Université de Lille

***École Doctorale* des Sciences de l'Homme et de la Société N°473**

***STL – Savoirs, Textes, Langage* UMR8163**

Mind the Gap

A Historico-Philosophical Investigation of the Ontological Status of Computer Programs

Une investigation historico-philosophique du statut ontologique des programmes informatiques

Nick WIGGERSHAUS

Thèse de doctorat de **Philosophie**

Dirigée par **Shahid RAHMAN**
Et par **Liesbeth DE MOL**

Présentée et soutenue publiquement
le 08/01/2025

Devant un jury composé de :

Tarja KNUUTTILA, Professeure, Universität Wien, Examinatrice

Raymond TURNER, Professeur émérite, University of Essex, Rapporteur,

Giuseppe PRIMIERO, Professeur, Università degli Studi di Milano Statale, Président

Julie JEBEILE, Maître de conférences, CNRS-Universität Bern, Rapporteure

This page intentionally left blank

To my Parents & Grandparents

Abstract

This thesis addresses the ontological status of computer programs. Previous studies have placed computer programs in almost every ontological category available and claimed that they have a 'dual nature.' My primary contribution to the debate is offering an alternative framework emphasizing computational implementation and its relata. I argue that we do not need to endorse the dual nature view by suggesting that 'program' is a polyseme – an umbrella term hosting various entities spanning the abstract-concrete dichotomy. The advantage of this view is the avoidance of positing metaphysically dubious entities. Instead, we can understand the ontology of programs as a network of relations between abstracta and concreta that *we* bundle together through implementation when using computing machines as epistemic tools.

To flash out and explore the claims of this alternative view, I first delve into the philosophical literature on implementation and taxonomize its different notions. In due course, I offer a unified theory of agential implementation, short UTAI. Specifically, UTAI advocates developing a series of related clarificatory case studies that track three different dependency relations between us and the ontologically different constituents under the term program.

Accordingly, I discuss the implications of the first dependency relation between programmers and programs as abstract objects. By adopting the Problem of Creation – a well-known issue from the philosophy of art – I present a fresh perspective on the metaphysical options that allow us to view programs as abstract entities. Next, I focus on the second dependency relation between human agents and physical computation. As a result, I provide a new argument for understanding computational implementation as a three-place relation and develop a suitable notion called Implementation-as (based on the DEKI account of scientific representation). Lastly, I address the third dependency relation between programmers and the material systems used for program execution. By combining the insights of interventionism, technical artifacts, and neo-mechanistic literature, I introduce the notion of 'physical programmability.'

Keywords: Computer Programs, Ontology, Problem of Creation, Implementation, Scientific Models, Programmability

Résumé

Cette thèse examine le statut ontologique des programmes informatiques. De précédentes études ont placé les programmes informatiques dans presque toutes les catégories ontologiques disponibles et ont affirmé qu'ils avaient une « double nature ». Ma principale contribution au débat consiste à proposer un cadre alternatif mettant l'accent sur l'implémentation informatique et ses relata. Je soutiens qu'il n'est pas nécessaire d'approuver le point de vue de la double nature en suggérant que le terme « programme » est un polysème - un terme générique abritant diverses entités couvrant la dichotomie abstrait-concret. L'avantage de ce point de vue est qu'il évite de poser des entités métaphysiquement douteuses. Au lieu de cela, nous pouvons comprendre l'ontologie des programmes comme un réseau de relations entre abstracta et concreta que nous rassemblons par la mise en œuvre lorsque nous utilisons des machines informatiques comme outils épistémiques.

Pour mettre en évidence et explorer les revendications de ce point de vue alternatif, je me plonge d'abord dans la littérature philosophique sur l'implémentation et je taxinomise ses différentes notions. En temps voulu, je propose une théorie unifiée de l'implémentation agentielle, l'UTAI. Plus précisément, l'UTAI préconise le développement d'une série d'études de cas clarificatrices connexes qui suivent trois relations de dépendance différentes entre nous et les constituants ontologiquement différents sous le terme de programme.

En conséquence, je discute des implications de la première relation de dépendance entre les programmeurs et les programmes en tant qu'objets abstraits. En adoptant le problème de la création - une question bien connue de la philosophie de l'art - je présente une nouvelle perspective sur les options métaphysiques qui nous permettent de considérer les programmes comme des entités abstraites. Ensuite, je me concentre sur la seconde relation de dépendance entre les agents humains et l'informatique physique. En conséquence, je fournis un nouvel argument pour comprendre l'implémentation informatique comme une relation à trois places et je développe une notion appropriée appelée Implémentation-as (basée sur le compte DEKI de la représentation scientifique). Enfin, j'aborde la troisième relation de dépendance entre les programmeurs et les systèmes matériels utilisés pour l'exécution des programmes. En combinant les idées de l'interventionnisme, des artefacts techniques et de la littérature néo-mécaniste, j'introduis la notion de « programmabilité physique ».

Mots-clés: Programmes informatiques, le problème de la création, ontologie, implémentation, modèles scientifiques, programmabilité

Acknowledgments

My fascination with computer programs, computing, and implementation started late. I believe at some point in Spring 2018. Earlier that year, I had successfully graduated from the History and Philosophy of Science Master in Utrecht with a thesis about the ontological status of information in physics. Investigating how so-called information measures such as *Shannon information* and *Kolmogorov Complexity* feature in physics, I gained first familiarity with concepts in the vicinity of computer science. My interest only seriously began when I first learned about the alleged dual nature of programs. Intuitively, the dual nature discourse had striking parallels with the dubious claims about the simultaneously abstract and physical nature of information. To this day, I still find such ideas utterly puzzling; how could programs both be abstract and concrete at the same time? This dissertation project is an attempt to shed light on this question and make sense of the ontological status of computer programs.

Little did I know what I had signed up for at the beginning of this endeavor. Due to circumstances beyond my control, my PhD experience has not been exactly the smoothest. From dealing with the challenges of COVID-19 that forced me to work remotely for a significant part of the process, then the headache of having to learn French (online), paired with working without a contract for more than two years, to the absence of supervisor(s) due to health reasons when it came time to write up my thesis. So, I can say with full confidence that I only made it thus far because of my family, friends, and some of the colleagues I met along this journey. I want to express my heartfelt gratitude and say *thank you* to everyone who offered me support and assistance in various ways! Among these people, certain individuals deserve to be singled out.

First, I want to thank my supervisors, Liesbeth De Mol and Shahid Rahman, for their guidance and encouragement with this project. Liesbeth's profound understanding of the diverse aspects of computing not only allowed me to check my philosophical reasoning against the backdrop of real-world examples but also encouraged me to break free from narrow disciplinary boundaries and venture into uncharted territories. Shahid, who co-supervised part of this thesis, is a well of wisdom in philosophy and French bureaucracy. His sharp observations helped me iron out some of the more questionable attempts in using formulae previously contained in this thesis; in addition, he also helped me navigate the idiosyncrasies of the French academic system. I am grateful for both their patience and the trust they have put in me.

Additionally, I am indebted to *PROGRAMme* (Nicola Angius, Troy Astarte, Arianna Borrelli, Selmer Bringsjord, Maarten Bullynck, Felice Cardone, Martin

Carlé, Edgar Daylight, Liesbeth De Mol, Marie-José Durand-Richard, Endy Hammache, Cliff Jones, Simone Martini, Baptiste Mélès, Lennart Melzer, Elisabetta Mori, Pierre Mounier-Kuhn, Alberto Naibo, Maël Pégny, Tomas Petricek, Mark Priestley, Giuseppe Primiero, Julian Rohrhuber, David Schmudde, Henri Stephanou, Mate Szabo, and Raymond Turner), a truly exceptional group of scholars with unmatched expertise in all aspects of computing, who have welcomed me with open arms as one of their members. As a junior researcher, this was truly a unique opportunity, and I profited immeasurably from the stimulating intellectual environment of the project. I will surely never forget all the long nights discussing all kinds of computing-related topics in Bertinoro and Lille.

These last years have also been more enjoyable, thanks to Sam Rijken and Noelia Iranzo Ribeira, with whom I had the pleasure of sharing the academic journey first remotely and later in person. I am glad that our research interests aligned, and we decided to organize the Fictions in Science and Metaphysics symposium for BSPS21.

In due course, several other people have exchanged ideas by email or in person, offered inspiring comments on various themes contained in this manuscript, and read portions and/or offered helpful comments on various themes contained in it. Worthy of mention in this regard are Edgar Daylight, Mike Cuffaro, André Curtis-Trudel, Roman Frigg, Phillipos Papayannopoulos, Alice Martin, Noelia Iranzo Ribeira, Ruward Mulders, Henri Stephanou and Raymond Turner. In addition, I have presented material in this work to audiences at various conferences and workshops, and I am grateful for all the comments I have received – but of course, none of them is responsible for any errors in my work.

I owe a special debt of gratitude to my loved ones outside of academia. Thanks to Cecilia, the last two years were filled with so much warmth, affection, and an incredible amount of emotional support. *Grazie amore*, for putting my academic struggles into perspective and tolerating my silly antics about the MONIAC. Finally, my family's unconditional support, their patience (even though they had no idea what I was doing), and their ability to listen enabled me to do all of this. Without the privilege of having such great people in my life, I wouldn´t have made it. *Vielen Dank, ohne euch hätte ich das nicht geschafft!*

Utrecht, the Netherlands
September 2024 *Nick Wiggershaus*

Contents

1 Introduction

1.1 Prologue

Imagine you are a young and curious intellectual property lawyer in the early 1970s. It´s the midst of the Cold War. The iron curtain divides Europe; in previous years, Africa lived through a period of radical political change as some 30 countries gained independence; and China went through a massive sociocultural movement. Paralleling these geopolitical events, there are significant technological advancements – with information technologies spearheading them. For you, the proliferation of new inventions is a blessing and secures your job. Put roughly, it is one of the main tasks of lawyers like you to classify new creations and inventions such as books, music, machines, and processes under your country's IP law. In a nutshell, there are three different kinds of categories in which novel inventions like these must be placed: First, patents give inventors the property right to a tangible technical or scientific invention or process. Second, copyright claims are meant for the protection of an original expression of an idea in a creative work – literary, dramatic, musical, or artistic work, and movies (fixed in some tangible medium) typically fall under this scope. Third, if something falls out of the scope of the first two categories, it cannot be legally protected. Now, for a couple of years, you have received increasing requests to grant IP protection to computer programs and software. How would you classify them? What kind of things are programs?[1]

Pondering these questions, you are eagerly awaiting the result of the US Supreme Court case of Gottschalk v. Benson (1972): the case is about the patentability of a system created by Gary Benson and Arthur Tabbot at Bell Telephone Laboratories that allowed the creation of a telephone network called a private branch exchange (PBX). Crucially, their method relied on a computational method that converted binary-coded decimal numerals into their binary equivalent. Finally, on November 20, the Supreme Court issued its ruling. The court proclaimed that Benson and Tabbot's invention was ineligible for patent protection because it stands at odds with the mental-step doctrine.[2] Traditionally, this doctrine has been used to reject those seeking patents for 'inventions' like algorithms – and the PBX was judged to be precisely that.

The decision irritates you. True, whilst one could 'execute' Benson and Tabbot's method in one's head, isn't the whole point that their method triggers

[1] The game of make-believe is freely inspired by Gerardo Con Diaz's (2019) book *Software Rights*.

[2] The mental-step doctrine holds that inventions that can be performed in the human mind or by a human using a pen and paper are not eligible for patent protection.

an automated physical process in an actual machine? Feeling somewhat uneasy with the Supreme Court's ruling, you continue your quest and tinker around with a different classification strategy – the widely used software/hardware distinction. On the face of it, the duality of the latter intuitively seems to correspond to the copyright/patent dichotomy; you believe you're onto something. If true, you just solved future legal cases worth billions of dollars. Case closed – you're set for life. However, things are not that easy. With big money involved your case better ought to be watertight, else industry will drown you in endless legal battles. Scratching beyond the surface of your initial strategy, you start getting doubts: Are computer programs software? What is the software/hardware distinction actually supposed to demarcate from one another? And what the heck are programs exactly?

In trying to answer these questions, you discover that your first classification strategy is in danger of serious conflations. First, it is all but clear that all computer programs can be considered as software. Doing your research about the emergence of computing technology, you find that some of the first programs (made before the appearance of the term 'software' and the emergence of 'computer science' as a subject)[3] were often portrayed as circuits of switches using relays or vacuum tubes: when setting up the first-generation electronic digital computers like ENIAC (devices that filled entire rooms) resulted in 'programs' appearing to be tangible hardwired switch settings of machines intended to perform a specific task. In fact, this materiality allowed some of your colleagues to secure patent protection for programs developed at hardware companies and industrial research labs (Con Diaz 2019, 3). So, can you conclude that programs are indeed just physical entities that should be subject to patent law? Generalizing from cases like these that point towards the techno-material nature of programs seems to undermine the idea that programs are a type of software considerably – at least to the extent that software is taken to be opposed to hardware and hardware is supposed to be 'hard,' i.e., presumably made of concrete, tangible components.

So perhaps the crux lies with the software-hardware distinction itself. In fact, shortly after the time you started pondering these questions, philosophically inclined software engineer James Moor (1978) published his article 'Three Myths of Computer Science' that confirmed your worries: One of the takeaways from Moor's article is that the software/hardware distinction is merely useful as a

[3] According to Shapiro (2000), influential statistician John Tukey coined the term 'software' in 1958 in opposition to the term hardware (which was already in use). For further details on the changing meaning of the term see Haigh (2002).

pragmatic distinction and a relative notion. According to him, the prefixes 'soft' and 'hard' refer to a person's ability to make changes:

> "At one extreme if at the factory a person who replaces circuits in the computer understands the activity as giving instructions, then for him a considerable portion of the computer may be software. For the systems programmer who programs the computer in machine language much of the circuitry will be hardware" (Moor 1978, 215)

If this assessment is correct, then software could be a tangible good as long as it is changeable. Yet others (albeit later), arrived at the puzzling conclusion that software is hardware (Suber 1988) or that software doesn't exist at all (Kittler 1993). (To avoid further conflations, I will avoid using the term 'software' in my analysis from here on).[4] You are at a loss – all of this is so confusing! Not only are you unsure where programs fall into the software/hardware dichotomy; you don't even know whether your choice was a good categorization scheme to begin with.

Worse, the longer you think about the matter, the more conceptions of how to conceive programs pop up in your mind: Many programs are formulated in special kinds of languages like ALGOL or FORTRAN, so aren't they some special kind of text?[5] And where do mathematical abstraction and the algorithms from Gottschalk v Benson fit into this picture? Determining the nature of programs seems to be a tough nut to crack. What are the morals we can draw from this story?

Let me pause here for a minute. Although the IP lawyer game of make-believe is entirely fabricated, the events and considerations described in this episode are not. From the mid-1960s onwards, the lack of IP protection for computer programs became a growing concern. With the incentive to protect these costly new inventions, the industry had a strong interest in settling the issue (in their favor). However, like our imaginary IP lawyer, courts having to decide how to legally protect software products struggled with determining its nature and characteristics. As Con Diaz describes, the legal debate (in the US) became a "doctrinal minefield" since no proposal for computer programs satisfied every stakeholder involved (2019, 6). Legal outcomes in favor of patents or copyrights hinged on whether the Patent Office and Courts judged programs to be machines, texts, or algorithms (ibid., 100).

---

[4] See Duncan (2014) for an extensive discussion (including the 'implausibility' of Moor's and Suber's arguments) on whether the software-hardware distinction can be maintained after all.
[5] In the US, the first deposit of a computer program for copyright registration of a program was in November 1961 (North American Aviation submitted a tape containing a program). Perhaps the first successful registration attempt was the FORTRAN program called 'Gaze-2, A One-Dimensional, Multigroup Neuron Diffusion Theory Code for the IBM-7090.' Hollaar (2002, I.B.).

In fact, in the mid-1970s matters became so pressing that US Congress installed a commission to settle the patent/copyright implications of information technologies. Consequently, 1974 saw the formation of the Commission on New Technological Uses of Copyrighted Works (CONTU). The commission was made up of experts from all different strives of life (yet remarkably lacking expertise in computing though) and took almost four years to submit their final report. CONTU eventually reached the unanimous decision that computer programs are entitled to legal protection, but alas "the unanimity has not extended to the precise form that protection should take." (CONTU, 12). While ultimately settling with the recommendation that programs ought to be protectable under copyrights, (leading to the Computer Software Right Act in 1980), the outcome was controversial. Noteworthy, the final report contained sections of dissent from one of its very own members: commissioners John Hersey.

Hersey, Pulitzer Prize winner and president of the Author's League of America stated that

> "[t]he heart of the argument lies in what flows from the distinction […] between the written and mechanical forms of computer programs: admitting these devices to copyright would mark the first time copyright had ever covered a means of communication, not with the human mind and senses, but with machines." (CONTU, 28)

In Con Diaz's words, Hersey believed that computer programs thus had some kind of 'hybrid nature,' for they seem to combine written and mechanical elements at once.

Even years after the implementation of CONTU's recommendation, the outcome sparked remarkable dissent. Allen Newell, a pioneer in computer science and cognitive psychology complained that the models that the law experts came up with were broken. At the end of his critical essay, he concluded:

> "I think fixing the [ontological] models is an important intellectual task. It will be difficult. The concepts that are being jumbled together-methods, processes, mental steps, abstraction, algorithms, procedures, determinism- ramify throughout the social and economic fabric. I am not worried about how new and refurbished models, if we could get them, will get back into the law. They will migrate back by becoming part of legal arguments, or legislation or whatnot. There are many different paths. The task is to get the new models. There is a fertile field to be plowed here, to understand what models might work for the law. It is a job for lawyers and, importantly, theoretical computer scientists. It could also use some philosophers of computation, if we could ever grow some. It is not a job for a committee or a commission. It will require sustained intellectual labor." (Newell 1986, 1035)

In this dissertation, I take up Newell's suggestion and set out to shed light on the puzzling nature of computer programs with a philosopher of computation's hat

on. More precisely, I shall do away with the legal battles and instead address their underlying philosophical question

> **Main Research Question:** *What is the ontological status of computer programs?*

1.2 Concretizing the Problem: Preliminaries & Diagnosis

Before explaining the primary strategy and guiding idea of addressing my research question, I need to make a few refinements. In the following, I will take three steps to set the stage and clarify the problem related to my research topic. First, I will discuss the overall relevance of this undertaking for philosophers and computer scientists. Second and third, I will explain the two constituents - ontology and computer programs - that define the main research question.

Relevancy Beyond Legal Controversies

Given that the Chimera of computer programs has riddled lawyers, computer scientists, and (some) philosophers for more than 50 years, but there are no signs of stoppage for the success of computing, what's the relevancy of this thesis project?

Despite the awareness of the problem, the contemporary literature on the metaphysical nature of computer programs remains rather short-supplied. Concerning other 'scientific' disciplines like Physics or the Life Sciences, the Philosophy of Computer Science is comparably small-scale. This raises questions about why we need such an endeavor in the first place and why inquiries about the topic matter (besides legal issues). I believe there are two types of answers to this.

On the one hand, there are answers justified from within philosophy. For philosophers, an entity that seems to evade standard metaphysical categorization is interesting. Just as metaphysicians pursue the study of what kind of things, say, artworks or technological artifacts are,[6] they might consider the non-straightforward case of computer programs, too. Studying programs might unearth loopholes in given metaphysical frameworks and thus contribute to some philosophical progress (especially within ontology and metaphysics).

On the other hand, properly characterizing computer programs has broader implications beyond metaphysical inquiries. Today, the application of computer programs is so pervasive that a clear understanding of their nature possibly benefits virtually every domain using them. For instance, computer programs are

[6] To briefly presage what's to come, I will engage with both characterizations of artworks and technological artifacts to shed light on the nature of programs.

foundational for fields like AI, robotics, increasing ethical concerns, and (of course) computer science. Without a clear philosophical underpinning, such discourses are in danger of significant conflations and category mistakes (cf. Daylight 2016, 14-16).

One of the most prominent of such category mistakes at large occurred roughly a decade after CONTU submitted its report and US Congress implemented its recommendations: In the early 1990s, a series of exchanges between the philosopher James Fetzer and several computer scientists in the prestigious *Communication of the ACM* unsettled many computing academics. At its core, the dispute concerned the verification of the correctness of computer programs. Commonly, correctness describes a special relationship between a program and its specification: a *proof* in (formal) program verification aims to verify that the program *text* (a set of instructions) matches the formal specification. Fetzer (1988) argued that the notion of 'program proof' suffers from a category mistake because it may only apply to idealized abstract machines but not real-world systems. Executing a program on the latter is a physical process that causally affects the behavior of material computing systems. A proof, however, is a concept that applies to the formal, abstract realms of logic and mathematics; it cannot establish the properties of a program as a causal entity running on a real, physical machine.[7]

Considering the ferocity and prevalence of the debate, one may expect that the dispute sparked a research program to settle the underlying issue. Yet, despite the matter that there is a problem at stake, a systematic (philosophical) explanation describing how to solve it did not emerge. Merely a handful of pages in, we already encountered several potential conflations that obfuscate finding a straightforward answer to a seemingly simple question. Differing conceptions are still 'jumbled together' today, resulting in abounding conflations. To make a long story short, the state of the art is scattered so that computer programs are placed in nearly every available ontological category (see Appendix A).[8]

In short, some believe that programs are physical objects or processes, while others view them as abstract logico-mathematical objects or special types of texts. Yet others argue that programs are technical or abstract artifacts, while others suggest that naturalized programs even constitute our minds. Despite the abundance of views, there is currently no consensus on the metaphysical nature of programs and how they should be classified. The problem, as we have seen in

[7] For summaries and critical analyses of the debate see (Colburn 2000, 135; MacKenzie 2004, 210-218; Tedre 2015, Ch.4).

[8] Since lengthy literature review sections are dull, I omitted many details for the sake of the readability of the introduction. For those interested in an extended overview of the state of the art, consult Appendix A.

the debates about the patentability, and verification of programs, is that it is easy to find counter-examples and inconsistencies such that no position seems to be plausible. Instead, adopting the 'dual nature view' or hybrid perspectives is popular, where programs have a plural or liminal nature. In the mid-1990s, renowned computer scientist Michael Jackson epitomized this approach by stating

> "Because software seems to be an intangible intellectual product we can colour it to suit our interests and prejudices. For some people the central product of software development is the computation evoked. For some it is the social consensus achieved in negotiating the specifications. For some it is a mathematical edifice of axioms and theorems. Some people have been pleased to have their programs described as logical poems. Some have advocated literate programming. Some see software as an expression of business policy." (Jackson 1995, 283)

I agree with Jackson that different communities and often even the same computer scientists, programmers, and users 'encounter' programs in all these guises in their practical work. However, per se, dualism or hybrid views do not dissolve the ontological question. The problem is, that without further explanation, these notions appear to be ad hoc answers that stand at odds with contemporary metaphysical orthodoxy.

One crucial first aspect to rectify the situation is breaking down the criteria that will license us to draw metaphysically sound conclusions. The next sub-section will clarify and narrow down these criteria.

Traditional Metaphysics & Category Systems

Ontology and Metaphysics address a wide range of questions (van Inwagen et al. 2023). Let me hence make more precise what this thesis is and is not about. The way I will conduct my metaphysical investigations are largely in line with contemporary analytical philosophy. For instance, according to Hofweber (2016, 8f), there are generally two main metaphysical questions – *primary ontological questions* (POQ) and *secondary ontological questions* (SOQ). Fine (2017, 98) echoes this characterization by maintaining that metaphysics can roughly be distinguished between Ontology and Metaphysics proper. Following Fine, I will keep referring to this practice as *traditional metaphysics*. By combining Hofweber's and Fine's position, something like the following picture emerges:

(POQ): Ontology poses the question of *what there is.*

(SOQ): Metaphysics proper investigates the *nature of what there is.*

At first sight, it appears, that metaphysicians typically must explore what exists *before* enquiring into its nature. Ontology seemingly precedes Metaphysics

proper because what does not exist cannot be investigated philosophically in a meaningful way.

Importantly, for the current undertaking the distinction between (POQ) ontology on the one hand and (SOQ) metaphysics proper on the other raises the following concern: Do we need to answer (POQ) about programs in the affirmative before we can proceed with (SOQ). In other words, we need to address the question '$(POQ)_{Prog.}$: Are there computer programs?'

If the answer is 'no,' the issue would be settled straight away and there would be no point in continuing the nature of programs if they don't exist. The broader implication would then 'simply' be that computer scientists (and lawyers) have it all wrong and that the collective idea about computing and how it shapes nearly every facet of our modern life is largely incorrect. If the answer is 'yes', then I can carry on with wondering what programs are like.

Now, my strategy to tackle these issues with respect to computer programs is to answer with 'yes, there are computer programs' and directly proceed with the secondary ontological question of what they are like. Answering this way is not meant to say that there are no deep-rooted philosophical issues at stake. On the contrary, aiming to arrive at judgements about primary ontological questions (e.g., are there numbers? Are there universals? Are there *really* everyday-objects like tables and toaster?) is notoriously contentious in philosophy. However, this is simply not the place to resolve these longstanding issues. Neither do I endorse a specific well-founded framework that defends my answer against skeptical metaphysicians nor do I want to engage with the daunting task to develop such a framework.[9]

However, for the success of this thesis project, there is another crucial issue we must reflect on: Specifying an ontological category system. Notwithstanding, developing or choosing a universal ontological classification scheme is challenging. The problem has its roots in Antiquity and persists until today – Aristotle's *Categories*, a seminal work in this field, has influenced numerous philosophers, including Aquinas, Descartes, Spinoza, Leibniz, Locke, Berkeley, Hume, Kant, Hegel, Brentano, and Heidegger (Studtmann 2024). Although these

[9] For those not convinced by the way I bracket primary ontology questions, it might be appeasing to know that it is not completely unwonted to engage with metaphysics proper first. One might start determining what the object of inquiry would be like, if there is such thing, and then use the result to answer the primary question in the negative (Hofweber 2016, Ch. 1.3). Recently, Steven French (2020), for instance, followed this strategy and concluded that 'there are no such thing as (scientific) theories'. While (spoiler) I won't reach such drastic conclusions, I invite the skeptic to see my thesis under the conditional that programs exist – I believe the content of this dissertation is informative and can be understood nevertheless. Interestingly, to the best of my knowledge, the only source maintaining that programs don't exist is Kittler (1993).

are only broad sketches, it is, not surprising that there is little agreement among philosophers on a more than two-thousand-year-old debate on what precisely an ontological category and a system thereof is. Despite enduring metaphysical controversies, I will from here on assume that an ontological category is a *kind of being* in which things might be claimed to exist (Lowe 2006, 20; cf. van Inwagen et al. 2023, §2.2 for a similar characterization). At least for the current purpose, this understanding of 'category' will be innocuous enough to proceed without major quarrels.

Following this, an ontological category *system* is a structured classification scheme of kinds of beings that ought to provide a complete inventory of what exists. The advantage of a pre-conceived system is that it allows us to make consistent metaphysical judgments about all kinds of entities under the scrutiny of SOQ. By the same token, it becomes the thesis' central motif to spell out the membership of computer programs in one of the systems' categories. Notably, this view also clarifies what this thesis is not about – the identity criteria of programs. Typically, identity concerns are one of the central features of metaphysical discourse. One may think of well-known thought experiments like 'Lumpy' or the 'Ship of Theseus.' Although I believe that this topic deserves more attention in some future research, I will, as much as possible, abstain from engaging with questions such as 'When are two programs the same? Does a small change in one line of code create an entirely new program?' and so on.[10] Attempting to place the notion of computer programs in a category system can be treated independently of controversies of their identity.

There is another problem worth considering though – not everyone subscribes to realism about category systems. Throughout the 20th century, many philosophers expressed their skepticism about the pursuit to find a fundamental/universal category system. In line with this thinking, it has become popular to engage in what Thomasson (1999, 116) has called a 'piecemeal approach', i.e., examining each purported type of entity separately and anew again. Anyhow, even if one thinks that this development is misguided, it is left open which category system should be the chosen one. As per Lowe (2006, §1.3 and §2; see also Thomasson (2022, §1.4)), there are various competing ontological systems available. Given this diversity and the lack of consensus, we encounter the following problem

Problem I: *Selecting an appropriate category system*; piecemeal or systematic approach?

[10] See White (2004), Cardone (2021), and Angius& Primiero (2018; 2023) for some recent attempts.

Before moving on, I want to mention one more caveat. Recently, the term 'ontology' has also gained popularity in computer and information science. In this context, 'ontology' has a different, rather descriptive connotation and can be understood as a taxonomy, i.e., a standardized framework that provides a set of terms for consistent data description and annotation across different research communities. Put differently, what distinguishes these ontologies from the category systems in the metaphysical tradition is that they do not set out to provide a *fundamental* category of being. Nevertheless, the resulting ontologies have significant practical benefits, as they promote consistency in data description and facilitate communication across disciplinary boundaries (interoperability). Examples include 'Gene Ontology,' 'Infectious Disease Ontology,' 'Plant Ontology,' and others (Arp et al. 2015, xxi).

A collaborative effort between philosophers, computer scientists, and information scientists has created globally applicable ontologies across different domains to keep up with these developments. Notable examples of this interdisciplinary work are the Basic Formal Ontology (BFO; (Arp et al. 2015)) and the Descriptive Ontology for Linguistic and Cognitive Engineering (DOLCE; (Gangemi et al. 2002)). I mention this development to steer away from potential misunderstandings since this thesis deals with computer science broadly construed, i.e., a field where these types of ontologies find widespread application. Testimony to this is the information-scientific flavored approaches that have also been applied to study the nature of computer programs (, Lando et al. 2007, Duncan 2014, Wang et al. 2014a, Wang et al. 2014b).

What's in a name?

When engaging in 'computer program'-talk, we should try to mean the same as the other participants in the debate, or miscommunication will occur. One helpful method to get a grip on the matter is by elucidating the term's etymology: The word 'program' (or 'programme' in British spelling) has its roots in the Greek word προγραφιν, consisting of προ ('before,' or 'pre') and γραφιν ('to write') Grier (1996, 51). As such, it did not originate in a computing context but underwent considerable transformations throughout history. De Mol & Bullynck (2021) explain that the word was generically employed to refer to a planned series of future actions or events. We still find common examples such as TV or radio programs, political programs, research programs, or training programs that are virtually used in the same vein today. Parallel to the description of a sequence of scheduled events in everyday life, the meaning of 'program' shifted in multiple engineering contexts where it began to designate different technological aspects. For instance, in the 19th century, so-called 'program clocks' were used to

automate time schedules in work environments or schools. In radio engineering, 'program' was also used to denote the electronic signals used to broadcast TV and radio programs at least from the 1920s onwards (Grier).

Like many other 'historical firsts,' the first instances of the name 'program' in a computing context are contentious. While some locate the origin within the ENIAC project (Grier 1996; Haigh & Priestley 2016), others identified other calculating machines of the late 1930s and early 1940s (e.g., the IBM ASCC/Harvard Mark I) to be already entangled with the term (De Mol & Bullynck 2021). Either way, the methods to plan sequences of computations for the automatic control of computing devices frequently relied on prior established technologies, like e.g., punched cards used for Jacquard looms and desk calculators. In this context, the term originally referred to a great variety of activities, designating how automatic control of computers could be organized on different scales.

From the 1950s onwards computing developed towards reliability, mass production, and standardization, and there were increasing attempts to determine common practices and define basic terms like 'program' in glossaries (De Mol & Bullynck 2022). One aspect that went hand in hand with this development/early professionalization of the field was that the configuration of computers became increasingly associated with formal languages closely related to logic and linguistics (Nofre et al. 2014). Roughly put, the development of new programming languages was gradually more detached from the details of specific machines. The ensuing language metaphor enabled one to regard programs as notations, strings of symbols, or special texts that could be studied and reasoned about independently of the underlying circuit settings. As Vee (2013) describes, for instance, this resulted in comparing programming to the notion of 'literacy', emphasizing the importance, flexibility, and power of writing with and for computers.

However, the characterization never fully stabilized and it would hence be a mistake to simply consider programs as mere texts or linguistic entities. The reason for that instability is that from the dawn of computing, the discipline's nature has been a matter of great concern. As many commentators have noted, computer science draws knowledge and methods from many fields. Wegner (1976), for instance, looked into separate characterizations of computer science as branches of (i) mathematics, (ii) engineering, and (iii) empirical science. The influential report 'Computing as a Discipline' by Denning et al. (1989), investigating what constitutes computer science qua discipline, echoes the tripartite distinction (i)-(iii). Today, partitions along essentially the same lines are still highly influential for discourses regarding the historical discipline-building

of computer science and how to give an adequate characterization of the foundations of computer science (cf. Eden 2007, Tedre 2015, De Mol 2015, Schiaffonati & Verdicchio 2014). The debate about the nature of the discipline and its scope has persisted for decades and continues to this day.

Owing to this epistemic pluralism, many central notions in computing bear a surprising amount of semantic ambiguity: Smith (1996, 73-74), for instance, comes to the sobering conclusion that there is no distinct ontological category that deserves to be called computation. More recently Pappayannopoulos (2023) argued that there are at least two conceptually different notions of algorithms identifiable in the literature. By the same token, the word 'program' is semantically indeterminate, too. Although being a widespread entity/ phenomenon, no single rigorous definition has gained traction: None of the usages of 'program' is universally accepted; available characterizations are not entirely co-extensional. Even though this parallel use of the term is somewhat unfortunate, it is well-established in the relevant literature and largely unproblematic for *practice*.

When engaging in philosophical business, ambiguity and equivocation are the sort of things one needs to avoid though. In fact, worries like these fuel another form of ontological skepticism which threatens forming a coherent metaphysical judgement of semantically indeterminate entities like programs: *Neo-Carnapianism*. Simply put, Neo-Carnapianism maintains that a considerable part of ontological problems reduces to verbal disputes.[11] According to this *deflationary* stance, there is no serious or actual problem underpinning all sorts of ontological questions. The idea is that epistemic agents from different linguistic, or cultural communities merely disagree about what a given term refers to (and not about what kind of being it is). Let's take 'football' as a toy example. An English-speaker from North-America would likely conclude that 'footballs are pointy,' whereas an Englishmen may argue that 'footballs are rather round.'[12] Even though both speakers use the same word they mean different things (a ball used in American football and a ball used in 'soccer'). The North-American and Englishmen come to different conclusions about the shape of footballs because they maintain a different linguistic framework. Yet, once the equivocation of the term 'football' is clarified, the argument about the properties of the objects and activities referred to, resolves too.

The point is that comparable forms of semantic ambiguity are pervasive beyond sports-vocabulary. Accordingly, dictionaries may classify countless

[11] The original seminal paper is Carnap's (1950). More recent versions of the idea that ontological disputes are due to different language frameworks, especially different quantifiers.
[12] The example is inspired by Effingham's introductory textbook on Ontology (2013, 169).

terms (from whichever domain) as *lexically ambiguous*. There are different types of ambiguity Sennett (2023), but typically, linguists and philosophers distinguish between two subspecies: While *homonomy* describes the accidental encoding of multiple meanings in the same sign or term (e.g., in English 'rock' may denote a 'stone' or a music genre), *polysemy* refers to a linguistic expression with multiple, albeit *related*, senses (Falkum & Vincente (2015), Sennett (2016), Carston (2021)) – as we have seen in the case of 'football.'[13] Distinguishing the two species of ambiguity is not always easy and several linguistic tests have been devised to identify polysemic terms.

When applying such a test to 'computer program,' it becomes clear that the term is an example of polysemy, too. For instance, we may help ourselves with the following statement to uncover (part of) its polysemic nature:

"That program is well written/beautifully coded. It runs fast."

Here, the pronoun *it* refers anaphorically to the physical object (particularly its execution), whereas the sense of program in the previous sentence is used in a textual sense. In fact, the semantic extension of 'program' entails several other, but related meanings, including what I call the Physical View, the Mathematical View, the Symbolic View, the Artifact View, and the Neural View (see Appendix A).[14] Although this is not the time and place to deep dive into the overall plausibility of the Neo-Carnapian rationale, the upshot of this brief discussion should be clear: Since the term 'computer program' forms a *polysemic web* of various ontologically different (but related) things, deflationary arguments and linguistic confusion may also hamper this thesis' undertaking.

There are a couple of ways to respond to this. Absent semantic identity, one seemingly obvious possible future path for the community would be attempting to converge towards one unique usage of 'program.' One may envision this to work similarly to how Lakatos (1976) describes the progress of mathematics, exemplified by rigorously characterizing the proof of the Euler characteristic defined for the polyhedron.[15] However, unlike Lakatos' polyhedrons, the notion of 'program' seems to undergo significantly faster changes than other disciplines'

[13] Importantly, both phenomena need to be distinguished from 'vagueness.' Usually, the notion is associated with the occurrence of borderline cases and the sorites paradox, e.g., 'when is a heap of sand no longer a heap of sand?' (Hyde & Raffman 2018) and (Sorensen 2023).
[14] Strictly speaking, the term 'program' also bears another form of polysemy, namely of cross-categorical nature (in a grammatical sense): 'program' as noun, and as verb (as in, 'to program a machine') with related senses across these grammatical categories. However, I won't further engage with this polysemic dimension.
[15] Anecdotally, in one of the various workshops of the PROGRAMme research group, we tried developing a comprehensive definition of 'computer program' in a brain storm session. Culminating in 'a layout of signs aimed at determining the behaviour of a machine.'

central concepts. Due to its instability over time, we constantly try to define a moving target and would, at best, only get a snapshot. Moreover, this 'precisification approach' would arguably fly in the face of computer science's current practice to successfully embrace epistemic pluralism. Therefore, I consider trying to define the term 'program' as moot.

