL to CX:c.Do the next if BX not= CX.(while GL>0).
    Otherwise skip.
mov-head # Loop back.
h-divide # If BX=CX.(when GL=0),divide.
nop-B # Label complemental Alpha
nop-C # No other nop-B - nop-C must exist before this.
```

Time	IP	Content	Registers	Heads	Stack
41232	194	pop	[322,0,0]	[322,0,28]	*Stk0[322,...]
...	...	...	...	...	...
41268	243	set-flow	[322,322,966]	[322,0,322]	*Stk0[644,...]
41269	244	mov-head	[322,322,966]	[322,0,966]	*Stk0[644,...]
	245	nop-C			
41270	246	h-search	[322,322,966]	[322,966,966]	*Stk0[644,...]
41271	247	h-copy	[322,0,0]	[322,966,247]	*Stk0[644,...]
41272	248	swap-stk	[322,0,0]	[323,967,247]	*Stk0[644,...]
...	...	...	...	...	...
41303	289	if-n-equ	[644,321,0]	[323,967,247]	*Stk0[644,...]
41304	290	mov-head	[644,321,0]	[323,967,247]	*Stk0[644,...]
	291	h-divide			

Table 3.7: Execution trace for the copy preparation until entering the copy loop.

Time	IP	Content
	246	h-search
41305, 41339, ..., 52151	247	h-copy
41306, 41340, ..., 52152	248	swap-stk
......	...	...
41337, ..., 52149, 52183	289	if-n-equ
41338, ..., 52150, 52184	290	mov-head
	291	h-divide

Table 3.8: Execution trace of the body of the copy loop (after the first, before the last execution).

Time	IP	Content	Registers	Heads	Stack
	246	h-search			
52185	247	h-copy	[644,1,0]	[643,1287,247]	*Stk0[644,...]
52186	248	swap-stk	[644,1,0]	[644,1288,247]	*Stk0[644,...]
...	...	...	...	...	...
52217	289	if-n-equ	[644,0,0]	[644,1288,247]	*Stk0[,...]
	290	mov-head			
52218	291	h-divide	[644,0,0]	[644,1288,247]	*Stk0[644,...]

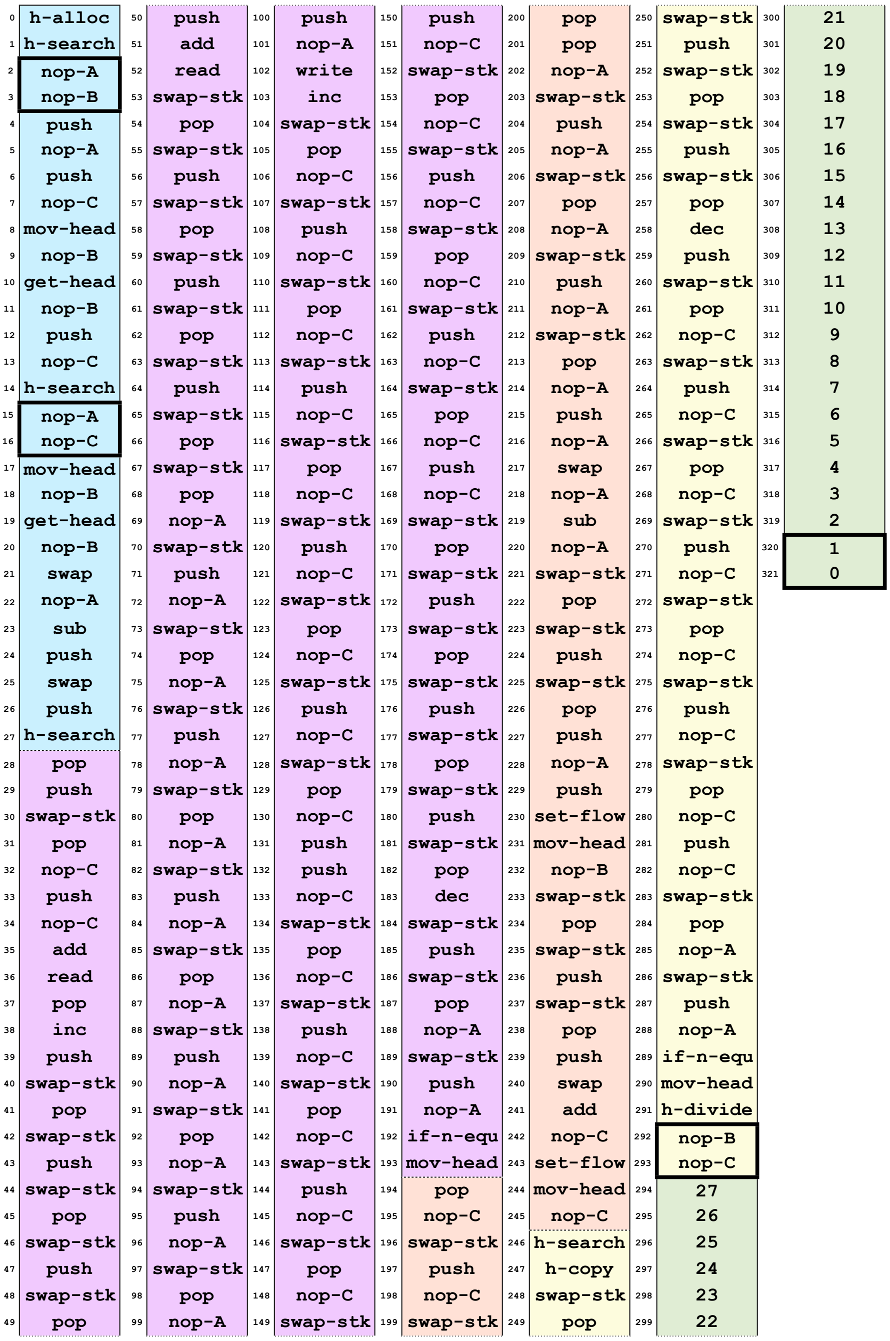
Table 3.9: Execution trace of the final phase of self-reproduction.

Characteristics

The prototype (or even the prototype phenome) is much larger in size and requires more steps to self-reproduce than the default (minimal, self-copying) ancestor (see 3.9 and 3.10 for comparison). More specifically, the designed prototype organism has a total memory image of 644 words (i.e., 322 for each half, phenome and genome), whereas the default ancestor has 15. It is not obvious whether the design of this prototype is minimal or optimal (and whether its being minimal or optimal is desirable); nevertheless, concerning the length of a von Neumann style self-reproducing organism, it necessarily will be longer than a simple self-copier in the first place. This is mostly due to the more complex procedures for self-reproduction. The copy loop of the prototype phenome can be regarded as roughly comparable (or "homologous", one may say in a biological term) to the entire memory image of the default self-copier. Moreover, compared to a (minimal) self-copier, the phenome of the von Neumann style self-reproducing organism needs extra instructions to conduct the decoding and to manage the more complex reproduction cycle (e.g., to judge when to stop the decoding loop and proceed into the copy loop), and to accommodate the lookup table. In addition to this, the combination of the small number of working spaces (i.e., two stacks to store necessary values and three registers to directly handle values) and the primitiveness of the assembly-like language makes the program structure relatively cumbersome. This explains, conversely, why the structure and the operation of working components of the ancestor had to be designed in the first place. In the current case, such a self-reproducer as the prototype is expected to be at least twice as long as a comparable default self-copier. Again, this is because the phenome corresponds to the genome on a sequential, one-to-one basis where one can locate a unique genotypic word corresponding to any given phenotypic word, hence the genome of prototype, corresponding to the phenome, should be as long as the phenome.

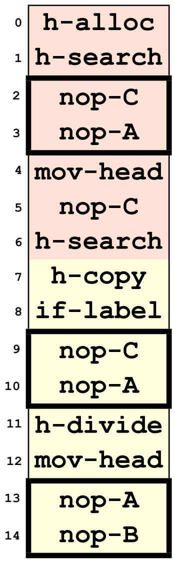
To compare and contrast, the memory images of the prototype phenome and the "default" ancestor are depicted in Figures 3.9 and 3.10. The same colours are used where the copy segment of the prototype and the entire self-reproducing segment of the default ancestor are comparable. The default ancestor has no such genome-phenome distinction as that of the prototype ancestor (i.e., without a genome, or a segment which is exclusively data to be somehow decoded).

Again, this novel program of the prototype relies on the genotype-phenotype mapping,



```
0   h-alloc
1   h-search
2   nop-A
3   nop-B
4   push
5   nop-A
6   push
7   nop-C
8   mov-head
9   nop-B
10  get-head
11  nop-B
12  push
13  nop-C
14  h-search
15  nop-A
16  nop-C
17  mov-head
18  nop-B
19  get-head
20  nop-B
21  swap
22  nop-A
23  sub
24  push
25  swap
26  push
27  h-search
28  pop
29  push
30  swap-stk
31  pop
32  nop-C
33  push
34  nop-C
35  add
36  read
37  pop
38  inc
39  push
40  swap-stk
41  pop
42  swap-stk
43  push
44  swap-stk
45  pop
46  swap-stk
47  push
48  swap-stk
49  pop
50  push
51  add
52  read
53  swap-stk
54  pop
55  swap-stk
56  push
57  swap-stk
58  pop
59  swap-stk
60  push
61  swap-stk
62  pop
63  swap-stk
64  push
65  swap-stk
66  pop
67  swap-stk
68  pop
69  nop-A
70  swap-stk
71  push
72  nop-A
73  swap-stk
74  pop
75  nop-A
76  swap-stk
77  push
78  nop-A
79  swap-stk
80  pop
81  nop-A
82  swap-stk
83  push
84  nop-A
85  swap-stk
86  pop
87  nop-A
88  swap-stk
89  push
90  nop-A
91  swap-stk
92  pop
93  nop-A
94  swap-stk
95  push
96  nop-A
97  swap-stk
98  pop
99  nop-A
100 push
101 nop-A
102 write
103 inc
104 swap-stk
105 pop
106 nop-C
107 swap-stk
108 push
109 nop-C
110 swap-stk
111 pop
112 nop-C
113 swap-stk
114 push
115 nop-C
116 swap-stk
117 pop
118 nop-C
119 swap-stk
120 push
121 nop-C
122 swap-stk
123 pop
124 nop-C
125 swap-stk
126 push
127 nop-C
128 swap-stk
129 pop
130 nop-C
131 push
132 push
133 nop-C
134 swap-stk
135 pop
136 nop-C
137 swap-stk
138 push
139 nop-C
140 swap-stk
141 pop
142 nop-C
143 swap-stk
144 push
145 nop-C
146 swap-stk
147 pop
148 nop-C
149 swap-stk
150 push
151 nop-C
152 swap-stk
153 pop
154 nop-C
155 swap-stk
156 push
157 nop-C
158 swap-stk
159 pop
160 nop-C
161 swap-stk
162 push
163 nop-C
164 swap-stk
165 pop
166 nop-C
167 push
168 nop-C
169 swap-stk
170 pop
171 swap-stk
172 push
173 swap-stk
174 pop
175 swap-stk
176 push
177 swap-stk
178 pop
179 swap-stk
180 push
181 swap-stk
182 pop
183 dec
184 swap-stk
185 push
186 swap-stk
187 pop
188 nop-A
189 swap-stk
190 push
191 nop-A
192 if-n-equ
193 mov-head
194 pop
195 nop-C
196 swap-stk
197 push
198 nop-C
199 swap-stk
200 pop
201 pop
202 nop-A
203 swap-stk
204 push
205 nop-A
206 swap-stk
207 pop
208 nop-A
209 swap-stk
210 push
211 nop-A
212 swap-stk
213 pop
214 nop-A
215 push
216 nop-A
217 swap
218 nop-A
219 sub
220 nop-A
221 swap-stk
222 pop
223 swap-stk
224 push
225 swap-stk
226 pop
227 push
228 nop-A
229 push
230 set-flow
231 mov-head
232 nop-B
233 swap-stk
234 pop
235 swap-stk
236 push
237 swap-stk
238 pop
239 push
240 swap
241 add
242 nop-C
243 set-flow
244 mov-head
245 nop-C
246 h-search
247 h-copy
248 swap-stk
249 pop
250 swap-stk
251 push
252 swap-stk
253 pop
254 swap-stk
255 push
256 swap-stk
257 pop
258 dec
259 push
260 swap-stk
261 pop
262 nop-C
263 swap-stk
264 push
265 nop-C
266 swap-stk
267 pop
268 nop-C
269 swap-stk
270 push
271 nop-C
272 swap-stk
273 pop
274 nop-C
275 swap-stk
276 push
277 nop-C
278 swap-stk
279 pop
280 nop-C
281 push
282 nop-C
283 swap-stk
284 pop
285 nop-A
286 swap-stk
287 push
288 nop-A
289 if-n-equ
290 mov-head
291 h-divide
292 nop-B
293 nop-C
294 27
295 26
296 25
297 24
298 23
299 22
300 21
301 20
302 19
303 18
304 17
305 16
306 15
307 14
308 13
309 12
310 11
311 10
312 9
313 8
314 7
315 6
316 5
317 4
318 3
319 2
320 1
321 0
```

Figure 3.9: The memory image of the prototype's phenome. The corresponding genome follows but it is not shown. The copy segment (the copy preparation and the copy loop) is comparable to the entire segment of the "default" ancestor (the standard self-copier coded via the default instruction set). The five segments are colour-coded. Framed sections represents the segments that serve as labels.



0	h-alloc
1	h-search
2	nop-C
3	nop-A
4	mov-head
5	nop-C
6	h-search
7	h-copy
8	if-label
9	nop-C
10	nop-A
11	h-divide
12	mov-head
13	nop-A
14	nop-B

Figure 3.10: The memory image of the "default" ancestor. The entire segment of this self-copier is comparable to the copy segment of the prototype's phenome. The two segments of the copy segment are colour coded in comparison to that of the prototype's phenome.

which is analogous to the mapping from a string of words (i.e. genome) to another string of words (i.e. phenome) underlain by the mapping between individual words that is defined by the lookup table. Importantly to the scope of the current research, it is at least possible that a mutant of the prototype turns out to be self-reproducing with a mutated genotype-phenotype mapping. The possibility hinges on whether the genotype-phenotype mapping gets mutated in a "backward compatible" manner. If a mutation only affects the translation of an instruction (for example, one that is not executed by the program of the prototype), the genotype-phenotype mapping may remain the same to the particular set of genome and phenome (i.e., backward compatible).

3.4 Observing the Prototype Behaviour

First and foremost, when seeded in Avida with no perturbation or mutation enabled, the prototype succeeded in self-reproducing. (In summary, perturbation here means a change within a memory image in general, whereas mutation refers to an inheritable one. The difference between these concepts is explained in more detail in Subsection 3.4.1.) In other words, an organism with the decomposition into genome and phenome (i.e., incorporating a genotype-phenotype mapping) decodes and copies its description as designed so as to reproduce itself. Thereby, it is demonstrated that a von Neumann style ancestor can be instantiated in this system.

Regarding evolutionary behaviour, one naïve expectation was that the prototype ancestor at least would be able to give rise to a "traditional" Avidian evolutionary process. More specifically, it was expected that the ancestor could self-reproduce reliably enough to increase in population under appropriate stochastic effects, such as perturbations that occur with certain rates and the way that a new offspring replaces the neighbouring nodes. Following an exponential growth of population, there would be a total population limit as imposed by the size of the Avida two-dimensional world.

It was expected that mutants (or, strains of descendant organisms carrying some accumulation of mutations) that are capable of breeding true while still retaining von Neumann's architecture may emerge (see Subsection 2.2.1 in Chapter 2). Such mutants may undergo selection in evolution according to their style and ability of self-reproduction (differently than the standard ancestor). Specific strains may or may not be able to survive in the population over generations.

Since the central interest of implementing the prototype is in characterising the mutational pathways where the genotype-phenotype mapping changes that it can give rise to when seeded as an instance of a von Neumann style self-reproducing ancestor, the main, immediate focus was put on what mutants would become dominant in population through an evolutionary run. Avida was seeded with this instantiated prototype ancestor.

3.4.1 Evolutionary Experiment using *Avida*

Evolutionary dynamics observed in Avida experiments is essentially affected by configurable variables. The configuration settings relevant from the evolutionary perspective are explained below, followed by the discussion on the concepts of mutation and perturbation within the current study.

Configuration

The Avida configuration allows one to set up variables for normal experiments mainly via the `avida.cfg` file. Excerpts of relevant groups of configurable variables from that file are shown in Appendix E.

There are variables related to the general Avida world attributes. The world size is a relevant attribute as it determines the carrying capacity of the world. This size can be set by specifying the width and height of the world (`WORLD_X` and `WORLD_Y`, respectively, in `ARCH_GROUP`). The size of an *update* is another relevant one. An update is a unit of time within an Avida evolutionary run, expressed in CPU steps (or, the number of instructions executed) per organism. This size is configurable as well (`AVE_TIME_SLICE` in `TIME_GROUP`).

For typical evolutionary experiments, variables related to reproduction (`REPRODUCTION_GROUP`) and division (`DIVIDE_GROUP`) are significant, a few relevant ones of which are shown below.

- `REPRODUCTION_GROUP`
  - `BIRTH_METHOD`: which neighbour to replace with an offspring.
  - `DEATH_METHOD`: whether to die of old age.
- `DIVIDE_GROUP`
  - `CHILD_SIZE_RANGE`: the maximum differential of an offspring relative to a parent.
  - `MIN_COPIED_LINES`: the minimum code fraction to be executed before division.

Likewise, variables related to mutation are important for typical evolutionary experiments. In the current study, the relevant variable is the permutation rate per write operation (COPY_MUT_PROB in MUTATION_GROUP), the rationale behind which is explained next.

Mutation versus Perturbation

Mutation is an important factor for the neo-Darwinian evolution to occur. Here, the concepts of mutation and perturbation are further clarified in the light of the current context.

In order to observe the evolutionary behaviour of the prototype, Avida was deliberately configured to allow only "single-point perturbations" to occur. Although there are other types of perturbation, including multi-point, insertion and deletion, they were omitted in this first instance for simplicity and traceability of mutational pathways; otherwise, it would be impractical to scrutinise the effect of a single mutational change. This specifically means that if the size of organism changes, it is likely to be due to a new reproduction mode that produces offspring of varied size, and not due to such a size-changing perturbation.

A single-point mutation, in the current context, normally means any spontaneous change of a single memory location within the genome. A change in a memory location within the phenome will not normally be inherited and therefore is not described as mutation. By contrast, if the standard (or "default") ancestor in Avida, which is a self-copier, undergoes any changes anywhere in its memory image, the changes will be all potentially inheritable and are thus referred to as mutation. On the contrary, in the current implementation of von Neumann's architecture of machine self-reproduction, not all changes in the memory image, but only those in the *genome* part are expected to be inheritable. Perturbations in phenome are by definition not mutations, since they will not be inherited.

It is noticeable that a mutation in this architecture of self-reproduction will exhibit delayed expression compared to the case of a self-copying architecture (see Figure 3.11 for a high-level schematic diagram of how a mutant is reproduced). Now that there is only a single-point mutation in the Avida world, once a mutational change in the genome occurs, it is one generation later that the change will be decoded into an offspring's phenome. That is, one generation is needed for a single-point copy mutation to occur (i.e., for a parent organism to reproduce an offspring), and then another generation is needed for it to get expressed (i.e., for the offspring to reproduce its offspring). This delay, however, does not apply in the case of the standard ancestor because when a mutation occurs, it is inherited and expressed immediately in the offspring. Likewise, such a delay may not occur if there are other ways available to occasionally perturb the memory content (e.g., "cosmic-ray" type of perturbations in the memory image, decoding errors, etc.), because changes made in the phenome normally end up being temporary.

Even with a mutation expressed in the phenome (as opposed to a mutation *inherited* in the genome), the organism can be fertile. Intuitively, the expression of a mutation is most likely to affect the reproductive process, because all of the prototype's phenome is concerned with self-reproduction (with nothing corresponding to ancillary machinery D). It is still conceivable, however, that somehow the reproductive functions work and succeed in producing an offspring organism that breeds true (i.e., inheriting the mutation as well as retaining the self-reproductive functions). It should be emphasised again that the current research is concerned with exploring the possibility that there might be a mutated

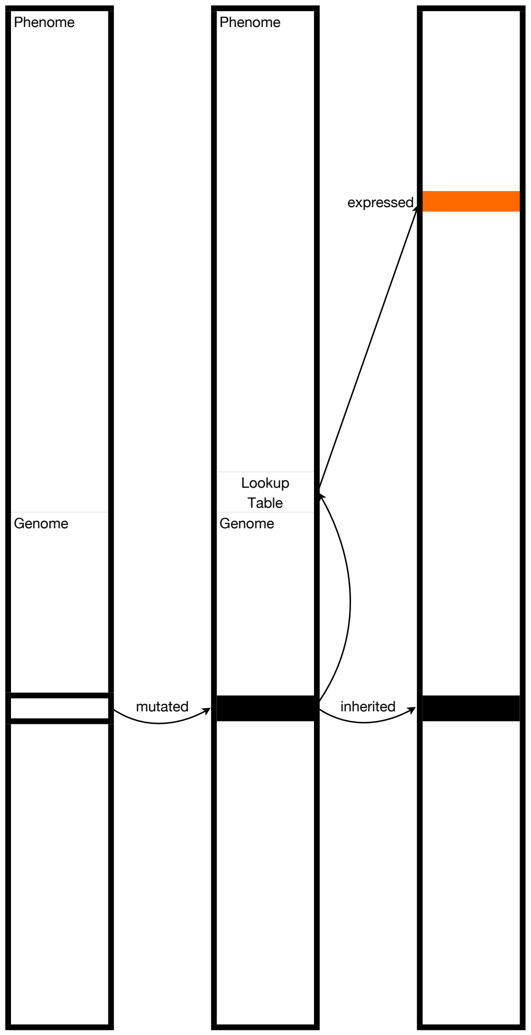


Figure 3.11: The mechanism of how a mutant is reproduced in the case of a von Neumann style self-reproducer such as the prototype (Generation 0, Generation 1, and Generation 2, from left to right). The mutated prototype (parent) reproduces the offspring with the mutation expressed. The mutation inherited and that expressed are located at corresponding relative addresses in the phenome and the genome respectively.

descendant (or *mutant*) with some von Neumann style self-reproducibility, even though the expressed mutation somehow has disturbed the decoding mechanism (corresponding to the programmable constructor A in the von Neumann architecture). In the current case, either the mechanism of decoding itself or the word-by-word mapping defined in the lookup table may be modified, while retaining a von Neumann style self-reproduction. Conceivably, the von Neumann style self-reproducibility may be lost and replaced with a different self-reproduction style, where mutation is inheritable. There are possibilities that some other part is disturbed (either the copier B or the controller C in the current case), but it is more likely to lead to an offspring which is sterile or not breeding true.

By default in Avida, the `h-copy` instruction and the `write` instruction, among the instructions presented in the instruction set configured, were the only ones that can trigger "single-point perturbations" whenever they are executed. For that reason, Avida per se cannot technically distinguish a perturbation in phenotype (i.e., such perturbations as caused by the `write` instruction in decoding) and that in genotype (i.e., such perturbations as caused by the `h-copy` instruction in copying). What the current research is interested in is inheritable mutations. So, only the cases with a changed genotype had to be manually picked up after Avida runs, in order to single out the mutation in the sense of the von Neumann architecture where it is not the phenotype (i.e., decoded entity) but the genotype (i.e., copied entity) that undergoes mutations.

With regard to a mutation rate, generally speaking, the value of the rate should be reasonably low so that a spontaneous neo-Darwinian evolution should occur; but the rate should not be extremely high so that Malthusian population growth of a strain is ensured (if the strain is not perturbed). In the current context, the rate of perturbation (`COPY_MUT_PROB`) needs to be adjusted according to the organism length, so that the probability for an organism to have at least one perturbation per reproduction cycle is approximately equal between the prototype (or designed) ancestor and the standard (or "default") ancestor. That is, if the perturbation rate per write operation is R_s for the standard ancestor, the probability that there is no perturbation in a location is $1 - R_s$ and the probability that the ancestor goes without mutation is $(1 - R_s)^{l_s}$, where l_s is the organism length. Therefore, the probability that the standard ancestor has at least one perturbation is $1 - (1 - R_s)^{l_s}$. By the same token, the probability that the prototype ancestor has at least one perturbation is $1 - (1 - R_p)^{l_p}$, where R_p is the perturbation rate per write operation for the prototype ancestor and l_p is the length of the prototype ancestor. The rate is not intended to be exact, but at least to be reasonably equal for the prototype ancestor compared to the standard ancestor.

3.4.2 Observation and Examination

Within the Avida framework explained above, a small scale experiment of 30 runs was conducted. Each run was seeded with the prototype ancestor and set to last for 50000 updates. The world size was set as 60×60 (`WORLD_X` and `WORLD_Y` are both set to be 60) so that the carrying capacity is 3600. One update was configured so that it amounts to 30 CPU time steps or instructions to be executed. That means 1,500,000 CPU time steps are available within a run, approximately 29 times as long as the gestation time of the prototype, 52218. If a population grows exponentially, the run is long enough to fill up the world (in practice, however, there is some spatial effect as an organism can place an offspring in the neighbourhood only).

As regards reproduction, birth and death methods (`BIRTH_METHOD` and `DEATH_METHOD`, respectively) were chosen so that an offspring replaces the oldest in the neighbourhood and that an organism never dies of old age. As for division, `CHILD_SIZE_RANGE` was set to be 2.0, meaning that the size of memory allocated by `h-alloc` is the double of the size of the parent (which is as mentioned in 3.2). `MIN_COPIED_LINES` was set to be 0.45, considering the ratio of the prototype genome to the whole length ($294 \div 644 \approx 0.45$).

The variable `COPY_MUT_PROB` was set to be 0.0001753. To calculate this rate of perturbation per write operation R_p so that these ancestors have the same probability of having at least one perturbation, the default values for the standard ancestor were considered. For the standard ancestor $R_s = 0.0075$ and $l_s = 15$; so for the prototype ancestor ($l_p = 644$) to have the same probability, it should be $R_p \approx 0.0001753$. The whole length of the prototype ancestor (not half the length that is to be copied) was used as l_p knowing that in the current configuration, both the `h-copy` instruction (used in the copying) and the `write` instruction (used in the decoding) can trigger the perturbation. The probability of no perturbation is 0.8932 ($\approx (1 - 0.0075)^{15}$ for the default ancestor, $\approx (1 - 0.0001753)^{644}$ for the prototype).

The configuration explained above can be summarised as follows in the order that it appears in the `avida.cfg` file.

- `ARCH_GROUP`
  - `WORLD_X`: 60
  - `WORLD_Y`: 60
- `MUTATION_GROUP`
  - `COPY_MUT_PROB`: 0.0001753
- `REPRODUCTION_GROUP`
  - `BIRTH_METHOD`: an offspring replaces the oldest in the neighbourhood.
  - `DEATH_METHOD`: an organism never dies of old age.
- `DIVIDE_GROUP`
  - `CHILD_SIZE_RANGE`: 2.0
  - `MIN_COPIED_LINES`: 0.45
- `TIME_GROUP`
  - `AVE_TIME_SLICE`: 30

For a more comprehensive list of configuration variables used in the current study, see Appendix E.

Dominance Shift by a Mutant Strain

Firstly, such runs where dominance shift by a mutant strain occurred were singled out. Dominance shift by a mutant refers to a situation where a population (which starts as a population of organisms of the prototype strain) gets dominated or outnumbered by, or even displaced by, organisms of any mutant strain at a later stage. Through run statistics (such as population snapshots, dominant strain snapshots, and sequence data of strains), only mutants (i.e. those carrying a perturbation in the genome) which became dominant were identified and picked up among all those which underwent perturbations regardless of the segments. Runs with dominance shift only by a non-mutant were ignored as being irrelevant. Out of 30, there were 6 runs where dominance shift by a mutant took place; in 4 runs, dominance shift took place but only by a non-mutant; in 20 runs, there was no dominance shift (the prototype remained dominant).

Next, to better understand the mechanism of dominance shift by a mutant, one of the 6 runs was arbitrarily picked out and examined. (The results from the other 5 runs are not described here, but briefly reflected in Subsection 3.5.1.) On a closer look into the run, it turned out that a descendant perturbed in the genome (or a mutant) and a descendant perturbed in the phenome (or a non-mutant), following the ancestor, became dominant (see Figure 3.12). Within the 50000-update run, these strains alternately became dominant two times each.

The observed dynamics of the dominant mutant can be summarised as follows.

1. The prototype ancestor starts self-reproduction and as a result the population of the organisms of this strain grows.

2. A single-point perturbation perturbs the strain of one of the organisms (so the offspring strain is different by one word at a particular memory location in the genome).

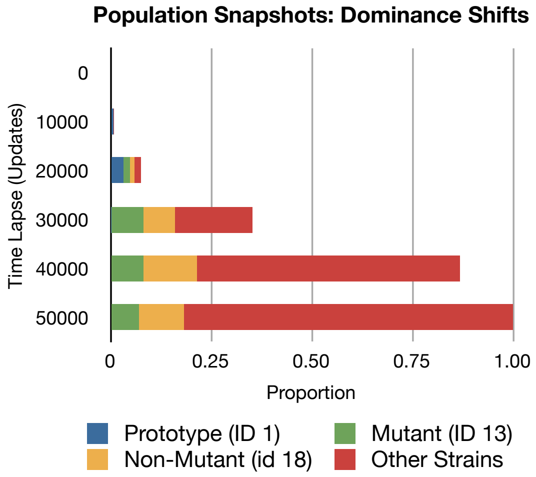
3. The offspring organism of this strain inevitably reproduces a mutant offspring which has the mutation expressed in the phenome.

4. The mutant offspring manages to self-reproduce and later becomes dominant in the population.[14]

Execution Profile of the Mutant

The mutant that became dominant was examined via a *trace file* in order to better understand the mechanism of its self-reproduction. A trace file is a program execution log of an organism of a particular strain which an Avida experimenter can collect (see its format in Appendix C). The file contains a log of step-by-step transitions of the virtual CPU states of an input organism of a particular strain, in other words, snapshots of the configuration

[14] As opposed to the dominant mutants, dominant non-mutants (i.e., with a perturbed phenome) may be of interest in its own right, but the current research is more focused on dominant mutants (i.e., with a perturbed genome); this is because, again, especially mutants with a changed genotype-phenotype mapping are of interest. In hindsight, some or all of the dominant non-mutants might be the same reproduction mode as the self-copier described in the rest of this section (i.e., losing the distinction of phenotype and genotype). As emphasised afterwards in Section 3.5, it is not easy to determine whether two self-reproducing organisms have the same reproduction mode, and basically case-by-case analysis would be required to be able to identify a reproduction mode.



Update	Dominant
0	Prototype (ID 1)
...	Prototype (ID 1)
22100	Mutant (ID 13)
...	Mutant (ID 13)
29400	Non-mutant (ID 18)
...	Non-mutant (ID 18)
29700	Mutant (ID 13)
...	Mutant (ID 13)
30100	Non-mutant (ID 18)
...	Non-mutant (ID 18)
50000	Non-mutant (ID 18)

Figure 3.12: Example population snapshots for a 50000-update run (top) and dominance shifts of this run (bottom). In the top bar chart, around the update 20000, the prototype remained dominant, within a relatively small total population (less than 0.2 of the capacity). By the update 30000, a mutant became dominant, while there was a non-mutant, which closely follows the mutant in number. The total population grew close to 0.4 of the capacity. By the update 40000, the non-mutant became dominant, with the total population being over 0.8 of the capacity, and this remains by the update 50000. By the end of the run, the total population grew to the capacity. There emerged a number of other strains within the population. The mutant (ID 13) was originally born at the update 13917, and the non-mutant (ID 18) at the update 14652. In the bottom table, the updates are shown in multiple of 100. A mutant and a non-mutant became dominant, alternately, which implies the closeness of the numbers of organisms of each strain over a certain period of time (approximately from the update 22100 to the update 30100). Note that ID numbers are given to strains by Avida systematically in the order of appearance, and that those shown here are valid only for this particular run.

of the virtual CPU and memory of an Avidian organism, for a certain configurable period of time or until division.

By studying the transition of the virtual CPU states logged in a trace file, one can create an *execution profile* showing a summary of the trace file, or of a given organism's behaviour. A trajectory following the instruction pointer (IP) relative to the strain's memory image is an important aspect of the execution profile. An execution count of the `write` instruction and the `h-copy` instruction is another. Thus, the execution profile may suggest something about the underlying mechanism of how the input organism of a particular strain self-reproduces. *Gestation time* is another relevant attribute with regard to the program execution calculated by counting the virtual CPU time steps (or the number of instructions executed) taken from the time when the program starts getting executed to the time when the `divide` instruction is executed. Gestation time is a reciprocal of reproduction rate. Measuring gestation time indicates how long it takes for an organism of a particular strain to self-reproduce, and hence how fast it self-reproduces.

See Figure 3.13 for the IP trace of this particular mutant in comparison with the prototype. Whereas the prototype has 322 instances of `write` execution and 322 instances of `h-copy` execution, the mutant has 2 instances of `write` execution and 643 instances of `h-copy` execution. Whereas the gestation time of the prototype is 52218, it is 22172 for the mutant.

As a separate experiment, it was confirmed that this mutant was able to displace the prototype when the two were seeded together, due to its faster self-reproduction (see Figure 3.14 for the comparison).

3.4.3 Degeneration of the Prototype: the Mechanism

The close-up mechanism of the mutant's self-reproduction can be described as follows. This mutant has a mutation expressed at the address `89`, inherited at the address `411`; the mutation of the word `13` into `14` at the address `411` gets expressed as the instruction `dec` rather than the original `push`. The location of the modified phenotypic word is in the middle of the preprocess of setting one of the two arguments for a `write` instruction execution, which is the value of the whole length. Figure 3.15 shows a high-level diagram of how the degenerate mutant self-reproduces by self-copying.

What the mutation caused is, in short, a malfunction in the (previous) decoding loop that inadvertently decrements and loses a necessary value (namely, the address of the lookup table) that should be kept stored in the stack. As the number of the originally intended values stored in the stack is decreased, the designed procedures are disturbed and they start handling incorrect values stored in stacks. In effect, the loss of a value caused the wrong use of another value stored in the stack and led to the failure of proper looping in the decode process.

From what is described above, one can conclude that this mutant reproduces by self-copying. The observed phenomenon is to be dubbed as *degeneration* of the von Neumann style prototype self-reproducer into a self-copier. Judging from this particular case in question, it is indicated that one step of single-point mutation is sufficient to cause the degeneration. The behaviour is explained below with relevant code fragments, as examined in the trace file step by step.

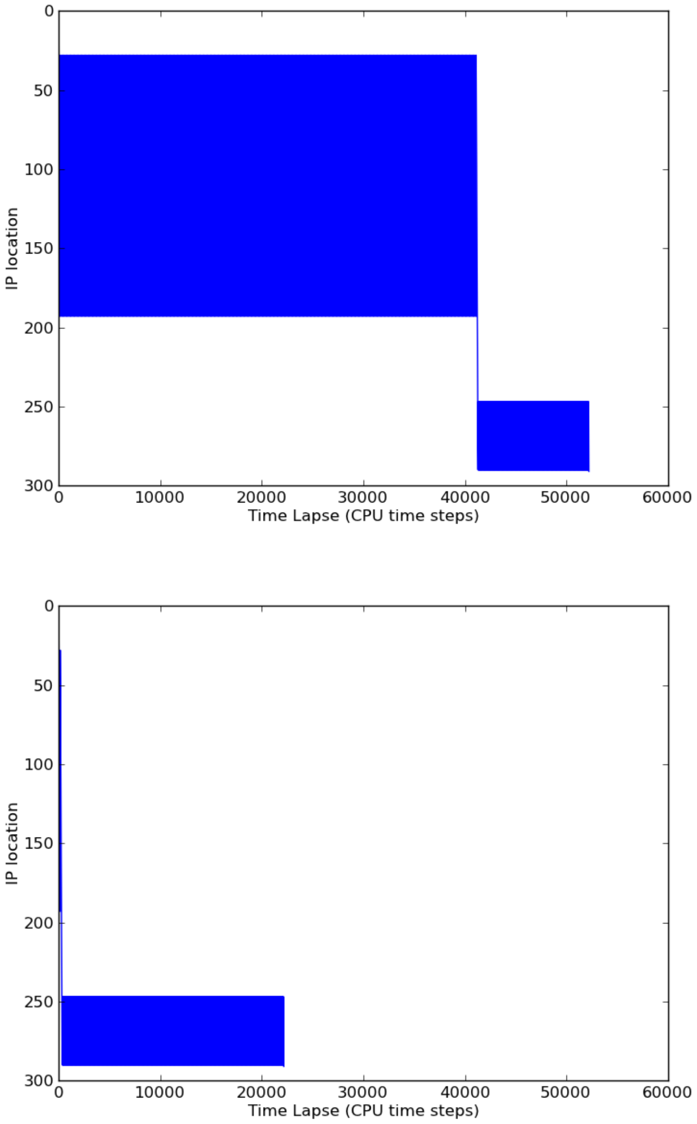


Figure 3.13: Example of actual IP traces of the prototype and a self-copying mutant (the one reported in the main text). IP traces can be extracted from original trace files that contain step-by-step execution of the organism program. The former has gestation time of 52218, the latter 22172. From the trace files, the `write` and `h-copy` execution count can also be extracted. The former has 322 instances of `write` execution and 322 instances of `h-copy` execution; the latter 2 instances of `write` execution and 643 instances of `h-copy` execution.

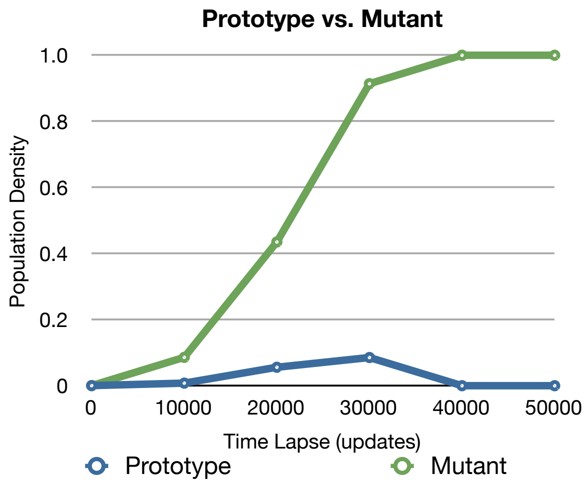


Figure 3.14: An example run for 50000 updates, seeded with the prototype and the mutant in question, with no perturbation. The world is a torus and the size is 60 × 60. The two were seeded diagonally (so that they sit at diagonal vertices of a 30 × 30 square). The mutant displaces the prototype, before update 40000.

Disrupted Decode Segment Behaviour

The code fragment found in Table 3.10 is taken from the "decode loop" segment, whose original function is to set up the value necessary for the next process (i.e., the whole length that is one of the arguments of the `write` instruction, in this case) from Stack 0 into the register AX. The function therefore is originally required to move several values stored in the two stacks one by one to reach and obtain the necessary value, while preserving the values in order (so as to function as designed).

Here, as shown in the code, the `dec` instruction at the address `89` is the modified phenotypic word. Now that the former `push` instruction disappeared, the value (the start address of the lookup table) intended to be pushed from the register AX to Stack 1 remains in AX. The `dec` instruction, whose behaviour is modified by the following `nop-A` instruction, decrements the value in AX, subsequently used as the address of the lookup table. Then from Stack 0 (as `swap-stk` switches back the active stack from Stack 1 to Stack 0), another value is popped (the label size) with the following `pop` instruction; its destination is AX as the `pop` instruction is modified by the following `nop-A` instruction, replacing the above value (i.e., the start address of the lookup table − 1). As a result, the number of functional values stored in the stack is one less than designed. This deficiency is complemented by the next value in the stack; the remaining values likewise shift one by one, with the last functional values being filled with an extra zero (one of fillers in the stack).

At the end of the "decode loop" segment, the `if-n-equ` instruction is used at the conditional branch by comparing values between BX and CX. This time, the condition is satisfied and the execution loops back. After the loopback, the instructions around the expressed mutation get executed for the second time (see Table 3.11).

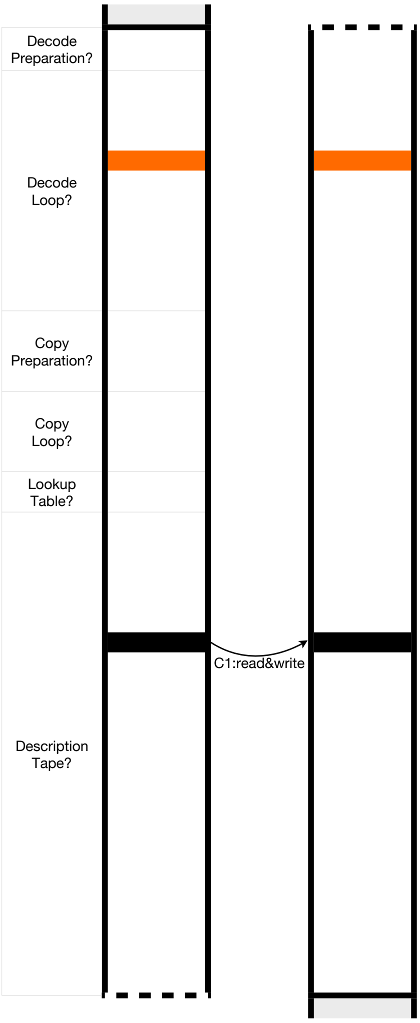


Figure 3.15: The schematic self-reproduction of the degenerate self-copying mutant of the prototype. This mutant self-reproduces essentially by self-copying, even though the memory image still closely resembles that of the prototype. In the mutant, there is no such decoding/copying decomposition as originally incorporated in the prototype.

Time	IP	Content	Registers	Heads	Stack
17	28	`pop`	[305,0,0]	[322,0,28]	*Stk0[322,...]
...	...	...	...	...	...
67	86	`pop`	[322,0,22]	[322,0,28]	*Stk0[294,...]
		`nop-A`			
68	88	`swap-stk`	[294,0,22]	[322,0,28]	*Stk0[2,...]
69	89	`dec`	[294,0,22]	[322,0,28]	*Stk1[322,...]
		`nop-A`			
70	91	`swap-stk`	[293,0,22]	[322,0,28]	*Stk1[322,...]
71	92	`pop`	[293,0,22]	[322,0,28]	*Stk0[2,...]
		`nop-A`			
...	...	...	...	...	...
143	192	`if-n-equ`	[1,321,0]	[322,0,28]	*Stk0[1,...]
144	193	`mov-head`	[1,321,0]	[322,0,28]	*Stk0[1,...]
	194	`pop`			

Table 3.10: First execution trace for the code around the modified phenotypic word (highlighted), and for the conditional branch at the end of the (previous) decode loop. The summary of the snapshots of the states (each right before the execution of the instruction at the address pointed at by IP at the time step) is shown on the right.

Time	IP	Content	Registers	Heads	Stack
145	28	`pop`	[1,321,0]	[322,0,28]	*Stk0[1,...]
...	...	...	...	...	...
195	86	`pop`	[321,1,10]	[322,0,28]	*Stk0[322,...]
		`nop-A`			
196	88	`swap-stk`	[322,1,10]	[322,0,28]	*Stk0[2,...]
197	89	`dec`	[322,1,10]	[322,0,28]	*Stk1[321,...]
		`nop-A`			
198	91	`swap-stk`	[321,1,10]	[322,0,28]	*Stk1[321,...]
199	92	`pop`	[321,1,10]	[322,0,28]	*Stk0[2,...]
		`nop-A`			
...	...	...	...	...	...
271	192	`if-n-equ`	[1,0,0]	[322,0,28]	*Stk0[1,...]
	193	`mov-head`			
272	194	`pop`	[1,0,0]	[322,0,28]	*Stk0[1,...]

Table 3.11: Second execution trace for the code around the modified phenotypic word (highlighted), followed by the execution of the code around the conditional branch. This "decode loop" segment is executed for the second time, but the `mov-head` instruction is skipped as the condition of BX≠CX is no longer satisfied. The execution then enters the "copy" phase.

As the "decode loop" fails the second time, the execution then enters the "copy" phase, starting from the "copy preparation" segment. This failure is due to the condition branch of BX≠CX which is not satisfied now that BX:0 = CX:0.

In terms of word writing during the execution of the mutated decode segment, the first procedures of the "decode loop" still functions despite the loss of value, and writes in the first word (the value of 22, corresponding to the `h-alloc` instruction) correctly at the start address of the phenome of the (putative) offspring being created. As the loop condition is satisfied this time, the execution loops back. In the second iteration of the loop, a second word is written in at the address corresponding to the second word of the offspring (i.e., at the relative address 1 of the phenome of the intended offspring memory image). This word is a wrong one, as the word for writing fetched by now turns out to be the value of 10 (corresponding to the `shift-r` instruction), not the original second word (the value of 25, corresponding to the `h-search` instruction).

Affected Copy Segment Behaviour

After the disrupted decode segment, the execution enters the copy segment. The value of 1, an inadvertent value, is used to locate the read head, and later the write head as well. This value comes from one of the "fillers" of the stack (the stack is by default filled with zeros at the beginning first) and incremented at a later point in the process. Where it was intended to calculate the remaining length by subtracting the genome start address from the whole length (i.e., the whole length − the phenome length), it subtracts the inadvertent 1 from whole length (whereas the whole length properly remains to get handled).

What is more, that inadvertent 1 is used as a value to locate the read head, instead of the genome start address (if it were starting the copy process after the decode, it would have been the genome start address that the first word should be copied from). Right after it is finished, the write head is located at the address of the whole length plus inadvertent 1, (instead of that of the whole length plus the genome start address, where a word would have been copied into in the prototype).

Those values implicitly explain how the copy process can start and eventually end up finishing by copying no more or no less than the rest of the memory image (see Tables 3.12, 3.13, and 3.14 for the code fragments demonstrating how the copy loop starts, how it loops back, and how it finishes). Up to this point, the value of the remaining length turns out to be the actual one 643, the remaining times to execute the `h-copy` instruction, leaving the correctly written first word so as to be able to start copying from the next one (replacing the miswritten second word). This is allowed by the fact that the read head is placed at the address 1, and the write head at the address 645, which turns out to be the source and the destination for the second word of the phenome, respectively.

Recapitulation

The analysis into this particular mutant which could cause degenerative dominant shift or degenerative displacement showed that the self-reproduction of the emerged mutant relies upon the copy process using the `h-copy` instruction rather than with the help of the `read` instruction and/or the `write` instruction, except for one word. A decisive factor for the

Time	IP	Content	Registers	Heads	Stack
310	246	`h-search`	[643,1,645]	[1,645,645]	*Stk0[644,...]
311	247	`h-copy`	[643,0,0]	[1,645,247]	*Stk0[644,...]
312	248	`swap-stk`	[643,0,0]	[2,646,247]	*Stk0[644,...]
...	...	...	...	...	...
343	289	`if-n-equ`	[644,642,0]	[2,646,247]	*Stk0[644,...]
344	290	`mov-head`	[644,642,0]	[2,646,247]	*Stk0[644,...]
	291	`h-divide`			

Table 3.12: Execution trace of the first copy loop.

Time	IP	Content
	246	`h-search`
345, 379, ..., 22105	247	`h-copy`
346, 380, ..., 22106	248	`swap-stk`
......	...	...
377, ..., 22103, 22137	289	`if-n-equ`
378, ..., 22104, 22138	290	`mov-head`
	291	`h-divide`

Table 3.13: Execution trace of the body of the copy loop (after the first, before the last execution).

Time	IP	Content	Registers	Heads	Stack
	246	`h-search`			
22139	247	`h-copy`	[644,1,0]	[643,1287,247]	*Stk0[644,...]
22140	248	`swap-stk`	[644,1,0]	[644,1288,247]	*Stk0[644,...]
...	...	...	...	...	...
22171	289	`if-n-equ`	[644,0,0]	[644,1288,247]	*Stk0[644,...]
	290	`mov-head`			
22172	291	`h-divide`	[644,0,0]	[644,1288,247]	*Stk0[644,...]

Table 3.14: Execution trace of the final phase of self-reproduction.

successful division of this mutant is that the read head and the write head are positioned correctly when the `h-divide` instruction is executed (i.e., the read head pointing at the start address of the offspring, and the write head at the end).

To recapitulate, the mechanism of how the mutant ends up self-producing by copying can be summarised as follows.

1. The (previous) *decode loop* using the `read` and the `write` instructions is destroyed by the mutation and that part starts to malfunction, iterating twice, writing two words (the first one correctly, the second one incorrectly).

2. The subsequent (previous) *copy loop* using the `h-copy` instruction, manages to function, by overwriting the second miswritten word made by the mutated decode loop, and by continuing to copy the rest of the memory image.

3. The program finishes when it reaches exactly the end of the strain, because the exact remaining length is set by the time the copy loop starts.

3.5 Reflective Remarks

In the course of designing and observing the prototype von Neumann style ancestor, two more questions were raised, concerning (a) identification or classification of reproduction mode and (b) alternative ancestor design. Reflecting the understanding gained through the study covered in this chapter, approaches towards these questions are discussed in this section.

3.5.1 Towards Identification of Reproduction Mode

It has been demonstrated that a von Neumann style self-reproducer can degenerate into a self-copier. The phenomenon of degeneration observed marks the immediate loss of the decomposition into genome and phenome, hence in particular the loss of the mutable, evolvable genotype-phenotype mapping.[15] That is to say, this particular von Neumann style self-reproducer could mutate into a self-copier that no longer has a division of labour between phenome (i.e. the active constructor-copier) and genome (i.e. the passive information storage), from the effect of a single one-point mutation.

As mentioned in Section 3.4, one of the six cases where dominance shift took place was elaborated as an example. To touch briefly on the other five examples than the one that is reported as a mutant which led to dominance shift, the observed mutants all had mutation expressed in the decode loop, and had gestation time less than half that of the prototype ancestor (namely, 22172, 22174, 22078, 22044, and 22172, in no particular order), with similar execution profile (of 1 or 2 `write` executions and 643 or 644 `h-copy` executions).

Judging crudely from the similarity between their attributes, they are conjectured to also be self-copiers, to the extent the first example of degeneration was concluded

[15] The mutant studied in Section 3.4 decodes one word (the first word of the offspring memory image; in this case, the value 22, which corresponds to the `h-alloc` instruction). However, this decoded word does not become the phenome of the offspring (as the offspring turns out to have no such decomposition as genome and phenome as its parent had). Put another way, although it might count as a part of phenotypic activity in a very weak sense, it does not qualify as an act of "decoding a genome".

to generate a self-copier, presumably with a similar mechanism. However, it is not clear whether they self-reproduce in exactly the same mechanism or reproduction mode; nor is it clear how similar their ways of self-reproducing are to one another (or how different they are from one another). It is not clear, either, how one can best classify reproduction modes of the mutants (or arbitrary mutants in Avida in general) based on similarity or commonality of self-reproduction mechanism. Even when the program of the organism looks the same, the use (or interpretation) of the words in the program can be completely different. In the same sense of the argument by McMullin (1995), a memory image (or a part of memory image) can be decoded differently depending on the context, and conversely, the same behaviour can arise from different memory images.

To judge whether any two organisms are using the same reproduction mode, their step-by-step state transitions in trace files have to be wholly examined and compared on a case-by-case basis. Describing the self-reproduction mechanism which a particular organism does employ (and more importantly, finding the organism's evolutionarily interesting mutational pathways) can be laborious and cumbersome. As a general rule, to determine the reproduction mode of an organism, the organism has to be examined on a step-by-step basis via a trace file. In the case of the prototype, it is difficult especially because of their relatively large size (e.g., 644, in the above case) and the complexity of their structures (e.g., the above inadvertent self-copying mechanism). It turned out, in this particular case study of Section 3.4, that a trace file size can be considerable since it is a step-by-step log: for this mutant, the trace file size is 55.9MB (in comparison, it is 131.4MB for the prototype). To produce and compare trace files (such as those for the above-mentioned mutants) of arbitrary sets of mutants seems to require an additional analysis, which can probably be computationally expensive and even ineffective, considering the evolutionary environment of the standard Avida.

In any case, once such a degenerate self-copying strain arises, it is not at all surprising that it displaces the von Neumann style self-reproducer in the population. Such self-copiers would be selectively favoured by the Avida system due to their reproduction rate being much higher than that of such von Neumann style self-reproducers as the prototype ancestor. The particular self-copier, being the same length as the prototype, takes less than half of the CPU cycles required by the prototype to self-reproduce. A factor underlying the difference between the necessary CPU cycles is the fact that the decode process needs to execute more instructions than the copy process for a word to be written into one memory location of the offspring. Self-copiers are generally expected to be faster in that they avoid the procedural cost of the decoding.

The topic of identifying or classifying reproduction modes will be revisited and further elaborated in Subsection 4.6.2 in Chapter 4, and in Section 5.3 in Chapter 5.

3.5.2 Reconsidering the Design of Ancestor

Ancestor design is an important aspect in any Avida experiments, concerned with setting up an initial condition of evolution. Not only the structural and functional design, but also the design of the instruction set on which the ancestor design is based is crucial, as it determines a part of the law governing the Avida world.

Although the classic set could have been used, the default 26-instruction set was chosen to be used as the core of the 28-instruction set upon which the prototype ancestor design is based. (The classic set can be enabled in the particular version of Avida, as shown in Section 3.2). Requirements for a design of an optimal instruction set for implementing a von Neumann style self-reproducer are not clearly defined (with respect to this, whether the default instruction set is an optimal design as a starter is yet to be known in the first place). In the current study, two instructions (namely `read` and `write`) were additionally enabled in order to support the reproduction process comprising such phases as copying and decoding. These were the two simplest instructions for basic reading and writing available among the instructions listed in the library found in the source code of the virtual hardware CPU of Avida.

The presented design of the prototype and the configuration of the instruction set were rather arbitrary.[16] These serve to give a valid proof of principle demonstration that can be used as an example and a springboard for further detailed research. Therefore, there could be an arbitrary number of different designs of von Neumann style self-reproducing ancestors based on different design conditions or design requirements, using a differently configured instruction set. It is clear that the design of the prototype could be reconsidered, from the fact that there are several “unemployed” instructions in the instruction set, namely, those functionally categorised as mathematical operations and input/output operations. Though not employed in the prototype, these instructions are included as part of the “default” instruction set and kept enabled in the configuration. Mathematical or input/output instructions are typically utilised for the computation when there is external fitness that organisms can gain through interacting with the external task environment via the input/output components, the effect of which is out of scope in the research of the current thesis. (In many Avida studies where there is external fitness, the standard self-copying ancestor can evolve to utilise such instructions to be competitive in the population.) As opposed to that case, the prototype is intended to start off without ancillary machinery in order to observe the plain reproductive potential of the von Neumann architecture. Through mutation, such mathematical or input/output instructions might possibly arise in programs and become utilised for some function; but, more likely, some functional effect in reproduction cycle may arise.

In retrospect, the method of reproduction which the classic instruction set assumes, is somewhat similar to the design of the novel prototype ancestor. In the sense that the read and write heads are utilised to a lesser extent in the implemented reproductive process, the design of the instruction set used in the current investigation may be considered as being slightly closer to the design of the classic set, than that of the default (newer) instruction set is. If that is the case, it is highlighted that the Avida system in the early stage of development did not very much utilise those heads (or, at least, did not rely on them as essential components). Most likely, by introducing the use of control heads and of instructions using them, the Avida system was intended to encapsulate a major series of procedures so as to lessen the size of an organism’s program. Roughly speaking, an operation performed by one instruction may be (at least slightly) less prone to mutation than the same operation performed by multiple instructions (thus, in this sense, the more

[16] For example, one could design an instruction from scratch, or design a new instruction set. In this regard, the design of the prototype ancestor can be said to rely more on what are originally provided within Avida.

"encapsulated", the more an instruction is able to avoid malfunctions in the operation performed by its sub-procedures).

The topic of the effect of an instruction set in designing an ancestor will be re-discussed in Subsection 4.7.3 in Chapter 4. Along the same line, the encapsulation of the `h-copy` instruction is discussed in Subsection 5.3.4 in Chapter 5.

3.6 Closing Remark

The Avida artificial life system as an experimental and modelling platform is overviewed in Section 3.2. It is followed by Section 3.3, where a prototype ancestor minimally equipped with the von Neumann style architecture of self-reproduction was designed and implemented in the Avida context. Its behaviour was preliminarily observed in Section 3.4, which led to a finding of degeneration of the designed ancestor via a step of point-mutation.

Through this first approach taken, it was demonstrated that a von Neumann style organism is implementable in Avida. The observation suggests a need for a more extensive and systematic analysis of the prototype's mutational pathways (*mutational analysis*). Naturally, there should ideally be some analytical tool available for general self-reproducers in Avida. The next Chapter 4 considers what would be required to develop such a tool, proposes a feasible method, and evaluates the method by demonstrating it using the prototype ancestor.

Relevant to the reflective remarks in Section 3.5, it is argued in Chapter 4 how frequently degeneration of the prototype occurs, based on a heuristic classification. There, the difficulties of classification of reproduction modes discussed above are partly tackled in the context of mutation analysis. From the perspective of the original purpose of the research, it should be noted that it is not enough to know reproducibility (whether a strain divides or not) or reproduction mode of mutants of the particular prototype. More interesting is evolvability of mutants of a von Neumann style self-reproducer in a more general situation of Avida, particularly, not of mutants that are simple self-copiers, but those retaining some von Neumann architecture. This point is discussed as well in the course of the investigation presented in Chapter 4.

Chapter 4

Automated Mutational Analysis

4.1 Overview

Overall, this chapter proposes a new analysis method based on the built-in tool of Avida by revisiting the concept of *viability*. The chapter opens by discussing the research direction to be taken, systematic *mutational analysis*, and its importance, providing a detailed exposition of it. Then Avida's original analysis mode is introduced and applied on a set of mutants of the prototype introduced in Chapter 3 as a starting point of the subsequent enhancement of analysis. The framework of the enhanced analysis is elaborated and results from the analysis are presented. The automation of the enhanced analysis follows, where a method of multi-step mutational analysis is demonstrated. Lastly, the contribution of this chapter is discussed, outlining remaining research questions as part of the conclusion. Additionally, an alternative design of the prototype ancestor is considered both in terms of evolvability compared to the prototype, and in terms of applicability of the developed analysis method.

4.2 Research Direction and Expositions

The research described in Chapter 3 was a step to characterise a von Neumann style self-reproducer in Avida. It was an approach to the original research question of finding changes in a genotype-phenotype mapping of such a self-reproducer with the von Neumann style architecture in Avida. Following this, within the scope of characterisation of a particular self-reproducer (the prototype ancestor), there are a few possibilities of further research directions including: (a) evolutionary characterisation through running standard Avida experiments; and (b) systematic analysis of specific mutational pathways.

Conducting a number of evolutionary experiments (with mutations and resource limits) using the prototype is one possibility. While it is practical and empirical to start with, the result may be limited to, or too particular to, a particular ancestor. Though it is worthwhile in and of itself to characterise a particular ancestor (such as the prototype), different designs of von Neumann style ancestors would have to be repeatedly run for experiments for a more comprehensive understanding of von Neumann style self-reproducer. That means, in this line of research, that better guidelines and clarification as to conditions would be needed for designing alternative ancestors; but it is not straightforward to know how effectively different ancestors could be designed and how many kinds of ancestors should be studied.

Apart from standard, evolutionary Avida experiments, another possible research direction is a systematic and somewhat abstract investigation: analysis of possible mutational pathways for a given ancestor, or *mutational analysis*. It requires the development of a generic method of analysis of mutational pathways of a particular strain. Mutational pathways here mean mutants that can be reached from a particular ancestor through mutations. An advantage of this line of research is that one can attain a foundational framework of analysis of mutational pathways. That means that a method developed can potentially be used not only for the existing prototype ancestor, but also for unknown ancestors of different strains.

In either case, how to classify strains by reproduction mode is a problem to be tackled. Different strains may have different reproduction modes: some may have a von Neumann style architecture, others may have another version of von Neumann style architecture. Even if distinct strains are classified as having the same reproduction mode, they may have more or less different mechanisms to realise that reproduction mode. There may be not only self-copying or von Neumann style architectures but may also be mixtures of those architectures (mixed to different extents). Moreover, strains may produce non-identical offspring or may not produce any offspring.

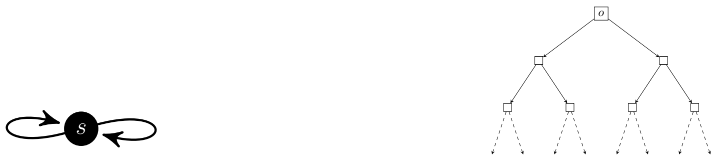
The current research takes the direction of investigating and developing a systematic analysis method for mutational pathways. This will be called mutational analysis for short. A generic, foundational framework of mutational analysis will be valuable for characterisation of strains based on mutational pathways. Once a mutational analysis method is developed, it may, in principle, provide a basis for more systematically investigating changes in genotype-phenotype mapping. This is directly relevant to the characterisation based on evolvability which strains can potentially give rise to. Mutational analysis will be useful if redesigned alternative ancestors are to be investigated and characterised. Insights gained through the development of a mutational analysis method and through characterisation of the particular hand-designed prototype ancestor using the method may benefit further research as well. Such insights could also help to plan evolutionary Avida experiments and alternative ancestor designs.

In the rest of this section, first, how strains can be best classified will be considered in order to revisit the concept of self-reproduction, and second, a research program defining and exploring the somewhat more general concept of *viability* will be proposed.

4.2.1 Exposition 1: Self-Reproduction Revisited

From studying organisms in the Avida world to such an extent as in Chapter 3, one might assume that there is a clear division between self-reproduction and something that is not. However, whether an organism divides or not, and if it divides, whether an identical organism is reproduced as a result of division or not, appear to be definitely relevant to, but not necessarily determinant for, self-reproduction. There is certainly subtlety in self-reproduction, not to mention "interesting" self-reproduction. The concept of self-reproduction will now be revisited and unpacked, and strains will be more finely classified.

Intuitively, what can be defined straightforwardly is a *simple direct self-reproducer* strain: an organism divides (as a parent) successfully and as a result have two identical



(a) Production graph of a simple direct self-reproducer strain as a seed strain. The "*s*" denotes *strain.*

(b) Production tree of a seed organism of a simple direct self-reproducer strain. The "*o*" denotes *organism.*

Figure 4.1: Lineage of direct simple self-reproducer strain.

organisms (as offspring). By "identical", it is meant that strains of the parent organism and those of the offspring have the same (initial) memory image, and therefore function or behave in the same way as long as those memory images are executed by a (reset) virtual CPU of each organism.

A self-reproducer strain represented as a production graph is shown in Figure 4.1a. A node represents a strain (as opposed to an individual organism). Distinct nodes would imply distinct strains. Edges represent production relationships. Each edge is directed, pointing from parent to offspring. An Avidian organism of a certain strain can be regarded as having precisely two or zero offspring. So, in the Avida world, each node must have either 0 or 2 arrows going out from itself. The strain represented by the graph in Figure 4.1a is called a simple direct self-reproducer, that is, a parent organism of this strain produces two offspring organisms of the same strain. Production graphs are one way of representing lineages, from the perspective of distinctive strains. In contrast to this, a lineage of an organism of a self-reproducer strain can be represented as a tree, as shown in Figure 4.1b. This representation is referred to as a production tree, describing a lineage tree generated by a seed organism as a result of production. Here, a network of individual organisms is a lineage, where nodes denote individual organisms. In this representation, nodes (individual organisms) are depicted separately regardless of whether or not any two organisms within a lineage are of the same strain. Production trees are another way of representing lineages, from the perspective of individual organisms. This chapter mainly focuses on reproduction graphs, rather than reproduction trees. (Strain classification is attempted in this subsection, using production graphs, and this representation of production graphs is mathematically revisited in Section 4.4 in pursuit of a method for analysing strains.)

The above case is an example of a simple direct self-reproducer. Naturally, there are the opposite cases to simple direct self-reproducer strains: *simple direct infertile* strains (or simply, *infertile* unless otherwise specified), as shown in Figure 4.2. If an organism of a certain strain fails to divide and hence ends up having no offspring, the strain is called infertile.

Of these cases shown above, a simple direct self-reproducer strain, as a seed strain, is evolutionarily significant, and may be evolutionarily interesting; whereas a simple direct infertile strain, as a seed, is of no evolutionary significance or interest. However, there are conceivably strains that potentially generate different patterns of lineages. To better clarify what strains can be of evolutionarily significance or interest, the way of classifying strains is considered and detailed next.

(a) Production graph of a simple direct infertile strain as a seed strain.

(b) Production tree of a seed organism of a simple direct infertile strain.

Figure 4.2: Lineage of direct simple infertile strain.

Strain Classification

If one only looks at a first division of an organism of a certain strain, there are logically cases other than simple direct self-reproducer or infertile above: namely, a strain reproduces (a) one identical, one non-identical offspring, and (b) two non-identical (all different) offspring. There is, however, more subtlety in classification of strains when reproduction is looked at from a longer-term perspective. It is conceivable that there are classes of strains as shown in Figure 4.3. The strain classification depicted in this diagram is intended to be more conceptual than precise or detailed. Note:

- There is a coarse distinction between *infertile* and the rest (or *fertile*, which includes *self-reproducer*).
- There is a *direct* self-reproducer class within the general self-reproducer class. In such cases, *every* strain generated from the seed strain (including the seed strain itself; presume this in this chapter unless otherwise noted) grows exponentially in time.
- Furthermore, there is a *simple/collective* distinction in the self-reproducer class. Simple means exactly one distinct strain involved; whereas collective means more than one distinct strain. In the simple class, “self” means a single distinct strain, whereas in the collective class, “self” would mean some group of distinct strains. In simple self-reproducers, classification by reproduction mode (such as self-copying and von Neumann style) is relevant, whereas in collective self-reproducer, it is *prima facie* unclear how this should be defined: it would depend on much more detailed examination of the collective interactions or relationships
- There is an *indirect* self-reproducer class. A seed strain falls in this class if it is not a direct self-reproducer, but gives rise to one or more strains that are direct self-reproducers; if all the latter are simple self-reproducers, then this is an indirect simple self-reproducer; if all are collective self-reproducers, then it is a collective self-reproducer; otherwise, it is classified as mixed.

To illustrate, a few examples of possible production graphs of the above conceivable classes are shown. Several strain classes are (roughly) progressively explained, starting from simple patterns with few strains.[1]

Figure 4.4 shows again the most straightforward classes: a simple direct infertile strain and a simple direct self-reproducer strain.

[1] Note that we present here, in as simple a form as possible, the conclusions from initially relatively undirected exploration. The methodology adopted involved an iterative process of generating potential production graphs with progressively more nodes/strains and revising or refining the proposed classification scheme as new possibilities where identified.

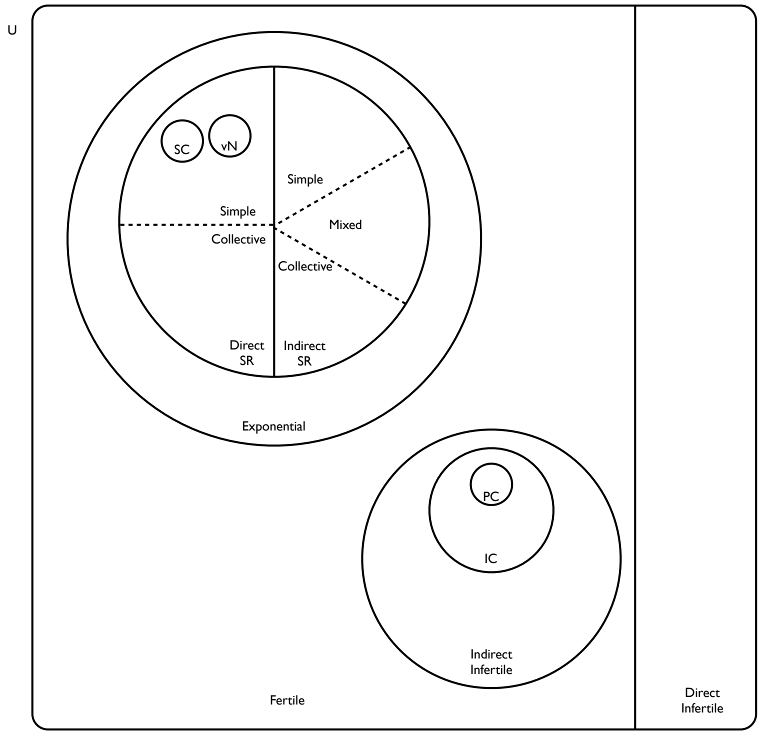
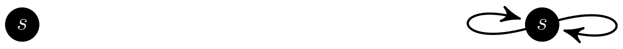


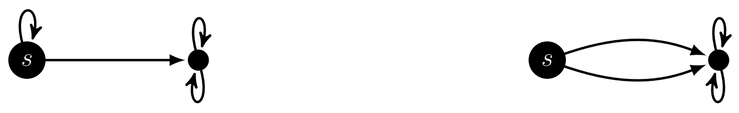
Figure 4.3: Venn diagram of strain classes. The whole set U is the space of Avidian strains. Acronyms are used for legibility in the diagram: SC refers to self-copying; vN refers to von Neumann style; SR refers to self reproducing; PC refers to pathological constructor. For further explanations of each class, see the main text.



(a) Simple direct infertile strain.

(b) Simple direct self-reproducer strain.

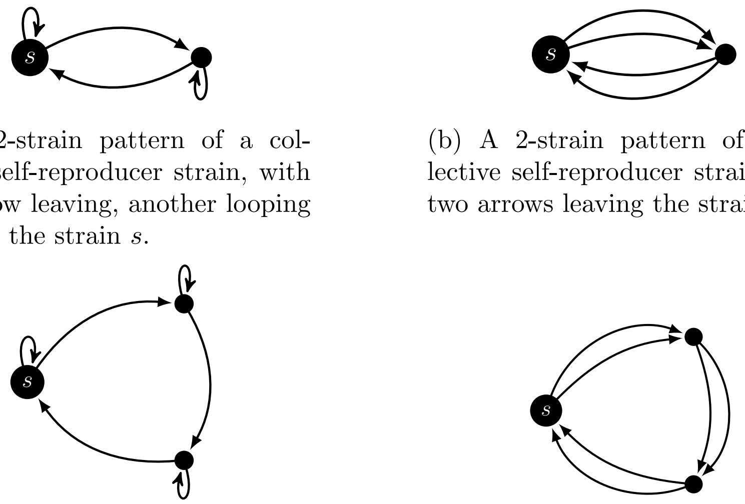
Figure 4.4: A simple direct infertile strain and a simple direct self-reproducer strain. These are the patterns of production graphs involving exactly one distinct strain. The node marked with an s is the *seed* strain, or a starting strain.



(a) An indirect simple self-reproducer strain, with one arrow leaving, and another looping back to, the strain s.

(b) An indirect simple self-reproducer strain, with two arrows leaving the strain s.

Figure 4.5: Indirect simple self-reproducer strains.



(a) A 2-strain pattern of a collective self-reproducer strain, with one arrow leaving, another looping back to, the strain s.

(b) A 2-strain pattern of a collective self-reproducer strain, with two arrows leaving the strain s.

(c) A 3-strain pattern of a collective self-reproducer strain, with one arrow leaving, another looping back to, the strain s.

(d) A 3-strain pattern of a collective self-reproducer strain, with two arrows leaving the strain s.

Figure 4.6: Collective direct self-reproducer strains.

Indirect self-reproducer is a class that exhibits such production graphs as shown in Figure 4.5.

Collective self-reproducer is a slightly more complex class. As opposed to simple direct, this is a class that exhibits such production graphs as shown in Figure 4.6. Here, patterns that are symmetrical and that involve 2 or 3 strains are shown as examples.

The classes explained so far can be considered as straightforward and most relevant in the interest of laying a foundation for the subsequent sections of the current chapter. However, as is implied from Figure 4.3, there can be other strains that do not fall into the classes described above. Such classes of strains are mentioned below for the sake of completeness:

- There remains the grey class of "fertile-but-not-self-reproducer" strains within indirect infertile.

- The direct/indirect distinction applies analogously to infertile as well. Direct infertile strains are easy to classify. But conceivably, there can be strains that are fertile because they produce something, but that end up producing all infertile strains. Such strains should be precisely classified as indirect infertile.

- A notable class of such "fertile-but-not-self-reproducer" strains is the *pathological constructor* class, having the potential of indefinite linear population growth. This class can be defined as having one offspring that is identical to a parent and one offspring that is infertile. That is, as a result of division of a pathological constructor, there are a pathological constructor and a sterile offspring. A consequence that this type of strain can bring about is indefinitely producing (or as the name suggests "pathologically constructing") numerous sterile offspring, to the extent that it affects a finite ecosystem. In this limited sense, this type of strain may have potential evolutionary significance like self-reproducers.[2] That said, this does not give rise to exponential growth, so is clearly distinguishable from self-reproducer.

- A more general class for pathological construction is conceivable, which can be referred to as an *iterative constructor*. Such a seed strain would generate more than two distinct strains in total, and would somehow iterate the same pattern as a pathological constructor. Qualitatively speaking, the relationship between pathological constructor and iterative constructor can be likened to that between a simple direct self-reproducer and a collective direct self-reproducer.

- Apart from the above, there is an infinitely *diversifying* class, where a seed strain produces an indefinitely growing set of distinct strains. Some such seeds may be potentially exponentially growing.

- Practically, there can exist strains that cannot be classified or are unknowable due to some observation time limit. This is because any observation to determine a strain class has to be concluded after some period of time.[3] Production graphs of such strains would be open, not closed with a finite membership.

- What was described so far are distinguishable classes. There are of course, combination of those classes. However, the class that is most relevant in this context is that of self-reproducer, as being exponentially growing and evolutionarily significant. There may be found other distinguishable and significant classes, too.

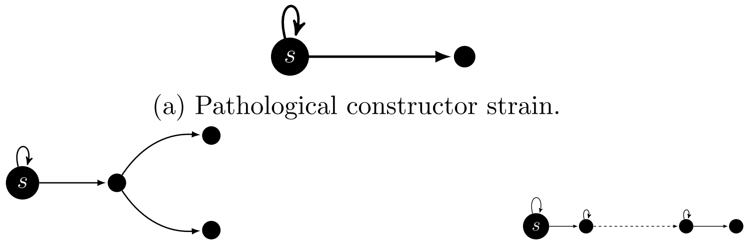
To illustrate, a few examples of possible production graphs of the above (less relevant) classes are shown.

Figure 4.7 shows the production graph of another characteristic class, a pathological constructor, together with the production graphs of two strains of iterative constructor, the general class of pathological constructor.

Figure 4.8 shows two simple examples of indirect infertile. This is a more generalised class of pathological or iterative constructors, in the sense that strains of this class even-

[2] Note that pathological constructors also have subtlety. Generally, when seeded in a spatial Avida, it will keep producing sterile offspring, and it is potentially linear population growth. Multiple pathological constructors will kill each other and one will survive producing sterile offspring (turnover); especially when `ALLOW_PARENT` is enabled (which means a parent on a parental node may be killed by an offspring), a seeded pathological constructors may end up producing sterile offspring surrounding itself, without no further population growth (see Appendix E for the configuration file).

[3] For the unclassified strains, there is another aspect to be considered: the length of time to run and trace an organism of a (single) strain for analysis. Practically, this value can be set relative to the prototype ancestor (as if in evolutionary experiments). This value must be revisited when in general cases, other factors (i.e. functions/activities other than self-reproduction) come into play that contribute to fitness changes (something other than production rate/gestation time).

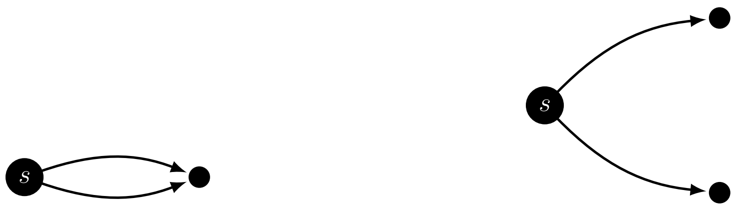


(a) Pathological constructor strain.

(b) An iterative constructor strain.

(c) A long iterative constructor strain.

Figure 4.7: A pathological constructor strain and an iterative constructor strain. The iterative constructor is a general class of pathological constructor. These are fertile in the sense they produce something, but identified (or conjectured to be) indirect infertile in the long run.



(a) An indirect infertile strain, with two arrows leaving the strain s, leading to one infertile strain.

(b) An indirect infertile strain, with two arrows leaving the strain s, leading to two infertile strains.

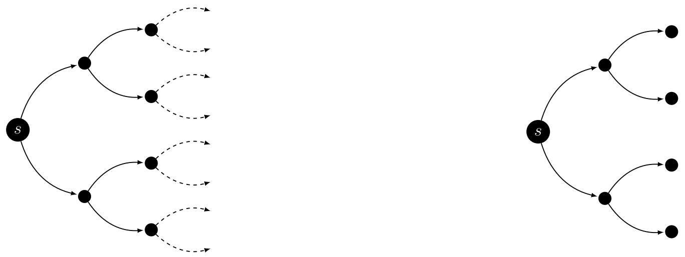
Figure 4.8: Indirect simple infertile strains.

tually lead to all infertile strains; whereas this class is distinguishable in that pathological or iterative constructors can generate indefinitely many offspring on an indefinitely large population, which infertile strains (whether direct or indirect) cannot.

See Figure 4.9 for a production graph of a strain of the diversifying class, beside a similar production graph of an indirect infertile strain. This class does not appear in the diagram of Figure 4.3, but can also be of interest theoretically as it may or may not grow exponentially. As mentioned, this class is less relevant compared to the self-reproducer class, in that a production graph of such a diversifying and exponentially growing strain would be unclosed, and cannot practically be observed and determined.

Example production graphs of other strains that do not fall in the aforementioned classes include, but are not limited to, ones in Figure 4.10.

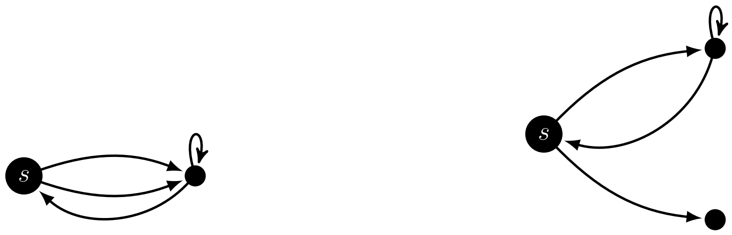
Subtlety in the notion of self-reproduction has now been illustrated. This is what renders any attempt at automated strain classification non-trivial. Importantly, simple direct self-reproducer is not the only class of strains that is interesting, or that can lead to exponential growth (i.e., evolutionarily significant) in the light of the research purpose (of finding genotype-phenotype mapping changes etc.). What is more, even self-reproducer strains may be of reproduction modes without genotype-phenotype mapping (e.g., a von Neumann style architecture that an ancestor has, may disappear through mutation). After all, the research is not as simple as checking simple direct self-reproducer among a certain set of strains. That is why this attempt of strain classification, even though it is not fully precise, is relevant. Through this attempt, it is suggested that there is a way of classifying



(a) A diversifying (and exponential) strain.

(b) An indirect infertile strain which is seemingly diversifying.

Figure 4.9: Diversifying strains. If the strain continues diversifying indefinitely, then, theoretically, the number of distinct strains as well as the population size will grow exponentially. In that case, the production graph will end up being unclosed. If the strain does not continue diversifying indefinitely, then, vice versa: neither the number of distinct strains nor the population size will grow exponentially; and the production graph will end up being closed.



(a) A fertile strain. This can be regarded as (atypical) collective self-reproducer.

(b) A fertile strain. This can be regarded as a combination of (atypical) self-reproducer and indirect infertile.

Figure 4.10: Other fertile strains. Strains that divide but do not fall in particular fertile classes would be classified merely as fertile, or other fertile. Such strains include but are not limited to the above cases.

strains, in a more fine-grained way, and more meaningfully based on its evolutionary potential which can translate as potential for exponential growth. This potential, viewed as *viability*, is elaborated next.

4.2.2 Exposition 2: Viability and Viability Analysis

Generally, for changes in a genotype-phenotype mapping to be detected, changes in reproduction modes would have to be detected; in turn, it would necessitate that a self-reproducer strain be somehow detected. In other words, self-reproducer strains are at least relevant, because it is a necessary condition for interesting strains. In this sense, the concept of being *viable* or *viability* in the current context will refer to self-reproduction as is, implying potential for exponential population growth (that is, under a circumstance without resource limits or variations), which is a necessary factor for the neo-Darwinian evolution, hence is evolutionarily significant. Inasmuch as viability is defined as potential for exponential growth, being a self-reproducer is being viable, but the reverse is not necessarily true. Thus, it is important and meaningful to be able to analyse viability of strains.

To recapitulate the strain classification, the two most straightforward classes are infertile and simple direct self-reproducer. If an organism of a strain fails to divide, the strain is classified as infertile, whereas if it succeeds to divide, the strain is classified (at least) as fertile. Some fertile strains may be self-reproducer. Simple direct self-reproducer is straightforward to classify, and definitely viable (i.e., of interest and significance evolutionarily, if defined qualitatively). Some fertile strains may turn out to be indirect infertile. Direct or indirect, infertile strains are of no particular interest or significance evolutionarily. (As explained, there is a class of pathological constructor that can be classified as indirect infertile and may still have evolutionary impact, but the impact should be negligible in the presence of any strains of a self-reproducer class in the evolutionary long run.) The complication here is that simple direct self-reproducer is not the only class that is viable, and to search for viable strains (or, self-reproducer strains in a broad sense) is not as straightforward because such strains may exhibit complex patterns of production graphs involving multiple distinct strains.

Essentially, the notion of self-reproduction turns out to be not as straightforward as it appears, so here it has to be more rigorously defined. In the first place, an organism of a certain strain either fails to divide, producing no offspring, or successfully divides to produce two offspring (or at least there exists a point of view to regard a strain as such). Among cases where a parent divides and produces two offspring, direct self-reproduction is a situation where, through division, the parent produces two offspring which are identical. That means those strains are represented as an identical memory image (consisting of an identical program and data which will give rise to the same behaviour when executed in the same way). There are conceivably indirect self-reproduction: a situation where a strain does not directly self-reproduce, but some descendent strain of it directly self-reproduces. For example, even if von Neumann reproduction mode is retained, it may take multiple generations for a genotype-phenotype mapping to stabilise as that of a direct self-reproducer. There is another class of collective self-reproducer. This may be a rather abstract class, but it can be defined as a group of multiple strains that mutually reproduce (again with the constraint that the number of any strain's offspring strictly has to be either zero or two). These are examples of strains that should not be simply excluded as being uninteresting just because they are not direct self-reproducers.

Viability analysis of strains will be one of the foci of the subsequent sections. Qualitatively speaking, viability analysis should mean more than organism *dividability* (or strain *fertility*) and strain *equality* analysis, but rather *evolvability* analysis. Fertility of organisms of certain strains is relatively straightforward to check. One would need to confirm whether an organism of a certain strain divides or not (hence reproduces or not) within a certain period of time. Whereas the word dividability here implies direct reproducibility, the word fertility is used to mean not only direct but also indirect reproducibility of something. Analysing equality of strains may be cumbersome especially when the strain size becomes large, but a comparison between two strains is basically straightforward. On the other hand, evolvability requires further consideration. Roughly speaking, evolvability here means the extent of having evolutionary potential or evolutionary future, such as potential of exponential population growth, from a lineage perspective. One would be able to detect a simple direct self-reproducer strain relatively easily, but that class is not the only possible pattern that leads to exponential growth or, more generally, has an evolutionary future. Analyses of dividability of organisms (or fertility of strains) and of equality of strains are essential for analysis of evolvability, but do not suffice when viability is considered. In Section 4.4, the viability quantification is discussed and introduced.

Furthermore, not only individual self-reproducer strains but also "collective" self-reproducer strains (i.e., strains that are self-reproducer viewed from a lineage as a whole) should be classified as fertile, and even as viable, having evolutionarily potential.[4]

Mutant Viability Analysis

Viability analysis will first be applied to mutants of a particular ancestor. This is an approach to lay the foundation of analysis of mutational pathways, and a step towards analysis of mutation of (and evolution of) a genotype-phenotype mapping as part of characterisation of a von Neumann style self-reproducing ancestor. Viability is not a sufficient condition, but a necessary condition of any evolutionarily interesting and significant genotype-phenotype mapping. (In particular, organisms that maintain von Neumann style self-reproduction are of interest from the perspective of the original purpose of this research stated earlier in Section 2.2 in Chapter 2.) Thus, in the search for viable strains, mutants are referred to as *candidates*.

To map out viability analysis of mutants, mutational pathways will be analysed for viability, starting from first-step single-point mutants, to multi-step mutants. Here, mutational pathways specifically refer to those which include first-step single-point mutants and subsequent (hypothetical) single-point mutants of generated strains (referred to as multi-step mutants, as opposed to first-step). See Figure 4.11 for the conceptual diagram of this analysis. Exhaustive analysis needs to be done wherever necessary and possible, but practically, scale of analysis has to be considered for the analysis to complete within a feasible period of time and to focus only on interesting and significant candidates, such as: how long to wait before determining a strain is direct infertile; how many distinct strains to trace for one source strain. For multi-step mutant analysis, how and how much of the spectrum of candidates to prune for further mutational analysis (or a *selection mechanism*) also needs be considered.

Thus, in this type of analysis, pruning mutant strains will be an important factor. Mutational analysis should ideally be able to exhaustively analyse mutant strains on all the possible mutational pathways over multiple steps of mutations. That is, it should be able to incubate an arbitrary seed strain and predict its potentially generated strains, and analyse each of them, recursively. Strains that are expected to be deterministically generated from mutants (not through additional mutations) should also be targets for further analysis. However, the more mutational steps are added, the more computationally expensive it becomes to have to exhaust all mutant possibilities. In other words, this type of investigation on mutational pathways boils down to mutation space exploration of Avidian strains. One drawback that it may encounter is the combinatory explosion of mutants to be analysed that are attained through mutational pathways from an ancestor as the number of mutational steps increases. There will exist a trade-off between how

[4] This distinction is similar to that between *individual autocatalysis* and *collective catalysis* in artificial chemistry (Kauffman, 1993).

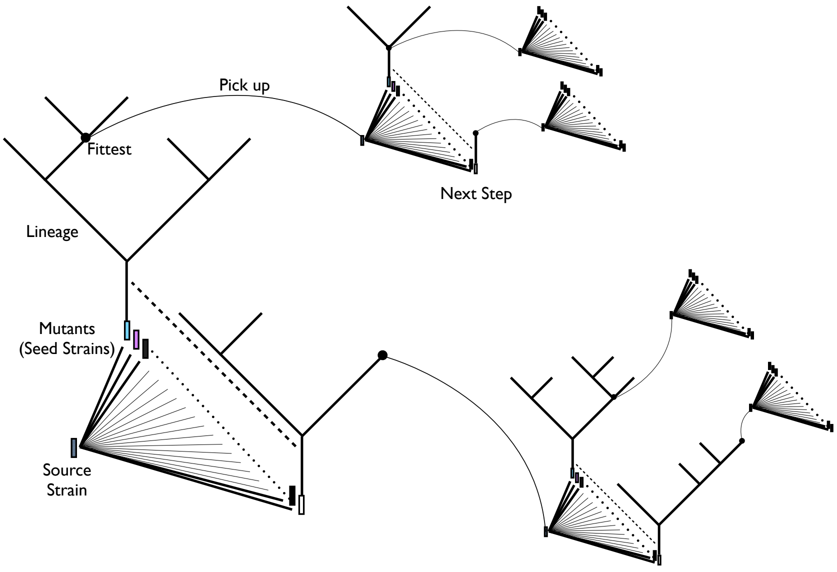


Figure 4.11: Conceptual diagram of mutational analysis. (Note that this is a highly conceptual sketch of the idea.) A source strain is set and its (first-step, single-point) mutants are systematically generated. Then, each of these mutants is traced as a seed strain. From a traced lineage (containing a seed strain and any generated strains), a fittest strain is selected for the next step analysis. From each selected strain, as a source strain, further-step mutants are systematically generated, and for them analysis is applied in the same manner. In principle the process can be recursively extended to an indefinite mutational depth.

deep in mutational pathways one can trace and analyse and how much one resorts to prune from a spectrum of mutants at a certain mutational step. Generally, the larger the size of strain and of instruction set, and the more kinds of mutation are enabled, the faster the combinatorial explosion becomes. It is not clear how many generations to track down in order to encounter interesting mutant strains either for the particular hand-designed prototype ancestor, or in more general cases. This is why heuristics are required.

Mutational analysis has to be able to only look to candidates that are interesting by some qualitative criteria; and to do so practically, it should be able to prune mutant strains by some quantitative criteria. It is initially worthwhile to estimate how expensive it is to screen candidates at one first mutational step. Investigation on quantitative pruning criteria is an effective step to take next, so that more mutational steps can be covered by analysis. The way to classify strains needs to be considered and explored further. Additionally, for each incubated mutant, the analysis should be able to pick up which strain expected to be generated in the lineage is the most interesting to trace further. Nonetheless, the pruning criteria can be used for this as well. Naturally, such a mutational analysis method has to have a reasonable way of strain classification as to candidates of evolutionary significance and interest. For strains to be evolutionarily significant, they must at least be viable, having potential for exponential growth of population, which is one of factors for the neo-Darwinian evolution besides self-reproducer population and mutational variation. On the other hand, for strains to be evolutionarily interesting in relation to the current research problem, they must be identified as having some kind of von

Neumann style self-reproduction architecture which has a genotype-phenotype mapping, or something else other than a pure self-copying style. Again, this identification, as mentioned in Section 3.5 in Chapter 3, is not straightforward to realise, and in fact is not going to be resolved in this thesis.

Results generated by using the built-in, pre-existing analysis tool of Avida are discussed in the next section. This is the initial step to create a framework of mutational analysis and a preparation leading to the subsequent section covering the enhancement of the analysis. In these sections, the first-step mutant strains of the prototype ancestor are focused upon. Following that, the automation of the analysis is described, covering the multi-step mutant analysis.

4.3 Avida *Analyze Mode*

The *analyze mode* is another mode of Avida, which provides various commands for studying organisms observed in the Avida standard mode (as introduced in Chapter 3). In this section, the original analyze mode is first explained, focusing on one relevant built-in command called `TRACE`, followed by the analysis of the prototype's mutants using this mode. Then, limitations in the analysis using the built-in `TRACE` command are identified, through which ideas for enhancement are spelt out.

4.3.1 *Analyze Mode* and the `TRACE` command

In the analyze mode, an experimenter can further study organisms obtained from standard Avida runs from a number of different aspects. Using provided commands, an experimenter would code a script in a file called `analyze.cfg`. Typically, "phenotypes" for specified "genotypes" are analysed. This mode presumes that "genotype" and strain are synonymous. In other words, for any seed strain input for analysis (i.e. a memory image, or a sequence of words), an organism with that strain (i.e., a memory image coupled with a virtual CPU) will be instantiated. This feature stems from the fact that the main purpose of trace is to analyse strains obtained from organisms that are actually observed in standard Avida experiments.

For viability research in the current study, an Avida command called `TRACE` is the most relevant. The analysis with this command will be referred to as the `TRACE` analysis, or simply as *trace*. Conducting the `TRACE` analysis with this command on some strain will be *tracing* a strain or a lineage which a strain is expected to generate. In brief, this command is used to trace a lineage of a given strain. Lineage here refers to a series of distinct strains that the strain is deterministically expected to produce as an ancestor under no perturbation. The starting strain will be referred to as a *seed strain.* A seed strain may turn out to be self-reproducer, or other classes of producers involving other distinct strains. As discussed in Section 4.2, patterns of lineages are diverse and will naturally differ depending on the seed strain.

More specifically, what is analysed by tracing a strain with the original analyze mode is a single branch of a lineage: that is, the analysis is a type of depth-first analysis, and it is presumed that a parent strain will not change after division, having one offspring. (This point is revisited and further discussed in Subsection 4.3.3.)

Refer to Appendix F for the function for the TRACE command of the original algorithm. In brief, the reproducibility for one organism of a given strain is analysed as follows:

```
 1: if organism of a given strain does not divide then
 2:    classify as non-dividing
 3: else
 4:    if the offspring is identical to the parent (immediate ancestor) then
 5:       classify as direct self-reproducer[5]
 6:    else
 7:       if the offspring is identical to any of the ancestors then
 8:          classify as indirect self-reproducer[6]
 9:       else
10:          classify as dividing (but not self-reproducer)
11:          incubate the offspring recursively
12:       end if
13:    end if
14: end if
```

A single cycle of this algorithm is referred to as an *incubation.* At a given incubation, an incubated organism can be classified as either non-dividing (at line 2), direct self-reproducer (at line 5), indirect self-reproducer (at line 8), and dividing (but not self-reproducer) (at line 10). A seed strain is an initially incubated strain, and as a result of a trace (i.e. a series of incubations), it will be classified as either direct or indirect infertile, direct or indirect self-reproducer, or fertile (but not self-reproducer). With this algorithm, only if the organism of an incubated strain divides and the offspring is not identical to any of ancestors, incubation occurs recursively. In practice, a cutoff time, or a maximum window of time to wait, is set to determine if a strain is infertile (or, if an organism divides or not). In order for a seed strain to be classified as some class that is at least fertile, division must have taken place (i.e., the h-divide instruction must have been executed, in the current context) before the cutoff time is reached; otherwise, the analysis run terminates classifying the organism of the incubated strain as non-dividing, and hence the seed strain as (direct or indirect) infertile.

Here, the word *descendant* is used for whatever strain is generated from a seed strain; whereas the word *ancestor* for whatever strain from which a strain (directly or indirectly) is produced, including a seed strain. In this algorithm, ancestors imply previously incubated strains. The words *parent* and *offspring* are used to describe an immediate relationship between strains after division. Thus, a parent is an immediate ancestor, and an offspring an immediate descendant. (This parent-offspring relationship will be revisited later in Subsection 4.3.3, which is relevant to the way this analysis traces a single branch of a lineage.)

A trace continues as long as a distinct strain is produced, so a single trace would consist

[5] Notice that here, only simple self-reproducer strains (as opposed to collective self-reproducer strains) are dealt with due to the way the analyze mode traces. This point will be revisited later in this section.

[6] Notice that the class of indirect self-reproducer that an incubation with this algorithm can classify is a partial class of, and is not necessarily equal to, the class of indirect self-reproducer introduced in Section 4.2. The relevance of this will be discussed later in this section.

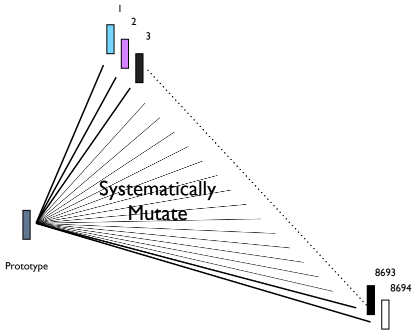


Figure 4.12: Schematic point-mutation spectrum generation of the prototype.

of incubations of a seed strain and its descendant strains. A trace starts with incubating a seed strain, possibly classifying it as direct infertile or direct self-reproducer. If the seed strain is classified as neither of these, the trace continues incubations of descendant strains recursively, until an incubated strain turns out to be either infertile (thus concluding the seed strain is indirect infertile) or self-reproducing (thus concluding the seed strain is direct self-reproducer or indirect self-reproducer, either of which is fertile). In practice, a depth limit, or a maximum number of incubations, needs to be set for a single trace. If the depth limit is reached, the seed strain is classified merely as fertile. In this analyze mode, seed strains that are classified as self-reproducer (whether direct or indirect) are called viable (i.e., it is a subset of what is qualitatively defined as in Section 4.2).

For each use of the `TRACE` command of a seed strain, a *trace file* (see Subsection 3.4.2 in Chapter 3) is generated. A trace file contains full step-by-step CPU state transitions for distinct strains encountered in a trace of a seed strain. See Appendix C for the original format of this type of file.

4.3.2 *Analyze Mode*'s Analysis of First-step Mutants

To evaluate the analysis performed by the `TRACE` command of the analyze mode, the prototype's first-step mutants were analysed.

Here, first-step mutants mean mutants of the prototype possibly obtained through a single point-mutation. First-step mutants of the prototype were systematically enumerated. The strains of these mutants were obtained from the prototype's initial memory image by sequentially replacing a word value in each memory location of the genome with each of the other possible word values and by expressing this change in a corresponding phenome. As the prototype size is 644 (the genome size being 322) and as the instruction set size is 28 (alternative words for each memory location being $27 = 28 - 1$), there are a total of 8694 ($= 322 \times 27$) possible one-point mutants (see Figure 4.12 for a conceptual schematic).

This size of the spectrum (8694) is a much more tractable size of space than the entire space of possible strains ($\sum_{l=1}^{L} w^l$, where w denotes the number of the possible word values in the current setting ($= 28$), l the strain length, and L the maximum strain length allowed in Avida, not explicitly limited), or the subspace of the strains with the same length as the prototype (28^{644}, the number of the possible word values to the power of the prototype's

Class		**Mutants**	
Self-Reproducer	**Direct**	892	(10%)
	Indirect	1	(0%)
Infertile	**Direct**	5214	(60%)
	Indirect	2587	(30%)
	Total	8694	

Table 4.1: Mutant classification using the original analyze mode based solely on the trace on a single branch of each lineage.

length).

For this specific study (and for the subsequent studies in the current chapter), the cutoff time is set to twice as long as the prototype's gestation time (i.e. $104436 = 2 \times 52218$). This setting is reasonable considering the fact that the candidates can have varied gestation times. It is possible that, with a longer runtime, organisms that are classified as non-dividing might be reclassified as dividing; they are, however, unlikely to be selectively favoured over the original prototype ancestor in the Avida system, where fitness hinges on reproduction rate, hence gestation time. On the grounds of this, one can discard these as non-dividing organisms.

After a few preparatory experiments with the mutants being seed strains, it turned out that the depth limit= 5 was sufficient for these 8694 mutants, except that, for 2 strains, a depth limit= 381 was required. So, within this analysis which considers only a single branch of lineage, there were no seed strain that is unclassified due to the depth limit. See Table 4.1 for mutant classification based only on the trace on a single branch of lineage. The majority (90%) of the mutants turned out to be direct or indirect infertile. It is noticeable that there is 1 indirect self-reproducer mutant. In the analysis, this is classified as dividing at the first incubation, then as direct self-reproducer at the second incubation. So, this “indirect self-reproducer” does not correspond to the indirect self-reproducer class classified within a single cycle of incubation (line 8 in the above algorithm), but rather to the class classified based on the logical classification introduced in Section 4.2.

4.3.3 *Analyze Mode*'s Assumption Revisited

As introduced in the previous subsection, the `TRACE` command is a tool which the analyze mode provides for viability analysis, but the analysis is somewhat coarse-grained. This is the reason why the classification based on this original analyze mode is unsatisfactory. If an offspring is determined to be identical to the initial memory image of the parent, the parent strain is classified automatically as self-reproducing, regardless of whether the parent itself changes or not. Under the scheme where only a single branch of lineage is traced, even a truly viable organism could be erroneously classified as non-viable. Moreover, as classification of self-reproducer strains is based on comparison of strains of a single branch of lineage, production patterns of strains that can be captured is not satisfactorily precise.

One underlying assumption highlighted in the use of the analyze mode is that any parent organism which divides, has its final state and one child (i.e. one parent and one child after division). The same phenomenon, conversely, can also be viewed as having two offspring.

Both views appear to be legitimate in the Avida literature. For example, on the one hand it is stated (Ofria & Wilke, 2004, pp.199-200):

> In most natural asexual organisms, the process of division results in organisms literally splitting in half, effectively creating two offspring. Thus, the default behaviour of Avida is to reset the state of the parent's CPU after the divide, turning it back into the state it was in when it was first born. In other words, all registers and stacks are cleared, and all heads are positioned at the beginning of the memory.

On the other hand, in the illustration of the memory allocation and division cycle, referred to from that paragraph, they also state: "The h-alloc command extends the memory, so that the program of the child organism can be stored. Later, on h-divide, the program is split into two parts, one of which turns into the child organism."

The Avida documentation[7] describes division as something configurable. There is an explanation about it found in a section on how a child is divided as follows:

> After a divide, we mark that we no longer have a mal (Memory ALlocation) active. If the parent is reset (i.e., we have two offspring, not a parent and child) we need to make sure not to advance the IP of the parent. The reset parent has its IP placed at the beginning of its genome, and we want to leave it there to execute the very first instruction.
>
> Finally, we tell the organism to activate the divide and do something with the child. Give the child to the population (or the test CPU as the case may be) to be dealt with, and reset the parent if we're splitting into two offspring.

As described in the documentation, there is a variable in the Avida main configuration file for divide, which offers several options regarding the divide method including one where "divide resets state of mother (after the divide, we have 2 children)."

At the source code level, in the definition of the class representing an Avidian organism, there are variables for the initial memory image, the final memory image, and the child memory image; and the pair of the final state and the child state can be regarded and treated as two offspring. However, the way the analyze mode is implemented, does not regard or treat an organism as having two equivalent offspring (i.e., one of the offspring is closely associated with the parent, while the other is not). In other words, the initial memory image is assumed not to change, and as a result, incubation of trace is applied only to the child memory image (as an initial memory image for a next incubation) and not to the final memory image also. Even if the option is selected in the configuration for an organism to have two offspring virtually, the hardwired structure in the trace does not support the idea.

Although there is a conspicuous view that after division there are two offspring in Avida, the analyze mode is designed based on an underlying view that a parent has its final state and one offspring, where the final state of the parent is not regarded or treated as

[7] In "4. Dividing off the Child" in "Guide to an Avidian Life Cycle" (2006). This is a part of the documentation included in the package of the version 2.10.0 of Avida. The same text as quoted here can be found at: `https://github.com/devosoft/avida/wiki/Development-%7C-Tutorial-%7C-Life-Cycle`.

another offspring. This discrepancy appears to be a kind of double standard. In effect, the analyze mode assumption that a parent remains the same and that there is one offspring and that tracing the offspring suffices even when a parent is regarded as dividing into two offspring seems to be the reason for the peculiar way of tracing a single branch of lineage. The analyze mode, too, should hold the view that a parent may change and that thus division makes two equivalent offspring (as opposed to what used to be called a final memory image and a child memory image) and should be able to incubate both offspring and trace both branches of any lineage.

Generally speaking, it is perfectly possible that the offspring that is previously the final state of the parent (the parent after division) may be somehow different from the (initial) parent and thus is worth tracing further. The current study adopts the point of view that "after division there are two offspring", based on which the analyze mode was modified. This modification will be discussed in Section 4.4.

Though less relevantly, the original analyze mode's assumption that a parent (initial memory image) will remain the same after division also implies that a parent will not change in size after division either. It is now assumed, to the contrary, that either offspring may be shorter or longer. In the current condition, this depends on the positions of the read head and the write head at division, because the `h-divide` instruction is the only instruction available that is capable of triggering division and it references the positions of the read head and the write head to determine the memory image of an offspring (and indirectly the new memory image of the "parent") (see Subsection 3.3.3 in Chapter 3). By default in the setting used in Chapter 3, an offspring is allowed to be up to twice as long as the parent, and the execution of the instruction for dividing off an offspring will discard whatever memory image is left after the write head. Thus in fact, the parent may also be longer or shorter than the original, too, depending on the position of the read head at division.

Reclassification: Considering Untraced Lineages

The analysis with the analyze mode was based only on a single branch of each lineage. Therefore, in the sense that it is disregarding untraced branches of lineages (or, *untraced lineages*), this classification is incomplete. Those with untraced branches of lineages should be correctly regarded as unclassified here. Simple direct infertile and simple direct self-reproducer can be considered as fully traced, as it can be classified straightforwardly.

Conceivably, some of those strains previously classified as direct or indirect self-reproducer seed strains may be with untraced lineages and require further analysis for a more precise classification. Similarly, some of those previously classified as indirect infertile seed strains may have untraced lineages and require further analysis. This infertile group of mutants has to be reconsidered in particular because, for those classified as infertile (whether direct or indirect), there may be some that are incorrectly classified not only as infertile but also as not viable.

One could analyse and classify first-division patterns manually to find somewhat more mutants of "fully traced" lineages, but the problem situation does not change because there are mutants having longer and larger lineages. Naturally, the more distinct strains are involved within a lineage, the more difficult to analyse the pattern and classify the seed

Class	**Mutants**	
Viable	871	(10%)
Non-Viable	5214	(60%)
Unclassified	2609	(30%)
Total	8694	

Table 4.2: Mutant classification by viability considering untraced lineages. Simple direct self-reproducer and simple direct infertile are considered as having fully traced lineages, whereas other classes are regarded as having untraced lineages, hence unclassified.

strain by viability. So manual classification is not a focus in the current study. That said, only simple direct infertile strains and simple direct self-reproducer strains are manually picked out here, since they are ones that are easier to classify and whose evolutionary significance is obvious. To say the least, simple direct infertile strains are not viable, and simple direct self-reproducer strains are viable. See Table 4.2 for a new classification reflecting this consideration of untraced lineages.

It is apparent that the majority of the mutants does not reproduce anything, hence has no evolutionary future (i.e., the non-viable 60% versus the viable 10%), while there still remains as many as 30% of unclassifiable mutants. These unclassifiable mutants are not negligible candidates, since one can not know *a priori* how frequently they are viable or even fertile. The classification should be automated as well as systematic, ideally even applicable to different sets of candidates.

Limitations: Unclassified Patterns

To emphasise the problem situation, the limitation of the current analysis tool surfaces when it encounters either cases where (a) both of the two offspring that a parent produces are non-identical from the parent and from each other, or where (b) one of the two offspring is identical to the parent but the other is non-identical. As identified above, the built-in analysis tool assumes that tracing one of them suffices to analyse the lineage in effect, whether an act of division produces two identical offspring or not. Again, when the built-in analysis tool searches for descendant generations further down, what it recursively incubates for tracking is only one of the two offspring that is arbitrarily labelled as being divided off from the parent.

Consider, for example, an organism producing one infertile offspring and one fertile offspring in a deterministic environment without mutation. If the fertile offspring produces a self-reproducer offspring, the original organism should be classified as (indirect) self-reproducer in that it exhibits the lineage with a self-reproducer. However, if the analysis tool is set to trace one of the offspring at the first division, which happens to be infertile, then the other offspring, which happens to produce a self-reproducing offspring, is not further traced, and the original organism ends up being classified as “non-viable”. This is not a proper viability classification because self-reproducers may be viable, with an evolutionary potential. In other words, for some candidates which start out producing two different offspring, only a subset of (or rather, a “branch” of) the whole expected lineage is revealed through the analysis tool; therefore, viability analysis cannot be guaranteed.

In sum, the analyze mode's analysis proceeds with the incubation time limit as well as with the recursion limit (i.e., how many generations per lineage to track down). In addition, there is the limitation due to the coarse-grained analysis pointed out above. Because of these limits, not all of the candidates have been properly analysed for (or classified by) viability, but rather for strain fertility (or organism dividability), although the majority of the candidates analysed turned out to be classified as simple direct infertile, hence non-viable.

4.4 Enhancement of the *Analyze Mode*

On the basis of the understanding of the analysis using the original analyze mode described in Section 4.3, an enhanced analysis method is proposed, and an implementation of enhanced analysis and its application to first-step mutants of the prototype are demonstrated. In doing so, a feasible scale of analysis is considered, as the enhanced analysis tool will now need to be able to handle potentially more numerous strains than the original analysis. With this being a pilot study, the plausibility of systematic multi-step mutation analysis is also considered.

The existence of the still unclassified candidates with untraced lineages highlighted the fact that there is a subtlety in the concept of viability. Specifically, the original analysis tool turns out to be only capable of revealing a single possible lineage pathway that a strain is expected to exhibit. To recapitulate, it traces only one (arbitrarily pre-determined as a "child") of the two offspring at each division, whereas the sub-lineage extending from the other offspring is not further traced. In this sense the viability analysis with it is not complete or comprehensive. It is therefore necessary to enhance the existing classification of candidates, in a systematic (and automated) way based on viability (and eventually, towards a way based on the reproduction mode) in the current system.

First and foremost, a function for the `TRACE` command was to be modified so that it allows the (breadth-first) creation of the lineage pathways, and the tracing of the lineage, for a given individual organism. Here, the idea is to represent lineage pathways using reproductive relationships among strains as production graphs, since analysis of matrices of production rates will provide more quantitative insights about dynamics of lineages (or networks of strains) represented by graphs.

4.4.1 Quantifying Viability

In this subsection, viability is defined quantitively. More specifically, viability is quantified on the basis of analysis of eigenvalues/eigenvectors of *production rate matrices*. First of all, to define production rate matrices, let a node denote a distinct strain, a directed edge a (re-)production, and an edge weight the production rate. Structures of production graphs, which are directed graphs, were already introduced in Section 4.2. Here, production rates, represented as edge weights in production graphs, are also introduced, and production graphs are now weighted directed graphs. In brief, a production graph represents a network of strains that produce one another. Distinct nodes represent distinct strains. Directed edges represent acts of production linking nodes of a strain that produces and of another strain that is produced. Weights of edges are production rates that are determined by gestation time. As an organism either has 0 or 2 offspring in Avida, a node either has 0 or 1 edges going out from itself, depending on whether the organism of the strain divides and produces another or not. Simply put, a network of strains is represented by a production graph, and by a matrix of production rates, with each element denoting a rate for an organism of a strain (corresponding a node) to produce another.

If in a production graph (or any subgraph) all the rates are the same and all the nodes have two edges coming in, then the lineage that the graph represents is expected to give rise to exponential growth of population (e.g., immediate self-reproducing strains like the standard Avida self-copiers and the von Neumann style prototype). If not, by quantitatively analysing production graphs, one can still gain insight into the population growth and hence analyse the viability.

The term viability is now to be used in a subtle but practical sense, meaning the potential for exponential growth which is a characteristic of self-reproducer, distinguishable from other types of population growth[8]. Under the enhanced analysis proposed in this chapter, a strain is viable if there is at least one strain that grows exponentially in its expected lineage.

Eigenvalues and eigenvectors of a production rate matrix indicate the population growth of the strains which is defined as a function of time. These can be used in order to distinguish distinct strains which will potentially give rise to exponential growth. In summary, eigenvalues and eigenvectors are solutions of dynamical systems of population when population growth is expressed as exponential. Particularly, eigenvalues are coefficients of the power of exponential functions (representing how population grows in terms of an exponential function), and corresponding eigenvectors represent proportions of the population of distinct strains involved. Eigenvalues can be complex numbers in general, but any imaginary part is associated with such population growth as oscillation, and it is not to be further considered in the current research. This is because, qualitatively speaking, oscillation is indicative of a "cyclic" kind of population dynamics. Such dynamics would entail a negative proportion of at least one subpopulation (i.e. at least one component of an eigenvector with a different sign), and such proportions of subpopulations would not be actually realised in Avida. Eigenvectors representing such proportions of subpopulations can reasonably be regarded as invalid in Avida (and in practice, of eigenvalues with valid corresponding eigenvectors, only real eigenvalues were encountered, as explained later). Analytically, a positive real eigenvalue is of interest, because it implies viability as the magnitude of exponential growth. The higher this value is, the faster the production rate, and vice versa. If there are multiple such eigenvalues from a production rate matrix, these correspond to multiple exponentially growing strains within a dynamical system. Among such eigenvalues (i.e. sets of strains), the biggest eigenvalue (i.e. the fittest set of strains) is expected to displace others in the neo-Darwinian evolution.

Before moving on to illustrations using hypothetical examples of production graphs (not yet corresponding to any specific cases encountered empirically in Avida), note that this quantitative analysis applies continuous approximation on the discrete system of individuals. In other words, this method assumes that population dynamics can be repre-

[8] For example, there can be constant, linear growth, polynomial growth, exponential decay, oscillation, etc. (See Szathmáry & Maynard Smith, 1997, for several types of population growth other than exponential, which is the type dealt with in this chapter.)

sented as a system of differential equations. Also, this approximation is deterministic as well (just as much as the analyze mode is deterministic), in the sense that it abstracts initial condition, and stochastic and spatial effects likewise (which can be observed in normal evolutionary Avida runs). This type of approximation using algebraic tools is justifiable on the grounds of principles given by: Maynard-Smith et al. (1989); Hirsch et al. (2004); Nowak (2006); Jain & Krishna (2006).

In preparation for subsequent examples, let r_n denote the production rate of a strain s_n. Let N be the size of a particular production graph, or the number of distinct strains involved. s_0 is a seed strain, and $s_1..s_{N-1}$ are descendant strains generated from s_0. Assume a rate represented by r is positive real value, unless explicitly specified to be 0. Let λ denote a general eigenvalue and $\mathbf{v}$ a general eigenvector. So λ_1, λ_2, ..., λ_N are eigenvalues, and $\mathbf{v_1}$, $\mathbf{v_2}$, ..., $\mathbf{v_N}$ corresponding eigenvectors. By definition, there can be as many eigenvalues as, or fewer than (in cases of repeated eigenvalues), the number of distinct strains in a lineage (N) (which translates as the dimension of the dynamical system); and eigenvectors, which are non-zero vectors, are as many as, or fewer than (in cases of linearly dependent eigenvectors), the number of eigenvalues.

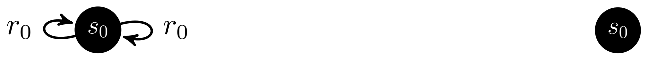
Lineages consisting of a single strain (i.e. production graphs involving a single strain), correspond to 1×1 production rate matrices. Naturally, lineages consisting of multiple distinct strains, hence production graphs involving multiple nodes, correspond to production rate matrices larger than 1×1. In analyses of such production rate matrices, when there are multiple eigenvalues, the maximum of the positive real parts of these eigenvalues is of particular interest as mentioned. This indication is valid only if the components of the corresponding eigenvector have the same signs (irrespective of zeros), since population of any strain must be natural numbers (i.e. non-negative) in order to be valid in the actual Avida world.

Example 1: Simple Direct Strains

The case of the simple direct self-reproducer strain is the most relevant and straightforward, which serves as the basis of this analysis. See Figure 4.13a for the production graph of a simple direct self-reproducer strain. The production rate matrix of this is represented as a 1×1 matrix $A = \begin{bmatrix} r_0 \end{bmatrix}$. For this, the eigenvalue is $\lambda_1 = r_0$, and the corresponding eigenvector is $\mathbf{v_1} = k \begin{bmatrix} 1 \end{bmatrix}$, where k is a non-zero arbitrary constant.

The population growth of this type of strain (as a seed strain), when written as an exponential function over time, has the eigenvalue as a coefficient, which is the production rate. Therefore, the population growth naturally hinges on the production rate in such a strain. The eigenvector, in this case, signifies the axis corresponding to this (seed) strain's population, along which the total population grows.

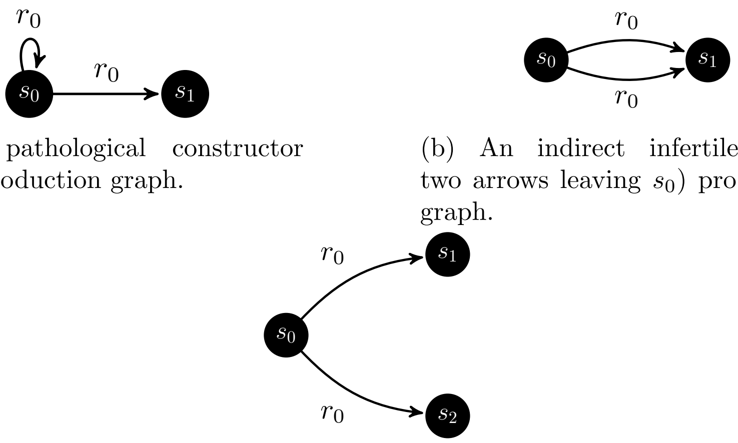
A special case of the above, where $r_0 = 0$, corresponds to a simple direct infertile strain. See Figure 4.13b for the production graph. This is represented as a 1×1 matrix $A = \begin{bmatrix} 0 \end{bmatrix}$. Accordingly, the eigenvalue is $\lambda_1 = 0$, and the corresponding eigenvector is $\mathbf{v_1} = k \begin{bmatrix} 1 \end{bmatrix}$, where k is an arbitrary scalar. Now, the eigenvalue is 0, so the population growth of this type of strain, when written as an exponential function over time, has the coefficient of 0; that is, the population stays constant. This type of strain, no matter how many of them might be seeded, does not give rise to a growing population, but a constant



(a) A simple direct self-reproducer strain production graph.

(b) A simple direct infertile strain production graph.

Figure 4.13: Production graphs of a simple direct infertile strain and a simple direct self-reproducer strain. Throughout the discussion, it is assumed $r_0 \geq 0$. Subfigure 4.13b is a special case of Subfigure 4.13a where $r_0 = 0$.



(a) A pathological constructor strain production graph.

(b) An indirect infertile (with two arrows leaving s_0) production graph.

(c) An indirect infertile (with one arrow leaving s_0, leading to two infertile strains) production graph.

Figure 4.14: Production graphs of indirect simple infertile strains.

population.

Example 2: Indirect Simple Infertile Strains

Next to a simple direct self-reproducer, a pathological constructor is straightforward to consider. See Figure 4.14a for the production graph. The production rate matrix of a pathological constructor strain can be represented as a 2×2 matrix $A = \begin{bmatrix} 0 & 0 \\ r_0 & 0 \end{bmatrix}$.

For this, the two eigenvalues are repeated: $\lambda_1 = \lambda_2 = 0$. As $\lambda_1 = \lambda_2$, naturally, the corresponding eigenvectors are linearly dependent: $\mathbf{v_1} = \mathbf{v_2} = k \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, where k is a non-zero arbitrary constant.

The population growth of this type of strain, like the infertile strain demonstrated above, is not exponential, but zero, when seeded with the strain s_1 (corresponding to the second component of the eigenvector, which is a non-zero value). This is what this quantitative analysis can verify. Qualitatively, however, it is known that this type of strain may have an impact on a finite population (as mentioned in Section 4.2) as an organism of the strain s_0 constantly produces organisms of the infertile strain s_1.

Now examples of indirect infertile, somewhat similar to pathological constructor, are considered. Production graphs are shown in Figures 4.14b and 4.14c. The production rate matrix of the production graph of Figure 4.14b, where two offspring are identical but distinct from the parent that is the seed strain, is represented as a 2×2 matrix

$A = \begin{bmatrix} -r_0 & 0 \\ 2r_0 & 0 \end{bmatrix}$, eigenvalues: $\lambda_1 = 0$, $\lambda_2 = -r_0$; eigenvectors: $\mathbf{v_1} = k_1 \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, where k_1 is a non-zero arbitrary constant. $\mathbf{v_2} = k_2 \begin{bmatrix} 1 \\ -2 \end{bmatrix}$, where k_2 is a non-zero arbitrary constant.

The eigenvalue λ_1 and the corresponding eigenvector $\mathbf{v_1}$ indicate that with the strain s_1 (corresponding to the second component of the eigenvector $\mathbf{v_1}$, the value of which is 1), and without the strain s_0 (corresponding to the first component of the eigenvector $\mathbf{v_1}$, the value of which is 0), the total population stays constant. The eigenvalue λ_2 and the corresponding eigenvector $\mathbf{v_2}$ indicate something that cannot be actually realised in Avida: the eigenvector $\mathbf{v_2}$ has components of different signs, indicating an impossible proportion of the strains seeded (i.e., s_0:$s_1 = 1 : -2$) that can lead to the total population growth to be exponential decay as indicated by the negative eigenvalue $\lambda_2 = -r_0$, where population of one of the seeded strains would have to be negative.

Likewise, the production rate matrix of the production graph of Figure 4.14c, where two offspring are not identical and both are distinct from the parent that is the seed strain, is represented as a 3×3 matrix $A = \begin{bmatrix} -r_0 & 0 & 0 \\ r_0 & 0 & 0 \\ r_0 & 0 & 0 \end{bmatrix}$. For this, eigenvalues are: $\lambda_1 = \lambda_2 = 0$; $\lambda_3 = -r_0$. The corresponding eigenvectors are: $\mathbf{v_1} = \mathbf{v_2} = \begin{bmatrix} 0 \\ l \\ m \end{bmatrix}$, where l and m are non-zero arbitrary constants; and $\mathbf{v_3} = k \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}$, where k is a non-zero arbitrary constant.

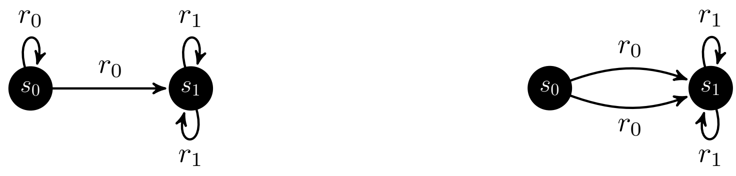
The eigenvalues $\lambda_1 = \lambda_2$, and the corresponding eigenvectors $\mathbf{v_1} = \mathbf{v_2}$, indicate that there are two infertile strains (i.e. s_1 and s_2, which do not contribute to population growth). These eigenvectors indicate that with the arbitrary proportion of these strains, the total population growth indicated by these eigenvalues (i.e. constant) is realised. The eigenvalue λ_3 and the corresponding eigenvector $\mathbf{v_3}$ are not to be further considered, for the same reason as the above.

Example 3: Indirect Simple Self-Reproducer Strains

Next, two examples of indirect simple self-reproducer are considered. The production rate matrix of an indirect simple self-reproducer strain shown in Figure 4.15a, where one offspring is identical to the parent and another offspring is distinct from them and is a self-reproducer strain, is represented as a 2×2 matrix $A = \begin{bmatrix} 0 & 0 \\ r_0 & r_1 \end{bmatrix}$.

The two eigenvalues are: $\lambda_1 = 0$, $\lambda_2 = r_1$. The corresponding eigenvectors are: $\mathbf{v_1} = k_1 \begin{bmatrix} r_1 \\ -r_0 \end{bmatrix}$, where k_1 is a non-zero arbitrary component; and $\mathbf{v_2} = k_2 \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, where k_2 is a non-zero arbitrary component.

The eigenvalue λ_1 and the corresponding $\mathbf{v_1}$ are not considered, as Avida cannot realise the population growth indicated by them. As for the eigenvalue λ_2 and the corresponding $\mathbf{v_2}$, it is indicated that the strain s_1 (represented by the second component of the eigen-



(a) An indirect simple self-reproducer strain (with one arrow leaving, another looping back to, s_0) production graph.

(b) An indirect simple self-reproducer strain (with two arrow leaving s_0) production graph.

Figure 4.15: Production graphs of indirect simple self-reproducer strains.

vector $\mathbf{v_2}$) is a self-reproducer exhibiting exponential growth with the rate r_1 (represented by the eigenvalue λ_2).

Likewise, the production rate matrix of an indirect simple self-reproducer strain shown in Figure 4.15b, where two offspring are identical and distinct from the parent and are a self-reproducer strain, is represented as a 2×2 matrix $A = \begin{bmatrix} -r_0 & 0 \\ 2r_0 & r_1 \end{bmatrix}$.

The two eigenvalues of this are: $\lambda_1 = -r_0$; $\lambda_2 = r_1$. The corresponding eigenvectors are: $\mathbf{v_1} = k_1 \begin{bmatrix} r_0 + r_1 \\ -2r_0 \end{bmatrix}$, where k_1 is a non-zero arbitrary component; and $\mathbf{v_2} = k_2 \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, where k_2 is a non-zero arbitrary component.

Again, the eigenvalue λ_1 and the corresponding $\mathbf{v_1}$ are not considered, as Avida cannot realise the population growth indicated by them (i.e., as both r_0 and r_1 are positive, one of the components of $\mathbf{v_1}$ is necessarily negative). On the contrary, the eigenvalue λ_2 and the corresponding $\mathbf{v_2}$ indicate that with the strain s_1 in the absence of the strain s_0 (i.e. $s_0{:}s_1 = 0 : 1$), the population would exponentially growth at the rate of r_1 (indicated by the eigenvalue).

Example 4: Collective Self-Reproducer Strains

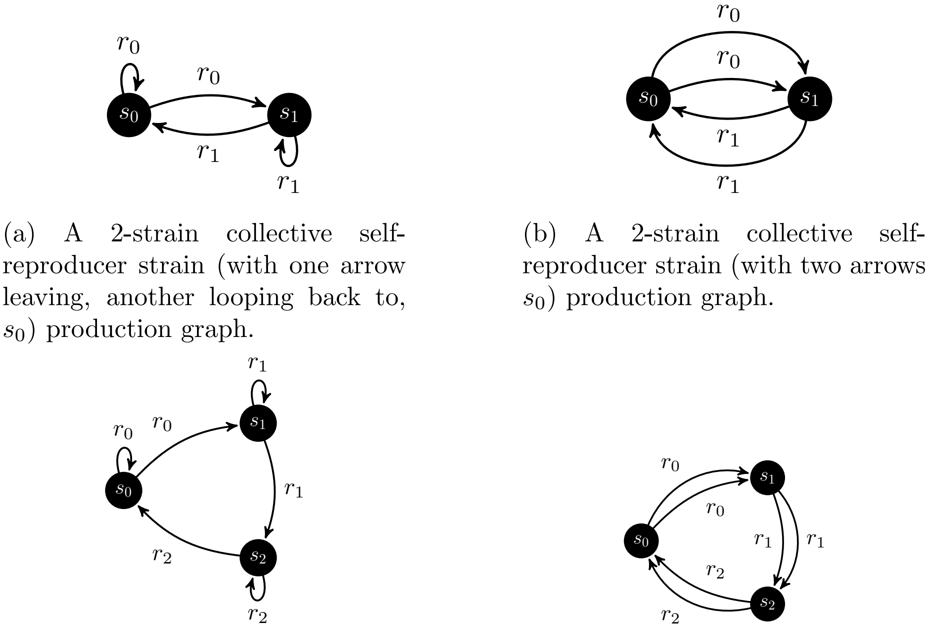
Lastly, four examples of collective self-reproducer are considered.

The production rate matrix of a collective self-reproducer strain is shown in Figure 4.16a, where one offspring is identical to the parent and another offspring is distinct from them, and where the distinct offspring likewise produces one offspring identical to itself and another offspring that is identical to the seed strain, is represented as a 2×2 matrix $A = \begin{bmatrix} 0 & r_1 \\ r_0 & 0 \end{bmatrix}$.

The eigenvalues are: $\lambda_1 = +\sqrt{r_0 r_1}$; $\lambda_2 = -\sqrt{r_0 r_1}$. The corresponding eigenvectors are: $\mathbf{v_1} = \begin{bmatrix} \sqrt{r_1} \\ \sqrt{r_0} \end{bmatrix}$; $\mathbf{v_2} = \begin{bmatrix} \sqrt{r_1} \\ -\sqrt{r_0} \end{bmatrix}$.

What this quantitative analysis of eigenvalues and eigenvectors shows is that with the proportion $s_0{:}s_1{=}\sqrt{r_1} : \sqrt{r_0}$, the total population growth is exponential with the rate of $+\sqrt{r_0 r_1}$. If $r_0 = r_1$, the exponential growth rate is r_0 and the proportion of strains in population that realises it is $s_0{:}s_1{=}1 : 1$. The eigenvalue λ_2 and the corresponding eigenvector $\mathbf{v_2}$ indicates exponential decay, but with an illegitimate proportion of subpopulations of strains.

The production rate matrix of a collective self-reproducer strain shown in Figure 4.16b,



(a) A 2-strain collective self-reproducer strain (with one arrow leaving, another looping back to, s_0) production graph.

(b) A 2-strain collective self-reproducer strain (with two arrows s_0) production graph.

(c) A 3-strain collective self-reproducer strain (with one arrow leaving, another looping back to, s_0) production graph.

(d) A 3-strain collective self-reproducer strain (with two arrows s_0) production graph.

Figure 4.16: Production graphs of collective direct self-reproducer strains.

where two offspring are identical and distinct from the parent and their produced offspring are identical to the seed strain, is represented as a 2×2 matrix $A = \begin{bmatrix} -r_0 & 2r_1 \\ 2r_0 & -r_1 \end{bmatrix}$.

The general eigenvalue λ is given as the solutions of: $(\lambda + r_0)(\lambda + r_1) - 4r_0r_1$, and the corresponding eigenvector $\mathbf{v}$ is given in the form of: $\mathbf{v} = k \begin{bmatrix} \lambda + r_1 \\ 2r_0 \end{bmatrix}$, where k is a non-zero arbitrary component.

For simpler forms of solutions, suppose $r_0 = r_1$, then the eigenvalues are: $\lambda_1 = r_0$; $\lambda_2 = -3r_0$. The corresponding eigenvectors are: $\mathbf{v_1} = k_1 \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, where k_1 is a non-zero arbitrary component; and $\mathbf{v_2} = k_2 \begin{bmatrix} 1 \\ -1 \end{bmatrix}$, where k_2 is a non-zero arbitrary component.

The eigenvector $\mathbf{v}_1$ together with the corresponding eigenvalue λ_1's real part implies that a population of the strains represented by those two components (i.e., the strains s_0 and s_1 in the graph) will grow exponentially in a collective way. The eigenvector $\mathbf{v}_2$'s components have different signs, indicating an illegitimate proportion of strains in population $s_0 : s_1{=}1 : -1({=}{-1} : 1)$. In a hypothetical situation where that proportion is realised, the population growth would exhibit exponential decay, as indicated by the eigenvalue λ_2.

The production rate matrix of a collective self-reproducer strain shown in Figure 4.16c, involving three distinct strains, where each mutually produces one identical, one non-identical offspring, is represented as a 3×3 matrix $A = \begin{bmatrix} 0 & 0 & r_2 \\ r_0 & 0 & 0 \\ 0 & r_1 & 0 \end{bmatrix}$. There is only one real eigenvalue: $\lambda_1 = \sqrt[3]{r_0 r_1 r_2}$. The corresponding eigenvector is: $\mathbf{v_1} = \begin{bmatrix} r_0 r_1 r_2 \\ r_0 \lambda^2 \\ r_0 r_1 \lambda \end{bmatrix}$.

For simpler forms of solutions, suppose $r_0 = r_1 = r_2$ then the eigenvalue is: $\lambda_1 = r_0$. The corresponding eigenvector is: $\mathbf{v_1} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$.

In this case, if these three strains have the same production rate ($r_0 = r_1 = r_2$), the total population growth, where the three strains exist equally (with the proportion $s_0 : s_1 : s_2 = 1 : 1 : 1$), is exponential with rate of r_0.

The production rate matrix of a collective self-reproducer strain shown in Figure 4.16a, involving three distinct strains, where each mutually produces two offspring not identical to itself, is represented as a 3×3 matrix $A = \begin{bmatrix} -r_0 & 0 & 2r_2 \\ 2r_0 & -r_1 & 0 \\ 0 & 2r_1 & -r_2 \end{bmatrix}$. The general eigenvalue λ is the solutions of: $(\lambda + r_0)(\lambda + r_1)(\lambda + r_2) - 8r_0 r_1 r_2$, and the corresponding eigenvector $\mathbf{v}$ is given in the form of: $\mathbf{v} = k \begin{bmatrix} 2r_2(\lambda + r_1) \\ (\lambda + r_0)(\lambda + r_1) \\ 4r_0 r_2 \end{bmatrix}$, where k is a non-zero arbitrary constant.

For simpler forms of solutions, suppose $r_0 = r_1 = r_2$, then the real eigenvalue is: $\lambda_1 = r_0$. The corresponding eigenvector is: $\mathbf{v_1} = k_1 \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$, where k_1 is a non-zero arbitrary constant.

In this case, too, if these three strains have the same production rate ($r_0 = r_1 = r_2$), the total population growth, where the three strains exist equally (with the proportion $s_0 : s_1 : s_2 = 1 : 1 : 1$), is exponential with rate of r_0.

In practice, individual self-reproducers (i.e. simple direct self-reproducer strains) are searched for in the current analysis. Theoretically, collective exponential growth as opposed to individual exponential growth can take place, and it is not algorithmically difficult to detect collective exponential growth (i.e., a positive real eigenvalue associated with a valid eigenvector with more than 1 components, as shown so far). However, this type of reproducers (if there appears any) are not in the scope of this enhanced analysis as retained for further analysis and investigation. This is because it is not as straightforward to computationally define a "collective self" (comprised of multiple strains) as to define a "individual self", and therefore it is unclear yet how to classify such collective reproduction by such a self of multiple strains, much less defining "collective von Neumann style self-reproduction".

4.4.2 Modifying the *Analyze Mode*

For the creation of a production rate matrix and the tracing of a network of strains which a production rate matrix represents, a mechanism to store not only memory image but also strain relationships was accordingly implemented for the `TRACE` command. The original analyze mode adopted a depth-first trace (or more strictly, a first move of a depth-first trace), due to which there was both a depth (or recursion) limit and a maximum number of distinct strains to be configured. As opposed to it, the (newly implemented) breadth-first trace only requires a maximum number of distinct strains. Refer to Appendix F for the modified algorithm.

More specifically, a `struct` was added to help maintain strain statuses such as: whether it is incubated; whether it divides; and if it does, which strains are its offspring. A queue structure was made use of in order to implement the breadth-first tree creation and trace which the new analysis intended to achieve. This queue manages the incubation of strains. Whenever a strain is instantiated, an organism corresponding to it is instantiated as well. Strains and organisms are corresponding, but are treated separately (i.e. different classes) because, as mentioned, each strain is technically embodied as an organism in the analyze mode.

All the steps in the original code for the environment input/output and external fitness, and those for unused "phenotype" attributes are basically left unmodified. These are generally relevant in the normal mode of Avida, but are not used or relevant in the current context. For example, mutation rates are irrelevant (i.e., no stochastic mutation is to be considered in the analyze mode) and are left unmodified; whereas gestation time is relevant, so it is handled and gets assigned a valid value in the process. The code proposed in this chapter will not be affected under a condition where environment input/output or external fitness are used.

In the process, offspring with distinct strains get incubated and traced (i.e., to be marked as fertile or infertile). Whether a strain is distinct or not is determined by testing for equality of memory images against all previously found distinct strains. If the strain is confirmed to be unseen, it is determined as a distinct strain and registered as already seen.

To illustrate conceptually, as an individual organism, an offspring is added to a production tree of individual organisms (i.e., without tracing individual organisms of known strains); whereas, as a strain, it is added to a production graph of strains. The production graph structure can be created by following offspring strains of each, which plays an important role in viability quantification discussed earlier.

The pseudocode for this modified algorithm is as follows.