Given the lack of rigor, we thus confront another central issue

Problem II: *Untangling the polysemic web of the term 'program.'*

The takeaway is that we must devise a strategy that blocks linguistic confusion creeping into our metaphysical investigation or otherwise we may get as many potential answers about the ontological status of computer programs as there are different meanings hidden in this polysemic complex.

1.3 The Project's Guiding Ideas

Let me summarize what I have discussed so far. I introduced the main research question in the previous sections and explained its relevance. No consensus about programs' metaphysical nature has been reached; scholars of different strives have characterized them in multiple, often contradictory, ways. By construing the question's main constituents – (i) ontological status and (ii) computer programs – I provided some necessary background about metaphysical investigations and clarified what kinds of things the term program picks out. This diagnosis unearthed two primary problems:

1. **Problem I**: On the one hand, we must be open about choosing a suited ontological category system.
2. **Problem II**: On the other hand, there is the polysemic nature of computer programs. When pressed into service in different contexts, 'program' fragments into several ontologically distinct and more precise concepts, each appropriate for its area of application.

What is the most comprehensive and effective response to these problems that will allow us to proceed fruitfully?

The Guiding Idea

My strategy for remedying the situation is by embracing the polysemic web we confront, head-on. Previously, I have said that polysemic terms bear at least two related senses. In the case of 'computer program,' multiple related senses are bundled together. What makes the case particularly urgent is that many of the senses that are thus related have different ontological flavors. Without handling this ambiguity well, we risk repeating past mistakes and are bound to commit

category mistakes. But instead of trying to untangle the situation by developing a rigorous definition, the general theme of my approach is different.

My guiding idea is to explicitly focus on the *relations* between all the ontologically different relata hiding behind this polysemic web. To clarify, the relata I am talking about are the ones that occurred in my previous analysis and the literature review (cf. Appendix A), the ones deemed responsible for the alleged duality/pluralism of programs: On the one hand, there is the domain of abstract, formal, and mathematical objects. On the other hand, there is the domain of the physical, of concrete systems, of events and processes unfolding in space and time.

Specifically, I believe the notion of implementation is vital to understanding how these entities connect. When I say 'implementation,' I refer (as a first stab) to the relationship between different computational domains. In addition, my thesis argues that agents play a critical role in mediating implementation. I will elaborate on both ideas extensively in the following chapters (see also Appendix B), but here is a graphic depicting the situation to get the gist of it (Fig. 1.1). Whereas the file icon stands for program texts, the laptop icon typifies a physical computer; both are related by the downward pointing black arrow (representing computational implementation). Moreover, all three items critically depend on epistemic agents (depicted by the black mannequin) and their practices.

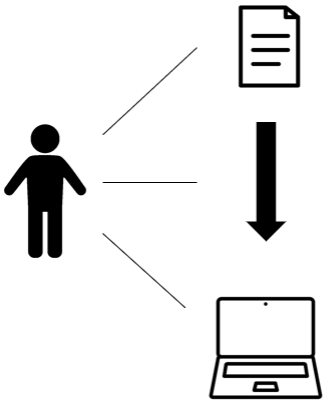

Fig. 1.1: Schematic depiction of the guiding idea of this dissertation.

One promising way of ontologically characterizing this network of relata and their relations is to establish an initial category system and make modifications and refinements as needed along the way (in terms of the terminology I previously used, one can regard this as a partial piece meal approach). This

approach is particularly effective when developed against the backdrop of widely recognized distinction between the 'abstract' and the 'concrete' the in philosophy (Falguera et al., 2022). It is effective, because typically, the abstract/concrete dichotomy is meant to be a simple but exhaustive two-category system – accordingly, every entity must either be abstract or concrete. Commonly, philosophers agree that objects like rocks, tables, or tigers are concrete. In contrast, mathematical objects (e.g., numbers, pure sets, and perhaps 'programs') are typically thought of as abstract entities. Other entities standardly considered to fall into the abstract side of the dichotomy are universals, propositions, types (as opposed to tokens), and – more controversial – fictional characters (e.g., *Sherlock Holmes*, or *Donald Duck*).

Arguably one of most common ways to characterize the abstract is then along these lines:

> **Abstract:** An object is abstract *iff* it has *no spatiotemporal location* and is *causally inefficacious.*

The lesson learned from this 'standard' view is that entities like mathematical objects are considered abstract objects, since they are none locatable in space-time and cannot be integrated into the causal pathway (for instance, one cannot stump their foot on √2). Similar considerations hold for other candidate abstract objects such as universals and propositions. In his *On the plurality of worlds* David Lewis (1986, 1.7 §) called this proposal of characterizing the abstract-concrete dichotomy the 'way of negation:' In contrast to concrete objects, abstract objects are construed by being *non*-spatiotemporal and acausal.

However, to further deepen our understanding on how the abstract-concrete distinction bears on the metaphysical nature of computer programs, it is illuminating to take a closer look at one of Lewis' other ways of drawing the distinction – *the way of abstraction*.[16] The reason for considering this notion is because it resembles how abstraction is standardly used and talked about in (computer) science. Arguably, it is here where conflations about different conceptions of abstractness may happen most frequently, since this way of characterizing abstraction seems to be the one most in line with historic use and the etymological roots of the term.[17] According to Lewis' way of abstraction

[16] In total, Lewis identified four methods to draw the line between the abstract and concrete; (i) the way of example, (ii) the way of conflation, (iii) the way of negation, and (iv) the way of abstraction. Today, (i)-(iv) are still often used to chart the different approaches of the abstract (Falguera et al. 2022).

[17] For the computing pioneer Dijkstra, for instance, a program is "[…] an *abstract* symbol manipulator, which can be turned into a concrete one" (Dijkstra 1989, own italics). In contrast, Colburn (1999) maintains that programs are "concrete *abstractions*." Are they talking about the same kind of abstractness?

> "abstract entities are abstractions from concrete entities. They result from somehow subtracting specificity, so that an incomplete description of the original concrete entity would be a complete description of the abstraction." (Lewis 1986, 84-85)

Under these circumstances, the 'abstract object' purports to hinge on the *mental* process in which concepts are created by omitting properties of one or several objects. Put differently, at least without further qualifications, the way of abstraction stands at odds with the standard way of characterizing the abstract-concrete distinction since the former requires some *epistemic* process, whereas the latter can be characterized mind-independently. It is therefore doubtful to what extent the way of abstraction can serve as an explanation of the origin of abstract entities like mathematical objects on *purely* ontological grounds.

As an epistemic notion though it abounds in science: Here the process is useful for the analysis of complex systems by reducing (irrelevant) properties or information. Frequently, this epistemic operation is referred to as *Aristotelian Abstraction*. Cartwright (1989, 197), for instance, portrays Aristotelian abstraction as an act through which one "strip[s] away—in one's imagination—all that is irrelevant to the concerns of the moment in order to focus on a single property or set of properties as if they were separate." This way, scientists get a handle on studying systems that would otherwise be too complicated.

'Abstraction' in computer science is a variant of (Aristotelian) abstraction, in so far as it is the operation or process of omitting one or more features of a complex object/system. A common story of how abstraction facilitates dealing with complex computing systems goes like this: Suppose you must program a first-generation digital stored computer. When configuring such a device, programs and data were encoded in a notation that closely corresponded to the given machine's hardware. Setting up computational devices in binary or machine code was cumbersome, error prone and required great ingenuity to 'translate' whichever problem was supposed to be solved computationally into machine code. As we will see later in more detail, the late 1950s brought forward types of encodings (nowadays widely referred to as programming languages) that enabled the programmer to describe sequences of computations through notations and formulas that were somewhat more *decoupled* from the underlying circuitry. At a first pass, instructions formulated in high level programming language have a higher degree of abstraction than logically equivalent machine code instructions, because they reflect less hardware details. These more 'abstract' languages have several advantages – e.g., they avoid unnecessary

machine dependence, and they are easier to read and write for the humans that devised them.[18]

Notably, one may further abstract away from already made abstractions, giving rise to different *Levels of Abstraction* (LoA). Beginning in the 1970s, such *levelism* became a prominent feature in (philosophy of) science, especially in connection to computing or disciplines where computational methods where employed. For instance, in his well-known *Vision* Marr (1982) suggested three levels of explanation for complex systems (such as our perceptual apparatus): (i) a computational level; (ii) an algorithmic level; and (iii) the implementation/hardware level. Importantly, such levels form a sort of hierarchy that is often characterized by different degrees of abstraction.[19] In other words, by leaving out certain details, one may reach a level that is more suitable for explaining the phenomenon of interest.

Throughout the history of computer science, the LoA concept has been pervasive and evoked a great deal of level talk. Primiero (2016; 2020), building on the work of Floridi (2008; 2011, Ch. 3), has arguably devised the most comprehensive notion of LoA suitable for computing. I emphasize this because Primiero's account supposedly provides both an epistemological and an ontological hierarchy. According to the latter, computational systems are stratified or layered entities in the sense that they are composed of various LoA. While I will go into more detail about this account in the following chapter, it is paramount to note some things here for clarity's sake.

While I am a proponent of the LoA view and its merits, it is crucial to exercise caution when interpreting the 'ontological hierarchy' it presents. This hierarchy should not be confused with a fundamental metaphysical one (one should instead think about it along the lines of the descriptivist spirit of ontologies in computer science I introduced earlier). Floridi, for instance, reminds us that LoA generally do not give rise to ontological levelism. When discussing an example, Floridi clarifies

[18] Given its usefulness in making complex systems tractable, there are many more instances where abstraction plays a role in computing. Data abstraction, the act of hiding irrelevant details in a data set, is another example. Accordingly, Donald Knuth (1997, Ch. 2) explains how abstraction enables us to systematically think about data structures (e.g., as a list, stack, or tree). More recently, Kramer (2007) advocates that the skill of forming abstractions correlates with being a successful software engineer. Colburn and Shute (2007) contrast the process with the notion of abstraction employed in mathematics, arguing that the former relies on information hiding, whereas the latter utilizes information neglect. Angius (2013) illuminates software verification through the lenses of abstraction (and idealization), and recently, Turner (2021) provided a more rigorous account by importing a modification of Frege's approach to abstraction into type-theory.

[19] N.b., 'abstraction' is by no means the sole feature responsible for constituting different levels. See Craver (2014) for a recent survey about different conceptions of levels.

> "I have shown how the analysis [of an example] may be conducted at different levels of epistemological abstraction without assuming any corresponding ontological levelism. Nature does not know about LoAs either." (Floridi 2008, 35)

The takeaway from concluding our discussion on epistemic/Aristotelian abstraction and its associated concept of LoA is that relying solely on this framework may not provide immediate answers about the ontological status of computer programs.

In carrying out this research program, I will make several substantive claims. Here is a brief selection of the most central ones:

- I will claim that at least two quite distinct notions of implementation require integration/unification for understanding the ontological status of computer programs.
- I will claim that, the garden variety of accounts of physical computation do not work (straightforwardly) when applied to computer programs. Especially naturalized accounts suffer from having turned a blind eye to the metaphysical nature of implementation *qua* relation.
- I will claim that appropriating some of the major insights and conceptual tools of scientific representation and modelling vindicate interpretational or agential theories of implementation.
- I will claim that the abstractness of computer programs is best understood through the so-called Problem of Creation and does not require *sui generis* solutions.
- I will advance a novel notion of physical programmability, specifying the conditions under which a system can be viewed as programmable.

1.4 Outline

From here on, the thesis contains four principal chapters, a conclusion, and three appendices. Here is how they unfold:

In Chapter two I start off with providing the framework for the rest of the thesis. I begin by considering two hitherto largely independently treated notions of implementation. For the sake of better distinction, I will refer to them as type-(A) and type-(B) implementation. The former is based on the normative notion of function-ascription (with 'A' for ascription); the latter is named after the so-called bridging problem (with 'B' for bridging) from the philosophy of applied mathematics. Juxtaposing both notions shows that their scope overlaps at the abstract physical interface and may mutually enrich each other. Specifically, I submit that (A) and (B) can be unified by appealing to use-based accounts of computation: The two notions can be combined by the conceptual machinery of

the literature on scientific representation (particularly, when concerned with material models). The result is sketch of a unified theory of agential implementation (UTAI) with different dependency relations (labelled (a)-(c)), where these relations give rise to the subsequent chapters.

The third chapter concerns dependency relation (a). I elucidate this relation by comparing programs to so-called repeatable artworks. The similarities between musical compositions and works of literature are especially instructive. Like such artworks, programs have different representational modes (e.g., symbolically, mathematically, diagrammatic) and implementational media (e.g., ink on paper, chalk on a whiteboard, electrical signals, punched cards, etc.). As such, they appear to be abstract objects that also suffer from the Problem of Creation – a problem from the philosophy of art about art abstracta. By appropriating the problem's most promising solutions to the philosophy of computing, I offer a novel metaphysical blueprint for future studies about the ontological status of computer programs. The upshot is that the abstract nature of programs does not require dubious *sui generis* solutions (e.g., a 'dual nature') but can, in fact, be discussed in more familiar philosophical territories.

Thereafter, chapter four sheds light on dependency relation (b) of the UTAI framework. Accordingly, I vindicate interpretational accounts of physical computation. Specifically, recent agential approaches that couch implementation in terms of scientific representation are corroborated. I strengthen such types by the introduction of a novel notion: Implementation-as. Implementation-as is theoretically underpinned by Frigg and Nguyen's DEKI account, a formalized account of scientific representation relying on Goodman's and Elgin's notion of representation-as. The DEKI account is especially suited for this because it relies on a material model – the MONIAC (a special-purpose hydraulic analog computer). Accordingly, a formal characterization of implementation-as emerges. I maintain that this result is a philosophically robust account, since it satisfies the most important desiderata (objectivity, extensional adequacy, explanation, miscomputation, taxonomy) for accounts of computation in physical systems. The upshot is that physical computation occurs when agents use material systems as epistemic tools to compute a function. Application of this new framework is illustrated for the MONIAC (an analog device) and the IAS-machine (a digital computer).

Chapter five illuminates dependency relation (c) through the notion of programmability. The philosophical discourse regarding programmability is scant and largely underdeveloped. In particular, reviewing the literature uncovers that only a limited amount of scholarship has examined the physical properties that enable a system to be programmed. This is a sorry condition, for

we seem to be unable to fully answer such questions as: How are programs integrated into the causal nexus? What does it mean for a physical system to be programmable? In the interest of answering these questions, I develop the here newly introduced notion of physical programmability.

> **Physical Programmability:** The degree to which the selected operation of an automaton can be reconfigured in a controlled way.

Subsequently, the strategy of my chapter is to explain the significance of the variables in the above's characterization. Accordingly, the function of (i) automaton; (ii) selected operation; (iii) reconfigured in a controlled way (iv) the degree to which, are discussed in detail.

Finally, I provide a **conclusion** (Chapter six). I begin by summarizing the central findings of this dissertation in order to canvass how the results of different chapters have informed us about the ontological status of computer programs.

2 Towards a Unified Theory of Implementation

2.1. Introduction

It sounds like a cliché, but the implementation of computation is ubiquitous. Not only are we surrounded by everyday devices such as laptops and smartphones that run our software, but computation is also at the core of foundational questions in computer science, robotics, AI, and cognitive science. Despite its ubiquity in computer science and adjacent fields, implementation is typically left informal. It is often associated with the realization, instantiation, or concretization of a plan or idea, *relating* two objects or domains with one another.[20] Considering the rapid developments in theory, technology, and areas of application of computing, various philosophical studies conceptually reconstructed what constitutes the implementation of computation in their respective fields. In light of this epistemic pluralism, different notions of implementation, in fact, often have a significantly different intellectual heritage. Confronted with a plurality of theories of implementation, the time is ripe to taxonomize them, shed light on their relationship systematically, and attempt to build bridges between them whenever possible.

To begin this task, I consider two of the most prominent clusters of implementation of the last few decades. For tractability, I refer to these views as *type-(A) implementation* (with '(A)' for ascription/artifact) and *type-(B) implementation* (with '(B)' for bridging). Type-(A) implementation emerged from (the philosophy of) computer science, particularly the concerns about the verification and correctness of so-called computational artifacts like computer programs.[21] Much of the corresponding discourse is couched in terms of *function ascription* (in the teleological sense) and pertains to the relation between different abstract levels or structures. Type-(B) implementation, on the other hand, emerged from the philosophy of mind (broadly construed) and concerns the nature of computation *qua* physical process in material systems. This notion is paramount to determining which systems compute and which don't and is often discussed regarding laptops, brains, and even the whole universe. Virtually all

[20] Overall, there are various (pre-theoretic) understandings of 'implementation,' even occurring in the domains of art, language, or other affairs (Rapaport 2005).

[21] After the fiercely held *verification debate* in the late 1980s and 1990s in the *communications of the ACM*, it was apparent that the field would benefit from a philosophical underpinning of the notion of verification and correctness. For a collection of some of the key contributions to verification, see Colburn et al. (1993). For a critical assessment of the debate, see McKenzie (2004, 197-218).

(B)-accounts share the idea that the evolution of a physical, real-world system maps to sequences of formal/abstract computational states. Until now, (A) and (B) have mainly been discussed separately.

However, throughout this chapter, I argue that the philosophy of computing would benefit from a novel theory of implementation that promotes greater synergy between two conceptions. This motivates me to engage in a project with the primary goal of comparing type-A and type-B implementation, clarifying their differences, and proposing a unified theory of implementation.

Here is the roadmap: Section §2.2 provides some general remarks about computational implementation. Subsequently, section §2.3 introduces type-(A) implementation, while section §2.4 portrays type-(B). Section §2.5 juxtaposes both implementation types by discussing their most prominent features (teleological functions and the relation between levels). Although they appear to apply to different computing domains and have different purposes at first, they are conceptually compatible. In section §2.6, I take my undertaking to the next level by suggesting that the unification of these two concepts can be achieved through the conceptual tools of the literature on material models and scientific representation. The resulting synthesis suggests that computational systems are *epistemic tools*, i.e., material artifacts used by agents for computation. When using material artifacts (akin to material scientific models) for computation, agents impute mathematical functions and ascribe teleological functions to engage in a form of object-based reasoning. I shall refer to this view as a *unified theory of agential implementation* (UTAI). Lastly, I conclude (sect. §2.7).

2.2. A primer on Implementation in Computer Science

The *Oxford Dictionary of Computer Science* provides a useful characterization of implementation to begin with

> **Implementation:** "[t]he activity of proceeding from a given design of a system to a working version (known also as an implementation) of that system, or the specific way in which some part of a system is made to fulfil its function."

The *relation* between design and its working version applies to various computational formalisms. For having a common understanding of this implementation relation, it is instructive to remind us about computational formalisms. While they are definable in a large variety of ways, the computer

science literature typically features two main ways of presenting computational formalisms (Turner 2018, 190):[22]

1. *Programming languages*, like C, Python, etc.
2. *Machine Models*, like Turing Machines (TM), Finite State Machines (FSM), etc.

In the following, I use the term 'model of computation' *Mc* for both. Models of computation are logico-mathematical formalisms that enable us to encode an abstract sequence of computations through a programming language, a machine table, a transition function, and so on. For instance, formally, the concept of a Turing Machine can be characterized as a quadruple $TM = (Q, \sum, m, \delta)$, where Q is a finite set of states q; $\sum$ is a finite set of symbols; m is the initial state $m \in Q$; δ is a transition function that determines the next move δ: $(Q \times \sum) \rightarrow (\sum \times \{L, R\} \times Q)$. TM's transition function δ maps from computational states to computational states (De Mol 2021). Put differently, transition functions like δ, computer programs written in a programming language, or any corresponding notions in theoretically equivalent *Mc* allow for the encoding of a sequence of computations.

In order for a system to compute, it has to implement a sequence of computations encoded in a 'program'/transition function specified by a given *Mc*. In practice, computational formalisms are often embedded in a special sort of computational hierarchy, composed of so-called Levels of Abstraction (LoA) (Floridi 2008; Primiero 2016; Primiero 2020). Accordingly, the application of implementation in computation is wide-ranging. Examples are 'the implementation of an algorithm in a high-level programming language' or 'the implementation of machine code instructions in a real-world computer.' Such 'level-talk' is frequent in computer science – a concrete textbook example of various implementation stages of a program is pictured overleaf in Fig. 2.1 (a). Instances like these may be generalized and accordingly culminate in a view as depicted under label (b) in Fig. 2.1: a (stored-program digital) computing system is typically composed of various LoA forming a computational hierarchy.[23] At the bottom of the hierarchy, one finds a *physical* system comprising various material components and their specific arrangement (the hardware). If set up and configured correctly, the system may execute a predetermined series of concrete computations. At the top of the hierarchy, one may find the most *abstract* level,

[22] The description of computational formalisms is inspired by a similar presentation in Rescorla (2013). Notably, some unconventional models can compute incomputable functions for a universal Turing machine. I will skip considering these types for now.
[23] See Hennessy and Patterson (2014, Ch. 2) for a fully worked-out textbook example. See also Scott (2009).

the program's formal specification. As a first stab, we can understand implementation as the relation between the different levels in such a hierarchy, connecting an abstract level to a less abstract one.

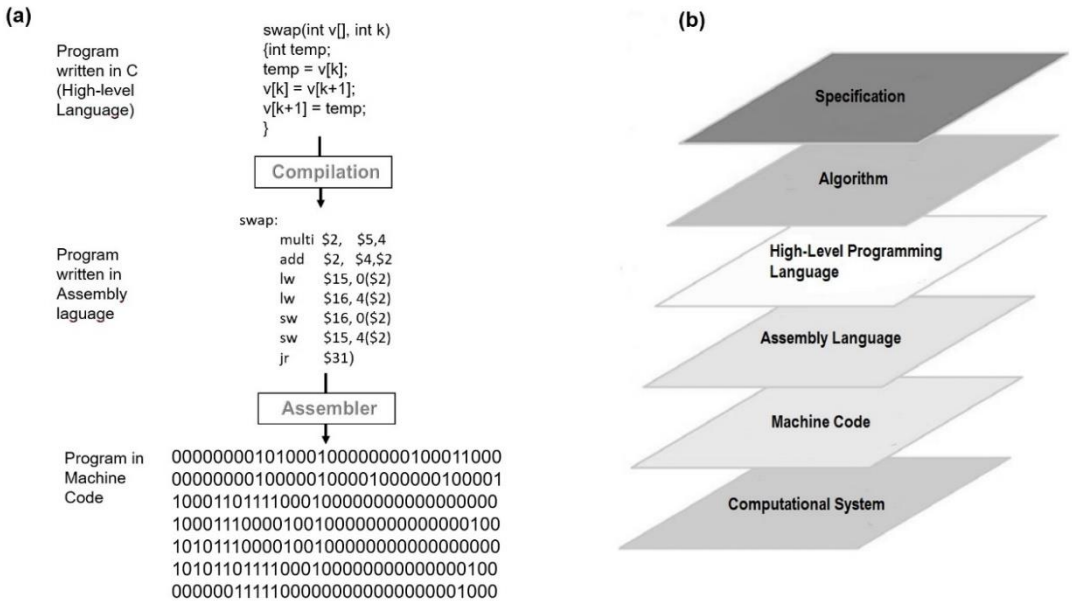

Fig. 2.1: Different depictions of the computational hierarchy. **(a)** A concrete instance of the different LoA and stages of implementation of a program written in C (example adapted from Patterson and Hennessy (2014, 15)). **(b)** Generalized image of typical LoA of a computer program.

'Abstract', in this context, has a double meaning. On the one hand, it refers to the degree to which language features are divorced from specific hardware details (Scott 2009, 111). In this sense, higher LoA entail fewer details about the underlying physical system. Based on that, we can understand abstraction as the inverse process of implementation. On the other hand, the computational objects corresponding to the different LoA may be abstract in a second sense. We may refer to them as abstract objects – as opposed to being concrete, material, or physical – since, as strings of symbols, they have no causal relations acting upon them. From this perspective, algorithms, e.g., are typically regarded as abstract objects.

Importantly, what follows from this brief discussion is this: Whereas the implementation of higher LoA is in the business of relating abstract objects (symbol structures), the last implementation stage is qualitatively different because it must relate an abstract symbol structure to a material system. The

program written in C in Fig. 1 (a) is an instance of the former, for it is a particular symbol structure that is translated into another one (i.e., compiled) into assembly language.[24] Analogously, when descending the computational hierarchy down to machine code, implementation is still a relation obtaining between different abstract strings of *symbols*. However, we require a different kind of relation at the abstract-physical interface – one that relates abstracta with concrete states of the putative physical system.

Today, two main approaches aim to cash out the requirements for connecting such different types of levels: type-(A) and type-(B) implementation. Somewhat surprisingly though, these two approaches are not in close contact with each other. My aim here is to change that. In what follows, I take a closer look at the philosophical characterizations of these implementation relations, beginning with type-(A) implementation.

2.3 Type-(A) Implementation

Perhaps symptomatic for a more general tendency of computer science, type-(A) implementation primarily focuses on the relation of upper LoA. Here, one central aim is determining the *correctness* of the various implementation stages. Two conditions are generally called for to meet the normative notion of correctness. On the one hand, a *formal specification* for the program and, on the other hand, a formal definition of the programming language's *semantics*. A program is then correct if there is a formal proof that the semantics of a program is consistent with the program's specification. Arguably, the first philosophical account of type-(A) is due to Rapaport (1999). He describes implementation as semantic interpretation (1999, 2005),

> **Implementation as semantic interpretation:** An object is an implementation of some syntactic domain *A* in medium *M iff* it is a semantic interpretation of a model of *A*,

i.e., a relation between semantics and syntax of different LoA. Rapaport claims that any correspondence between two domains where one is used to grasp the other is 'semantic correspondence.'

Rapaport's approach aims to allow both mere 'translations' of one programming language into another (symbolic implementation) and even for the qualitatively different case at the abstract-physical interface (i.e., the relation

[24] Alternatively, programs can also be translated through an interpreter, such that the source code is directly executed (line by line) without previously having been compiled into machine code.

between the bottom layers of the computational hierarchy).[25] While a program written in a high-level programming language may not be immediately implementable in a physical system, the so-called 'correspondence continuum' (i.e., roughly put, a notion of transitivity) is supposed to ensure their connection. For that reason, the program must go through a series of translation processes, where each time, a level that previously acted as a semantic domain turns into a syntactic one for another level below. At last, the 'implementation cascade' bottoms out at the physical level, providing the semantic domain upon which the semantics for all previous levels is built.

However, concerns were voiced about the way Rapaport employs his notion of semantics as a given, raising questions about whether an independent semantic account is required.[26] While the notion of semantic interpretation adequately describes *that* implementation requires semantics, it lacks the rigor to describe *how* these semantic features come about. The semantic approach does not explain how the physical level obtains its semantic capacities as the bedrock for the entire computational hierarchy. Therefore, philosophers of computer science suggested two improved accounts.

First, to account for an independent and external account of semantics, Turner considered the technical artifact literature and adopted the notion of *function ascription* (2012, 2014, 2018, 2020). Originally, the conception of technical artifacts and their functions should cover intentionally produced everyday objects like screwdrivers, coffee-makers, and trains (Kroes 2012). They are said to have a 'dual nature': Next to their respective causal/structural properties, this class of artifacts bears normative or teleological features. The function of a coffee maker is to brew coffee; a broken or malfunctioning coffee maker does not work correctly. Only when 'the how' (the structural properties) realizes 'the what' (functional properties) in the right way can one claim to have a properly working coffee machine. By thus transposing the core insights of the technical artifacts framework to computational entities, the conception of *computational artifacts* was born (Turner 2018). Accordingly, computational artifacts like programs exhibit a *function-structure duality* with

> **Implementation as function-structure relation:** The relation between specification (function) and the structure of the (computational-)artifact.

Importantly, artifact function here is an intentional notion derived from the use plan formulated by designers. The functions are bestowed to artifacts based on

[25] While Rapaport's conception of implementation is thus ostensibly applicable to the abstract-physical interface (the realm of type-(B) implementation), it does not consider its specific problems, which will be discussed in the following section (§2.3).

[26] For a more detailed summary of these arguments, see (Primiero 2020, 207f) and (Turner & Angius 2020).

the intentions and desires of human agents or an epistemic community. I will come back to the role of teleological functions later. For now, it suffices to acknowledge that a programmers' specification (an intentional notion) provides criteria for correctness and malfunction.

Second, following these developments, Primiero (2016; 2020) addressed issues with both the implementation as semantic interpretation and implementation as function-structure relation. The problem with both is that they merely provide an account of implementation for any two neighboring levels rather than the *entire* computational hierarchy. For instance, to eventually reach the bottom of the hierarchy (the physical system), Turner's version of the computational artifact approach relies on repeatedly flipping the function-structure pair; the process must be repeated for every level in the computational hierarchy. Although the structure-function relation may ensure correctness between any two LoA, Primiero argues that the view fails to establish the desired transitivity of correctness throughout the entire computational hierarchy (i.e., between more than just two LoA). The result is an impoverished characterization of miscomputation.

For this reason, Primiero advanced a notion of implementation that considers multiple LoA of the computational hierarchy (intention, algorithm, high-level programming language, machine-code operation, execution), where an epistemological construct and ontological domain constitute each LoA,

> **Implementation as the relation of LoA:** An implementation *I* is a relation of instantiation between pairs composed by an epistemological construct *E* and an ontological domain *O* of a computational artefact**.**

The idea of the *EO*-pairs here is congruent to the function-structure relation, as the epistemological levels provide "the structure to understand the behaviour of the ontology" (Primiero 2020, 194). However, this view of implementation enables a more fine-grained notion of correctness because it differentiates between different layers/EO-pairs of the computational hierarchy. Consequently, one may, e.g., define concepts such as *functional correctness* or *procedural correctness* (related to different *EO*-pairs) and a corresponding detailed taxonomy of miscomputation (cf. Fresco & Primiero (2013), Floridi et al. (2015), Primiero (2020, 211-12)).

2.4 Type-(B) Implementation

Let me switch gears and examine our second implementation-framework. The central concern of this discourse is the so-called *Problem of Implementation*. In virtually all cases, physical computation is characterized in terms of the mathematical theory of computation (cf. sect. §2) and a "mathematics first"

attitude (Curtis-Trudel 2022), according to which some computational formalism of computability theory is the starting point for the definition of physical computation. In due course, one must explain how to bridge the gap between computational formalisms *Mc* and a physical system S_C. Specifically, the main idea is that formal abstract computational state transitions $m_i \rightarrow m_{i+1}$ need to 'mirror' the physical state transitions $s_j \rightarrow s_{j+1}$ of the material system. Often, the situation is depicted in a diagram, as seen in Fig. 2, where the upper horizontal arrow denotes computational state transitions of *M*c (specified by δ), the lower horizontal arrow denotes physical state transitions of *S*c, and f denotes the mirroring (i.e., the 'implementation' function):[27]

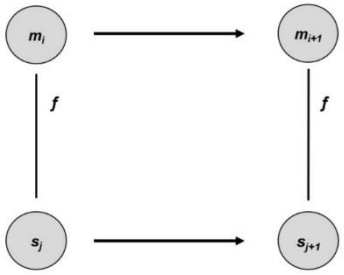

Fig. 2.2: A typical depiction of the core idea underlying physical computation

Subsequently, many scholars working on physical computation agreed that there are at least two main issues, albeit related, concerning implementation: Not only do they want to demarcate those systems that seemingly compute (e.g., laptops and brains) from those that don't (rocks), they would also like to determine which computation is executed rather than another. Closely following suit with (Sprevak 2018 and Ritchie & Piccinini 2018), the problem of implementation concerns:

> **COMP** The conditions under which a physical system is computing.
>
> **IDENT** The conditions that specify that a computational system implements one computation rather than another.

On this view, implementing a specific computation is constituted by two features. While COMP determines *that* a given physical system is computing, IDENT concerns *what* it computes. COMP and IDENT are intertwined in a way that makes it difficult to understand the latter without at least some preliminaries of the former.

[27] To the best of my knowledge, one of the first instances of this diagram in the philosophical literature can be found in Cummins (1989).

Virtually all potential answers attempt to solve the Problem of Implementation by couching the mirroring or implementation relation between *Mc* and S_C as a relation between mathematical structure and physical system in terms of a mapping *f*. This thought is reflected in the so-called *simple mapping account* (SMA)[28] and was, among others, articulated by Putnam (1988). The main idea is based on a simple mapping between abstract formalism M_C and a physical system S_C. Accordingly, the SMA postulates that a physical system S_C implements a computation *iff:*

Simple Mapping Account (SMA)

1. There is a mapping *f* from the states s_j of S_C to states m_i of M_C, such that
2. Under *f*, S_C's physical state transitions are morphic to M_C's formal state transitions (specified by δ), such that if S_C is in state s_1 where $f(s_1)= m_1$, then S_C evolves into state s_2 where $f(s_2)= m_2$.

The approach is elegant and straightforwardly captures what's pictured in Fig. 2; the SMA has basically become the starting point to solving the problem of implementation.

However, it is widely agreed that the SMA has two undesirable consequences. First, the SMA is charged with trivializing the notion of concrete computation. Given an open physical system S_C, one may carve out its physical states in whichever arbitrary way such that they are morphic to M_C. In other words, structure is too cheap to come by – any arbitrary computational description (like a *hello world* program written in C) with a sequence of computational states $m_1 \rightarrow m_2 \rightarrow \ldots \rightarrow m_i$ can be mapped on an arbitrary evolution of physical state transitions (of, say, a rock). So, according to the SMA, every macroscopic object realizes all kinds of computations, a position known as unlimited pancomputationalism.

Moreover, there is the issue of computational identity IDENT – the question of which of the multitude of computational profiles that simultaneously apply to a system is implemented.[29] The claim goes: Even if there were somehow a unique computational structure to begin with, structure alone would fail to deliver an account of basic computational individuation, as one needs extra ingredients to discern which computation is carried out. Such issues about computational indeterminacy especially pertain to 'bottom-up' research like cognitive science, the unconventional computing community, and reverse engineering. Roughly put, these disciplines investigate natural or artificial computing systems in empirical terms in which there is no a priori given M_C that specifies what's supposed to be computed. Even if we were to successfully identify which

[28] The term was introduced by Godfrey-Smith (2009) and made prominent by Piccinini (2015).
[29] To the best of my knowledge, the problem was first mentioned by Shagrir (2001).

physical states count as computational vehicles (e.g., neuronal spikes or flow of charge), we may have to choose between competing theories about what is computed. The problem is that computational vehicles do not wear a label on their sleeves. This gives rise to two sub problems *grouping indeterminacy* (how to group different states together) and *interpretative indeterminacy* (how to label states ones they are grouped). Frequently, the literature exemplifies the case (of interpretative indeterminacy) with a system S_C implementing logical duals, like a logic gate with the following behavior:

$\textbf{Input}_A$	$\textbf{Input}_B$	Output
5V	5V	5V
5V	0V	0V
0V	5V	0V
0V	0V	0V

Table 2.1: Logic gate

Under the assignment 0V → F, 5V → T, the truth table (Table 1) of the logic gate corresponds to an AND-gate. However, by flipping the labels (0V → T, 5V → F) the same device implements an OR-gate. Now, the issue is that the same system appears to simultaneously implement multiple computations (conjunction and disjunction) at once. The phenomenon generalizes to many other gates and computational systems. Fresco et al. (2021) recently called such physical systems *multiply specifiable* if they possess at least two logically non-equivalent labeling schemes when using the same labels (e.g., 'F' and 'T'). Consequently, the question arises, which of the two labeling schemes is the preferred one?

In response to the triviality arguments and computational indeterminacy most physical computation/type-(B) implementation accounts have amended the SMA by introducing additional features to address either one or both:

Extended Mapping Account (EMA)
SMA's first and second clause + additional conditions constraining *f*.

Since the initial formulation of the SMA, a plethora of potential candidate conditions have mushroomed, resulting in a fragmented physical computation discourse.

One class of approaches, seeks to tackle (unlimited) pancomputationalism. Often the common strategy has been to strengthen the conditional of SMA's second clause ('if $f(s_1)=m_1$, then $f(s_2)=m_2$') because it covers only one specific, instead of all possible execution traces. For that reason, some argued that putative computing systems need to have *counterfactual* state transitions (Copeland 1996). Roughly put, the idea is that if the system S_C had been in a physical state that

maps onto m_i, it would have evolved into a state that maps onto m_{i+1}. Others formulated similar requirements in terms of a suited *causal* structure (Chalmers (1996), Scheutz (1999)) or *dispositional theories* (Klein 2008). Counterfactual, causal, and dispositional constraints ensure that the mapping *f* applies to *all* of M_C's potential execution traces and not just, as previously the SMA, a single particular one $\delta(m_1, i)$ (where *i* is some input). As such, these types of constraints are a typical feature of type-(B) implementation.