```
while queue is not empty do
  dequeue
  incubate the strain
  if strain does not divide then
    mark as not divided
    mark as having no offspring
  else
    mark as divided
    test the first offspring against each of already seen distinct strains to decide whether
    the first offspring is unseen
    if the first offspring is unseen then
      mark the parent as having this as a first offspring
      instantiate and add in the array of distinct strains/organisms
```

```
            enqueue
          end if
          if the second offspring is unseen then
            mark the parent as having this as a second offspring
            instantiate and add in the array of distinct strains/organisms
            enqueue
          end if
        end if
      end while
```

Last but not least, to reduce the load of information output, the format of a trace file was modified as well so that it does not contain all the snapshots of the incubated organism's virtual CPU state as in the original version, except for memory images of a parent and of a first offspring and a second offspring (previously referred to as initial, final, and child) for each incubated strain (see Appendix C). At the end of the trace file in the new format was added the production graph represented as a list of strains. A strain in this list is represented as a 7-tuple denoting index, size, incubation, division, gestation time, first offspring index, and second offspring index.

4.4.3 Preparation: Estimating a Tractable Analysis Scale

Before applying the enhanced analysis proposed above to the first-step mutants, a tractable scale of analysis needed to be estimated. This was motivated by the fact that tracing full lineages of all encountered distinct strains generated from mutants may not be feasible. The number of distinct strains encountered using the original analyze mode was relatively small (which was 380 per lineage maximum), because trace was applied on a single branch of lineage. Now, however, trace is to be applied on full lineages, each of which may potentially contain more than one such branch, hence more distinct strains. The word *tractable* is used here to mean feasible, or more concretely,"traceable" in terms of the maximum number of distinct strains to be traced, or size of lineage to be traced.

In order to estimate a tractable scale, different values for the maximum number of distinct strains that can be stored per incubation were tried. It turned out that, even when relaxing the maximum number of distinct strains up to 37500 (which is rather arbitrarily determined), 13 mutants were found to be still remaining *indeterminate*. Being indeterminate with this analysis means that the lineages are not closed when represented as a production graph and the evolutionary potential cannot be fully determined. In other words, indeterminate strains generate indefinitely large production graphs, hence too many distinct strains. (In comparison to *indeterminate*, *unclassified* was used in Section 4.3 to refer to strains that have untraced lineages, which implied that those lineages are likely to be closed.) One might instead treat the strains on the edge of the lineage as if they are all infertile and non-viable, but here they are simply distinguished as indeterminate.

From the analysis with the maximum number of distinct strains being set 37500, the number of distinct strains generated per mutant strain analysed is distributed as shown in Figure 4.17. For the remaining indeterminate 13, a greater value of 50000 as the maximum number of distinct strains was tried separately, but they turned out to be still indeterminate. Though these mutants may be of interest evolutionarily at least, and might usefully be the subject of additional investigation, they are considered as outside of the scope of the current study.

Runtime Estimation

Now, runtime of the enhanced analysis is considered and estimated in order to determine a reasonable maximum number of distinct strains per lineage. With the maximum number of distinct strains being 37500, the actual analysis runtime was approximately 5 hours in order for the modified Avida analyze mode to produce the whole trace files for one mutational step on the hardware platform used.[9] The total strains incubated during the trace amounted to 518,811. Using those figures, it can be estimated that the analysis runtime per incubation is 0.034 [sec.].

The estimated number of incubations within 10 hours (= 36000 seconds) will be 1,058,824 ($\approx$ 36000 $\div$ 0.034). For the current set of 8694 mutants, there were 518,811 incubated strains, with the maximum number of distinct strains per lineage being set to be 37500. It is therefore estimated that about a double-sized analysis can be conducted within 10 hours: there is a capacity of approximately 548,583 incubations (= 1067394 [estimated maximum incubations for 10 hours] -518811 [actual incubations for the first step mutant analysis]).

That being so, what size of a set of second-step mutants is most reasonable to aim at? There are 880 strains that turned out to have potential for exponential growth with at least one such strain within lineage. Assuming each of the them is exactly as long as the prototype, each will generate 8694 possible second-step mutants as well. If each of them requires 518,811 incubations just as the first-step case, then the analysis time would be intractable with respect to runtime (i.e., $8694 \times 518811 = 4,510,542,834$ incubations, so the estimated analysis runtime for these incubation is: 0.034 [sec./incubation] $\times$ 4,510,542,834 [incubations] $\approx$ 153,358,456[sec.] $\approx$ 1775[days]). Instead, for the whole selection of second-step mutants to be covered, the maximum number of distinct strains per lineage was decided to be tightened. For example, assume that the capacity for a 10-hour analysis run is 548,583 incubations and that each first-step mutant generates 8694 possible second-step point-mutants. From this, the estimated maximum number of distinct strains per lineage is 63 (= 548583 $\div$ 8694) strains per incubation, that is, in the order of 10^1 to 10^2.

The completion rate of analysis over different maximum number of distinct strains per lineage was estimated. According to the estimation shown in Table 4.3, setting the maximum number of distinct strains as, say, 100 would complete analysis for 99.7 per cent of the mutants, leaving 28 indeterminate strains out of the whole set of 8694 mutants. It follows that solely relaxing the maximum number of distinct strains per lineage would not proportionally lead to the increase of determinate strains. On the basis of these heuristics, the maximum number of distinct strains was set somewhat arbitrarily to 100, which should be sufficient for screening interesting strains among the majority of candidates by heuristically classifying the longest lineages as indeterminate.

[9] The spec of the computer used (MacBook Pro) is: 2.4 GHz Intel Core2Duo Processor and 4GB RAM.

Max. Distinct Strains	**Determinate Mutants (%)**
0	0.0000
1	0.6999
2	0.9055
3	0.9729
4	0.9802
5	0.9853
6	0.9855
7	0.9888
8	0.9890
9	0.9891
10	0.9893
12	0.9894
19	0.9895
22	0.9911
27	0.9913
29	0.9915
33	0.9916
42	0.9917
49	0.9923
61	0.9924
65	0.9925
76	0.9928
77	0.9942
80	0.9945
81	0.9946
82	0.9967
83	0.9968
113	0.9971
126	0.9972
215	0.9974
427	0.9975
431	0.9976
759	0.9978
1814	0.9979
1919	0.9984
2797	0.9985
37500	0.9985

Table 4.3: Estimated analysis completion over maximum number of distinct strains per lineage. The completion rate is represented as the percentage of determinate mutants of all being analysed.

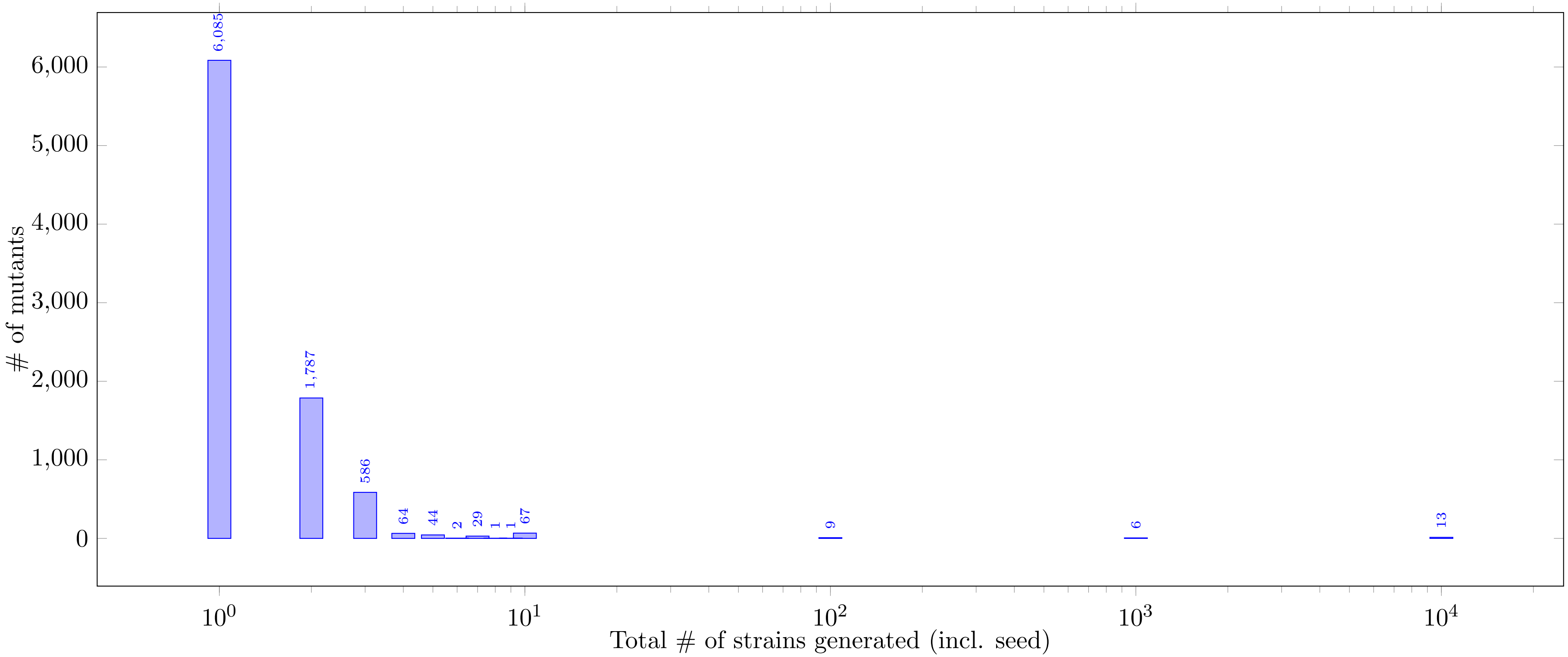


Figure 4.17: Distribution of the 8694 first-step point-mutants of the prototype by total number of distinct strains generated deterministically under the trace analysis (recursive incubation). The total number of strains generated means a seed strain plus descendant strains; hence the minimum number is 1, meaning the seed strain incubated is either direct infertile or direct self-reproducer, generating no other distinct strain. For legibility, those between 10^1 and 10^2 are combined to fall in 10^1, likewise, those from 10^2 and 10^3 in 10^2, those between 10^3 and 10^4 in 10^3.

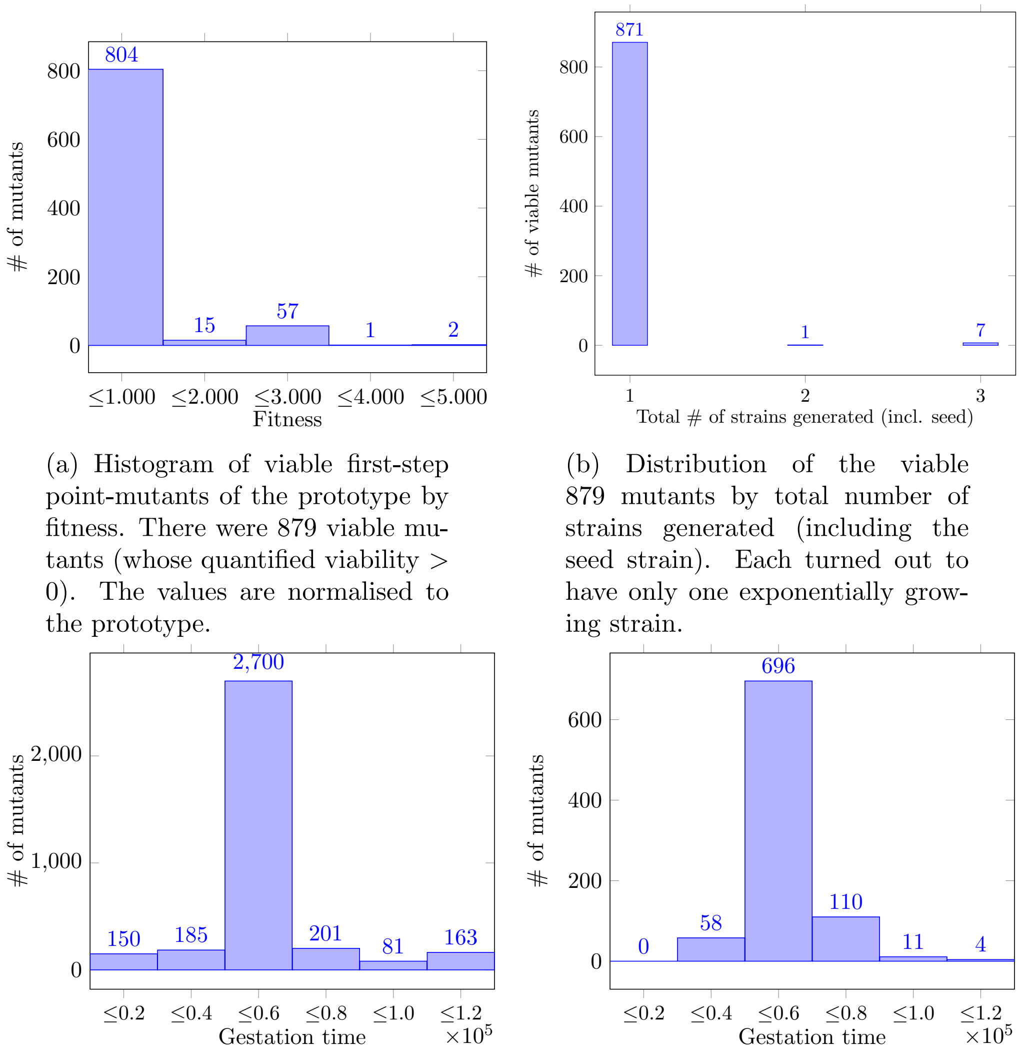
4.4.4 Enhanced Viability Analysis: First-step Mutants

With the modification explained above, the analysis tool was now enhanced with the ability to fully cover the lineage expected from a given seed strain. As a pilot study, the first-step 8694 mutants of the prototype were analysed using the enhanced tool. A feasible analysis scale is considered, because the number of strains that need to be handled will be obviously increased, compared to the original analyze mode, following which a fitness distribution is shown based on quantified viability. Then, mutants of more interest are considered and discussed. A distinctive seed strain found in the analysis is introduced as a case study. This will serve as a basis for further mutation steps, where more steps will be covered, but with a limited number of strains for trace, as the number of strains to be traced at each mutational step needs to be kept within the feasible scale.

Fitness Distribution

Figure 4.18a shows the histogram of the first-step mutants by quantified viability, or the maximum real part of valid eigenvalues as explained in the previous section. Or, more simply, this amounts to (hypothetical, deterministic) *fitness*. There is one cluster at fitness < 1.000 (normalised to the prototype) which has the great majority (804 out of 879) of the mutants. The viable 879 mutants distribute as shown in Figure 4.18b by the number of generated strains including seed strains. There are 871 viable mutants with a total of only one strain within each lineage. This indicates that their seed strains alone are viable. It can be inferred that they are themselves self-reproducer strains (of whatever reproduction mode), because for a single strain to be viable, the strain is necessarily self-reproducer. Those 871 viable mutants, as seed strains, do not give rise to other distinct strains, not to mention other viable distinct strains. On the other hand, the remaining 8 ($= 1 + 7$) viable mutants give rise to at least one viable (self-reproducer) strain other than the seed strain. The viable strains (whether they are generated strains or seed strains) are ones requiring further consideration. Out of the 8694 first-step mutants, there were 3480 fertile mutants (that is, the rest, 5214 mutants, are infertile or indeterminate). A histogram by gestation time for the 3480 fertile mutants is shown in Figure 4.18c. A histogram by gestation time for the viable 879 mutants is shown in Figure 4.18d. In either histogram, it is apparent that there is a cluster in the middle “$40000 <$ gestation time ≤ 60000”, in which the prototype’s gestation time 52218 would fall.

From Figure 4.18a, it is noticeable that there is one outstanding leftmost cluster, while there are other clusters to the right with higher fitness values spread out. From Figure 4.18b, it is apparent that most of the 879 non-trivial mutants generate only one viable individual strain that is actually the mutant incubated as the seed strain itself. In Figure 4.18c, there is one outstanding cluster “$40000 <$ gestation time ≤ 60000”. There are both clusters of shorter gestation time and of longer gestation time. A natural hypothesis is that those in clusters of shorter gestation time (to the left, ≤ 40000) include self-copier mutants, and that the middle outstanding cluster includes von Neumann style self-reproducer mutants. Whatever the reasons behind the outstanding cluster with gestation time around the prototype gestation time and the spread of gestation time are, further investigation would be required to determine whether there is a reasonable correlation



(a) Histogram of viable first-step point-mutants of the prototype by fitness. There were 879 viable mutants (whose quantified viability > 0). The values are normalised to the prototype.

(b) Distribution of the viable 879 mutants by total number of strains generated (including the seed strain). Each turned out to have only one exponentially growing strain.

(c) Histogram of the first-step point-mutants by gestation time. There were 3480 fertile mutants out of the total of 8694.

(d) Histogram of the viable 879 mutants by gestation time.

Figure 4.18: Fitness distribution of the first-step mutants.

Segment	**Length**	**Mutant Group Size**
Decode Preparation	28	756
Decode Loop	166	4482
Copy Preparation	52	1404
Copy Loop	48	1296
Lookup Table	28	756
Total	322	8694

Table 4.4: Lengths of the five functional segments of the prototype's phenome and the corresponding mutant group sizes.

between gestation time and reproduction mode, and if there is, what the correlation is. Nevertheless, it is possible that interesting strains retaining the von Neumann architecture may be found in the outstanding cluster (or neighbouring clusters). It is noteworthy that the first encountered self-copier reported in Chapter 3 should fall on the cluster "20000 < gestation time ≤ 40000 and " in Figure 4.18d , as it is a self-copier mutant with gestation time 22172.

Mutants with a Mutation in the Lookup Table

Here, as part of the mutant characterisation, it is useful to group the first-step mutants by mutation region. Mutation region means the segment of the phenome where a point-mutation that was applied is expressed. The size of each group of the mutants can be calculated from each segment's length. Length of each segment and size of each mutant group are shown in Table 4.4. Allow each mutant group to be called in the form of "mutation region" + mutants: namely, Decode Preparation mutants, Decode Loop mutants, Copy Preparation mutants, Copy Loop mutants, and Lookup Table mutants.

For general understanding of the tendency among mutants, Table 4.5 shows how the mutants classified by mutation region distribute in terms of viability. Viable and Non-Viable are determined by whether a mutant has fitness > 0.

Next, Lookup Table mutants are considered. They are of particular interest because a mutated lookup table may lead to a modified genotype-phenotype mapping. As far as Lookup Table mutants are concerned, one can think of a phenomenon called *word loss*: as the lookup table defines how individual genome words/symbols (or "codons") are translated, a mutation expressed in that segment may lose an entry that associates a codon to another word, causing loss of one codon from the table. This is true because the lookup table has entries each corresponding to each of possible words defined by the instruction set, and because a word brought about by a mutation is one of those possible words. Once this happens, the word associated to a codon lost from the lookup table will not appear as a decoded word in translation. This is why it is referred to as word loss. As

Mutation Region	Viable	Indeterminate	Non-Viable	Total
Decode Preparation	45 (6%)	1 (0%)	710 (94%)	756
Decode Loop	222 (5%)	5 (0%)	4255 (95%)	4482
Copy Preparation	322 (23%)	2 (0%)	1080 (77%)	1404
Copy Loop	101 (8%)	20 (2%)	1175 (91%)	1296
Lookup Table	189 (25%)	0 (0%)	567 (75%)	756
Total	879 (10%)	28 (0%)	7787 (90%)	8694

Table 4.5: Viability of the first-step point-mutants by mutation region.

a consequence of word loss, there will be another codon with one more entry in the table. As a result, the lookup table will have *redundancy.* The point here is that word loss may or may not render the prototype, which is viable, non-viable. See Table 4.6 for the word frequency in the phenome and viability caused by word loss.

As explained in Chapter 3, there are employed words (instructions used to code the phenotypic segment except for the lookup table) and unemployed words (the others). In hindsight, it is not surprising that unemployed words can be lost without losing the strain's viability, or self-reproducibility. However, it is an interesting fact that that means employed words are so essential that in the case of being lost from the lookup table, the strain would be rendered infertile, rather than fertile somehow. One evolutionary significance of this observation is that it can be seen as the increased redundancy due to the repetition of the word through loss of unemployed words, which amounts to a decreased repertoire of translation. Although this does represent genuine evolution of the genotype-phenotype mapping, which is the central topic of the thesis, this particular kind of change rather represents an evolutionary *impoverishment* of the mapping, or an apparent *regression* or reduction in subsequent evolutionary potential. Though changes in the genotype-phenotype mapping which support wider variety of expression would be of more interest, that is impossible on this particular measure in the current system considering the fact that the prototype, which the starting prototype ancestor, already has the maximum, richest repertoire by design.

Second to Lookup Table mutants, the group of Decode Loop mutants would be of interest in the context of the evolutionary characterisation of the von Neumann style prototype. They are, however, not going to be further classified as part of the analysis, not only because classification by mutation sub-region of the decode loop is a more high-level task, but because the analysis for mutated decoding style would eventually be more

Word	Opcode	Freq. (Active/Phenome)	Viable when Lost?
0	`nop-A`	29 / 30	
1	`nop-B`	7 / 8	
2	`nop-C`	40 / 41	
3	`if-n-equ`	2 / 3	
4	`if-less`	0 / 1	yes
5	`if-label`	0 / 1	yes
6	`mov-head`	6 / 7	
7	`jmp-head`	0 / 1	yes
8	`get-head`	2 / 3	
9	`set-flow`	2 / 3	
10	`shift-r`	0 / 1	yes
11	`shift-l`	0 / 1	yes
12	`inc`	2 / 3	
13	`dec`	2 / 3	
14	`push`	54 / 55	
15	`pop`	49 / 50	
16	`swap-stk`	80 / 81	
17	`swap`	4 / 5	
18	`add`	3 / 4	
19	`sub`	2 / 3	
20	`nand`	0 / 1	yes
21	`h-copy`	1 / 2	
22	`h-alloc`	1 / 2	
23	`h-divide`	1 / 2	
24	`IO`	0 / 1	yes
25	`h-search`	4 / 5	
26	`read`	2 / 3	
27	`write`	1 / 2	
		294 / 322	

Table 4.6: Word frequency in the prototype phenome and viability affected by word loss. "Active" in the third column of "Freq." (Frequency) means the executed segment of the phenome (phenotypic segment except for Lookup Table), as opposed to the phenome as a whole including Lookup Table.

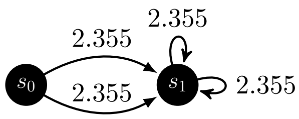


Figure 4.19: Production graph of a first-step mutant (id 2403). The rates shown are relative to the prototype.

complicated than for mutated word-by-word mapping. It is essentially the same difficulty as that of reproduction mode classification.

A Case Study: Distinctive First-Division Pattern

A first-step mutant (id 2403) was hand-picked as it was noticeable for having a distinctive first-division pattern (see Figure 4.19 for its production graph structure) which no other first-step mutant showed. The mutation is inherited at 411 and expressed at 89. (It is noticeable that the mutation location is the same as that of the self-copier reported in Section 3.4 in the previous chapter.)

A closer look clarified the strain's reproduction mechanism: it produces two non-identical offspring with a two-word difference in the memory image (i.e., a word in the lookup table and another one in the genome), and the offspring are of an identical self-reproducer strain. On the surface (in terms of the execution profile presented in Figure 4.20) these strains (s_0 and s_1) appear to be the same, but are two distinct strains in reality, exhibiting two distinct reproductions. It is evident that the seed strain (s_0) is not a self-reproducer. The descendant strain (s_1, generated from s_0) is a self-reproducer, and judging from gestation time, it appears to be a self-copier, if not necessarily purely.

The seed strain s_0 is found to be interesting in that it demonstrates "rewriting" of itself (as a parent) and results in an offspring strain capable of self-reproducing. (Here, the word "rewriting" is tentatively used in order to mean that a parent modifies some part of itself by writing something on it; whereas the word "overwrite" would be used if the parent wrote something over what it had previously written as a part of its putative offspring.) The organism of this strain actually rewrites elements of its own lookup table and its own genome (one word for each), as if changing the genotype-phenotype mapping. (Here, it is stated as "as if", because the strain's reproduction mode does not appear to be von Neumann style self-reproduction.) More specifically, the mutation changed 13 into 0 at the address 411 of this organism. This is expressed as 27 (opcode for the write instruction) where it was originally 14 (opcode for the push instruction) at the address 89. That means now there are two write instructions in the decode loop.

The behaviour of the organism of this strain (s_0) is illustrated in more detail. There are in total four executions of the write instruction. That is, two write instructions, one as an expressed mutation and another as an original content, are executed for two iterations. See Table 4.7 for how two write instructions are executed for the first iteration.

In effect, the mutation ends up causing the rewriting of the lookup table. The former write with a modifier nop-A goes on to write the value 294 (the content held in AX) into the address 294 (which is $294 + 0$, the sum of the values held in AX and BX). (Without a modifier, by default, the instruction would write the content held in CX, rather than that held in AX, into the address at the value held in AX plus that held in BX.) The value to

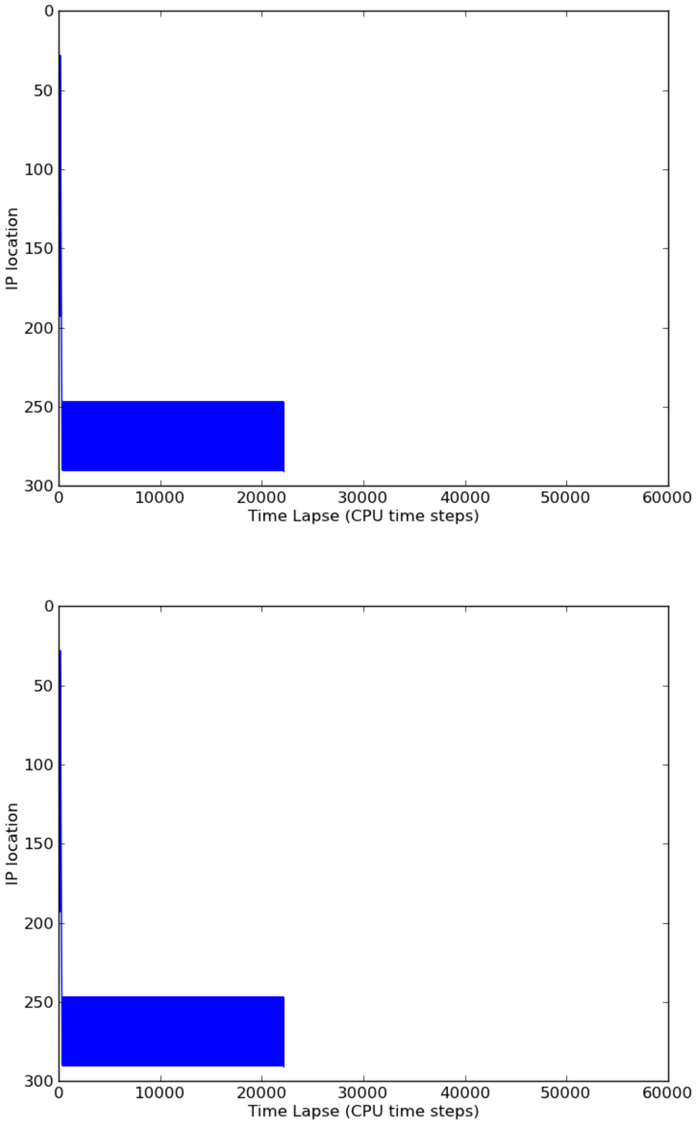


Figure 4.20: Execution profiles of strains within the lineage of an example first-step mutant strain (id `2403`) (Top: s_0; Bottom: s_1). This is a Decode Loop mutant. The top profile corresponds to the seed strain s_0 producing two offspring of the strain s_1; the bottom profile corresponds to the strain s_1 self-reproducing. Each strain has gestation time of 22172. This gestation time is noticeably identical to that of the self-copier reported in Section 3.4 in the previous chapter. The strain s_1's reproduction mode is similar to that of the self-copying one. In terms of the `write` and `h-copy` execution count, each has 4 instances of `write` execution and 643 instances of `h-copy` execution.

Time	IP	Content	Registers
69	89	write	[294,0,22]
	90	nop-A	
70	91	swap-stk	[294,0,22]
...	...	...	
77	102	write	[644, 0, 22]
78	103	inc	[644, 0, 22]

Table 4.7: The execution of the first and second `write` instructions in the first-step mutant (id `2403`). The former `write` instruction (highlighted) is the expressed mutation, which replaced the original instruction `push` at the location. The latter `write` instruction is the original one.

be written, `294`, is correctly stored or processed in the CPU, but on transfer to memory, it is handled as `14`, because any integer is made to fall within the range of the size of the instruction set by design of Avida and because 294 mod 28 = 14. Therefore, the original memory content `27` at the address `294` is replaced with `14`. The address `294` is the first word of the lookup table. The change of the lookup table signifies that now the word `0` will translate as the word `14`, instead of `27`.

The latter `write` instruction with no modifier is the original, and writes the value `22` (opcode for the `h-alloc` instruction) as intended to, the first word of the prospective offspring memory image, at the address `644`.

After the above iteration, the IP loops back (in the way the decode loop normally does) and the `write` instruction is executed for the third and fourth times likewise. The third execution of the `write` instruction, with a modifier, goes on to write the value 322 (the content now held in AX) into the address `323` (= 322+1, the sum of the values held in AX and BX). That value to be written is likewise reduced modulo the instruction set size (28). This is `14` again as 322 mod 28 = 14. Therefore the current value `2` at the address `323` is replaced with `14`. This is a change in the genome (or, more precisely, in the segment which used to be the genome, but that the intended von Neumann style architecture now seems to have been lost). It is the second word of the previous genome, which originally coded the word `2` so as to be translated into `25` (opcode for the `h-search` instruction), the second word of the prospective offspring phenome; but now it has been changed to the value `14`, which would be translated into `13` (opcode for the `dec` instruction) according to the lookup table. The fertility of the offspring appears to hinge on the effect of this translated word, because it is not the originally intended opcode `25` (opcode for the `h-search` instruction) that will now be executed as the second word of the offspring's program, but instead, the opcode `13` (opcode for the `dec` instruction). The `h-search` instruction is crucial for the reproduction cycle since it will locate the beginning of the lookup table, using the label following it, as a part of preparation after memory allocation.

The fourth execution of the `write` instruction, the originally coded one, ends up writing the value `10` (opcode for the `shift-r` instruction) next to the value `22` (opcode for the `h-alloc` instruction) that had been first written in the previous iteration.

After this total of four executions of the `write` instruction, the execution inadvertently proceeds and enters the subsequent segment of the memory image. This turns out to be the "copy" phase, where the originally coded `h-copy` instruction is executed. This instruction overwrites the most recently written value `10` (opcode for the `shift-r` instruction), altering it to `25` (opcode for the `h-search` instruction), which is in fact the originally intended word. As a result, once this succeeds, the parent strain ends up being able to continue its reproduction cycle via copying.

This is reminiscent of the self-copier mutant reported in Section 3.4 in the previous chapter. Like that case, once executed, the `h-copy` instruction is so strong as to proceed with the copying process without interruption, leaving no room for translation/interpretation, and certainly finishing at the right point, as long as the value (of the counter) which tracks *how many words to go until the copy process finishes* remains unharmed by the point where the copy loop (with the `h-copy` instruction) is entered. Of course, employing the `h-copy` instruction is not enough. An instance of the `h-copy` instruction in isolation cannot achieve self-reproduction by self-copying; rather, other instructions are also required so that starting and finishing conditions are somehow satisfied for the `h-copy` operation.

In summary, the organism of this mutant strain s_0 produces two offspring with a distinct strain s_1 with changes `27` into `14` at the address `294`, and `2` into `14` at the address `323`, within the lookup table and the genome, respectively. (Note that these two words of the parent have been "rewritten".) This can be viewed as the "initial" memory image changes itself, and the "final" memory image is identical to the "child" memory image, and these ("final" and "child") turn out to be self-reproducer. (Recall this distinction by the original Avida analyze mode pointed out in Section 4.3.)

This mutant strain s_0 may not be typical, but is non-trivial in its own right for exhibiting such a reproduction mode. This mutant is viable insofar as giving rise to a self-reproducer strain s_1. The strain s_1 is a self-reproducer strain, hence is viable. However, taking into account its execution profile and short gestation time, this strain is most likely to be a self-copier strain. It is, again, a subtle judgement because the similarity in memory image or in attributes such as gestation time does not necessarily imply the similarity in the reproduction mode.

4.5 Automation of the Analysis

The result from the enhanced analysis for the first-step mutants provided heuristics for further mutational analysis by clarifying that the vast majority of the mutants of the prototype are infertile (hence non-viable) and can be excluded from deeper, recursive, mutational investigation. Another fact is that the indeterminate mutants (i.e., generating too many distinct strains) were a small minority. Ignoring the infertile and the indeterminate strains, it is now necessary to extract strains of interest with viability, according to some heuristic criteria.

The enhanced analysis is a mutant viability analysis as part of evolutionary characterisation. Viability is defined qualitatively in Section 4.4 as having evolutionarily interesting potential, and quantitatively as having values that represent the potential of exponential growth (and viability can be simply referred to as fitness, in a framework limited to this analysis). With quantified viability, it is naturally possible to regard viability as something varying continuously (i.e., the higher the value is, the more viable the strain is considered), or as something binary by setting a threshold (e.g., strains whose viability is above x are *viable*).

Quantified viability is tightly coupled to gestation time, as production rates are calculated from gestation times. So gestation time is one simple but convenient attribute as discussed in Chapter 3: heuristically, if it is too short or too long, the strain should not be retained for further investigation in a stronger, qualitative sense. If gestation time is too short, the strain's reproduction mode is more likely to be self-copying, and if it is too long, it is unlikely to be competitive in the Avida world where production rate is the only fitness. To reflect this idea, it is reasonable to introduce a ranking-based selection method using that value to determine strains to analyse for a further mutation step. These aspects of viability are considered in automating the enhanced analysis.

4.5.1 Automation Cycle: Single-step Analysis

The enhanced analysis was automated to realise multi-step mutational analysis. An automated analysis should take as input, a single ancestral strain, and produce as output, a set of trace files with lineage information collected from the mutants generated from the *source* strain, ideally, iteratively for an arbitrary number of generations. The proposed algorithm has three stages as follows.

1. Pre-analysis: Generate a set of possible point-mutants from the source strain.
2. Avida trace: Run the (modified) Avida analyze mode and generate trace files.
3. Post-analysis: Extract a set of mutant strains of interest, or worth further, recursive analysis.

Pre-analysis

Possible point-mutants are enumerated systematically. Suppose a source (input) strain has an equal decomposition into phenome and genome as the prototype does (i.e., the first half and the latter half). It is assumed that a lookup table that is as long as the instruction set is located at the end of the phenome, just like the prototype. The pre-analysis first generates permutations of each memory location of the genome according to the applicable instruction set. This serves as mutating the source strain genome in all possible ways. Then for each, the mutation is expressed in the phenome, generating a mutant strain. A list of these mutant strains is produced at the end of this process.

This process of generating possible mutants is at best heuristic, especially beyond the first mutation of a known ancestor. This method may be easily applied to arbitrary Avidian strains, but the assumption that a strain is a von Neumann style self-reproducer is not very reasonable (much less that it is the same style as the prototype). Even if the strain somehow turns out to be a von Neumann style self-reproducer, the precise location of the genome, or of the existence and location of the lookup table in the phenome would be uncertain in general.

Avida Trace

The modified analyze mode of Avida is called and run. As a result, a set of trace files of the mutants as seed strains are output, each containing all the generated and incubated strains in the lineage. Naturally, the number of trace files depends on the number of mutants to be analysed (as generated in the pre-analysis), or hence the genome size of the source strain.

Post-analysis

Non-trivial, or potentially viable, mutants are selected for further mutational analysis. Firstly, in order to quantify viability, the post-analysis calculates eigenvalues and eigenvectors from the matrix representing the approximate population dynamics for each lineage containing strains generated from a seed strain. Secondly, the post-analysis produces a summary of the lineage information for each mutant strain based on the trace files. At the end of this process, from the seed strains generating at least one individually exponentially growing strain, those with highest fitness (i.e., a fittest strain in lineage with a highest maximum real part of eigenvalues of the matrix gained from the production graph) and with gestation time longer than the half of the prototype's gestation time are selected. (These selection criteria or conditions may vary in the future based on experimenter preferences.)

Notice that here the attention for further analysis is restricted to direct simple self-reproducer strains which have highest quantified viability, or highest fitness, among the finite set of distinct strains generated from seed strains. Fittest strains would survive in the neo-Darwinian evolution, and thus are of interest at least. Ideally, again, the analysis should be able to determine whether a strain is a von Neumann style self-reproducer strain and in what way it is realised, but this is difficult and is not what the current analysis aims at doing, as discussed in Section 4.2. Collectively exponentially growing strains would arguably be of evolutionary interest, but these are discounted, because the current analysis focuses on pure von Neumann style self-reproducers (like the prototype). There is no tool available for analysing a genotype-phenotype mapping or finding changes within a genotype-phenotype mapping in general, unless analysing on a case-by-case basis. It would be much more difficult in cases of collective self-reproducer strains, as there is not a sufficient definition of von Neumann style collective self-reproduction yet.

To recapitulate the selection method, heuristically eliminated as being non-interesting mutants are:

- indeterminate mutants;
- mutants whose production rate matrix's eigenvalues and eigenvectors do not indicate individual exponential growth;
- mutants with extremely short gestation time compared to the prototype; and
- mutants whose production rate matrix's eigenvalues and eigenvectors do indicate exponential growth, but collective exponential growth.

This method is illustrated next with a specific experiment.

4.5.2 Multi-step Analysis

Section 4.4 reveals that the first-step analysis ends up with 879 mutants which are viable (i.e., having the potential of generating at least one exponentially growing strain), and 871 of them turn out to have no more than one such strain—that is the seed strain itself. Assuming that only point-mutation exists and that mutants retain the von Neumann style architecture as the prototype, the automated multi-step enhanced mutational analysis was run for the second-step mutants and the third-step mutants. Depending on the feasible scale of analysis, the number of strains that are selected for further mutational step analysis was varied. Suppose the scale of analysis is fixed, then the more mutational steps the analysis takes, the larger the proportion of potentially interesting strains that needs to be pruned (i.e., the smaller proportion of potentially interesting strains that needs to be selected) for the subsequent step.

For two-step analysis to be completed within a plausible wall-clock time of up to 10 hours in total (the choice of which was not strict, but can be considered as a reasonable amount of experimental runtime, greater than the order of 1 hour, and less than the order of 100 hours), a small group of 10 strains were selected for a second-step based on the ranking of fitness. (Likewise, this choice was heuristic, but it is a reasonable amount greater than the order of 1 strain and less than the order of 100 strains. Considering that viable mutants were 10% of 8694, the order of 100, an interesting portion should be less than this.)

For a two-step analysis of the scale discussed above, the number of strains to be traced are 11 ($= 1 + 10$). For a third-step analysis to be of the same scale, the selection size of 3 was chosen, considering the computational tractability, where the number of analysed strains are 13 ($= 1 + 3 + (3 \times 3)$; as the selection size is fixed, this is the closest to 11). Three strains are selected based on the fitness ranking at the end of each step.

In the selection, the condition was set that gestation time must be longer than the half of the prototype's (i.e., $26109 = 52218/2$) to reduce the likelihood of self-copiers from being selected, and shorter than twice of it ($= 52218 \times 2$), for a strain to be selected from among viable strains in lineages for further analysis.

Second-step Mutants

The tree in Figure 4.21 shows the selected top 10 mutants of the prototype, and those of the top mutant of the first step. Only a part of the whole tree is presented for simplicity: the actual analysis covered all 10 selected mutants for each of the first-step mutants.

A naming convention is adopted: a strain is uniquely specified by the id number in such a format as "`org-0-3431`" and "`org-0-3431-7210`", where the numerical mutant id is consecutively concatenated with a hyphen as the generation increments. The prefix "`org-`" is used for the sake of convenience: it comes from the original Avida system where any strain must be instantiated as an organism internally, and an id number is issued to each organism.

The ranking of the top 10 mutant strains of the first-step is shown in Table 4.8. They were analysed for the second-step. Six of them are found to be viable and seed strains (s_0), whereas the other four are viable and descendant strains (all happened to be s_1).

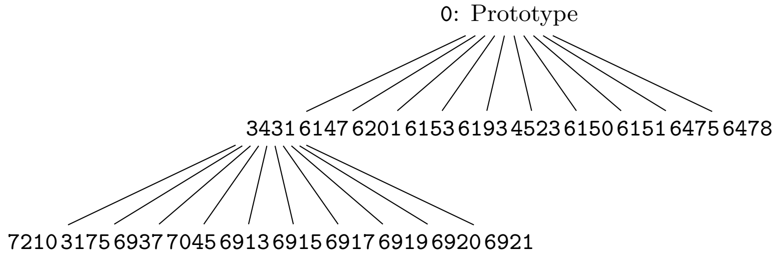


Figure 4.21: Excerpt of the tree of mutant strains for trace up to the second mutation step with a selected top 10 of the ranking of fitness. In the process of the automated analysis, an identification number is given to a mutant in each generation and is used to identify a strain within a tree of mutant strains for trace. The leftmost bottom strain, for example, is uniquely referred to as "org-0-3431-7210". The mutation generation of a strain is visible with such a notation of identification.

Mut. ID	**Strains**	**Viable**	**Fittest**	**Fitness**	**Gest. Time**
3431	1	1	s_0	1.373	38095
6147	3	1	s_1	1.300	40186
6201	3	1	s_1	1.295	40314
6153	3	1	s_1	1.076	48634
6193	3	1	s_1	1.076	48634
4523	1	1	s_0	1.003	52090
6150	1	1	s_0	1.003	52217
6151	1	1	s_0	1.003	52217
6475	1	1	s_0	1.003	52217
6478	1	1	s_0	1.003	52217

Table 4.8: The selected top 10 first-step mutants of the ranking of the prototype, org-0. The column "Mut. ID" is the id number given to the mutant strain used as a seed strain. The column "Strains" shows the total number of strains that have appeared in the lineage including the seed strain. "Viable" shows the number of viable strains in the lineage (seed strain and generated strains inclusive). "Fittest" is the strain which is the fittest in terms of fitness in the lineage. "Fitness" shows the fittest strain's quantified viability and "Gest. Time" its gestation time.

Table 4.9 shows the second-step mutant of the first-step mutant org-0-3431, which is the top of the ranking, as an example.

No difference is found in fitness values of some of the ranked mutants. This implies that the ranking ended up relying upon the mutant id numbers, and that the selection may have amounted to extracting a certain number of mutants from a group of equivalent strains. Nevertheless, those selected mutants can be regarded as some representatives of the mutant set with high fitness. There may be, however, a bias related to the mutation region: the proximity of mutant ids roughly signifies the proximity of mutation regions, and chances are that mutants outside of this ranking might turn out to be of interest. It is noticeable that the fitness value is recurring in the ranking. So there may be ones with reasonably high fitness value but not selected, that have fallen out of the ranking (those with the same value of fitness, with ids from org-6913 to org-6921 in the case of the org-0-6131's mutants). Moreover, having the same fitness value does not mean having

Mut. ID	Strains	Viable	Fittest	Fitness	Gest. Time
7210	100	1	s_0	1.833	37245
3175	1	1	s_0	1.781	29345
6937	3	1	s_1	1.598	32638
7045	3	1	s_1	1.587	32887
6913	3	1	s_1	1.572	33223
6915	3	1	s_1	1.572	33223
6917	3	1	s_1	1.572	33223
6919	3	1	s_1	1.572	33223
6920	3	1	s_1	1.572	33223
6921	3	1	s_1	1.572	33223

Table 4.9: The selected top 10 second-step mutants of the first-step mutant `org-0-6131`. The lineage of the top mutant (Mut. ID `7210`) has exactly 100 strains within. One can infer that this as a boundary case, from the fact that the recursion limit is set to be 100 and the mutant instance which reaches the limit is excluded from the ranking; this will apply to other such mutant instances, too.

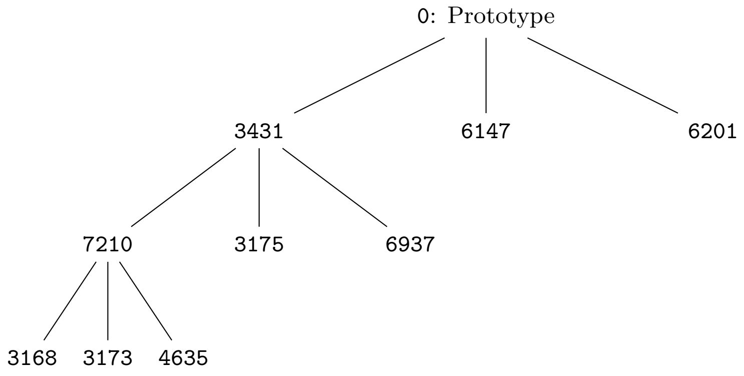


Figure 4.22: Excerpt of the tree of the mutants up to the third-step with a selected top 3 mutants from the ranking.

the same dynamics, the same structure of lineage, and so on. So this ranking method may be easy to implement, but may not always be the most reasonable method.

To show a profile, the first-step mutant `org-0-3431` was found to have second-step mutants including:

- 43 indeterminate mutants that reach the preset recursion limit of 100;
- 2024 mutants that generate a total of 1 viable strain.

Third-step Mutants

See Figure 4.22 for the tree of selected mutants over three steps of mutation. The top 3 naturally overlaps the top 10 presented above. This analysis looks one-step deeper into the mutation tree, but with a narrower scope.

Take the second-step mutant `org-0-3431-7210` for example, whose top 3 of the ranking is shown in Table 4.10. As no difference is found in the fitness values of the first and second mutant instances, the ranking again relied upon the mutant id number; and as for the

Mut. ID	Strains	Viable	Fittest	Fitness	Gest. Time
3168	100	1	s_0	2.799	35895
3173	100	1	s_0	2.799	35895
4635	53	1	s_0	2.788	36245

Table 4.10: The selected top 3 third-step mutants of the second-step mutant org-0-3431-7210.

third mutant instance, the selection may have extracted one from a group of equivalent strains. At least, just as in Table 4.9, the three selected mutants can be regarded as some representatives with high fitness of the mutant set. Again, there may be a bias related to the mutation region, as the proximity of mutant ids roughly signifies the proximity of mutation regions. That is, there may be ones with the same or reasonably near fitness, that have fallen outside the top 3 of the ranking (from mutants after id 4635 in the case of org-3431-7210's mutants).

This mutant org-0-3431-7210 was found to have third-step mutants including:

- 289 indeterminate mutants that reach the preset recursion limit of 100;
- 2 mutant that generate a total of 5 viable strains;
- 4 mutant that generate a total of 4 viable strains;
- 23 mutant that generate a total of 3 viable strains;
- 121 mutant that generate a total of 2 viable strains;
- 1697 mutant that generate a total of 1 viable strains.

4.6 Reflections on the Methodology of Mutational Analysis

Starting from revisiting the concept of self-reproduction through strain classification, and redefining viability as potential for exponential growth, a method of effectively analysing mutational pathways of the prototype, not stochastically (evolutionarily) but deterministically (systematically), is explored and investigated in the preceding sections of this chapter.

4.6.1 Viability Analysis Results Overview

The analysis using the preexisting Avida analyze mode yielded an insight into distinguishable viable candidates (see Section 4.3). The viable candidates (10%) are ones that are direct self-reproducer and need further scrutiny for reproduction mode, while the non-viable candidates (60%) are ones that need not. At any rate, it was revealed that while the vast majority (70%) of the candidate set were classified as either viable or non-viable, the rest (30%) of the candidates remained unclassified. This analytical process suggested that there is subtlety in the concept of viability. For more concrete mutational analysis, the concept of viability was revisited and redefined to mean more than an organism's immediate dividability (or a strain's direct fertility). In other words, viability is now defined as the potential for exponential growth, that is, the ability for any of the possible mutants (linked through mutational pathways) to generate at least one viable strain within a lineage. This can be interpreted as fitness within a hypothetical, deterministic framework of the proposed analysis.

An enhanced analysis tool was developed based on the built-in analysis tool of the original Avida (see Section 4.4). It was at least intended to further classify the candidates remaining unclassified on the basis of the finer-grained strain classification and the redefined concept of viability. The enhanced analysis shed more light on the mutational pathways of the prototype, covering previously untraced lineages. Now, only large lineages are classified as indeterminate. Regarding the power of this analysis tool, it is clear that the enhanced analysis can noticeably reduce the size of previously unclassified mutants. The first-step point-mutants of the prototype were reclassified now with 28 indeterminate mutants (0.3%) and 8666 determinate mutants (99.7%), out of which 879 (10.0%) are viable with potential of exponential population growth. With either the old analysis tool or this enhanced tool, the viable ones remains almost the same percentage of 10% of the mutant set.

With regards to the analysis itself of the first-step mutants, the mutation of genotype-phenotype mapping in a weaker sense was observed in the form of losing translation repertoire. This phenomenon observed in evolutionary process was interesting in its own right, but limited, as it was not elaboration but rather impoverishment of genotype-phenotype mapping.

For more mutation steps and for more efficiency, the enhanced analysis was automated (see Section 4.5). A selection mechanism was incorporated to screen only non-trivial candidates in applying the enhanced analysis on more mutants beyond the first step, in order for the analysis scale not to become too large. The second- and third-step mutation analyses were demonstrated, clarifying that it is still rare for the prototype to give rise to a non-trivial self-reproducer. At least, within the range up to three mutation steps, the analysed mutational pathways exhibit diverse strains with varied potential lineages, but those which lead to evolutionarily non-trivial strains were somehow either seed strains or the first descendant strains within lineages.

Turning back to the question of degeneration, there are no more than 58 self-copiers of the same type as first encountered which have fairly short gestation time, and these are estimated to account for less than 0.7% of the whole, and for less than 6.6% of viable strains. If this estimation is correct, it can be regarded as rare for the same type of degeneration to take place. This group may, however, include diverse self-reproducers, not limited to the standard self-copiers or the original von Neumann style self-reproducers. The mutants whose gestation time is around or more than the prototype may be von Neumann style self-reproducers or something unseen, but may arguably include self-copiers as well. Yet, the style of self-reproduction of such self-copiers should be categorised as different from the pure self-copying, as having noticeably longer gestation time (e.g., with idling time).

Overall, although the presented results are rather foundational and preliminary, it should serve as a minimum platform by proposing an extendable analysis method for evolutionary characterisation of self-reproducers in Avida, including the prototype von

Neumann style self-reproducer. For a more general analysis tool, the ability to distinguish reproduction modes should be investigated and incorporated. As to the design of the ancestor (and the instruction set), more consideration is recommended, especially with respect to the design of the lookup table or even the mechanism of translation itself. This consideration will help this line of research to reveal the evolvability of self-reproducer strains and of the genotype-phenotype mapping in Avida.

4.6.2 Further Modification and Enhancement of Analysis

Through the investigation in this chapter, more light is shed on a general methodology of mutational analysis in Avida. That being said, questions regarding further possible modification and enhancement are raised in the course of this investigation as briefly discussed here.

Even with reduced trace information output (presented in Section 4.4) and with limited recursions, limited mutation steps, and limited range of candidates (demonstrated in Section 4.5), the automated enhanced mutational analysis is still computationally expensive. Nevertheless, one can theoretically extend the proposed automation on an arbitrary organism's mutational pathways for any given number of mutation steps in Avida, depending on available computational resource. Parallelisation may be an option for fast high-performance computing, both for the Avida analysis and the pre- and post-analysis.

From the efficiency point of view, if the number of distinct strains to be incubated increases considerably, the process of strain equality checking found in each iteration could be improved using a hash table technique. However, the proportion of the overall computational cost taken by this check may be small. It is not yet clear either whether such mutants with a large lineage are necessarily interesting or uninteresting evolutionarily.

For better classification of reproduction mode (raised earlier in Subsection 3.5.1 in Chapter 3), it is recommended to have more heuristics. A possible enhancement is to incorporate an automatic mechanism of execution profiling (e.g., following the IP, following the `write` and/or `h-copy` executions) in the post-analysis. Studying executed parts of the memory that roughly correspond to the phenome, or the phenome's active part, would help guide the decomposition of genome and phenome if there is any. Once reproduction modes are classified better, one could preclude obviously uninteresting individuals from being selected as next source strains from which mutants are generated. As suggested, it is difficult to determine a reproduction mode as there can exist a wide spectrum of reproducers ranging from a pure self-copier to a pure von Neumann style self-reproducer. Analysis such as measuring word frequency may be useful, combined with execution profiling, but would not be sufficient alone.

To briefly spell out a possible approach of determining reproduction modes, one could algorithmically distinguish certain word level characteristics by profiling memory usage: executed-only, read-only, written-only, and mixed patterns of these. Patterns can be described more finely, taking into account how many times executed/read/written or the order of these. This could be extended to identify blocks or segments with common characteristics, such as that genotype is a read-only segment, (as opposed to a written or executed segment, or a never-read segment). This could then heuristically help or guide an attempt at demarcating genotype (or at least a candidate of genotype) from phenotype.

Combining this approach to the previously mentioned execution profiling may be a more reasonable tool for the purpose. Also, in this way, the problematic assumption that the structure does not change can be relaxed.

Towards the more important analysis, detection of changes in the genotype-phenotype mapping, it is necessary to distinguish the decomposition of genotype and phenotype. If such a decomposition is ever detected, it might be useful to classify mutants of any step explicitly, such as by mutation location and by Levenshtein distance from the input strain, to the extent that a mutant database could be created. The likelihood of a mutant experiencing a change in genotype-phenotype mapping presumably varies depending on the mutation location (or, more specifically, the location(s) where mutation is expressed). In the design of the prototype, for example, the regions where mutation most likely can cause any change in genotype-phenotype mapping are the lookup table and the decode loop, both serving as a part of the programmable constructor A of von Neumann's architecture. In a sense, the lookup table is a segment that underpins word-by-word translation (hence genotype-phenotype mapping as well) while the decode loop is a segment that represents the organism's style of translation. In brief, a change in the phenome's decode loop or lookup table, if there is any, is more likely to lead to a genotype-phenotype mapping change than in other segments. Then one might prioritise such mutants for further analysis. Of course, there may still be times when detailed case studies of individual strains, rather than automated analysis, are as effective (or more effective), especially when reproduction modes are expected to be diverse.

4.7 Additional Investigation: Redesigning the Prototype

By now, it has been pointed out that the prototype von Neumann style self-reproducer is prone to degenerate into a self-copying “look-alike”. Successful self-reproduction of such mutant self-copiers seems to be particularly facilitated by the `h-copy` instruction. As long as the read head and the write head are positioned in such a way that the parent's memory image which remains when the execution enters the copy segment, is covered, such self-copiers can safely start, continue, and finish the copy process so as to reproduce an identical offspring. Avoiding the cost of decoding, they are advantageous compared to such self-reproducers as the prototype in Avida, which favours fast reproducers regardless of the reproduction mode. Since the main challenge was originally to explore viability of a von Neumann style self-reproducing ancestor, some mechanism might be identified so as to prevent the degeneration or to mitigate the displacement by self-copiers. A possible solution is to eliminate the intrinsic advantage of such self-copiers at the Avida system level. This would be done by executing individual programs of organisms regardless of the length, rather than by allocating certain CPU time slices. As very long programs may potentially cause a prolonged run, some cutoff time should be predefined. This way, one can compare strains of different sizes purely by how they achieve production, without having to take quickness of production into account. This can be effective because there is no established algorithm for discriminating self-copying from self-reproduction with a genotype-phenotype mechanism. Another approach, which is discussed in this section, is to design a new prototype ancestor which does not rely on the `h-copy` instruction (see

Subsection 3.5.2 in Chapter 3 for the point raised with regard to this motivation).

4.7.1 Designing `h-copy`-free Prototypes

The `h-copy` instruction per se is not capable of, or sufficient for, self-copying, because many more conditions need to be satisfied in order to realise self-copying. However, once the `h-copy` instruction is successfully executed, it is powerful enough so as to greatly facilitate the copy process. Hence, it is of interest to design a von Neumann style self-reproducer without employing the `h-copy` instruction. On the basis of the prototype von Neumann style self-reproducer, there are two plausible, progressive steps for such a redesign. A first step is to recode the copy process within the phenome active segment, substituting with the `read` instruction and the `write` instruction. A second step is to eliminate any word containing an opcode for the `h-copy` instruction from the entire memory image, that is, as found in the phenome passive segment and the genome. The structure with five segments remains the same. Simply put, the first step has the `h-copy` instruction in the instruction set, whereas the second step entirely removes the `h-copy` instruction from the instruction set.

However, even if a redesigned prototype does not rely on the `h-copy` instruction, the Copy Loop segment may still be exploited to become a self-copier. Or, the copy segment may become more vulnerable than one using the `h-copy` instruction, as the copying without using the `h-copy` instruction will have an exposed stretch of code to perform the same function as the original prototype. (That is, it may require a longer code because it realises the same procedures as the `h-copy` instruction without using it, just like unpacking the `h-copy` instruction.) If that is the case, a redesigned prototype has a stretch that can potentially be perturbed for codes of the particular procedures which otherwise an instance of the `h-copy` instruction would do. Simply put, in a redesigned prototype, the likelihood that copying is perturbed is different from the prototype.

Version One: The 28-Instruction `h-copy`-free Prototype

In order to consider how to replace the function performed by the `h-copy` instruction into an equivalent using the `read` instruction and the `write` instruction, the `h-copy` instruction is unpacked here into subprocesses. The defined procedure of the `h-copy` instruction is as follows:

1. read a word from the address pointed at by the read head;
2. write the word to the address pointed at by the write head; and
3. move forward both the heads by one memory location.

When recoded using the `read` instruction and the `write` instruction, the phenome active segment becomes 313 words long (see the listing in Appendix A). As the lookup table is the same as the prototype, 28 words long, the phenome total length is 341 words long. Hence the whole length is 682 words long, potentially generating 9207 possible point-mutants. When seeded, this recoded version of the prototype divides with gestation time 59392.

It should be noted that this 28-instruction `h-copy`-free prototype seemingly includes the `h-copy` instruction's opcode: in the lookup table (i.e. the phenome passive part), and in the genome (i.e. the passive description tape). As the lookup table still includes that instruction's opcode, the way it encodes the phenome involves encoding a certain word into the word interpretable as the `h-copy` instruction (if ever executed), namely, `6` (`mov-head`) to `21` (`h-copy`). Thus within the genome, there appear words interpretable as the `h-copy` instruction (if ever executed by any chance) as often as there are `mov-head` instructions coded in the phenome active segment, also as often as the numerical value of 6 is found in the phenome passive part (i.e., in the lookup table). By design, only the phenome active part gets executed: only the words within that part count as instructions in a strict sense. Even if a memory location contains the same word, it is interpreted as an instruction only when executed. Since the lookup table is not to be executed (only accessed by a process) by definition, the words listed in it do not represent instructions, but numerical values potentially "interpretable as" certain instructions according to the order in which they are listed (i.e., they only have potential to be interpreted as an instruction in the event of/on execution).

In this version, mutation can bring about the change of a word into the word interpretable as the `h-copy` instruction, without which the prototype ancestor is coded. If a mutation brings about the word interpretable as the `h-copy` instruction in the active segment, and if other conditions for it to get executed and for it to be finished are satisfied, then it may turn out to be a self-copier or a non-reproducer; whereas, if such a word appears within a passive segment, then the mutant should definitely retain the von Neumann style reproduction mode.

Version Two: The 27-Instruction `h-copy`-free Prototype

Following the 28-instruction `h-copy`-free prototype, it is then conceivable to design a version of the prototype where neither the `h-copy` instruction nor a word interpretable as the `h-copy` instruction can occur, by employing a 27-instruction set which excludes the `h-copy` instruction from the instruction set previously used. Then, nothing can bring about a word interpretable as the `h-copy` instruction as it does not exist in that particular Avida world. Based on the new instruction set with 27 instructions, the genome (i.e. the encoded phenome) can be coded without any word interpretable as the `h-copy` instruction.

The whole length of this version is 680 words long (313 words for the phenome active part plus 27 words for the lookup table, with 340 words for the genome) (see the listing in Appendix A), which will potentially generate 8840 possible point-mutants. When seeded, this version of the prototype divides with gestation time 59218.

4.7.2 Analyses of the `h-copy`-free Prototypes

The same enhanced analysis as described in this chapter was applied to the 28-instruction `h-copy`-free prototype, and to the 27-instruction `h-copy`-free prototype, under the same condition as the original prototype: top 10 selection for mutational steps 1 and 2, and top 3 selection for mutational step 3. It was again assumed that the genotype-phenotype decomposition is retained as dividing the whole memory image into the first half and the latter half.

Some settings of the analysis must be changed according to the gestation time. In the original prototype's case, the lower bound of gestation time was set to be 52218×0.5 (in the post-analysis) and the upper bound 644×163 (in the Avida main configuration file). Now, for the redesigned versions, the 28-instruction `h-copy`-free prototype and the 27-instruction `h-copy`-free prototype, the lower bound was changed to be the half of each gestation time: 59392×0.5 and 59218×0.5, respectively. In addition, the upper bound was changed to be approximately twice each gestation time: 682×174 and 680×174, respectively (note that in the Avida configuration file, the value is to be set as multiple of the size).

The first-step mutants of the 28-instruction `h-copy`-free prototype and of the 27-instruction `h-copy`-free prototype distribute by total number of strains as shown in Figure 4.23 and Figure 4.24. Those first-step mutants of these recoded prototypes are distributed by gestation time as shown in Figure 4.25 and Figure 4.26.

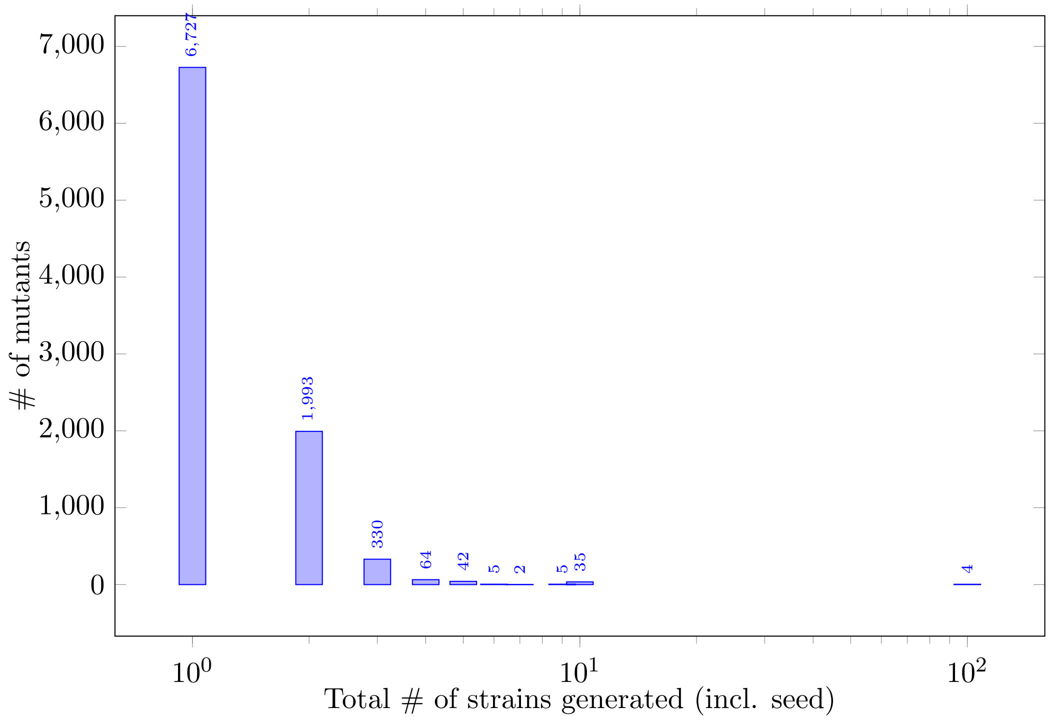


Figure 4.23: Distribution of the 9207 first-step point-mutants of the `h-copy`-free 28-instruction prototype by total number of strains generated. For legibility, those between 10^1 and 10^2 are combined to fall in 10^1.

Judging from the first-step mutational analysis, the `h-copy`-free 28-instruction prototype has:

- 4 indeterminate second-step mutants ($< 0.05\%$);
- 9203 determinate second-step mutants ($> 99.95\%$), out of which 946 (10% of total) are with potential of exponential growth.

Likewise, the `h-copy`-free 27-instruction prototype has:

- 10 indeterminate second-step mutants ($< 0.2\%$);
- 8830 determinate second-step mutants ($> 99.8\%$), out of which 917 (10% of total) are with potential of exponential growth.

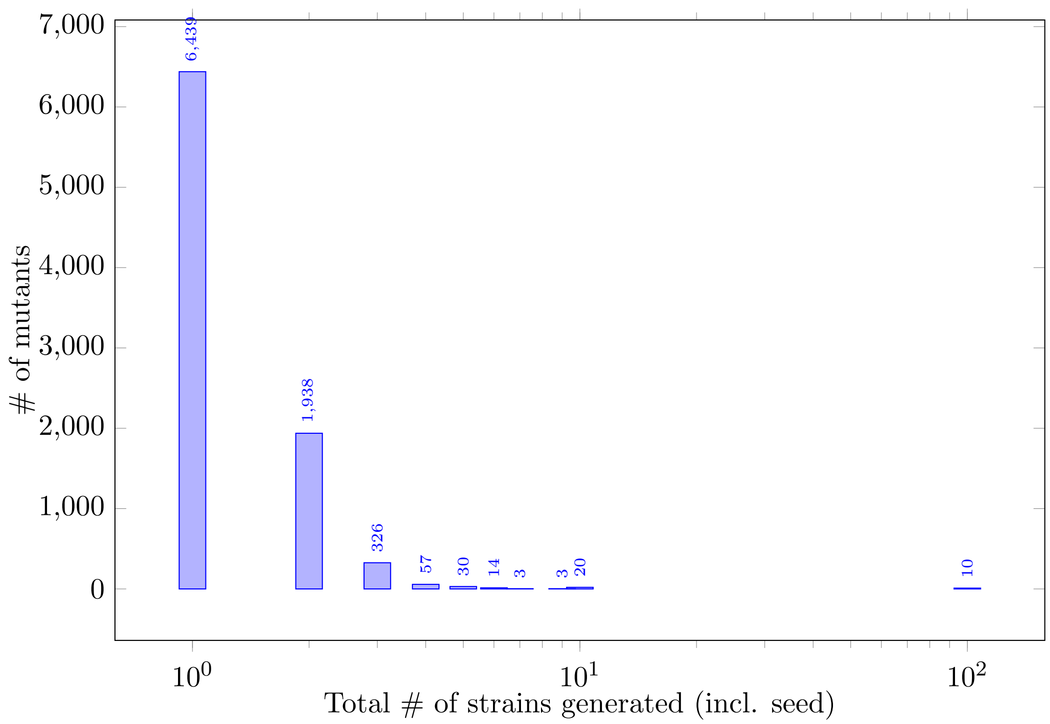


Figure 4.24: Distribution of the 8840 first-step point-mutants of the `h-copy`-free 27-instruction prototype by total number of strains generated. For legibility, those between 10^1 and 10^2 are combined to fall in 10^1.

Neither of these versions led to any first-step mutant which generates more than one exponentially growing strain in this particular setting, just as the original prototype. Similarly, in those cases, no mutants selected for subsequent analysis were found to be generating more than one exponentially growing strain, except for one in the 28-instruction version prototype's mutational pathways at the second-step level.

Lineage of the `h-copy`-free 28-instruction Prototype

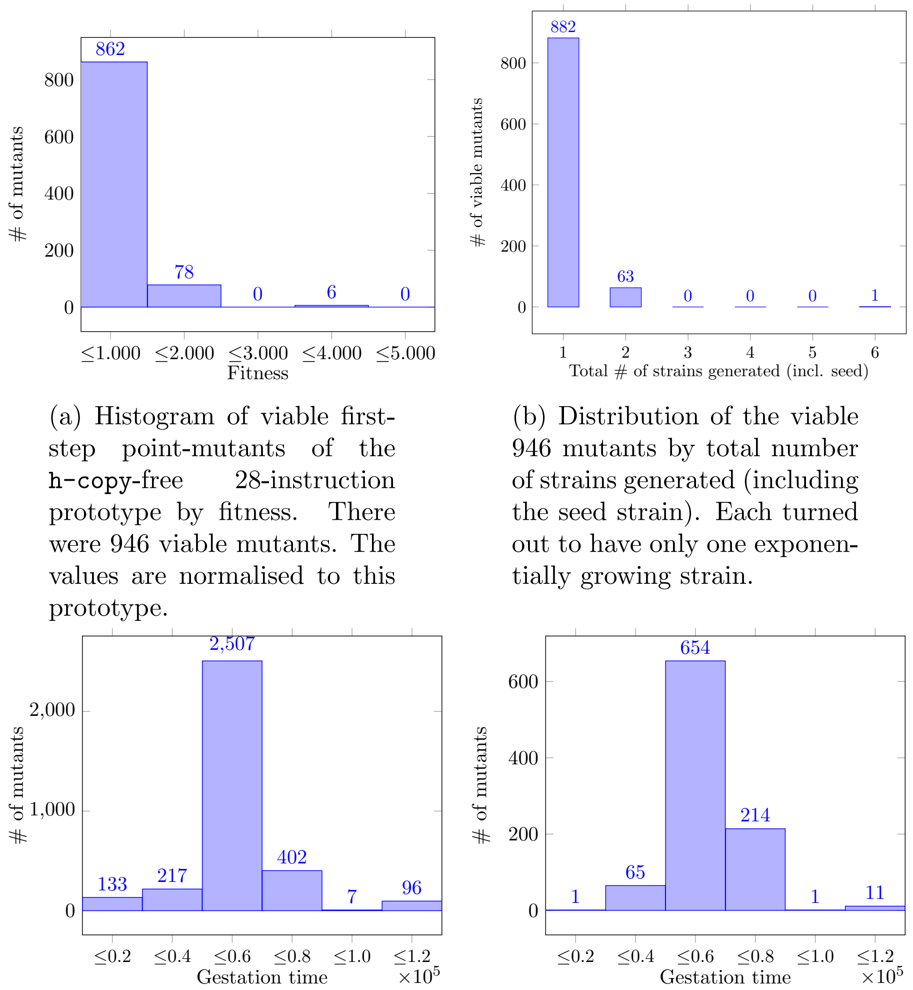
Among the first-step mutants of the `h-copy`-free 28-instruction prototype (named `hcf28_org-0`) are:

- 4 indeterminate mutants that reach the recursion limit;
- 0 mutant that generates more than 1 viable strain;
- 947 mutants that generate 1 viable strain.

The tree of the mutants analysed is shown in Figure 4.27. The ranking of its first-step mutants is shown in Table 4.11.

The ranking of the second-step mutants of `hcf28_org-0-6131` is shown in Table 4.12. Among the whole second-step mutants of the first-step mutant `hcf28_org-0-6131` are:

- 24 indeterminate mutants that reach the recursion limit;
- 0 mutant that generates more than 1 viable strain;
- 1434 mutants that generate 1 viable strain.



(a) Histogram of viable first-step point-mutants of the `h-copy`-free 28-instruction prototype by fitness. There were 946 viable mutants. The values are normalised to this prototype.

(b) Distribution of the viable 946 mutants by total number of strains generated (including the seed strain). Each turned out to have only one exponentially growing strain.

(c) Histogram of fertile first-step point-mutants by gestation time. There were 3362 fertile mutants out of the total of 9207.

(d) Histogram of the viable 946 mutants by gestation time.

Figure 4.25: Fitness distribution of the first-step mutants of the 28-instruction `h-copy`-free prototype.

The ranking of third-step mutants of `hcf28_org-0-6131-6563` is shown in Table 4.13. Among the whole third-step mutants of the second-step mutant `hcf28_org-0-6131-6563` are:

- 24 indeterminate mutants;
- 0 mutant that generates more than 1 viable strain;
- 1462 mutants that generate 1 viable strain.

Lineage of the `h-copy`-free 27-instruction Prototype

Among the first-step mutants of the `h-copy`-free 27-instruction prototype (named `hcf27_org-0`) are:

- 10 indeterminate mutants;

Mut. ID	Strains	Viable	Fittest	Fitness	Gest. Time
6131	2	1	s_0	3.742	59392
5901	2	1	s_0	3.730	59439
6301	2	1	s_0	3.712	59392
6400	2	1	s_0	3.201	59392
6184	2	1	s_0	3.094	59392
3900	1	1	s_0	1.883	31512
4062	1	1	s_0	1.883	31512
4224	1	1	s_0	1.883	31512
3928	1	1	s_0	1.883	31513
1281	1	1	s_0	1.883	31558

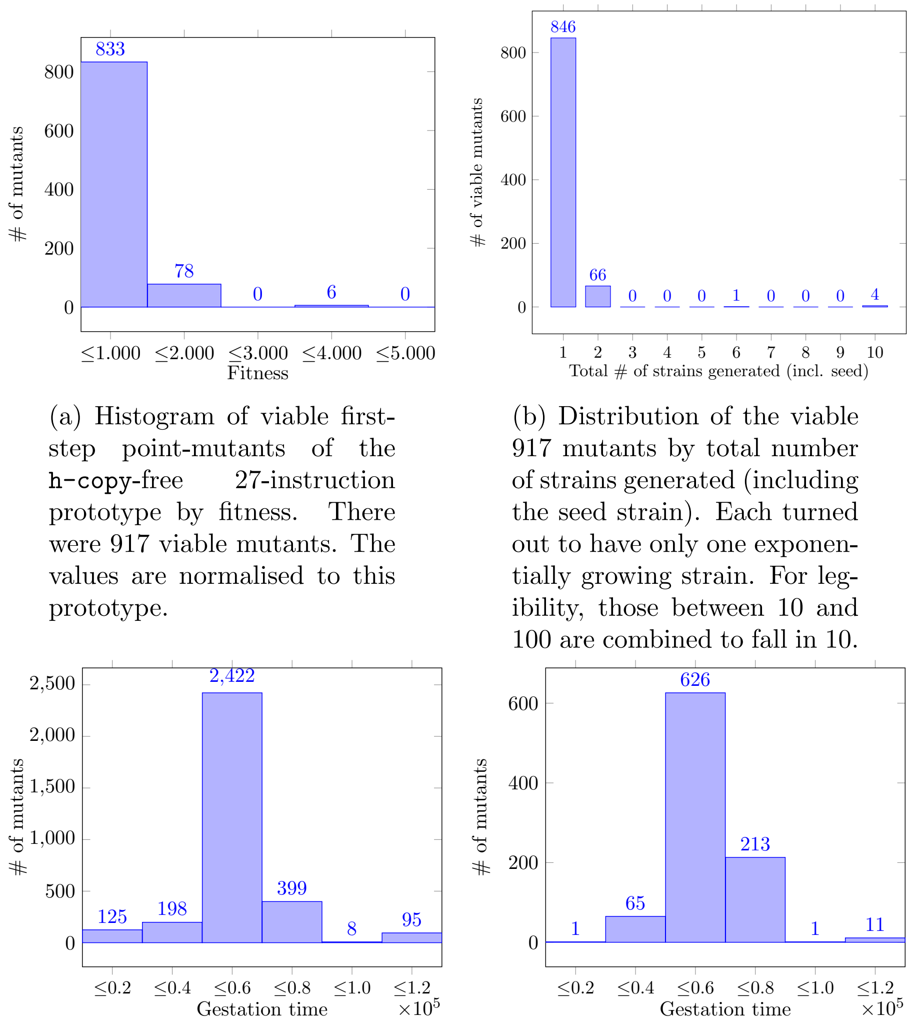
Table 4.11: The first-step top 10 point-mutants of the ranking of the 28-instruction `h-copy`-free prototype `hcf28_org-0`.

Mut. ID	Strains	Viable	Fittest	Fitness	Gest. Time
6563	2	1	s_0	3.908	58712
2377	3	1	s_0	3.754	31512
1021	2	1	s_0	3.742	59051
1051	2	1	s_0	3.742	59051
1052	2	1	s_0	3.742	59051
3504	2	1	s_0	3.742	59051
912	2	1	s_0	3.742	59051
1002	2	1	s_0	3.742	59392
1006	2	1	s_0	3.742	59392
1012	2	1	s_0	3.742	59392

Table 4.12: The second-step top 10 point-mutants of the ranking of the first mutant `hcf28_org-0-6131`.

Mut. ID	Strains	Viable	Fittest	Fitness	Gest. Time
2377	3	1	s_0	3.920	30152
1021	2	1	s_0	3.908	58371
1051	2	1	s_0	3.908	58371

Table 4.13: The third-step top 3 point-mutants of the ranking of the first mutant `hcf28_org-0-6131-6563`.



(a) Histogram of viable first-step point-mutants of the `h-copy`-free 27-instruction prototype by fitness. There were 917 viable mutants. The values are normalised to this prototype.

(b) Distribution of the viable 917 mutants by total number of strains generated (including the seed strain). Each turned out to have only one exponentially growing strain. For legibility, those between 10 and 100 are combined to fall in 10.

(c) Histogram of the first-step point-mutants by gestation time. There were 3247 fertile mutants out of the total of 8840.

(d) Histogram of the viable 917 mutants by gestation time.

Figure 4.26: Fitness distribution of the first-step mutants of the 27-instruction `h-copy`-free prototype.

- 0 mutant that generates more than 1 viable strain;
- 918 mutants that generate 1 viable strain.