A second class of attempts addressed the problems associated with IDENT (cf. Lee (2020) for an overview). Prominently featured here are to so-called *semantic* theories of computation. In a nutshell, they state that content is essential to computational states. Historically, this framework arguably developed separately from the SMA and other EMAs.[30] The two primary reasons why semantic accounts are widely accepted are as follows. First, the semantic account is catered to the Computational Theory of Mind and various brain sciences, which suggest that cognition (partially) relies on our brains performing computations. As brain states are believed to have content and process information, computational states must do the same. Consequently, according to the semantic view, computational states must possess 'aboutness' and carry external content or meaning. Second, the computational states of many computing devices manipulate meaningful symbols, and the semantic view can provide a solid foundation for understanding how these devices operate. In more recent form, proponents of the semantic view like Shagrir (2001, 2022) and Sprevak (2010), maintain that such semantic elements determine a privileged labeling scheme and hence do away with computational indeterminacy.

Another prominent framework comprises *mechanistic* accounts of computation (e.g., Milkowski (2013), Fresco (2014), Piccinini (2007, 2015)), according to which computation must be implemented in specific computational mechanisms.[31] One merit of mechanistic accounts is their capability to draw from the rich conceptual resources of the neo-mechanistic literature, especially on mechanistic *explanation*. What's more, as some formulations of the mechanistic account adhere to teleological functions, they also incorporate *functional* features, rendering computing systems as functional mechanisms that may fail to operate correctly (i.e., they may miscompute). Since one can formulate the mechanistic

[30] Fodor's statement can summarize the core idea of semantic accounts, "There is no computation without representation." (Fodor 1975; Pylyshyn 1984) is testimony to this development.

[31] N.b., in so far as mechanisms have a causal structure or are said to have counterfactual state transitions, the mechanistic account can also be interpreted as a yet more refined version of the previous EMAs, with the extra condition that putative computational states need to correspond to material components.

account without normative considerations, this feature seems to be logically independent of the mechanistic framework and could, in principle, apply to other EMAs. In the same vein, although I am not aware of such developments, one could combine mechanistic accounts with semantic elements or vice versa.

2.5 Juxtaposing (A) and (B)

What exactly is the relationship between type-(A) and (B) implementation? To recap, while both implementation types provide bridges that connect an abstract level to a less abstract one, they differ in initial purpose and scope: On the one hand, type-(A) implementation allows us to evaluate the correctness of computational artifacts' levels by normative requirements (i.e., the specification) of the stakeholders (programmers, users) involved. In this context, normative judgments pivot on teleological function ascription and various LoA. Type-(B) implementation, on the other hand, addresses one implementational stage only – the abstract-concrete dichotomy (the lowest LoA). Its purpose is to characterize physical computation in both natural and artificial systems formally. The theoretical framework underpinning virtually all characterizations of concrete computation is the idea that there is a mapping bridging the gap between abstract computational formalism (e.g., symbolic machine code) and the dynamic evolution of the physical states of the putative physical computing system.

Given their scope, both implementation theories are not mutually exclusive, for there is a juncture in (i) artificial computing systems at the (ii) abstract-physical interface (see Fig. 3). From the perspective of type-(A) implementation, the insights of type-(B) implementation are relevant

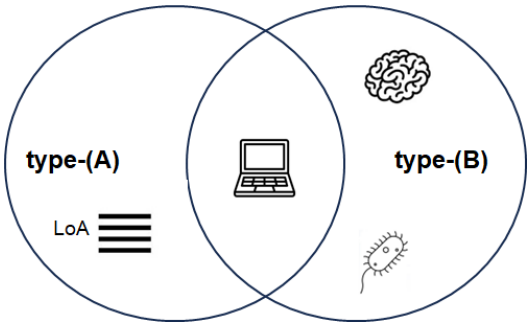

Fig. 2.3: A schematic Venn diagram of the intersection of type-(A) and (B) implementation. Their domains of application overlap in (i) artificial computing devices at (ii) the abstract-physical interface.

to address the implementation of a computational hierarchy at the abstract-physical interface. Vice versa, from the type-(B)'s point of view, the knowledge

contained in the type-(A) discourse offers a nuanced picture about artificial systems and computer scientists' related concerns and practices.

However, despite the overlap, there has been only a limited exchange between these two research domains. This separated development probably concerns their origin from different research traditions. Remember, philosophers developed type-B accounts as part of a broader project to articulate a version of the CTM. While it was always assumed that the computations carried out by the brain (at least, according to the classical, digital CTM) are the same as those carried out by computers, the focus remained primarily on natural systems, particularly the brain.[32] As a result, a tendency remains to take designed systems for granted and bracket higher-level programming practice as addressed by type-(A).

So, to further advance our general understanding of implementation and make it a cooperative endeavor, a more fine-tuned analysis of the relationship between type (A) and (B) is desirable. For so doing, I juxtapose the two different implementation types regarding their most salient features: (i) teleological function-ascription and (ii) mappings between levels. The following subsections elucidate these salient features across both approaches in more detail and pave the way for presenting a unified approach in sect. §2.6.

2.5.1 Teleology

Let me begin the juxtaposition with the role of teleology in implementation. Many theories of function emerged in the context of biological traits but have subsequently inspired accounts of artifact functions. Unlike organisms that may develop functions in evolutionary processes, artifacts are purposefully created and must account for human intentions. Overall, the question is how to balance intentional, evolutionary, and causal (non-intentional) parameters.[33] To date, there is no consensus in the literature about teleological functions and whether we should distinguish or opt for a unifying approach regarding natural and artificial systems.

Many of these concerns affect our understanding of implementation. For instance, as we have seen type-(A) accounts contain ingredients that allow us to check whether a program (or any given sequence of computations) is implemented and executed correctly at every LoA. The strategy was to address

[32] Some strands of the mechanistic account of computation are an exception since they also shed light on the subfield of computer architecture (Piccinini 2015, Ch. 8-11). Yet, this view still shows little engagement with other relevant practices in computer science.

[33] For a short survey of the debate, see, e.g., (Preston 2018, §2.3); for arguably the most detailed account, consult Houke's and Vermaas' (2010) ICE-theory (incorporating elements from intentional, causal-role, and evolutionist function theories).

correctness via an intentionally chosen function-structure pair. To give a concrete example, I borrow the following case from Turner (2020,19-21): Suppose we want to prove the correctness of a program *P* written in the WHILE programming language to find the greatest common divisor (GCD) of two integers. Then, we may only determine *P*'s correctness if we have some specification as a 'normative yardstick' (the teleological function) telling us what the given program should do. Subsequently, we may use the following expression

$$\forall x : \text{Num}.\forall y : \text{Num} \cdot \forall z : \text{Num} \cdot P(x, y, z) \rightarrow \text{Gcd}(x, y, z)$$

as a correctness condition for the successful implementation of *P*. With this formal expression at hand, we can establish correctness by a formal proof showing that *P*'s input/output behavior agrees with the logical specification. Albeit being a simple example, it illustrates a pattern of reasoning that underlies most considerations about correctness of computational artifacts.

Contrast this intentional approach with the type-(B) camp, where remarkably few studies have been designed to make *any* normative judgements of physical computation to begin with. This is a severe shortcoming because it ignores some of the central concerns of computer science: Absent a normative framework, we cannot address miscomputation, the verificationist-debate or account for the intricate correctness criteria of computer programs and software systems.

One of the few exceptions that allows for miscomputation is Piccinini's (2015; 2020) *functional* version of the mechanistic account (cf. also Mollo 2018 and Tucker 2018). In keeping with the functional-mechanistic account, physical computation equals the transformation of some (medium-independent) computational vehicle in accordance with a *rule*.[34] Similar to (formal) specifications, rules determine what should be computed. Since they can be violated, miscomputation occurs if the computational mechanism malfunctions (i.e., violates the rule). In principle, this is a welcome feature, for it stabs in the right direction for accommodating the concerns about correctness in computing at the abstract-physical interface.

The crux is how these rules come about. According to Piccinini, the rule that a computational mechanism should follow is determined by the so-called *goal-contribution account* of teleological functions. On that account, "[a] teleological function (generalized) is a stable contribution to a goal (either objective or subjective) or organisms by either a trait or an artifact of the organism." (Piccinini 2015, 116). Now, despite saying that subjective goals and artifacts are considered,

[34] In line with Egan (2019), I take it that 'rule-talk' is just a different (normatively connotated) way of referring to acting in accordance with a model of computation. See also my elucidations on the mechanistic account in Appendix B.

most recent work on the functional-mechanistic account merely pays lip service to them.[35]

Given its intellectual heritage, it is perhaps not surprising that the focus of the mechanistic account almost exclusively lies on natural systems and how they 'objectively' can be the bearer of teleological functions.[36] The problem with focusing on physical computation in terms of natural teleology is that it overlooks the widely employed notion of correctness in computer science. Alas, how human agents bestow artifactual functions to a material computing system according to their desires, beliefs, and intentions to determine computational correctness is virtually left unspecified. The upshot of this analysis is that there is an imbalance regarding the assumptions underpinning the usage of teleological functions in type-(A) and (B) implementation. Can we sidestep this to glue the two different notions together?

2.5.2 The Mapping between levels

The second point of comparison concerns the different implementation *relations* at the abstract physical interface. As we have seen, type-(A)'s approach is that a higher LoA's symbol structure must correctly translate into a symbol structure corresponding to a lower LoA. The *agreed-upon* semantics determine what then counts as viable implementation between abstract structures, else it remains unclear how the different structures are supposed to correspond to one another. While much more can be said about the precise characterization of the semantics, what suffices for the present juxtaposition is that the provenience of the mapping *f* essentially hinges on the stipulations and conventions regarding the semantics made by the designers and programmers. Now, when it comes to the 'bottom' of the computational hierarchy, type-(A)'s previous assumption to construe implementation as a relation that links abstract objects to other abstract objects or structures no longer holds. The reason is that the abstract-physical interface requires a fundamentally different underlying equivalence relation – a cross-categorical relation between abstract and physical objects.

This is where type-(B) enters the picture and could enrich our general understanding of computational implementation in computer science. So, let us look closer at how the implementation relation is conceived in this framework by revisiting the SMA. According to its first clause, a mapping function *f* takes

[35] Schweizer (2019) and Anderson (2019) are notable exceptions.

[36] Typically, proponents of the teleo-mechanistic view of physical computation flash out natural teleology by claiming that living organisms share a set of capacities (survival, development, reproduction, and helping). These capacities are thought to give rise to a functional organization, allowing them to pursue these capacities (Piccinini 2020, 68); see also Dewhurst (2018) and Mollo (2018).

physical states and maps them onto computational states $f: S_C \rightarrow M_C$. Two qualifications are noteworthy about this relation: First, the mapping goes from physical states to computational ones; if it were the other way around, we would engage in computational modeling. Computability theory studies models of computation that are distinct from computational models used in scientific practice. The former models are used to explore computation in its proper sense, while the latter models are used to simulate natural phenomena using computational techniques (Milkowski 2014). Secondly, the mapping we are concerned with is supposed to connect two different ontological domains.

In spite of the fact that the EMAs of type-(B) literature have brought forward an impressive amount of literature with various constraints on the implementation relation *f*, the metaphysical nature of the mapping relation does typically not take center stage in the discourse of physical computation. Exemplary is a statement by Chalmers, stating that

> "[t]he definition of implementation does not appeal to any specific mapping relation: rather, it quantifies over mapping relations, which can be any function from physical states to formal states. I also do not know what it is for a relation to have metaphysical commitments." (Chalmers 2012, 231)

Similarly, Sprevak stresses that it is a "strategic error" to focus on the metaphysical nature of *f* (Sprevak 2018, 176).

However, remaining silent about the metaphysical nature of the mapping may, at least for the current project, come with the cost of an impoverished or partially incomplete picture of how type-(A) and (B) implementation relate.

To inspect the metaphysical nature of computational implementation more closely, it is helpful to turn to similar cases. When realizing that we must link physical objects to logico-mathematical ones, one notices that what we are dealing with is a special instance of a much more general issue: the relation between mathematical objects (of computability theory) and the physical world – a relation raising notorious questions in the philosophy of science and applied mathematics (Wigner 1960; Steiner 1998). Following Contessa, I refer to the general issue as the bridging problem, "the problem of how to bridge the gap between [abstract] models and the world" (Contessa 2010, 516). Put differently, we need to explain the correspondence between two ontologically different categories – viz., how mathematical objects relate to the physical.

Trying to solve the issue quickly leads to quite a technical territory that would hamper the current discussion. For the sake of clarity, I opted to discuss the general idea here and shift an in-depth discussion into Appendix B instead. Very roughly put, the important point for now is that virtually all contemporary solutions to the bridging problem appeal to a mapping between mathematical and physical structures. However, one of the main issues with this 'mapping

view'[37] is that ordinary functions only obtain between the domains of set-theoretic structures – yet physical objects are not set-theoretic structures. Thus, without further qualifications, maintaining that morphisms obtain between (abstract) mathematical structures and physical objects amounts to a category mistake (e.g., Frigg 2006, 55; van Fraassen 2008, 237f; Vos 2022). To put a long story short, a solution to the bridging problem requires an account of how material systems can offer set-theoretical structures.

However, it is wide consensus that this task is all but straightforward. Bueno and Colyvan, for instance, remind us

> "Put simply, the world does not come equipped with a set of objects (or nodes or positions) and sets of relations on those. These are either constructs of our theories of the world or identified by our theories of the world. Even if there is some privileged way of carving up the world into objects and relations […], such a carving, it would seem, is delivered by our theories, not by the world itself. What we require for the mapping account to get started is something like a pre-theoretic structure of the world (or at least a pre-modeling structure of the world)." (Bueno & Colyvan 2011, 347).

Can we overcome the structure generation problem concerning the levels employed in type-(A) and (B) implementation?

2.6 A Unified Theory of Agential Implementation

After surveying the implementation landscape and providing an in-depth analysis, two problematic instances still need to be addressed in aligning type-(A) and type-(B) implementation. On the one hand, there needs to be more consistency between the normative features determining correctness. On the other hand, the bridging problem called the philosophical plausibility of a naturalized implementation relation into question. In order to advance, we need an explanation of how we can account for a physical system's mathematical/computational structure.

The remainder of the chapter proposes a remedy to this situation in the form of a unified theory of implementation in two steps: First, I submit that (A) and (B) can be unified by bringing them into conversation with the conceptual machinery of the literature on material models and scientific representation. Second, based on these insights, I sketch a use-based account of implementation. Since the common denominator among these findings is the stipulations and conventions of (human) agents, the novel framework traces different agential

[37] Sometimes, when thinking of models and representations in terms of such a formal relation between structures (e.g., a mapping), the issue is also referred to as the 'mapping view' (Pincock 2004, Batterman 2010, Bueno & Colyvan 2011).

involvements in terms of dependency relations. Accordingly, the result is a *Unified Theory of Agential Implementation* (UTAI).

2.6.1 Material Models as a remedy

According to what Giere (1999) calls the *representational conception*, scientific models are used by scientists for the purpose of representing some (real-world) system, where the latter is commonly referred to as target system T. Scientists use models and their representational capacities for drawing various sort of conclusions about the target system (e.g., explanation, prediction, confirmation); a practice known as *surrogate reasoning* (Swoyer 1991). Accordingly, scientific representation is characterized as the relation f between a model M and its dedicated target system T, such that $f: M \rightarrow T$.

The reason why I propose the representational conception as a remedy is twofold: First, representations can be faulty and misrepresent the target. While scientists may allow for minor deviations up to some previously defined threshold, larger differences count as misrepresentation (the model does not do what it should do). This aspect will be crucial to account for the analogous case of miscomputation and correctness considerations discussed in §2.5.1. Second, both models and targets come in various 'ontological flavors.' On the one hand, one typically distinguishes between (i) material and (ii) theoretical models.[38] On the other hand, target systems are either (a) real-world systems or (b) hypothetical scenarios.[39] Consequently, various modeling scenarios result from the possible combinations of (i)-(ii) and (a)-(b). As such, the modeling relation – like the problem of implementation – may also be a special instance of the bridging problem.[40] More precisely, the crucial commonality between scientific representation and implementation of computation is that both essentially require a mapping that relates mathematical structures to a physical substrate.

Given these similarities, one may solve our previously identified issues regarding teleological function ascription and the bridging problem by bringing one domain (modeling) into conversation with another (computing). Put differently, the idea is that what counts for scientific models, *mutatis mutandis*,

[38] The term model denotes a heterogenous collection of things – models come in the form of descriptions, as material objects, or as abstract (mathematical) objects; for an extended list of 120 types of models, see (Frigg 2022, Ch. 16).

[39] Weisberg, for instance, investigates the case of hypothetical modeling (2013, 121-134), where models may represent nonexistent targets, possibilities, or impossible targets (e.g., models where the targets are perpetual motion machines or multiple sexes populations).

[40] For instance, assuming that M is a theoretical model relying on mathematical structure as a representational vehicle, one needs to specify how the parts of the structure are mapped to the physical make-up of the target system. Put differently, the representational relation f needs to bridge the abstract-concrete dichotomy.

applies to computing devices. In fact, in some cases, the distinction between what counts as a scientific model and what counts as a computer is diminishingly small. Take, for instance, Frigg & Nguyen's (2018) example of the Philips-Newlyn machine. The machine uses the flow of water through a specifically designed pipe system to model the distribution of commodities in a national Keynesian economy. However, also known as 'MONIAC' (Monetary National Income Analogue Computer) the device can equally well be regarded as a special-purpose liquid-based analog computer. Instead of representing a selected economic scenario, the MONIAC can, in principle, be used to compute (a small set of) differential equations.[41]

Of course, these similarities are not restricted to the flow of water. Overall, many different physical properties can be used as representational- or computational vehicles, respectively. For instance, when surveying the material variety of such vehicles, Sterret notes that

> "[…] electronic circuits were used as analogues of anything that could be formalized as a solution of certain classes of differential equations, and ever more sophisticated machines were developed to deal with ever larger classes of differential equations and problems. Other examples of analogues used for computation are mechanical analogues such as the geared devices built in the seventeenth century, the soap bubble analogue computers invoking minimization principles that were used to efficiently solve difficult mathematical problems in the twentieth century and biological analogue computers of the twenty-first century such as amoeba-based computing (ABC) analogue models." (Sterret 2017, 858)

Qua models, various physical systems may be employed as representational vehicles for an explanation or prediction of a target system. *Qua* computer, one may use physical systems as surrogates to read off the results of a sequence of computation specified under a model of computation M_C. The difference is that instead of a real-world target system, one then simply reasons about a particular 'hypothetical scenario,' where the latter is characterizable by a transition function δ that's compatible with a model of computation M_C.

But how does the modeling literature address the 'structure generation problem' we encountered earlier? Generally speaking, philosophers of science like Suárez (2003) and van Fraassen (2008) criticize naturalized attempts of scientific representation, namely that the mapping *f* between model and target reduces to a factual, mind-independent relation since it flies into the face of the bridging problem. Without answering the structure generation problem, it is highly contentious to defend a somehow naturally occurring mapping relation.

[41] I will discuss this device and how it features in physical computation in much more detail in Chapter 4.

Given that naturalized accounts about mappings fail to elucidate how and when such a relation comes about without explaining the assumption of some privileged preconceived structure, they are moot. Faced with the inadequacy of naturalized accounts of representation, philosophers of science nowadays commonly agree that scientific representation is contingent on human *agents establishing* a mapping to the intended target. The upshot is that the relation between model *M* and target *T* does not simply obtain 'naturally,' i.e., without the decisions, conventions, and stipulations of some scientists. Often, this type of correspondence is called a *three-place relation* because it entails (i) a model, (ii) a target, and (iii) an epistemic agent. In consequence, van Fraassen, for instance, emphasized the necessary involvement of human agents in formulating his *Hauptsatz* of scientific representation, as "[*t*]*here is no representation except in the sense that some things are used, made, or taken to represent some things as thus or so.*" (van Fraassen 2008, 23).[42]

In recent years, several scholars have (independently) turned considerations about scientific models and representation like these into an approach to physical computation (e.g., Care 2010; Horseman et al. 2014; Fletcher 2018; Papayannopoulos 2020) – something which will be scrutinized much more fully in Chapter 4. For now, it suffices to say that this accumulation of research suggests that material models and computers are (fine-tuned) physical objects employed by human agents as *epistemic tools* for their specific context-dependent purposes. In what follows, I will present the underlying assumptions and features of this way of seeing things.

2.6.2 UTAI and its features

At last, let me introduce the theory that enables the unification of type-(A) and (B) implementation: UTAI. Figure 2.4 provides a schematic depiction of UTAI and its most important features – the implementation of a model of computation M_C at the abstract-physical interface through the execution of (one of the traces) its corresponding transition function δ. In what follows, the various elements of the graphic are discussed in detail.

First, the abstract-physical interface is illustrated by the dotted line horizontally running through the diagram. In comparison to Fig. 1 (b), higher LoA and the full computational hierarchy are implied to be represented in the 'abstract realm' (upper half above the dotted line), where one deals with symbolic implementation (type-(A) implementation). In line with the SMA, the equivalence between the state transitions of M_C (specified by a transition function

[42] Many of these conclusions drawn in the philosophy of science coincide with the technical literature from the philosophy of applied mathematics (cf. Appendix B).

δ) and the evolution of physical states of a putative computing system S_C is pictured through a diagram as, e.g., found in Cummins (1989) or Ladyman (2009) (cf. Fig. 2.2). The labels (1) – (3) correspond to the assumptions of scientific representation/modeling-based theories of computation:[43]

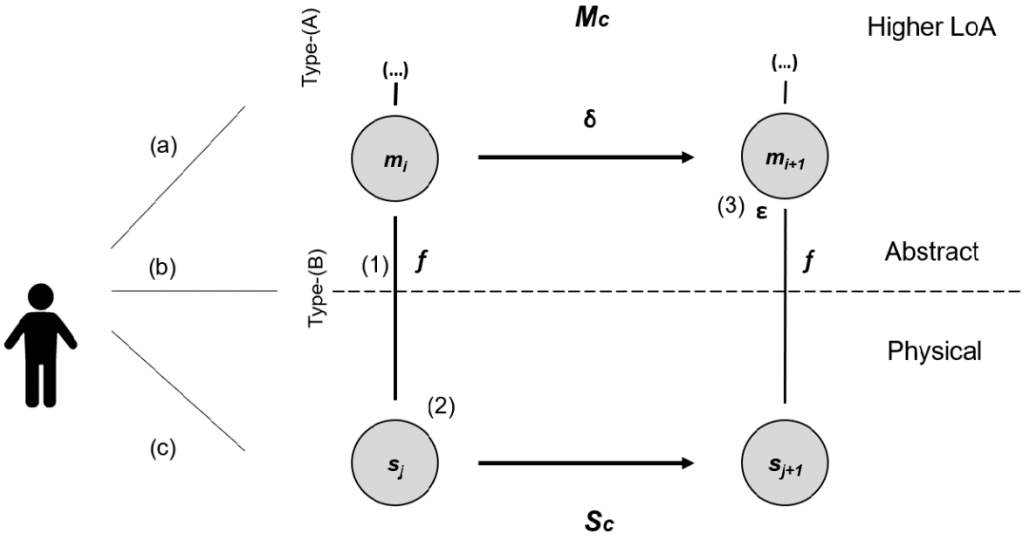

Fig. 2.4: Schematic depiction of a unified theory of agential implementation (UTAI).

Assumption (1) Implementation is based on a *representation relation f* between the designated physical states of the putative computing system and the abstract states transitions defined by δ of a computational formalism M_C (cf. Ladyman 2009, Horseman et al. 2014, Fletcher 2018, Papayannopoulos 2020), such that $f: S_C \rightarrow M_C$ holds. Analogous to the case of material scientific models, where certain features of the model act as a representational vehicle to represent (features of) a target system, implementation comes about when certain (selected) features of a putative computing system act as a computational vehicle to compute a function. Note that this development is a departure from the EMA, in as much as scientific representation/ implementation can no longer be deemed a binary relation, but a ternary one, because it is only due to the use/stipulations of some agent that the linkage between model and target arises in the first place.

Assumption (2) The starting state m_i (i.e., the input state) of M_C must be related to some initial starting state of the material system s_j. Likewise, the output state m_{i+1} requires decoding it from s_{j+1} by reading it off or performing some type of measurement. The rightwards-pointing black arrows in the diagram, labeled by δ and S_C respectively, stand for the transition of the computational states m_i and

[43] In as much as features (1)-(3) can already be found in the previous literature, UTAI is indebted to their respective insights.

m_{i+1} and the temporal evolution of the computational vehicle/label bearers s_j (both represented by the grey circles).

Whilst maintaining that the act of encoding input-states and decoding output-states after a computation has been argued for before (e.g., Churchland & Sejnowski (1992, 63)), the here advocated view is different because the acts of encoding and decoding are formulated in terms of scientific representation. Importantly then, encoding and decoding are – in line with the notion of scientific representation – directional (no longer isomorphisms) and require to be carried out by some agent or epistemic community.

Assumption (3) Following the suggestion by Horsman and collaborators (Horsman et al. 2014), I argue that the notion of scientific representation enables us to address *inaccuracy* as a form of miscomputation.[44] Simply put, the idea is that one may introduce an acceptable margin of error ε between the output states of M_C and output states of the physical system. If the chosen computing system is completely reliable (i.e., run without errors), each computation cycle yields perfect equivalence between the state transitions of M_C and the physical evolution of the system; there is no inaccuracy. However, complete reliability is a pipe dream. Real-world scenarios typically do not behave in a perfectly preconceived idealized manner. As a result, the abstract series of computation and the physical outcome only coincide up to ε.[45] Depending on the chosen value of error interval ε, the same process could count as correct computation in one case and as miscomputation in another.

While I think that (1)-(3) of the previous accounts are primarily correct, UTAI entails further features and more fine-grained factors. Specifically, UTAI needs to elucidate the involvement of and dependency on human agents concerning the imputation of mappings, the creation of structure, and the ascription of teleological functions. To uncover the intricate interrelations between human agents and the different, UTAI explicitly tracks these dependency relations,[46] denoted by (a)-(c), between human agents (represented by the black mannequin) and various elements in the implementation process. In the remainder of the chapter, I explain the implications of these dependency relations in detail:

---

[44] Philosophers of science commonly agree that there is at least a second type of misrepresentation, viz., mistargeting. Transposing this view to computation amounts to the case where users (accidentally) implement the wrong computational formalism. I will leave this discussion out for now.

[45] Once again, one can appropriate the discussion of scientific modeling, where philosophers of science criticized the idea that scientific representation reduces to isomorphism because it cannot make room for distortions.

[46] 'Dependence relation', here, is understood as a relation between different entities, where one entity is dependent on another.

Dependency relation (a) emphasizes the crucial involvement of human agents in higher LoA. It is paramount for at least two reasons: First, following the desires, intentions, and pragmatic concerns of the programmers, a 'computational problem' is formulated. In actual programming practice, this typically leads to a (formal) specification, determining what is supposed to be achieved. The specification acts as the normative yardstick to check correctness. As discussed in the type-(A) implementation literature, the specification may be seen as an ascribed teleological function. It is necessary to provide judgments about correct execution and faulty behavior (miscomputation).

Furthermore, dependency relation (a) allows for illuminating a second crucial involvement of human agents. After agreeing on a specification, practitioners may then devise an algorithm. Next, the algorithm is typically formulated in a suitable computational formalism M_C (e.g., a programming language of your choice). The process described here roughly corresponds to the construction of the various LoA in the computational hierarchy (Fig. 1). The bottom line is that specifications, the algorithms targeted at the specific problem, the ensuing source code, and so on are *all* dependent on human ingenuity. Put differently, computer programs appear not to be discovered; they seem to be created by human agents for diverse practices such as scientific endeavors, business, entertainment, and many more. Ignoring the agential dependence of computational artifacts bears the danger of unreasonably rendering the implementation of computational artifacts in naturalistic terms.

Dependency relation (b) concerns the mapping f that bridges the abstract-physical interface. Implementation may occur when agents come up with a structure-generating description (e.g., through information hiding) and a suitable mapping relating the abstract and concrete realms. As argued at length in sections §2.5.1 and §2.5.2 (see also Appendix B), naturalized approaches are ill-suited to address the bridging problem adequately. Instead, we necessitate the interpretational capacities of agents to overcome the structure generation problem. Structure generation is contingent on agents because it demands that specific properties/capacities of the system are *selected* and *interpreted* as computational states or vehicles s_j.

Furthermore, to bridge the gap between an abstract model of computation M_C and a concrete computing system S_C, a mapping relation between the two is eventually needed. Such a relation calls for the stipulations of human agents. Users impute their chosen computational formalism onto the putative physical computing system. The material system can then be used as an epistemic tool, i.e., as a surrogate to carry out the intended series of computations determined by the previously created program.

Lastly, following these considerations, the implementation relation $f: S_C \rightarrow M_C$ bridging the abstract-physical interface (represented by the horizontal dotted line) is no longer conceivable as a mere binary relation. Instead, f is a ternary relation – because it necessarily depends on the stipulations of agents – characterized by a *representational quadruple* $\langle m_i, f, s_j, \text{ Agent}\rangle$.

Dependency relation (c) characterizes the physical interactions of the human agent(s) with the putative computing system as epistemic tool. Ideally, a computing system is not only sufficiently reliable for repeated executions but also reconfigurable. Physical reconfiguration is needed to reprogram the computing system ultimately. The mere imputation of a different model of computation M_C onto the very same unchanged structure is insufficient for implementation. The underlying physical setup from which the structure was generated must also change; otherwise, it will result in a mismatch. So, what we require from a programmable system is that a different starting state would have led to a different but corresponding output state. Put differently, to be re-programmed, thus calls for a counterfactual explanation.[47]

The notions of modeling and scientific representation that underpin UTAI allow incorporating the crucial constraint that *counterfactual* claims about the computing system hold. In the context of scientific modeling, Bokulich (2011, 39), for instance reminds us that for a model *M* to effectively explain a given target phenomenon *T*, it is essential that its counterfactual structure closely aligns with that of *T*. In other words, the elements of the model should, in a loose sense, 'reproduce' the relevant features of the phenomenon being explained. Like in the case of the EMA, the counter-factual condition rules out stipulative fiat, i.e., the completely unconstrained usage of arbitrary systems for computation that would collapse into interpretational pancomputationalism. In other words, while interpretation is a necessary condition, it is insufficient because the computational vehicles of the computing system need to behave suitably.

2.7 Conclusion

This chapter surveyed different notions of computational implementation whose connection has been underexplored so far. While type-(A) implementation concerns correctness criteria of (abstract) computational artifacts, type-(B) implementation addresses physical computation. The main takeaways of this chapter are threefold:

[47] N.b., this insight is virtually similar to the ones of the counterfactual/causal/dispositional EMAs).

(1) First, I clarified the implementation landscape: Despite having emerged from different research traditions, I showed that the different implementation types discussed in this chapter are not mutually exclusive. Instead, they partially overlap under two conditions: considering (i) designed systems at (ii) the abstract-physical interface.

(2) Second, based on a subsequent in-depth comparison, I argued that type-(A) and (B) may mutually enrich each other regarding teleological function ascription and bridging the abstract physical interface. This cross-fertilization is vital to explain the implementation of computational artifacts like computer programs in real-world machines. However, without further qualification, there remain more considerable conceptual obstacles familiar from the philosophical literature of teleological functions and applied mathematics.

(3) Third, I provided a specific way of thinking about overcoming these obstacles and thereby coherently aligning the two different implementation theories. Particularly, by bringing the implementation literature in conversation with the literature of scientific modeling and representation, I sketched a unifying framework called UTAI (unified theory of agential implementation). The reason why this is fruitful is that in both modeling and computing, agents engage in the form of object-based reasoning, where artifactual functions are externally attributed, and agents impute a mapping relation between concrete system and abstract target/program. My analysis showed that accounts, like UTAI, sketched in agential terms, offer the right resources to accommodate the main underlying assumptions of both type-(A) and (B) implementation: stipulated mappings, generated structures, and ascribed teleological functions. In sum, UTAI has the explanatory virtue of facilitating cross-talk between so far rather separated discourses and kinds of literature.

In the following chapters, I will continue this analysis by focusing on UTAI's three dependency relations between epistemic agents and ontologically different aspects of computer programs.

3 The Problem of Creation meets Computer Programs

The previous chapter laid the groundwork for a systematic study of the unified theory of agential implementation (UTAI). This framework, which identifies three distinct dependency relations in the implementation of computational formalisms like computer programs, will be examined in detail in the following three chapters. The idea is that tracking these relations will help us to make sense of the connection between the different entities gathered under the polysemic term 'program.' In this chapter, we will start by focusing on the first relation, which is the connection between epistemic agents and programs as abstract entities.

3.1. Introduction

According to a prominent view, computer programs do not appear to be concrete objects, yet we commonly think of them as created entities that can be interacted with. This is somewhat surprising since philosophical orthodoxy holds that abstract objects are not integrated into the causal pathway and can, therefore, not be created.[48] Ergo, a pressing philosophical problem is creeping up:

> 'In which way can these program qua abstract objects be the products of our creation?'

Unfortunately, this issue has not fully received the attention it deserves until now. In light of this problem, my goal in this chapter is to state more precisely in what sense we can classify computer programs as *abstract objects*. My starting point to unscramble the situation is to pick up the theme from the Prologue and rethink how lawmakers of the 1970s struggled to come to terms with classifying programs under patent law. Specifically, one episode sticks out: While some argued that they should be conceived as physical switch settings or parts of real-world machines, others suggested classifying them similarly to novels or musical scores. Notably, reflection on the ontological status of (art)works has led to the idea of thinking of them as abstract objects (Thomasson 2006; Sanfilippo 2021). Throughout this chapter, I assimilate this reasoning, argue why it is a plausible stance toward programs, and explore its ramifications.

[48] The standard metaphysical view states that abstract objects exist eternally and cannot be created. Most philosophers understand creation as a causal relationship between the creator and their creation. However, abstract objects are causally inert and hence cannot enter such a relationship, so it is unclear what kind of process the creation of an abstract object involves (Mag Uidhir 2013, 11).

In a nutshell, the underlying rationale is this: Many scholars from the philosophy of art, metaphysics, librarianship, and literary studies consider their corresponding (art)works abstract because they cannot be identified with a concrete instance; they appear to (simultaneously) exist in different media without being reducible to any specific token. Importantly, most participants of the debate understand 'abstract' here in the same sense as the abstract-concrete dichotomy (Falguera et al. 2022) that is relevant to us, i.e., as non-spatio temporal entities, as appealed to in the Introduction and the previous chapter. On the face of it, these so-called *repeatable artworks* thus face the same dilemma as programs: The crux is that the premise about the existence of art-abstracta stands at odds with the plausible assumption that these artworks, qua artifacts, are intentionally being *created* by a specific human being at a specific time and place. Typically, the conundrum is called *the Problem of Creation* (PoC). Albeit underappreciated in the philosophy of computing so far, I will show that many of the philosophical problems and solutions associated with this way of thinking about repeatable artworks applies *mutatis mutandis* computer programs.

The main takeaway from my application of the (POC) to programs is twofold. On the one hand, from the perspective of the philosophy of computer science, my approach enables us to step outside the beaten paths of the metaphysical inquiry in computing and offer a new angle on the ontology of programs. On the other hand, from the perspective of contemporary metaphysics, my approach steers the debate about the ontological status of computer programs towards more established philosophical territory. Notably, it shows that the abstract nature of programs does not require the postulation of complete *sui generis* solutions (e.g., a 'dual nature') but must be couched along the axis of Platonsim, Nominalism, and Creationism.

In what follows, this chapter is divided into five sections. Section §2 provides some necessary preliminaries to apply the (PoC) to computer programs. Next, section §3 introduces the (PoC) in more detail and outlines a taxonomy of its most typical responses. Thereafter, in section §4, I discuss the implications of the differing metaphysical frameworks when adopted to computing. Lastly, I conclude (sect. §5).

3.2 Setting the Stage

Before we dive into the essential philosophical details of the (PoC), I want to provide some initial clarifications about programs. If computer programs are to be the subject of the (PoC), we must have a clear grasp of (i) what one exactly means by 'program,' (ii) how they are created, and (iii) in what sense they are abstract. Accordingly, in Section §2.1, I will introduce an example program that

will be frequently referenced throughout this chapter. Following that, in Section §2.2, I will briefly outline how programs are created. Lastly, in Section §2.3, I will discuss the abstract nature of programs more formally.

3.2.1 An example as conceptual laboratory

Although relatively unproblematic in everyday life, it is a somewhat contested question among philosophers of computing what exactly might be meant by the concept of 'program' (Gemignani 1981; Lonati et al. 2022). With so many variations in the understanding of programs – regarding them as abstract entities, physical entities, or entities that span multiple ontological categories – using the term unexplicated is potentially misleading.

However, in line with my arguments about the polysemic nature of the term in this thesis' introduction (Ch. 1), I will not define programs here. Instead, I only rely on what I take to be a paradigm instance of a computer program written in a high-level programming language, as depicted in Fig. 3.1. The *source code* is written in *C* and, and the program uses a while-loop to find the greatest common divisor (GCD) of two integers. Instead of relying on a rigorous characterization of *all* computer programs, *this* example will act as my 'conceptual laboratory' in due course.