The tree of the mutants analysed is shown in Figure 4.28. The ranking of its first-step mutants is shown in Table 4.14.

The ranking of its second-step mutants `hcf27_org-0-5904` is shown in Table 4.15. Among the whole second-step mutants of `hcf27_org-0-5904` are:

- 24 indeterminate mutants;
- 0 mutant that generates more than 1 viable strain;
- 1392 mutants that generate 1 viable strain.

The ranking of its third-step mutants of `hcf27_org-0-5904-6320` is shown in Table 4.16. Among the whole third-step mutants of `hcf27_org-0-5904-6320` are:

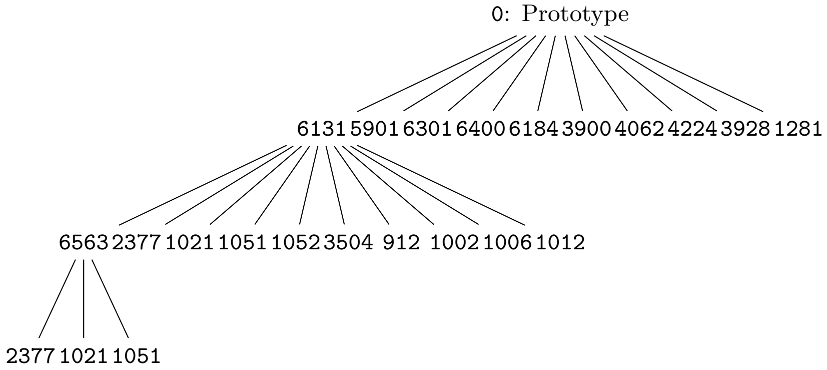


Figure 4.27: Tree of selected mutants of the 28-instruction `h-copy`-free prototype mutants up to the third mutation generation.

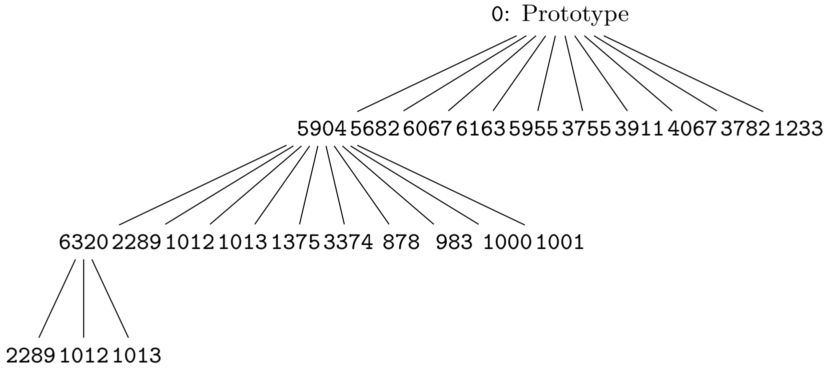


Figure 4.28: Tree of selected mutants of the `h-copy`-free 27-instruction prototype mutants up to the third mutation generation.

- 24 indeterminate mutants;
- 0 mutant that generates more than 1 viable strain;
- 1419 mutants that generate 1 viable strain.

Analysis Results

Qualitatively speaking, both of the redesigned prototype ancestors exhibit similar characteristics to the original prototype ancestor. For example, the distributions of the first-step point mutants by total number of generated strains of the redesigned prototype ancestors showed the same trend as the original prototype ancestor, where the majority of mutants do not generate descendant strains, and where, as the number of potentially generated strains increases, the number of such mutants decreases. In terms of fitness, the redesigned prototype ancestors exhibit the same trend, with almost the same shape of the histograms. The percentage of viable mutants was approximately 10% for any of these ancestors (i.e., 10.3% for the `h-copy`-free 28-instruction prototype, 10.4% for the `h-copy`-free 27-instruction prototype, and 10.1% for the original prototype). Of viable mutants, those with a fitness value equal to or less than the respective prototypes are in the majority. These are similar to the original prototype ancestor, except that the original prototype ancestor have somewhat more mutants whose fitness is more than 2.0 but equal to or less

Mut. ID	Strains	Viable	Fittest	Fitness	Gest. Time
5904	2	1	s_0	3.743	59218
5682	2	1	s_0	3.731	59265
6067	2	1	s_0	3.713	59218
6163	2	1	s_0	3.221	59218
5955	2	1	s_0	3.115	59218
3755	1	1	s_0	1.883	31420
3911	1	1	s_0	1.883	31420
4067	1	1	s_0	1.883	31420
3782	1	1	s_0	1.883	31421
1233	1	1	s_0	1.883	31466

Table 4.14: The first-step top 10 point-mutants of the ranking of the 27-instruction `h-copy`-free prototype `hcf27_org-0`.

Mut. ID	Strains	Viable	Fittest	Fitness	Gest. Time
6320	2	1	s_0	3.908	58540
2289	3	1	s_0	3.754	31420
1012	2	1	s_0	3.743	58878
1013	2	1	s_0	3.743	58878
1375	2	1	s_0	3.743	58878
3374	2	1	s_0	3.743	58878
878	2	1	s_0	3.743	58878
983	2	1	s_0	3.743	58878
1000	2	1	s_0	3.743	59218
1001	2	1	s_0	3.743	59218

Table 4.15: The second-step top 10 point-mutants of the ranking of the first mutant `hcf27_org-0-5904`.

Mut. ID	Strains	Viable	Fittest	Fitness	Gest. Time
2289	3	1	s_0	3.920	30064
1012	2	1	s_0	3.908	58200
1013	2	1	s_0	3.908	58200

Table 4.16: The third-step top 3 point-mutants of the ranking of the first mutant `hcf27_org-0-5904-6320`.

than 3.0, unlike the redesigned prototype ancestors, neither of which have mutants with fitness within that range. The reason behind this difference may be of interest for future research.

In terms of gestation time, the majority was those that fall within the same gestation time range as the respective prototypes, and the farther from this range the gestation time is, the fewer mutants there are which have such a gestation time. These redesigned prototype ancestors are, again, very similar to the original prototype ancestor. Assuming gestation time represents a reproduction mode, the percentage of self-copiers (whose gestation time is half of the respective prototype ancestors among fertile mutants) is nearly the same as well (i.e., 6.5% and 6.1% for the `h-copy`-free 28-instruction prototype and for the `h-copy`-free 27-instruction prototype, respectively, whereas 5.3% for the original prototype). The percentages of supposed self-copiers are basically similar.[10]

In the automation of the analysis spanning a few, two and three, mutational steps, what was noticeable in the results of the redesigned prototype ancestors is that the captured fittest mutants were all source strains within lineages, as opposed to the original prototype ancestor, whose fittest strains within lineages included not only seed strains but also (first) descendant strains. This dissimilarity between the redesigned and the original prototypes is interesting in itself, but one would need further investigation in order to tell whether that this happened by coincidence or not, and if it is not by coincidence, why.

It is difficult to conclude that the `h-copy` instruction was a decisive factor behind the degeneration into a self-copier from these results solely. Probably, the exclusion of the `h-copy` instruction per se does not substantially affect the likelihood of self-copy mutation. It is presumably because, despite not using the `h-copy` instruction, the redesigned prototype ancestors follow the same subroutines as the original prototype ancestor and the structure of the self-reproductive cycle of them is essentially the same as the original. Ancestors with different structures should be designed and examined in order to deepen the understanding on this.

4.7.3 Reflective Remark: Redesigning the Prototype

In the preceding subsections, `h-copy`-free designs of the prototype, a von Neumann style ancestor in Avida, are introduced and described. Two different versions of the prototype were analysed for evolutionary potential under the same analysis tool. As far as the analysis could reveal, neither of them encounters any more non-trivial self-reproducer than the original prototype. Nevertheless, to determine whether they are resiliently viable in evolution compared to the original prototype, analysis deeper than three mutation steps, ideally with a wider range of candidate selection per generation, would be required. Analysing those versions, it was also minimally demonstrated that the proposed method was applicable to other strains, or other von Neumann style self-reproducers.

In the first place, this additional investigation was motivated by the previously discovered mechanism of degeneration to a self-copier (in Chapter 3) which indicated a potential of a certain instruction specialised for copying (`h-copy`). It was at least clarified that that

[10] Or numerically, on the surface, the percentage even increases slightly in those redesigned versions; but of course this assumption is very approximate in the first place, so concluding that it increases would not be reliable. At least it is safe to say that these percentages are comparable to each other in terms of the order, less than 10%.

type of quick degeneration to a self-copier of a less than half of the prototype gestation time is estimated to occur approximately in 5% of the viable first-step point-mutants: it is not the majority, but more like a part of diversity. Other reproduction modes among the rest of the viable mutants, either between or beyond the standard self-copying and the von Neumann style self-reproduction, await further observations. It is speculated that the broader the analysis scope is, the more diverse reproduction modes it may encounter.

It was following the above consideration that the two `h-copy`-free versions of the prototype were designed. There is an analogous cluster of mutants with gestation time about half of the gestation time of the `h-copy`-free 28-instruction prototype, which are speculated to be cases of the degeneration corresponding to the first encountered one; they account for approximately 6% of the viable first-step point mutants. For the `h-copy`-free 27-instruction prototype as well, there is a corresponding cluster of approximately 6%. Those versions were analysed with the same enhanced analysis as the prototype did. Within the current limited scope of observation, there was no noticeable difference from the original prototype in this deterministic analysis framework. In a stochastic framework as in the standard Avida, either of those `h-copy`-free von Neumann style self-reproducers may or may not emerge advantageous in the Avida world.

With regards to redesigning the prototype, the work presented above is only a preliminary, illustrative example. This is a wider scope to explore better or more interesting design of an ancestral von Neumann style self-reproducer (not necessarily even with the same phenotypic structure consisting of the five segments). The proposed enhanced analysis is applicable to them as much as to the original prototype. That being said, in the particular case studied, the analysis covering a few mutation generations could not find immediately non-trivial viable mutants generated from those redesigned versions of the prototype: for example, there was no mutant with more than one distinct exponentially growing strain. This analysis hinges upon the presumption that the mutants somehow retain the genotype phenotype decomposition; however, importantly, this does not necessarily hold. An additional step has to be taken to be able to distinguish the decomposition. Implementing it would contribute to the development of a more generic mechanism that distinguishes reproduction modes. Results regarding gestation time showed a similar tendency to the original prototype, indicating that there may still be a number of degenerative cases even where the `h-copy` instruction does not exist. If that is the case, it is presumably because of the structural design itself. The effect of these new versions on evolvability could be further investigated by seeding these and running experiments over an “evolutionary” period of time, or by detailed case study examination.

4.8 Closing Remarks

In Chapter 4, mutational analysis that was conducted for evolutionary characterisation is described. The subtlety of self-reproduction and viability are discussed through conceptually classifying strains in Section 4.2. Section 4.4 makes progress in enhancing the existing mutational analysis, using mutants of the prototype introduced in Chapter 3 as a model set for analysis. In it, the modification of the built-in analysis tool of Avida introduced in Section 4.3 is described, and how the modified analysis can cover the full lineage of an incubated individual is explained. Further, viability quantification (fitness) is introduced as a generic indication of potential for exponential growth. In the following Section 4.5, the enhanced analysis was automated to be able to cover multiple steps of mutation. Additionally, preliminary attempts of redesigning the prototype were made in Section 4.7, and the same enhanced analysis scheme proposed in the previous sections was applied to two versions of redesigned prototype.

The enhanced analysis tool succeeded in demonstrating a finer-grained classification of mutants. In the case of prototype's first-step mutants, 10% were viable. The automation succeeded in performing multi-step mutational analysis as proposed and picking up more evolutionarily interesting candidates among others. However, it was not actually able to discover any particularly interesting mutant in terms of the original research motivation, partly because of the tool's incapability of classifying the reproduction mode and presumably essentially because of some factors inherent in the structure of the prototype. To partially examine the design of the prototype, two alternative `h-copy`-free prototype ancestors were introduced and the same enhanced analysis was applied to them. It was demonstrated that these alternative designs exhibit mostly similar evolutionary characteristics as far as this analysis tool could reveal. Both in the original prototype's case and in the redesigned prototypes' cases, less than 10% of fertile first-step mutants turned out to be putative degenerate self-copiers. At the same time, however, it is recommended that further research in this direction (i.e. reproduction mode classification along with von Neumann style self-reproducing ancestor design) be conducted for a more proper understanding of the ancestral design and its effect on how it can direct behaviours on mutational pathways.

Chapter 5

Conclusions and Implications

5.1 Thesis Summary

The significance of the mutable genotype-phenotype mapping of the von Neumann architecture of machine self-reproduction is introduced in Chapter 2. That is, the evolvability of the genotype-phenotype mapping and the effect of such a mapping on the evolution of the self-reproducer that employs it are of interest. Thus, the current research of this thesis was concerned with: designing and seeding an instance of such a self-reproducer (in Chapter 3); and in exploring an analysis method, proposing an analysis framework, and demonstrating how a strain's evolutionary characteristics can be systematically investigated (in Chapter 4).

Chapter 3 describes and presents the construction of a prototype von Neumann style ancestral self-reproducer with a genotype-phenotype mapping subject to evolution within the particular artificial life platform Avida. As part of the investigation, the evolutionary behaviour and dynamics of the prototype is characterised. It is clarified that mutational proximity between a self-copying reproducer and a von Neumann style self-reproducer can indeed be surprisingly close. It would be premature to make large-scale generalisation from the given results; however, it can be said that it is theoretically possible for a self-copying reproducer to give rise to a von Neumann style self-reproducer in Avida, in a way reversing the observed process of the mutation. Although, in practice, that particular pathway could not evolve, it serves as a proof of principle.

The discovery of the self-reproducer's quick evolutionary degeneration into a self-copier raises a question: How likely is it that such degeneration takes place? Accordingly, this is one of the questions tackled in Chapter 4. Chapter 4, however, takes a more holistic approach. Firstly, a logical exposition on strain classification and self-reproduction is laid out in order to investigate mutant strains, focusing on a necessary condition, referred to as *viability.* Then a framework where viability is quantified is proposed. Within this framework, the first-step mutants of the prototype are examined. The investigation of the spectrum of single-point mutants of the particular self-reproducer prototype in an attempt to classify viable mutant candidates for evolvable genotype-phenotype mapping is illustrated. The same analysis method was applied to some multi-step mutants for demonstration purposes.

5.2 Contribution and Concluding Remarks

A prototype von Neumann style self-reproducer implemented in Avida was characterised via an automated enhanced analysis tool. The enhanced analysis incorporates detection of viability which is defined as potential of exponential growth, necessary for any sustained, neo-Darwinian evolution. A finer-grained mutant classification was achieved. Theoretically, the framework is scalable and extendable for wider spectra of, and more steps of, hypothetically possible point-mutant strains of an arbitrary organism. There were some mutants that have a viable strain generated deterministically but "indirectly" within the lineage. In most cases, however, seed strains or first descendant strains were the viable ones.

Again, the original intention was to search for evolutionary elaboration, variation, or complexification of genotype-phenotype mapping. Simply put, there are two conditions for this: (a) that there are viable variations; and (b) that there are selectively favoured or neutral variations. The current research was primarily about the first condition, exploring the potential viability, and questioned what happens if one does not discount this potential. It was investigated through a minimal model; and indeed, found unlikely to occur, at least in this one example case. The result was based on systematic, but heuristic, investigation, in contrast to actually running evolutionary experiments.

On the other hand, the latter condition about selection was not focused on in the current research, so the selective performance is one appropriate topic in the future direction. While one of the current findings is that the reproduction architecture can change (even to ones with no identifiable genotype-phenotype mapping), ultimately, it is ideal to be able to locate genotype-phenotype mapping changes. However, even (algorithmically) determining working components of an arbitrary mutant strain is not a trivial task. (This situation would presumably be formally undecidable, equivalent to the halting problem, in general.) Nonetheless, it is recommended to heuristically attempt it. The inspiration of demonstrating evolutionary complexification of genotype-phenotype mapping has not yet been found impossible or impractical; and the question whether viable reproduction (self-consistency in the genotype-phenotype mapping) may be recovered deterministically is left open.

The current research leads to improved understanding of a von Neumann style self-reproducer in Avida from two perspectives using the hand-designed prototype ancestor: mutational pathways and ancestral design.

5.2.1 Mutational Pathways of the Prototype

In the preliminary experiment of observing the behaviour of a prototype von Neumann style self-reproducer in Avida, displacement of the hand-designed prototype ancestor by a viable self-reproducer which preserves the von Neumann style architecture was not observed. What was observed, in fact, was the degenerative displacement by self-copiers, not the alteration or sophistication of the von Neumann style architecture. In other words, no mutation of genotype-phenotype mapping, or a mutated "programmable constructor" with a working functionality, was observed, except for loss of repertoire of unemployed instructions. On the one hand, it signifies the loss of the decomposition of genotype and phenotype for the individual ancestor; but on the other hand, it signifies the loss of self-referential logic, the loss of genotype-phenotype mapping, and the loss of the division of labour for the lineages derived from the ancestor.

Two slightly similar taxonomic concepts were introduced: *unclassified* and *indeterminate*. Unclassified strains refer to ones which have lineages untraced due to the design of the analysis tool which arbitrarily picks up one of the two lineage paths at each division, whereas indeterminate strains refer to ones which have lineages untraced due to a large lineage size. The enhanced analysis succeeded in minimising the number of previously unclassified individuals of the particular set of mutants, reducing close to zero percent. The indeterminacy resulted from the existing minority of lineages that can potentially grow indefinitely diverse, exhausting the memory space for storing distinct strains. Practically, those cases are treated as exceptions and need case-by-case analysis. Though excluded for technical reasons, it is ideally worthwhile to include those lineages as viable mutants may still arise in such a lineage.

Analysing the first-step mutants, determinate ones can be classified as either viable or non-viable. This finding that viable mutants account for approximately 10% of the whole is non-trivial in its own right. The percentage can be regarded as high, in the sense that the designed prototype has its active part whose memory image occupies more than 10% of the whole memory image; the active part is expected to be vulnerable to a single-point mutation as the program is coded in a more complicated manner compared to the standard organism in Avida. On the contrary, the percentage can also be regarded as low, from the perspective of the typical situation of Avida where an experimenter would use mutation as the main drive of the evolution of "ancillary machinery" (which was omitted from the prototype early in the design phase). In any case, the result urges reconsideration on the design of the prototype for better understanding of the implementation of a von Neumann style self-reproducer in Avida.

Naturally, the proposed type of systematic analysis suffers from intrinsic combinational explosion. Current limitations include:

- limited generations for a lineage per incubation;
- limited size of selection of candidates from concurrent mutants;
- limited number of mutation steps covered; and
- limited mutation types other than point mutation.

Furthermore, the analysis tool simply assumes that the reproduction mode it handles is the same von Neumann style self-reproduction architecture as the particular prototype, the original design of which is proposed earlier. Further enhancement would at least require some kind of mechanism for detecting the reproduction mode employed by an organism. This is short-circuiting the process of generating mutants in the sense that it assumes the same genotype-phenotype architecture as the prototype with the structure of the lookup table within the phenome. Even if admitting this assumption, there is another short-circuit where mutations are enforced to be expressed by the automation in the preparation before incubation. Ideally, it should be strains that carry a mutation in the genome, but that have not expressed it, that should be incubated. That is, the analysis would be better to take an arbitrary phenome and "attach" a corresponding genome on a case-by-case basis.

5.2.2 Ancestor Design

A single mutation can change the behaviour radically, as in the reported case of degeneration, within the current system. It should be stressed that it is not reasonable to expect a self-reproducer to acquire significant additional functionality (corresponding to the "ancillary machinery"), especially when no mutation to change the organism's size takes place. To support the emergence of such ancillary behaviour, an effective approach may be to enable more general mutational changes such as insertion, because the prototype initially lacks any "spare capacity" in its memory image for ancillary machinery. This is analogous to the case of typical standard self-copying ancestors in Avida studies which have instances of the `nop-A` instruction at a stretch for the future evolution of computation. Other possible approaches include: imposing a fitness function (i.e., external, or environmental); or, enabling organisms to interact in a more sophisticated way. These approaches are reminiscent of those taken by the original Avida. In other words, while the current research of the von Neumann style self-reproduction is directed towards intrinsic potential when instantiated in the Avida universe, a future direction of the research may reasonably be towards extrinsic potential, or the effects of interaction with the environment or with other individuals.

Some tendency for degeneration was expected, given that self-copiers intrinsically have an advantage for efficiency, and become dominant once they appear in this particular system. As the presented result is highly dependent on this advantage intrinsic to the system, it cannot immediately be extrapolated to other in-silico platforms (much less in-vitro or in-vivo situations). However, what was remarkable here is the finding of a surprisingly accessible mutational pathway from the designed ancestor to the self-copier: the composition of the memory image is changed only by two words.

It is not clear whether the degeneration to a self-copier is peculiar to the designed ancestor, or more generic to von Neumann style self-reproducers in Avida. Further research is therefore recommended in the regard of their evolutionary behaviour and character. Once again, it will be necessary to develop a method to classify a set of strains by reproduction mode and combine it as another layer with the enhanced analysis.

More generally, a stochastic framework (as in the standard Avida) can be thought of as evolutionary search of a kind. This is a framework that, for instance, may eventually reveal such a mutational pathway that is unexpected (or, "novel") from an engineering point of view. So, though less relevantly, the direction of the current research of enhancing mutation analysis may lead to, or be beneficial for, shedding more light on novelty search. This is of course not to deny the importance of a systematic framework for better understanding of the evolutionary characterisation of an organism.

5.3 Future Prospects

The present research work on the implementation of a von Neumann style self-reproducing ancestor and the associated enhanced analysis is foundational and methodological. The research revealed some characteristics of and problems of the design of the prototype ancestor, and these need to be reconsidered.

Concerning the presented investigation and the proposed framework, there are two immediate questions to be answered.

- For more effective mutation analysis: What mechanism is best for detecting the genotype-phenotype mapping mutation and non-trivial evolution, and how can the automated enhanced analysis be made optimal?
- For more effective ancestor design: What kind of design of a seed von Neumann style ancestor would have non-trivial evolvability of the genotype-phenotype mapping, and how can such an ancestor effectively be designed?

Further questions that are not particularly addressed in the current research can be put as follows.

- For a better evolutionary understanding: What selection pressure can effectively cause mutation and evolutionary elaboration of the genotype-phenotype mapping?
- For a general understanding: What extrapolation, generalisation and applications can be speculated from this line of research, outside Avida?

5.3.1 More Effective Mutation Analysis

The mutation analysis could be enhanced with a capability for deeper and large-scaled viability classification, realised with more computational resource. One straightforward approach is to apply the proposed analysis for more mutation steps. Heuristics must be gained about the scale of analysis (i.e. the number of mutational steps) and how to prune mutants (i.e. the size of selection for subsequent step). Assume that roughly m strains can be analysed per hour, and that s candidates at the end of each step are selected for the subsequent analysis. Then, the number of strains to be analysed by the n-th step can be given as: $\sum_{g=0}^{n} s^g$; and the analysis time in hours as: $\sum_{g=0}^{n} s^g/m$. (Let $m = 2$, $s = 2$, and $n = 10$ for example, and analysis will take more than 42 days.) The method of this enhanced analysis may be a costly but effective way to narrow down non-trivial candidates, or to discard non-viable (hence trivial) candidates.

Moreover, longer evolutionary runs are preferable to get more complete profiles of mutants. Even looking at a first few generations of fertile mutants would be enough for the preliminary purpose of judging the mutational effect. In doing so, it would be more efficient presumably to run multiple Avida simulations in parallel since there are more than three hundred memory locations in the genome and there are nearly thirty kinds of instructions in the instruction set from which the after-mutation memory content are chosen. Besides, enabling parallel simulations will certainly speed up Avida experiments in general given appropriate hardware support. These are assuming single-point mutations, but for a more realistic analysis, various mutation types should be introduced.

As mentioned in Chapters 2 and 3, Avida presupposes computation universality informally. As long as this holds, to say that there must be paths to organisms (machines) with some arbitrary complexity in this Avida universe is valid; but the point here is that they are not necessarily accessible within a reasonable time or scale or the neo-Darwinian selection.

Steps for Reproduction Mode Classification

To discuss steps for reproduction mode identification and classification, it must be reiterated that in the development of the enhanced analysis tool, it was blindly assumed that the von Neumann style self-reproduction is still present in each self-reproducer (as indicated in Chapter 4). The analysis will be improved if, for any given self-reproducers, it can automatically distinguish von Neumann style self-reproducers from self-copiers. Although it is fundamentally difficult to algorithmically determine reproduction modes (as clarified both in Chapters 3 and 4), a first few steps towards such capability can be heuristically taken by:

- measuring gestation time and size,
- constructing an execution profile (for the whole program),
- constructing an execution profile of the `h-copy` instruction (if present),
- constructing an execution profile of the `read` instruction, and
- constructing an execution profile of the `write` instruction.

The combination of gestation time and size crudely suggests the superficial efficiency of self-reproduction. This method may suggest which mutants are self-copiers, which are generally more efficient than having decoding process, though this is not a definitive indication. An execution profile for the whole program of a strain helps understand how the strain is structurally segmented. An execution profile of the `h-copy` instruction is indicative of the likelihood that the self-reproducer is a self-copier that utilises the `h-copy` instruction. If an organism turns out to be a self-reproducer and to use the `h-copy` instruction even once, it may be a self-copier like the reported one, unless it has some preventive mechanism (moreover, having such a mechanism may or may not increase the size of the organism). If a self-reproducer has a decode process, an execution profile of the `read` instruction will indicate the use of the `read` instruction, since there is no other instruction in the instruction set used that can read anything from anywhere in the memory image. An execution profile of the `write` instruction may provide useful information when combined with other execution profiles: if it is found that there are instances of the `write` instruction and of the `h-copy` instruction, then the self-reproducer may have copy and decode segments; and if, in addition, the `write` instruction is used as many times as the `h-copy` instruction, then the reproduction mode may be similar to the original prototype.

Simply put, the execution of the instructions `h-copy`, `read` and `write`, is key to reproduction, so they naturally deserve attention during reproduction mode detection. The `h-copy` and `read` instructions are the ones that can read from an arbitrary location when executed, while the `h-copy` and `write` instructions are the ones that can write into an arbitrary location when executed. More heuristics might be obtained for generalisation of the relationship between the observable reproduction mode and these attributes through case studies of currently available candidates, looking into trace files produced by the original Avida analysis tool. However, any single such attribute does not guarantee the reproduction mode, so it may not be straightforward to automate the analysis. Detecting the execution solely of the `h-copy` instruction to eliminate self-copiers may be an easy first step. This is, at most, heuristic, because not much is known yet about different reproduction modes in Avida comprehensively. With more empirical studies on this, a unifying view may be obtained.

5.3.2 More Effective Ancestor Design

The minimality or optimality (in terms of size and the number of steps) of the hand-designed prototype ancestor is unknown. Nonetheless, as described, the prototype was designed so that it has the constructor, the copier, and the control (A, B and C) and is without the ancillary machinery (D) part. So it should be reasonably close to minimal and optimal for the design, considering the steps it is supposed to take, compared to the standard Avida ancestor. If the ancestor is to be redesigned, a different initial genotype-phenotype mapping may be selected. Importantly, it may be necessary to redesign (and probably enrich) the instruction set to be used, since the default instruction set was tailored more for observing particular biologically-inspired emergent phenomena. That is, there seems to be some arbitrariness in the choice of the 26 instructions in the default instruction set, and one cannot tell a priori whether these are all equally essential for whatever purposes (e.g. self-reproduction) they are intended for. Furthermore, ideas for a more human-tractable design scheme, something suitable for manual programming, should be considered and tested, because ancestors cannot necessarily be automatically or algorithmically designed easily, and may all have to be hand-designed. (In such cases, one needs to be aware whether and how much the Avida system is altered from the original. There may be an argument for building some alternative platform from scratch, but that is out of the current purview). A good design would be such that would not necessarily sacrifice evolvability over human-programmability. By extension, there would be a role for some higher level language "compiler"-type tool to support human programmers in such a line of research.

Degeneration experienced by the prototype can be regarded as fragility, while it can also be regarded as adaptation in the Avida world in a sense. In any case, in terms of the purpose to investigate the von Neumann style self-reproduction architecture in an evolutionary context, such degeneration is one problem to be overcome. It was partly overcome by designing the `h-copy`-free versions of the prototype. The first of these was initially without any `h-copy` instruction to execute, albeit was still accessible via mutation. The second excluded the `h-copy` instruction instruction completely from the instruction set. Compared to the prototype, at least, the ease of self-copying based on the `h-copy` instruction is impaired in both versions (to a greater extent in the latter version). For better characterisation, these versions would need to be analysed for more generations, as well as with a wider scope. The effectiveness can also be verified by seeding either or both of them together with the original prototype in the standard Avida over sufficient time steps for sufficient duration. It will clarify whether degeneration is peculiar to the original prototype employing `h-copy`, or is essentially generic to the design of the von Neumann architecture of self-reproduction. Where degeneration is of less frequency, the ancestral and derived von Neumann style self-reproducers (more likely) will survive and evolve to be observed.

The current research aimed towards the evolution of genotype-phenotype mapping, so the initial choice of genotype-phenotype mapping was crucial. This is likely to hold in general, as the evolvability probably to a certain extent hinges on the initial mapping, at least in the sense that the initial condition generally matters in evolutionary dynamics. The lookup table of the current design is nothing but one of all possible implementations of translation. Moreover, the lookup table does not have to be a one-to-one substitution code; this was just for the relative simplicity and ease of reverse engineering and lineage analysis. Alternatively, it is conceivable that a mapping translates multiple words into one word, in a similar way to the way in which codons of multiple nucleotides translate into one amino acid. In that case, genotype and phenotype would be of different sizes. Also, the relative positioning of genotype and phenotype does not have to be in the order of the prototype, or even fixed. The copying process could take place prior to the decoding process, and the decoding source could be the (already copied) offspring's genome, rather than the parent's genome. This would eliminate the typical 1 generation delay in mutations being expressed (as can be seen in Figure 3.11 in Chapter 3).

Another possible approach of redesigning the prototype is to decouple the segments (the phenome active segment, the phenome passive segment, and the genome) into separate components, with each component equally undergoing evolutionary process. By way of contrast, the original prototype was designed to have all the segments contiguous in a single memory. (In this sense, the design of the prototype can be said to depend on the coreworld type of system architecture.) Any design would be compatible with the aim, as long as genotype-phenotype decomposition is retained and the genotype-phenotype mapping is designed to be exposed to mutations in some manner.

As mentioned in the motivation of Section 4.7 in Chapter 4, an approach to prevent degeneration in the standard Avida is conceivable: allocate the CPU time *regardless of* the organism size or the reproduction rate, that is, allow the organisms' reproduction on a one-by-one basis (skipping infertile organisms by judging with some preset cut-off time). This approach would offset the advantage of self-copiers (which are faster), even though it may result in relatively substantial runtime as faster reproducers have to wait for slower reproducers. Since the original Avida system does not exactly have such a world mode or a command, some modification at the source code level would be required in order to realise this type of CPU time allocation.[1]

At any rate, modifying the Avida system beyond some extent may call instead for developing a totally new system. However, with such a level of modification into the Avida world's operating system, it is questionable if it can still reasonably be called Avida. Whether the research is exclusively designed for Avida or not, any radical modification of

[1] There is a configuration variable that may be useful for this purpose (`BASE_MERIT_METHOD`). When this variable is set to 0, every organism gets a constant base *merit* independent of size. Merit is a value that signifies a relative speed of an organism's CPU. Enabling this method may be an option to equalise the rate of execution, but not exactly allowing organisms of different sizes to finish executing one by one. Setting the variable `SLICING_METHOD` to 0 so as to give some large constant evenly to every organism might be similarly a useful idea, but it would require some heuristics in order to determine how large the constant should exactly be, and it can potentially be computationally costly by giving the same large amount of CPU time to shorter ones.

the system itself should be considered and applied with clear and explicit motivation.

5.3.3 Better Evolutionary Understanding

In the previous two subsections, further investigations both of analysis methods for better characterisation of whatever reproducers, and of design possibilities of von Neumann style self-reproducers, are recommended. Those lines of research will help reveal what potential the von Neumann architecture (as a reproduction mode) has, and help clarify what mutational pathways give rise to what kind of unprecedented, non-trivial reproduction modes (or, more generally, characteristics or behaviours).

Again, apart from the static, deterministic approach of designing the ancestor and the enhanced analysis, it may be worthwhile to conduct dynamic, stochastic Avida experiments in answering those questions. In such an experiment, an ancestor is seeded and run over generations for a number of times, just as in the standard Avida experiments. Mutants of the ancestor that become dominant are ones that deserve the deterministic mutation analysis or a further evolutionary observation. More generally speaking, in such an evolutionary context, Popper (1972) argues that "the hopeful behavioural monster" (as opposed to the *anatomical* monster) is not necessarily likely to be lethal, and that it can be a significant change that gives rise to a novel evolutionary pathway by creating a new niche where subsequent mutations that are otherwise detrimental are now beneficial. The findings of the current investigation suggest that interestingly there are estimated to be diverse reproduction modes among the mutationally close strains categorised as viable. Depending on the stochastic variables of the world in which they are situated, it is possible that different reproduction modes (different von Neumann style self-reproducers) may become dominant.

In order to observe a novel, interesting feature emerging or evolving, it might be necessary to focus not only on individuals separately, but also collectively. The standard Avida allows local interaction, as in replacing a neighbouring node with an offspring; it can be configured so that all the organisms are neighbours and can be replaced by any other organism's offspring (analogous to a stirred petri dish). Avida is (whether explicitly or implicitly) fairly configurable even right out of the box, so that organisms have energy exchange, sexual reproduction, and so on, which are features that involve interaction with other individuals. However, at any rate, fine-tuning the manner of (or the level of) the interaction of organisms in the arena of Avida may be effective in order to shed light on collective style of self-reproduction. As mentioned in Chapter 4, there may be a reproduction mode of collective (autocatalytic) self-reproducers, which would require to define what "collective von Neumann style self-reproduction" means.

As suggested in the existing Avida literature where self-copier organisms can evolve into those with higher complexity (e.g. higher computational ability), the reproduction mode of self-copying has significant evolvability in its own right. In studies where a fitness function plays a central role, the interaction with the environment (typically in the form of calculation of input/output numbers from/to the environment) matters. This would be more relevant when ancillary machinery is taken into account. Investigation of ancillary machinery in the initial design contributing to reproduction may be another interesting topic. However, for the next immediate steps of the current line of research, a redesigned ancestor does not necessarily have to have ancillary machinery (though it might emerge in the course of evolution), since the current research is focused more on the core evolution exhibited by, or the potential of, one design of von Neumann style self-reproducers. It is still worth revisiting the self-reproduction using von Neumann's architecture, as the potential of the von Neumann style self-reproducer in a computational world has not been fully revealed. Despite the analysis being unable to identify von Neumann style self-reproducers of particular interest (much less more arbitrary reproduction modes), a possibility of mutational pathways where a genotype-phenotype mapping evolves has not been denied as trivial.

5.3.4 For a General Understanding: Speculative Remarks

On top of those above-mentioned future directions, there are further and associated (and somewhat more abstract and higher-level) questions posed by the current research within Avida as below:

- What can be extrapolated from the findings in Avida to other coreworld type or non-coreworld type platforms?
- What conditions might facilitate or encourage the decomposition into genotype and phenotype, or a von Neumann style self-reproducer, to emerge spontaneously?
- What high-level or complex features of (self-)reproduction can be achieved as a result of evolved genotype-phenotype mapping?

Extrapolation beyond *Avida*?

Among other program-based artificial life platforms, it has been demonstrated that it is possible to implement in *Tierra* such a von Neumann style self-reproducer as the prototype presented in this thesis (Baugh & McMullin, 2012). In the course of evolution, the system seeded with such an ancestor can produce pathological constructors, which are small-sized offspring that can lead to ecological collapse by spreading out quickly and displacing the working ancestor. In Avida this would not happen as each individual organism residing in a cell cannot read and parasitically rely on another's memory content, unless considering switching on the configurable injection feature or allowing parasitism. This phenomenon may be classified in general as a type of degeneration (as in the degeneration presented in Chapter 3) induced from the design of the von Neumann style self-reproducer relative to the system structure. At least reconsidering the design and redesigning it would be worthwhile in order to effectively protect self-reproducers equipped with the architecture in the world of the system.

Regarding deterministic analysis of mutational pathways, it is arguably theoretically possible to develop such a tool for Tierra; but it is more laborious to implement a similar mechanism to the presented tool simply because the system does not originally come with a basic analysis tool to start with. Moreover, as implied, the Tierra system, without cells that separate individual organisms as in Avida, presumes the possibility of interaction of organisms in the soup by means of reading and utilising one another. Unlike Avida, where mutants that can exist in the standard Avida world are ones that are theoretically predictable and classifiable via a deterministic analysis and vice versa, it is questionable whether a deterministic characterisation of mutational pathways more effectively adds to describing the general Tierra dynamics, where any individual organisms can read one another and can give rise to new individuals, limitlessly. Thus, a deterministic analysis of mutational pathways in Tierra might have to be something that is closer to an analysis of soup configuration as a whole rather than of individual organisms.

Apart from the above platform, it is noteworthy that in the context of another platform, Stringmol (mentioned in Subsection 2.2.3 in Chapter 2 in the context of artificial chemistry), Nellis & Stepney (2011) argue about the importance of "embodying" the copying process. Although it is not appropriate to use the term "embodiment" of copying in Avida (since it would be confusing to say embodying some process in a *virtually* simulated world of *virtual* artificial-life organisms), here the concept can probably be understood as "decomposition" of the copying process. Nellis and Stepney propose "ALife organisms should not blindly use the copy operations provided by programming languages" and regard copying as "an embodied *process*, rather than as a computational result". They suggest that a decomposed copying process should be used for more evolvability within such an artificial life framework as Stringmol, instead of a stochastic copying process. This is reminiscent of the discussion on the use of the `h-copy` instruction (see Chapters 3 and 4) where the addition of the `read` and `write` instructions is justified, saying that although it did not have to necessarily be these exact instructions (`read` and `write`), it was recommended to include somewhat lower-level (hence somewhat more flexible but perhaps more primitive in a sense) instruction than the `h-copy` instruction in the instruction set. The inclusion of these was originally for the purpose of easing the process of decoding, but it might have also contributed as one of the factors to increasing evolvability. That being said, it cannot be concluded yet whether it is the case; with respect to this, further investigation on evolvability within Avida would be necessary.

Emergence of Genotype and Phenotype?

Emergence of the von Neumann style self-reproduction architecture gives another point of view to the research. Since it is through one step of a single-point mutation that the designed von Neumann style ancestor became a self-copier (as demonstrated in Chapter 3), it is one step as well, conversely, for that particular self-copier to become the von Neumann ancestor. If such a self-copier is seeded in the Avida world and hit by such a mutation as to reverse the degeneration (while one has to acknowledge that perturbations affecting a genotype-phenotype mapping may be genetically irreversible in general), this may be called *emergence* of a von Neumann self-reproducer. Of course, the probability that such a self-copier (as degenerated from the von Neumann style ancestor) configuration emerges in the first place, when starting from a standard (far shorter, or far less complex) self-copying ancestor would be vanishingly small. Still, it is interesting to question whether a self-copier can in general give rise to a von Neumann style self-reproducer, or what selection pressure might cause such a transformation. As far as a realistic scenario is concerned, it can be speculated that such emergence would occur segment by segment, rather than through randomly accumulated mutations.

Loosely, it can be conjectured that the concept of "division of labour" may be a key factor to understanding the emergence of the reproduction architecture based on genotype-phenotype mapping (if it does not emerge spontaneously). While the evolution of the decomposition of genotype and phenotype may or may not be explained by the evolutionary potential that it can open, the emergence of it is also intriguing, as the genotype-phenotype mapping is self-referential by nature (imposing a "chicken-or-the-egg" kind of causality dilemma: genotype gives rise to phenotype that in turn gives rise to the genotype). As far as the *prima facie* structure is concerned, it would be best explained by the division of labour, because the decomposition is between the (functionally) active part and the (functionally) passive part of von Neumann's architecture.

In molecular biology, as introduced in Section 2.3, the interrelationship between DNA, RNA and proteins is said to be as a result of the division of labour, and, given these three actors, it evolved as there is certain advantageous robustness in a changing milieu. That advantage resonates with the advantage derived from different levels of mechanisms such as error correction and horizontal gene transfer (such as in genetic recombination and sexual reproduction). Macroscopically, those mechanisms are believed to have evolved as being advantageous by providing stability and/or by promoting diversity in adapting to radically changing environments and once such mechanisms appear, they start evolving themselves at a new level; for example, in a general biological context, sexual reproduction can be interpreted as a mechanism that emerges as a result of dividing the labour of reproduction, and resulting sexes with different and interdependent functions then start evolving, for example, sexual dimorphism, and so forth. For example, see Misevic et al. (2006) for this line of research in the context of Avida, and Sterelny & Griffiths (2012) for more general discussion in the wider context of evolutionary biology. Thus, considering the potential advantages of dividing a particular labour and how it can occur, may be an effective approach to exploring possible potential of the von Neumann architecture (e.g., further specialising the divided labour and/or preserving the division and the specialisation).

Evolution by Means of Genotype and Phenotype?

In this thesis, an implementable design of the architecture and the analysis method of the evolutionary potential of the architecture in a particular artificial life platform, Avida, are explored and proposed. The whole thesis is a part of a research programme that may reinforce, or add to, existing evolutionary theories and studies, especially in terms of the reproduction architecture characterised by the decomposition of genotype and phenotype. Therefore, technical and social insights or applications, if any, will be mainly gained through an enriched model of evolution based on the genotype-phenotype self-reproduction architecture, examples of which include: evolution of genetic codes (symbol systems, or communication, more generally); evolution of different levels of systems (immune systems such as Stepney et al. (2005), or more generally, homeostatic systems); or evolution of brains and consciousnesses. An example of this line of research about potential of physical symbol systems is grounded in a model of paired evolutionary arenas by Fernando (2013). Systems may require a key of physical symbols to attain evolutionary open-endedness. Similar points are made by Rocha (2001), again emphasising the evolutionary potential of the von Neumann style architecture of self-reproduction.

These domains are expected to have different evolutionary dynamics as they define the notion of “individuality” and “self” in their own arenas differently, hence a different framework of reproduction modes among them (which may or may not depend on the genotype-phenotype separation). Relevant to this, technical benefits from further understanding of evolution based on genotype and phenotype may include improvements of novelty search (e.g. Lehman & Stanley, 2011), as studying that reproduction architecture involves studying diversity and non-trivial, open-ended evolutionary features.

5.4 Final Words

The motivation of the research covered in this thesis originated from the ultimate objective of observing the spontaneous evolution or any growth of complexity via the evolution of genotype-phenotype mapping. The provided result from characterising the implemented von Neumann style prototype ancestral self-reproducer is contrary to the elaboration of the mapping through evolution, but it was indeed indicated that mutational pathways of such a self-reproducer are clearly still worth exploring, with a more sophisticated method of analysing viability. Despite the current research being essentially foundational and preliminary, the characterisation of an instance of von Neumann style self-reproducer and the development of enhanced analysis tools provide a substantial contribution to the field. That is, this research helps reveal what tendency and deficiency the particular design can have, and how such a self-reproducer can be better explored within, and potentially outside of, Avida.

References

Adami, C. (1997). *Introduction to artificial life* (Corrected ed.). Berlin: Springer.

Adami, C. (2002). Ab initio modeling of ecosystems with artificial life. *Natural Resource Modeling*, *15*(1), 133–145.

Adami, C., & Brown, C. T. (1994). Evolutionary learning in the 2D artificial life system "avida". *adap-org/9405003*.

Adami, C., Ofria, C., & Collier, T. C. (2000). Evolution of biological complexity. *Proceedings of the National Academy of Sciences*, *97*(9), 4463–4468.

Anderson, C. J., & Harmon, L. (2014). Ecological and mutation-order speciation in digital organisms. *The American Naturalist*, *183*(2), 257–268.

Baugh, D., & McMullin, B. (2012). The emergence of pathological constructors when implementing the von Neumann architecture for self-reproduction in tierra. In T. Ziemke, C. Balkenius, & J. Hallam (Eds.), *From animals to animats 12* (pp. 240–248). Berlin: Springer.

Bedau, M. A. (2003). Artificial life: organization, adaptation and complexity from the bottom up. *Trends in Cognitive Sciences*, *7*(11), 505–512.

Bobrik, M., Kvasnicka, V., & Pospichal, J. (2008). Artificial chemistry and molecular Darwinian evolution of DNA/RNA-like systems I-typogenetics and chemostat. In *Computational intelligence in medical informatics* (pp. 295–336). Springer.

Boden, M. A. (Ed.). (1996). *The philosophy of artificial life.* Oxford: Oxford University Press.

Buckley, W. R. (2008). Computational ontogeny. *Biological Theory*, *3*(1), 3–6.

Burks, A. (Ed.). (1970). *Essays on cellular automata.* Champaign, Illinois: University of Illinois Press.

Codd, E. (1968). *Cellular automata.* Waltham, Massachusetts: Academic Press, Inc.

Covert, A. W., Lenski, R. E., Wilke, C. O., & Ofria, C. (2013). Experiments on the role of deleterious mutations as stepping stones in adaptive evolution. *Proceedings of the National Academy of Sciences*, *110*(34), E3171–E3178.

de Boer, F. K., & Hogeweg, P. (2012). Less can be more: RNA-Adapters may enhance coding capacity of replicators. *PLoS ONE*, *7*(1).

De Beule, J. (2011). Von Neumann's legacy for a scientific biosemiotics. *Biosemiotics*, 1–4.

De Beule, J., Hovig, E., & Benson, M. (2010). Introducing dynamics into the field of biosemiotics. *Biosemiotics*, 1–20.

Dewdney, A. K. (1984). In the game called core war hostile programs engage in a battle of bits. *Scientific American*, *250*(5), 15–19.

Dittrich, P., Ziegler, J., & Banzhaf, W. (2001). Artificial chemistries–a review. *Artificial life*, *7*(3), 225–275.

Fernando, C. (2013). Design for a Darwinian brain: Part 1. philosophy and neuroscience. *Biomimetic and Biohybrid Systems*, 71–82.

Gilbert, W. (1986). Origin of life: The RNA world. *Nature*, *319*(6055), 618–618. doi: 10.1038/319618a0

Goldsby, H. J., Serra, N., Dyer, F., Kerr, B., & Ofria, C. (2012). The evolution of temporal polyethism. *Artificial Life*, *13*, 178–185.

Gollihar, J., Levy, M., & Ellington, A. D. (2014). Many paths to the origin of life. *Science*, *343*(6168), 259–260.

Hessel, J., & Goings, S. (2013). Evolving multicellularity in digital organisms through reproductive altruism. In *Proceeding of the fifteenth annual conference companion on genetic and evolutionary computation conference companion, GECCO '13 companion* (pp. 1707–1710). ACM.

Hickinbotham, S., Clark, E., Stepney, S., Clarke, T., Nellis, A., Pay, M., & Young, P. (2011). Molecular microprograms. In *Advances in artificial life, Darwin meets von Neumann, ECAL 2009* (pp. 297–304). Springer.

Hickinbotham, S., Stepney, S., Nellis, A., Clarke, T., Clark, E., Pay, M., & Young, P. (2011). Embodied genomes and metaprogramming. In *Advances in artificial life, ECAL 2011* (pp. 334–341). MIT Press.

Hickinbotham, S. J., Clark, E., Stepney, S., Clarke, T., Nellis, A., Pay, M., & Young, P. (2010). Diversity from a monoculture: Effects of mutation-on-copy in a string-based artificial chemistry. In *Artificial life XII* (pp. 24–31). MIT Press.

Hirsch, M. W., Smale, S., & Devaney, R. L. (2004). *Differential equations, dynamical systems, and an introduction to chaos* (Vol. 60). Waltham, Massachusetts: Academic Press.

Hofstadter, D. R. (1979). *Gödel, escher, bach: an eternal golden braid.* New York: Basic Books.

Hofstadter, D. R. (1985). The genetic code: Arbitrary. In *Metamagical themas: Questing for the essence of mind and pattern* (pp. 671–699). London: Penguin Books.

Holland, J. H. (1975). *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* Michigan: University of Michigan Press.

Hutton, T. J. (2010). Codd's self-replicating computer. *Artificial Life*, *16*(2), 99–117.

Jain, S., & Krishna, S. (2006). Graph theory and the evolution of autocatalytic networks. *Handbook of Graphs and Networks: From the Genome to the Internet*.

Kauffman, S. A. (1993). *The origins of order: Self-organization and selection in evolution.* Oxford: Oxford University Press.

Knoester, D. B., Goldsby, H. J., & McKinley, P. K. (2013). Genetic variation and the evolution of consensus in digital organisms. *Evolutionary Computation, IEEE Transactions on*, *17*(3), 403–417.

Langton, C. G. (1984). Self-reproduction in cellular automata. *Physica D: Nonlinear Phenomena*, *10*(1), 135–144.

Langton, C. G. (1994). *Artificial life III: proceedings of the workshop on artificial life, held june 1992 in santa fe, new mexico.* Boston: Addison-Wesley, Advanced Book Program.

Lehman, J., & Stanley, K. O. (2011). Evolving a diversity of virtual creatures through novelty search and local competition. In *Proceedings of the 13th annual conference on genetic and evolutionary computation* (pp. 211–218). ACM.

Lenski, R. E., Ofria, C., Collier, T. C., & Adami, C. (1999). Genome complexity, robustness and genetic interactions in digital organisms. *Nature*, *400*(6745), 661–664. doi: 10.1038/23245

Lenski, R. E., Ofria, C., Pennock, R. T., & Adami, C. (2003). The evolutionary origin of complex features. *Nature*, *423*(6936), 139–144. doi: 10.1038/nature01568

Maley, C. C. (1994). The computational completeness of Ray's Tierran assembly language. In C. G. Langton (Ed.), *Artificial life III* (pp. 503–514). Boston: Addison-Wesley, Advanced Book Program.

Mange, D., Stauffer, A., Peparaolo, L., & Tempesti, G. (2004). A macroscopic view of self-replication. *Proceedings of the IEEE*, *92*(12), 1929–1945.

Maynard-Smith, J., et al. (1989). *Evolutionary genetics.* Oxford: Oxford University Press.

McKinley, P., Cheng, B. H., Ofria, C., Knoester, D., Beckmann, B., & Goldsby, H. (2008). Harnessing digital evolution. *Computer*, *41*(1), 54–63.

McMullin, B. (1993). What is a universal constructor. *Dublin City University School of Electronic Engineering Technical Report bmcm9301*.

McMullin, B. (1995). Replicators don't! *Advances in Artificial Life*, 158–169.

McMullin, B. (2000). John von Neumann and the evolutionary growth of complexity: Looking backward, looking forward. *Artificial Life*, *6*(4), 347–361. doi: 10.1162/106454600300103674

McMullin, B. (2004). Thirty years of computational autopoiesis: A review. *Artificial Life*, *10*(3), 277–295. doi: 10.1162/1064546041255548

McMullin, B. (2012). Architectures for self-reproduction: Abstractions, realisations and a research program. In C. Adami, D. M. Bryson, C. Ofria, & R. T. Pennock (Eds.), *Artificial life 13* (pp. 83–90). Cambridge, Massachusetts: MIT Press.

McMullin, B., Taylor, T., & von Kamp, A. (2001). Who needs genomes? In *Atlantic symposium on computational biology and genome information systems and technology.*

Misevic, D., Ofria, C., & Lenski, R. E. (2006). Sexual reproduction reshapes the genetic architecture of digital organisms. *Proceedings of the Royal Society B: Biological Sciences*, *273*(1585), 457.

Misevic, D., Ofria, C., & Lenski, R. E. (2010). Experiments with digital organisms on the origin and maintenance of sex in changing environments. *Journal of heredity*, *101*(suppl 1), S46–S54.

Mitchell, M. (2009). *Complexity: A guided tour.* Oxford: Oxford University Press.

Moore, E. F. (1962). Machine models of self-reproduction. *Essays on Cellular Automata*, 187–203.

Morris, H. C. (1987). Typogenetics: A logic for artificial life. In *Artificial life* (pp. 369–396). Addison-Wesley.

Nellis, A., & Stepney, S. (2011). Embodied copying for richer evolution. In *Advances in artificial life, ECAL 2011* (pp. 597–604). MIT Press.

Nobili, R., Pesavento, U., Scientifico, L., & Nievo, I. (1994). John von Neumann's automata revisited. *Artificial Worlds and Urban Studies*.

Nowak, M. A. (2006). *Evolutionary dynamics: exploring the equations of life.* Cambridge, Massachusetts: Harvard University Press.

Ofria, C., Adami, C., & Collier, T. C. (2002). Design of evolvable computer languages. *Evolutionary Computation, IEEE Transactions on*, *6*(4), 420–424.

Ofria, C., Adami, C., Collier, T. C., & Hsu, G. K. (1999). Evolution of differentiated expression patterns in digital organisms. In D. Floreano, J.-D. Nicoud, & F. Mondada (Eds.), *Advances in artificial life* (Vol. 1674, pp. 129–138). Berlin: Springer.

Ofria, C., & Wilke, C. O. (2004). Avida: A software platform for research in computational evolutionary biology. *Artificial Life*, *10*(2), 191–229. doi: 10.1162/106454604773563612

Pargellis, A. N. (1996). The evolution of self-replicating computer organisms. *Physica D: Nonlinear Phenomena*, *98*(1), 111–127. doi: 10.1016/0167-2789(96)00089-9

Pargellis, A. N. (2001). Digital life behavior in the Amoeba world. *Artificial Life*, *7*(1), 63–75. doi: 10.1162/106454601300328025

Pargellis, A. N. (2003). Self-organizing genetic codes and the emergence of digital life. *Complexity*, *8*(4), 69–78. doi: 10.1002/cplx.10095

Pattee, H. (1982). Cell psychology: an evolutionary approach to the symbol-matter problem. *Cognition and Brain Theory*, *5*(4), 325–341.

Pattee, H. (1995). Evolving self-reference: matter, symbols, and semantic closure. *Communication and Cognition-Artificial Intelligence*, *12*(1-2), 9–27.

Pattee, H. (2005). The physics and metaphysics of biosemiotics. *Journal of Biosemiotics*, *1*(1), 281–301.

Pesavento, U. (1995). An implementation of von Neumann's self-reproducing machine. *Artificial Life*, *2*(4), 337–354.

Popper, K. R. (1972). *Objective knowledge: An evolutionary approach* (Revised ed.). USA: Oxford University Press.

Rasmussen, S., Knudsen, C., Feldberg, R., & Hindsholm, M. (1990). The coreworld: Emergence and evolution of cooperative structures in a computational chemistry. *Physica D: Nonlinear Phenomena*, *42*(1–3), 111–134.

Ray, T. (1994). An evolutionary approach to synthetic biology: Zen and the art of creating life. *Artificial Life*, *1*(1/2), 179–209.

Rocha, L. M. (2001). Evolution with material symbol systems. *Biosystems*, *60*(1), 95–121.

Schrödinger, E. (1944). *What is life?* Cambridge University Press.

Sims, K. (1994). Evolving 3D morphology and behavior by competition. *Artificial life*, *1*(4), 353–372.

Sipper, M. (1998). Fifty years of research on self-replication: An overview. *Artificial Life*, *4*(3), 237–257.

Stepney, S., Smith, R. E., Timmis, J., Tyrrell, A. M., Neal, M. J., & Hone, A. N. W. (2005). Conceptual frameworks for artificial immune systems. *International Journal of Unconventional Computing*, *1*(3), 315–338.

Sterelny, K., & Griffiths, P. E. (2012). *Sex and death: An introduction to philosophy of biology.* University of Chicago press.

Szathmáry, E., & Maynard Smith, J. (1997). From replicators to reproducers: the first major transitions leading to life. *Journal of Theoretical Biology*, *187*(4), 555–571.

Takeuchi, N., Hogeweg, P., & Koonin, E. V. (2011). On the origin of DNA genomes: Evolution of the division of labor between template and catalyst in model replicator systems. *PLoS Computational Biology*, *7*(3).

Taylor, T. (2013). Evolution in virtual worlds. *The Oxford Handbook of Virtuality. Oxford University Press.*.

Taylor, T. J. (1999). From artificial evolution to artificial life. *Unpublished doctoral dissertation, Division of Informatics, University of Edinburgh, UK*.

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. A correction. *Proceedings of the London Mathematical Society*, *2*(1), 544.

von Neumann, J. (1951). The general and logical theory of automata. *Cerebral mechanisms in behavior*, 1–41.

von Neumann, J. (1966). *Theory of self-reproducing automata* (A. Burks, Ed.). Champaign, Illinois: University of Illinois Press.

Wilke, C. O., Wang, J. L., Ofria, C., Lenski, R. E., & Adami, C. (2001). Evolution of digital organisms at high mutation rates leads to survival of the flattest. *Nature*, *412*(6844), 331–333.

Williams, L. R. (2011). Artificial cells as reified quines. In *Advances in artificial life, ECAL 2011.* Cambridge, Massachusetts: MIT Press.

Williams, L. R. (2013). Evolution of tail-call optimization in a population of self-hosting compilers. In *Advances in artificial life, ECAL 2013* (Vol. 12, pp. 86–93). Cambridge, Massachusetts: MIT Press.

Williams, L. R. (2014). Self-replicating distributed virtual machines. In *The fourteenth conference on the synthesis and simulation of living systems, ALIFE 14* (Vol. 14, pp. 711–718). Cambridge, Massachusetts: MIT Press.

Wittgenstein, L. (1953). *Philosophical investigations* (G. Anscombe & R. Rhees, Eds.). Oxford.

Zykov, V., Mytilinaios, E., Adams, B., & Lipson, H. (2005). Self-reproducing machines. *Nature*, *435*(7038), 163–164.

Appendix A

Phenome Design and Mechanism

The mechanism of the hand-designed phenome segment of the prototype is described line by line. For legibility, the description uses some abbreviations for the values stored in the stack: “G” for genome start address; “GL” for remaining length; “LT” for lookup table start address; “l” for label size; “WL” for whole length; “c” for constant zero for comparison; “d” for relative source address in parent genome; “D” for relative destination address in offspring (all of which are explained in context in the code listing below). For the detailed step-by-step state transition of the prototype, refer to the actual (“old”-format) trace file provided at: `http://alife.rince.ie/th_phd_2014/`. Refer to the appendix of the article by Ofria & Wilke (2004) for more comprehensive explanation on the behaviours of instructions.

Listing A.1: Decode Preparation