To avoid any ambiguity, it is vital to note what I do (and do not) intend to pick out with my example. As we have seen in the previous chapter, it is common to think that computational systems are built in a hierarchical fashion (e.g., Primiero 2016; cf. Fig. 2.1). Accordingly, there are various interconnected levels such as specification, algorithm, source code, machine code, hardware and so on. Facing this 'stratified ontology,' we must specify on which level our example is located. This task is relatively straightforward: Given that the code is written in C, we can classify it as a well-formed set of high-level programming language instructions, i.e., a symbolic (type-(A)) implementation of the proposed algorithm in source code.

Importantly, this characterization allows us to distinguish our C program from its neighboring upper and lower LoA. Regarding the former, the higher LoA, it is important to distinguish our program from the GCD algorithm it implements. Despite hinting at the difficulties of defining the term 'algorithm' in the Introduction (see also Appendix A), I use 'algorithm' here in a sense similar to Newell:

> "An algorithm is more abstract than a program. Given an algorithm, it is possible to code it up in any programming language. You might think that a program should be something like an algorithm plus implementation details. Thus, you examine the text of a purported algorithm-if you find an implementation detail, you know it is a mere program." (Newell 1986, 1029).

On this view, the corresponding (GCD) algorithm could have thus been implemented in an entirely different programming language, like Java or Python.

```c
#include<stdio.h>

void main()
{
    int a, b, gcd, remainder, numerator, denominator;

    printf("Enter two integers\n");
    scanf("%d %d", &a, &b);
    if (a > b)
    {
        numerator = a;
        denominator = b;
    }
    else
    {
        numerator = a;
        denominator = b;
    }
    remainder = numerator % denominator;
    while (remainder != 0)
    {
        numerator   = denominator;
        denominator = remainder;
        remainder   = numerator % denominator;
    }
    gcd = denominator;
    printf("GCD of %d and %d = %d\n", a, b, gcd);
}
```

Fig. 3.1: Example C program to find the GCD of two integers using a While-loop.[49]

Regarding the lower LoA, we have to remember that source code in a high-level language is rarely directly executed on a computer. It is typically translated into machine code using interpreters or compilers. In each case, different compilers, for instance, may further optimize the resulting machine code to the underlying hardware. As a result, the original GCD program may fragment into many different lower-level implementations in different computational ecosystems (e.g., on a Thinkpad Carbon x1, Gen 5, with Ubuntu version 22.04 LTS as opposed to on an Apple MacBook Air macOS Sonoma).[50]

[49] https://www.sanfoundry.com/c-program-gcd-two-numbers-recursive-euclid-algorithm/#c-gcd-program-method-4

[50] I thank Liesbeth De Mol for pressing me on the matter and suggesting these particular example implementations.

Moving forward, it is thus crucial to bear in mind that my investigation throughout this chapter is exclusively limited to the symbolic level in which the source code is written (this comes close to what I call the Notational View in Appendix A).[51]

3.2.2 A brief sketch on how Programs are created

Philosophers standardly define 'artifacts' as objects made or produced for a specific purpose (Hilpinen 2017). Typically, when creating a (material) artifact, the makers intentionally modify one or more objects until it becomes the desired end product (Hilpinen 1993, 165). General actions of artifact-making include separation, reshaping, and assembly using different techniques or routines (Hilpinen 2017). As programs are intentionally produced entities, we can consider them sub-types of artifacts (see Appendix A).[52] However, since the working assumption of this chapter is that it is plausible to assume that programs are abstract objects, it is essential to take a closer look at their production process and ask: How exactly are programs created?

Given the ubiquity and importance of programming in modern society, accounts documenting the production of programs run the gamut – including, but not limited to, historical accounts such as Grier's (2013), pleas for various programming styles like structured programming (Dijkstra 1968), hundreds of textbooks that teach students basic programming techniques (Abelson et al. 1996), project managing frameworks from the realm of software engineering that elucidate the do's and don'ts of (large) labor-intensive projects (Brooks 1978), and ethnographic studies (Button & Sherrok 1995).[53]

Commonly, the story about the creation of programs goes something like this: At the start, the aim is to create a specific program. However, in the realm of computer science, we cannot simply transfer our desires and intentions to our computers. We must first translate them into a computational formalism that the machine can execute. This typically involves writing source code in a high-level programming language. The programming process is multi-step, error-prone, and often laborious. To stay on course with our original goal, we ideally create a *specification*. Specifications serve as the 'blueprint' for producing small and complex large-scale computer programs. In other words, they define what the

[51] Although I have only provided a small programs, I think that, without loss of generality, many of this chapter's results apply mutatis mutandis to larger or even large-scale programs.
[52] As will become clear throughout the chapter, Platonists about programs (if there are any such persons) may argue that assuming that programs are produced is to throw out the baby with the bath water. Instead of creating programs, a Platonist would suggest that we somehow 'discover' them.
[53] This list represents merely a fraction of the abundant sources available. However, I do believe that it is enough to convey the general message of the current subsection.

program is supposed to do (Primiero 2020, 208). Therefore, understanding specifications is crucial for comprehending the process of creating programs. As Turner (2011, 135) points out, 'programming is not an aimless activity'.

At a first stab, we may turn to Cantwell Smith's characterization of program specification

> "A specification: a formal description in some standard formal language, specified in terms of the model, in which the desired behavior is described. [...] [A]ll it has to do is to specify what proper behavior would be, independent of how it is accomplished. For example, a specification for a milk-delivery system might simply be: "make one milk delivery at each store, driving the shortest possible distance in total". That's just a description of what has to happen. [...] Specifications, to use some of the jargon of the field, are essentially declarative; they are like indicative sentences or claims (Smith, 1985, 20)."

As rightly pointed out, specifications are often formally written; think of logical notations like Z, B, and VDM. However, Cantwell Smith's definition is too strict, as it limits the notion of specification to a formal description of our original problem. While a complete formalization of the specification is desirable, it is not always practically achievable.[54] In a more recent paper, Turner (2011) reminds us that specifications can take several non-formal forms such as natural language description or graphical representation. That is why Duran clarifies

> "Computational practice has shown us that specifications cannot be fully formalized. Rather, they must be conceived as 'semi-formal' descriptions of the behavior of a target system. By this I mean that formal as well as non-formal descriptions coexist in the specification. In other words, mathematical and logical formulae coexist with documentation, instructions, and other information written in plain English." (Duran 2018, 41)

Regarding my small exemplary program, one may imagine where its specification was informally given as *A program written in C to find the greatest common divisor of two integers using a While-loop*.[55] Depending on one's skill level, it should be a simple routine for a trained programmer to transform this rudimentary specification into something like our exemplary program.[56]

Although my example program is simple, its creation process exemplifies a pattern that underlies many successful creations of programs written in (high-level) programming languages, I take it as sufficient evidence to support this chapter's working hypothesis that programs are creatable.

[54] My simple example program in Fig. 3.1 is an exception.
[55] For a formal specification, see sect. §2.5.1 (Ch. 2).
[56] For a more detailed discussion on the functional specification of a program implementing the GCD, see (Turner 2018, 44-47).

3.2.3. In what sense Programs are Abstract: The Physical Object Hypothesis

Why and in what sense can programs be conceived as abstract objects? In the introduction to this chapter, I already informally discussed potential answers. I wrote that 'abstract' refers to the metaphysical category determined by the abstract-concrete distinction and (hence) signifies being none spatio-temporal. Specifically, I invoked an *argument from analogy* based on repeatable artworks. Researchers from various disciplines consider repeatable artworks abstract because they cannot be plausibly identified with an individual copy. I will now adopt this reasoning to computer programs.

To present these thoughts more precisely, it is helpful to frame them in a more formal framework. In the philosophy of art, the problem at stake is frequently discussed under the name of the *Physical Object Hypothesis* (POH). Adapted from Mag Uidhur (2013, 8, fn. 4), the reasoning of the (POH) can be summarized as follows,

Physical Object Hypothesis (POH):

(POH)$_1$ There are such things as artworks.

(POH)$_2$ Artworks are either repeatable or non-repeatable.

(POH)$_3$ Repeatable artworks cannot be coherently construed as concreta.

(POH)$_4$ So, if there are such type of artworks, then those artworks must be abstract objects.

(POH)$_5$ There are such artworks.

(POH)$_6$ So, there are such things as artworks that are abstract objects.

In the philosophy of art, there seems to be general agreement that reasoning along the lines (POH)$_1$ to (POH)$_6$ must be taken seriously.[57]

However, what exactly are repeatable artworks, and why can't we construe them as concrete objects? The common belief is that physical objects of the same sort are different only if they do not occupy the same spatiotemporal location. Now, when it comes to repeatable artworks however, the relevant identity criteria are distinct for they can be in different spatiotemporal locations. As Levinson summarizes,

> "Philosophers have long been puzzled about the identity or nature of the art object in nonphysical arts, e.g., music and literature. In these arts—unlike painting and sculpture—there is no particular physical 'thing' that one can plausibly take to be the artwork itself." (Levinson 1980, 5)

[57] Wollheim (1968) and Wolterstorff (1980) are prominent examples; for an overview of the history of the ontology of art, see Livingston (2008). For a survey of similar conclusions in other disciplines, see Sanfilippo (2021).

Put differently, unlike sculptures or paintings, repeatable artworks are characterized by their inability to be singled out by specific/individual copies. Not only can they survive changes in their material support, but they could also have been made of different materials. Repeatable artworks are *modally flexible* with respect to the matter they are made of. In the wake of these conclusions, most scholars working on the ontology of artworks agree that the (POH) hence calls for an investigation into the metaphysical nature of such art-abstracta.

Now, I submit that programs in the sense of the one introduced a few pages ago (sect. §2.1) also violate the (POH) and must, therefore, also be understood as abstract objects. Although programs undeniably differ from repeatable artworks in many important ways,[58] they share the feature that's crucial for the applicability of the (POH) – repeatability. Let me unpack the reasoning behind this claim to make sure that the argument holds; it essentially hinges on two observations:

The first (and more straightforward observation) is that nothing in the logical structure of the (POH)'s argument depends on 'artworks' per se. In principle, one may plug in any other kind of entity 'X' as long as all of the six propositions equally apply to the entity chosen. Importantly, in our case, we modify the (POH) for our purposes and fill in the term 'computer program.' Second, in order to assert that programs are abstract in the relevant sense, we need to make sure that they indeed conflict with the modified (POH). For so doing, it is vital to look at the second premise $(POH)_2$, according to which programs are either repeatable or non-repeatable. This leads us to the question – in what sense are programs repeatable?

To answer this question, it is chief to note that 'repeatability' in this context is the capacity of an entity to have multiple instances. Admittedly, 'repeatability' is not in the standard computer science vocabulary. Instead, it is customary in the philosophy of (computer) science to speak of *multi-realizability*. 'Multi realizability' is an influential notion in the philosophy of mind (Bickle 2020) and is frequently employed in the discourse on physical computation (Milkowski 2016).[59] In the context of computation, multi realizability then expresses the idea that many distinct physical systems can implement the *same* sequence of computations; they are so-to say, repeatable. As Duncan aptly summarizes

[58] It is important not to misconstrue programs as artworks (though some programs might be).
[59] In fact, as we have seen in my discussion on type-(B) implementation in the previous chapter, it is the multi-realizability of physical computation that is thought to give rise to the so-called triviality arguments. Since physical computation is not bound to a specific substrate, it can be realized in many physical systems. Very roughly put, this versatility makes it difficult to draw the boundaries between systems that compute and those that don't (the problem of extensional adequacy).

> "A software program, similar to a novel, is a generically dependent entity. A particular software program does not depend on a particular independent entity (such as a particular DVD or flash drive) in order to exist. Rather, a software program exists as long as it is borne by some independent entity. For example, if you destroy my DVD of Microsoft Word, Microsoft Word (the software program) does not cease to exist." (Duncan 2014, 38)

Now, none of this should strike us as controversial. After all, everything I said so far is basically contemporary philosophical jargon for expressing that programs – in the textual sense – are portable and frequently have copies. In fact, one of the main points of developing high-level programming languages was to develop a notational scheme decoupled from the underlying machines' idiosyncrasies and make them portable (see Appendix A, Notational View). As such, it does not make sense to speak of the location of a program. Just as there can be many copies of the novel Sherlock Holmes, there can, e.g., be many implementations of our textual example program.

3.2.4 Taking Stock

In order to apply the (PoC) to computer programs, we need to know what we mean when we talk about programs and in what sense they are artifacts whilst simultaneously thinking about them as abstracta. In line with these demands, this section introduced a simple program as our conceptual laboratory (§2.1), described how they are created, and discussed how the well-known (POH) from the philosophy of art licensed us to think of programs as abstract (sect. §2.3).

So far, I have only introduced the (PoC) informally and in bypassing. In the next section, I will flesh out the issue more rigorously.

3.3. The Problem of Creation

The Problem of Creation (PoC) is a philosophical problem from the philosophy of art originally pertaining to works of literature, musical compositions, and fictional characters (Deutsch 1991; Cameron 2008; Irmak 2020). In recent years, the problem's scope has successfully been extended to other metaphysically puzzling entities, such as scientific theories (French & Vickers 2011; French 2020). My goal in this section is to introduce the problem in general terms and provide an overview of its potential answers. The purpose of this presentation is that it will be helpful in eventually applying the (PoC) and its ramifications to the realm of computing.

So here it is. When referring to the entity under scrutiny as *X*, the pattern of reasoning of the (PoC) takes the following form:

The Problem of Creation (PoC):

X_1: *X*s are abstract objects (POH).

X_2: *X*s are created.

X_3: Abstract objects cannot be created.

At first sight, there are good reasons to accept propositions (X_1)-(X_3) *individually*, as all of them appear perfectly well-founded. The first proposition, (X_1), hinges on the validity of the (POH) that we previously discussed in sect. §2.3. In other words, as long as the (POH) holds for whatever we plug in for *X*, *X* is an abstract object in the sense that it is causally inefficacious.[60] Regarding the second proposition, (X_2), creation is typically regarded as a causal activity. On this view, we are causing *X*'s existence. With this in mind, the last proposition, (X_3), asserts that abstract objects (i.e., objects lacking spatiotemporal location) cannot be created, since it would require a causal process between an agent and a spatio-temporal entity.

However, the three propositions are mutually *inconsistent*. This paradox has sparked substantial debate for many years, leading to the question of which proposition of (X_1)-(X_3) we are willing to reject. Accordingly, three major options can be identified:

1. *Platonism*
2. *Nominalism*
3. *Creationism*

In a nutshell, each of these three positions is the result of rejecting one of the (PoC)'s three propositions (X_1)-(X_3).

In what follows, it is vital to understand the metaphysical implications of each of them because, as I will argue, they apply *mutatis mutandis* to computer programs. Accordingly, each option will be mapped out in the separate subsections (§3.1) to (§3.3). Since the following three positions can be discussed on a general level (i.e., pertaining to a general metaphysical doctrine) and in particular subdomains (mathematics, aesthetics, music, etc.), there is a tremendous amount of literature to keep track of. To do justice to these different options, it is worthwhile to illuminate the relevant details. While this may appear as getting sidetracked too much into the philosophy of art, this 'detour' will turn out to be helpful for identifying the corresponding notions for computer programs.

[60] The reasoning here is that most accounts of causation assume the relation between cause and effect to be a spatiotemporal relation in the realm of concreta.

3.3.1 Platonism

Platonism is the view that posits the existence of abstract objects, which are non-physical and non-mental entities that exist outside of space and time. Under contemporary Platonism, abstract objects are unchanging and entirely causally inert, i.e., they cannot physically interact with other objects (Balaguer 2016). Accordingly, abstract objects' existence is deemed mind-independent and does not crucially hinge on us. As such, Platonism is one of the potential answers to the (PoC): In order to resolve the paradox, Platonists endorse that there are abstract objects (X_1) and that they cannot be created (X_3), while denying, that X is created (X_2). Put differently, the view advocates that abstract objects are *not* created but exist independently of us.

Philosophers have applied this metaphysical view to a wide array of things that people have considered to be abstract objects: Logico-mathematical objects like numbers, propositions, universals, words and sentences, fictional characters like Donald Duck and Sherlock Holmes, and repeatable artworks like novels and musical compositions. In principle, endorsing Platonism does not require one to be a Platonist about this entire list of objects. In other words, one can follow a piecemeal approach and be a mathematical Platonist but favor non-Platonist proposals when it comes to fictional characters or computer programs.

Accordingly, there may be different reasons to maintain a Platonistic outlook towards different entities X. For instance, some philosophers have defended mathematical Platonism due to mathematics' essential role in science. [61] Especially Putnam (1971) and Quine (1976) argued that we should believe in the existence of abstract mathematical entities because of mathematics' *indispensability* in the empirical science. In the literature, the argument is known as the 'Quine-Putnam indispensability argument' (Colyvan 2001b; Liggins 2008; Colyvan 2024).[62]

Besides that, some philosophers consider, e.g., works of music to be abstract objects and endorse a form of musical Platonism (Kivy 1983; Dodd 2000; Dodd 2002; Dodd 2007; see Kania 2013, 198-205 for a summary of contemporary musical Platonism).[63] In this vein, Dodd, e.g., contends that musical works are abstract eternal types, where the latter correspond to sound structures that we can discover. Musical Platonism (and Platonism in general), hence denies musical works (and many other abstract objects) the status of artifacts. As a response, the composition of a musical work cannot be seen as an act of creation but instead

[61] There are many different versions of mathematical Platonism (Bueno 2020, 92; Linnebo 2024).
[62] Despite the name, the Quine-Putnam argument differs from Quine's and Putnam's individual positions. I will not delve into the details and plausibility of the overall argument here.
[63] Similarly, Richard Wollheim (1968) argues that literary works are types of which copies are tokens.

should be understood as a 'creative discovery' where composers bring to light something already 'there' (Dodd 2000, 427-434). So, whenever someone (or something) produces, say, a melody *m*, they produce a token of the type (i.e., a 'melody *m* sound structure'). N.b. though, when considering non-mathematical candidate abstract objects, we arguably lack any comparable indispensability argument that would warrant ontological commitment to them. Musical compositions or fictional characters are different from the kind of things that are indispensable to our best scientific theories.

Moreover, unlike the discovery of mathematical proofs or theorems (which can go wrong), Beethoven could not have made a mistake with his 'discovery' of the Archduke Trio (Sharpe 2001). As French (2020, 102) remarks, musical compositions thus seem to be conceptually dependent on their creative act in such a way that the process could not lead to anything other than the work. However, there is no such dependence when it comes to the discovery of mathematical proofs. Put differently, while mathematicians can devise a flawed mathematical proof, it is questionable whether a musical composition can be wrong.

The point of contrasting these different forms of Platonism is that the view may be corroborated – or argued against – by differing arguments pertaining to specific entities. These considerations are essential to remember when turning our attention to computer programs. Before moving on to the next answer of the (PoC), I must still mention one of the main points of contention of all forms of Platonism – the process of discovering abstract objects. While Platonists see it as a benefit not having to account for the creation of abstract objects, the flip side is that it poses a challenge of explaining how we can know of and discover these objects.

In the philosophy of mathematics, some aspects of the issue are discussed under the label of the *Benacaref Problem.* The problem is named after Paul Benacerraf, who first presented it as a challenge for mathematical realism in his 'Mathematical Truth' (Benacarref 1973). The problem has been widely influential and is thought to generalize to Platonism of other abstract objects (Clarke-Doane 2016). The argument concerns our lack of epistemic access to mathematical and other abstract objects. Generally speaking, epistemic access arguments start with the assumption that causal relations give rise to our cognitive apparatus.[64]

[64] Benacaref formulated the argument in terms of a causal theory of knowledge. Today's majority of philosophers deny that it holds in full generality. Many, therefore, resorted to Field's (1980) presentation of the problem that is couched independently of any theory of knowledge. For the sake of the current somewhat coarse-grained presentation of the topic, I omit the details for now; see, e.g., (Clarke-Doane 2016, 20-22) and (Cowling 2017, 135-138) for more detailed discussion.

Perception, for instance, crucially hinges on the causal interaction between agents and the world. In so far as perception and other causal cognitive processes furnish us with much of our knowledge, it is difficult to envision how we might acquire knowledge or justified beliefs about some subject matter without standing in a causal connections to it (Cowling 2017, 131). Absent any further qualifications, this reasoning suggests that our ability to gain knowledge of subject matters from which we are causally isolated is moot. As such, the Platonist about any *X* should also address the epistemological challenge of how we can have access to *X* as a causally efficacious entity.

3.3.2 Nominalism

Nominalism, sometimes dubbed 'anti-realism,' is the second main answer to the (PoC). This metaphysical position rejects proposition (X_1) by maintaining that a candidate abstract object *X* does not exist or turns out not to be abstract after all. In the latter case, we need to think of *X* in terms of some suitable concrete replacement. Put briefly, there are thus two principal flavors of nominalism (Kania 2013, 207-208), as one can reject the (PoC)'s first propositions in two different ways:

(a) Eliminativism: According to eliminativist theories, entities/objects *X* do not exist at all,

(b) Materialism: According to materialist theories, the objects *X* in question do exist but not as abstract objects

As with Platonism, the scope of Nominalism can vary greatly (e.g., some may wish to eliminate only specific entities from our ontology). One of the standard appeals of these options is that they are thought to be metaphysically parsimonious as they do not posit the existence of 'mysterious' abstract objects. Although (a) and (b) bear some similarities in terms of motivation, it is helpful for conceptual clarity to discuss them separately.

Option (a): Eliminativism

The distinguishing claim of option (a) – Eliminativist theories – is to deny the existence of entity *X* (e.g., mathematical objects, universals, repeatable artworks, and so on) wholesale.[65] According to an all-encompassing form of Eliminativism, there are only concrete objects (a shared commitment with option (b)), but none of them are identifiable with the abstract object in question; according to

[65] N.b., as with Platonism, one may choose a piecemeal approach and only endorse an Eliminativist attitude for specific entities.

particular forms of Eliminativism, we should only deny existence to specific objects. What are some of the strategies to motivate this view?

One prominent line of reasoning in this regard is how mathematical nominalists occasionally attempt to reconstruct the usage of mathematical language (in science). Hartry Field's rejection of the indispensability argument of mathematics is a prominent case in point. Field (1980) argues that mathematics is dispensable to science by exemplifying how we can avoid its usage in Newtonian gravitation theory.[66] Often, his and other's approaches in this vein are dubbed *mathematical fictionalism* since our mathematical statements do not turn out to be true (due to their not being any corresponding mathematical entities) (Balaguer 2023).

Outside the realm of mathematics, philosophers with an Eliminativist leaning have employed similar strategies of avoiding reference to the to-be-eliminated abstract objects. For example, while those who wish to eliminate works of music from our ontology would admit that there are performances, recordings, creative actions of the composers, etc., they would deny that any of these can be identified with the musical work itself – ergo, there are no works of music.[67] Note that as earlier in the mathematical, this conclusion raises urgent questions; if works of music do not exist then what exactly are we talking about when speaking of musical compositions? The point is that similar pressing questions generalize to *all* sorts of seemingly abstract entities *X* that are supposed to be eliminated from our ontology.

One of the main strategies to answer these sorts of questions is *paraphrasing*. The idea can be understood against the backdrop of Quine's criterion of ontological commitment defended in his influential article 'On What There Is' (Quine 1948). According to Quine, the usage of a statement containing a name or singular term of the form 'There is some *X*' commits us to the existence of the term *X* (or anything fitting that description).[68] The idea of paraphrasing is to rewrite our sentences in such a way that we can eschew reference to the particular entity, and therefore avoid ontological commitment to it.

[66] Another prominent series of objections against mathematical indispensability arguments can be found in the work of Penelope Maddy; see, e.g., Maddy (1992).

[67] As Kania (2013) clarifies in his overview of musical Nominalism, only a few have opted for the elimination of musical works. Rudner's (1950) is arguably the closest position in that regard (though, according to Kania, it is possible to interpret his account as Materialist). Other Examples are Cameron's (2008) 'There Are No Things that are Musical Works' and Steven French's and Peter Vicker's work based on it (French & Vickers 2011; French 2020).

[68] Strictly speaking, Quine's criterion only applies to theories (i.e., sets of sentences) formulated in first-order predicate logic that contain existential and universal quantifiers. We thus need to translate a sentence in question into first-order logic and then assess its ontological commitments based on what the translation quantifies over (Bricker 2016).

Exemplifying the procedure for musical works will clarify matters. According to the proponent of the paraphrasing strategy, to talk about (abstract) works of music is to merely 'superficially' talk about these abstract entities. While talking about works of music might be very useful and common practice in everyday life, it turns out the fundamental furniture of the world does not include such works. This can be made sense of by claiming that sentences like 'There are musical works' admit two systematically different kinds of uses (Dorr 2005; 2008). According to Dorr, we may use sentences in a superficial and *fundamental* way. So, when we engage in everyday life talk about works of music in plain English, we are said to merely talk about matters in a superficial way. What really matters, though, is our ontological commitment when we speak about entities in a fundamental way (sometimes philosophers call this fundamental language *Ontologese*).

Despite the claims of Eliminativists that their view is ontologically simpler, it requires a considerable amount of theoretical underpinning for accepting such philosophically thorny concepts like 'fundamentally' and 'Ontologese.' Additionally, one may reasonably doubt that consistent and plausible paraphrases can always be found for sentences involving the entity *X that* they want to eliminate.[69]

Option (b): Materialism

According to option (b) – materialist theories – entity *X* is not actually an abstract object but something concrete. In the case of musical works, candidate concrete manifestations are score copies, performances, recordings, playing records of musical performances, and so on (Tillman 2011, 15).[70]

Analogously to the previous metaphysical views presented so far, one may be a materialist about some things while taking a different stance about other entities. Tillman aptly captures some of the alleged merits of (musical) Materialism:

> "Some of the advantages of any form of musical materialism are obvious […]: if musical materialism is true, there is no mystery about how a musical work can be created, temporally located, and hearable." Tillman (2011, 28)

Note that Tillman's explanation applies to Materialism in general (as opposed to Materialism about musical compositions specifically) by erasing the adjective

[69] For a short version of this argument, see (van Inwangen 1977, 303-304); for an in-depth analysis of the problems associated with paraphrasing, see (Wetzel 2009, 53-92).

[70] Matters are further complicated because most metaphysicians distinguish between endurantism and perdurantism; see Appendix A, the Physical View. In line with this distinction, Tillman maps out the different options (Tillman 2011).

'musical.' While this sort of parsimony is generally regarded as one of the merits of Materialism, it does not come without challenges.

Most notably, Materialism typically flies in the face of our beliefs informed by our practices. For instance, according to musical Materialism, a musical work is a concrete object or event; likewise, mathematical objects like the number 3 or √-17 would be physical. As explained earlier when introducing the (POH), the decisive point is that many practitioners and philosophers of art consider (due to their repeatability) their works to be abstract. Now, what is the reply to this situation?

In defending musical Materialism, Caplan and Matheson (2006) supply an answer that may be extended to other forms of Materialism. Given the (POH)'s insights, they also reject that a musical composition is identical to an *individual concrete* performance. Instead, the authors identify works of music with the *mereological sum* of all the performances, scores, and other concrete particulars. However, as Wollheim previously pointed out (Wollheim 1968, 6), considering a work as the totality of all its copies potentially poses problems. Equating the work with the class of its copies may be problematic since the former may be finished, whereas the latter is not (since new copies are created or old ones destroyed). Moreover, any proponent of such a fusion strategy needs to explain the relation between the alleged copies of a class; put differently, one needs to spell out what qualifies them as an appropriate member.

3.3.3 Creationism

The last alternative to solve the (PoC) is *Creationism* (sometimes also called 'abstract creationism'). Creationism refers to those views according to which it is possible to create abstract objects. In other words, creationist views embrace (X1) and (X2) while rejecting (X3). Similarly to the previous options, the view is a theoretical umbrella for a host of different proposals about different entities (Friedell 2021). Rather than cataloging all of them, I canvass a few sources and the most crucial features that typically underpin them. One of the main motivations to be a creationist (about works of literature) is described by Deutsch, who states that

> "[…] authors do not literally discover their stories. Conan Doyle did not somehow find out that the proposition that Sherlock Holmes is a detective is true in the stories he set out to write down. On the contrary, he simply *stipulated* that this proposition is to be true in the stories. Anyone who holds that literary creation is not literal creation but rather literal discovery, has a great deal of explaining to do." Deutsch (1991, 212)

Deutsch's doubts about the discovery of 'stories' may also apply to other kinds of abstract entities. Many practitioners and philosophers alike believe that (at

least some) abstract objects are the result of human creation. Popper (1978), for instance, argued that next to the world of physical states and that of mental states, there is a third ontological category ("World 3") that contains abstract (cultural) artifacts like the American Constitution, Beethoven's Fifth Symphony or Newton's theory of gravitation. Similarly, in his 'Creatures of Fiction,' van Inwangen (1977) argues that fictional characters require a placement in their own ontological category (that enables them to be abstract yet created) and Searle (1995) notes that many cultural and institutional entities can be brought into existence (through the intentional act of merely representing them).

Arguably, one of the most sophisticated frameworks in this regard is Amie Thomasson's *artifactual theory* (Thomasson 1999). Rooted in the phenomenological tradition and work of Roman Ingarden (1979), the artifactual theory was initially developed to tackle the ontological status of fictional characters. Roughly put, as per Thomasson, fictional entities are contingently existing abstract objects – called abstract artifacts. The important thing is that similarly metaphysically puzzling entities like works of literature, symphonies, constitutions, money, and perhaps computer programs can all be characterized as abstract artifacts, too (since they are all abstract, created, and may cease to exist).

One standardly evoked objection against her or similar proposals is that recognizing abstract artifacts requires an updated category system. The problem is that abstract artifacts do not fit into the traditional abstract-concrete dichotomy. Although abstract artifacts also lack spatial location, they are not timeless – they were created at a particular time and place and may cease to exist. As such, they do not fit into the realm of eternally unchanging platonic objects (Thomasson 1999, 37-38). Hence, the challenge for the creationist is to provide an adequate category system (with at least one more category) for abstract objects that are created.[71] (Based on simplicity criteria, the admission of additional ontological categories is too hard to stomach).

Another point of contention concerns the act of creation. The worry with creating abstract objects is that there should be no causal interaction between the abstract and concrete, as the concrete domain is typically considered causally closed. In response, some abstract Creationists have suggested that the physical can stand in a causal relation with the abstract or have challenged the idea that creation is a causal process. For instance, Irmak (2020) argues that existential dependence (featured in the artifactual theory) allows for the non-causal creation

[71] The artifactual theory offers a theory of existential dependence, introducing varying degrees of mental and material dependence and their relation to each other. According to Thomasson, this idea eventually leads to a multi-dimensional ontology – a system of existential categories that has much less trouble hosting previously metaphysically troublesome appearing entities.

of abstract artifacts. While creation still involves causal interactions, the interactions in question are between concrete objects and/or events but not between the creator and the abstract object itself. All that the creation of abstract objects involves is the manipulation of the entities and events on which the existence of abstract artifacts depends.

3.3.4 Recapitulation

This section placed a magnifying glass over the (PoC) and sketched the three main options to solve the issue. By doing so, I shed light on what kind of philosophical issues we must think about and what sort of answers we can expect. The upshot is that none of the three previously portrayed philosophical positions is internally inconsistent or incoherent – each of them is a defensible view. However, that said, each position also faces serious objections. Thinking through these objections requires reflecting on broader, long-lasting metaphysical puzzles regarding causation, the abstract-concrete dichotomy, ontological parsimony, paraphrasing, and so on. The task now consists of sorting out to what extent these issues carry over to the realm of computing.

3.4 From Art to Computing

The earlier sections have set the stage for understanding the ontological status of computer programs under the (PoC) framework. Due to their multiple realizability, it is reasonable to consider computer programs created in high-level programming languages as abstract objects. Furthermore, I briefly mentioned that our understanding of how programs are created is supported by a wealth of literature on their production. Based on these initial findings, I submit that we can also apply the (PoC) to computer programs:

The Problem of Creation (PoC) applied to Programs:

***P*$_1$:** Programs are abstract objects (POH).

***P*$_2$:** Programs are created.

***P*$_3$:** Abstract objects cannot be created.

As I will now show, transposing the (PoC) to computing supplies us with an updated, metaphysically sound range of answers to my initial question about what kinds of (abstract) things programs could be. Having identified (i) a Platonistic stance, (ii) a Nominalistic stance, and (iii) a Creationist view as the main contenders, I now explore the plausibility of each of these options when applied to programs.

3.4.1 Are Programs Platonic objects?

One of the main options for viewing programs as abstract objects is under the umbrella of Platonism. On this view, the source code of my example (GCD) program turns out to be a non-physical and non-mental entity existing out of space-time. Given the wealth of different Platonistic frameworks available today, the claim may have different motivations. As far as I can tell, two approaches have been *sketched* so far – an indirect and a direct one:

The first, the indirect one, stems from a mathematical outlook on computer science and its objects. (I call this position indirect because it is primarily informed by a certain stance on mathematical objects and not directly on programs itself). Due to the pervasive employment of logico-mathematical concepts in computer science, its practitioners may view programming as essentially a mathematical activity.[72] Although my previous survey about the potential answers to the (PoC) revealed that there are nominalist alternatives concerning mathematical objects, Mathematical Platonism remains a widely embraced option.

In a recent critical review of Turner's *Computational Artifacts* (Turner 2018), Selmer Bringsjord essentially expressed a version of Mathematical Platonism about programs when arguing, contra Turner, that

> "I doubt very much that there *are* any artifacts of computer science. The reason is that the core elements of computer science are logicist, and as such are immaterial. As to computer *engineering*, well, yes, that might be a rather different story, but it is one we ought to ignore: we are discussing not philosophy of computer engineering, but of computer *science*." (Bringsjord 2019, 340)

Although, Bringsjord appears to overlook the possibility that, as per abstract creationism, artifacts can be abstract, his quote nicely encapsulates the fact that computer programs are tightly interwoven with logic and mathematics that other candidate abstract objects like works of literature and music are not. So, if you believe that computer programs are some sort of logico-mathematical entities *and* you simultaneously subscribe to Mathematical Platonism, then you are indirectly committed to (your preferred version of) Platonism about programs.

The second approach, the direct one, suggests that computer programs are Platonic objects (without necessarily claiming that they are also mathematical objects). Seeing things this way comes close to having a Platonistic attitude towards works of literature and music encountered earlier. This view, therefore, lends itself to conceptual borrowing from non-mathematical theories of Platonsim. One very recent case in point is the work of Begley (2024), which advertises a realist metaphysics of software maintenance. In clarifying his

[72] See my elucidation on the Mathematical View in Appendix A.

understanding of 'software' in this context, he also discusses the ontological status of computer programs:[73] Based on Katz's *Realistic Rationalism* (Katz 1998), in which an ontology of composite objects is defended, Begley claims that we can identify software "as being a set of sets of program *types*" (Begley 2024, 180; own emphasis, where we should understand the latter as "made up of algorithms, that is, generally, finite progressions of operation types." (ibid.).

In particular, he compares algorithms to so-called discourse types (informed by Katz's theory of linguistic entities). Going into the intricate details of Katz's approach would lead us too far astray, but very roughly put, on this view, algorithms are similar to novels, poems, and speeches since they are language expressions, too.[74] Begley, therefore, seems to suggest a Platonistic stance towards programs because of their similarities with these repeatable artworks; he thus answers the (PoC) of computer programs with the Platonist option.

Methodologically, both the indirect and the direct Platonistic view in its *current form* leave a couple of questions unaddressed.[75] First, as with any Platonist theory, there is the drawback of epistemic access type arguments and the unintuitive result that programs are not created and cannot be destroyed; instead, they can be found and lost. Applied to my initial example program, one must reconcile how our ordinary understanding of programming as a creative activity is compatible with the notion that the source code depicted in Fig. 3.1 was 'discovered.'[76]

Second, pertaining to the indirect view (that regards programs as mathematical objects of some sort), it remains somewhat unclear which of computer science's entities in the computational hierarchy are supposed to be Platonic objects. While I limited my focus on a particular LoA, namely the program's source code, the proponents of the indirect view might be better advised to maintain their Platonic stance towards algorithms. On this view, one could, for instance, maintain that the (GCD) algorithm implemented in our exemplary C-program is a Platonic object, while the source code is not.

Lastly, regarding the direct view, providing a more precise distinction between programs and algorithms would be beneficial. For instance, although Begley informs the reader that his account is informed by the practices of theoretical computer science (formal program verification and computational

[73] In the following, I will only focus on his elucidations on programs to keep things simple.
[74] This interpretation is possible because, as mentioned earlier in the Introduction (see also Appendix A, the Mathematica. View), the concept of 'algorithm' is subject to many different interpretations, too.
[75] I stress 'current form' here, because both views have not been fully developed yet.
[76] This is essentially the concern that Bringsjord expressed in his quote about computer *engineering*.

complexity are explicitly mentioned), his view not only seems to equate programs and algorithms but also maintains that the latter are primarily linguistic entities.