```
h-alloc  # Allocate space for child. # AX:Whole Length (WL)
h-search # Locate the start of Lookup Table (LT). # BX:distance to LT,CX:labelsize (l)
nop-A # Label Alpha.Looks for nop-B,nop-C.
nop-B #
push  # Push AX:WL.(Stack-0:WL,c,0,..)(c==0:for comparison)
nop-A #
push  # Push CX:l.(Stack-0:l,WL,c,0,..)
nop-C #
mov-head # Move Read Head to LT.
nop-B # (Read Head)
get-head # Get CX:LT
nop-B # (of Read Head)
push  # Push CX:LT.(Stack-0:LT,l,WL,c,0,..)
nop-C #
h-search # Locate the start of Genome. # BX:d to G,CX:labelsize
nop-A # Label Beta.Looks for nop-B,nop-A.
nop-C #
mov-head # Move Read Head to G.
nop-B #
get-head # Get CX:G.AX:WL,BX:d to G.
nop-B # (of Read Head)
swap  # AX:d to G,BX:WL,CX:G
nop-A # between AX and BX
sub   # BX:GL (=BX-CX=WL-G) (="Length to Go before Genome")
push  # Push BX:GL.(Stack-0:GL,LT,l,WL,c,0,..)
swap  # AX:d to G,BX:G,CX:GL
push  # Push BX:G.(Stack-0:G,GL,LT,l,WL,c,0,..)
h-search # Set Flow Head to mark the start of the loop. # BX:0,CX:0
```

Listing A.2: Decode Loop

```
pop   # Get G and d,to read word into CX. Pop BX:G.(Stack-0:GL,LT,l,WL,c,0,..)
push  # Push BX:G.(Stack-0:G,GL,LT,l,WL,c,0,..)
swap-stk # Stack-1 acitive.(Stack-1:d,D,0,..) (D="Destination in daughter" used later)
pop   # Pop CX:d.(Stack-1:D,0,..) (=distance from G)
nop-C #
push  # Push CX:d.(Stack-1:d,D,0,..)
nop-C #
add   # BX:G+d,CX:d
read  # Read one word at BX:G+d,then CX:word.
pop   # BX:d.(Stack-1:D,0,..)
inc   # BX:d++
push  # BX:d.(Stack-1:d,D,0,..)
swap-stk # Get LT at BX so as to read word’ (word translated) into CX # Stack-0 active
pop   # Pop BX:G.(Stack-0:GL,LT,l,WL,c,0,..)
swap-stk # Stack-1 active
push  # Push BX:G.(Stack-1:G,d,D,0,..)
swap-stk # Stack-0 active
pop   # BX:GL.(Stack-0:LT,l,WL,c,0,..)
swap-stk # Stack-1 active
push  # Push BX:GL.(Stack-1:GL,G,d,D,0 ..)
swap-stk # Stack-0 active
pop   # Pop BX:LT.(Stack-0:l,WL,c,0,..)
push  # Push BX:LT.(Stack-0:LT,l,WL,c,0,..)
add   # BX=LT+word,CX:word
read  # Read one word at BX:LT+word,then CX:word’
swap-stk # Get D at BX. (Preprocess for write) # Stack-1 active
pop   # Pop BX:GL.(Stack-1:G,d,D,0,..)
swap-stk # Stack-0 active
push  # Push BX:GL.(Stack-0:GL,LT,l,WL,c,0,..)
swap-stk # Stack-1 active
pop   # Pop BX:G.(Stack-1:d,D,0,..)
swap-stk # Stack-0 active
push  # Push BX:G.(Stack-0:G,GL,LT,l,WL,c,0,..)
swap-stk # Stack-1 active
pop   # Pop BX:d.(Stack-1:D,0,..)
swap-stk # Stack-0 active
push  # Push BX:d.(Stack-0:d,G,GL,LT,l,WL,c,0,..)
swap-stk # Stack-1 active
pop   # Pop BX:D.(Stack-1:0,..)
swap-stk # Get WL at AX. (Preprocess for write). # Stack-0 active
pop   # Pop AX:d.(Stack-0:G,GL,LT,l,WL,c,0,..)
nop-A #
swap-stk # Stack-1 active
push  # Push AX:d.(Stack-1:d,0,..)
nop-A #
swap-stk # Stack-0 active
pop   # Pop AX:G.(Stack-0:GL,LT,l,WL,c,0,..)
nop-A #
swap-stk # Stack-1 active
push  # Push AX:G.(Stack-1:G,d,0,..)
nop-A #
swap-stk # Stack-0 active
pop   # Pop AX:GL.(Stack-0:LT,l,WL,c,0,..)
nop-A #
swap-stk # Stack-1 active
push  # Push AX:GL.(Stack-1:GL,G,d,0,..)
nop-A #
swap-stk # Stack-0 active
pop   # Pop AX:LT.(Stack-0:l,WL,c,0,..)
nop-A #
```