3.4.2 Nominalism about programs?

The second main line of answers denies the existence of programs qua abstract objects. What exactly are the ramifications of the stance regarding programs? From the perspective of (the philosophy of) computer science, such anti-realistic attitudes are perhaps the most unexpected or implausible views. Not surprisingly, the most motivating factor stems from broader metaphysical principles, not computational ones. For example, one often-invoked argument to reject abstract objects is Ockham's razor. This principle advertises metaphysical parsimony by stating that we should not unnecessarily introduce more (types of) entities to our fundamental ontology than needed. As we have seen in the previous section, there are two primary strategies to render the role of abstract objects obsolete – the eliminativist and the materialist option. Let me treat them in turn.

Eliminating programs

The eliminativist option seeks to deny the existence of abstract programs. Again, at first glance, eliminating programs from our ontology may seem preposterous as it completely runs against our intuition. (N.b., based on this intuition, I even neglected what I called primary ontological questions (POQ) at the beginning of my thesis and assumed that there, in fact, are programs). For instance, the very fact that you are reading this was made possible by various computational aids and computer programs. In light of these obstacles, the eliminativist thus has to offer an argument that shows how our currently best theories in computer science do not commit us to the existence of programs after all. How could such an argument possibly look like?

One way to answer this question is to look elsewhere. For instance, despite similar initial worries about other (ostensible) abstract objects, recently, eliminativist-flavored approaches have gained currency both in the philosophical discourse of art and science. Based on the work of Cameron (2008), in which he defends a view that reconciles competing intuitions about the existence of musical works by appealing to Ontologese (the language that only refers to 'fundamental' entities), French & Vickers (2011) and French (2020) have formulated an analogous proposal for scientific theories. Roughly put, Cameron maintains that English sentences like 'there are works of music' are true despite there not being actually such works. He relies on a meta-ontological view

whereby 'X exists' can be true in English without committing us to an entity that is X. Following suit, French and Vickers have amended Cameron's theory to accommodate scientific theories. Although, as far as I am aware, no one has attempted to develop this particular way of thinking about programs, it seems one of the most promising starting points. On this view, one would hence deny the existence of abstract programs like my example C program. English sentences about the abstract nature of programs would hence merely be a *facon de parler* that can be paraphrased away.

However, given the absence of a fully worked-out eliminativist framework for computer programs,[77] it remains somewhat speculative which amendments this view would have to make to accommodate programs. To start filling this gap, I will briefly assess the situation after discussing the materialist option.

Materialism about programs

Despite everything I have said about the abstract nature of computer programs in this chapter, the view that programs are material is not entirely unappealing, as it could resolve many philosophical concerns pertaining to the metaphysically troubling nature of programs. After all, many view computer programs as physically executable entities that are involved in the causal pathway.

Perhaps the closest who expresses such a materialist position is Marcus Rossberg. In his (Rossberg 2012), he discusses the destruction of works of art, including computer art. In trying to supply an ontology for programs, he states that

> "A computer program itself is repeatable, of course; it can run on different computers and at different times. In order not to jeopardize destructibility, we can follow our now familiar method and opt for the plausible account of programs (and operating systems) as equivalence classes of inscriptions. The inscriptions will typically not be ink on paper but electronic and on some computer storage device such as a hard drive, memory card, or old-school flopp disk. Either way, such inscriptions will be concrete, physical objects" (Rossberg 2012, 73)

This orientation mirrors Caplan & Matheson's (2006) fusion strategy we encountered earlier. To recap, the authors maintained that musical works are not abstract but the composite of all musical performances. To talk of abstract works is to talk *as if there* were abstract entities (yet only such things as concrete performances, score copies, and so on exist). Works reduce to linguistic items – general names or descriptions – that serve as convenient tools to refer to certain

[77] Kittler (1993) is an exception. However, his essay was arguably meant to be a polemic commentary on how software products constrain the user instead of an attempt to eliminate programs from our ontology.

classes of concrete particulars (Goehr 1992, 16-17). In the case of a musical work, works are no more than extensionally defined classes of performances.

Analogously, Rossberg suggests thinking of programs as equivalence classes of inscriptions. However, alas, he, or anyone else for that matter, has not provided the means to recognize the members of the corresponding equivalence class. Similar to the (potential) eliminativist project about computer programs, the materialist version remains, so far, in an infant stage.

In the wake of my explorative quest of charting the consequences of the (PoC), let me point out some of the potential obstacles that both the here-developed nominalist versions need to address to present themselves as viable alternatives.

One aspect that strikes me as worth discussing about Nominalism about programs is whether it would leave our understanding of physical computation intact. For instance, if one were to develop an eliminativist attitude towards programs, it would stand at odds with today's insights of the type-(B) implementation literature (cf. Chapter 2), which necessitates a mapping between abstract computational states and physical states. Dismissing the existence of abstract programs not only renders the concept of mappings obsolete but also raises questions about an alternative characterization of physical computation. [78]

In so far as the problem of implementation is a special instance of the bridging problem,[79] the would-be eliminativist could be well-advised to take inspiration from the debates about similar worries in the philosophy of mathematics. Roughly put, nominalists face the challenge of explaining the astounding applicability of mathematics in science, despite not being committed to these entities. Since, on this view, mathematical objects do not exist, it becomes unclear how referring to such entities can contribute to the empirical success of science. Broadly construed, there are two different kinds of answers for the nominalist (Bueno 2022): The first requires reformulating mathematical or scientific theories to avoid commitment to mathematical objects, for instance, proposed by Field (1980). The second one does not require the reformulation of theories; instead, it explains how no commitment to mathematical objects is involved when using these theories (e.g., suggested by Azzouni 2004). Going down either of those roads would thus entail a complete revision of how most theoreticians have thought about computational implementation until now. [80]

[78] As Curtis-Trudel (2022) recently pointed out, so-called unificationist theories of implementation face serious objections in accounting for physical computation.

[79] Typically, the applicability of mathematics is spelled out in 'mapping accounts' that establish a correspondence between the mathematical and the physical (see Appendix B for an in-depth discussion).

[80] Of course, driven by ontological parsimony, nominalists can choose an anti-realist attitude towards physical computation, so none of what I said would be their concern.

Likewise, if one were to develop a materialist attitude about programs, one would also have to revise the current solutions to the Problem of Implementation. To recap, roughly put, the crux is how to differentiate physical systems that carry out computations from those that do not. Again, the reason to rethink the issue is that it is no longer sensible to couch the problem in terms of a mapping between abstract computational states and physical states; according to the materialist, our ontology does not entail the former but only the latter. Materialists, hence, need to establish a criterion to define an equivalence class of inscriptions to distinguish electronic inscriptions on computer storage devices, memory cards, or old-school floppy disks from other random physical states. For example, they likely do not want to include electronic states in appliances like toasters or rice cookers to count as inscriptions of computer programs.

3.4.3. Are Programs Abstract Artifacts?

Viewing programs as artifacts has grown in popularity among philosophically inclined scholars in recent years (Lando et al. 2007; Faulkner & Runde 2010; Irmak 2013; Duncan 2014; Turner 2011; 2014; 2018; Wang 2016; Sanfilippo 2021).[81] Today, these views arguably dominate the, albeit scattered, literature on the ontological status of computer programs (see Appendix A). However, perhaps reflecting the novelty and relatively fragmented state of the debate in general, a consensus has yet to be formed about which theory of artifacts we ought to subscribe to. Two popular conceptions stick out.

On the one hand, there is the Computational Artifact View (Lando et al. 2007; Turner 2011; 2014; 2018). In the previous chapter, we encountered this view when discussing the different notions of type-(A) implementation. Remember – the takeaway was that the approach is based on the technical artifact literature. The latter postulates a duality between structural and functional properties, where the structural side is satisfied by the *physical* objects involved and the functional side by intentionality (e.g., Kroes 2012). Although technical artifacts were initially devised exclusively for physical systems and their causal structure, the novelty about computational artifacts is that they supposedly also account for abstract objects and their abstract structure. As Turner proposes, we can employ *formal languages* to account for a computational artifact's abstract structure:

> "At both the functional and the structural level, computational artifacts employ formal languages for the expression of their functional and structural properties." (Turner 2018, 29)

[81] See Appendix A for a more detailed summary and comparison of the positions.

So, while computational artifacts retain the structure-function duality, their structure is no longer physical but *symbolic* in nature. It will be useful to bear this last point in mind until after my discussion of the second main option.

On the other hand, there is the view that programs are temporal abstracta (Irmak 2012). In arguing that most philosophical explanations of software have failed to recognize its *artifactual* yet abstract nature, Irmak started to develop such a view. [82], [83] Interestingly, his account is directly informed by Thomasson's (1999) artifactual theory and, thus, a bona fide example of abstract Creationism. In particular, he thus rejects the idea that programs are eternal mind-independent objects. Instead he suggests that a program is an abstract artifact, i.e., a temporal, nonspatial, repeatable, and a contingent entity that exist due to a certain purposeful creative act by one or more human agents. As per Irmak, we should therefore avoid regarding programs as types (which are typically seen as platonic objects; cf. Begley's (2024) position) and should not think of their implementation in physical systems in terms of the type/token distinction.[84]

Although both views arrive at virtually the same conclusion – i.e., that symbolic programs like my example (GCD) one written in C are best seen as abstract objects that can be created – their different intellectual heritage reveals some crucial philosophical differences: First, the notion of abstract artifacts developed in the arts does not have its roots in the technical artifacts literature and, hence, usually does not bear any additional normative function. Put differently, they are not characterized in terms of the function-structure duality that (according to the contemporary type-(A) implementation literature) is said to be essential for the correctness criteria of computational artifacts. Second, coming back to the nature of the symbolic structures mentioned above, notice that the abstractness of computational artifacts hinges on the assumption that these formal language expressions are abstract. While we have seen throughout this chapter that this is a widely embraced view (particularly under Platonism and Creationism), there are Nominalist alternatives undermining the idea.

Given both these shortcomings, future studies could try to merge the computational artifact with the abstract artifact view. In addition, it is paramount to note that both views have, so far, virtually remained silent about the standard objections against Creationism: the problem of how abstract objects can be

[82] In an email conversation, Irmak told me that 'software' may be used interchangeably with 'program.'

[83] The work by Wang et al. (2014a; 2014b) and Wang (2016) further refined Irmak's original proposal by focusing on the identity criteria of programs in the context of code changes. However, as stated in the general Introduction (Ch. 1), I will abstain from delving into the debate on when two programs are the same.

[84] Unfortunately, no alternative approach for the implementation is suggested.

created and the potential costs that come with an updated ontological category system that is needed to accommodate abstract objects that may start and cease to exist. For instance, in so far as we would endorse the abstract artifact view about programs, we would have to adjust our standard abstract-concrete distinction accordingly.

3.5 Discussion & Conclusion

Coming from the UTAI framework, I began this chapter with the quest to illuminate how some entities under the umbrella term 'computer program' are abstract objects. Conceiving programs as abstracta then led me to turn my attention to the notorious Problem of Creation (PoC) from the philosophy of art. Since the premises of the (PoC) are jointly inconsistent, philosophers have developed three main lines to debug the case. Each line comes with its own benefits and costs. While the (PoC) presents rich philosophical material, a systematic overview of the tradeoffs involved in adopting Platonism, Nominalism, and Creationism about programs has virtually been neglected in the philosophy of computing. In this chapter, I address the issue directly and argue that the concept from philosophy or art remains effective when applied to computer programs. *Prima facie*, it is not obvious which standard options to reject and which we ought to favor. If the philosophy of computing, particularly the debate on the ontological status of computer programs, keeps growing into an independent enterprise, then we can expect a hefty research program to flash out these *potentially defensible views for computer programs* precisely in the coming years.

Given that this reads more like the beginning than a conclusion to this chapter's question, I want to close by immediately responding to a possible objections. Particularly, one may object something along the following lines: 'Well, your presentation hasn't *solved* anything – you didn't answer the question you posed initially (about creating programs). All you did was raise additional questions. Even if all of this about the (PoC) is correct, you have now plagued us with even more options we need to consider. Wouldn't it be better to narrow, rather than widen, the scope of options?'

I agree that a definitive answer would be preferable (like in most philosophical puzzles). Yet, progress sometimes requires a step backward to appreciate a topic from a clearer point of view. I strongly believe that this chapter is a case in point. To defend this claim, let me briefly explain how embedding the discourse of the ontological status of computer programs in the context of the (PoC) can bring clarity to our discussions.

Primarily, the tripartite distinction of Platonism, Nominalism, and Creationism may act as a methodological blueprint for future studies within the philosophy of computing. Although I very much sympathize with Abstract Creationism about programs (mainly due to its descriptive adequacy of the relevant practice), my analysis has shown that it would be philosophically irresponsible to ignore the view's vulnerability to broader metaphysical issues (e.g., requiring an updated category system or explanation of how abstract objects are creatable). In other words, the reason why I did not provide a clear answer as to how programs qua abstract objects are created has to do with metaphysics in general, not with computing in particular.[85]

Furthermore, by endorsing the (PoC), the philosophy of computing can further mature through establishing a dialogue with contemporary metaphysical debates. To paraphrase what Mag Uidhur (2013) expressed, albeit in the context of the ontology of art – by pursuing this strategy, we can make the discourse about the metaphysical nature of computer programs less insular and, therefore, more attractive to new participants of the debate. For instance, seen through the lenses of the (PoC), the *sui generis* dual nature of view of programs loses appeal. Instead of positing metaphysically puzzling entities with a mixed ontology, my proposal allows us to frame the debate about the ontological status of computer programs in much more robust terms of existing debates on abstract objects. Like in other metaphysical inquiries, where it is customary to distinguish between types and tokens, universals and particulars, numbers and numerals, works and their instantiation, we can now more clearly distinguish between programs, qua abstract entities, and concrete manifestations. Importantly, this is not to say there is no puzzle about how these abstract objects then bridge the abstract-concrete dichotomy; this is, perhaps, still *the* significant puzzle that must be solved to fully grasp the ontological status of computer programs.

As we transition to the next chapter, I will offer a novel account of the *relationship* between these abstract objects and the physical systems that 'realize' them by devising a new theory of computational implementation.

[85] In fact, some participants of the debate even believe that there is a sort of 'stalemate' between the different positions. For instance, in his work on (mathematical) Platonism versus anti-Platonism, Balaguer (1998). Grafton Cardwell (2020) and Friedell (2021) address the broader philosophical commitments

4 Implementation-as: From Art & Science to Computing

Following the UTAI framework outlined in Chapter 2, the previous chapter systematically analyzed dependency relation (a)—how programs on an abstract level are contingent on us. It is now time to explain how these programs are implemented in physical systems. Put differently, we must examine what relates the abstract and the physical. Here, dependency relation (b) becomes relevant. How does the implementation rely on the epistemic agents who establish and use these programs?

4.1 Introduction

Computability theory allows us to engage formally with computation in mathematical terms. However, studying computation merely formally does not provide any details about its physical implementation. The fundamental problem that any account of physical computation must answer is how the two different ontological domains of the formal and physical are related. In the literature, this is known as the *Problem of Implementation* (Sprevak 2018; Ritchie & Piccinini 2018). Solving the issue is essential for disciplines such as the foundation of computer science, AI, robotics, and cognitive science. As a result, a vast literature of potential candidate frameworks has been presented.[86] Which of the proposals truly captures the nature of physical computation?

In order to judge competing accounts, Piccinini (2007; 2015) presented a convenient heuristic to evaluate them. Five desiderata were advanced:[87]

Desiderata of Physical Computation

(1) *Objectivity*: An account of physical computation should make it, at least in part, a matter of fact whether a system is implementing a computational function. The intention is to align computation with scientific practice and scientific objectivity.

(2) *Extensional Adequacy*: An account of computation should avoid triviality; in slogan form, it should proclaim that the *right things compute* (laptops and perhaps brains) and the *wrong things do not compute.*

[86] Some key sources that deal with (parts of) the Problem of Implementation are Putnam (1988), Searle (1992), Copeland (1996), Chalmers (1996), Scheutz (1999), Klein (2008), Ladyman (2009), Sprevak (2010), Milkowski (2013), Fresco (2014), Horsman et al. (2014), Rescorla (2014), Piccinini (2007; 2015), Dewhurst (2018), Fletcher (2018), Mollo (2018).

[87] I follow a slightly adjusted version of Duwell (2021) which merged "the right things compute" and "the wrong things don't compute" under "extensional adequacy."

(3) *Explanation*: The computations performed by a material system should, at least partly, explain its behavior and capacities
(4) *Miscomputation*: Sometimes, computation goes wrong. An account of physical computation should account for faulty behavior.
(5) *Taxonomy*: An account of computation should be able to untangle the different computational capacities of different systems (e.g., general purpose or fixed purpose; analog, digital, or quantum).

Virtually all solutions propagate that there is an equivalence relation between the computational formalisms of the mathematical theory of computation and the putative computing system. Simply put, the idea is to establish a mapping f between the sequence of states of an abstract model of computation M_C and the state transitions of a physical system S, such that $f: S_C \rightarrow M_C$. However, to date, no account of physical computation has championed all the others.[88]

This chapter contributes to the discourse by extending a promising recent line of research. In a nutshell, the idea is that the metaphysics of implementation bears notable similarities to scientific representation, as both relations rely on mappings between the physical and the formal. However, as we will see shortly, the idea still needs to be developed to its fullest. In this chapter, I respond to this issue by developing the novel, more detailed *implementation-as* framework. What sets this contribution apart from previous ones is that it relies on a *specific* notion of scientific representation rather than a generic one. Implementation-as is underpinned by the DEKI account (Frigg & Nguyen 2018), a formalized account of scientific representation based on Goodman's and Elgin's representation as originally developed in the philosophy of art. As I will show, the resulting account squares well with the standard desiderata of physical computation and is a viable alternative. Due to its agential involvement, it is especially suited for computer science practice.

The chapter is organized into several sections: Section 4.2 describes the state of the art of recent research to tackle the Problem of Implementation in terms of scientific representation. In section 4.3, I introduce the DEKI account. To facilitate discussion, I follow Frigg and Nguyen in introducing their account by appealing to the MONIAC, a hydraulic analog computer. In section 4, I transpose the DEKI account's features to the computing realm, giving rise to Implementation-as. Next, I evaluate this new theory of implementation against the five desiderata of physical computation. Lastly, I close with a brief discussion and comparison of Implementation-as to existing physical computation accounts.

[88] Of course, it is a viable option to take a pluralistic stance concerning accounts of computation.

4.2 Scientific Representation Accounts in Computing

Scientific representations concern a wide array of phenomena. One may use diagrams, mathematical equations, or material objects for representations in science. Most generally, any representation that is the result of scientific practice may be deemed a scientific representation. In this chapter, we are primarily interested in the case of (material) scientific models and how they represent.

To recap, in Chapter 2 we already encountered these scientific instruments, when proposing them as a remedy to link together type-(A) and type-(B) implementation. Particularly, I described how scientists use models to represent real-world or hypothetical systems for explanations, prediction, and confirmation. As such, philosophers of science typically characterize scientific representation as the relationship between a model M and its dedicated target system T as $f: M \rightarrow T$. What's crucial to remember for the current discussion is the following: While one then may use computational methods to model or simulate various real-world targets, philosophers of computing warned that one should not confuse the ability to model a system computationally with thinking that it also genuinely computes. At first sight, one is therefore well-advised to be cautious about using modeling techniques to solve the Problem of Implementation.

However, despite these worries, a new line of research proposed to couch implementation in terms of scientific representation and modeling. Let's call this approach the Scientific Representation account (SRA). Although this research cluster is still relatively scattered, it differs from traditional proposals of physical computation because it argues that the mapping relation *f* explicitly needs to be understood as a form of scientific representation. This perspective is based on a combination of epistemological, metaphysical, and historical considerations.

For instance, when developing a model of computation called *L-machines*, Ladyman (2009) suggests that physical computation might be contingent on (scientific) representation. Another case in point is Care's (2010) historical study shedding light on the use-centric history of analog computing as modeling. Likewise, but from a philosophical angle, Papayannopoulos (2020) highlighted the conceptual commonalities between analog computers and analog models (when developing a notion of analog computation). Arguably the technically most detailed account in that vein today is the *Abstraction/Representation (AR) Theory* introduced by Horsman, Stepney, Wagner, and Kendon (2014) and developed further in several publications.[89] Horsman and collaborators provide sophisticated 'commuting diagrams' in virtue of the *representational triple*

[89] Horsman (2015, 2017), Horsman Kendon, Stepney, (2017, 2018) and Horsman et al. (2017).

$\langle m_i, f, s_j \rangle$, where *f* is perceived as some *general* scientific representation account, and m_i and s_j corresponding computational and physical states, respectively. Subsequently, Fletcher (2018), Szangolies (2020), and Duwell (2021) critically assessed (AR) Theory under philosophical considerations and concluded that the approach is a viable contender if formulated in agential terms.

Importantly, these new SRAs reject the idea that something is physically computing because we can model it computationally. What the SRAs are after instead is the commonality of the metaphysical nature of the mappings involved in scientific representation and computational implementation, respectively. In both cases, we require a relation that links the physical and formal realms.

However, there remains a limitation with existing SRAs: So far, they merely allude to scientific representation in vague or generic terms. This lack of clarity is problematic since there is a wide range of scientific representation accounts, with isomorphism accounts, similarity accounts, inferentialism, and fictionalism being the most prominent options (Frigg & Nguyen 2021). Each option requires us to adopt significantly different or opposing metaphysical and epistemological assumptions. Utilizing differing notions of scientific representation can, therefore, lead to substantially different SRAs and understandings of concrete computation. For instance, if we used Suppes' isomorphism account (2002), according to which scientific representation is a two-place relation reducing to isomorphisms between structures, the resulting account of physical computation would be no different from some traditional mapping accounts in the physical computation literature. If, on the other hand, one were to follow Cohen & Callender's general Griceanism approach (2006), which suggests that anything may represent anything else (by mere stipulation), then scientific representation-based computation would be in danger of collapsing into (interpretational) pancomputationalism. So, without answering 'Which account of scientific representation should we use to portray computational implementation?' the development of SRA remains unfinished.

I will set out to change this shortcoming in the course of this chapter. To do so, I will acquaint us with the DEKI account in the next section. This brief familiarization with the DEKI will be paramount for an improved SRA called Implementation-as.

4.3 Scientific Representation, Representation-as, & DEKI

4.3.1 From Art to Science

Despite the seeming simplicity of scientific representation's underlying idea, precisely defining it is a contentious matter. One successful problem-solving strategy has been to seek answers in the study of art and languages. A case in point is the notion of *representation-as*, introduced by Nelson Goodman and Catherine Elgin (Goodman 1976; Elgin 1983). According to their theory of symbols, there are three fundamental 'modes of reference': (i) representation-of; (ii) Z-representation; and (iii) representation-as. This tripartite distinction stems from the observation that many representations represent an object as something else. A common pictorial example is caricatures. Take for instance the depiction of Winston Churchill as a bulldog. Letting '*X*' stand for the representing thing (a caricature); '*Y*' for the thing represented (Winston Churchill); '*Z*' stands for the kind of representation (a bulldog). The caricature features all the relevant distinctions of representation at once. First, the caricature is a representation-of Churchill, because it denotes the former English Prime minister. Secondly, the caricature is also a *Z*-representation, where here '*Z*=bulldog' since it exemplifies the features of a bulldog. Thirdly, the caricature represents Churchill as a bulldog, because the bulldog features (such as being stubborn or resilient) are imputed to him. In the remainder of the chapter, such XYZ-triplets with their corresponding notions of denotation, exemplification and imputation will be chief for understanding the notions of representation-as and implementation-as, respectively.

Subsequently, philosophers such as Hughes (1997), Elgin (2010, 2017), and van Fraassen (2008) appropriated the representation-as conception to the scientific realm. In what follows, I introduce what arguably is the most sophisticated of such accounts: Frigg and Nguyen's DEKI account.

4.3.2 The DEKI Account

In a recent number of publications, Frigg & Nguyen (2017 2018, 2020, 2021) introduced their so-called DEKI account, providing a full-fledged and systematized account of scientific representation based on representation-as. DEKI applies both to material models and non-concrete models.[90] I follow suit with the authors to discuss the account based on a material model – the Philips-

[90] Importantly, the account has been developed independently of the Problem of Implementation.

Newlyn machine (also known as *MONIAC*).[91,92] Standing about 2m tall, more than 1m wide and almost 1m deep, the device comprises several see-through plastic tanks and tubes filled with colored water. Attached to the tanks are pulleys, sluices, gauges, and pens (used to plot graphs). The design of the machine uses pumps and gravity to let water accumulate in different reservoirs containing floats that drive the different components in the mechanism depending on the water level.

Qua scientific model, the purpose of the machine is to model a national economy by the circular flow of water – the flow of the water stands for the exchange of commodities. Each of the machine's tanks corresponds to different features of an economy (national income, governmental spending, etc.). Depending on the configuration of the mechanical components of the MONIAC, different amounts of water accumulate in the different tanks, allowing the modeling of various economic scenarios. Fig. 4.1 shows a simplified scheme of these components and how they enable the device to work in connection with the notion of representation-as.[93]

Frigg and Nguyen suggest formalizing these considerations through Elgin's and Goodman's analysis of representation in the art world. In case of the MONIAC, they explain that

> "[…] the idea behind the machine is that hydraulic concepts are made to correspond to economic concepts. This means that we turn system of pipes and reservoirs into an economy-representation by interpreting certain selected X-features as Z-features. The water in a certain reservoir is interpreted as money being saved; the level of water in the reservoir is interpreted as a quantity of money; and so on." (Frigg & Nguyen 2020a, 166)

Since denotation, exemplification, and imputation constitute the core of representation-as, they also find application in their full-fledged account of scientific representation. To be informative in the scientific arena though, a fourth element – the notion of a 'key' – is introduced. Keys are meant to adjust model features to target features, because typically model features can rarely be

[91] The name MONIAC (standing for 'Monetary National Income Analog Computer') is more common in the US, where the coinage of the term was due to economist Abba Lerner "to suggest money, the ENIAC, and something mechanical." ('The Moniac' 1952, 101).

[92] Multiple authors have provided technical descriptions of the machine, its underlying economic theory, and its history (see e.g., Phillips 1950; Newlyn 1950; Barr 1988; Bissel 2007; Morgan 2012, 172-216).

[93] N.b., there is a difference when applying the XYZ-triplet to models like the MONIAC as opposed to caricatures. Whereas the latter can rather straightforwardly be identified as, e.g., a bulldog-representation, it is much less obvious how the MONIAC's water-filled pipes and tanks are supposedly an economy-representation. The problem is that the machine does not instantiate actual economic features. For the sake of modeling, scientists hence need to *translate* the flow of water into the 'flow' of commodities under an agreed-upon interpretation.

transferred unaltered to a target (e.g., one may need a scale factor or a conversion of units). Together these four salient features form the acronym DEKI. In sum, the following picture emerges:

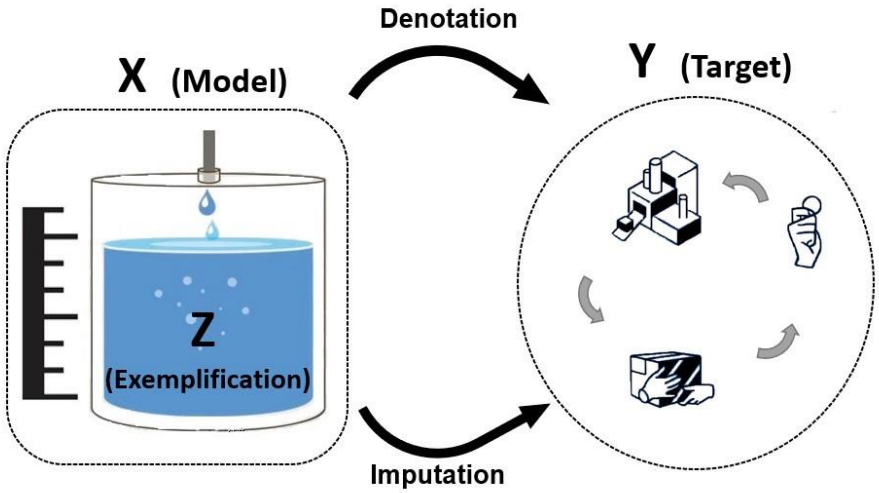

Fig. 4.1: Schematic depiction of the MONIAC at work, representing an economy through the flow of water. Applying the XYZ-triplet and the corresponding notions of denotation, exemplification, and imputation to a scientific context results in the notion of representation-as. Roughly put, X takes on the role of the (material) model (e.g., a tank filled with water); Y takes on the role of the target (e.g., an economy); Z takes on the role of the exemplified features of the representing object.

DEKI-account

A model is defined as an ordered pair $M = \langle X, I \rangle$, where X is an object and I is an interpretation. I is what turns a selected object X into a model. M represents Y as such and so *iff* conditions (1)-(4) are met:

(1) An interpreted object X (the model M), like the MONIAC, *denotes* a target Y (e.g., the British economy).

(2) M *exemplifies* Z-features. For instance, to be an economy representation, the MONIAC needs to exemplify economy-features (Z-features). However, often scientific models do not directly exemplify the required Z-features. The MONIAC e.g., is nothing but a sophisticated collection of pipes and tanks filled with water; it only has such-and-such dimensions, weighs so and so many kg, has n-number of components, etc. It merely instantiates the flow of water; it does not realize economic features such as the exchange of commodities. To turn such a model's features into the required Z-features, we need to resort to the interpretational capacities of the designers and users. Only under a specific agreed-upon interpretation

I are the scientists licensed to translate features of their model into *Z*-features $I : X \rightarrow Z$.

(3) There is a *key K* that systematically *translates* the exemplified *Z*-features $\{Z_1,\ldots, Z_n\}$ of the model, into another set of *Y*-features (the features of the target). In the case of the MONIAC, units of volumes of water (that are interpreted as the flow of commodities) must be translated into units of a specific currency. Furthermore, the time of the machine operating must be translated into the time of economic cycles. Depending on the denoted target, a key may associate one liter of water with e.g., 1 million pounds or 5 million US dollars.

(4) *M imputes* at least one of the 'keyed-up' features to the target. If the users of the MONIAC are interested in say, only tax revenue, they might only impute one single feature (corresponding to tax revenue) to the target.

The result is an intentional conception of scientific representation, as all its features (1)-(4) require different interpretations in the form of intersubjective agreements of the scientists using them. Through the selection of an appropriate material system, target phenomena are represented as something else. The MONIAC for instance represents the flow of money as the flow of water.

What is interesting about the DEKI in combination with the MONIAC is that under different assumptions, the very same device may be regarded as a special purpose hydraulic analog *computer* instead of a scientific model. It thus serves as an ideal gateway for establishing a link between scientific representation and implementation.

4.4 From Science to Computing: Implementation-as

I will now transpose the DEKI framework to computational implementation. The goal is to create a clear understanding of physical computation, especially an SRA that utilizes a *concrete* concept of scientific representation. This results in the introduction of *implementation-as*. The successful transposition requires a careful adaptation of the original DEKI account to the computing domain. In the next four subsections, I will show how the adaptation from the scientific arena to computing plays out. The discussion unfolds along the most salient features of the DEKI account, viz., denotation, exemplification, keying-up, and imputation.

4.4.1 Denotation

Generally, we need to think of denotation as the dyadic relation of a name (or label) and a bearer it applies to. The relation is established by an interpretive act. Elgin, for instance, states that "[r]epresentation- of— that is, denotation— can be achieved by fiat. We simply stipulate: let x represent y and x thereby becomes a

representation of y." (Elgin 2017, 253). Whilst originally a linguistic concept, she argues that there is nothing intrinsic in the notion of denotation that would restrict it to language only. Both symbols and what they denote can be of many different types. Consequently, Goodman and Elgin both apply denotation to other instances:

> "Pictures, equations, graphs, charts, and maps represent their subjects by denoting them. They are representations of the things that they denote. [...] It is in this sense that *scientific models* represent their target systems: they denote them." Elgin (2010, 2; own italics)

In the scientific context, denotation is taken to establish a connection between a model *X* and its intended target *Y*. Put differently, denotation establishes which target is supposed to be represented. Now, I submit that denotation applies *mutatis mutandis* to physical computation.

At first, this may not strike one as surprising for denotation is also not an unfamiliar notion in computing. For instance, the notion of denotational semantics is paramount for computer scientists to formally determine the meanings of programming languages. Likewise, when following popular interpretations that computers are symbol manipulators, one may subscribe to the view that the manipulated symbol structures denote information, data, etc. In the literature of physical computation, the so-called *semantic accounts* turn such a reading into a philosophical approach: as Fodor (in)famously proclaimed, there is "no computation without representation." (Fodor 1981, 180). The slogan especially embraces the metaphysical assumptions underpinning those branches of cognitive science that maintain that the brain computes. Exemplary of the 'aboutness' of neural computation is Marr's hypothetical case of the apocryphal grandmother cell (a cell that fires only when one's grandmother is in sight) (Marr 2010, 15). Today, semantic accounts may come in vastly varying degrees of commitment to what kind of processing of representations is essential for computation. More recent versions, for instance, may share the most salient constraints of some of the EMAs (e.g., causal, counterfactual, or disposition) but call for the additional condition that computational states must be representational (see Shagrir (2020) for an overview).

However, implementation-as should not be characterized as just another semantic account. Importantly, when it comes to implementation-as the choice of the potentially denoted target is restricted to the to-be-implemented sequence of computations.[94] So, in contrast to Marr's example, denotation may not be used to establish a dyadic relation to one's grandmother or any other external events, etc.

[94] In what follows, I will use expressions such as 'sequence of computations', 'computational formalism', and 'program' interchangeably.

Here the notion is *exclusively* reserved for the relation between a material system and a computational formalism *P* which specifies a sequence of computations.

> **Denotation**: Establishing which computational formalism *P* is supposed to be implemented in the putative material computing system.

As such, one of the key features of denotation (as a stipulative act) is that it enables the programmers and users to *specify* which sequence of computations ought to be implemented. Without denotation, we were not able to determine which computational formalism or program *P* (instead of *Q*, *R*, *S*, …) is originally intended to be run by the material device. What's correct behavior in the execution of *P*, may count as malfunctioning (miscomputation) of *Q*. And without knowing what is supposed to be computed, we would be unable to judge correct implementations from faulty ones. A prominent case from the philosophical literature is captured by Kripke's remark about Wittgenstein's hypothetical *rule-following* machines:

> "How is it determined when a malfunction occurs? By reference to the program of the machine, as intended by its designer, not simply by reference to the machine itself. […] Whether a machine ever malfunctions and, if so, when, is not a property of the machine itself as physical object but is well defined only in terms of its program, as stipulated by its designer." (Kripke 1982, 34f)

Assigning a physical system or device to perform a certain task rather than another is not exclusively limited to computation but rather ubiquitous to technology. In computing specifically though, we then assign the teleological function to compute a specific mathematical/computational function *P* to a material system; denotation is chief for specifying which computational function *P* is supposed to be implemented. What makes function ascription (in the teleological sense) a special case when it comes to computing is that we exclusively assign the execution of a rule or *mathematical/computational* function to a system. When assigning teleological functions like brewing coffee to a coffee machine, driving screws into a wall to screwdrivers, etc., the assigned functions concern physical properties and activities (e.g., pouring hot water onto ground coffee) and not formal, mathematical, or computational ones. This raises the question, how can concrete material systems exemplify computation?

4.4.2 Exemplification

Objects and systems can exemplify all sorts of properties in multiple ways. For instance, caricatures can exemplify bulldogs by pictorial means. In everyday life, properly working coffee machines exemplify concrete mechanisms that enable them to brew coffee. In the scientific realm, models like the MONIAC require an interpretative element since the hydraulic device, on its own, cannot exemplify

economic properties without our intentions. I submit that the same scenario holds for computation: physical systems can exemplify computational properties by turning selected features into computational states through interpretational exemplification. Analogously to the modeling case, we can define interpretational exemplification as

> **Interpretational Exemplification**: I: $X \rightarrow Z_C$. Turning selected X-features into computational states Z_C through an interpretation.

Accordingly, this characterization enables physical systems, which would otherwise merely be mechanical, hydraulic, or electronic, to act as computing systems by turning selected features into computational states.

However, *prima facie* such unlimited interpretational freedom is problematic as almost any object could count as a computing system by mere stipulation, violating the extensional adequacy criterion. Therefore, pragmatical and theoretical factors constrain potential exemplification, particularly the selection of (i) suitable interpretations and (ii) adequate X-features. Combined, these factors will determine why we take some physical systems to be computational and others not.

The advantage of agential SRAs (in contrast to other interpretational theories of computational implementation) is that they can establish a more rigorous foundation for suitable interpretation by adhering to the theoretical insights of how scientists use interpretations when engaging in scientific modeling. The DEKI account, for instance, informs us

> "[w]hile one is initially free to choose [X]-properties and Z-properties freely, once a choice is made, representational content is constrained. […] Free choices, once made, are highly constraining. This is why models are epistemically useful." (Frigg & Nguyen 2018, 214)

Thus, it is necessary to *agree upon* and subsequently *hold fixed* the selected X-features and their interpretation as computational states under a one-to-one relation. Choosing a different set of states $X = \{X_1,\ldots, X_n\}$ requires a new interpretational process for every new candidate set of computational vehicles.

While agreement and fixation are necessary to employ objects like the MONIAC as a computational device, they do not suffice. Taking a rock or wall, arbitrarily picking out some of their properties as X-features, and holding these fixed, still does not turn them into useful computers. Here, additional pragmatic and conceptual considerations come into play.