```
swap-stk # Stack-1 active
push  # Push AX:LT.(Stack-1:LT,GL,G,d,0,..)
nop-A #
swap-stk # Stack-0 active
pop   # Pop AX:l.(Stack-0:WL,c,0,..)
nop-A #
swap-stk # Stack-1 active
push  # Push AX:l.(Stack-1:l,LT,GL,G,d,0,..)
nop-A #
swap-stk # Stack-0 active
pop   # Pop AX:WL.(Stack-0:c,0,..)
nop-A #
push  # Push AX:WL.(Stack-0:WL,c,0,..)
nop-A #
write # Write CX:word' at AX+BX:WL+D.
inc   # D++.(D="Destination in daughter")
swap-stk # Stack-1 active
pop   # Pop CX:l.(Stack-1:LT,GL,G,d,0,..)
nop-C #
swap-stk # Stack-0 active
push  # Push CX:l.(Stack-0:l,WL,c,0,..)
nop-C #
swap-stk # Stack-1 active
pop   # Pop CX:LT.(Stack-1:GL,G,d,0,..)
nop-C #
swap-stk # Stack-0 active
push  # Push CX:LT.(Stack-0:LT,l,WL,c,0,..)
nop-C #
swap-stk # Stack-1 active
pop   # Pop CX:GL.(Stack-1:G,d,0,..)
nop-C #
swap-stk # Stack-0 active
push  # Push CX:GL.(Stack-0:GL,LT,l,WL,c,0,..)
nop-C #
swap-stk # Stack-1 active
pop   # Pop CX:G.(Stack-1:d,0,..)
nop-C #
swap-stk # Stack-0 active
push  # Push CX:G.(Stack-0:G,GL,LT,l,WL,c,0,..)
nop-C #
swap-stk # Stack-1 active
pop   # Pop CX:d.(Stack-1:0,..)
nop-C #
push  # Push BX:D.(Stack-1:D,0,..)
push  # Push CX:d.(Stack-1:d,D,0,..)
nop-C #
swap-stk # Get c at CX. (Preprocess for comparison) # Stack-0 active
pop   # Pop CX:G.(Stack-0:GL,LT,l,WL,c,0,..)
nop-C #
swap-stk # Stack-1 active
push  # Push CX:G.(Stack-1:G,d,D,0,..)
nop-C #
swap-stk # Stack-0 active
pop   # Pop CX:GL.(Stack-0:LT,l,WL,c,0,..)
nop-C #
swap-stk # Stack-1 active
push  # Push CX:GL.(Stack-1:GL,G,d,D,0,..)
nop-C #
swap-stk # Stack-0 active
pop   # Pop CX:LT.(Stack-0:l,WL,c,0,..)
nop-C #
swap-stk # Stack-1 active
push  # Push CX:LT.(Stack-1:LT,GL,G,d,D,0,..)
```

```
nop-C #
swap-stk # Stack-0 active
pop   # Pop CX:l.(Stack-0:WL,c,0,..)
nop-C #
swap-stk # Stack-1 active
push  # Push CX:l.(Stack-1:l,LT,GL,G,d,D,0,..)
nop-C #
swap-stk # Stack-0 active
pop   # Pop CX:WL.(Stack-0:c,0,..)
nop-C #
swap-stk # Stack-1 active
push  # Push CX:WL.(Stack-1:WL,l,LT,GL,G,d,D,0,..)
nop-C #
swap-stk # Stack-0 active
pop   # Pop CX:c.(Stack-0:0,..)
nop-C #
push  # Push CX:c.(Stack-0:c,0,..)
nop-C #
swap-stk # Get GL at BX (Preprocess for comparison) # Stack-1 active
pop   # Pop BX:WL.(Stack-1:l,LT,GL,G,d,D,0,..)
swap-stk # Stack-0 active
push  # Push BX:WL.(Stack-0:WL,c,0,..)
swap-stk # Stack-1 active
pop   # Pop BX:l.(Stack-1:LT,GL,G,d,D,0,..)
swap-stk # Stack-0 active
push  # Push BX:l.(Stack-0:l,WL,c,0,..)
swap-stk # Stack-1 active
pop   # Pop BX:LT.(Stack-1:GL,G,d,D,0,..)
swap-stk # Stack-0 active
push  # Push BX:LT.(Stack-0:LT,l,WL,c,0,..)
swap-stk # Stack-1 active
pop   # Pop BX:GL.(Stack-1:G,d,D,0,..)
dec   # BX:GL--.(Decrement as one word is read and written)
swap-stk # Stack-0 active
push  # Push BX:GL.(Stack-0:GL,LT,l,WL,c,0,..)
swap-stk # Prepare for the next loop # Stack-1 active
pop   # Pop AX:G.(Stack-1:d,D,0,..)
nop-A #
swap-stk # Stack-0 active
push  # Push AX:GL.(Stack-0:G,GL,LT,l,WL,c,0,..)
nop-A #
if-n-equ # Branch. # Compare BX:GL to CX:c.Do the next and loop back if BX not= CX
    .(while GL>0).
mov-head # If BX=CX.(when GL=0),onto the next phase. # AX:GL,BX:GL,CX:c.
```

Listing A.3: Copy Preparation

```
pop   # Get a new GL by doing WL-G. # CX:G.(Stack-0:GL,LT,l,WL,c,0,..)
nop-C #
swap-stk # Stack-1 active
push  # CX:G. (Stack-1:G,d,D,0,..)
nop-C #
swap-stk # Stack-0 active
pop   # BX:GL.(Stack-0:LT,l,WL,c,0,..)
pop   # AX:LT.(Stack-0:l,WL,c,0,..)
nop-A #
swap-stk # Stack-1 active
push  # AX:LT.(Stack-1:LT,G,d,D,0,..)
nop-A #
swap-stk # Stack-0 active
pop   # AX:l.(Stack-0:WL,c,0,..)
nop-A #
swap-stk # Stack-1 active
```

```
push  # AX:1.(Stack-1:1,LT,G,d,D,0,..)
nop-A #
swap-stk # Stack-0 active
pop   # AX:WL.(Stack-0:c,0,..)
nop-A #
push  # AX:WL.(Stack-0:WL,c,0,..)
nop-A #
swap  # AX:GL,BX:WL,CX:G
nop-A #
sub   # AX:WL-G (=GL)
nop-A #
swap-stk # Stack-1 active
pop   # BX:1.(Stack-1:LT,G,d,D,0,..)
swap-stk # Stack-0 active
push  # BX:1.(Stack-0:1,WL,c,0,..)
swap-stk # Stack-1 active
pop   # BX:LT.(Stack-1:G,d,D,0,..)
push  # AX:GL. (Stack-1:GL,G,d,D,0,..)
nop-A #
push  # BX:LT.(Stack-1:LT,GL,G,d,D,0,..)
set-flow # Set Read Head at G.AX:GL,BX:LT,CX:G # Set the Flow Head at CX:G.
mov-head # Move the Read Head to G.
nop-B # Read Head
swap-stk # Set Write Head at WL+G. # Stack-0 active
pop   # BX:1.(Stack-0:WL,c,0,..)
swap-stk # Stack-1 active
push  # BX:1.(Stack-1:1,LT,GL,G,d,D,0,..)
swap-stk # Stack-0 active
pop   # BX:WL.(Stack-0:c,0,..)
push  # BX:WL.(Stack-0:WL,c,0,..)
swap  # AX:GL,BX:G,CX:WL
add   # CX:WL+G
nop-C # (Sum into CX)
set-flow # Set the Flow Head at CX:WL+G.
mov-head # Move the Write Head to WL+G where Flow Head is at.
nop-C # (Write Head)
```

Listing A.4: Copy Loop

```
h-search # Mark the start of the loop.AX:GL(debris),BX:0,CX:0. # *(Stack-0:WL,c,0,..)(Stack-1:1,LT,GL,G,d,D,0,..)
h-copy   # Copy a word from Read Head to Write Head; inc both.
swap-stk # Get GL into BX and decrement. # Stack-1 active
pop   # BX:1.(Stack-1:LT,GL,G,d,D,0,..)
swap-stk # Stack-0 active
push  # BX:1.(Stack-0:1,WL,c,0,..)
swap-stk # Stack-1 active
pop   # BX:LT.(Stack-1:GL,G,d,D,0,..)
swap-stk # Stack-0 active
push  # BX:LT.(Stack-0:LT,1,WL,c,0,..)
swap-stk # Stack-1 active
pop   # BX:GL.(Stack-1:G,d,D,0,..)
dec   # BX:GL-- as a counter
push  # BX:GL.(Stack-1:GL,G,d,D,0,..)
swap-stk # Get c into CX. # Stack-0 active
pop   # CX:LT.(Stack-0:1,WL,c,0,..)
nop-C #
swap-stk # Stack-1 active
push  # CX:LT.(Stack-1:LT,GL,G,d,D,0,..)
nop-C
swap-stk # Stack-0 active
pop   # CX:1.(Stack-0:WL,c,0,..)
nop-C #
```

```
swap-stk # Stack-1 active
push   # CX:l.(Stack-1:l,LT,GL,G,d,D,0,..)
nop-C #
swap-stk # Stack-0 active
pop    # CX:WL.(Stack-0:c,0,..)
nop-C #
swap-stk # Stack-1 active
push   # CX:WL.(Stack-1:WL,l,LT,GL,G,d,D,0,..)
nop-C #
swap-stk # Stack-0 active
pop    # CX:c.(Stack-0:0,..)
nop-C #
push   # CX:c.(Stack-0:c,0,..)
nop-C #
swap-stk # Prepare for the next loop. # Stack-1 active
pop    # AX:WL.(Stack-1:l,LT,GL,G,d,D,0,..)
nop-A #
swap-stk #
push   # AX:WL.(Stack-0:WL,c,0,..)
nop-A #
if-n-equ # Branch. # Compare BX:GL to CX:c.Do the next if BX not= CX.(while GL>0).
    Otherwise skip.
mov-head # Loop back.
h-divide # If BX=CX.(when GL=0),divide.
nop-B # Label complemental Alpha
nop-C # No other nop-B - nop-C must exist before this.
```

Listing A.5: Lookup Table

```
27 # 0 to 27/write
26 # 1 to 26/read
25 # 2 to 25/h-search
24 # 3 to 24/IO
23 # 4 to 23/h-divide
22 # 5 to 22/h-alloc
21 # 6 to 21/h-copy
20 # 7 to 20/nand
19 # 8 to 19/sub
18 # 9 to 18/add
17 # 10 to 17/swap
16 # 11 to 16/swap-stk
15 # 12 to 15/pop
14 # 13 to 14/push
13 # 14 to 13/dec
12 # 15 to 12/inc
11 # 16 to 11/shift-l
10 # 17 to 10/shift-r
9  # 18 to 9/set-flow
8  # 19 to 8/get-head
7  # 20 to 7/jmp-head
6  # 21 to 6/mov-head
5  # 22 to 5/if-label
4  # 23 to 4/if-less
3  # 24 to 3/if-n-equ
2  # 25 to 2/nop-C
1  # 26 to 1/nop-B # No other nop-B - nop-A label must exist before this.
0  # 27 to 0/nop-A # Also used as Label complemental Beta.
```

Section 4.7 in Chapter 4 introduced two variations of the prototype, redesigned without using the h-copy instruction. Below, the code snippets of these two h-copy-free versions of the prototype are shown. For hcf-28_org-0, which uses the same 28-instruction set, the decode segment and the lookup table are omitted for they are the same as the prototype's.

For hcf-27_org-0, which uses the 27-instruction set excluding the h-copy instruction, the phenome active part (the decode and the copy segments) is omitted for it is the same as hcf-28_org-0.

Listing A.6: Copy Preparation of hcf-28_org-0

```
pop      # Get a new GL by doing WL-G. # CX:G.(Stack-0:GL,LT,l,WL,c,0,..)
nop-C    #
swap-stk # Stack-1 active
push     # CX:G. (Stack-1:G,d,D,0,..)
nop-C    #
swap-stk # Stack-0 active
pop      # BX:GL.(Stack-0:LT,l,WL,c,0,..)
pop      # AX:LT.(Stack-0:l,WL,c,0,..)
nop-A    #
swap-stk # Stack-1 active
push     # AX:LT.(Stack-1:LT,G,d,D,0,..)
nop-A    #
swap-stk # Stack-0 active
pop      # AX:l.(Stack-0:WL,c,0,..)
nop-A    #
swap-stk # Stack-1 active
push     # AX:l.(Stack-1:l,LT,G,d,D,0,..)
nop-A    #
swap-stk # Stack-0 active
pop      # AX:WL.(Stack-0:c,0,..)
nop-A    #
push     # AX:WL.(Stack-0:WL,c,0,..)
nop-A    #
swap     # AX:GL,BX:WL,CX:G
nop-A    #
sub      # AX:WL-G (=GL)
nop-A    #
swap-stk # Stack-1 active
pop      # BX:l.(Stack-1:LT,G,d,D,0,..)
swap-stk # Stack-0 active
push     # BX:l.(Stack-0:l,WL,c,0,..)
swap-stk # Stack-1 active
pop      # BX:LT.(Stack-1:G,d,D,0,..)
push     # AX:GL. (Stack-1:GL,G,d,D,0,..)
nop-A    #
push     # BX:LT.(Stack-1:LT,GL,G,d,D,0,..)
swap-stk # Stack-0 active
pop      # BX:l.(Stack-0:WL,c,0,..)
swap-stk # Stack-1 active
push     # BX:l.(Stack-1:l,LT,GL,G,d,D,0,..)
swap-stk # Stack-0 active
pop      # BX:WL.(Stack-0:c,0,..)
swap-stk # Stack-1 active
push     # BX:WL.(Stack-1:WL,l,LT,GL,G,d,D,0,..)
swap     # AX:WL,BX:GL,CX:G
nop-A    #
swap     # AX:WL,BX:G,CX:GL
push     # BX:G.(Stack-1:G,WL,l,LT,GL,G,d,D,0,..)
```

Listing A.7: Copy Loop of hcf-28_org-0

```
get-head # # CX:here(242)
set-flow # Flow Head to CX:242
pop      # BX:G.(Stack-1:WL,l,LT,GL,G,d,D,0,..)
read     # Read one word at BX:G into CX:word
write    # Write CX:word at AX+BX:WL+G
inc      # BX:G++ as a relative address (say "CD" for "copy destination")
```

```
swap-stk # Stack-0 active
push  # BX: CD.(Stack-0:cD,c,0,..)
swap-stk # Stack-1 active
pop   # BX:WL.(Stack-1:l,LT,GL,G,d,D,0,..)
swap-stk # Stack-0 active
push  # BX:WL.(Stack-0:WL,CD,c,0,..)
swap-stk # Stack-1 active
pop   # BX:l.(Stack-1:LT,GL,G,d,D,0,..)
swap-stk # Stack-0 active
push  # BX:l.(Stack-0:l,WL,CD,c,0,..)
swap-stk # Stack-1 active
pop   # BX:LT.(Stack-1:GL,G,d,D,0,..)
swap-stk # Stack-0 active
push  # BX:LT.(Stack-0:LT,l,WL,CD,c,0,..)
swap-stk # Stack-1 active
pop   # BX:GL.(Stack-1:G,d,D,0,..)
dec   # BX:GL-- as a counter
push  # BX:GL.(Stack-1:GL,G,d,D,0,..)
swap-stk # # Stack-0 active
pop   # CX:LT.(Stack-0:l,WL,cD,c,0,..)
nop-C #
swap-stk # Stack-1 active
push  # CX:LT.(Stack-1:LT,GL,G,d,D,0,..)
nop-C #
swap-stk # Stack-0 active
pop   # CX:l.(Stack-0:WL,CD,c,0,..)
nop-C #
swap-stk # Stack-1 active
push  # CX:l.(Stack-1:l,LT,GL,G,d,D,0,..)
nop-C #
swap-stk # Stack-0 active
pop   # CX:WL.(Stack-0:CD,c,0,..)
nop-C #
swap-stk # Stack-1 active
push  # CX:WL.(Stack-1:WL,l,LT,GL,G,d,D,0,..)
nop-C #
swap-stk # Stack-0 active
pop   # CX:CD.(Stack-0:c,0,..)
nop-C #
swap-stk # Stack-1 active
push  # CX:CD.(Stack-1:CD,WL,l,LT,GL,G,d,D,0,..)
nop-C #
swap-stk # # Stack-0 active
pop   # CX:c.(Stack-0:0,..)
nop-C #
push  # CX:c.(Stack-0:c,0,..)
nop-C #
swap-stk # Stack-1 active
if-n-equ # Branching. # Compare BX:GL to CX:c.Do the next if BX not = CX.(while GL >0).Otherwise skip.
mov-head # Loop back.
set-flow # Proceed to divide # Flow Head to AX:WL
nop-A #
mov-head # Read Head to Flow Head at WL
nop-B # (move Read Head)
pop   # BX:CD.(Stack-1:WL,l,LT,GL,G,d,D,0,..)
swap  # AX:c,BX:CD,CX:WL
nop-C # (between CX and AX)
add   # CX:WL+CD.(CX=BX+CX)
nop-C # (output into CX)
set-flow # Flow Head to CX:WL+CD.
mov-head # Write Head to WL where Flow Head is at.
nop-C # (move Write Head)
```

```
h-divide # Divide at Read and Write Heads.
nop-B # Label complemental Alpha
nop-C # No other nop-B - nop-C must exist before this.
```

Listing A.8: Lookup Table of hcf-27_org-0

```
0 # 0 to 26/write
1 # 1 to 25/read
2 # 2 to 24/h-search
3 # 3 to 23/IO
4 # 4 to 22/h-divide
5 # 5 to 21/h-alloc # no h-copy exists
6 # 6 to 20/nand
7 # 7 to 19/sub
8 # 8 to 18/add
9 # 9 to 17/swap
10 # 10 to 16/swap-stk
11 # 11 to 15/pop
12 # 12 to 14/push
13 # 13 to 13/dec
14 # 14 to 12/inc
15 # 15 to 11/shift-l
16 # 16 to 10/shift-r
17 # 17 to 9/set-flow
18 # 18 to 8/get-head
19 # 19 to 7/jmp-head
20 # 20 to 6/mov-head
21 # 21 to 5/if-label
22 # 22 to 4/if-less
23 # 23 to 3/if-n-equ
24 # 24 to 2/nop-C
25 # 25 to 1/nop-B # No other nop-B - nop-A label must exist before this.
26 # 26 to 0/nop-A # Also used as Label complemental Beta.
```

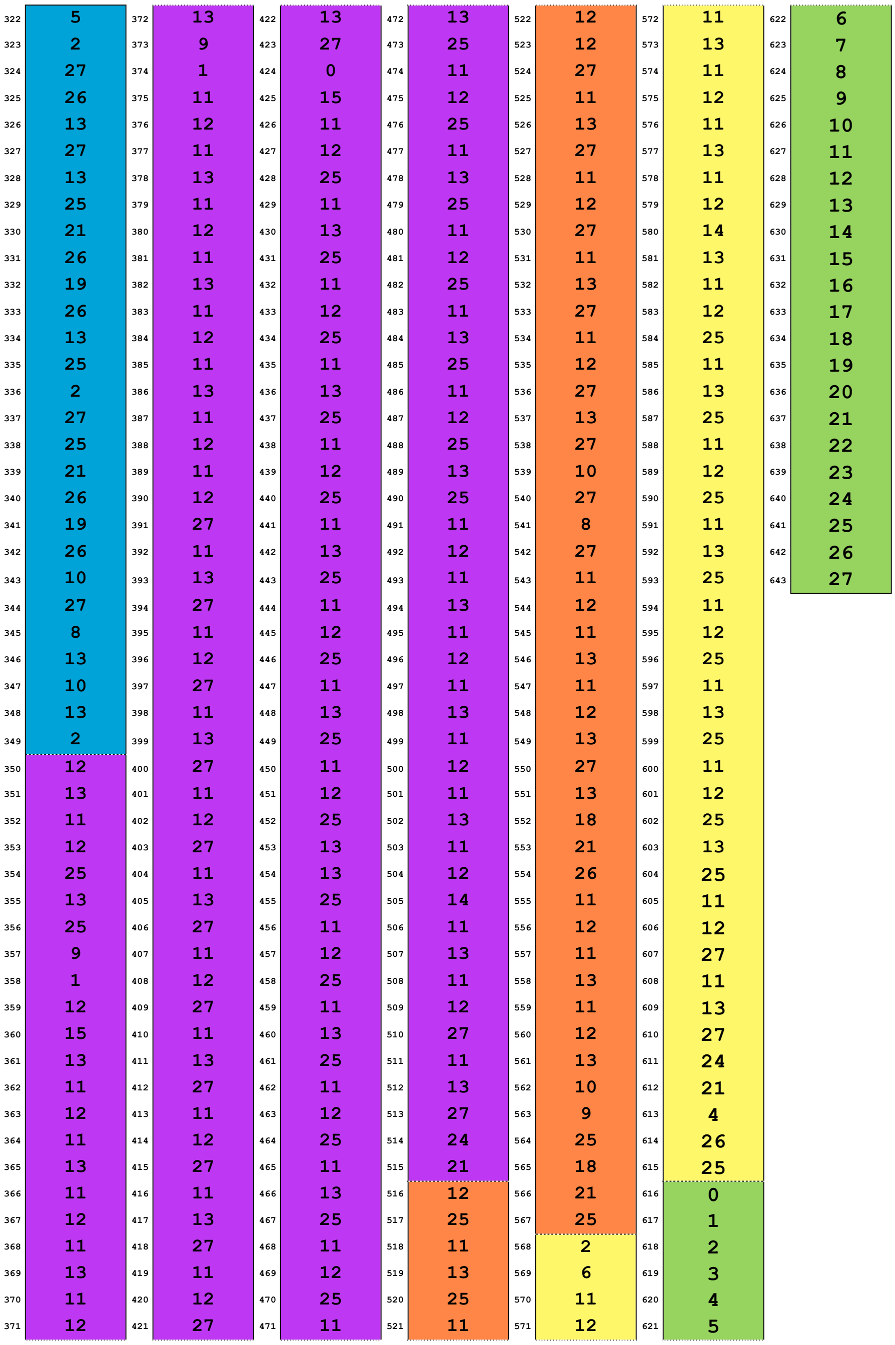
Appendix B

Selected Memory Images

The whole memory image (or code) of the prototype is presented. The memory image of the self-copying mutant degenerated from the prototype reported in Section 3.4 is presented afterwards.

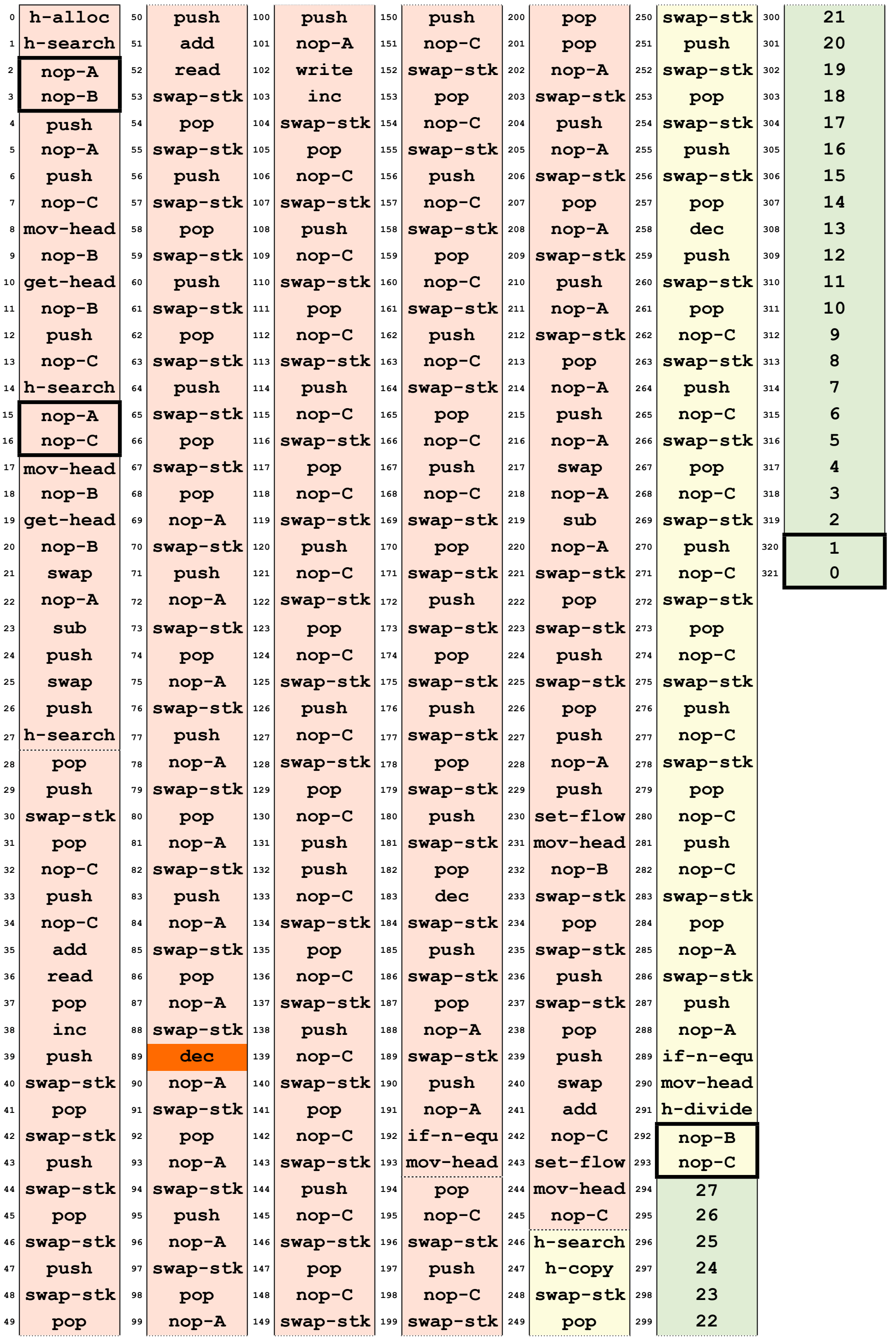
```
0   h-alloc
1   h-search
2   nop-A
3   nop-B
4   push
5   nop-A
6   push
7   nop-C
8   mov-head
9   nop-B
10  get-head
11  nop-B
12  push
13  nop-C
14  h-search
15  nop-A
16  nop-C
17  mov-head
18  nop-B
19  get-head
20  nop-B
21  swap
22  nop-A
23  sub
24  push
25  swap
26  push
27  h-search
28  pop
29  push
30  swap-stk
31  pop
32  nop-C
33  push
34  nop-C
35  add
36  read
37  pop
38  inc
39  push
40  swap-stk
41  pop
42  swap-stk
43  push
44  swap-stk
45  pop
46  swap-stk
47  push
48  swap-stk
49  pop
50  push
51  add
52  read
53  swap-stk
54  pop
55  swap-stk
56  push
57  swap-stk
58  pop
59  swap-stk
60  push
61  swap-stk
62  pop
63  swap-stk
64  push
65  swap-stk
66  pop
67  swap-stk
68  pop
69  nop-A
70  swap-stk
71  push
72  nop-A
73  swap-stk
74  pop
75  nop-A
76  swap-stk
77  push
78  nop-A
79  swap-stk
80  pop
81  nop-A
82  swap-stk
83  push
84  nop-A
85  swap-stk
86  pop
87  nop-A
88  swap-stk
89  push
90  nop-A
91  swap-stk
92  pop
93  nop-A
94  swap-stk
95  push
96  nop-A
97  swap-stk
98  pop
99  nop-A
100 push
101 nop-A
102 write
103 inc
104 swap-stk
105 pop
106 nop-C
107 swap-stk
108 push
109 nop-C
110 swap-stk
111 pop
112 nop-C
113 swap-stk
114 push
115 nop-C
116 swap-stk
117 pop
118 nop-C
119 swap-stk
120 push
121 nop-C
122 swap-stk
123 pop
124 nop-C
125 swap-stk
126 push
127 nop-C
128 swap-stk
129 pop
130 nop-C
131 push
132 push
133 nop-C
134 swap-stk
135 pop
136 nop-C
137 swap-stk
138 push
139 nop-C
140 swap-stk
141 pop
142 nop-C
143 swap-stk
144 push
145 nop-C
146 swap-stk
147 pop
148 nop-C
149 swap-stk
150 push
151 nop-C
152 swap-stk
153 pop
154 nop-C
155 swap-stk
156 push
157 nop-C
158 swap-stk
159 pop
160 nop-C
161 swap-stk
162 push
163 nop-C
164 swap-stk
165 pop
166 nop-C
167 push
168 nop-C
169 swap-stk
170 pop
171 swap-stk
172 push
173 swap-stk
174 pop
175 swap-stk
176 push
177 swap-stk
178 pop
179 swap-stk
180 push
181 swap-stk
182 pop
183 dec
184 swap-stk
185 push
186 swap-stk
187 pop
188 nop-A
189 swap-stk
190 push
191 nop-A
192 if-n-equ
193 mov-head
194 pop
195 nop-C
196 swap-stk
197 push
198 nop-C
199 swap-stk
200 pop
201 pop
202 nop-A
203 swap-stk
204 push
205 nop-A
206 swap-stk
207 pop
208 nop-A
209 swap-stk
210 push
211 nop-A
212 swap-stk
213 pop
214 nop-A
215 push
216 nop-A
217 swap
218 nop-A
219 sub
220 nop-A
221 swap-stk
222 pop
223 swap-stk
224 push
225 swap-stk
226 pop
227 push
228 nop-A
229 push
230 set-flow
231 mov-head
232 nop-B
233 swap-stk
234 pop
235 swap-stk
236 push
237 swap-stk
238 pop
239 push
240 swap
241 add
242 nop-C
243 set-flow
244 mov-head
245 nop-C
246 h-search
247 h-copy
248 swap-stk
249 pop
250 swap-stk
251 push
252 swap-stk
253 pop
254 swap-stk
255 push
256 swap-stk
257 pop
258 dec
259 push
260 swap-stk
261 pop
262 nop-C
263 swap-stk
264 push
265 nop-C
266 swap-stk
267 pop
268 nop-C
269 swap-stk
270 push
271 nop-C
272 swap-stk
273 pop
274 nop-C
275 swap-stk
276 push
277 nop-C
278 swap-stk
279 pop
280 nop-C
281 push
282 nop-C
283 swap-stk
284 pop
285 nop-A
286 swap-stk
287 push
288 nop-A
289 if-n-equ
290 mov-head
291 h-divide
292 nop-B
293 nop-C
294 27
295 26
296 25
297 24
298 23
299 22
300 21
301 20
302 19
303 18
304 17
305 16
306 15
307 14
308 13
309 12
310 11
311 10
312 9
313 8
314 7
315 6
316 5
317 4
318 3
319 2
320 1
321 0
```

Figure B.1: Prototype Phenome Image



322	5	372	13	422	13	472	13	522	12	572	11	622	6
323	2	373	9	423	27	473	25	523	12	573	13	623	7
324	27	374	1	424	0	474	11	524	27	574	11	624	8
325	26	375	11	425	15	475	12	525	11	575	12	625	9
326	13	376	12	426	11	476	25	526	13	576	11	626	10
327	27	377	11	427	12	477	11	527	27	577	13	627	11
328	13	378	13	428	25	478	13	528	11	578	11	628	12
329	25	379	11	429	11	479	25	529	12	579	12	629	13
330	21	380	12	430	13	480	11	530	27	580	14	630	14
331	26	381	11	431	25	481	12	531	11	581	13	631	15
332	19	382	13	432	11	482	25	532	13	582	11	632	16
333	26	383	11	433	12	483	11	533	27	583	12	633	17
334	13	384	12	434	25	484	13	534	11	584	25	634	18
335	25	385	11	435	11	485	25	535	12	585	11	635	19
336	2	386	13	436	13	486	11	536	27	586	13	636	20
337	27	387	11	437	25	487	12	537	13	587	25	637	21
338	25	388	12	438	11	488	25	538	27	588	11	638	22
339	21	389	11	439	12	489	13	539	10	589	12	639	23
340	26	390	12	440	25	490	25	540	27	590	25	640	24
341	19	391	27	441	11	491	11	541	8	591	11	641	25
342	26	392	11	442	13	492	12	542	27	592	13	642	26
343	10	393	13	443	25	493	11	543	11	593	25	643	27
344	27	394	27	444	11	494	13	544	12	594	11		
345	8	395	11	445	12	495	11	545	11	595	12		
346	13	396	12	446	25	496	12	546	13	596	25		
347	10	397	27	447	11	497	11	547	11	597	11		
348	13	398	11	448	13	498	13	548	12	598	13		
349	2	399	13	449	25	499	11	549	13	599	25		
350	12	400	27	450	11	500	12	550	27	600	11		
351	13	401	11	451	12	501	11	551	13	601	12		
352	11	402	12	452	25	502	13	552	18	602	25		
353	12	403	27	453	13	503	11	553	21	603	13		
354	25	404	11	454	13	504	12	554	26	604	25		
355	13	405	13	455	25	505	14	555	11	605	11		
356	25	406	27	456	11	506	11	556	12	606	12		
357	9	407	11	457	12	507	13	557	11	607	27		
358	1	408	12	458	25	508	11	558	13	608	11		
359	12	409	27	459	11	509	12	559	11	609	13		
360	15	410	11	460	13	510	27	560	12	610	27		
361	13	411	13	461	25	511	11	561	13	611	24		
362	11	412	27	462	11	512	13	562	10	612	21		
363	12	413	11	463	12	513	27	563	9	613	4		
364	11	414	12	464	25	514	24	564	25	614	26		
365	13	415	27	465	11	515	21	565	18	615	25		
366	11	416	11	466	13	516	12	566	21	616	0		
367	12	417	13	467	25	517	25	567	25	617	1		
368	11	418	27	468	11	518	11	568	2	618	2		
369	13	419	11	469	12	519	13	569	6	619	3		
370	11	420	12	470	25	520	25	570	11	620	4		
371	12	421	27	471	11	521	11	571	12	621	5		

Figure B.2: Prototype Genome Image



0	h-alloc	50	push	100	push	150	push	200	pop	250	swap-stk	300	21
1	h-search	51	add	101	nop-A	151	nop-C	201	pop	251	push	301	20
2	nop-A	52	read	102	write	152	swap-stk	202	nop-A	252	swap-stk	302	19
3	nop-B	53	swap-stk	103	inc	153	pop	203	swap-stk	253	pop	303	18
4	push	54	pop	104	swap-stk	154	nop-C	204	push	254	swap-stk	304	17
5	nop-A	55	swap-stk	105	pop	155	swap-stk	205	nop-A	255	push	305	16
6	push	56	push	106	nop-C	156	push	206	swap-stk	256	swap-stk	306	15
7	nop-C	57	swap-stk	107	swap-stk	157	nop-C	207	pop	257	pop	307	14
8	mov-head	58	pop	108	push	158	swap-stk	208	nop-A	258	dec	308	13
9	nop-B	59	swap-stk	109	nop-C	159	pop	209	swap-stk	259	push	309	12
10	get-head	60	push	110	swap-stk	160	nop-C	210	push	260	swap-stk	310	11
11	nop-B	61	swap-stk	111	pop	161	swap-stk	211	nop-A	261	pop	311	10
12	push	62	pop	112	nop-C	162	push	212	swap-stk	262	nop-C	312	9
13	nop-C	63	swap-stk	113	swap-stk	163	nop-C	213	pop	263	swap-stk	313	8
14	h-search	64	push	114	push	164	swap-stk	214	nop-A	264	push	314	7
15	nop-A	65	swap-stk	115	nop-C	165	pop	215	push	265	nop-C	315	6
16	nop-C	66	pop	116	swap-stk	166	nop-C	216	nop-A	266	swap-stk	316	5
17	mov-head	67	swap-stk	117	pop	167	push	217	swap	267	pop	317	4
18	nop-B	68	pop	118	nop-C	168	nop-C	218	nop-A	268	nop-C	318	3
19	get-head	69	nop-A	119	swap-stk	169	swap-stk	219	sub	269	swap-stk	319	2
20	nop-B	70	swap-stk	120	push	170	pop	220	nop-A	270	push	320	1
21	swap	71	push	121	nop-C	171	swap-stk	221	swap-stk	271	nop-C	321	0
22	nop-A	72	nop-A	122	swap-stk	172	push	222	pop	272	swap-stk		
23	sub	73	swap-stk	123	pop	173	swap-stk	223	swap-stk	273	pop		
24	push	74	pop	124	nop-C	174	pop	224	push	274	nop-C		
25	swap	75	nop-A	125	swap-stk	175	swap-stk	225	swap-stk	275	swap-stk		
26	push	76	swap-stk	126	push	176	push	226	pop	276	push		
27	h-search	77	push	127	nop-C	177	swap-stk	227	push	277	nop-C		
28	pop	78	nop-A	128	swap-stk	178	pop	228	nop-A	278	swap-stk		
29	push	79	swap-stk	129	pop	179	swap-stk	229	push	279	pop		
30	swap-stk	80	pop	130	nop-C	180	push	230	set-flow	280	nop-C		
31	pop	81	nop-A	131	push	181	swap-stk	231	mov-head	281	push		
32	nop-C	82	swap-stk	132	push	182	pop	232	nop-B	282	nop-C		
33	push	83	push	133	nop-C	183	dec	233	swap-stk	283	swap-stk		
34	nop-C	84	nop-A	134	swap-stk	184	swap-stk	234	pop	284	pop		
35	add	85	swap-stk	135	pop	185	push	235	swap-stk	285	nop-A		
36	read	86	pop	136	nop-C	186	swap-stk	236	push	286	swap-stk		
37	pop	87	nop-A	137	swap-stk	187	pop	237	swap-stk	287	push		
38	inc	88	swap-stk	138	push	188	nop-A	238	pop	288	nop-A		
39	push	89	dec	139	nop-C	189	swap-stk	239	push	289	if-n-equ		
40	swap-stk	90	nop-A	140	swap-stk	190	push	240	swap	290	mov-head		
41	pop	91	swap-stk	141	pop	191	nop-A	241	add	291	h-divide		
42	swap-stk	92	pop	142	nop-C	192	if-n-equ	242	nop-C	292	nop-B		
43	push	93	nop-A	143	swap-stk	193	mov-head	243	set-flow	293	nop-C		
44	swap-stk	94	swap-stk	144	push	194	pop	244	mov-head	294	27		
45	pop	95	push	145	nop-C	195	nop-C	245	nop-C	295	26		
46	swap-stk	96	nop-A	146	swap-stk	196	swap-stk	246	h-search	296	25		
47	push	97	swap-stk	147	pop	197	push	247	h-copy	297	24		
48	swap-stk	98	pop	148	nop-C	198	nop-C	248	swap-stk	298	23		
49	pop	99	nop-A	149	swap-stk	199	swap-stk	249	pop	299	22		

Figure B.3: The Mutant Image (First Half)

322	5	372	13	422	13	472	13	522	12	572	11	622	6
323	2	373	9	423	27	473	25	523	12	573	13	623	7
324	27	374	1	424	0	474	11	524	27	574	11	624	8
325	26	375	11	425	15	475	12	525	11	575	12	625	9
326	13	376	12	426	11	476	25	526	13	576	11	626	10
327	27	377	11	427	12	477	11	527	27	577	13	627	11
328	13	378	13	428	25	478	13	528	11	578	11	628	12
329	25	379	11	429	11	479	25	529	12	579	12	629	13
330	21	380	12	430	13	480	11	530	27	580	14	630	14
331	26	381	11	431	25	481	12	531	11	581	13	631	15
332	19	382	13	432	11	482	25	532	13	582	11	632	16
333	26	383	11	433	12	483	11	533	27	583	12	633	17
334	13	384	12	434	25	484	13	534	11	584	25	634	18
335	25	385	11	435	11	485	25	535	12	585	11	635	19
336	2	386	13	436	13	486	11	536	27	586	13	636	20
337	27	387	11	437	25	487	12	537	13	587	25	637	21
338	25	388	12	438	11	488	25	538	27	588	11	638	22
339	21	389	11	439	12	489	13	539	10	589	12	639	23
340	26	390	12	440	25	490	25	540	27	590	25	640	24
341	19	391	27	441	11	491	11	541	8	591	11	641	25
342	26	392	11	442	13	492	12	542	27	592	13	642	26
343	10	393	13	443	25	493	11	543	11	593	25	643	27
344	27	394	27	444	11	494	13	544	12	594	11		
345	8	395	11	445	12	495	11	545	11	595	12		
346	13	396	12	446	25	496	12	546	13	596	25		
347	10	397	27	447	11	497	11	547	11	597	11		
348	13	398	11	448	13	498	13	548	12	598	13		
349	2	399	13	449	25	499	11	549	13	599	25		
350	12	400	27	450	11	500	12	550	27	600	11		
351	13	401	11	451	12	501	11	551	13	601	12		
352	11	402	12	452	25	502	13	552	18	602	25		
353	12	403	27	453	13	503	11	553	21	603	13		
354	25	404	11	454	13	504	12	554	26	604	25		
355	13	405	13	455	25	505	14	555	11	605	11		
356	25	406	27	456	11	506	11	556	12	606	12		
357	9	407	11	457	12	507	13	557	11	607	27		
358	1	408	12	458	25	508	11	558	13	608	11		
359	12	409	27	459	11	509	12	559	11	609	13		
360	15	410	11	460	13	510	27	560	12	610	27		
361	13	411	14	461	25	511	11	561	13	611	24		
362	11	412	27	462	11	512	13	562	10	612	21		
363	12	413	11	463	12	513	27	563	9	613	4		
364	11	414	12	464	25	514	24	564	25	614	26		
365	13	415	27	465	11	515	21	565	18	615	25		
366	11	416	11	466	13	516	12	566	21	616	0		
367	12	417	13	467	25	517	25	567	25	617	1		
368	11	418	27	468	11	518	11	568	2	618	2		
369	13	419	11	469	12	519	13	569	6	619	3		
370	11	420	12	470	25	520	25	570	11	620	4		
371	12	421	27	471	11	521	11	571	12	621	5		

Figure B.4: The Mutant Genome Image (Latter Half)

Appendix C

Trace File Formats

Compare the old and the new trace file formats here. The old format contains the whole series of the virtual CPU state from the start to the division (or the time-out), plus the "final" memory image and the "child" memory image (if division occurred). The new format contains the initial memory, the final memory and the child memory, plus the adjacency graph representing the lineage graph. This excludes CPU state transitions in the middle to reduce the file size. Note that sequences of strains are shown in alphabetical labels internally in Avida (i.e., `a`, `b`, `c`, `d`, ..., `y`, `z`, `A`, `B` correspond to `0`, `1`, `2`, `3`, ..., `24`, `25`, `26`, `27`), which are not used in the body text.

Listing C.1: Old Trace File Example

```
---------------------------
U:-1
1 IP:0    AX:0 [0x0]  BX:0 [0x0]  CX:0 [0x0]
  R-Head:0 W-Head:0 F-Head:0  RL:
* Stack 0: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000
  Stack 1: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000
  Mem (644):  wzaboaocgbiboczacgbibratorozpoqpcocsApmoqpqoqpqoqposAqpqoqpqoq
pqoqpqpaqoaqpaqoaqpaqoaqpaqoaqpaqoaqpaoaBmqpcqocqpcqocqpcqocqpcqocqpcoocqpcq
ocqpcqocqpcqocqpcqocqpcqocqpcocqpqoqpqoqpqoqpnqoqpaqoadgpcqocqppaqoaqpaqoaqp
aoarataqpqoqpoaojgbqpqoqporscjgczvqpqoqpqoqpnoqpcqocqpcqocqpcqocqpcocqpaqoad
gxbcBAzyxwvutsrqponmlkjihgfedcbafcBAnBnzvAtAnzcBzvAtAkBinkncmnlmznzjbmpnlmln
lmlnlmnjblmlnlmlnlmlnlmlmBlnBlmBlnBlmBlnBlmBlnBlmBlnBlmBnBaplmzlnzlmzlnzlmzl
nzlmzlnzlmznnzlmzlnzlmzlnzlmzlnzlmzlnzlmzlnzlmznzlmlnlmlnlmlnlmolnlmBlnByvmz
lnzlmmBlnBlmBlnBlmBnBkBiBlmlnlmnBnsvAlmlnlmnkjzsvzcglmlnlmlnlmonlmzlnzlmzlnz
lmzlnzlmznzlmBlnByveAzabcdefghijklmnopqrstuvwxyzAB
  MeritBase:644 Bonus:1 Errors:0 Donates:0
  Task Count (Quality):

Input (env): 0x0f13149f 0x3308e53e 0x556241eb
Input (buf):
Output:
---------------------------
ABOUT TO EXECUTE: h-alloc
---------------------------


......
```

```
U:-1
52218 IP:291    AX:644 [0x284]  BX:0 [0x0]  CX:0 [0x0]
  R-Head:644 W-Head:1288 F-Head:247  RL:
* Stack 0: 0x00000284 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000
  Stack 1: 0x00000002 0x00000126 0x00000000 0x00000142 0x00000142 0x00000142
 0x00000000 0x00000000 0x00000000 0x00000000
  Mem (1932):  wzaboaocgbiboczacgbibratorozpoqpcocsApmoqpqoqpqoqposAqpqoqpqo
qpqoqpqpaqoaqpaqoaqpaqoaqpaqoaqpaqoaqpaoaBmqpcqocqpcqocqpcqocqpcqocqpcoocqpc
qocqpcqocqpcqocqpcqocqpcqocqpcocqpqoqpqoqpqoqpnqoqpaqoadgpcqocqppaqoaqpaqoaq
paoarataqpqoqpoaojgbqpqoqporscjgczvqpqoqpqoqpnoqpcqocqpcqocqpcqocqpcocqpaqoa
dgxbcBAzyxwvutsrqponmlkjihgfedcbafcBAnBnzvAtAnzcBzvAtAkBinkncmnlmznzjbmpnlml
nlmlnlmnjblmlnlmlnlmlnlmlmBlnBlmBlnBlmBlnBlmBlnBlmBlnBlmBnBaplmzlnzlmzlnzlmz
lnzlmzlnzlmznnzlmzlnzlmzlnzlmzlnzlmzlnzlmzlnzlmznzlmlnlmlnlmlnlmolnlmBlnByvm
zlnzlmmBlnBlmBlnBlmBnBkBiBlmlnlmnBnsvAlmlnlmnkjzsvzcglmlnlmlnlmonlmzlnzlmzln
zlmzlnzlmznzlmBlnByveAzabcdefghijklmnopqrstuvwxyzABwzaboaocgbiboczacgbibrato
rozpoqpcocsApmoqpqoqpqoqposAqpqoqpqoqpqoqpqpaqoaqpaqoaqpaqoaqpaqoaqpaqoaqpao
aBmqpcqocqpcqocqpcqocqpcqocqpcoocqpcqocqpcqocqpcqocqpcqocqpcqocqpcocqpqoqpqo
qpqoqpnqoqpaqoadgpcqocqppaqoaqpaqoaqpaoarataqpqoqpoaojgbqpqoqporscjgczvqpqoq
pqoqpnoqpcqocqpcqocqpcqocqpcocqpaqoadgxbcBAzyxwvutsrqponmlkjihgfedcbafcBAnBn
zvAtAnzcBzvAtAkBinkncmnlmznzjbmpnlmlnlmlnlmnjblmlnlmlnlmlnlmlmBlnBlmBlnBlmBl
nBlmBlnBlmBlnBlmBnBaplmzlnzlmzlnzlmzlnzlmzlnzlmznnzlmzlnzlmzlnzlmzlnzlmzlnzl
mzlnzlmznzlmlnlmlnlmlnlmolnlmBlnByvmzlnzlmmBlnBlmBlnBlmBnBkBiBlmlnlmnBnsvAlm
lnlmnkjzsvzcglmlnlmlnlmonlmzlnzlmzlnzlmzlnzlmznzlmBlnByveAzabcdefghijklmnopq
rstuvwxyzABaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  MeritBase:644 Bonus:1 Errors:0 Donates:0
  Task Count (Quality):

Input (env): 0x0f13149f 0x3308e53e 0x556241eb
Input (buf):
Output:
---------------------------
ABOUT TO EXECUTE: h-divide
---------------------------
  MeritBase:644 Bonus:1 Errors:0 Donates:0
  Task Count (Quality):



# Final Memory: wzaboaocgbiboczacgbibratorozpoqpcocsApmoqpqoqpqoqposAqpqoqpq
oqpqoqpqpaqoaqpaqoaqpaqoaqpaqoaqpaqoaqpaoaBmqpcqocqpcqocqpcqocqpcqocqpcoocqp
cqocqpcqocqpcqocqpcqocqpcqocqpcocqpqoqpqoqpqoqpnqoqpaqoadgpcqocqppaqoaqpaqoa
qpaoarataqpqoqpoaojgbqpqoqporscjgczvqpqoqpqoqpnoqpcqocqpcqocqpcqocqpcocqpaqo
adgxbcBAzyxwvutsrqponmlkjihgfedcbafcBAnBnzvAtAnzcBzvAtAkBinkncmnlmznzjbmpnlm
lnlmlnlmnjblmlnlmlnlmlnlmlmBlnBlmBlnBlmBlnBlmBlnBlmBlnBlmBnBaplmzlnzlmzlnzlm
zlnzlmzlnzlmznnzlmzlnzlmzlnzlmzlnzlmzlnzlmzlnzlmznzlmlnlmlnlmlnlmolnlmBlnByv
```

```
mzlnzlmmBlnBlmBlnBlmBnBkBiBlmlnlmnBnsvAlmlnlmnkjzsvzcglmlnlmlnlmonlmzlnzlmzl
nzlmzlnzlmznzlmBlnByveAzabcdefghijklmnopqrstuvwxyzAB
# Child Memory: wzaboaocgbiboczacgbibratorozpoqpcocsApmoqpqoqpqoqposAqpqoqpq
oqpqoqpqpaqoaqpaqoaqpaqoaqpaqoaqpaqoaqpaoaBmqpcqocqpcqocqpcqocqpcqocqpcoocqp
cqocqpcqocqpcqocqpcqocqpcqocqpcocqpqoqpqoqpqoqpnqoqpaqoadgpcqocqppaqoaqpaqoa
qpaoarataqpqoqpoaojgbqpqoqporscjgczvqpqoqpqoqpnoqpcqocqpcqocqpcqocqpcocqpaqo
adgxbcBAzyxwvutsrqponmlkjihgfedcbafcBAnBnzvAtAnzcBzvAtAkBinkncmnlmznzjbmpnlm
lnlmlnlmnjblmlnlmlnlmlnlmlmBlnBlmBlnBlmBlnBlmBlnBlmBlnBlmBnBaplmzlnzlmzlnzlm
zlnzlmzlnzlmznnzlmzlnzlmzlnzlmzlnzlmzlnzlmzlnzlmznzlmlnlmlnlmlnlmolnlmBlnByv
mzlnzlmmBlnBlmBlnBlmBnBkBiBlmlnlmnBnsvAlmlnlmnkjzsvzcglmlnlmlnlmonlmzlnzlmzl
nzlmzlnzlmznzlmBlnByveAzabcdefghijklmnopqrstuvwxyzAB
```

Listing C.2: New Trace File Example

```
---------------------------
# Init Memory: wzaboaocgbiboczacgbibratorozpoqpcocsApmoqpqoqpqoqposAqpqoqpqo
qpqoqpqpaqoaqpaqoaqpaqoaqpaqoaqpaqoaqpaoaBmqpcqocqpcqocqpcqocqpcqocqpcoocqpc
qocqpcqocqpcqocqpcqocqpcqocqpcocqpqoqpqoqpqoqpnqoqpaqoadgpcqocqppaqoaqpaqoaq
paoarataqpqoqpoaojgbqpqoqporscjgczvqpqoqpqoqpnoqpcqocqpcqocqpcqocqpcocqpaqoa
dgxbcBAzyxwvutsrqponmlkjihgfedcbafcBAnBnzvAtAnzcBzvAtAkBinkncmnlmznzjbmpnlml
nlmlnlmnjblmlnlmlnlmlnlmlmBlnBlmBlnBlmBlnBlmBlnBlmBlnBlmBnBaplmzlnzlmzlnzlmz
lnzlmzlnzlmznnzlmzlnzlmzlnzlmzlnzlmzlnzlmzlnzlmznzlmlnlmlnlmlnlmolnlmBlnByvm
zlnzlmmBlnBlmBlnBlmBnBkBiBlmlnlmnBnsvAlmlnlmnkjzsvzcglmlnlmlnlmonlmzlnzlmzln
zlmzlnzlmznzlmBlnByveAzabcdefghijklmnopqrstuvwxyzAB

# Final Memory: wzaboaocgbiboczacgbibratorozpoqpcocsApmoqpqoqpqoqposAqpqoqpq
oqpqoqpqpaqoaqpaqoaqpaqoaqpaqoaqpaqoaqpaoaBmqpcqocqpcqocqpcqocqpcqocqpcoocqp
cqocqpcqocqpcqocqpcqocqpcqocqpcocqpqoqpqoqpqoqpnqoqpaqoadgpcqocqppaqoaqpaqoa
qpaoarataqpqoqpoaojgbqpqoqporscjgczvqpqoqpqoqpnoqpcqocqpcqocqpcqocqpcocqpaqo
adgxbcBAzyxwvutsrqponmlkjihgfedcbafcBAnBnzvAtAnzcBzvAtAkBinkncmnlmznzjbmpnlm
lnlmlnlmnjblmlnlmlnlmlnlmlmBlnBlmBlnBlmBlnBlmBlnBlmBlnBlmBnBaplmzlnzlmzlnzlm
zlnzlmzlnzlmznnzlmzlnzlmzlnzlmzlnzlmzlnzlmzlnzlmznzlmlnlmlnlmlnlmolnlmBlnByv
mzlnzlmmBlnBlmBlnBlmBnBkBiBlmlnlmnBnsvAlmlnlmnkjzsvzcglmlnlmlnlmonlmzlnzlmzl
nzlmzlnzlmznzlmBlnByveAzabcdefghijklmnopqrstuvwxyzAB
# Child Memory: wzaboaocgbiboczacgbibratorozpoqpcocsApmoqpqoqpqoqposAqpqoqpq
oqpqoqpqpaqoaqpaqoaqpaqoaqpaqoaqpaqoaqpaoaBmqpcqocqpcqocqpcqocqpcqocqpcoocqp
cqocqpcqocqpcqocqpcqocqpcqocqpcocqpqoqpqoqpqoqpnqoqpaqoadgpcqocqppaqoaqpaqoa
qpaoarataqpqoqpoaojgbqpqoqporscjgczvqpqoqpqoqpnoqpcqocqpcqocqpcqocqpcocqpaqo
adgxbcBAzyxwvutsrqponmlkjihgfedcbafcBAnBnzvAtAnzcBzvAtAkBinkncmnlmznzjbmpnlm
lnlmlnlmnjblmlnlmlnlmlnlmlmBlnBlmBlnBlmBlnBlmBlnBlmBlnBlmBnBaplmzlnzlmzlnzlm
zlnzlmzlnzlmznnzlmzlnzlmzlnzlmzlnzlmzlnzlmzlnzlmznzlmlnlmlnlmlnlmolnlmBlnByv
mzlnzlmmBlnBlmBlnBlmBnBkBiBlmlnlmnBnsvAlmlnlmnkjzsvzcglmlnlmlnlmonlmzlnzlmzl
nzlmzlnzlmznzlmBlnByveAzabcdefghijklmnopqrstuvwxyzAB
Note: Divided
---------------------------
Index Size Incubated? Divided? GestationTime LeftChildIndex RightChildIndex
0 644 true true 52218 0 0
```

Appendix D

Instruction Library Entries

As of the version 2.10.0, the library lists 476 entries of instructions. Below, an overview of it (i.e., an excerpt from the original Avida source code in C++) is presented with the comments according to the original wording and terminology. There is no official instruction-by-instruction description provided in the documentation for them except for those featured as the default instruction set.

Listing D.1: Instruction Library

```cpp
static const tInstLibEntry<tMethod> s_f_array[] = {
  /*
   Note: all entries of cNOPEntryCPU s_n_array must have corresponding
   in the same order in tInstLibEntry<tMethod> s_f_array,
   and these entries must be the first elements of s_f_array.
   */
  tInstLibEntry<tMethod>("nop-A", &cHardwareCPU::Inst_Nop, (nInstFlag::DEFAULT | nInstFlag::NOP), "No-operation instruction; modifies other instructions"),
  tInstLibEntry<tMethod>("nop-B", &cHardwareCPU::Inst_Nop, (nInstFlag::DEFAULT | nInstFlag::NOP), "No-operation instruction; modifies other instructions"),
  tInstLibEntry<tMethod>("nop-C", &cHardwareCPU::Inst_Nop, (nInstFlag::DEFAULT | nInstFlag::NOP), "No-operation instruction; modifies other instructions"),

  tInstLibEntry<tMethod>("nop-X", &cHardwareCPU::Inst_Nop, 0, "True no-operation instruction: does nothing"),
  tInstLibEntry<tMethod>("if-equ-0", &cHardwareCPU::Inst_If0, 0, "Execute next instruction if ?BX?==0, else skip it"),
  tInstLibEntry<tMethod>("if-not-0", &cHardwareCPU::Inst_IfNot0, 0, "Execute next instruction if ?BX?!=0, else skip it"),
  tInstLibEntry<tMethod>("if-equ-0-defaultAX", &cHardwareCPU::Inst_If0_defaultAX, 0, "Execute next instruction if ?AX?==0, else skip it"),
  tInstLibEntry<tMethod>("if-not-0-defaultAX", &cHardwareCPU::Inst_IfNot0_defaultAX, 0, "Execute next instruction if ?AX?!=0, else skip it"),
  tInstLibEntry<tMethod>("if-n-equ", &cHardwareCPU::Inst_IfNEqu, nInstFlag::DEFAULT, "Execute next instruction if ?BX?!=?CX?, else skip it"),
  tInstLibEntry<tMethod>("if-equ", &cHardwareCPU::Inst_IfEqu, 0, "Execute next instruction if ?BX?==?CX?, else skip it"),
  tInstLibEntry<tMethod>("if-grt-0", &cHardwareCPU::Inst_IfGr0),
  tInstLibEntry<tMethod>("if-grt", &cHardwareCPU::Inst_IfGr),
  tInstLibEntry<tMethod>("if->=-0", &cHardwareCPU::Inst_IfGrEqu0),
  tInstLibEntry<tMethod>("if->=", &cHardwareCPU::Inst_IfGrEqu),
  tInstLibEntry<tMethod>("if-les-0", &cHardwareCPU::Inst_IfLess0),
  tInstLibEntry<tMethod>("if-less", &cHardwareCPU::Inst_IfLess, nInstFlag::DEFAULT, "Execute next instruction if ?BX? < ?CX?, else skip it"),
  tInstLibEntry<tMethod>("if-<=-0", &cHardwareCPU::Inst_IfLsEqu0),
  tInstLibEntry<tMethod>("if-<=", &cHardwareCPU::Inst_IfLsEqu),
  tInstLibEntry<tMethod>("if-A!=B", &cHardwareCPU::Inst_IfANotEqB),
  tInstLibEntry<tMethod>("if-B!=C", &cHardwareCPU::Inst_IfBNotEqC),
  tInstLibEntry<tMethod>("if-A!=C", &cHardwareCPU::Inst_IfANotEqC),
  tInstLibEntry<tMethod>("if-bit-1", &cHardwareCPU::Inst_IfBit1),
  tInstLibEntry<tMethod>("if-grt-X", &cHardwareCPU::Inst_IfGrX),
  tInstLibEntry<tMethod>("if-equ-X", &cHardwareCPU::Inst_IfEquX),

  tInstLibEntry<tMethod>("if-aboveResLevel", &cHardwareCPU::Inst_IfAboveResLevel),
  tInstLibEntry<tMethod>("if-aboveResLevel.end", &cHardwareCPU::Inst_IfAboveResLevelEnd),
  tInstLibEntry<tMethod>("if-notAboveResLevel", &cHardwareCPU::Inst_IfNotAboveResLevel),
  tInstLibEntry<tMethod>("if-notAboveResLevel.end", &cHardwareCPU::Inst_IfNotAboveResLevelEnd),

  // Probabilistic ifs.
  tInstLibEntry<tMethod>("if-p-0.125", &cHardwareCPU::Inst_IfP0p125, nInstFlag::STALL),
  tInstLibEntry<tMethod>("if-p-0.25", &cHardwareCPU::Inst_IfP0p25, nInstFlag::STALL),
  tInstLibEntry<tMethod>("if-p-0.50", &cHardwareCPU::Inst_IfP0p50, nInstFlag::STALL),
  tInstLibEntry<tMethod>("if-p-0.75", &cHardwareCPU::Inst_IfP0p75, nInstFlag::STALL),

  // The below series of conditionals extend the traditional Avida
  // single-instruction-skip to a block, or series of instructions.
  tInstLibEntry<tMethod>("if-less.end", &cHardwareCPU::Inst_IfLessEnd, nInstFlag::STALL),
  tInstLibEntry<tMethod>("if-n-equ.end", &cHardwareCPU::Inst_IfNotEqualEnd, nInstFlag::STALL),
  tInstLibEntry<tMethod>("if->=.end", &cHardwareCPU::Inst_IfGrtEquEnd, nInstFlag::STALL),
  tInstLibEntry<tMethod>("else", &cHardwareCPU::Inst_Else, nInstFlag::STALL),
  tInstLibEntry<tMethod>("end-if", &cHardwareCPU::Inst_EndIf, nInstFlag::STALL),

  tInstLibEntry<tMethod>("jump-f", &cHardwareCPU::Inst_JumpF),
  tInstLibEntry<tMethod>("jump-b", &cHardwareCPU::Inst_JumpB),
  tInstLibEntry<tMethod>("call", &cHardwareCPU::Inst_Call),
  tInstLibEntry<tMethod>("return", &cHardwareCPU::Inst_Return),

  tInstLibEntry<tMethod>("throw", &cHardwareCPU::Inst_Throw),
  tInstLibEntry<tMethod>("throwif=0", &cHardwareCPU::Inst_ThrowIf0),
  tInstLibEntry<tMethod>("throwif!=0", &cHardwareCPU::Inst_ThrowIfNot0),
  tInstLibEntry<tMethod>("catch", &cHardwareCPU::Inst_Catch),

  tInstLibEntry<tMethod>("goto", &cHardwareCPU::Inst_Goto),
  tInstLibEntry<tMethod>("goto-if=0", &cHardwareCPU::Inst_GotoIf0),
  tInstLibEntry<tMethod>("goto-if!=0", &cHardwareCPU::Inst_GotoIfNot0),
  tInstLibEntry<tMethod>("label", &cHardwareCPU::Inst_Label),

  tInstLibEntry<tMethod>("pop", &cHardwareCPU::Inst_Pop, nInstFlag::DEFAULT, "Remove top number from stack and place into ?BX?"),
  tInstLibEntry<tMethod>("push", &cHardwareCPU::Inst_Push, nInstFlag::DEFAULT, "Copy number from ?BX? and place it into the stack"),
  tInstLibEntry<tMethod>("swap-stk", &cHardwareCPU::Inst_SwitchStack, nInstFlag::DEFAULT, "Toggle which stack is currently being used"),
  tInstLibEntry<tMethod>("flip-stk", &cHardwareCPU::Inst_FlipStack),
  tInstLibEntry<tMethod>("swap", &cHardwareCPU::Inst_Swap, nInstFlag::DEFAULT, "Swap the contents of ?BX? with ?CX?"),
  tInstLibEntry<tMethod>("swap-AB", &cHardwareCPU::Inst_SwapAB),
  tInstLibEntry<tMethod>("swap-BC", &cHardwareCPU::Inst_SwapBC),
  tInstLibEntry<tMethod>("swap-AC", &cHardwareCPU::Inst_SwapAC),
  tInstLibEntry<tMethod>("copy-reg", &cHardwareCPU::Inst_CopyReg),
  tInstLibEntry<tMethod>("set_A=B", &cHardwareCPU::Inst_CopyRegAB),
  tInstLibEntry<tMethod>("set_A=C", &cHardwareCPU::Inst_CopyRegAC),
  tInstLibEntry<tMethod>("set_B=A", &cHardwareCPU::Inst_CopyRegBA),
  tInstLibEntry<tMethod>("set_B=C", &cHardwareCPU::Inst_CopyRegBC),
  tInstLibEntry<tMethod>("set_C=A", &cHardwareCPU::Inst_CopyRegCA),
  tInstLibEntry<tMethod>("set_C=B", &cHardwareCPU::Inst_CopyRegCB),
  tInstLibEntry<tMethod>("reset", &cHardwareCPU::Inst_Reset),

  tInstLibEntry<tMethod>("pop-A", &cHardwareCPU::Inst_PopA),
  tInstLibEntry<tMethod>("pop-B", &cHardwareCPU::Inst_PopB),
  tInstLibEntry<tMethod>("pop-C", &cHardwareCPU::Inst_PopC),
  tInstLibEntry<tMethod>("push-A", &cHardwareCPU::Inst_PushA),
  tInstLibEntry<tMethod>("push-B", &cHardwareCPU::Inst_PushB),
  tInstLibEntry<tMethod>("push-C", &cHardwareCPU::Inst_PushC),

  tInstLibEntry<tMethod>("shift-r", &cHardwareCPU::Inst_ShiftR, nInstFlag::DEFAULT, "Shift bits in ?BX? right by one (divide by two)"),
  tInstLibEntry<tMethod>("shift-l", &cHardwareCPU::Inst_ShiftL, nInstFlag::DEFAULT, "Shift bits in ?BX? left by one (multiply by two)"),
  tInstLibEntry<tMethod>("bit-1", &cHardwareCPU::Inst_Bit1),
  tInstLibEntry<tMethod>("set-num", &cHardwareCPU::Inst_SetNum),
  tInstLibEntry<tMethod>("val-grey", &cHardwareCPU::Inst_ValGrey),
  tInstLibEntry<tMethod>("val-dir", &cHardwareCPU::Inst_ValDir),
  tInstLibEntry<tMethod>("val-add-p", &cHardwareCPU::Inst_ValAddP),
  tInstLibEntry<tMethod>("val-fib", &cHardwareCPU::Inst_ValFib),
  tInstLibEntry<tMethod>("val-poly-c", &cHardwareCPU::Inst_ValPolyC),
  tInstLibEntry<tMethod>("inc", &cHardwareCPU::Inst_Inc, nInstFlag::DEFAULT, "Increment ?BX? by one"),
  tInstLibEntry<tMethod>("dec", &cHardwareCPU::Inst_Dec, nInstFlag::DEFAULT, "Decrement ?BX? by one"),
  tInstLibEntry<tMethod>("zero", &cHardwareCPU::Inst_Zero, 0, "Set ?BX? to zero"),
  tInstLibEntry<tMethod>("all1s", &cHardwareCPU::Inst_All1s, 0, "Set ?BX? to all 1s in bitstring"),
  tInstLibEntry<tMethod>("neg", &cHardwareCPU::Inst_Neg),
  tInstLibEntry<tMethod>("square", &cHardwareCPU::Inst_Square),
  tInstLibEntry<tMethod>("sqrt", &cHardwareCPU::Inst_Sqrt),
  tInstLibEntry<tMethod>("not", &cHardwareCPU::Inst_Not),

  tInstLibEntry<tMethod>("add", &cHardwareCPU::Inst_Add, nInstFlag::DEFAULT, "Add BX to CX and place the result in ?BX?"),
  tInstLibEntry<tMethod>("sub", &cHardwareCPU::Inst_Sub, nInstFlag::DEFAULT, "Subtract CX from BX and place the result in ?BX?"),
  tInstLibEntry<tMethod>("mult", &cHardwareCPU::Inst_Mult, 0, "Multiple BX by CX and place the result in ?BX?"),
  tInstLibEntry<tMethod>("div", &cHardwareCPU::Inst_Div, 0, "Divide BX by CX and place the result in ?BX?"),
  tInstLibEntry<tMethod>("mod", &cHardwareCPU::Inst_Mod),
  tInstLibEntry<tMethod>("nand", &cHardwareCPU::Inst_Nand, nInstFlag::DEFAULT, "Nand BX by CX and place the result in ?BX?"),
  tInstLibEntry<tMethod>("or", &cHardwareCPU::Inst_Or),
  tInstLibEntry<tMethod>("nor", &cHardwareCPU::Inst_Nor),
  tInstLibEntry<tMethod>("and", &cHardwareCPU::Inst_And),
  tInstLibEntry<tMethod>("order", &cHardwareCPU::Inst_Order),
  tInstLibEntry<tMethod>("xor", &cHardwareCPU::Inst_Xor),

  // Instructions that modify specific bits in the register values
  tInstLibEntry<tMethod>("setbit", &cHardwareCPU::Inst_Setbit, nInstFlag::DEFAULT, "Set the bit in ?BX? specified by ?BX?'s complement"),
  tInstLibEntry<tMethod>("clearbit", &cHardwareCPU::Inst_Clearbit, nInstFlag::DEFAULT, "Clear the bit in ?BX? specified by ?BX?'s complement"),

  // Treatable instructions
  tInstLibEntry<tMethod>("nand-treatable", &cHardwareCPU::Inst_NandTreatable,
      nInstFlag::DEFAULT, "Nand BX by CX and place the result in ?BX?, fails if
      deme is treatable"),

  tInstLibEntry<tMethod>("copy", &cHardwareCPU::Inst_Copy),
  tInstLibEntry<tMethod>("read", &cHardwareCPU::Inst_ReadInst),
  tInstLibEntry<tMethod>("write", &cHardwareCPU::Inst_WriteInst),
  tInstLibEntry<tMethod>("stk-read", &cHardwareCPU::Inst_StackReadInst),
  tInstLibEntry<tMethod>("stk-writ", &cHardwareCPU::Inst_StackWriteInst),

  tInstLibEntry<tMethod>("compare", &cHardwareCPU::Inst_Compare),
  tInstLibEntry<tMethod>("if-n-cpy", &cHardwareCPU::Inst_IfNCpy),
  tInstLibEntry<tMethod>("allocate", &cHardwareCPU::Inst_Allocate),
  tInstLibEntry<tMethod>("divide", &cHardwareCPU::Inst_Divide, nInstFlag::STALL),
  tInstLibEntry<tMethod>("divideRS", &cHardwareCPU::Inst_DivideRS, nInstFlag::
      STALL),
  tInstLibEntry<tMethod>("c-alloc", &cHardwareCPU::Inst_CAlloc),
  tInstLibEntry<tMethod>("c-divide", &cHardwareCPU::Inst_CDivide, nInstFlag::STALL
      ),
  tInstLibEntry<tMethod>("inject", &cHardwareCPU::Inst_Inject, nInstFlag::STALL),
  tInstLibEntry<tMethod>("inject-r", &cHardwareCPU::Inst_InjectRand, nInstFlag::
      STALL),
  tInstLibEntry<tMethod>("transposon", &cHardwareCPU::Inst_Transposon),
  tInstLibEntry<tMethod>("search-f", &cHardwareCPU::Inst_SearchF),
  tInstLibEntry<tMethod>("search-b", &cHardwareCPU::Inst_SearchB),
  tInstLibEntry<tMethod>("mem-size", &cHardwareCPU::Inst_MemSize),

  tInstLibEntry<tMethod>("get", &cHardwareCPU::Inst_TaskGet, nInstFlag::STALL),
  tInstLibEntry<tMethod>("get-2", &cHardwareCPU::Inst_TaskGet2, nInstFlag::STALL),
  tInstLibEntry<tMethod>("stk-get", &cHardwareCPU::Inst_TaskStackGet, nInstFlag::
      STALL),
  tInstLibEntry<tMethod>("stk-load", &cHardwareCPU::Inst_TaskStackLoad, nInstFlag
      ::STALL),
  tInstLibEntry<tMethod>("put", &cHardwareCPU::Inst_TaskPut, nInstFlag::STALL),
  tInstLibEntry<tMethod>("put-reset", &cHardwareCPU::Inst_TaskPutResetInputs,
      nInstFlag::STALL),
  tInstLibEntry<tMethod>("IO", &cHardwareCPU::Inst_TaskIO, nInstFlag::DEFAULT |
      nInstFlag::STALL, "Output ?BX?, and input new number back into ?BX?"),
  tInstLibEntry<tMethod>("IO-Feedback", &cHardwareCPU::Inst_TaskIO_Feedback,
      nInstFlag::STALL, "Output ?BX?, and input new number back into ?BX?, and
      push 1,0, or -1 onto stack1 if merit increased, stayed the same, or
      decreased"),
  tInstLibEntry<tMethod>("IO-bc-0.001", &cHardwareCPU::Inst_TaskIO_BonusCost_0_001
      , nInstFlag::STALL),
  tInstLibEntry<tMethod>("match-strings", &cHardwareCPU::Inst_MatchStrings,
      nInstFlag::STALL),
  tInstLibEntry<tMethod>("sell", &cHardwareCPU::Inst_Sell, nInstFlag::STALL),
  tInstLibEntry<tMethod>("buy", &cHardwareCPU::Inst_Buy, nInstFlag::STALL),
  tInstLibEntry<tMethod>("send", &cHardwareCPU::Inst_Send, nInstFlag::STALL),
  tInstLibEntry<tMethod>("receive", &cHardwareCPU::Inst_Receive, nInstFlag::STALL)
      ,
  tInstLibEntry<tMethod>("sense", &cHardwareCPU::Inst_SenseLog2, nInstFlag::STALL)
      , // If you add more sense instructions
  tInstLibEntry<tMethod>("sense-unit", &cHardwareCPU::Inst_SenseUnit, nInstFlag::
      STALL), // and want to keep stats, also add
  tInstLibEntry<tMethod>("sense-m100", &cHardwareCPU::Inst_SenseMult100, nInstFlag
      ::STALL), // the names to cStats::cStats() @JEB

  tInstLibEntry<tMethod>("sense-resource0", &cHardwareCPU::Inst_SenseResource0,
      nInstFlag::STALL),
  tInstLibEntry<tMethod>("sense-resource1", &cHardwareCPU::Inst_SenseResource1,
      nInstFlag::STALL),
  tInstLibEntry<tMethod>("sense-resource2", &cHardwareCPU::Inst_SenseResource2, nInstFlag::STALL),
  tInstLibEntry<tMethod>("sense-faced-resource0", &cHardwareCPU::Inst_SenseFacedResource0, nInstFlag::STALL),
  tInstLibEntry<tMethod>("sense-faced-resource1", &cHardwareCPU::Inst_SenseFacedResource1, nInstFlag::STALL),
  tInstLibEntry<tMethod>("sense-faced-resource2", &cHardwareCPU::Inst_SenseFacedResource2, nInstFlag::STALL),

  tInstLibEntry<tMethod>("if-resources", &cHardwareCPU::Inst_IfResources, nInstFlag::STALL),
  tInstLibEntry<tMethod>("collect", &cHardwareCPU::Inst_Collect, nInstFlag::STALL),
  tInstLibEntry<tMethod>("collect-no-env-remove", &cHardwareCPU::Inst_CollectNoEnvRemove, nInstFlag::STALL),
  tInstLibEntry<tMethod>("destroy", &cHardwareCPU::Inst_Destroy, nInstFlag::STALL),
  tInstLibEntry<tMethod>("nop-collect", &cHardwareCPU::Inst_NopCollect),
  tInstLibEntry<tMethod>("collect-specific", &cHardwareCPU::Inst_CollectSpecific, nInstFlag::STALL),

  tInstLibEntry<tMethod>("donate-rnd", &cHardwareCPU::Inst_DonateRandom),
  tInstLibEntry<tMethod>("donate-kin", &cHardwareCPU::Inst_DonateKin),
  tInstLibEntry<tMethod>("donate-edt", &cHardwareCPU::Inst_DonateEditDist),
  tInstLibEntry<tMethod>("donate-gbg", &cHardwareCPU::Inst_DonateGreenBeardGene),
  tInstLibEntry<tMethod>("donate-tgb", &cHardwareCPU::Inst_DonateTrueGreenBeard),
  tInstLibEntry<tMethod>("donate-shadedgb", &cHardwareCPU::Inst_DonateShadedGreenBeard),
  tInstLibEntry<tMethod>("donate-threshgb", &cHardwareCPU::Inst_DonateThreshGreenBeard),
  tInstLibEntry<tMethod>("donate-quantagb", &cHardwareCPU::Inst_DonateQuantaThreshGreenBeard),
  tInstLibEntry<tMethod>("donate-gbsl", &cHardwareCPU::Inst_DonateGreenBeardSameLocus),
  tInstLibEntry<tMethod>("donate-NUL", &cHardwareCPU::Inst_DonateNULL),
  tInstLibEntry<tMethod>("donate-facing", &cHardwareCPU::Inst_DonateFacing),
  tInstLibEntry<tMethod>("receive-donated-energy", &cHardwareCPU::Inst_ReceiveDonatedEnergy, nInstFlag::STALL),
  tInstLibEntry<tMethod>("donate-energy", &cHardwareCPU::Inst_DonateEnergy, nInstFlag::STALL),
  tInstLibEntry<tMethod>("update-metabolic-rate", &cHardwareCPU::Inst_UpdateMetabolicRate, nInstFlag::STALL),
  tInstLibEntry<tMethod>("donate-energy-faced", &cHardwareCPU::Inst_DonateEnergyFaced, nInstFlag::STALL),
  tInstLibEntry<tMethod>("donate-energy-faced1", &cHardwareCPU::Inst_DonateEnergyFaced1, nInstFlag::STALL),
  tInstLibEntry<tMethod>("donate-energy-faced2", &cHardwareCPU::Inst_DonateEnergyFaced2, nInstFlag::STALL),
  tInstLibEntry<tMethod>("donate-energy-faced5", &cHardwareCPU::Inst_DonateEnergyFaced5, nInstFlag::STALL),
  tInstLibEntry<tMethod>("donate-energy-faced10", &cHardwareCPU::Inst_DonateEnergyFaced10, nInstFlag::STALL),
  tInstLibEntry<tMethod>("donate-energy-faced20", &cHardwareCPU::Inst_DonateEnergyFaced20, nInstFlag::STALL),
  tInstLibEntry<tMethod>("donate-energy-faced50", &cHardwareCPU::Inst_DonateEnergyFaced50, nInstFlag::STALL),
  tInstLibEntry<tMethod>("donate-energy-faced100", &cHardwareCPU::Inst_DonateEnergyFaced100, nInstFlag::STALL),
  tInstLibEntry<tMethod>("rotate-to-most-needy", &cHardwareCPU::Inst_RotateToMostNeedy, nInstFlag::STALL),
  tInstLibEntry<tMethod>("request-energy", &cHardwareCPU::Inst_RequestEnergy, nInstFlag::STALL),
  tInstLibEntry<tMethod>("request-energy-on", &cHardwareCPU::
      Inst_RequestEnergyFlagOn, nInstFlag::STALL),
  tInstLibEntry<tMethod>("request-energy-off", &cHardwareCPU::Inst_RequestEnergyFlagOff, nInstFlag::STALL),
  tInstLibEntry<tMethod>("increase-energy-donation", &cHardwareCPU::Inst_IncreaseEnergyDonation, nInstFlag::STALL),
  tInstLibEntry<tMethod>("decrease-energy-donation", &cHardwareCPU::Inst_DecreaseEnergyDonation, nInstFlag::STALL),
  tInstLibEntry<tMethod>("donate-resource0", &cHardwareCPU::Inst_DonateResource0, nInstFlag::STALL),
  tInstLibEntry<tMethod>("donate-resource1", &cHardwareCPU::Inst_DonateResource1, nInstFlag::STALL),
  tInstLibEntry<tMethod>("donate-resource2", &cHardwareCPU::Inst_DonateResource2, nInstFlag::STALL),
  tInstLibEntry<tMethod>("IObuf-add1", &cHardwareCPU::Inst_IOBufAdd1, nInstFlag::STALL),
  tInstLibEntry<tMethod>("IObuf-add0", &cHardwareCPU::Inst_IOBufAdd0, nInstFlag::STALL),

  tInstLibEntry<tMethod>("rotate-l", &cHardwareCPU::Inst_RotateL, nInstFlag::STALL),
  tInstLibEntry<tMethod>("rotate-r", &cHardwareCPU::Inst_RotateR, nInstFlag::STALL),
  tInstLibEntry<tMethod>("rotate-left-one", &cHardwareCPU::Inst_RotateLeftOne, nInstFlag::STALL),
  tInstLibEntry<tMethod>("rotate-right-one", &cHardwareCPU::Inst_RotateRightOne, nInstFlag::STALL),
  tInstLibEntry<tMethod>("rotate-label", &cHardwareCPU::Inst_RotateLabel, nInstFlag::STALL),
  tInstLibEntry<tMethod>("rotate-to-unoccupied-cell", &cHardwareCPU::Inst_RotateUnoccupiedCell, nInstFlag::STALL),
  tInstLibEntry<tMethod>("rotate-to-next-unoccupied-cell", &cHardwareCPU::Inst_RotateNextUnoccupiedCell, nInstFlag::STALL),
  tInstLibEntry<tMethod>("rotate-to-occupied-cell", &cHardwareCPU::Inst_RotateOccupiedCell, nInstFlag::STALL),
  tInstLibEntry<tMethod>("rotate-to-next-occupied-cell", &cHardwareCPU::Inst_RotateNextOccupiedCell, nInstFlag::STALL),
  tInstLibEntry<tMethod>("rotate-to-event-cell", &cHardwareCPU::Inst_RotateEventCell, nInstFlag::STALL),

  tInstLibEntry<tMethod>("set-cmut", &cHardwareCPU::Inst_SetCopyMut),
  tInstLibEntry<tMethod>("mod-cmut", &cHardwareCPU::Inst_ModCopyMut),
  tInstLibEntry<tMethod>("get-cell-xy", &cHardwareCPU::Inst_GetCellPosition),
  tInstLibEntry<tMethod>("get-cell-x", &cHardwareCPU::Inst_GetCellPositionX),
  tInstLibEntry<tMethod>("get-cell-y", &cHardwareCPU::Inst_GetCellPositionY),
  tInstLibEntry<tMethod>("dist-from-diag", &cHardwareCPU::Inst_GetDistanceFromDiagonal),

  // State Grid instructions
  tInstLibEntry<tMethod>("sg-move", &cHardwareCPU::Inst_SGMove),
  tInstLibEntry<tMethod>("sg-rotate-l", &cHardwareCPU::Inst_SGRotateL),
  tInstLibEntry<tMethod>("sg-rotate-r", &cHardwareCPU::Inst_SGRotateR),
  tInstLibEntry<tMethod>("sg-sense", &cHardwareCPU::Inst_SGSense),

  // Movement instructions
  tInstLibEntry<tMethod>("tumble", &cHardwareCPU::Inst_Tumble, nInstFlag::STALL),
  tInstLibEntry<tMethod>("move", &cHardwareCPU::Inst_Move, nInstFlag::STALL),
  tInstLibEntry<tMethod>("move-to-event", &cHardwareCPU::Inst_MoveToEvent, nInstFlag::STALL),
  tInstLibEntry<tMethod>("if-event-in-unoccupied-neighbor-cell", &cHardwareCPU::Inst_IfNeighborEventInUnoccupiedCell),
  tInstLibEntry<tMethod>("if-event-in-faced-cell", &cHardwareCPU::Inst_IfFacingEventCell),
  tInstLibEntry<tMethod>("if-event-in-current-cell", &cHardwareCPU::
      Inst_IfEventInCell),

  // Threading instructions
  tInstLibEntry<tMethod>("fork-th", &cHardwareCPU::Inst_ForkThread),
  tInstLibEntry<tMethod>("forkl", &cHardwareCPU::Inst_ForkThreadLabel),
  tInstLibEntry<tMethod>("forkl!=0", &cHardwareCPU::Inst_ForkThreadLabelIfNot0),
  tInstLibEntry<tMethod>("forkl=0", &cHardwareCPU::Inst_ForkThreadLabelIf0),
  tInstLibEntry<tMethod>("kill-th", &cHardwareCPU::Inst_KillThread),
  tInstLibEntry<tMethod>("id-th", &cHardwareCPU::Inst_ThreadID),

  // Head-based instructions
  tInstLibEntry<tMethod>("h-alloc", &cHardwareCPU::Inst_MaxAlloc, nInstFlag::DEFAULT, "Allocate maximum allowed space"),
  tInstLibEntry<tMethod>("h-alloc-mw", &cHardwareCPU::Inst_MaxAllocMoveWriteHead),
  tInstLibEntry<tMethod>("h-divide", &cHardwareCPU::Inst_HeadDivide, nInstFlag::DEFAULT | nInstFlag::STALL, "Divide code between read and write heads."),
  tInstLibEntry<tMethod>("h-divide1RS", &cHardwareCPU::Inst_HeadDivide1RS, nInstFlag::STALL, "Divide code between read and write heads, at most one mutation on divide, resample if reverted."),
  tInstLibEntry<tMethod>("h-divide2RS", &cHardwareCPU::Inst_HeadDivide2RS, nInstFlag::STALL, "Divide code between read and write heads, at most two mutations on divide, resample if reverted."),
  tInstLibEntry<tMethod>("h-divideRS", &cHardwareCPU::Inst_HeadDivideRS, nInstFlag::STALL, "Divide code between read and write heads, resample if reverted."),
  tInstLibEntry<tMethod>("h-read", &cHardwareCPU::Inst_HeadRead),
  tInstLibEntry<tMethod>("h-write", &cHardwareCPU::Inst_HeadWrite),
  tInstLibEntry<tMethod>("h-copy", &cHardwareCPU::Inst_HeadCopy, nInstFlag::DEFAULT, "Copy from read-head to write-head; advance both"),
  tInstLibEntry<tMethod>("h-search", &cHardwareCPU::Inst_HeadSearch, nInstFlag::DEFAULT, "Find complement template and make with flow head"),
  tInstLibEntry<tMethod>("h-search-direct", &cHardwareCPU::Inst_HeadSearchDirect, 0, "Find direct template and move the flow head"),
  tInstLibEntry<tMethod>("h-push", &cHardwareCPU::Inst_HeadPush),
  tInstLibEntry<tMethod>("h-pop", &cHardwareCPU::Inst_HeadPop),
  tInstLibEntry<tMethod>("set-head", &cHardwareCPU::Inst_SetHead),
  tInstLibEntry<tMethod>("adv-head", &cHardwareCPU::Inst_AdvanceHead),
  tInstLibEntry<tMethod>("mov-head", &cHardwareCPU::Inst_MoveHead, nInstFlag::DEFAULT, "Move head ?IP? to the flow head"),
  tInstLibEntry<tMethod>("jmp-head", &cHardwareCPU::Inst_JumpHead, nInstFlag::DEFAULT, "Move head ?IP? by amount in CX register; CX = old pos."),
  tInstLibEntry<tMethod>("get-head", &cHardwareCPU::Inst_GetHead, nInstFlag::DEFAULT, "Copy the position of the ?IP? head into CX"),
  tInstLibEntry<tMethod>("if-label", &cHardwareCPU::Inst_IfLabel, nInstFlag::DEFAULT, "Execute next if we copied complement of attached label"),
  tInstLibEntry<tMethod>("if-label-direct", &cHardwareCPU::Inst_IfLabelDirect, nInstFlag::DEFAULT, "Execute next if we copied direct match of the attached label"),
  tInstLibEntry<tMethod>("if-label2", &cHardwareCPU::Inst_IfLabel2, 0, "If copied label compl., exec next inst; else SKIP W/NOPS"),
  tInstLibEntry<tMethod>("set-flow", &cHardwareCPU::Inst_SetFlow, nInstFlag::DEFAULT, "Set flow-head to position in ?CX?"),

  tInstLibEntry<tMethod>("res-mov-head", &cHardwareCPU::Inst_ResMoveHead, nInstFlag::STALL, "Move head ?IP? to the flow head depending on resource level"),
  tInstLibEntry<tMethod>("res-jmp-head", &cHardwareCPU::Inst_ResJumpHead, nInstFlag::STALL, "Move head ?IP? by amount in CX register depending on resource level; CX = old pos."),

  tInstLibEntry<tMethod>("h-copy2", &cHardwareCPU::Inst_HeadCopy2),
  tInstLibEntry<tMethod>("h-copy3", &cHardwareCPU::Inst_HeadCopy3),
  tInstLibEntry<tMethod>("h-copy4", &cHardwareCPU::Inst_HeadCopy4),
  tInstLibEntry<tMethod>("h-copy5", &cHardwareCPU::Inst_HeadCopy5),
  tInstLibEntry<tMethod>("h-copy6", &cHardwareCPU::Inst_HeadCopy6),
  tInstLibEntry<tMethod>("h-copy7", &cHardwareCPU::Inst_HeadCopy7),
  tInstLibEntry<tMethod>("h-copy8", &cHardwareCPU::Inst_HeadCopy8),
  tInstLibEntry<tMethod>("h-copy9", &cHardwareCPU::Inst_HeadCopy9),
  tInstLibEntry<tMethod>("h-copy10", &cHardwareCPU::Inst_HeadCopy10),

  tInstLibEntry<tMethod>("divide-sex", &cHardwareCPU::Inst_HeadDivideSex, nInstFlag::STALL),
  tInstLibEntry<tMethod>("divide-asex", &cHardwareCPU::Inst_HeadDivideAsex, nInstFlag::STALL),

  tInstLibEntry<tMethod>("div-sex", &cHardwareCPU::Inst_HeadDivideSex, nInstFlag::STALL),
  tInstLibEntry<tMethod>("div-asex", &cHardwareCPU::Inst_HeadDivideAsex, nInstFlag::STALL),
  tInstLibEntry<tMethod>("div-asex-w", &cHardwareCPU::Inst_HeadDivideAsexWait, nInstFlag::STALL),
  tInstLibEntry<tMethod>("div-sex-MS", &cHardwareCPU::Inst_HeadDivideMateSelect, nInstFlag::STALL),

  tInstLibEntry<tMethod>("h-divide1", &cHardwareCPU::Inst_HeadDivide1, nInstFlag::STALL),
  tInstLibEntry<tMethod>("h-divide2", &cHardwareCPU::Inst_HeadDivide2, nInstFlag::STALL),
  tInstLibEntry<tMethod>("h-divide3", &cHardwareCPU::Inst_HeadDivide3, nInstFlag::STALL),
  tInstLibEntry<tMethod>("h-divide4", &cHardwareCPU::Inst_HeadDivide4, nInstFlag::STALL),
  tInstLibEntry<tMethod>("h-divide5", &cHardwareCPU::Inst_HeadDivide5, nInstFlag::STALL),
  tInstLibEntry<tMethod>("h-divide6", &cHardwareCPU::Inst_HeadDivide6, nInstFlag::STALL),
  tInstLibEntry<tMethod>("h-divide7", &cHardwareCPU::Inst_HeadDivide7, nInstFlag::STALL),
  tInstLibEntry<tMethod>("h-divide8", &cHardwareCPU::Inst_HeadDivide8, nInstFlag::STALL),
  tInstLibEntry<tMethod>("h-divide9", &cHardwareCPU::Inst_HeadDivide9, nInstFlag::STALL),
  tInstLibEntry<tMethod>("h-divide10", &cHardwareCPU::Inst_HeadDivide10, nInstFlag::STALL),
  tInstLibEntry<tMethod>("h-divide16", &cHardwareCPU::Inst_HeadDivide16, nInstFlag::STALL),
  tInstLibEntry<tMethod>("h-divide32", &cHardwareCPU::Inst_HeadDivide32, nInstFlag::STALL),
  tInstLibEntry<tMethod>("h-divide50", &cHardwareCPU::Inst_HeadDivide50, nInstFlag::STALL),
  tInstLibEntry<tMethod>("h-divide100", &cHardwareCPU::Inst_HeadDivide100, nInstFlag::STALL),
  tInstLibEntry<tMethod>("h-divide500", &cHardwareCPU::Inst_HeadDivide500, nInstFlag::STALL),
  tInstLibEntry<tMethod>("h-divide1000", &cHardwareCPU::Inst_HeadDivide1000, nInstFlag::STALL),
  tInstLibEntry<tMethod>("h-divide5000", &cHardwareCPU::Inst_HeadDivide5000, nInstFlag::STALL),
  tInstLibEntry<tMethod>("h-divide10000", &cHardwareCPU::Inst_HeadDivide10000, nInstFlag::STALL),
  tInstLibEntry<tMethod>("h-divide50000", &cHardwareCPU::Inst_HeadDivide50000, nInstFlag::STALL),
  tInstLibEntry<tMethod>("h-divide0.5", &cHardwareCPU::Inst_HeadDivide0_5, nInstFlag::STALL),
  tInstLibEntry<tMethod>("h-divide0.1", &cHardwareCPU::Inst_HeadDivide0_1, nInstFlag::STALL),
  tInstLibEntry<tMethod>("h-divide0.05", &cHardwareCPU::Inst_HeadDivide0_05,
      nInstFlag::STALL),
  tInstLibEntry<tMethod>("h-divide0.01", &cHardwareCPU::Inst_HeadDivide0_01,
      nInstFlag::STALL),
  tInstLibEntry<tMethod>("h-divide0.001", &cHardwareCPU::Inst_HeadDivide0_001,
      nInstFlag::STALL),

  // High-level instructions
  tInstLibEntry<tMethod>("repro_deme", &cHardwareCPU::Inst_ReproDeme, nInstFlag::
      STALL),
  tInstLibEntry<tMethod>("repro", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-sex", &cHardwareCPU::Inst_ReproSex, nInstFlag::
      STALL),
  tInstLibEntry<tMethod>("repro-A", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-B", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-C", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-D", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-E", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-F", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-G", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-H", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-I", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-J", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-K", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-L", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-M", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-N", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-O", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-P", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-Q", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-R", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-S", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-T", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-U", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-V", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-W", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-X", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-Y", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),
  tInstLibEntry<tMethod>("repro-Z", &cHardwareCPU::Inst_Repro, nInstFlag::STALL),

  tInstLibEntry<tMethod>("put-repro", &cHardwareCPU::Inst_TaskPutRepro, nInstFlag
      ::STALL),
  tInstLibEntry<tMethod>("metabolize", &cHardwareCPU::Inst_TaskPutResetInputsRepro
      , nInstFlag::STALL),

  tInstLibEntry<tMethod>("sterilize", &cHardwareCPU::Inst_Sterilize),

  tInstLibEntry<tMethod>("spawn-deme", &cHardwareCPU::Inst_SpawnDeme, nInstFlag::
      STALL),

  // Suicide
  tInstLibEntry<tMethod>("kazi", &cHardwareCPU::Inst_Kazi, nInstFlag::STALL),
  tInstLibEntry<tMethod>("kazi5", &cHardwareCPU::Inst_Kazi5, nInstFlag::STALL),
  tInstLibEntry<tMethod>("die", &cHardwareCPU::Inst_Die, nInstFlag::STALL),
  tInstLibEntry<tMethod>("suicide", &cHardwareCPU::Inst_Suicide, nInstFlag::STALL)
      ,
  tInstLibEntry<tMethod>("relinquishEnergyToFutureDeme", &cHardwareCPU::
      Inst_RelinquishEnergyToFutureDeme, nInstFlag::STALL),
  tInstLibEntry<tMethod>("relinquishEnergyToNeighborOrganisms", &cHardwareCPU::
      Inst_RelinquishEnergyToNeighborOrganisms, nInstFlag::STALL),
  tInstLibEntry<tMethod>("relinquishEnergyToOrganismsInDeme", &cHardwareCPU::
      Inst_RelinquishEnergyToOrganismsInDeme, nInstFlag::STALL),

  // Energy level detection
  tInstLibEntry<tMethod>("if-energy-low", &cHardwareCPU::Inst_IfEnergyLow,
      nInstFlag::STALL),
  tInstLibEntry<tMethod>("if-energy-not-low", &cHardwareCPU::Inst_IfEnergyNotLow,
      nInstFlag::STALL),
  tInstLibEntry<tMethod>("if-faced-energy-low", &cHardwareCPU::
      Inst_IfFacedEnergyLow, nInstFlag::STALL),
  tInstLibEntry<tMethod>("if-faced-energy-not-low", &cHardwareCPU::
      Inst_IfFacedEnergyNotLow, nInstFlag::STALL),
  tInstLibEntry<tMethod>("if-energy-high", &cHardwareCPU::Inst_IfEnergyHigh,
      nInstFlag::STALL),
  tInstLibEntry<tMethod>("if-energy-not-high", &cHardwareCPU::Inst_IfEnergyNotHigh
      , nInstFlag::STALL),
  tInstLibEntry<tMethod>("if-faced-energy-high", &cHardwareCPU::
      Inst_IfFacedEnergyHigh, nInstFlag::STALL),
  tInstLibEntry<tMethod>("if-faced-energy-not-high", &cHardwareCPU::
      Inst_IfFacedEnergyNotHigh, nInstFlag::STALL),
  tInstLibEntry<tMethod>("if-energy-med", &cHardwareCPU::Inst_IfEnergyMed,
      nInstFlag::STALL),
  tInstLibEntry<tMethod>("if-faced-energy-med", &cHardwareCPU::
      Inst_IfFacedEnergyMed, nInstFlag::STALL),
  tInstLibEntry<tMethod>("if-faced-energy-less", &cHardwareCPU::
      Inst_IfFacedEnergyLess, nInstFlag::STALL),
  tInstLibEntry<tMethod>("if-faced-energy-more", &cHardwareCPU::
      Inst_IfFacedEnergyMore, nInstFlag::STALL),
  tInstLibEntry<tMethod>("if-energy-in-buffer", &cHardwareCPU::
      Inst_IfEnergyInBuffer, nInstFlag::STALL),
  tInstLibEntry<tMethod>("if-energy-not-in-buffer", &cHardwareCPU::
      Inst_IfEnergyNotInBuffer, nInstFlag::STALL),
  tInstLibEntry<tMethod>("get-energy-level", &cHardwareCPU::Inst_GetEnergyLevel,
      nInstFlag::STALL),
  tInstLibEntry<tMethod>("get-faced-energy-level", &cHardwareCPU::
      Inst_GetFacedEnergyLevel, nInstFlag::STALL),
  tInstLibEntry<tMethod>("if-faced-request-on", &cHardwareCPU::
      Inst_IfFacedEnergyRequestOn, nInstFlag::STALL),
  tInstLibEntry<tMethod>("if-faced-request-off", &cHardwareCPU::
      Inst_IfFacedEnergyRequestOff, nInstFlag::STALL),
  tInstLibEntry<tMethod>("get-energy-request-status", &cHardwareCPU::
      Inst_GetEnergyRequestStatus, nInstFlag::STALL),
  tInstLibEntry<tMethod>("get-faced-energy-request-status", &cHardwareCPU::
      Inst_GetFacedEnergyRequestStatus, nInstFlag::STALL),

  // Sleep and time
  tInstLibEntry<tMethod>("sleep", &cHardwareCPU::Inst_Sleep, nInstFlag::STALL),
  tInstLibEntry<tMethod>("sleep1", &cHardwareCPU::Inst_Sleep, nInstFlag::STALL),
  tInstLibEntry<tMethod>("sleep2", &cHardwareCPU::Inst_Sleep, nInstFlag::STALL),
  tInstLibEntry<tMethod>("sleep3", &cHardwareCPU::Inst_Sleep, nInstFlag::STALL),
  tInstLibEntry<tMethod>("sleep4", &cHardwareCPU::Inst_Sleep, nInstFlag::STALL),
  tInstLibEntry<tMethod>("time", &cHardwareCPU::Inst_GetUpdate, nInstFlag::STALL),

  // Promoter Model
  tInstLibEntry<tMethod>("promoter", &cHardwareCPU::Inst_Promoter),
  tInstLibEntry<tMethod>("terminate", &cHardwareCPU::Inst_Terminate),
  tInstLibEntry<tMethod>("promoter", &cHardwareCPU::Inst_Promoter),
  tInstLibEntry<tMethod>("terminate", &cHardwareCPU::Inst_Terminate),
  tInstLibEntry<tMethod>("regulate", &cHardwareCPU::Inst_Regulate),
  tInstLibEntry<tMethod>("regulate-sp", &cHardwareCPU::
      Inst_RegulateSpecificPromoters),
  tInstLibEntry<tMethod>("s-regulate", &cHardwareCPU::Inst_SenseRegulate),
  tInstLibEntry<tMethod>("numberate", &cHardwareCPU::Inst_Numberate),
  tInstLibEntry<tMethod>("numberate-24", &cHardwareCPU::Inst_Numberate24),

  // Bit Consensus
  tInstLibEntry<tMethod>("bit-cons", &cHardwareCPU::Inst_BitConsensus),
  tInstLibEntry<tMethod>("bit-cons-24", &cHardwareCPU::Inst_BitConsensus24),
  tInstLibEntry<tMethod>("if-cons", &cHardwareCPU::Inst_IfConsensus, 0, "Execute
      next instruction if ?BX? in consensus, else skip it"),
  tInstLibEntry<tMethod>("if-cons-24", &cHardwareCPU::Inst_IfConsensus24, 0, "
      Execute next instruction if ?BX[0:23]? in consensus , else skip it"),
  tInstLibEntry<tMethod>("if-less-cons", &cHardwareCPU::Inst_IfLessConsensus, 0, "
      Execute next instruction if Count(?BX?) < Count(?CX?), else skip it"),
  tInstLibEntry<tMethod>("if-less-cons-24", &cHardwareCPU::Inst_IfLessConsensus24,
       0, "Execute next instruction if Count(?BX[0:23]?) < Count(?CX[0:23]?), else
       skip it"),

  // Bit Masking (higher order bit masking is possible,
  // just add the instructions if needed)
  tInstLibEntry<tMethod>("mask-signbit", &cHardwareCPU::Inst_MaskSignBit),
  tInstLibEntry<tMethod>("maskoff-lower16bits", &cHardwareCPU::
      Inst_MaskOffLower16Bits),
  tInstLibEntry<tMethod>("maskoff-lower16bits-defaultAX", &cHardwareCPU::
      Inst_MaskOffLower16Bits_defaultAX),
  tInstLibEntry<tMethod>("maskoff-lower15bits", &cHardwareCPU::
      Inst_MaskOffLower15Bits),
  tInstLibEntry<tMethod>("maskoff-lower15bits-defaultAX", &cHardwareCPU::
      Inst_MaskOffLower15Bits_defaultAX),
  tInstLibEntry<tMethod>("maskoff-lower14bits", &cHardwareCPU::
      Inst_MaskOffLower14Bits),
  tInstLibEntry<tMethod>("maskoff-lower14bits-defaultAX", &cHardwareCPU::
      Inst_MaskOffLower14Bits_defaultAX),
  tInstLibEntry<tMethod>("maskoff-lower13bits", &cHardwareCPU::
      Inst_MaskOffLower13Bits),
  tInstLibEntry<tMethod>("maskoff-lower13bits-defaultAX", &cHardwareCPU::
      Inst_MaskOffLower13Bits_defaultAX),
  tInstLibEntry<tMethod>("maskoff-lower12bits", &cHardwareCPU::
      Inst_MaskOffLower12Bits),
  tInstLibEntry<tMethod>("maskoff-lower12bits-defaultAX", &cHardwareCPU::
      Inst_MaskOffLower12Bits_defaultAX),
  tInstLibEntry<tMethod>("maskoff-lower8bits", &cHardwareCPU::
      Inst_MaskOffLower8Bits),
  tInstLibEntry<tMethod>("maskoff-lower8bits-defaultAX", &cHardwareCPU::
      Inst_MaskOffLower8Bits_defaultAX),
  tInstLibEntry<tMethod>("maskoff-lower4bits", &cHardwareCPU::
      Inst_MaskOffLower4Bits),
  tInstLibEntry<tMethod>("maskoff-lower4bits-defaultAX", &cHardwareCPU::
      Inst_MaskOffLower4Bits_defaultAX),

  // Energy usage
  tInstLibEntry<tMethod>("double-energy-usage", &cHardwareCPU::
      Inst_DoubleEnergyUsage, nInstFlag::STALL),
  tInstLibEntry<tMethod>("halve-energy-usage", &cHardwareCPU::
      Inst_HalveEnergyUsage, nInstFlag::STALL),
  tInstLibEntry<tMethod>("default-energy-usage", &cHardwareCPU::
      Inst_DefaultEnergyUsage, nInstFlag::STALL),

  // Messaging
  tInstLibEntry<tMethod>("send-msg", &cHardwareCPU::Inst_SendMessage, nInstFlag::
      STALL),
  tInstLibEntry<tMethod>("retrieve-msg", &cHardwareCPU::Inst_RetrieveMessage,
      nInstFlag::STALL),
  tInstLibEntry<tMethod>("bcast1", &cHardwareCPU::Inst_Broadcast1, nInstFlag::
      STALL),
  tInstLibEntry<tMethod>("bcast2", &cHardwareCPU::Inst_Broadcast2, nInstFlag::
      STALL),
  tInstLibEntry<tMethod>("bcast4", &cHardwareCPU::Inst_Broadcast4, nInstFlag::
      STALL),
  tInstLibEntry<tMethod>("bcast8", &cHardwareCPU::Inst_Broadcast8, nInstFlag::
      STALL),

  // Alarms
  tInstLibEntry<tMethod>("send-alarm-msg-local", &cHardwareCPU::
      Inst_Alarm_MSG_local, nInstFlag::STALL),
  tInstLibEntry<tMethod>("send-alarm-msg-multihop", &cHardwareCPU::
      Inst_Alarm_MSG_multihop, nInstFlag::STALL),
  tInstLibEntry<tMethod>("send-alarm-msg-bit-cons24-local", &cHardwareCPU::
      Inst_Alarm_MSG_Bit_Cons24_local, nInstFlag::STALL),
  tInstLibEntry<tMethod>("send-alarm-msg-bit-cons24-multihop", &cHardwareCPU::
      Inst_Alarm_MSG_Bit_Cons24_multihop, nInstFlag::STALL),
  tInstLibEntry<tMethod>("alarm-label-high", &cHardwareCPU::Inst_Alarm_Label),
  tInstLibEntry<tMethod>("alarm-label-low", &cHardwareCPU::Inst_Alarm_Label),

  // Interrupt
  tInstLibEntry<tMethod>("send-msg-interrupt-type0", &cHardwareCPU::
      Inst_SendMessageInterruptType0, nInstFlag::STALL),
  tInstLibEntry<tMethod>("send-msg-interrupt-type1", &cHardwareCPU::
      Inst_SendMessageInterruptType1, nInstFlag::STALL),
  tInstLibEntry<tMethod>("send-msg-interrupt-type2", &cHardwareCPU::
      Inst_SendMessageInterruptType2, nInstFlag::STALL),
  tInstLibEntry<tMethod>("send-msg-interrupt-type3", &cHardwareCPU::
      Inst_SendMessageInterruptType3, nInstFlag::STALL),
  tInstLibEntry<tMethod>("send-msg-interrupt-type4", &cHardwareCPU::
      Inst_SendMessageInterruptType4, nInstFlag::STALL),
  tInstLibEntry<tMethod>("send-msg-interrupt-type5", &cHardwareCPU::
      Inst_SendMessageInterruptType5, nInstFlag::STALL),
  tInstLibEntry<tMethod>("msg-handler-type0", &cHardwareCPU::Inst_START_Handler),
  tInstLibEntry<tMethod>("msg-handler-type1", &cHardwareCPU::Inst_START_Handler),
  tInstLibEntry<tMethod>("msg-handler-type2", &cHardwareCPU::Inst_START_Handler),
  tInstLibEntry<tMethod>("msg-handler-type3", &cHardwareCPU::Inst_START_Handler),
  tInstLibEntry<tMethod>("msg-handler-type4", &cHardwareCPU::Inst_START_Handler),
  tInstLibEntry<tMethod>("msg-handler-type5", &cHardwareCPU::Inst_START_Handler),
  tInstLibEntry<tMethod>("moved-handler", &cHardwareCPU::Inst_START_Handler),
  tInstLibEntry<tMethod>("end-handler", &cHardwareCPU::Inst_End_Handler),

  // Placebo instructions
  tInstLibEntry<tMethod>("skip", &cHardwareCPU::Inst_Skip),

  // @BDC additions for pheromones
  tInstLibEntry<tMethod>("phero-on", &cHardwareCPU::Inst_PheroOn),
  tInstLibEntry<tMethod>("phero-off", &cHardwareCPU::Inst_PheroOff),
  tInstLibEntry<tMethod>("pherotoggle", &cHardwareCPU::Inst_PheroToggle),
  tInstLibEntry<tMethod>("sense-target", &cHardwareCPU::Inst_SenseTarget),
  tInstLibEntry<tMethod>("sense-target-faced", &cHardwareCPU::
      Inst_SenseTargetFaced),
  tInstLibEntry<tMethod>("sensef", &cHardwareCPU::Inst_SenseLog2Facing),
  tInstLibEntry<tMethod>("sensef-unit", &cHardwareCPU::Inst_SenseUnitFacing),
  tInstLibEntry<tMethod>("sensef-m100", &cHardwareCPU::Inst_SenseMult100Facing),
  tInstLibEntry<tMethod>("sense-pheromone", &cHardwareCPU::Inst_SensePheromone),
  tInstLibEntry<tMethod>("sense-pheromone-faced", &cHardwareCPU::
      Inst_SensePheromoneFaced),
  tInstLibEntry<tMethod>("sense-pheromone-inDemeGlobal", &cHardwareCPU::
      Inst_SensePheromoneInDemeGlobal),
  tInstLibEntry<tMethod>("sense-pheromone-global", &cHardwareCPU::
      Inst_SensePheromoneGlobal),
  tInstLibEntry<tMethod>("sense-pheromone-global-defaultAX", &cHardwareCPU::
      Inst_SensePheromoneGlobal_defaultAX),
  tInstLibEntry<tMethod>("exploit", &cHardwareCPU::Inst_Exploit, nInstFlag::STALL)
      ,
  tInstLibEntry<tMethod>("exploit-forward5", &cHardwareCPU::Inst_ExploitForward5,
      nInstFlag::STALL),
  tInstLibEntry<tMethod>("exploit-forward3", &cHardwareCPU::Inst_ExploitForward3,
      nInstFlag::STALL),
  tInstLibEntry<tMethod>("explore", &cHardwareCPU::Inst_Explore, nInstFlag::STALL)
      ,
  tInstLibEntry<tMethod>("movetarget", &cHardwareCPU::Inst_MoveTarget, nInstFlag::
      STALL),
  tInstLibEntry<tMethod>("movetarget-forward5", &cHardwareCPU::
      Inst_MoveTargetForward5, nInstFlag::STALL),
  tInstLibEntry<tMethod>("movetarget-forward3", &cHardwareCPU::
      Inst_MoveTargetForward3, nInstFlag::STALL),
  tInstLibEntry<tMethod>("supermove", &cHardwareCPU::Inst_SuperMove, nInstFlag::
      STALL),
  tInstLibEntry<tMethod>("if-target", &cHardwareCPU::Inst_IfTarget),
  tInstLibEntry<tMethod>("if-not-target", &cHardwareCPU::Inst_IfNotTarget),
  tInstLibEntry<tMethod>("if-pheromone", &cHardwareCPU::Inst_IfPheromone),
  tInstLibEntry<tMethod>("if-not-pheromone", &cHardwareCPU::Inst_IfNotPheromone),
  tInstLibEntry<tMethod>("drop-pheromone", &cHardwareCPU::Inst_DropPheromone,
      nInstFlag::STALL),