Similarly to the case of scientific models, there are additional pragmatic factors that constrain the choice of X-features. One of these factors is successful use. When selecting physical states as computational vehicles for usage, we typically select programmable vehicles we can reliably configure according to

our desires. Put differently, we should be able to put the system into a specific initial state (from a set of potential input states) to compute the output of our chosen computational problem. Moreover, a particular physical device is useful to us as a computer only when its salient states are distinguishable by us with our measuring devices. Only when this condition is met can we extract the results implied by the computational formalism we are interested in. Other practical considerations may include the system's reliability in repeating computations.

Conceptually, it is essential interpret only those physical states or carriers (*X*-features) as computational vehicles that demonstrate a sufficient degree of counterfactual state transitions. This demand aligns with the literature on scientific representation and the overwhelming consensus of the various EMAs. Therefore, it is paramount for agents to select (and usually construct) potential computational vehicles that exhibit a reliable degree of counterfactual dependence. Such counterfactual support is chief for using scientific models for surrogate reasoning and turning computational devices into epistemically fruitful instruments. To better understand this, consider the following two quotes. Concerning scientific models, Bokulich for instance reminds us that

> "[…] in order for a model M to explain a given phenomenon P, we require that the counterfactual structure of M be isomorphic in the relevant respects to the counterfactual structure P. That is, the elements of the model can, in a very loose sense, be said to "reproduce" the relevant features of explanandum phenomenon." (Bokulich 2011, 39)

In the same vein, Piccinini provides a summary in his (2015, 19-25), showing that it is wide consensus that the microphysical state transitions of a material computing system require counterfactual support:

> "In other words, the pure *counterfactual account* requires the mapping between computational and microphysical descriptions to be such that the counterfactual relations between the microphysical states are isomorphic to the counterfactual relations between the computational states." (Piccinini 2015, 19)

What this means in the case of the MONIAC is that different calibrations of the knobs, valves and tanks filled with water need to bring out reliable changes in behavior. 'If the input/initial conditions had been different' the output must be different accordingly. Such counterfactual support is crucial for the implementation of a computational function. Only if the *X*-features are chosen in such a way that different set-ups yield different interpretable outputs can material models/computers such as the MONIAC be used to model target systems like an economy or a computational formalism.[95] Controlling these

[95] These computational states correspond to a model of computation; in the case of the MONIAC, the model of computation is characterized by a set of differential equations. Often,

counterfactual dependencies of computational devices is what enables to physically program these machines and use them to compute functions.

Together the conditions about interpretation (agreement and holding it fixed), and selection of *X*-features, where some are of pragmatic nature (programmability, distinguishability, reliability) and some of conceptual nature (counterfactual state transitions), are jointly sufficient to restrict those things that don't compute. (I will talk more about this in sect. 5).

4.4.3 Encoding a Labeling Scheme

To recap, while denotation specified which computational formalism is supposed to be implemented, interpretational-exemplification imposes which properties of a putative computing system are taken to be as computational states. So far, these two steps are insufficient for the implementation of computations, for we only determined that something may act as a computer (not what it actually computes). Scholars of physical computation widely agree though that one needs to specify the conditions that a computational system implements one computation rather than another (IDENT). Now, in order to relate exemplified computational states to a specific model of computation, we need to define for what kind of computations they are employed.

One crucial aspect for determining such a computational profile is to allude to the notion of a *key*. According to DEKI, exemplified properties are 'keyed up' with properties that are supposed to be imputed to the target. While the name 'keying-up' is inherited from the DEKI account, I suggest resorting to the more common terminology used in computing, where the discussion is usually framed under the label of *encoding* or fixing a *labeling scheme* (cf. Copeland (1996)).

> **Encoding a labeling scheme**: Relating the set of interpreted computational vehicles Z_C with a set $P=\{P_1,\ldots, P_j\}$ of states that are presumed to be imputed to the targeted computational formalism.

In what follows, I introduce the arguably two most relevant types of encodings for computing.[96] The two types roughly correspond to analog and digital computers respectively.[97]

---

the seminal paper by Pour-El (1974) is taken as the theoretical basis for models of analog computation. For a survey of such different models see Bournez & Pouley (2021).

[96] Whether the two types of keys are exhaustive or not, such that there might be other kinds of keys relevant for computing – for instance, in the case of quantum computing – is the subject of future research.

[97] In the context of computing, the digital/analog distinction is a vexed issue; simply put there are two major camps: According to one view, analog computation is understood as an *analogy* (the behavior of a damped spring-mass might be modeled by electronic components that analogously showcase similar behavior); according to the second view, the operation of an analog computer should be understood based on the manipulation of continuous values. An

The first type of encoding essentially hinges on the same idea as the keys employed in material (scale) models. Certain physical magnitudes are selected to scale with the chosen features of a target system. For example, Weisberg (2013) and Pincock (2022) discuss this in detail, based on the *San Francisco Bay-Delta model* and a scale model of *Lituya Bay* for modeling rockslide-generated tsunamis, respectively. However, in most cases, the selected *X*-features cannot be directly imputed to the target *Y*. In the case of the just mentioned scale models e.g., the key is not simply equivalent to the scale factor, as fluid dynamics don't scale completely proportional.[98] Similar keys are necessary for scaling in analog computers. Ulman, for instance, describes that machine units of a given analog machine must be adjusted to the denoted computational problem (cf. Ulman (2013, 55 and 123-14) and Ulman (2020, §2.1 and 58)).

Based on the work of Lewis (1971), Maley formalized this idea, developing the so-called *Maley-Lewis account* that's supposed to cover the case of analog computation. Simply put the Maley-Lewis account captures the idea of scaling, i.e., the more the representing physical magnitude Z_C increases or decreases (in a systematic way), the more the property that's denoted in- or decreases. These insights yield the formulation of the first type of encoding (cf. Maley (2011, 124)):

> **Type 1: Encoding (Scaling) by magnitude.** As Z increases (or decreases) by a margin *d*, Q increases as a *linear* function of X+*d* (or X-*d*); *E*:Z →*P*.

When it comes to the implementation of *digital computation* though, a digital labeling- scheme is needed. As Maley explains, numbers are typically represented by (i) a series of digits and (ii) a base.[99] A digit series is then interpreted as the relative value of the digits. Translating this idea into a digital version of a key, the second type of encoding is defined as:

> **Type 2: Encoding digitally (labeling scheme).** A digital encoding *E*: Z →*P* represents a number/symbol via its digits, where 'digit' means a symbol (typically a numeral) in a specific place. In addition, we require a base, which is used to interpret the relative value of digits.[100]

---

in-depth exorcism of the analog/digital distinction lies beyond the editorial scope of this chapter.

[98] And in the case of the MONIAC, we don't even have a scale model of a Keynesian economy at all, but an object where certain features are selected (X-features) such that their covariation tells us something about the denoted target Y. Remember, physical quantities like 'flow of water' must be related to 'flow of money' via a system of units.

[99] By understanding 'numbers' in a loose sense, the method can be applied to symbols that are part of an alphabet.

[100] Formally, the digital representation of a number '$d_n\ d_{n-1}$ ... $d_1\ d_0$' is captured by the formula $(d_n \times b^n) + (d_{n-1} \times b^{n-1}) + \cdots + (d_1 \times b^1) + (d_0 \times b^0)$. In base 10 "sixty-five" e.g., is hence represented as "65" (6*10)+(5*1).

Having elucidated how to determine a computational profile, implementation-as requires a final step.

4.4.4 Imputation

Lastly, *imputation* is the final necessary component of the implementation-as framework. As a first stab, "imputation can be analyzed in terms of property ascription", (Frigg and Ngyuen 2018, 217). Let me briefly return to the scientific modeling context for the sake of clarifying what kind of properties are ascribed to what. When scientists use a scientific model to reason about a target system, they must be able to ascribe features of the former to the latter $f: M \rightarrow T$. Put differently, we may thus say that the model imputes features to the target. The MONIAC, a material model, imputes its exemplified (under an interpretation) economic features to the dedicated target. I propose to appropriate this practice to computing, such that *material* systems implement a computational formalism (the analog to the target) by relying on imputation.

The reason why we appropriate imputation from representation-as to computing is that we want to systematically relate the interpreted and encoded computational vehicles of a material system to the denoted computational formalism (cf. steps (1)-(3)). As such, imputation has a comparable function to the mathematical notion of a morphism (relating physical states and abstract computational states) evoked by the EMA.

> **Imputation**: Ascribing encoded computational states to a computational formalism.

But what are the ramifications of referring to the relation as an 'imputation' instead of a mapping? The philosophically relevant message is that the mapping is *stipulated by human agents*: As an agential theory of implementation, implementation-as relies on a, at least partly, mind-dependent notion of computation – we use devices as an aid for our computational goals which otherwise would need to be carried out by hand or in one's head. Imputation can be understood as the notion that relates the interpreted and encoded computational vehicles of the surrogate system we use for computation with the computational problem we wish to be solved. Implementation-as advocates for a stipulated implementation-relation. Such a relation has two principal advantages.

First, the advantage of a stipulated implementation relation is that it does not stand at odds with the state-of-the-art insights of applied mathematics (see Rescorla (2014, §4) for a similar remark in the context of computing). Called the *application*- or *bridging problem*, philosophers of applied mathematics seek to

address the notorious issue of how the mathematical relates (or bridges) to the physical. In a nutshell, the problem is that mere morphisms between physical states and mathematical/states do not obtain, because strictly speaking functions only obtain between set-theoretic structures (and physical substrates do not offer such a unique structure (Psillos 2006, van Fraassen 2008)). In response, most recently suggested solutions to the bridging problem state that the mappings between the physical and mathematical are mind-dependent (Pincock 2004, Batterman 2010, Bueno & Colyvan 2011, Nguyen & Frigg 2021). Put differently, at least some stipulations of agents are needed to *create* a structure and hence bridge the gap between abstract mathematical objects and concrete physical states.

Now, in so far as theories of implementation need to spell out how logico-mathematical models of computation relate to the physical, the problem of implementation is a special instance of the application/bridging problem (see Appendix B for a detailed discussion). Therefore, if not specified otherwise, accounts of physical computation should preferably be in line with the insights of the philosophy of applied mathematics. Imputation (a mind-dependent notion) is explicitly compatible with this demand. Accordingly, computational vehicles are associated with the logico-mathematical states of the implemented computational formalism.[101]

The second advantage and essential feature of imputation is that it bears a normative component – the pairing of exemplified features with features of the computational formalism can be right or wrong, hence explaining miscomputation. What's right is determined by the denoted computational formalism. Again, mere morphisms seem to fail the miscomputation-desideratum.[102] While the denotation-relation constitutes what is supposed to be implemented, imputation is the relation that pairs exemplified computational states and formal computational states (of the target). Only when imputation matches *all* the elements of the physical computational states required for a series of computations, then the denoted program *P* might be implemented correctly. Strictly speaking, if there is a mismatch, the system may compute in a way it should not; it is said to miscompute.[103]

---

[101] The argument may pose a problem for naturalized or mind-independent theories of implementation. The seriousness of this threat may be subject to future research.

[102] For essentially the same argument against using morphisms in accounts of scientific representation see (Suárez 2003).

[103] There are various ways in which this computational norm can be broken. Fresco & Primiero (2013), offer a detailed taxonomy of the miscomputation of *software*, stating that miscomputation can occur at any level of abstraction, ranging from faulty specifications, through the algorithmic level, down to the machine. At the abstract physical interface, errors might be due to wear and tear or insufficient counterfactual support (Schweizer 2019, 38-40).

4.4.5 Taking Stock

Subsuming the various elements appropriated from the scientific representation discourse results in an explicitly spelled out agential SRA:

Implementation-as

Let the ordered pair $C=\langle X, I\rangle$ be a computational device, where X is a material system and I an interpretation. Let P be the computational formalism/program. C implements P as Z_C *iff* all the following conditions are satisfied:

(1) C denotes P.

(2) C exemplifies Z-properties $Z_1,\ldots,Z_n$ under an interpretation $I: X \rightarrow Z_C$.

(3) C comes with a computational encoding associating the set $\{Z_1,\ldots, Z_n\}$ with a (possibly identical) set of properties $\{P_1,\ldots, P_m\}$. $E\{Z_i\}=\{P_j\}$

(4) C imputes at least one of the properties $P_1,\ldots, P_m$ to P.

The resulting framework is baptized implementation-as, acknowledging the influence of representation-as from the philosophy of art and science. This approach is methodologically different from previous accounts of physical computation couched in generic terms of scientific representation like *L-machines* or *A/R-theory*, because it builds on a specific scientific representation proposal. Concretely, the framework deploys Frigg and Nguyen's DEKI account of scientific representation (of material models).

The takeaway is that analogously to how a material scientific model (based on an interpreted object X) is used to represent a target system *Y* as thus-and-so, the core idea of implementation-as is to use a physical system (based on an interpreted object X) to implement a series of computations or a program P. Simply put, the computational formalism is the target that is supposed to be implemented. Both scientific representation and physical implementation are instances of object-based reasoning. In the former case, we manipulate and interpret a material model as a surrogate to reason about a target system/phenomenon. Concerning the latter, we configure and alter (i.e., program) a physical computing system to obtain the result of a computational function. As such, almost the entire DEKI-analysis of the MONIAC *qua* scientific model equally applies to the machine when interpreted as an analog computer.

Having cashed out the main features of implementation-as, the remainder of the chapter demonstrates how implementation-as applies to a case study (sect. 4) before philosophically evaluating the novel theory (sect. 5) and concluding with a discussion on how it relates to existing accounts (sect. 6).

4.5 Case Study: The IAS-machine

In this section, I will show how the concept of implementation-as is not just a theoretical discussion or limited to an analog hydraulic computer but is also applicable to a widely known and influential device called the *IAS-machine*.[104] This machine embodies the architectural principles of the *von Neumann architecture*, which is still commonly used today. First, I will portray the components of the machine and how it was programmed in detail (section 4.1). Next, I will demonstrate how implementation-as sheds light on how the machine implements physical computation (section 4.2). As we will see, the application of the implementation-as framework is relatively straightforward despite its, at first, seemingly heavy formalism.

4.5.1 Technicalities and Programming

The IAS-machine was one of the first binary stored-program computers, storing instructions and data in the same memory. For enabling these features, different components need to act as different computational states. The designers relied on vacuum tubes for the circuitry and Williams tubes (cathode ray tubes) for the memory. These components then formed three basic units: [105]

1. The main memory unit (M)
2. The Central Processing Unit (CPU): Containing Control-Unit (CU) and Arithmetic-Logic Unit (ALU)
3. The Input/Output device (I/O)

Considering the functioning of these units and their underlying components in detail further clarifies our understanding of how exemplification and encoding work in the case of a stored program digital machine. So, let me briefly look at each of these units in detail, starting with the memory.

[104]Many authors have provided technical descriptions of the IAS-machine, how it was programmed, and its history (Burcks et al. 1946; Estrin 1952; Ware 1953; Bigelow 1980; Burcks 1980; Aspray 1990; Priestley 2018). It is a stored-program digital computer that was constructed over the course of six years by a team of scientists and engineers under the leadership of von Neumann at Princeton's Institute of Advanced Studies (IAS). The machine was completed in 1952 and had a significant influence on future generations of computers in and outside of the industry, both in the US and overseas e.g., ILLIAC, MANIAC (in Los Alamos), and the IBM 701 (Aspray 1990, 86-94).

[105] These elements (or "main organs") were mentioned in different forms by von Neumann (1945), where they were called CA (central arithmetical), CC (central control), M (memory), I and O (input and output devices) and R (some external recording medium).

The memory was of 'Williams type' and composed of 40 standard commercial "off the shelf" (Bigelow 1980, 302) 5CP1A cathode ray tubes (relying on the emission properties of cathode-ray-tube phosphor screens). It had 1024 storage locations or memory addresses, called words. Each word is 40 bits long and may contain (1) a number word or (2) an instruction word (see Fig. 2).

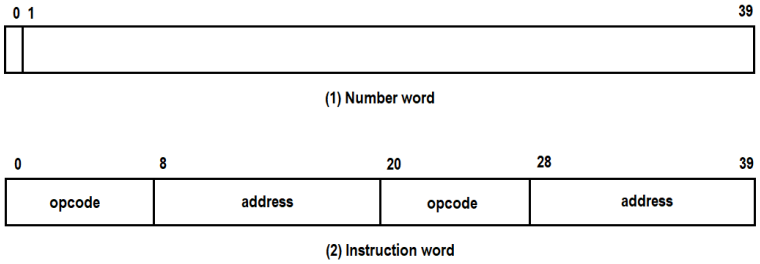

Fig. 4.2. Depiction of the two different types of words.

Instructions occupied two times 20 bits, where the first eight bits are opcode and the remaining twelve bits indicate the address of a register. Overall, the instruction set of the IAS machine contained 21 different instructions (Burks et al 1946, 42). A line of code of a program written for the machine then may look like this 00000001000111110100000001010001111101 01. The first eight bits (grey font) are opcode and correspond to the instruction "[c]lear accumulator and add number located at position x in the Selectron into it." (Load M(x_i)); the following twelve bits correspond to a memory address x; the next eight bits (grey font) are opcode and correspond to the instruction "[a]dd number located at position x in the Selectrons into the Accumulator" (Add M(x_j)). [106] It is sequences of bits like these, composed of the machines' specific instruction set that may comprise a program *P*.[107] As we will see, the reason why these details are relevant for the application of implementation-as is that they warrant multiple, distinct instances of interpretational exemplification.

Concerning the second main component, the CPU, the IAS-machine has seven different registers (Accumulator, Arithmetic Register, Control Counter, Control Register, Function Table Register, Memory Address Register, and Selectron Register) of which only the Accumulator and the Arithmetic register are 'visible' to the programmer (both holding 40 bits).[108] These registers utilized about 1700

[106] For a more elaborate and detailed example see for instance Priestley (2018).
[107] In the same vein, modern microprocessors are too compatible with specific ISAs (Instruction set architecture), like x86, where "[t]he ISA serves as the boundary between the software and hardware.", (Hennessy & Patterson 2012, 11).
[108] In today's terminology the 'Control Counter' is known as Program Counter and has a 12-bit width; the Control Register holds the instruction currently executing (20-bit width). The

to 2300 commercially available miniature double triodes, [109] where most of them where of type 6J6 (other models used where 5670, 5687, and a few 6AL5 scattering diodes). Like modern garden variety CPUs, it executes instructions of programs, such as arithmetic (e.g., adding integers of above's example program *P*), I/O operations and logic controlling.

Lastly, the selected I/O components are an important element to consider. They afford the interface through which the users can interact and program the device. Without input mechanisms like punched cards, teletypewriters or keyboards, programmers and users had virtually no reliable means to load instructions or data into memory. In the same vein, the lack of an output medium (e.g., some kind of screen) would render the computational system a black box. It is these outputs however that ultimately need to be in tune with the denoted computational formalism/program *P*. At first, the engineers of the machine relied on perforated teletype tape which in late 1951 was replaced by IBM punched cards (Bigelow 1980, 306).

What turned the IAS-machine into a digital one is that it was operated under a digital encoding. This design choice both appealed to the intended logical nature of the machine ('being a yes-no system') and facilitated the use of existing electronic components (flip-flops), such that

> "[o]ur fundamental unit of memory is naturally adapted to the binary system since we don't attempt to measure gradations of charge at a particular point in the Selectron but are content to distinguish two states. The flip-flop again is truly a binary device." (Burks et al. 1946, 7).

In addition, the composition, or architecture, constituted by the three interconnected units M, CPU, and I/O enabled the IAS-machine to store instructions (and data) in memory. As such, the machine stands in contrast to early digital machines like ENIAC or analog devices like the MONIAC that had to be reprogrammed manually similar to plugboards or read instructions from external tape.

4.5.2 Implementation-as at work

Equipped with some basic understanding of the inner workings of the IAS-machine and how it was programmed, let me sketch how the most salient features of implementation-as come to fruition. As explained throughout the chapter, the core notion of implementation-as is that properties of the designated

---

Function Table Register holds the current opcode and is 8 bits wide, whereas the Memory Address Register holds the current memory address and is 12 bits wide.

[109] The precise number of triodes used in the machine diverge among different authors. Whilst Estrin (1952) mentions 2300 triodes, Ware (1953) speaks of ca. 1700, and Bigelow (1980) mentions about 2000.

computational vehicle are associated with the abstract computational states of a computational formalism $\{P_1, \ldots, P_m\}$ through a set of exemplified computational states $\{Z_{1, \ldots,} Z_n\}$. To implement a specific sequence of computations, the putative computing system needs to undergo four steps: denotation, interpretational exemplification, encoding, and imputation.

Here, we assume that the IAS-machine is our *X*, i.e., our vehicle of computation. As discussed in the previous section, our *X* is composed of many different components (e.g., cables, 6J6 triodes, …), forming three interconnected units (M, CPU, and I/O). As such, it can be considered a computing system under a series of fine-tuned interpretations *I* of some agent (typically the user of an epistemic community who share the same conventions regarding a device). Specifically, the IAS-machine then implements a computational formalism/ program *P iff* the following four steps apply:

(1) First, the device *X* denotes *P*. In the case of the IAS-machine, a typical program *P* will look like a list of machine-code instructions each of 40-bit length as just introduced in the previous section. As such, *P* acts as the normative yardstick to evaluate executions between correct and faulty ones (miscomputation). To eventually implement *P* correctly, different components of the IAS-machine need to relate to different sections of the code.

(2) Second, given our agreed upon interpretation *I*, we note that the IAS-machine exemplifies certain computational features $\{Z_{1, \ldots,} Z_n\}$. According to the general scheme outlined above, exemplification hinges on our interpretational capacities $I : X \rightarrow Z$. For instance, the previous discussion of the technicalities of the IAS-machine showed that the following components play different roles in exemplifying computational features: 5CP1A cathode ray tubes are employed for holding data and instructions in memory; the CPU (with its seven registers) relies on miniature triodes (mostly of type 6J6); the I/O used punched cards to program the machine in order code.

(3) Third, one needs to choose an encoding or labeling scheme. Since the IAS-machine was constructed as a binary digital computer, parting with the "longstanding tradition of building digital machines in the decimal system" (Burks et al. 1946, 7), it operates as a binary digital computer processing both digital data and instructions in a binary format. Accordingly, we adopt a binary digital encoding as described in sect. 3.3. Standardly, one then associates the absence (considering a certain threshold) of the flow of charge as '0' and the flow of charge as '1'.

(4) Finally, the just encoded computational states $\{P_1, \ldots, P_m\}$ are imputed to our 'targeted' program *P*. Since computer scientists, programmers and users

usually opt for the correct implementation of computational artifacts, we ideally require that the entire set $\{P_1, \ldots, P_m\}$ is related to P.

To wrap up, the IAS-machine implements computations as the flow of charge. The straightforward and successful application of implementation-as to the IAS-machine suggests that this new agential theory of implementation can be applied to other computers as well. Despite significant technical differences, many modern computing machines still incorporate the basic architectural design choices of this influential device. I believe that it is sufficiently complex and bears enough similarities to the functioning of contemporary computers. Although new technological advancements may lead to greater complexity, there is no reason why implementation-as cannot be applied to these cases.

4.6 Is Implementation-as a good theory of computation?

At last, let me briefly evaluate the in this article developed theory of implementation. The discussion proceeds along the lines of the desiderata of physical computation introduced in the introduction (Sect. 1). As I will show, implementation-as accommodates all the desiderata and should therefore be considered a viable theory of physical computation.

(1) Objectivity. Nowadays, philosophers of science commonly agree that there are considerable obstacles to cashing out theories of scientific representation in naturalistic terms. That is why most approaches are formulated as intentional conceptions (Frigg & Nguyen 2020a, 2020b). The DEKI account is a case in point, for all its salient features hinge on scientists' interpretational capacities. As discussed at length, implementation-as inherited many of the key features – and accordingly, it may be called an agential theory of implementation. Now, does relying on interpretational features undermine the objectivity of implementation-as?

The answer is nuanced. Reiss & Sprenger (2020) survey various conceptions of scientific objectivity – as stated by Fletcher (2018) and Duwell (2021), theories of physical computation based on agential notions of scientific representation may only undermine an overly rigid notion of objectivity. Since implementation-as appeals to agents and their stipulations, it may be incompatible with what Duwell refers to as *strong objectivity* (i.e., an account of objectivity according to whether a system is representational/computational is completely mind-independent). However, relying on agential notions of scientific representation does not undermine *weak objectivity* (Duwell 2021, 19). Accordingly, scientists may reach intersubjective agreements if an object counts as a scientific model, which parts of the world it is presumed to represent, and so on. Once such

intersubjective agreements are held fixed, practitioners may engage in scientific reasoning without their personal preferences or any substantial personal biases.

Implementation-as adheres to standards of objectivity in these latter, less rigid terms. Once the combined stipulative elements of denotation, interpretational-exemplification, encoding, and imputation are agreed upon and held fixed, computation under the regime of implementation-as is as objective as the scientific practice of modeling and free of personal arbitrary beliefs, desires, and intentions.

(2) Extensional Adequacy. A good theory of physical computation should properly systematize paradigmatic computing systems (laptops, calculators, smartphones) as computational; it should also judge instances of non-computing systems as non-computational. The examples of the MONIAC and the successful application to the IAS-machine show that implementation-as does not have trouble classifying paradigmatic examples of computing systems as computational. What works in the case of the IAS-machine, generalizes to other real-world machines. In so far as the physical system exemplifies computational properties that are keyed-up/encoded and imputed to states of a computational formalism (which is denoted by the system), the system may implement the formalism as such and so.

However, saying which systems do not compute proves more challenging. The main concern is that without any restrictions on interpretation, any object could be trivially turned into a computer by stipulative fiat. However, the Implementation-as framework avoids this problem because it is a hybrid account that relies both on interpretational aspects and mind-independent physical features. In other words, it characterizes a computing system as $C=\langle X, I\rangle$, where X represents the physical features, and I represents the interpretation. As a result, cases where neither condition applies $\langle 0, 0\rangle$ are non-computational, as well as cases where we bestow objects with an interpretation but where they lack adequate physical features $\langle 0, 1\rangle$ (such as Putnam's rock or Searle's wall). Additionally, systems with sufficient counterfactual support $\langle 1, 0\rangle$ but lacking one or more of the four implementation-as features (denotation, interpretational exemplification, encoding, imputation) are dismissed as being computational.[110] According to implementation-as, physical computation only occurs when both these elements come to pass simultaneously $\langle 1, 1\rangle$. When visualizing the different scenarios in a graph, the following picture emerges (Fig. 4.3):

[110] One may call such kinds of systems 'quasi-computational,' because in many cases, it is still desirable to analyze them in terms of computational.

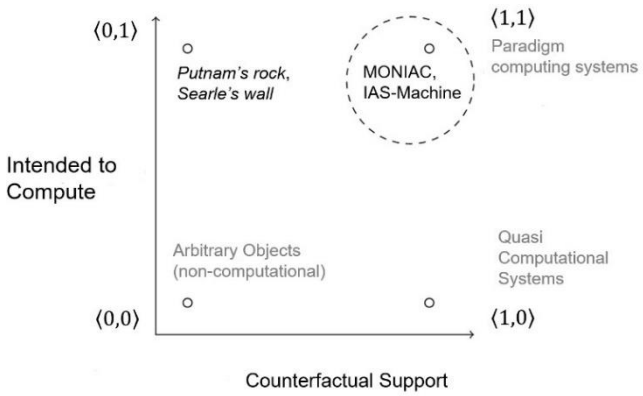

Fig. 4.3: The 'hybrid approach' (having to rely on both interpretation and the right degree of counterfactual support) ensures that implementation-as considers that the right things compute and the wrong things don't. Figure inspired by a similar graphic in Artiga (2023) in a different context (teleological functions).

As can be seen, the extensional scope of the implementation-as framework is smaller compared to those accounts that merely rely on counterfactual state transitions (e.g., Chalmers 1996, Scheutz 1999). While such accounts need to bite the bullet of limited pancomputationalism (because every counterfactual/causal structure would compute some function), [111] my account does not suffer from this defect because interpretational exemplification further limits what may count as a computational vehicle. As previously explained, interpretations need to be agreed upon and held fixed. Additionally, the selection of X-features needs to follow practical considerations (programmability, distinguishability, reliability) and rely on counterfactual state transitions.

(3) Explanation. According to the third desideratum, a good account of concrete computation should be able to explain (at least some of) a system's capacities computationally. There are different ways to understand this requirement. On the one hand, the computational properties of a system may be explained by what it implements. For instance, the IAS-machine implementing our exemplary program *P* explains why it adds integers the way it does, its efficiency, etc. Yet, on the other hand, under implementation-as material systems may only exemplify computational states if agents bestow them with the task to do so – without the agent's stipulations, the chosen vehicles are not computational. Does

[111] This sort of pancomputationalism is limited, since, although it states that every object computes, it does deny the much stronger thesis that these objects also implement every possible computation.

this mean that computational explanations then merely reduce to agents' desires to use something as a computer? No, because as I have argued implementation-as is a 'hybrid'-account – the agents also need to choose suitable physical states that may act as computational vehicles. That's why the current framework must additionally resort to the particular underlying scientific theories that describe the behavior of the chosen vehicles. As such, the explanations offered by implementation-as are no longer distinctively computational but may be physical, chemical, or biological (cf. Duwell 2021, 37). In the case of the MONIAC e.g., the flow of water is taken as a computational vehicle. To explain the behavior of the machine, we must consult hydrodynamics and the scientific theories describing the dynamics of the mechanical components.

(4) Miscomputation. One of the main advantages of interpretational accounts of computation is the straightforward explanation of judging the (in)correctness of a computational process. Unlike their naturalized counterparts, they do not have to deal with the difficult issue of natural teleology. Instead, maintain that agents bestow the computing system with teleological functions to compute. Therefore, the philosophy of computer science borrowed some of the function ascription frameworks from the philosophy of technology (Turner 2018, Anderson 2019). As a result, an interpretational account of computation implementation-as can accommodate *different* notions of miscomputation. Let me briefly discuss these notions separately.

First, programmers and users may sometimes have disagreements about which program is supposed to be implemented. Although it seems like an easily avoidable mistake, denotation is crucial for determining (in)correct computational implementation. This is because figuring out the precise (teleological) function of a computing system is epistemically inaccessible and cannot just be read off. Prominent computer scientist Weizenbaum brought up this up in a thought experiment, stating that if one day in the distant future, a highly advanced society would find one of our present-day computers, they could never know with certainty to have gotten the alleged program P just right (Weizenbaum 1976, 132ff.). Albeit a high degree of understanding might be achievable through observing its output patterns, black-box testing and attempts of reverse engineering, reclaiming absolute certainty of the computer's specification might be impossible. Likewise, Dennett (1990) comes to a similar conclusion with a real-world example of a discovered 'computer' – the Antikythera mechanism. The (teleological and mathematical) function of the ancient Greek device was initially obscure, and today scholarship is still puzzled by it.

Secondly, miscomputation may be caused by faulty imputations. As argued above, faulty imputations may occur either through wear and tear or because of insufficient counterfactual support. Both conditions lead to a mismatch between the different execution traces of the denoted computational formalism M_C and the putative computing system.

(5) Taxonomy. Encoding a labeling scheme is crucial for determining for what kind of computations a system may be used for. I described the encodings corresponding to the arguably two most widespread instances of computing – digital and analog. Accordingly, the encodings of the interpreted computational vehicles enable us to discern two major kinds of computing systems and their different capacities.

Furthermore, implementation-as does not need to allude to the 'narrow' notion of program execution only. When judging various accounts of physical computation, Piccinini criticized some earlier approaches that would equate physical computation with program execution, because this may raise trouble for classifying systems that are said to compute by means other than running programs. [112] Implementation-as does not need to appeal to the notion of program execution in order to be applied successfully; nothing in its four salient features hinges on program execution. Rather, whether a system can be classified to compute by virtue of program execution depends on the denoted computational formalism (and arguably on one's definition of what a program is).

In sum, the results of this brief evaluation showed that implementation-as squares well with most of the desiderata. As such, it has the potential to apply to two, traditionally separately discussed approaches of computational implementation. On the one hand, it may retroactively apply to some of the previously mentioned SRAs, specifically those that suggest a formulation in agential terms (Fletcher 2018, Anderson 2019, Papayannopoulos 2020, Szangolies 2020, and Duwell 2021). On the other hand, the framework offers a modest interpretational account of physical computation. Implementation-as is a modest account because it draws on both interpretational and non-interpretational features (e.g., requiring counterfactual state transitions). In contrast to previous somewhat arbitrary interpretational accounts (according to which everything can be regarded as computing through mere interpretation), the account presented here clarifies how interpretational and non-interpretational features connect abstract and physical computational states.

[112] For instance, he argues that some neural networks compute by means other than program execution (Piccinini2008). Another (potential) case in point is analog computers, where some scholars believe that they compute despite not executing a program.

4.7 Discussion and Concluding Remarks

In recent years, transposing insights from the scientific representation discourse into physical computation has resulted in a fruitful new perspective on computational implementation: so-called SRAs. I have contributed to this trend by providing a new rigorous description of a theory of implementation called Implementation-as. What distinguishes my approach from previous SRAs is that it relies on a concrete version of scientific representation rather than a general one – the DEKI account of material models. The resulting account is a hybrid approach because it depends on the users' stipulative abilities and the physical characteristics of the material system. The upshot is that implementation only occurs when agents use the carefully chosen material object to model a desired abstract computational formalism. In particular, agents may use a material computing system as a computing device if they engage with denotation, exemplification, encoding, and imputation. Combined, these four activities portray the commonalities of physical computation as suggested by implementation-as.

Importantly, my analysis showed that this new agential/interpretational SRA makes the grade with many of the standardly evoked desiderata. For these reasons, I submit that implementation-as is a promising alternative to existing accounts of physical computation. To conclude I will briefly put my undertaking into perspective, commenting on how my approach contrasts to prominent existing accounts, in particular extended mapping accounts (EMA), semantic accounts, and mechanistic accounts. But for reasons of editorial scope, I cannot offer an in-depth comparison to the entire spectrum of currently available approaches and refer the interested reader to full-fledged surveys.

It is widely accepted that simple mapping accounts (SMA) trivialize computation due to unlimited pancomputationalism. This defect led to the development of more sophisticated approaches: Counterfactual, causal, and dispositional accounts extended the mapping account with restrictive conditions that prevent too liberal mappings. In section 3.2, I discussed that implementation-as also requires us to select suitable computational vehicles that can sufficiently exhibit counterfactual state transitions.

Although implementation-as and 'traditional' EMAs share these similarities, there is an essential difference between the two. Traditional EMAs assume that the implementation relation is a naturalistic/mind-independently obtaining two-place relation between physical and abstract computational states. In contrast, SRAs generally advocate for an interpretation of the mapping by virtue of scientific representation. This commitment is crucially different because many available options of scientific representation are three-place relations obtaining

iff we allow agents and their intentional capacities into the picture. That is why many SRA-proponents argued that they should be conceived as an agential theory of implementation. The implementation-as framework makes this explicit, and I argue that its successful application requires the mind-dependent activities of denotation, exemplification, encoding, and imputation.

The semantic account further restricts EMAs by requiring that computational states always carry meaning or semantic content. In a previous section (3.1), I discussed the connection between my approach and semantic accounts. Both SRAs and semantic accounts emphasize the importance of representation in computation. However, there are notable differences how representation is used and understood in both frameworks.

In the implementation-as framework, scientific representation is utilized to bridge the gap between abstract computational states and physical states without the need to commit to external content. In general, SRAs only have a minimal requirement for content: physical states merely need to be the bearer of logico-mathematical content (of the implemented model of computation). Any additional semantic content or meaning the computational vehicles have, is irrelevant to the successful application of SRAs and hence implementation-as. (However, the user of the computing device may assign semantic content or meaning to computations if needed).[113] In contrast, semantic accounts use representation in a broader sense, where computational states may represent external states of affairs (for example, grandmothers, when thinking of Marr's apocryphal grandmother cell). This sense of representation is more relevant to cognitive science, which assumes that brain states are representational.

Regarding the relationship between Implementation-as and mechanistic accounts, the question about their link is nuanced. Depending on which mechanistic version one choses for comparison, there are different degrees of shared commitments. Generally, mechanistic accounts state that functional mechanisms are central to computing; computational vehicles need to be components of a mechanism. In its current formulation, the implementation-as framework does not specifically share that commitment. However, even though, computational vehicles need not be part of a mechanism for the successful application of implementation-as, nothing in the formulation of my account rules out that computing systems $C=\langle X, I\rangle$ cannot be mechanisms. In fact, both previously discussed cases – the MONIAC and the IAS-machine – are *bona fide* mechanisms. Future research should elucidate if this fact is accidental or whether

[113] See Fletcher 2018, 452-53 for a similar discussion concerning AR-Theory.

a combination of the views might lead to an even more robust theory of physical computation.

5 Physical Programmability

In line with the UTAI framework (cf. Chapter 2), the previous chapter introduced a novel theory of computational implementation. As such, Implementation-as illuminated how different features under the umbrella 'program' are connected across the abstract-concrete dichotomy. This chapter sheds further light on the physical side of things and scrutinizes the relationship between agents and the physical system they program (dependency relation (c)).