  // Opinion instructions.
  // These are STALLs because opinions are only relevant with respect to time.
  tInstLibEntry<tMethod>("set-opinion", &cHardwareCPU::Inst_SetOpinion, nInstFlag
      ::STALL),
  tInstLibEntry<tMethod>("get-opinion", &cHardwareCPU::Inst_GetOpinion, nInstFlag
      ::STALL),
  tInstLibEntry<tMethod>("get-opinionOnly", &cHardwareCPU::
      Inst_GetOpinionOnly_ZeroIfNone, nInstFlag::STALL),
  tInstLibEntry<tMethod>("clear-opinion", &cHardwareCPU::Inst_ClearOpinion,
      nInstFlag::STALL),
  tInstLibEntry<tMethod>("if-opinion-set", &cHardwareCPU::Inst_IfOpinionSet,
      nInstFlag::STALL),
  tInstLibEntry<tMethod>("if-opinion-notset", &cHardwareCPU::Inst_IfOpinionNotSet,
       nInstFlag::STALL),

  // Data collection
  tInstLibEntry<tMethod>("if-cell-data-changed", &cHardwareCPU::
      Inst_IfCellDataChanged, nInstFlag::STALL),
  tInstLibEntry<tMethod>("collect-cell-data", &cHardwareCPU::Inst_CollectCellData,
       nInstFlag::STALL),
  tInstLibEntry<tMethod>("kill-cell-event", &cHardwareCPU::Inst_KillCellEvent,
      nInstFlag::STALL),
  tInstLibEntry<tMethod>("kill-faced-cell-event", &cHardwareCPU::
      Inst_KillFacedCellEvent, nInstFlag::STALL),
  tInstLibEntry<tMethod>("collect-cell-data-and-kill-event", &cHardwareCPU::
      Inst_CollectCellDataAndKillEvent, nInstFlag::STALL),
  tInstLibEntry<tMethod>("read-cell-data", &cHardwareCPU::Inst_ReadCellData),
  tInstLibEntry<tMethod>("read-faced-cell-data", &cHardwareCPU::
      Inst_ReadFacedCellData, nInstFlag::STALL),
  tInstLibEntry<tMethod>("mark-cell-with-id", &cHardwareCPU::Inst_MarkCellWithID),
  tInstLibEntry<tMethod>("get-id", &cHardwareCPU::Inst_GetID),

  // Synchronization
  tInstLibEntry<tMethod>("flash", &cHardwareCPU::Inst_Flash, nInstFlag::STALL),
  tInstLibEntry<tMethod>("if-recvd-flash", &cHardwareCPU::Inst_IfRecvdFlash,
      nInstFlag::STALL),
  tInstLibEntry<tMethod>("flash-info", &cHardwareCPU::Inst_FlashInfo, nInstFlag::
      STALL),
  tInstLibEntry<tMethod>("flash-info-b", &cHardwareCPU::Inst_FlashInfoB, nInstFlag
      ::STALL),
  tInstLibEntry<tMethod>("reset-flash-info", &cHardwareCPU::Inst_ResetFlashInfo,
      nInstFlag::STALL),
  tInstLibEntry<tMethod>("hard-reset", &cHardwareCPU::Inst_HardReset, nInstFlag::
      STALL),
  tInstLibEntry<tMethod>("get-cycles", &cHardwareCPU::Inst_GetCycles, nInstFlag::
      STALL),

  // Neighborhood-sensing instructions
  tInstLibEntry<tMethod>("get-neighborhood", &cHardwareCPU::Inst_GetNeighborhood, nInstFlag::STALL),
  tInstLibEntry<tMethod>("if-neighborhood-changed", &cHardwareCPU::Inst_IfNeighborhoodChanged, nInstFlag::STALL),

  // Reputation instructions
  tInstLibEntry<tMethod>("donate-frm", &cHardwareCPU::Inst_DonateFacingRawMaterials, nInstFlag::STALL),
  tInstLibEntry<tMethod>("donate-spec", &cHardwareCPU::Inst_DonateFacingRawMaterialsOtherSpecies, nInstFlag::STALL),
  tInstLibEntry<tMethod>("donate-if-donor", &cHardwareCPU::Inst_DonateIfDonor, nInstFlag::STALL),
  tInstLibEntry<tMethod>("donate-string", &cHardwareCPU::Inst_DonateFacingString, nInstFlag::STALL),

  tInstLibEntry<tMethod>("get-neighbors-reputation", &cHardwareCPU::Inst_GetNeighborsReputation, nInstFlag::STALL),
  tInstLibEntry<tMethod>("get-reputation", &cHardwareCPU::Inst_GetReputation, nInstFlag::STALL),
  tInstLibEntry<tMethod>("get-raw-mat-amount", &cHardwareCPU::Inst_GetAmountOfRawMaterials, nInstFlag::STALL),
  tInstLibEntry<tMethod>("get-other-raw-mat-amount", &cHardwareCPU::Inst_GetAmountOfOtherRawMaterials, nInstFlag::STALL),
  tInstLibEntry<tMethod>("pose", &cHardwareCPU::Inst_Pose, nInstFlag::STALL),
  tInstLibEntry<tMethod>("rotate-to-rep", &cHardwareCPU::Inst_RotateToGreatestReputation, nInstFlag::STALL),
  tInstLibEntry<tMethod>("rotate-to-rep-and-donate", &cHardwareCPU::Inst_RotateToGreatestReputationAndDonate, nInstFlag::STALL),
  tInstLibEntry<tMethod>("rotate-to-rep-tag", &cHardwareCPU::Inst_RotateToGreatestReputationWithDifferentTag, nInstFlag::STALL),
  tInstLibEntry<tMethod>("rotate-to-rep-lineage", &cHardwareCPU::Inst_RotateToGreatestReputationWithDifferentLineage, nInstFlag::STALL),
  tInstLibEntry<tMethod>("rotate-to-tag", &cHardwareCPU::Inst_RotateToDifferentTag, nInstFlag::STALL),
  tInstLibEntry<tMethod>("if-donor", &cHardwareCPU::Inst_IfDonor, nInstFlag::STALL),
  tInstLibEntry<tMethod>("prod-string", &cHardwareCPU::Inst_ProduceString, nInstFlag::STALL),

  // Group formation instructions
  tInstLibEntry<tMethod>("join-group", &cHardwareCPU::Inst_JoinGroup, nInstFlag::STALL),
  tInstLibEntry<tMethod>("orgs-in-my-group", &cHardwareCPU::Inst_NumberOrgsInMyGroup, nInstFlag::STALL),
  tInstLibEntry<tMethod>("orgs-in-group", &cHardwareCPU::Inst_NumberOrgsInGroup, nInstFlag::STALL),

  // Network creation instructions
  tInstLibEntry<tMethod>("create-link-facing", &cHardwareCPU::Inst_CreateLinkByFacing, nInstFlag::STALL),
  tInstLibEntry<tMethod>("create-link-xy", &cHardwareCPU::Inst_CreateLinkByXY, nInstFlag::STALL),
  tInstLibEntry<tMethod>("create-link-index", &cHardwareCPU::Inst_CreateLinkByIndex, nInstFlag::STALL),

  // Must always be the last instruction in the array
  tInstLibEntry<tMethod>("NULL", &cHardwareCPU::Inst_Nop, 0, "True no-operation instruction: does nothing"),
};
```

Appendix E

Configuration File (Excerpt)

Three relevant groups of configurable variables are listed below, with the comments according to the original wording and terminology. Several unused fields are also included in the listing for the sake of completeness.

Listing E.1: ARCH_GROUP

```
### ARCH_GROUP ###
# Architecture Variables
WORLD_X 60  # Width of the Avida world
WORLD_Y 60  # Height of the Avida world
WORLD_Z 1   # Depth of the Avida world
WORLD_GEOMETRY 2  # 1 = Bounded Grid
   # 2 = Torus
   # 3 = Clique
   # 4 = Hexagonal grid
   # 5 = Partial
   # 6 = Lattice
   # 7 = Random connected
   # 8 = Scale-free
SCALE_FREE_M 3 # Number of connections to add per cell when using a scale-free geometry.
SCALE_FREE_ALPHA 1.0 # Attachment power (1=linear).
SCALE_FREE_ZERO_APPEAL 0.0 # Appeal of cells with zero connections.
RANDOM_SEED 17676175 # Random number seed (0 for based on time)
#RANDOM_SEED 0 # Random number seed (0 for based on time)
HARDWARE_TYPE 0   # 0 = Original CPUs
   # 1 = New SMT CPUs
   # 2 = Transitional SMT
   # 3 = Experimental CPU
   # 4 = Gene Expression CPU
SPECULATIVE 1  # Enable speculative execution
TRACE_EXECUTION 0 # Trace the execution of all organisms in the population (default=OFF, SLOW!)
IO_EXPIRE 1 # Is the expiration functionality of ’-expire’ I/O instructions enabled?
```

Listing E.2: MUTATION_GROUP

```
### MUTATION_GROUP ###
# Mutations
POINT_MUT_PROB 0.0   # Mutation rate (per-location per update) #default=0.0
COPY_MUT_PROB 0.0001753 # Mutation rate (per copy) for von neumann ancestors
#COPY_MUT_PROB 0.0   # Mutation rate (per copy) for von neumann ancestors
COPY_INS_PROB 0.0 # Insertion rate (per copy)
COPY_DEL_PROB 0.0 # Deletion rate (per copy)
COPY_UNIFORM_PROB 0.0   # Uniform mutation probability (per copy)
```

```
   # - Randomly applies any of the three classes of mutations (ins, del, point).
COPY_SLIP_PROB 0.0   # Slip rate (per copy)
DIV_MUT_PROB 0.0  # Mutation rate (per site, applied on divide)
DIV_INS_PROB 0.0  # Insertion rate (per site, applied on divide)
DIV_DEL_PROB 0.0  # Deletion rate (per site, applied on divide)
DIV_UNIFORM_PROB 0.0 # Uniform mutation probability (per site, applied on divide)
   # - Randomly applies any of the three classes of mutations (ins, del, point).
DIV_SLIP_PROB 0.0 # Slip rate (per site, applied on divide)
DIVIDE_MUT_PROB 0.0  # Mutation rate (max one, per divide)
DIVIDE_INS_PROB 0.0  # Insertion rate (max one, per divide)
DIVIDE_DEL_PROB 0.0  # Deletion rate (max one, per divide)
DIVIDE_SLIP_PROB 0.0 # Slip rate (per divide) - creates large deletions/duplications
DIVIDE_POISSON_MUT_MEAN 0.0   # Mutation rate (Poisson distributed, per divide)
DIVIDE_POISSON_INS_MEAN 0.0   # Insertion rate (Poisson distributed, per divide)
DIVIDE_POISSON_DEL_MEAN 0.0   # Deletion rate (Poisson distributed, per divide)
DIVIDE_POISSON_SLIP_MEAN 0.0  # Slip rate (Poisson distributed, per divide)
DIVIDE_UNIFORM_PROB 0.0 # Uniform mutation probability (per divide)
   # - Randomly applies any of the three classes of mutations (ins, del, point).
DEATH_PROB 0.0 # Death rate (parent organism, per divide)
INJECT_INS_PROB 0.0  # Insertion rate (per site, applied on inject)
INJECT_DEL_PROB 0.0  # Deletion rate (per site, applied on inject)
INJECT_MUT_PROB 0.0  # Mutation rate (per site, applied on inject)
SLIP_FILL_MODE 0  # Fill insertions from slip mutations with 0=duplication, 1=nop-X, 2=random, 3=scrambled, 4=nop-C
SLIP_COPY_MODE 0  # How to handle 'on-copy' slip mutations:
   # 0 = actual read head slip
   # 1 = instant large mutation (obeys slip mode)
PARENT_MUT_PROB 0.0  # Per-site, in parent, on divide
SPECIAL_MUT_LINE -1  # If this is >= 0, ONLY this line is mutated
META_COPY_MUT 0.0 # Prob. of copy mutation rate changing (per gen)
META_STD_DEV 0.0  # Standard deviation of meta mutation size.
MUT_RATE_SOURCE 1 # 1 = Mutation rates determined by environment.
   # 2 = Mutation rates inherited from parent.
MIGRATION_RATE 0.0   # Uniform probability of offspring migrating to a new deme.
```

Listing E.3: REPRODUCTION_GROUP

```
### REPRODUCTION_GROUP ###
# Birth and Death
BIRTH_METHOD 1 # Which organism should be replaced on birth?
   # 0 = Random organism in neighborhood #default
   # 1 = Oldest in neighborhood
   # 2 = Largest Age/Merit in neighborhood
   # 3 = None (use only empty cells in neighborhood)
   # 4 = Random from population (Mass Action)
   # 5 = Oldest in entire population
   # 6 = Random within deme
   # 7 = Organism faced by parent
   # 8 = Next grid cell (id+1)
   # 9 = Largest energy used in entire population
   # 10 = Largest energy used in neighborhood
   # 11 = Local neighborhood dispersal
PREFER_EMPTY 1 # Give empty cells preference in offsping placement?
ALLOW_PARENT 1 # Allow births to replace the parent organism?
DISPERSAL_RATE 0.0   # Rate of dispersal under birth method 11
   # (poisson distributed random connection list hops)
DEATH_METHOD 0
   # 0 = Never die of old age.
   # 1 = Die when inst executed = AGE_LIMIT (+deviation)
   # 2 = Die when inst executed = length*AGE_LIMIT (+dev) #default
AGE_LIMIT 20   # Modifies DEATH_METHOD
AGE_DEVIATION 0   # Creates a distribution around AGE_LIMIT
```

```
ALLOC_METHOD 0 # (Orignal CPU Only)
   # 0 = Allocated space is set to default instruction.
   # 1 = Set to section of dead genome (Necrophilia)
   # 2 = Allocated space is set to random instruction.
DIVIDE_METHOD 1
   # 0 = Divide leaves state of mother untouched.
   # 1 = Divide resets state of mother
   # (after the divide, we have 2 children)
   # 2 = Divide resets state of current thread only
   # (does not touch possible parasite threads)
   # 3 = Divide resets mother stats, but not state.
   # 4 = 3 + child inherits mother registers and stack values.
EPIGENETIC_METHOD 0  # Inheritance of state information other than genome
   # 0 = none
   # 1 = offspring inherits registers and stacks of first thread
   # 2 = parent maintains registers and stacks of first thread
   # 3 = offspring and parent keep state information
INJECT_METHOD 0
   # 0 = Leaves the parasite thread state untouched.
   # 1 = Resets the calling thread state on inject
GENERATION_INC_METHOD 1
   # 0 = Only the generation of the child is increased on divide.
   # 1 = Both the generation of the mother and child are increased on divide.
   # (good with DIVIDE_METHOD 1).
RESET_INPUTS_ON_DIVIDE 0
   # Reset environment inputs of parent upon successful divide.
REPRO_METHOD 1 # Replace existing organism: 1=yes
INHERIT_MULTI_THREAD_CLASSIFICATION 0
   # Inherit the parental classification of multithreaded
POPULATION_CAP 0  # Carrying capacity in number of organisms
```

Listing E.4: DIVIDE_GROUP

```
### DIVIDE_GROUP ###
# Divide Restrictions
CHILD_SIZE_RANGE 2.0 # Maximal differential between child and parent sizes.
   # (Checked BEFORE mutations applied on divide.)
MIN_COPIED_LINES 0.5 # Code fraction which must be copied before divide.
MIN_EXE_LINES 0.45   # Code fraction which must be executed before divide.
   # 0.45 for the prototype ancestor.
   # (based on Phenome size / Gene size = 294/644 = 0.4565...)
MIN_GENOME_SIZE 0
   # Minimum number of instructions allowed in a genome. 0 = OFF
MAX_GENOME_SIZE 0
   # Maximum number of instructions allowed in a genome. 0 = OFF
REQUIRE_ALLOCATE 1   # (Original CPU Only) Require allocate before divide?
REQUIRED_TASK -1  # Task ID required for successful divide.
IMMUNITY_TASK -1  # Task providing immunity from the required task.
REQUIRED_REACTION -1 # Reaction ID required for successful divide.
IMMUNITY_REACTION -1 # Reaction ID that provides immunity for successful divide.
REQUIRED_BONUS 0.0   # Required bonus to divide.
REQUIRE_EXACT_COPY 0
   # Require offspring to be an exact copy (only divide mutations allowed).
REQUIRED_RESOURCE -1
   # Resource ID required for successful divide
   # (organism must have this resource in internal bins).
REQUIRED_RESOURCE_LEVEL 0.0
   # Level of resource required for successful divide (see REQUIRED_RESOURCE).
IMPLICIT_REPRO_BONUS 0
   # Call Inst_Repro to divide upon achieving this bonus. # 0 = OFF
IMPLICIT_REPRO_CPU_CYCLES 0
   # Call Inst_Repro after this many cpu cycles. 0 = OFF
IMPLICIT_REPRO_TIME 0
```

```
   # Call Inst_Repro after this time used. 0 = OFF
IMPLICIT_REPRO_END 0
   # Call Inst_Repro after executing the last instruction in the genome.
IMPLICIT_REPRO_ENERGY 0.0  # Call Inst_Repro if organism accumulates this amount of energy.
```

Listing E.5: TIME_GROUP

```
### TIME_GROUP ###
# Time Slicing
AVE_TIME_SLICE 30 # Ave number of insts per org per update
SLICING_METHOD 1  # 0 = CONSTANT: all organisms get default...
   # 1 = PROBABILISTIC: Run _prob_ proportional to merit. (default)
   # 2 = INTEGRATED: Perfectly integrated deterministic.
   # 3 = DemeProbabalistic, each deme gets the same number of CPU cycles, which are awarded probabalistically within each deme.
   # 4 = ProbDemeProbabalistic, each deme gets CPU cycles proportional to its living population size, which are awarded probabalistically within each deme.
   # 5 = CONSTANT BURST: all organisms get default, in SLICING_BURST_SIZE chunks
SLICING_BURST_SIZE 1 # Sets the scheduler burst size, when supported.
BASE_MERIT_METHOD 3  # 0 = Constant (merit independent of size)
   # 1 = Merit proportional to copied size
   # 2 = Merit prop. to executed size
   # 3 = Merit prop. to full size (default)
   # 4 = Merit prop. to min of executed or copied size
   # 5 = Merit prop. to sqrt of the minimum size
   # 6 = Merit prop. to num times MERIT_BONUS_INST is in genome.
BASE_CONST_MERIT 100 # Base merit when BASE_MERIT_METHOD set to 0
DEFAULT_BONUS 1.0 # Initial bonus before any tasks
MERIT_DEFAULT_BONUS 0   # Scale the merit of an offspring by this default bonus
   # rather than the accumulated bonus of the parent? 0 = OFF
MERIT_BONUS_INST 0   # in BASE_MERIT_METHOD 6, this sets which instruction counts
   # (-1 = none, 0 = First in INST_SET.)
MERIT_BONUS_EFFECT 0 # in BASE_MERIT_METHOD 6, this sets how much merit is earned
   # per instruction (-1 = penalty, 0 = no effect.)
MERIT_INC_APPLY_IMMEDIATE 0   # Should merit increases (above current) be applied immediately, or delayed until divide?
TASK_REFRACTORY_PERIOD 0.0 # Number of updates affected by refractory period
FITNESS_METHOD 0  # 0 = default, 1 = sigmoidal,
FITNESS_COEFF_1 1.0  # 1st FITNESS_METHOD parameter
FITNESS_COEFF_2 1.0  # 2nd FITNESS_METHOD parameter
FITNESS_VALLEY 0  # in BASE_MERIT_METHOD 6, this creates valleys from
   # FITNESS_VALLEY_START to FITNESS_VALLEY_STOP
   # (0 = OFF, 1 = ON)
FITNESS_VALLEY_START 0  # if FITNESS_VALLEY = 1, orgs with num_key_instructions
   # from FITNESS_VALLEY_START to FITNESS_VALLEY_STOP
   # get fitness 1 (lowest)
FITNESS_VALLEY_STOP 0   # if FITNESS_VALLEY = 1, orgs with num_key_instructions
   # from FITNESS_VALLEY_START to FITNESS_VALLEY_STOP
   # get fitness 1 (lowest)
MAX_CPU_THREADS 1 # Number of Threads a CPU can spawn
THREAD_SLICING_METHOD 0 # Formula for and organism's thread slicing
   # (num_threads-1) * THREAD_SLICING_METHOD + 1
   # 0 = One thread executed per time slice.
   # 1 = All threads executed each time slice.
NO_CPU_CYCLE_TIME 0  # Don't count each CPU cycle as part of gestation time
MAX_LABEL_EXE_SIZE 1 # Max nops marked as executed when labels are used
MERIT_GIVEN 0.0   # Fraction of merit donated with 'donate' command
MERIT_RECEIVED 0.0   # Multiplier of merit given with 'donate' command
MAX_DONATE_KIN_DIST -1  # Limit on distance of relation for donate; -1=no max
MAX_DONATE_EDIT_DIST -1 # Limit on genetic (edit) distance for donate; -1=no max
MIN_GB_DONATE_THRESHOLD -1 # threshold green beard donates only to orgs above this
```

```
   # donation attempt threshold; -1=no thresh
DONATE_THRESH_QUANTA 10 # The size of steps between quanta donate thresholds
MAX_DONATES 1000000  # Limit on number of donates organisms are allowed.
PRECALC_PHENOTYPE 0  # 0 = Disabled
   # 1 = Assign precalculated merit at birth (unlimited resources only)
   # 2 = Assign precalculated gestation time
   # 3 = Assign precalculated merit AND gestation time.
   # 4 = Assign last instruction counts
   # 5 = Assign last instruction counts and merit
   # 6 = Assign last instruction counts and gestation time
   # 7 = Assign everything currently supported
   # Fitness will be evaluated for organism based on these settings.
FASTFORWARD_UPDATES 0   # Fast-forward if the average generation has not changed in this many updates. (0 = OFF)
FASTFORWARD_NUM_ORGS 0  # Fast-forward if population is equal to this
GENOTYPE_PHENPLAST_CALC 100   # Number of times to test a genotype's
   # plasticity during runtime.
```

Listing E.6: GENEOLOGY_GROUP

```
### GENEOLOGY_GROUP ###
# Geneology
TRACK_MAIN_LINEAGE 1 # Keep all ancestors of the active population?
   # 0=no, 1=yes, 2=yes,w/sexual population
THRESHOLD 3
   # Number of organisms in a genotype needed for it to be considered viable.
GENOTYPE_PRINT 0  # 0/1 (OFF/ON) Print out all threshold genotypes?
GENOTYPE_PRINT_DOM 0
   # Print out a genotype if it stays dominant for this many updates. 0 = OFF
SPECIES_THRESHOLD 2  # max failure count for organisms to be same species
SPECIES_RECORDING 0  # 1 = full, 2 = limited search (parent only)
SPECIES_PRINT 0   # 0/1 (OFF/ON) Print out all species?
TEST_CPU_TIME_MOD 163   # Time allocated in test CPUs (multiple of length)
TRACK_PARENT_DIST 0  # Track parent distance during run. This is unnecessary when track main lineage is on.
```

Appendix F

Analysis Code

The TRACE command is the key command implemented in the original analysis mode, which was modified to realise the enhanced analysis. The function call relationship as found in the Avida source code is as follows:

Listing F.1: Pseudo Function Call Relationship for the TRACE Command

```
/*
Function call relationship for the analyze mode's TRACE command
 with function names and arguments.
For further information anout the class definition, dependency,
 structure, etc., refer to the actual source code.
*/
void cAnalyze::CommandTrace(cString cur_string)
   bool cTestCPU::TestGenome(cAvidaContext& ctx, cCPUTestInfo& test_info, const cGenome& genome) // with three arguments
      bool cTestCPU::TestGenome_Body(cAvidaContext& ctx, cCPUTestInfo& test_info, const cMetaGenome& genome, int cur_depth)
         bool cTestCPU::ProcessGestation(cAvidaContext& ctx, cCPUTestInfo& test_info, int cur_depth)
            bool cHardwareCPU::SingleProcess(cAvidaContext& ctx, bool speculative)
               void cHardwareStatusPrinter::TraceHardware(cHardwareBase& hardware, bool bonus)
                  void cOrganism::PrintStatus(ostream& fp, const cString& next_name) // printing of details is done here
            void cHardwareStatusPrinter::TraceTestCPU(int time_used, int time_allocated, const cOrganism& organism)
               void cOrganism::PrintFinalStatus(ostream& fp, int time_used, int time_allocated) const
```

The original function for the analysis using the TRACE command follows.

Listing F.2: Original TestGenome_Body Function

```
bool cTestCPU::TestGenome_Body(cAvidaContext& ctx, cCPUTestInfo& test_info, const cMetaGenome& genome, int cur_depth)
{
   assert(cur_depth < test_info.generation_tests);

   // Input sizes can vary based on environment settings, must at least initialize
   m_use_random_inputs = test_info.GetUseRandomInputs(); // save this value in case ResetInputs is used.
   if (!test_info.GetUseManualInputs())
      m_world->GetEnvironment().SetupInputs(ctx, input_array, m_use_random_inputs);
   else
```

```
      input_array = test_info.manual_inputs;

   receive_array.Resize(3);
   if (test_info.GetUseRandomInputs()) {
      receive_array[0] = (15 << 24) + ctx.GetRandom().GetUInt(1 << 24); // 00001111
      receive_array[1] = (51 << 24) + ctx.GetRandom().GetUInt(1 << 24); // 00110011
      receive_array[2] = (85 << 24) + ctx.GetRandom().GetUInt(1 << 24); // 01010101
   } else {
      receive_array[0] = 0x0f139f14;   // 00001111 00010011 10011111 00010100
      receive_array[1] = 0x33083ee5;   // 00110011 00001000 00111110 11100101
      receive_array[2] = 0x5562eb41;   // 01010101 01100010 11101011 01000001
   }

   if (cur_depth == 0) test_info.used_inputs = input_array;

   if (cur_depth > test_info.max_depth) test_info.max_depth = cur_depth;
    // records how far it went.

   // Setup the organism we're working with now.
   if (test_info.org_array[cur_depth] != NULL) {
      delete test_info.org_array[cur_depth];
   }
   cOrganism* organism = NULL;

   if (test_info.GetInstSet()) organism = new cOrganism(m_world, ctx, genome,
       test_info.GetInstSet());
   else organism = new cOrganism(m_world, ctx, genome);

   // Copy the test mutation rates
   organism->MutationRates().Copy(test_info.MutationRates());

   test_info.org_array[cur_depth] = organism;
   organism->SetOrgInterface(ctx, new cTestCPUInterface(this, test_info, cur_depth)
       );
   organism->GetPhenotype().SetupInject(genome.GetGenome());

   // Run the current organism.
   ProcessGestation(ctx, test_info, cur_depth);

   // Must be able to divide twice in order to form a successful colony,
   // assuming the CPU doesn't get reset on divides.
   //
   // The possibilities after this gestation cycle are:
   // 1: It did not copy at all => Exit this level.
   // 2: It copied true => Check next gestation cycle, or set is_viable.
   // 3: Its copy looks like an ancestor => copy true.
   // 4: It copied false => we must check the child.

   // Case 1:  ////////////////////////////////////
   if (organism->GetPhenotype().GetNumDivides() == 0) return false;

   // Case 2:  ////////////////////////////////////
   if (organism->GetPhenotype().CopyTrue() == true) {
      test_info.depth_found = cur_depth;
      test_info.is_viable = true;
      return true;
   }

   // Case 3:  ////////////////////////////////////
   bool is_ancestor = false;
   for (int anc_depth = 0; anc_depth < cur_depth; anc_depth++) {
      if (organism->OffspringGenome().GetGenome() == test_info.org_array[anc_depth
          ]->GetGenome()){
```

```
         is_ancestor = true;
         const int cur_cycle = cur_depth - anc_depth;
         if (test_info.max_cycle < cur_cycle) test_info.max_cycle = cur_cycle;
         test_info.cycle_to = anc_depth;
      }
   }
   if (is_ancestor) {
      test_info.depth_found = cur_depth;
      test_info.is_viable = true;
      return true;
   }

   // Case 4:  ////////////////////////////////////
   // If we haven't reached maximum depth yet, check out the child.
   if (cur_depth + 1 < test_info.generation_tests) {
      // Run the offspring's genome.
      return TestGenome_Body(ctx, test_info, organism->OffspringGenome(), cur_depth + 1);
   }

   // All options have failed; just return false.
   return false;
}
```

The modified function for the enhanced analysis using the TRACE command is as follows:

Listing F.3: Modified TestGenome_Body Function

```
int cTestCPU::TestGenome_Body(cAvidaContext& ctx, cCPUTestInfo& test_info, const cMetaGenome& genome, int cur_depth, int cur_index)
{
   // First severel procedures are left unchanged, from here...
   assert(cur_index < test_info.generation_tests);

   // Input sizes can vary based on environment settings, must at least initialize
   m_use_random_inputs = test_info.GetUseRandomInputs(); // save this value in case ResetInputs is used.
   if (!test_info.GetUseManualInputs())
      m_world->GetEnvironment().SetupInputs(ctx, input_array, m_use_random_inputs);
   else
      input_array = test_info.manual_inputs;

   receive_array.Resize(3);
   if (test_info.GetUseRandomInputs()) {
      receive_array[0] = (15 << 24) + ctx.GetRandom().GetUInt(1 << 24);  // 00001111
      receive_array[1] = (51 << 24) + ctx.GetRandom().GetUInt(1 << 24);  // 00110011
      receive_array[2] = (85 << 24) + ctx.GetRandom().GetUInt(1 << 24);  // 01010101
   } else {
      receive_array[0] = 0x0f139f14;  // 00001111 00010011 10011111 00010100
      receive_array[1] = 0x33083ee5;  // 00110011 00001000 00111110 11100101
      receive_array[2] = 0x5562eb41;  // 01010101 01100010 11101011 01000001
   }

   if (cur_depth == 0) test_info.used_inputs = input_array;

   if (cur_index > test_info.max_depth) test_info.max_depth = cur_index;
   // ...left unchanged till here.

   // For "breadth first trace" (and tree creation)
   std::queue<int> bft_queue; // intended to hold indices

   if (cur_index == 0) { // initial process. incubate the first one in the strain_array[]
      sStrain* strain = new sStrain();
      test_info.strain_array[cur_index] = strain;
      test_info.strain_array[cur_index]->size = genome.GetSize();
      test_info.strain_array[cur_index]->incubated = true;

      cOrganism* organism = NULL; // set up the organism
      // is needed anyway to compare to elements in the array
      if (test_info.GetInstSet()) organism = new cOrganism(m_world, ctx, genome, test_info.GetInstSet());
      else organism = new cOrganism(m_world, ctx, genome);
      organism->MutationRates().Copy(test_info.MutationRates()); // just in case, copy the test mutation rates

      test_info.org_array[cur_index] = organism; // register org in the array for comparison later
      organism->SetOrgInterface(ctx, new cTestCPUInterface(this, test_info, cur_index)); // part of original code
      organism->GetPhenotype().SetupInject(genome.GetGenome()); // just in case, for completeness

      bft_queue.push(cur_index);
   }

   assert (test_info.strain_array[cur_index]!=NULL);

   while (!bft_queue.empty()) {
      // "cur_index" is now used to point to an index for a new offspring!!
      int par_index = bft_queue.front(); // meaning it "may" become a parent, if and only if it divides
      bft_queue.pop(); // dequeue

      // TRACE: produce a trace file of this strain as usual
      ProcessGestation(ctx, test_info, par_index);

      cHardwareTracer* tracer = test_info.GetTracer();
      // to print out whether divided or not

      test_info.strain_array[par_index]->gestationTime = test_info.org_array[par_index]->GetPhenotype().GetGestationTime();

      // if not divided
      if (test_info.org_array[par_index]->GetPhenotype().GetNumDivides() == 0) { // if not divided
         test_info.strain_array[par_index]->divided = false;
         if (tracer != NULL) test_info.org_array[par_index]->PrintCaseRecursion(*(tracer->GetStream()), 0);

         test_info.strain_array[par_index]->leftChildIndex = -1;
         test_info.strain_array[par_index]->rightChildIndex = -1;
      }
      else {// if divided
         test_info.strain_array[par_index]->divided = true;
         if (tracer != NULL) test_info.org_array[par_index]->PrintCaseRecursion(*(tracer->GetStream()), 1);

         bool flagSeen;

         // 1) this I call LEFT side of the tree, always fully checked first
         // first check if organism->FinalGenome().GetGenome()
         // is previously seen in the org_array[]
         for (int i=0; i<=cur_index; i++) {
            flagSeen = false;
            if (test_info.org_array[i]->GetGenome() == test_info.org_array[par_index]->FinalGenome().GetGenome()) {
               test_info.strain_array[par_index]->leftChildIndex = i; // set leftChildIndex to the index seen
               flagSeen = true;
               break;
            }
         }
         if (flagSeen == false) { // if this first child of the offspring is never seen (new),
            cur_index++;

            test_info.strain_array[par_index]->leftChildIndex = cur_index; // register as a child of the parent

            if (0 <= cur_index && cur_index < test_info.generation_tests) {
               // test before passing onto the next call
               // forget the depth limit now that it is breadth first
               if (test_info.strain_array[cur_index]==NULL) { // embody a strain for this child
                  sStrain* strain = new sStrain();
                  test_info.strain_array[cur_index] = strain;
                  strain->size = test_info.org_array[par_index]->FinalGenome().GetSize();
                  strain->incubated = true; // here the child is incubated

                  cOrganism* organism = NULL; // set up the organism
                  // is needed anyway to compare to elements in the array
                  if (test_info.GetInstSet()) organism = new cOrganism(m_world, ctx, test_info.org_array[par_index]->FinalGenome(), test_info.GetInstSet());
                  else organism = new cOrganism(m_world, ctx, test_info.org_array[par_index]->FinalGenome());
                  organism->MutationRates().Copy(test_info.MutationRates()); // just in case, copy the test mutation rates

                  test_info.org_array[cur_index] = organism; // register org in the array for comparison later
                  organism->SetOrgInterface(ctx, new cTestCPUInterface(this, test_info, cur_index)); // part of original code
                  organism->GetPhenotype().SetupInject(test_info.org_array[par_index]->FinalGenome().GetGenome()); // just in case, for completeness

                  bft_queue.push(cur_index); // enqueue
               }
            }
            else {
               std::cout << " Out of Bound " << std::endl;
               std::cout<< " queue size: " << bft_queue.size()
               << ", queue front: " << bft_queue.front()
               << ", empty?: " << boolalpha << bft_queue.empty()
               << std::endl;

               return cur_index;
            }
         } // if this first child of the offspring is already seen, do nothing.

         // 2) then move on to the other RIGHT side of the tree
         // first check if organism->OffspringGenome().GetGenome()
```

```
         // is previously seen in the org_array[]
         for (int i=0; i<=cur_index; i++) {
            flagSeen = false;
            if (test_info.org_array[i]->GetGenome() == test_info.org_array[par_index]->OffspringGenome().GetGenome()) {
               test_info.strain_array[par_index]->rightChildIndex = i; // set rightChildIndex to the index seen
               flagSeen = true;
               break;
            }
         }
         if (flagSeen == false) { // if this second child is never seen (new),
            cur_index++;

            test_info.strain_array[par_index]->rightChildIndex = cur_index; // register as the other child of the parent

            if (0 <= cur_index && cur_index < test_info.generation_tests) {
               // test before passing onto the next call
               // forget the depth limit now that it is breadth first
               if (test_info.strain_array[cur_index]==NULL) { // embody a strain for this child
                  sStrain* strain = new sStrain();
                  test_info.strain_array[cur_index] = strain;
                  strain->size = test_info.org_array[par_index]->OffspringGenome().GetSize();
                  strain->incubated = true; // here the other child is incubated

                  cOrganism* organism = NULL; // set up the organism
                  // is needed anyway to compare to elements in the array
                  if (test_info.GetInstSet()) organism = new cOrganism(m_world, ctx, test_info.org_array[par_index]->OffspringGenome(), test_info.GetInstSet());
                  else organism = new cOrganism(m_world, ctx, test_info.org_array[par_index]->OffspringGenome());
                  organism->MutationRates().Copy(test_info.MutationRates()); // just in case, copy the test mutation rates

                  test_info.org_array[cur_index] = organism; // register org in the array for comparison later
                  organism->SetOrgInterface(ctx, new cTestCPUInterface(this, test_info, cur_index)); // part of original code
                  organism->GetPhenotype().SetupInject(test_info.org_array[par_index]->OffspringGenome().GetGenome()); // just in case, for completeness

                  bft_queue.push(cur_index); // enqueue
               }
            }
            else {
               std::cout << " Out of Bound " << std::endl;
               std::cout<< " queue size: " << bft_queue.size()
               << ", queue front: " << bft_queue.front()
               << ", empty?: " << boolalpha << bft_queue.empty()
               << std::endl;

               return cur_index;
            }
         } // if this second child of the offspring is already seen, do nothing.

      } // else ends
      std::cout << " After a while, queue size: " << bft_queue.size()
      << ", queue front: " << bft_queue.front()
      << ", empty?: " << boolalpha << bft_queue.empty()
      << std::endl;
   } // while ends

   return cur_index; // always return the next index
}
```

Listing F.4: Added Struct Representing a Strain

```
struct sStrain {
    int size;
    bool incubated;
    bool divided;
    int gestationTime;
    int leftChildIndex; // first
    int rightChildIndex; // second
};
```

Appendix G

Automation Scripts

Scripts (in Python) used for the pre-analysis, the Avida trace, and the post-analysis (see Section 4.5 in Chapter 4) are shown.

Listing G.1: `f0verarching.py`

```
import subprocess
import sys
import os
import shutil
import glob
import fGenerateMutants
import fPostAnalysis
import fDigest
import fExtract

def overarch(cur_g, cur_ss):
    print "current generation: ", cur_g, ", source strain: ", cur_ss
    if cur_g < 3: # set generations to go
        # 1) pre-analysis
        print "now pre-analysis..."
        fGenerateMutants.main(cur_g, cur_ss)

        print "now renaming analyze.cfg..."
        # create a new analyze.cfg
        with open("./analyze.cfg", ’w’) as f:
            f.write("VERBOSE\n")
            f.write("LOAD " + "generatedMutants_" + cur_ss + ".pop" + "\n")
            f.write("TRACE\n")
        f.close()


        # 2) main analysis
        print "now main analysis..."
        # enhanced avida -a
        # produces trace files in ./data/archive

        p = subprocess.Popen(["./avida", "-a"], stdout=subprocess.PIPE, stderr=
            subprocess.STDOUT)
        for line in iter(p.stdout.readline, ""):
            line = p.stdout.readline()
            if line:
                print line
            else:
                break
        p.wait() #to wait until the trace files are ready??


        # 3) post-analysis
        print "now postAnalysis..."
        fPostAnalysis.main() # from trace files, produces lineages and dynamics folders and files beneath

        print "now digest..."
        maxArraySize = 100
        fDigest.main(maxArraySize)

        print "now extract..."
        next_g = cur_g + 1
        fExtract.main(next_g, cur_ss)


        # rename the data folder so as not to get over-written
        shutil.move("./data", "./data_" + cur_ss)
        # next turn a data folder will be created by avida

        # rename the past analyze.cfg to save
        os.rename("./analyze.cfg", "./analyze_" + cur_ss + ".cfg")
        shutil.move("./analyze_" + cur_ss + ".cfg", "./pastAnalyzeCfgs/analyze_" + cur_ss + ".cfg")


        # recursively iterate
        p = "./generation-" + str(next_g) + "/" + cur_ss + "-*.pop"
        for file in glob.glob(p):
            next_ss = file.split("/")[-1].split(".")[0]
            print "next generation:", next_g, ", next source strain: ", next_ss
            overarch(next_g, next_ss)



g = 0 # generation, or depth
#ss = "org-0" # in the generation-0 folder
#ss = "hcf28_org-0"# h-copy-free version one
ss = "hcf27_org-0" # h-copy-free version two

#### check list before running the script ####
 # ! 1) change the instset in "avida.cfg" and in "fGenerateMutants.py" too (28/27)
 # ! 2) change the lower bound in "fExtract.py" (down the middle),
 # ! and the upper bound in "avida.cfg" ("TEST_CPU_TIME_MOD")
 # ! (depending on the gest_time and the size of the first source strain)
 # ! 3) set the max depth above ("cur_g <"),
 # ! and the selection size in the "fExtract.py" ("top ==")

if not os.path.exists("./mutantsWithTags"):
    os.makedirs("./mutantsWithTags")
if not os.path.exists("./pastAnalyzeCfgs"):
    os.makedirs("./pastAnalyzeCfgs")

overarch(g, ss)
```

Listing G.2: fGenerateMutants.py

```
import sys

def main(current_generation, source_strain):
    #prototype
    fi = open("./generation-" + str(current_generation) + "/" + source_strain + ".pop", 'r')

    for line in fi:
        if '#' in line or '\n' == line:
            continue
        else:
            inseq = line
            print "line " + inseq + "\n is set."
            break

    #instset = "abcdefghijklmnopqrstuvwxyzAB" #28 insts
    instset = "abcdefghijklmnopqrstuvwxyzA" #27 insts


    #strings (are immutable)
    phenome = inseq[:(len(inseq)/2)]
    ttable = phenome[-len(instset):]
    genome = inseq[(len(inseq)/2):]

    #create possible mutations (combinations)
    mutgenomes = []
    for i in range(len(genome)):
        for inst in instset:
            if genome[i] != inst: #exclude itself
                amutg = genome[:i] + inst + genome[i+1:] #temporary string
                mutgenomes.append(amutg) #list of mutated genome sequences(strings)

    #decode
    mutants = [] #list of mutants (with mutations carried and expressed)
    for mutgenome in mutgenomes: #string
        anexpp = []
        for letter in mutgenome: #to create a phenome with a mutation expressed
            anexpp.append(ttable[instset.find(letter)])
        mutants.append("".join(anexpp)[:] + mutgenome[:]) #turn lists into a string and add to a list

    #to produce the input file (.pop) for the Avida analyze mode
    f = open("./generatedMutants_" + source_strain + ".pop", 'w')
    f.write("#filetype genotype_data\n")
    f.write("#format sequence\n\n")
    for mutant in mutants:
        f.write(str(mutant) + "\n")
    f.close()

    #to tag with the locations
    #(where and how the mutation is expressed and carried)
    f = open("./mutantsWithTags/mutantsWithTag_" + source_strain + ".txt", 'w')
    i = 0
    j = 1
    for mutant in mutants:
        f.write(str(mutant) + "\t")
        if i >= j*(len(instset)-1):
            j += 1
        f.write(str(j-1) + "_" + str(j-1+len(inseq)/2) + "\t")
        f.write(inseq[j-1] + "->" + mutant[j-1] + "_" + inseq[j-1+len(inseq)/2] + "->" +mutant[j-1+len(inseq)/2] + "\n")
        i += 1
    f.close()
```

Listing G.3: fPostAnalysis.py

```
from scipy.linalg import eig
import glob
import os

path1 = "./data/archive/*.trace"
```

```python
# assuming you are on the same level as avida etc.
# underneath we make the "lineages" directory and the "dynamics" directory

def init_matrix(n, init_v):
    matrix = [[init_v for i in range(n)] for j in range(n)]
    return matrix

def main():
    if not os.path.exists("./data/archive/lineages"):
        os.makedirs("./data/archive/lineages")

    if not os.path.exists("./data/archive/dynamics"):
        os.makedirs("./data/archive/dynamics")

    for file in glob.glob(path1):# work on trace files
        with open(file) as f1:
            # step 1: extract adjacency lists from trace files
            fout1 = open("./data/archive/lineages/" + str(file.split("/")[-1]).split(".")[0] + ".lineage", "w")
            # for a starter read'em all
            lines = f1.readlines()
            i = lines.index("Index Size Incubated? Divided? GestationTime LeftChildIndex RightChildIndex\n")
            lineage = []
            # only after the above label come strains as adjacency lists
            for s in lines[i+1:]:
                strain = s.split()
                lineage.append(strain)
            fout1.write(str(lineage))
            fout1.write("\n")
            fout1.close()

            # step 2: create the matrix and populate with rates and calculate the dynamics
            fout2 = open("./data/archive/dynamics/" + str(file.split("/")[-1]).split(".")[0] + ".txt", "w")
            # convert strings to ints or bools # revised on 2 Nov 2013, after adding "size"
            for s in lineage:
                s[0] = int(s[0]) # index
                s[1] = int(s[1]) # size
                if s[2] == 'true': # incubated?
                    s[2] = True
                else:
                    s[2] = False
                if s[3] == 'true': # divided?
                    s[3] = True
                else:
                    s[3] = False
                s[4] = int(s[4]) # gestation time
                s[5] = int(s[5]) # "left" child index (formerly "final" memory image) dealt with first
                s[6] = int(s[6]) # "right" child index (formerly "child" memory image) dealt with second

            print "lineage " + str(file.split("/")[-1]).split(".")[0].split("-")[1] #, ": ", lineage
            fout2.write("There are " + str(len(lineage)) + " distinct strains.\n")

            # collect the gestation time to create a rate vector (use reciprocal)
            gest_time = []
            rates = []
            for s in lineage:
                if s[4] != 0:
                    gest_time.append(s[4])
                    rates.append(1.0/s[4])
                else:
                    gest_time.append(0)
                    rates.append(0)

            fout2.write("gest_time: " + str(gest_time) + "\n")
            fout2.write("rates: " + str(rates) + "\n")

            if len(lineage) < 3000:
                transition = init_matrix(len(lineage), 0)

                for s in lineage:# so that a column shows a parent producing offspring which correspond to rows
                    if s[3] == True: # if divided
                        transition[s[0]][s[0]] = transition[s[0]][s[0]] - rates[s[0]] # itself disappears at the rate
                        if s[5] < len(lineage): # id for the left offspring
                            transition[s[5]][s[0]] = transition[s[5]][s[0]] + rates[s[0]] # the offspring appears at the rate
                        else:
                            fout2.write("leftChild of " + str(s) + " hit the recursion limit, regarded as no divide\n")
                        if s[6] < len(lineage): # id for the right offspring
                            transition[s[6]][s[0]] = transition[s[6]][s[0]] + rates[s[0]] # the offspring appears at the rate
                        else:
                            fout2.write("rightChild of " + str(s) + " hit the recursion limit, regarded as no divide\n")

                D,V = eig(transition) # should have the same dimension by definition
                fout2.write("eigenvalues: " + str(D) + "\n")
                fout2.write("eigenvectors: " + str(V) + "\n")

                # for detecting exponential growth for known mutants
                expStrains = []
                for i in xrange(0,len(D)): # access each column (e.vec for each e.val) and see if the signs are the same
                    if (all(item >= 0 for item in [row[i] for row in V]) or all(item <= 0 for item in [row[i] for row in V])):
                        if D[i].real > 0: # if e.vec is valid, then see if the e.vec real part is positive
                            expStrains.append(i) # count in this strain if both e.vec and e.val are valid
                fout2.write(str(len(expStrains)) + " are exp. growing strains: " + str(expStrains) + "\n")
                if max(D.real) > 0:
                    fout2.write(" max real part (>0): " + str(max(D.real)))
                    # get one youngest fittest strain
                    fout2.write(" by: " + str([i for i,val in enumerate(D.real) if val == max(D.real)]) + "\t")
                    # and its gest time ([4] of each strain in lineage)
                    fout2.write(str([lineage[i][4] for i,val in enumerate(D.real) if val == max(D.real)]) + "\n")
            else:
                fout2.write("too big a matrix (>=3000)\n")
            fout2.close()
        f1.close()
```

Listing G.4: `fDigest.py`

```python
# produce a digest version of the analysis
# extract interesting mutants for the next generation analysis
# by judging potential of exp. growth, from "dynamics"

import glob

def main(maxArraySize):
    path_dyn = "./data/archive/dynamics/*.txt"

    recurHitLins = []
    maxHitLins = []
    tooBig = []
    maxInd = 0
    numDistinctStrains = []
    numExpStrains = []

    f1 = open("./data/archive/summary", "w")
    f1.write("strain_id"+"\t"+"derived_strains"+"\t"+"1st_gest_time"+"\t"+"exp_growing"+"\t"+"max_real_part"+"\t"+"by_whom"+ "\t" +"its_gest_time\n")
    for file in glob.glob(path_dyn):
        print "now", file.split("/")[-1].split(".")[0]
        with open(file) as f:
            lines = f.readlines()
            recurHitCount = 0
            this_id = str(file.split("/")[-1].split(".")[0])
            f1.write(this_id + "\t") # strain id
            for line in lines:
                if "distinct strains." in line:
                    numDistinctStrains.append(int(line.split()[2])) # for distribution
                    f1.write(str(line.split()[2]) + "\t")
                    if maxArraySize == int(line.split()[2]):
                        maxHitLins.append(file.split("/")[-1].split(".")[0])
                if "gest_time" in line:
                    this_gt = line.split()[1].strip('[],')
                    f1.write(this_gt + "\t") # gest_time
                if "hit" in line:
                    recurHitCount += 1
                    if maxInd < int(line.split()[2].strip("[,")):
                        maxInd = int(line.split()[2].strip("[,"))
                if "too big" in line:
                    tooBig.append(file.split("/")[-1].split(".")[0])
                    f1.write("too_big\t")
                if " are exp. growing strains: " in line:
                    numExpStrains.append(int(line.split()[0]))
                    f1.write(str(int(line.split()[0])) + "\t")
                if " max real part (>0): " in line:
                    f1.write(" " + str(line).rstrip().split()[4] + "\t")
                    # which strain has this max real part? its gest time?
                    f1.write(" " + str(line).rstrip().split()[6].split()[0].strip('[,]') + "\t")
                    f1.write(" " + str(line).rstrip().split()[7].split()[0].strip('[,]') + "\t")

            if recurHitCount > 0:
                recurHitLins.append(file.split("/")[-1].split(".")[0]) # should be recursion_limit and times
                f1.write("recursion_hit_" + str(recurHitCount) + "\t")

            f1.write("\n")

    print numDistinctStrains
    print numExpStrains

    print len(recurHitLins), " have reached the recursion limit at least once."
    print recurHitLins
    print "(Of them, ", maxInd, " is the max index.)"

    print len(maxHitLins), " have reached the array size limit."
    print maxHitLins
    print "(Including them, ", len(tooBig), " are too many distinct strains (>= 3000).)"
    print tooBig

    f1.close()
```

Listing G.5: fExtract.py

```
import os

def main(next_g, cur_ss):
    # with new summary format: [0]: org-id, [1]: total distinct strains,
    # [2]: first gest time, [3]: exp growing strains, [4]: if any, max real part of eig.val.
    # collect lists by checking values >1 in a column for exp. growth
    # sort it by the key (max real part of eigenvalue)
    # pick up ids from the top, only if gest_time is okay, up to 10(?) strains

    ## collect ids of the extracted strains
    extracted_strains = []
    with open("./data/archive/summary", 'r') as f:
        lines = f.readlines()
        for line in lines[1:]:
            try:
                max_real = int(line.split()[3])
                if max_real > 0:
                    strain_entry = line.split()
                    extracted_strains.append(strain_entry)
            except:
                pass

    f.close()

    # assuming extracted lines necessarily have the [4] element
    # also [5] by whom the maxt real part is had, and [6] its gest time
    ranking = sorted(extracted_strains, key=lambda x: float(x[4]), reverse=True)
    #print ranking

    pickups = []
    whom = []
    top = 0
    for strain in ranking:
        if int(strain[6]) > 59218*0.5: #59218 for "27" ver. #59392*0.5:#for "28" ver.  #52218*0.5:#for priginal prototype
            pickups.append(strain[0])
            whom.append(strain[5])
            top += 1
            if top == 3: # 10: # set how many from the top to be selected
                break

    print top, pickups, whom

    loc_tra = "./data/archive/"

    if not os.path.exists("./generation-" + str(next_g)):
        os.makedirs("./generation-" + str(next_g))

    ctr = 0
    for str_id in pickups:
        with open(loc_tra + str_id + ".trace") as ft:
            new_str_id = cur_ss + "-" + str_id.split("-")[1]
            fn = open("./generation-" + str(next_g) + "/" + new_str_id + ".pop", 'w')
            fn.write("#filetype genotype_data\n" + "#format sequence\n\n") # format
            lines = ft.readlines()
            nth = 0
            for line in lines:
                if "# Init Memory:" in line:
                    if nth == int(whom[ctr]):
                        fn.write(line.split()[3])
                        break
                    else:
                        nth += 1
        ctr += 1
        ft.close()
        fn.close()
```

Appendix H

Glossary

The research in this thesis deals with the von Neumann style self-reproduction architecture and its evolutionary characterisation within Avida. Throughout the thesis, there are some terms and concepts that are introduced or defined somewhat differently from those found in the Avida literature or in other evolutionary research contexts. This glossary presents simple descriptions of such relevant terms frequently used in the main text.

Copy

Write to a memory location (of a prospective offspring) a word identical to one read from another memory location (of a putative parent), as a result of program execution. Or, a process of copying words. Compare with *decode*.

Decode

Write to a memory location (especially, within a phenome of a prospective offspring) a word that is read from another location (especially, within a genome of a putative parent) and is translated (especially, through a lookup table). Compare with *copy*.

Fertile

Capable of directly producing offspring. Compare with *viable*.

Genome

Passive segment of program within a von Neumann style self-reproducer. Compare with *phenome*.

Genotype

Class of identical genomes. Compare with *phenome*.

Lineage

Network of organisms/strains that a seed organism/strain deterministically generate.

Memory

Circular component of an Avidian organism on which a program is executed by CPU. (In the Avida literature, this is called genome instead.)

Mutation

Inheritable change in memory image. Compare with *perturbation*.

Organism

Program coupled with CPU in the Avida world. Compare with *strain*.

Perturbation

Change in memory image. Compare with *mutation*.

Phenome

Active segment of program within a von Neumann style self-reproducer. Compare with *genome*.

Phenotype

Class of identical phenomes. Compare with *genome*.

Production Graph

A way of representing lineages. Used to represent lineages of distinct strains. Compare with *production tree*.

Production Tree

A way of representing lineages. Used to represent lineages of individual organisms. Compare with *production graph*.

Strain

Memory image of a program in the Avida world. Compare with *organism*. (In the Avida literature, this is called genotype instead.)

Viable

Capable of producing evolutionarily significant (especially, self-reproducing) offspring directly or indirectly. Compare with *fertile*.