Introduction & Motivation. How are programs integrated into the causal nexus? What does it mean for a physical system to be programmable? Which of a program's features let it appear as physical entity? The strategy of conceiving computation abstractly at a level of symbol manipulation and programs as sets of instructions fails to account for the *physical* properties that render a system programmable. For addressing these issues this chapter introduces the notion of *physical programmability*. Physical programmability accommodates insights from well-established research territories like (computational) mechanisms; interventionism; human-machine interaction; theoretical computer science, and is compatible with real-world examples. I propose that the ensuing characterization of physical programmability

> **Physical Programmability:** *The degree to which the selected operations of an automaton can be reconfigured in a controlled way.*

Subsequently, the structure of this chapter unfolds like so: First (section §5.1), I provide a critical overview of a handful of existing accounts concerned with the programmability of sequence executing systems. Then, I introduce the novel notion of physical programmability by presenting the various elements/variables contained in its definition. Accordingly, I begin with elucidating the conception of material automata (section §5.2). Thereafter (section §5.3), I shed light on which kinds of operations are permissible to fall under the scope of programmability by relying on the notion of mechanisms. Next (section §5.4), I explain how 'interventionism' allows us to understand the reconfiguration of programmable systems. Penultimately (section §5.5), I analyze to what extent programmability comes in different degrees. Lastly (section §5.6), I provide some concluding remarks and discuss various open questions.

5.1 A critical overview of Programmability

One way of thinking about programmability is in terms of the ability to change the behavior of a sequence-operating system. We may program our VHS, washing machine, or computer. Pre-theoretically, programmability is commonly viewed as 'the property of being programmable' and applied to a wide variety of different kinds of either virtual or concrete (computing) systems. However, a characterization along these lines is circular and lacks rigor. Without further analysis, such a definition of programmability remains uninformative at best.

Despite the importance of the notion (for computing), the literature on the programmability of physical systems is scarce. To the best of my knowledge, there are only a handful of sources that explicitly aim to elucidate the matter. To get started, I will, therefore, conduct a thorough and critical assessment of the proposals by Conrad, Zenil, Piccinini, and Haigh & Priestley. This in-depth analysis will provide a comprehensive understanding of the existing literature. As we will see, while differing significantly, these accounts coincidentally[114] share four salient features of programmability: A specification of the type of system programmability applies to (*scope*); the kind of *operations* it can perform; *the way in which* re-programming is achieved; and a *grading system* according to which programmability comes in different degrees. These four ingredients (cf. table 1 at the end of this section) will subsequently serve as the kick off for my refined physical programmability proposal.

5.1.1 Programmability as a trade-off principle

The formulation of a more rigorous notion of programmability was initially attempted in *The Price of Programmability* (Conrad 1988). Conrad, a biophysicist who studied biological computing systems, posits a trade-off principle that links computing and evolution. He attributed three fundamental properties to these systems: programmability, efficiency, and evolutionary adaptability. Inspired by contemporary computing technology and the linguistic metaphor associated with programming languages, Conrad characterizes programmability as the "ability to prescriptively communicate a program to an actual system" (Conrad 1988, 286). By subsequent refinement, Conrad offers two more fine-grained notions (that he deemed crucial for understanding the adaption of biological systems): *effective programmability and structural programmability*.

Relying on the anthropomorphic notion of communication, a system is effectively programmable "[…] if it is possible to *communicate* programs in an exact manner, using a finite set of primitive operations and symbols, without approximation." (Conrad 1988, 287-88; own emphasis). Conrad contends that

[114]In fact, none of the illuminated approaches in this section cross-reference each other.

effective programmability is achievable in three ways: (a) by relying on an interpreter, (b) by building a physical realization of the relied-upon model of computation, and (c) by utilizing a compiler. Structural programmability builds on effective programmability and bears the additional condition that "a program is mapped by its structure" (ibid., 288). Put differently, some systems may be effectively programmable but not structurally programmable. Conrad states that only the first two ways, (a) using an interpreter or (b) building a physical realization, suffice for structural programmability. The implication is that the corresponding program of such merely effectively programmable systems would not be mapped by their structure.

Despite capturing some intuitive notion of programmability, I have three principal reservations about Conrad's account: First, the definition of (effective/structural) programmability fails to explain what kinds of manipulations are permissible. Relying on the communication metaphor for human-computer interaction obfuscates which physical properties are essential for the device's programmability.

Second, in the build-up of his argument, Conrad appears to rely on a dubious understanding of the *Church-Turing thesis*, confusing computational modeling with physical computation proper. [115] Since Conrad's original publication in the late 1980s, a rich literature on physical computation emerged, emphasizing that computational modeling and concrete computation must not be confused, or else one slips into trivial forms of pancomputationalism. However, it is only due to this conflation that Conrad can apply programmability and his trade-off system to all sorts of (biological) systems.

Third, the distinction between effective and structural programmability remains somewhat opaque when it comes to the "mapped by its structure" condition. According to the literature on physical computation, implementation requires some mapping between the formal model of computation and physical substratum (see Piccinini & Maley (2021) for an overview). Effective programmability contradicts these research insights without clarifying what the 'mapped by its structure' condition is supposed to amount to.

5.1.2 Programmability as the foundation of computation

Independently of the previous account, Zenil introduced an approach to programmability closely entangled with computation (Zenil 2010; Zenil 2012; Zenil 2013; Zenil 2014; Zenil 2015). His view about programmability emerged from his so-called *behavioural standpoint* – an approach to physical computation claimed to part ways with the common approach of the so-called simple mapping

[115] Conrad's version is much closer to what Copeland (2024) has called the 'Maximality Thesis.'

accounts. So, instead of relying on a mapping between an abstract model of computation and a physical computing system, a compression-based metric is advanced that acts as a 'grading system' of a material object's ability to be (re-)programmed. The ability to be programmed is regarded as a necessary condition for physical computation.

Accordingly, physical computation *cannot* be separated from programmability, where the latter is defined as "[…] the ability of a system to change, to react to external stimuli (input) in order to alter its behaviour." (Zenil 2015, 112). In fact, programmability is regarded as a condition *sine qua non* – "[h]ence, we make the assumption that central to the claim that something computes is the capability of a system to be reprogrammed." (Zenil 2014, 111). Only when a material system is said to be reprogrammable can it be considered computing; the ability of physical computation reduces to programmability.

By connecting programmability to a general notion of 'variability' (defined by a formal measure),[116] Zenil aims to provide a theoretical basis to quantify the degree of change due to some external input. Simply put, the variability measure assigns values to outcomes (states of a system) depending on different initial conditions.

While offering advantages in terms of a formal programmability measure, Zenil's account also presents certain concerns. Most notably his work does not tell us how to differentiate genuine programming from other (arbitrary) interactions with a material system. Without additional constraints, the account may be trivialized, as any physical interaction could potentially be considered programming. Although a normative condition is presented, indicating that the system should behave as intended, it remains unclear whose intentions should be the deciding factor.

5.1.3 Soft & Hard Programmability

A third attempt to define programmability emerged from Piccinini's work about the teleo-mechanistic account of computation (Piccinini 2008, Piccinini 2015). He writes "[a]ny machine that can be easily modified to yield different output pattern may be called 'programmable'." (Piccinini 2015, 184). Interestingly, his notion does not exclusively apply to computing systems. Opposed to Conrad and Zenil, his conception may apply to *non-computing* mechanisms as long as they operate in sequence (such as weaving looms and juke boxes). Nevertheless, Piccinini maintains that programmability is a gradual concept (though a non-formalized one). He suggests framing it in phenomenological terms, i.e., the

[116] Zenil employs *Kolmogorov Complexity* (aka Algorithmic Information) as the basis for his formal variability measure.

easier one obtains different output patterns, the higher the amount of programmability. Based on these preliminaries, his work offers a taxonomy of four different cases of programmability:

(a): The first type of programmability corresponds to the configuration of non-computing systems just presented. Piccinini does not pay much attention to this type of programmability and I will hence skip its assessment for lack of analyzable material.

(b): More attention is paid to the second type though: *hard programmability*. Hard programmability refers to computing systems (n.b., non-computing systems are no longer discussed) where components are *mechanically* modified: In order to implement a specific computational function *f(i)*, with input *i*, the machine's operators need to adjust the pattern in which the computing components are "spatially joined together" (Piccinini 2015, 185).

(c) & (d): When systems are not (re)programmed mechanically, they are considered *soft programmable*. Here, the "modification involves the supply of appropriately arranged digits (instructions) to the relevant components of the machine" (Piccinini 2015, 185). Two cases are distinguished, leading to his third and fourth types of programmability, respectively: On the one hand, *external* soft programmability is defined by the insertion of the instructions (encoded in a string of digits) through an external medium (for instance punched cards). On the other hand, *internal* soft programmability applies to devices that can store programs (strings of digits) inside of them. The distinction between external and internal soft programmability thus alludes to architectural features: external soft programmability refers to devices without internal memory, internal soft programmability applies to devices with internal storage.

As already mentioned earlier, it is paramount to emphasize that his taxonomy is *not* categorized by which kind or how many (computational) functions can in principle be implemented in a system though. Instead, Piccinini's grading system is based on the way in which a (computing) system is manipulated. Choosing to spell out programmability in terms of *how* a device is set up, is supposed to capture some of our everyday experiences of programming devices. Either machines are programmed through cumbersome mechanical modification (hard programmability), or by supplying instructions to computing components ex- (external soft programmability) or internally (internal soft programmability).

While considering everyday experiences from a programmer's perspective is a welcome feature, at a closer look, the categorization scheme raises at least two worries. First, the distinction between hard- and soft programmability is somewhat reminiscent of the notorious software-hardware dichotomy. The

software-hardware distinction is ill-defined.[117] I think that Piccinini's distinction is subject to similar criticism, i.e., hard and soft programmability appears to be relative to some arbitrarily drawn line, since ultimately every programmable system is changed 'mechanically' (at least, if understood in terms of mechanism; cf. sect. §3).

Moreover, the formulation 'to supply instructions' is potentially misleading. Such communication metaphors attribute cognitive capacities like 'understanding instructions' to (computing) machines. Anthropomorphizing machines risks overlooking the underlying physical properties and mechanisms that allow for programmability in the first place.

5.1.4 Program Execution ≠ Programmability

Lastly, Haigh & Priestley (2018) recently developed a notion of programmability for historical discussion meant to classify *COLOSSUS* with respect to other well-studied historical digital computing machines.[118] What distinguishes their conception from the previous ones is that it does not appeal to computability theory or any other formal apparatus. Instead, the authors state that two conditions are necessary for a system to be programmable, viz.,

> "[…] "programmability" as applied to a device requires not only that the device carries out a sequence of distinct operations over time, i.e. that it follows a program, but also that it allows a given user to define new sequences of operations." (Haigh & Priestley 2018, 18)

Their two necessary conditions are intertwined: On the one hand, a programmable device needs the capacity to carry out a sequence of operations; on the other hand, the sequence of operations must *in principle* be changeable by the users. The authors rightly emphasize that the latter feature is dependent on the former, as a system must be able to execute sequences of operations to enable users to change them. As such, their notion of programmability is rather inclusive – also allowing non-computing devices such as programmable washing machines. Importantly, this definition allows COLOSSUS to be classified as a program-executing device, despite being non-programmable in their terms.

[117] Remember, for instance (Moor 1978), according to which the software-hardware distinction is merely a pragmatic one, dependent on context and the skills of the programmers and users.
[118] COLOSSUS was a British top-secret electronic codebreaking device built from 1943-1945. Haigh & Priestley argue that the machine was not built to carry out numerical computations but designed to decrypt teleprinter encryption of German communication during WWII. Despite not being a (general-purpose) computer, the authors claim that the machine automatically executed a program (i.e., implemented a specified series of discrete operations). Notwithstanding, Haigh and Priestly state that COLOSSUS was not programmable since the users could not fundamentally alter the program of operations performed by the machine.

Developed for a historical argument of only one particular device, their characterization of programmability has a couple of weaknesses when applied to other automata. Chief among them is that their approach does not elucidate different degrees of programmability – according to their binary view a system is either (completely) programmable or not. In addition, the authors stay silent about which kinds of interactions ought to be considered as re-programming (as opposed to arbitrary interactions).

5.1.5 Taking Stock

This brief overview showed a small and disconnected variety of philosophically inclined attempts towards programmability of material (computing) systems. The results are summarized in Table 5.1.

	Conrad	**Zenil**	**Piccinini**	**Haigh & Priestley**
Type of Material System	Natural & technical	Natural & technical	Technical & natural (?)	Technical
Operations	Computation	Computation	Sequenceable operations	Sequenceable operations
Mode of Reconfiguration	Instruction	-	Mechanical, Instructions	-
Degree/ Grading System	Effective and Structural	Quantitative measure (algorithmic information)	Hard and soft	-

Table 5.1: Overview/Comparison of the different features of the here presented accounts of programmability.

Given that programmability is often overlooked in philosophical discourse, its individual methods, scope, and aims can vary greatly. Nonetheless, some commonalities have emerged in the accounts of programmability. Therefore, I take it that a good account of programmability should specify the type of system to which it applies, the operations that are considered, how configurations are achieved, and in what sense it is a gradual notion.

With these requirements in mind, I submit an improved notion called physical programmability: *The degree to which the selected operations of an automaton can be reconfigured in a controlled way*. In the following sections, I will clarify how the elements in my characterization of physical programmability - automaton,

selected operations, reconfigured in a controlled way, and the degree to which it is achieved - are anchored in contemporary research traditions.

5.2 Material Automaton

Virtually any system's behavior can be changed or manipulated in one way or another, but not every change of arbitrary objects amounts to (re)programming. For this reason, it is desirable to constrain programmability to specific systems only. In this section, I explain how the *material automaton* variable contained in my definition serves this purpose by restricting the scope of physical programmability to real-world automata. In what follows, it is thus crucial to specify what real-world automata are.

The term 'automaton,' originating from the Greek word *αὐτόματα*, means 'self-moving'. Historically speaking, material automata have existed since ancient times and include mechanical clocks, automated musical instruments, looms, and calculators (Ambrosetti 2010). They can perform operations like sound production, weaving, or physical computation, based on varying degrees of energy and control autonomy. (A more elaborate concrete example will be discussed at the end of the section). Today, one may characterize an automaton as

> **Automaton:** System with the ability to execute a predetermined series of operations (to some degree) autonomously.

Despite this precise characterization, the term 'automaton' bears some ambiguity in common language and philosophical discourse. Depending on one's understanding of 'system,' the term 'automaton' may refer to two different ontological domains. On the one hand, modern automata theory is the study of *abstract machines* and an integral part of theoretical computer science.[119] On the other hand, there are those already mentioned tangible real-world automata – these systems are particulars locatable in space-time.

Differentiating between abstract and concrete automata is crucial for avoiding category mistakes. For instance, as per Sloman (2002), material automata display energy autonomy and control autonomy. A device that depends on a human operator to provide energy (e.g., by turning a knob) has low energy autonomy, whereas a device with an integrated energy source, like an engine or battery, has

[119] For instance, by defining different classes of abstract computing systems such as finite state machines, pushdown automata, Turing machines, etc. we can study the theoretical limits of computation (cf. Hopcraft et al 2001). A Turing machine, e.g., provides a formal procedure for computing a function, yet the machine qua abstract object is not something physical at all. Often programmability is discussed with these formal devices; Turing machines, for instance, are said to have a higher programmability than FSM, as they compute more functions.

high energy autonomy. Similarly, a machine that necessitates frequent user intervention to control its actions has low control autonomy (e.g., a car), while a system where a predetermined set of actions can be executed without any intervention on the control mechanism has high control autonomy. Now, applying categories like energy and control autonomy to logico-mathematical entities like Finite State Machines would be a fallacy because these abstract formalisms cannot be driven by real-world motors. Importantly, physical programmability is hence only intended for *material* automata.

5.2.1 Automata as technical artifacts

To further distinguish material automata from ordinary physical objects like rocks, tables, and tigers, it is helpful to rely on the theoretical framework of technical artifacts. Technical artifacts are special types of artifacts that are characterized by their 'dual nature' - constituted by both mind-dependent functional features and mind-independent structural features (cf. Baker 2006; Kroes & Meijers 2006; Kroes 2012; Preston 2018, §2.3). Structure determines what an artifact can do, while function is what the artifact is intended to be used for. Due to this normativity, some researchers (Vermaas & Houkes 2003; Houkes & Vermaas 2010) argued that technical functions require intentionality. Accordingly, an agent or epistemic community intentionally ascribes a function to an object for a specific purpose.

From this theoretical standpoint, material automata can be viewed as technical artifacts because they are (i) intentionally created devices with (ii) the ability to execute a predetermined sequence of operations. Let me briefly look at these requirements separately.

The first necessary characteristic to be considered a material automaton is that the system's structure must be able to exhibit sequential behavior (e.g., through a mechanism). However, this is a cheap property that many systems possess: Given some interpretative flexibility, a wide range of systems appear to act in sequence – the dynamical macroscopic behavior of systems like hurricanes or rivers, for instance.[120] That is why mere sequential behavior is insufficient to qualify as an automaton qua technical artifact.

Therefore, we must adhere to the second fundamental trait of technical artifacts - intentional function ascription. In the case of a designed program

[120] Worse, one may even argue that *prima facie* seemingly static systems (like rocks and tables) have an ability to operate in sequence. In a different context, philosophers like Putnam (1988) and Searle (1990) have employed such reasoning to argue that objects like rocks and walls, seen at a *microscopic* level, showcase an internal dynamical behavior (that is interpretable as a sequence of operations). The reason for this is that the physical state of ordinary systems does in fact traverse physical state space and is not completely static.

executing material automaton, one of the system's function is to execute a predetermined series of manipulable operations. It is important to note that the 'predetermined' clause requires an intentional sequence set up by an epistemic agent or community.[121] Put differently, this requirement excludes natural systems that can act in sequence as material automata because their course of action is not *intentionally* predetermined by designers, programmers, or users. Although we may *describe* dynamic systems like hurricanes and cells in terms of theoretical automata, they should not be considered material automata defined by technical artifacts.

5.2.2 An Example: The Musa flute player

So far, we primarily approached the topic of material automata theoretically. To provide further clarification for the rest of the chapter, I discuss the relevant concepts with a concrete, historical example: an ancient flute player. Albeit seemingly simple at first, I shall occasionally return to this example to discuss several philosophical issues relevant to the remainder of this chapter.

One of the first audio automata or 'music boxes' that play melodies with minimal human intervention originated in the ninth century CE, when three scholarly brothers from Baghdad, known as the Banu Musa, built an automatic flute player (Levaux 2017, §3.1). The device was powered by water and operated using differences in air and hydraulic pressure, generated by a filled reservoir. This structure generated wind for the creation of the sounds of the flute. By additionally utilizing a cylindric rotating drum with teeth and small levers that opened or closed the flute's nine holes depending on the size and positioning of the raised pins, different melodies emerged (Koetsier 2001, 590-591). Fig. 1 illustrates the underlying mechanism responsible for the energy autonomy that enables the automatic functioning of the device.

[121] Ascribing teleological functions to arbitrary systems (with the ability to act in sequence) is insufficient to turn it into a technical artifact. Mere function ascription leaves room for ready-made artifacts or so-called *object trouvés* (meaning found objects – a concept from the art world). If that were the case, one could simply promote natural objects, which can be utilized to serve human purposes, into technical artifacts. A simple example is a rock that may be used as a hammer. Similarly, one could turn systems like hurricanes or cells into a material automaton by interpreting their dynamical behavior sequentially.

Importantly, the Musa flute player's designers did likely not intend their machines to be reprogrammable after construction (d'Udekem 2013, 177). Once operational, the system was designed to perform a fixed melody (a sequence of tones) and was not responsive to any external input.[122] The lack of a programming mechanism or external interface made it impossible to control and predetermine a (new) series of operations unless the machine was disassembled or destroyed.[123]

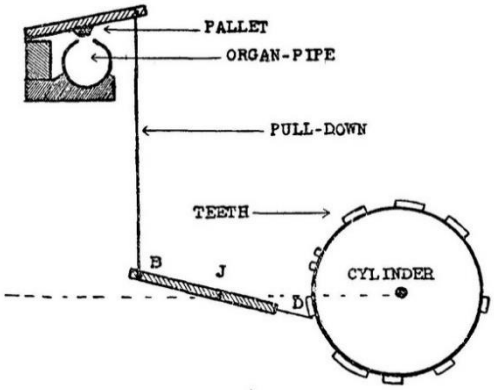

FIG. 9.—THE BANŪ MŪSĀ AUTOMATIC HYDRAULIC ORGAN. (Reconstructed.)

Showing the way in which the teeth of the cylinder or recording barrel opened the holes of the horizontal organ-pipe by means of pallets.

Fig. 5.1: Depiction of the mechanism that constitutes the Musa flute player. The teeth of the cylinder or 'program barrel' opened the organ-pipe via levers through pallets. (Image taken from Farmer (1931, 101)).

The upshot of discussing this peculiar device is that not all sequence-controlled automata are also programmable. To account for (more) programmable types of automata, I will provide further conceptual resources that elucidate how humans can intervene and control more complex devices. Particularly, I will shed light on the notions of input, output, and interface, facilitating our understanding of how human agents can *intervene* on the device's control structure/*mechanism* such that its operations change. Accordingly, the

[122] As such, 'programming' (in a limited and basic sense) may only take place during the construction phase of the device. The reason is that the mechanism responsible for producing the flute player's melody is internal to the system and completely hidden from its users. Since the mechanism is not meant to be changeable, there is no need for external means of regulation through an interface. Without a recognizable interface, re-programming is unfeasible.

[123] As Simon (1996, 6) points out, designers may only ever achieve a 'quasi-independence' of their technologies from the outside world. Biologists may have similar discussions concerning the phenomenon of homeostasis of certain kinds of organisms (Glennan 2017,114-115). No item can be entirely shielded from environmental influences, and the insulation of the flute player's inner workings may break down due to strong vibrations, extreme temperatures, or exposure to strong magnetic fields. Additionally, a skilled individual might be able to work around the insulation and 'hack' into the system and access the control mechanism of the machine, revealing unforeseen (non-intended) interfaces.

following section introduces how an automaton's operations hinge on its underlying mechanism.

5.3 Selected operation

In the 17th century, the term 'mechanism' surfaced by following Greek and Latin terms of *machine* (Dijksterhuis 1956). In recent philosophical discourse, mechanisms gained considerable traction with the so-called (neo-) mechanistic turn around the beginning of the millennium.[124] Since then, mechanistic talk in philosophy of science mushroomed and has brought forward a rich literature applied to large variety of research domains like physics, chemistry, biology, cognitive science, economics to only name a few.

I propose that programmable automata and the structure that enable their operations are fruitfully describable in terms of mechanisms. In particular, the mechanistic framework provides insights that deepen our understanding of 'operation' and how human agents may exercise control over a programmable system. In this section, I explain how we can understand operations by virtue of mechanisms. I will discuss the exercise of control afterward in sect. §4.

At first, marrying mechanisms with automata may seem hardly original – the very term 'mechanism' derives from 'machine' and has a technological connotation. Yet, using the philosophical notion of mechanism to analyze engineered systems is surprisingly scarce (van Eck 2017); arguably, most mechanistic research applies to the life sciences. This is a pity, because the mechanistic framework not only enables us to look at the physical components responsible for the automaton's *operation* but also allows for the integration of further conceptual tools from the mechanistic literature that can act as a philosophically robust bedrock for programmability.

5.3.1 Mechanisms

While I want to focus on mechanisms in programmable automata, it is chief that we first grasp the most salient features of mechanisms in general.[125] In what follows, I will rely on the stabilized 'consensus conception' of Illari and Williamson (2012), according to which a mechanism is defined as

[124] For a more thorough (but still tractable) historical overview of the mechanistic turn see Kästner (2017, Ch. 3).

[125] There is a wide array of systematic work about the nature of mechanisms. Some of the most influential accounts brought forward are Bechtel & Richardson (1993); Glennan (1996); Machamer et al. (2000); Bechtel & Abrahamsen (2005); and Craver (2007b).

> **Mechanism:** "A mechanism for a phenomenon consists of entities and activities in such a way that they are responsible for the phenomenon.", (Illari & Williamson, 120).

It is widely accepted that the spatial, temporal, and active relations between entities and activities (the micro-behaviors) are responsible for the mechanism's phenomenon (the macro-behavior). Characterizations along these lines appear to imply some form of mechanistic hierarchy: There are at least two *levels* comprised of acting entities (the parts) on the one hand, and an exhibited higher-level phenomenon (the whole) on the other. Typically, the higher-level phenomenon of some mechanism/system is referred to as *S*'s Ψ-ing, where *S* denotes the system, and Ψ-ing its corresponding phenomenon. The mechanism's entities are referred to as X_i and their activities are denoted by $\{\phi_1, \phi_2, ..., \phi_n\}$ (cf. Craver 2007b). Figure 2 pictures two mechanistic levels with the aforementioned elements.

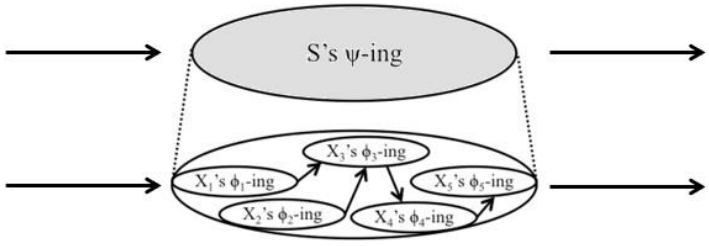

Fig.5.2: Schematic representation of a mechanism. (cf. Krickel 2018; Craver 2007b)

Such a two-level image can be (and is in fact often) expanded into a multi-level hierarchy. Every working entity X_i of a mechanism *S* can itself be subject to further decomposition into a sub-mechanism; mechanisms can thus be 'nested' several times. When then 'horizontally' analyzing the relation of the components X_i's on a given level we are speaking of *interlevel* relationships; when referring to 'vertical' relations between levels we speak of *intralevel* relations.

The mechanistic level image advocates a localized approach to levels of organization – the hierarchy applies *relative* to a given mechanism *S* and its lower-level components. While the level metaphor is ubiquitous in (the philosophy of) science, it is important to not conflate mechanistic levels with the other prominent level views (Craver 2014; Eronen 2015).[126]

[126] Particularly in the current context of computing, the conception of mechanistic levels does not equate with LoA of computational artifacts. Though one certainly can apply the methodology of LoA to mechanistic levels, there is one important difference: the mechanistic framework is limited to spatio-temporal entities only. In contrast, the notion of LoA may also be

What's key about the mechanistic framework for our quest to define physical programmability is that offers the right means to uncover the functioning of material automata. Specifically, I submit that *S*'s Ψ-ing ought to be interpreted as the automaton's (sequential) operations; here *S* corresponds to the automaton and Ψ-ing denotes its operations.

5.3.2 Input-Output mechanisms

An additional benefit of utilizing the neo-mechanistic framework for characterizing physical programmability is that we can rely on the notion of what Glennan calls *input-output mechanisms* (2017, 113-116; referred to as 'I/O' from here on). I/O mechanisms are a subclass of the generic definition of mechanisms. The focus is shifted to a phenomenal description, especially to the patterns a mechanism's phenomenon produces. As per Glennan, I/O mechanisms are systems whose actions or outputs are responsive to inputs and describable by a functional relation between input and output variables

$$f(i)=o,$$

where i denotes the input(s), o the output(s), and f their functional relation. Reasoning along these lines allows for the threefold distinction between mechanisms that bring about outputs as a result of inputs (the 'regular' I/O case); mechanisms that produce outputs independently of inputs (no-input/output);[127] and mechanisms that remain stable/provide a constant output when presented with varying inputs (input/no-output) (Glennan 2017, 116).

Moreover, by conceptualizing the inputs and outputs of the I/O mechanism as variables that can take on different values, we can easily use mathematical representations to describe *S*'s Ψ-ing. Using a mathematical representation has two primary advantages:

First, it allows us to flash out the possibility space of a material automaton's behavior in terms of (finite) automata theory.[128] Especially the notion of finite deterministic automata (FDA) and the corresponding state diagrams are useful models to study the potential material automaton's execution traces.[129] It is

applied to abstract/formal entities. Another crucial difference between LoA and the mechanistic hierarchy is the intralevel relation between different levels. Whereas the former relies on some form of leaving out selected details (abstraction), the mechanistic intralevel relations are of a different nature. I shall return to the importance of levels in section §4.

[127] The Musa flute player is a case in point.

[128] It is important to note that while we should be cautious not to confuse abstract automata of the logico-mathematical realm with concrete real-world machines, we can still use the conceptual framework of automata theory to model actual material devices.

[129] Theoretically, a FDA can be defined as a five tuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q denotes a finite set of states, Σ *is the finite set of input symbols,* δ is a transition function, q_0 is the start state, and F a set of final states (Hopcroft et al. 2001, 46). Depending on the input label a from the

handy for systems that execute a (predetermined) sequence of operations based on external events (inputs): Vending machines, elevators, traffic lights, and combination locks are typical examples of the textbook literature (see e.g., Mozgovoy 2010, 92-95). We may additionally draw state diagrams or contingency trees that visualize different execution traces depending on different inputs. As such, the concepts associated with automata theory allow us to analyze the behavior of material automata, predict their behavior, and reason about which series of operations are, in principle, executable.

Secondly, as we will delve into in the next section (§4), the mathematical representation of I/O mechanisms plays a crucial role in making the concept of 'reconfiguration' understandable through causal modeling. This is possible because that we can assign logical (Boolean), discrete, or continuous values to the variables associated with a mechanism, and since the functional form of relations between them can be characterized as linear, quadratic, logarithmic, etc.

5.3.3 Selection

I argued that the operations of an automaton are characterizable as a 'phenomenon' in terms of the mechanistic framework. However, without further qualification, the threat of an underdetermination problem remains with this view since a given system S may showcase multiple phenomena at once. When we judge an item to bear a certain degree of programmability, we typically do so with only one specific phenomenon (Ψ-ing) in mind. Physical programmability only makes sense relative to a specifically selected series of operations -- yet some systems may simultaneously exhibit multiple potential phenomena.

To exemplify the issue, consider the example of the flute player I previously discussed in section (§2). The takeaway was that the device has virtually no programmability since one cannot modify its sequence of operations in a controlled manner. However, during our assessment, we glossed over the fact that the ancient automaton simultaneously produces several phenomena (e.g., vibrations, sound, heat, etc.). Admittedly, most of these phenomena are just an accidental byproduct. Nevertheless, the numerous different phenomena require a specification or selection of a specific phenomenon, or else the notion of programmability remains underdetermined (i.e., the same system/device/automaton may bear (different degrees of) programmability concerning more than one type of phenomenon).

---

alphabet Σ, transitions $\delta(q, a) \rightarrow p$ connect the states (e.g., q and p). Multiple transition labels may form a 'word' $w = a_1, a_2, \ldots a_n$, (i.e., a string over the alphabet Σ). A word is valid for a given FDA if the sequence of transition labels leads from the initial state q_0 to a final one contained in F. A string of inputs w that is compatible with the FDA can be interpreted as a program describing an execution trace within the set of possible behaviors.

To remedy the issue, I added the 'selection'-clause in my characterization of physical programmability. The idea of the selection clause is to guides/inform us in the selection process of the material automaton's operations and single out a specific phenomenon, dependent on the interest of an individual or an epistemic community. N.b., as such, the selection-clause works hand in hand with the idea to restrict the applicability of physical programmability to designed material automata only. What is particularly helpful in this regard is my previous description that material automata are technical artifacts (cf. sect. §2.1). Due to their function-structure duality, technical artifacts bear specifically ascribed normative functions. In the case of the Musa flute player, for instance, its function is to produce a pre-determined sequence of sounds. In other words, the phenomenon physical programmability is supposed to capture usually coincides with the intended operation the material automaton should carry out.

In the ensuing paragraphs, I will illuminate how these selected operations can be altered in a controlled way.

5.4 Reconfigured

In the previous sections, I occasionally helped myself to the terms 'manipulation' or 'intervention' only using these terms informally. It is high time to discuss these concepts in more detail. Specifically, I submit that the main theoretical underpinnings of physical programmability are so-called manipulability- or agency theories of causation, which are a subset of causal interventionism. (cf. Woodward (2023) for a survey of manipulability theories). In a nutshell, manipulability theories aim to elucidate causal structures through

> **Difference-making**: *C* is a cause of *E* (the effect) *iff* manipulating *C* in the right way affects (makes a difference on) *E*.

The motivation to rely on such theories is threefold: First, utilizing a manipulability-based approach allows us to straightforwardly account for how programmers and users exercise control over a (computing) system through (causal) interaction/manipulation:

> "When a relationship is invariant under at least some interventions, it is potentially *usable for purposes of manipulation and control* —potentially usable in the sense that while it may not as a matter of fact be possible to carry out an intervention on X it is nonetheless true that if an intervention on X were to occur, this would be a way of manipulating or controlling the value of Y." (Woodward 2002, S370; own emphasis).

As such, it is paramount to note that manipulability theories do not need to employ any communication metaphor (as is arguably often the case in the context of programming computers through instructions). That way, we can study

human interaction with a programmable automaton without necessarily having to appeal to programming languages or anthropomorphic metaphors that distract us from what is happening during reconfiguration at the physical level.

Second, manipulability and agency theories typically rely on counterfactual reasoning (if *C* would have been different, *E* would have been so-and-so). This feature is advantageous because it allows for applying physical programmability to physical computation. As we have seen in the previous chapters on physical computation (Chapter 4 and Chapter 5), counterfactual support is essential for implementing computations and determining which computations would have occurred if the input had been different (cf. Piccinini 2015, Ch. 2).

Lastly, interventionism applies to various systems, including – crucially for this undertaking – mechanisms. This compatibility allows to integrate physical programmability in contemporary philosophical debates, facilitating the exchange of ideas and fostering cross-fertilization. Put differently, the advantage of the approach is that it enables the bringing together of diverse philosophical concepts under the umbrella of physical programmability.

To better understand how interventionism's features figure in programmability, it is helpful to familiarize us with the details of its formal machinery (sect. §4.1). This will be important to understand how the combination of mechanisms with the interventionist account play out in the context of the so-called *Mutual Manipulability* (MM) concept (sect. §4.2).

5.4.1 The Formal Machinery of Interventionism

Interventionism in its contemporary form (see, e.g., Woodward (2003) and Pearl 2009)) originated from combining features from causal modeling and manipulability theories. This theory's main achievement was to devise a formal notion of 'intervention' that does not require human agency.[130] Based on so-called structural models, causal relations (in science) can be precisely represented through a rigorous formal framework, providing us with criteria to analyze specific situations/systems to e.g., draw causal inferences without adhering to human terms. Accordingly, we can portray causal relations either by directed acyclic graphs (DAG) or structural equations.

[130]While manipulability theories capture the intuition of how to portray causal structure, earlier versions of manipulability theories were long objected to for relying on the anthropocentric notion of 'manipulation.' Depicting causes C as vehicles for manipulating effects E, often (at least in older versions) assigns central significance to human action. Adhering to human agency was seen to fly in the face of the idea that causal relations are part of the mind-independent world. Considered a bug in the original theory, it is a welcome and crucial feature of physical programmability since it conceptually aligns with the required pre-determined set up of automata by agents.

The notion of structural equations enables us to translate talk about causal relations into talk about relations between variables. These variables stand for properties or events obtaining different values. Formally, a causal model is then definable as $\langle V, S \rangle$, where V denotes a set of variables and S is a set of corresponding structural equations (we have already seen an instance of this in the discussion of I/O mechanisms in sect. §3).[131]

DAGs are best introduced by way of example. Consider therefore the following case of *Ohm's law* adopted from Hausman (2005). The corresponding structural equation is the familiar formula $I = \frac{U}{R}$, (with U for voltage, R for resistance, and I for current); the corresponding DAG is depicted in Fig. 3.

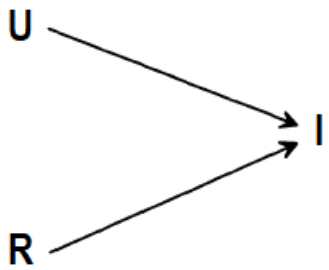

Fig. 5.3.: A directed acyclic graph (DAG) of *Ohm's Law* with the structural equation *I=U/R* (graphic adopted from (Hausman 2005)).

Analogous to what can be seen in (Fig. 3), variables are always represented as nodes, and arrows are drawn from causes to their direct effects. Based on these conventions, DAGs generally allow us to easily read off dependencies. Importantly, each 'parent-child' relationship in the causal graph represents a stable physical dependency, allowing a *surgical* intervention on one such relationship without changing the others. An intervention is called surgical if no other causal relationships in the systems are affected. For instance, if a value of U were to change (i.e., take on a different value) while maintaining the same resistance R, we would see a change in the current I *caused* by that change.

[131] Standardly, structural equations are defined as $x_i = f_i(pa_i, u_i), i = 1, \ldots n$, where pa_i denote the set of variables (the 'parents') that directly determine the value of X_i and where U_i stands for errors or disturbances (see Pearl 2009, 27). Each of these structural equations corresponds to a causal dependency relation. Changing the values of variables (of a given structural causal model) under external interventions uncovers those causal dependencies. In this way, the intuitive content of causal claims (*C* causes *E*) is preserved, yet concerns about the dependency of agents are side-stepped.

Wiggling on one of the parent variables *U* or *R* therefore enables us to directly causes a change in the value of *I*.[132]

I submit that making use of the insights of interventionism elucidate how we can reconfigure real-world programmable automata. However, in order to fully generalize interventionism's conceptual resources and adjust them to programmability, we need to apply them to the mechanisms of programmable material automata. How exactly does the formal machinery of interventionism apply to mechanisms?

5.4.2 Control through Mutual Manipulability

In recent years, there has been a growing interest in applying causal modeling to higher-level phenomena, particularly in the interventionist framework and its application to general types of mechanisms and their phenomena. I will now demonstrate how applying interventionist concepts to I/O mechanisms can help us understand how to control programmable automata. Importantly, we can change an automaton's behavior by two conceptually different types of interventions: First, we can wiggle on the inputs *i* of the corresponding I/O mechanism *f(i)=o*. Changing the course of action this way brings about a particular pre-determined execution trace due to providing different 'data.' Secondly, and more relevant to the current discussion, we can alter the mechanism's internal functional relation *f* between the inputs and outputs. This second way of varying the system's behavior in a controlled way amounts to genuine reprogramming.

However, combining interventionism with mechanisms is a non-trivial matter. While there is agreement *that* there is a philosophical problem, there is less consensus on *how* to solve it. To understand the issues at hand, we need to look at the intra-level relationship of mechanistic levels again (cf. sect. §3): Whereas the relations among acting entities X_i are widely considered as causal (black arrows in Fig. 3), the relations between any individual part and the explanandum phenomenon *S*'s *Ψ*-ing are up to debate (dotted vertical line in Fig. 3). Mechanistic philosophers typically distinguish between etiological (causal) and *constitutive relations* (see e.g., (Ylikoski 2013)). Applying interventionism to causal relations is unproblematic, as tracking causal dependencies is one of interventionism's main objectives. However, it is the second, non-causal

[132] N.b., when employing this kind of thinking, we are engaging with modal reasoning, "[c]ausal relationships between variables thus carry a hypothetical or counterfactual commitment: they describe what the response of Y would be if a certain sort of change in the value of X were to occur." (Woodward 2003, 40) It is thus now generally accepted that interventionism is a counterfactual theory (of causation); the notion of a surgical intervention that unearths causal relationships requires counterfactuals.

relationship between *S*'s Ψ-ing and *X*i's ϕ-ing that requires substantial philosophical caution.

In short, the issue is to determine which of the various entities X_i and their properties *constitute* S's Ψ-ing and what exactly the constitution is (Kaiser & Krickel 2017). This puzzle is generally discussed under the name of *constitutive relevance*. The most widely accepted proposed solution is Craver's so-called *mutual manipulability* account (MM) (2007a, 2007b), which suggests that constitutive relevance is defined by how scientists *manipulate* a mechanism's component in experimental research practice in order to study its behavior. As Craver explains,

> "a component is relevant to the behavior of a mechanism as a whole when one can wiggle the behavior of the whole by wiggling the behavior of the component and one can wiggle the behavior of the component by wiggling the behavior as a whole. The two are related as part to whole and they are mutually manipulable.", (Craver 2007b, 153).

The idea is that some ideal intervention on a component X_i's ϕ-ing alters the phenomenon (S's Ψ-ing) and vice versa, i.e., some ideal intervention on S's Ψ-ing in turn also makes a difference for the component's ϕ-ing. While a characterization along these lines essentially captures how we can intervene in mechanisms to program them, I will briefly return to some recent philosophical problems concerning these matters when closing this chapter.

5.5 The degree to which

Lastly, I need to clarify in which sense physical programmability is a gradual property/notion. At a first stab, the gradual nature of programmability appears to correlate with the variability of the potential behavior of the system under scrutiny. Take, for instance, the apparatus with virtually no control autonomy we encountered earlier – the ancient flute player of the Musa brothers. While the flute player fulfills all the requirements to be conceived as a material automaton, the device is not programmable. After construction, it always carries out the same 'program' (i.e., it always plays the same melody). The only way to influence *S*'s Ψ-ing, where *S* denotes the system at hand and Ψ-ing its selected operation, is by reassembling the device altogether.

Contrast the non-programmable flute player with a material automaton designed to have (external) control features. In this case, a replaceable external medium allows for controlled manipulation of the pre-determined sequence. So-called *Jacquard looms* (as shown in Fig. 4) are a prominent example of automatic sequence control (Randell 1994; Koetsier 2001). Jacquard looms were special kinds of weaving looms that were (re-)configurable by a chain of punched cards

to produce fabrics with a desired pattern.[133] The punched card's pattern of holes determined which of the loom's levers was activated when pressed against a dedicated control mechanism (e.g., some 'read-out' lever). Operators could

Fig. 5.4: 19th century engraving of a Jacquard loom. The desired weaving pattern on the fabric could be controlled (by the operators) by inserting a series punched cards (here, at the top) into the machine. Technologically similar control mechanisms are also used in some musical automata and even computing devices.

change the pattern of holes to alter or intervene, i.e., to 'reprogram' the machine's behavior according to their desires. Similar technological considerations found application in more sophisticated musical automata controlled by pegged cylinders. Reprogramming such devices as advanced music automata or looms was possible *after* construction and without rebuilding the entire system. Ostensibly, the Jacquard loom purports to have higher programmability than the Musa flute player because it displays a higher variability among its potential sequential behavior (which we can achieve in a controlled way). Notably, as previously pointed out, the variability we are interested in concerns manipulating the functional relation *f* with regards to an I/O mechanism $f(i)=o$.

Some historico-philosophically inclined scholars have tried to pack these kinds of considerations about *all sorts of* programmable automata into a theoretically more robust classification scheme. Koetsier (2001), for instance, analyzes the different degrees of programmability of pre-20th century automata like music automata, clocks, and looms, concluding that

[133] Essentially the same control mechanism was also employed in many computing machines. See Campbell-Kelly (1991) for a detailed treatment.

> "[t]he earliest programmable machines were musical automata. Next came the programmable drawloom. The first designs of programmable drawlooms are from the 18th century. It is remarkable that the main predecessor of the Jacquardloom was designed by De Vaucanson, the well-known 18th century builder of music automata. Following the success of the Jacquardloom the idea of a programme in the form of punched cards was used by Babbage in the 1830s in his design of the first programmable computer. Later punched cards were also used widely to programme music automata." (Koetsier 2001, 602)

Similarly, D'Udekem Gevers (2013) delves into the fascinating evolution of automatic devices from the 12th to the 19th century. Her analysis provides a detailed taxonomy of programs (in a broad sense) implemented in clocks, musical instruments, looms, and early computers. Accordingly, we ought to distinguish between material automata whose sequence is internally fixed, externally modifiable/replaceable, and fully manageable by a stored program.

Others devised grading schemes that underscore the conceptual differences between early music automata (similar to the Musa flute player), Jacquard looms, and special-purpose calculators. As Brennecke (2000) explains, looms and music boxes can only execute fixed sequences (after being programmed) since they are all controlled by (a program held on) some read-only medium. On the other hand, many special-purpose calculators and modern computers have additional control structures that can use their output as input. This feedback enables the influence or control of the original sequence of operations through *iterations* and *conditional branching*. (I will shortly return to the importance of these features below). Lastly, Copeland & Sammaruga (2021) developed a 'hierarchy of programming paradigms' in which they exclusively focused on computing machines of the 1930s and 40s involved in the emergence of the *stored program concept*. Similarly to previous works, the authors suggest differentiating between machines that require physical rewiring, have an external memory medium, and contain stored programs.

What is the takeaway from these classification schemes of real-world programmable automata? How does it help us to couch the gradual nature of programmability? Although above's proposals differ slightly in scope and methodology, they all classify *general-purpose* computing machines as the ones with the highest programmability. Simply put, general-purpose computers – sometimes called universal machines – are said to be capable of implementing virtually every *computable* function (as defined by computability theory). In other words, the class of computable functions acts as an *upper limit* to the sequence of operations a computing machine can carry out.[134] If we can set up a machine to implement all these functions, it is fully programmable; machines that can

[134] I am ignoring hypercomputation, etc., for now.

implement only a smaller range of functions are therefore less programmable. In sum, we can make sense of the degree to which a material automaton is programmable as

The degree to which: The amount/share of the set of possible functions (in the sense of the I/O-mechanism $f(i)=o$) the system can implement.

However, despite the seeming plausibility, there are potential caveats to this view that require clarification:

Caveat 1: There is no universal measure

First, it is vital to remember that the notion of general-purpose/universal machines applies to computable functions only. However, since physical programmability applies to a broad range of material automata, each with operations other than computing, we also want our grading scheme to cover these cases. The problem is that the label 'general purpose' as presented so far merely makes sense regarding a specific phenomenon (namely, 'physical computation'). It can, hence, not be a benchmark for full programmability concerning sequenceable operations that are different from physical computation, like sound production or weaving. It is, therefore, questionable how far our initial intuition of a universal measure of the degree of programmability fits other kinds of material automata that lack a formal theoretical underpinning and do not have a clearly defined class of all potentially implementable functions (for instance, it strikes me as doubtful that there is rigorously definable set of possible sequences of operations regarding melodies or weaving). In light of these concerns, it seems wise to maintain a pluralist stance and judge a system's degree of programmability relative to a chosen operation.

Caveat 2: Relying on informal notions

Second, over the years, many researchers raised caution when using expressions like 'stored program concepts,' 'general purpose machine,' and 'universality' (see (Olley 2010) for an accessible survey of the relevant literature). The problem is that (in textbooks) these terms are frequently used interchangeably, potentially leading to misunderstandings when we try to make judgments about an automaton's degree of programmability. To clarify: Typically, the stored program concept refers to internally storing instructions and data in the same writable memory. The concept enables the manipulation of instructions based on intermediate results, such that the machine can perform iterations and conditional branching. These control structures are widely believed to render a

machine *Turing complete/universal*. Given unlimited storage, the machine could implement any computable function.[135]

However, this design choice is just one of many ways to achieve (quasi) Turing completeness. Machines with different architectures that store data and instructions in entirely different manners could also be Turing complete. Rojas, for instance, discusses examples of achieving universal machines by other means (Rojas 1996, Rojas 1998, Rojas 2023). In his (Rojas 1996), he proves that conditional branching can be substituted by unconditional branching such that externally stored looping programs using indirect addressing and no branches can be as powerful as machines operating under the stored program paradigm. (This requires simulating a branch by carrying out multiple paths of the branch and negating any contributions from the path that a genuine branching would not take.)

Now, the reason why this is relevant for the current discussion is that these results can be transposed to machines that do *not* store programs and data in the same medium. In particular, Rojas argued that Konrad Zuse's *Z3* (Rojas 1998) and *Z4* (Rojas 2023, 149-154) computing machines could, *in principle*, implement the same range of computable functions as a device constructed under the stored program principle. Interestingly, Rojas's universality proof for the *Z3* sparked a host of similar works that showed that ancient computers, never designed to be universal, are so in principle (Copeland & Sommaruga 2021, 88-89). The upshot of the work of Rojas and others is that general-purpose machines can thus be constructed by different designs (that do not store programs and data in the same memory). Universality is achievable in many, though not immediately apparent, architectures. Accordingly, it may turn out extremely challenging to determine the range of computable functions some unconventional machines can implement (Bromley 1983) and, in turn, judge their degree of programmability. We thus should be careful with judgments about the degree to which some systems are programmable.

5.6 Concluding remarks and Open Questions

Programs devised by human agents may consist of simple to highly complex sequences of operations. The sequenced operations range from sound (music boxes) and weaving (Jacquard looms) to computation. To execute any desired sequence of operations, the chosen system must be configured appropriately, requiring specific (physical) interactions: the machine needs to be programmable.

[135] It is important to note that real-world machines are only potentially universal, as they cannot be given unlimited storage. Therefore, today's computing machines can only perform computations that a TM with bounded tape can achieve.

Unfortunately, philosophical discourse regarding programmability is scant and largely underdeveloped. This contribution extended this area of investigation by developing an original and robust notion of what I refer to as

> **Physical Programmability**: *The degree to which the selected activity/function/operation/phenomenon on an automaton can be reconfigured in a controlled way.*

What distinguishes this novel notion is that it weaves together well-established theoretical and philosophical discourses into a tailored framework that accounts for how we set up our machines. Subsequently, I fleshed out the corresponding variables in that characterization and explained how they are connected. Accordingly, the main takeaways are fourfold: First, the domain of systems that can be bestowed with the property of being physically programmable is limited to 'material automata.' Two, the selected operation of these material automata is explained best through the neo-mechanistic framework. Third, I expanded the understanding of 'reconfiguration in a controlled way' by establishing a connection between mechanisms and manipulability theory (especially Interventionism á la Woodward). Fourth, by discussing various examples of automata, I showed that physical programmability is a gradual notion and comes in different degrees.

Given the novelty of the subject, there remain open questions and prospects for further development. Two issues are of particular importance.

Fathanded interventions

The ongoing debate in the mechanism discourse has resulted in the first open question. It concerns the interplay between interventionist framework and mechanisms. More concretely, in recent years experts have extensively scrutinized the plausibility of MM Couch (2011), Leuridan (2012), Romero (2015), and Kästner (2017). Whereas interventionism is an approach to causation, constitutive relevance is deemed a non-causal relation. Accordingly, interventions on mechanisms may violate the surgicality condition and are hence called *fat-handed* ((Scheines 2005, 932) and (Woodward 2008, 209)) since they make a difference in the mechanism and (at least some of) its acting entities.[136]

To date, the constitutive-relevance debate remains an active field of research without consensus: Romero (2015), Baumgartner & Gebhartner (2016), and

[136] A concise summary is given by Kästner and Anderson (2018, §3): "Since wholes cannot be manipulated without affecting any of their parts, interventions into the whole will always be non-surgical, that is, fat-handed, with respect to some part. Rather than intervening into X (the whole) with respect to Y (the part), we actually intervene on X and Y simultaneously by carrying out I."

Baumgartner & Casini (2017) propose to revise standard Interventionsim á la Woodward (2003) and add different types of so-called *fathandedness criteria* to MM. Yet Krickel (2018), raised doubts regarding fat-handedness-approaches, proposing an alternative that is supposed to rescue the original version MM. The challenge remains to create a coherent theoretical framework for interventionism and mechanisms alike.[137]

Although a considerable body of research has couched the debate primarily on a technical level, less attention has been paid to the result of plugging in specific phenomena for S's Ψ-ing. Future research could therefore focus on specific phenomena related to sequenced operations; particularly 'physical computation' and its connection to programmability appear to be a worthwhile area of investigation. Despite there being a well-established theory of mechanistic computation (see for instance, Piccinini 2015, Mollo 2018, Dewhurst 2018), previous research has so far overlooked the challenges associated with programmability and in particular interventionism.

Programmability and its relation with other computing paradigms

The second issue concerns the application of programmability to computing systems. I argued that physical programmability should only be applied to physical systems whose computationally individuated states can, in principle, be intervened upon such that the implemented computational function can be altered reliably. However, whether its application is compatible with non-digital or interactive computing systems is somewhat unclear. While concerning computing devices, this chapter exclusively focused on (sequential) digital machines; it remains an open question to what extent physical programmability can be successfully applied to natural, analog, or quantum computing instances. Each of these cases bears their unique challenges: Natural computing systems like the brain are often held to compute by means other than program execution because they are not intentionally set up by human design choices.[138] The absence of intentional function ascription casts doubt on the appropriateness to speak of programmability (at least as devised here) with these systems. Analog computers, in contrast, raise the issue that some consider them to compute 'in one go', i.e., they do not compute in sequential steps. If this assessment is correct,

[137] Despite the challenges, I agree with (Kästner and Andersen 2018) that both interventionism and MM have solid empirical foundations (see, for instance, Craver (2007b, 144-152) for some details on the empirical grounding of experimentation on mechanisms). Thus, it is not necessary to give up on the mechanistic framework or the idea that we can intervene on mechanisms. Rather, the focus should be on construing the theoretical underpinnings of intervention-based inquiry into mechanisms in a coherent way.

[138] Analogously, one may also formulate the issue for ML systems because there we encounter the similar worry that it is not the humans who predetermine and thus program the machine.

then the application of physical programmability to analog devices may be in jeopardy due to tensions with the demand of executing a *sequential* series of operations. Quantum computers may require special treatment due to the non-classical behavior of quantum states. For instance, the interaction and read-out of quantum states to program a quantum computer may require care with phenomena such as collapse or decoherence.

Furthermore, it remains unclear whether physical programmability and other forms of interaction are compatible. Specifically, it would be worthwhile to investigate whether this concept is consistent with the interactive computing paradigm. Over the past few decades, laptops, smartphones, etc. have evolved into interactive systems, in which programs accept (external) inputs from users during runtime. Consequently, human-computer interaction raises several new issues related to computability theory and accounts of physical computation, where the course of computation is left unaltered (Martin et al. 2023). Currently, there is no agreement on whether it is necessary to differentiate between a priori programming and altering the course of computations during execution. Should we only refer to the former as programming?

6 Conclusion

My thesis, *Mind the Gap*, allows us to examine the ontological status of computer programs from new perspectives. Throughout the manuscript, it became apparent that the research topic is significant because, to this day, a consensus remains elusive, and opinions diverge significantly. Therefore, one of the main goals in writing this dissertation was to provide a cohesive and thorough overview. The information available is mostly scattered across various discourses – it is now consolidated into a single monograph, making it easier to access.[139]

To illustrate the complexity, I initiated the thesis with a hypothetical scenario grounded in real-world events: A young IP lawyer in the early 1970s grappling with the legal classification of software. This case revealed substantial conceptual disharmony and ontological uncertainty surrounding computer programs. When scouting the relevant literature beyond the legal one, we learned that matters are similarly diffused today (cf. Appendix A). No single conception of the nature of programs would be entirely satisfactory as mutually exclusive characterizations such as texts, configurations of machines, or algorithms all appear to be plausible options. What gives?

While previous studies interpreted this ambiguity as programs having a 'dual nature,' I think they needlessly complicated the debate due to being confused by reflecting on the language they use, particularly the term 'computer program.' As a result, almost any discourse underpinned by the metaphysical nature of computer programs (e.g., in the legal (Con Diaz 2019) and verificationist debate (MacKenzie 2004; Tedre 2015)) remains inconclusive, at best. In order to systematically unscramble things, Chapter 1 provided some necessary terminological clarifications by taking a closer look at the origins of the expression 'computer program' and what it is supposed to refer to. Arguably influenced by the rampant epistemic pluralism of computer science, the takeaway is that the notion lacks a clear, agreed-upon definition. While similar observations have been made before (e.g., Eden 2007; Eden & Turner 2007), I

[139] I extensively drew from various philosophical literatures to grapple with the question of the metaphysical nature of computer programs. From the Philosophy of Technology, I incorporated the concept of artifacts and teleological functions, discussing them with examples of (computing) devices such as the MONIAC, the IAS machine, music boxes, and Jacquard looms; from the Philosophy of Science, I used the conceptions of (material) scientific models and representation, interventionism, and mechanisms. From the Philosophy of Applied Mathematics, I transposed the insights about the applicability of mathematics to computing. From the Philosophy of Art, I put the so-called Problem of Creation into service, and from the Philosophy of Language, I employed the concept of polysemy. Lastly, in the realm of the Philosophy of Computing, I relied on the insights of the physical computation discourse.

made them more precise and submitted that 'computer program' is a polyseme. This insight may help us retroactively clarify (at least in parts) the legal and verificationist debates. Although many of the debate's participants employed the same expression, 'computer programs,' they either referred to ontologically different things (e.g., abstract or concrete things) or had trouble stating the programs' ontological status precisely because they bundled ontologically different things together. I coined the term 'polysemic web' to underscore that 'computer program' can *refer to many ontologically different but related things.*

Faced with potential linguistic quarrels, I avoided an even deeper semantic analysis of the term program. Instead, I proposed to track/emphasize the relations between the elements hiding behind the term in its web. In order to keep things simple from a metaphysical point of view, I suggested starting to place these elements across a simple two-category system – the abstract-concrete distinction – and shed light on their connection. Specifically, I argued that the concept of 'computational implementation' from the philosophy of computing could help clarify the situation. By making an abstract scenario concrete, implementation can be seen as a connection between ontologically different relata (abstract and concrete). To put further meat on this idea, I surveyed the literature on implementation in Chapter 2. My first noteworthy discovery was that two largely separated bodies of literature on computational implementation exist. I henceforth called the corresponding notions type (A) and type (B) implementation. Surprisingly, both notions have mainly developed independently of one another. To remedy the situation, I juxtaposed the two notions.

Next, in the wake of my conclusion, I created a framework based on the conceptual tools of the philosophy of science literature suited to accommodate them both. The upshot was that two understandings of implementation are combinable when alluding to the conceptual tools of the material models and scientific representation literature. This conceptual borrowing is productive because, in both modeling and computing, agents engage in object-based reasoning, where artificial functions are externally attributed, and agents establish a mapping relation between a concrete system and an abstract target/program. To highlight the central role of epistemic agents and the framework's ability to unify (A) and (B), I called it a *Unified Theory of Agential Implementation* (UTAI).

Since UTAI gives rise to three distinct dependency relations between epistemic agents and (a) 'abstract programs,' (b) computational implementation, and (c) the physical computing system, I devoted the rest of my inquiry to shed light on (a)-(c).

When considering dependence relation (a) in Chapter 3 to shed light on programs qua abstract objects, I utilized the well-known Problem of Creation (PoC) from the Philosophy of Art literature. The crux of the (PoC) is that certain, so-called repeatable artworks are deemed abstract since they have multiple instantiations. However, since we typically assume that artworks are artifacts (i.e., intentionally created objects) *and* also think abstract objects cannot be created, we have a triplet of mutually inconsistent propositions. Very roughly put, to resolve the paradox, one has to give up one of the prima facie plausible propositions, and three major options emerge: Platonism, Nominalism, and Creationism.

My motivation for appealing to the (PoC) was the similarities between repeatable artworks, such as works of literature and musical compositions, on the one hand, and the textual view on programs on the other. The key lies in programs' multi-realizability. Since programs written in standard high-level programming languages are portable and can have many different copies, they exhibit the same kind of 'repeatability.' Provided that programs are thus subject to the (PoC), we have a new and robust theoretical underpinning to refine the ontological sorts of questions we can ask about programs in the future. The most attractive feature of this research program is that it allows us to do so in contemporary metaphysical terms. In other words, it enables us to foster synergies with much more mature ontological debates and steer us away from dubious dual nature talk.

Next, Chapter 4 tackled dependence relation (b), i.e., the way in which the implementation relation hinges on the practice of human agents. Having framed the Problem of Implementation in terms of the Bridging Problem of Applied Mathematics in Ch. 2, I indicated that assuming a dyadic relation between physical system and abstract logico-mathematical computational formalism is metaphysically mysterious. As detailed in the supplementary material in Appendix B, it is the consensus of philosophers of applied mathematics (e.g., Batterman 2010; Bueno & Colyvan 2011; Nguyen & Frigg 2021) that a third relatum is crucial to make sense of the math-world relation.

Upon further exploring these considerations, I provide a new way to think about interpretational accounts of physical computation, specifically recent versions that couch implementation in terms of scientific representation. Particularly, my novel notion called 'Implementation-as' extended recent research in the philosophy of computing of so-called scientific representation accounts (SRA). The underlying idea of all (SRA)s is that it holds promise to couch computational implementation in terms of scientific representation since both relations rely on mappings between the physical and the formal.

Implementation-as departs from the previous (SRA)s by fleshing out the idea for the first time in terms of a specific notion of scientific representation - Frigg and Nguyen's DEKI account. (Frigg & Nguyen 2018). This new framework's application was illustrated in the MONIAC (an analog device) and the IAS-machine (a digital computer). Subsequently, my analysis shows that the resulting proposal provides a philosophically rigorous theory of computational implementation, satisfying the most standardly evoked desiderata for theories of implementation.

Lastly, I close by discussing the relation of Implementation-as to already established accounts of physical computation. Traditionally, accounts with any interpretational elements have been shunned in the literature, for they seem to fly in the face of the Computational Theory of Mind and purport to paint physical computation in an arbitrary light. Although future research needs to determine whether the account is compatible with the cognitive science project, Implementation-as' intricate constraints undermine any worries about arbitrariness.

Finally, in Chapter 5, I devoted my attention to dependence relation (c) – the relation between epistemic agents and the physical computing instrument that enables them to implement a program. In a nutshell, the chapter's main result consists of delivering an account of what it is for a physical system to be programmable. Despite its significance in computing and beyond, I showed that today's philosophical discourse on programmability is impoverished. My contribution offers a definition of physical programmability as the degree to which the selected operations of an automaton can be reconfigured in a controlled way. The framework highlights several key insights: the constrained applicability of physical programmability to material automata, the characterization of selected operations within the neo-mechanistic framework, the understanding of controlled reconfiguration through the causal theory of interventionism, and the recognition of physical programmability as a gradual notion. The account can be used to individuate programmable (computing) systems and taxonomize concrete systems based on their programmability.

Big picture-wise, the most important takeaway is that the term 'program' is a polyseme that denotes ontologically different, albeit *related*, things. In this thesis, I have introduced the UTAI framework (inspired by the literature on material scientific models) to track these relations. In sum, UTAI underscores the involvement of human agents that use computers as epistemic tools. On this view, particularly three dependency relations associated with three distinct philosophical problems require our attention: The Problem of Creation, which determines the abstract nature of programs; the question of physical

programmability, which determines the physical side of things; and the Problem of Implementation, which addresses how the two ontological domains of abstract programs and the physical world relate.

On this note, I would like to close by addressing the following question: What are the implications of this thesis for *future* studies in the philosophy of computing and adjacent fields? I want to conclude with some programmatic suggestions about what I believe to be the most pressing issues left unanswered or raised by my thesis.

First, much work remains to be done concerning terminological clarifications, rendering the expression 'computer program' and the relationship with its cognates like 'software' and 'algorithm' more precise. Especially challenging in this regard will be appreciating computer science's epistemic pluralism and the widespread use of computational terms in differing communities and research traditions. Accordingly, it will be difficult to devise a definition that will satisfy the majority of stakeholders involved.

Second, it is important to note that the computing landscape is constantly evolving. During my dissertation project, we observed firsthand the rapid rise of AI applications in the public sphere. However, the question of whether we should consider deep neural networks, for example, as computer programs remains open. While I believe that it is reasonable to classify them as computational artifacts, future research needs to delve into the nuances of their (dis)similarities with 'classical' programs.

Lastly, it is worthwhile to advance some of the frameworks developed in this thesis further in their own right, detached from the question about the ontological status of computer programs. As already briefly mentioned in bypassing, Implementation-as, for instance, may be a viable contender in the landscape of contemporary theories of computational implementation independent of this thesis' main topic. Similarly, it would be interesting to explore further some of the research trajectories enabled by the (PoC). A research program along these lines would make the debate about the ontological status of computer programs less insular and may put the discussion at the center of contemporary metaphysics. Regarding Physical Programmability, it would be fascinating to explore the possibility of developing a fully formalized programmability measure similar to computability or complexity theory. On a different note, one could merge some critical insights of Implementation-as and Physical Programmability to address largely ignored phenomena such as interactive computing.

Appendix A: A Guide to the Chimera of Programs

This appendix charts the varying views of the ontological status of computer programs. It aims to illuminate how and why the nature of programs can be understood in so many ways. For so doing, I have surveyed a rather large, heterogeneous array of opinions about the nature of computer programs: Philosophers, computer scientists, lawyers, and other investigators have placed computer programs in nearly every available ontological category. Some consider them physical objects, others abstract logico-mathematical objects, special kinds of texts, etc. Accordingly, my survey is positioned at the crossroads of rich and well-developed traditions in corresponding fields such as Philosophy of Science, Mathematics, Technology, and Art.

Given that I engage with so many fields, the reader may wonder where this work belongs and what it is good for. The material presented in the following pages may be helpful in a couple of ways: First, in its own right, this part of the appendix may serve as an extended and updated overview of the metaphysical nature of computer programs.[140] This is needed to clarify implicit assumptions, enable comparison, avoid further conflations, enhance philosophical rigor, and help navigate a largely unstructured body of literature. Second, I hope it will facilitate building bridges across the different debates mentioned here in future research.

Methodologically, I proceed like this: Similar to a recent survey for educational purposes about the different guises of programs (Lonati et al. 2022), I provide different clusters containing views about the metaphysical nature of programs. In presenting so many views across various fields, I do not intend to show off how much I read. I attempted to summarize what I think are some of the most important takeaways I distilled from humbly engaging with these literatures. While so doing, I tried to make this appendix accessible to audiences with varying backgrounds. Although I try to be systematic and thorough in my review (e.g., each view branches into further specific positions), the resulting taxonomy should be taken with a grain of salt. Even though many of the following categorizations seemingly stand at odds with each other, they are intertwined in a way that does not allow strict/sharp separation.[141] I believe that

[140] Generally speaking, few studies explicitly describe the ontological status of computer programs; exceptions are Gemignani (1981) and Lonati et al. (2022).

[141] For instance, as we will see, a case in point is the metaphysical understanding of programs related to 'programming languages' sits at the border of the abstract mathematical objects and notational artifacts.

this is not a flaw of my taxonomy, but rather a symptom of the *epistemically heterogenous* nature of computer science and the polysemic nature of the terms 'program' I described in the introduction.[142]

In what follows, my overview is divided in five overarching sections: In A.1., I sum up the Physical View. A.2., contains an overview about the Mathematical View. A.3., dubbed the Symbolic View, surveys material emphasizing the symbolic nature of programs. Next, A.4., summarizes views according to which programs are sorts of artifacts. In A.5., I present what I call the Neural View. Lastly, in A.6., I cash out the State of the Art.

A.1 The Physical View

As seen in the prologue, considering programs as physical objects has been one of IP lawyers' main strategies to secure programs' patent protection (cf. Con Díaz 2019). According to this view, programs are a physical machine's unique configuration/switch setting. Analogously, my considerations in Chapter 5, *Physical Programmability*, remind us that the physicality of programs raises interesting philosophical questions related to the philosophical literature on concrete technical artifacts, their underlying mechanisms, and how we can intervene in them. Moreover, such a view emphasizes that programs appear to be "executable entities" (Lonati et al. 2022) anchored in real-world ongoings.[143] In what follows, I refer to frameworks that advocate for some kind of physical understanding of programs as the Physical View.

In order to unpack the notions gathered under the umbrella of the Physical View, it is helpful to take a more extensive discussion of metaphysics into account: the duality between *continuants* and *occurrents* (see, e.g., Simons 2000). Typical examples for continuants are organisms (e.g., cats, and dogs); atoms and stars; artifacts like chairs and tables; the quality of being red; and social entities like countries or football clubs. The common denominator of continuants is that they are objects that exist in time ("continuants persist by enduring" (Simons 2000, 59). In contrast, prime instances for occurrents are events, happenings, or processes such as philosophy conferences, football matches, photosynthesis, and subatomic particle collisions. Therefore, occurrents are characterizable as objects in time with temporal parts.

Over the next two subsections, I show that this division is also reflected in different metaphysical understandings about programs as physical entities.

[142] I adopted the expression 'epistemically heterogenous' for this context from (Imbert & Ardourel 2023).

[143] Another noteworthy case is the so-called *verificationist* debate I briefly touched upon in the introductory chapter.

Accordingly, I distinguish the physicality of programs between two different cases, viz., a static and a dynamic reading.

Static

According to the static reading, programs may be considered part of a machine (cf. Fig. A1). This idea was perhaps more apparent when using first-generation computers like ENIAC, where switch settings were visible/tangible. The machine had to be physically configured to execute the operations required for a given computation in the correct sequence. 'Programming' the ENIAC thus

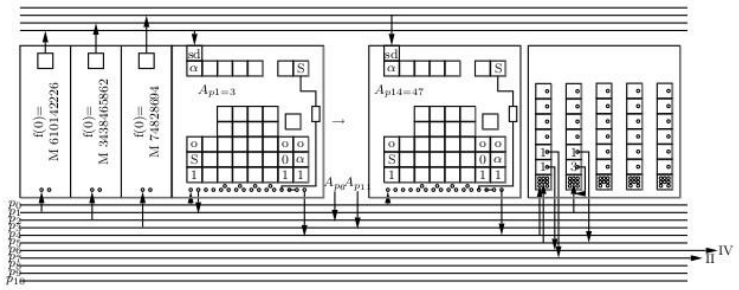

Fig. A.1: Schematic depiction of an ENIAC 'program.' (taken from Bullynck & De Mol 2010, 140).

involved physically wiring the relevant units to the so-called program lines, which connected all the machine units. Given this, it would not have been practical or fruitful to write down a program for the ENIAC as a list of commands. Instead, the scientists, employed wiring diagrams that showed the connections that had to be made between functional units to ensure the correct sequencing of operations. (Bullynck & De Mol 2010; Priestley 2011, 111-115).

Today, we typically no longer need to set switches manually since the process is automated. Consequently, it is easy to forget the 'old way' of doing things (and the static view). However, as Gemignani aptly put it in his survey article more than 40 years ago, seeing programs as part of the machine (or a configuration of it) appears to be necessary to make sense of the notion of 'control':

> "If a program controls a computer in the same way that a distributor controls the sequence of firing the sparkplugs in an internal combustion engine, then the program can reasonably be viewed as an integral part of the computer itself. This view is strengthened by the fact that a general purpose computer will not, indeed cannot, carry out its appointed task until it has been properly programmed. The programming sets the switches, in effect redesigns the internal structure of the machine, becomes an inseparable part of the machine, if the machine is to perform as the program was written to make it perform. The program may thus be viewed as a machine part or as the completion of a previously incomplete machine." (Gemignani 1981, 187)

Besides motivations stemming from the notion of control and machine configurations, there is another way of identifying programs as 'static' (physical objects). I cashed out this view in Chapter 3 when tracing the philosophically viable answer to the so-called *Problem of Creation*. Accordingly, one may think of programs as unique kinds of text and identify them with *particular* physical tokens of their inscriptions (e.g., existing as files on disk or in memory until they are executed). Typically, this sort of text (often called *program script*) contains instructions in a human-readable format, which is then processed (e.g., by automated compilation or interpretation) to be machine-readable. I will return to this last point ('programs qua texts') when considering the Symbolic View below (§A.3).

Process/Dynamic

On the other hand, there is a pervasive view that programs bring about or even *are* sorts of real-world processes (sometimes called *program process*). Emphasizing the empirical side of things is, for instance, prominently discussed in the literature about the nature of computer *science* as a discipline. One of the earliest and most famous suggestions on computer science as an empirical field dates back to a 1976 paper by Newell and Simon (Newell & Simon 1976). In a nutshell, the idea is that the discipline exhibits scientific potential in the form of experimentation like in the natural sciences. Put differently, computing processes (such as program execution) are regarded as entities that can be experimented with.

Subsequently, however, the question arose about whether computer science qualifies as a science in the same sense as the natural sciences. As succinctly stated by Mahoney, the issue is that

> "There is nothing natural about software or any science of software. Programs exist only because we write them, we write them only because we have built computers on which to run them, and the programs we write ultimately reflect the structures of those computers. Computers are artifacts, programs are artifacts, and models of the Computers are artifacts, programs are artifacts, and models of the world created by programs are artifacts. Hence, any science about any of these must be a science of a world of our own making rather than of a world presented to us by nature." (Mahoney 2000, 25)

Later, I will discuss a few more specific aspects of these observations when addressing the *Artifact View* (§A.4.). For now, I simply note that Mahoney's remarks warrant caution when comparing computing with disciplines like physics or chemistry that investigate natural phenomena.

In recent years, some scholars, such as Tedre (2011) and Schiaffonati &Verdicchio (2014), have critically assessed the implications of computer science

as an empirical or experimental field. As Tedre (2011) summarized, the 'pro-science argument' frequently asserts that although computer science is not a natural science, it is nevertheless empirical or experimental. The idea is that computer scientists still follow the scientific method, which involves exploring and observing phenomena, forming hypotheses, and empirically testing those hypotheses. Now, even though these sources admittedly do not seek to address the ontological status of computer programs directly, their underlying assumptions seem to suggest that computational processes (like program execution) are physical processes that can be studied like other physical phenomena.

What are some of the philosophical upshots of the Physical View? One interesting discussion point is that the Physical View seems to equate programs with the lowest level of the computational hierarchy. This way of viewing things may have important implications for discriminating between different programs.[144]

For instance, consider the example of two compatible IBM computers from the early 1960s – the 709 and the 7090.[145] These devices are especially suitable for discussion because the 7090 is a second-generation transistorized version of the earlier tube-based 709. Despite using different electronic components, both devices have the same logical layout.[146] Therefore, they are entirely 'software compatible' and can run the same source code and implement the identical *computational hierarchy* (i.e., various of the same LoA; cf. Chapter 2), albeit (and this is the crucial part) with different underlying components. Even though the *computational hierarchy* is almost identical, the proponent of the static reading of the physical view might still argue that the two machines implement different programs since identity conditions solely depend on specific machine parts.[147]

Although the example is simple, it exemplifies a pattern of reasoning that concerns others, if not all, implementations of the same source code in different machines (which typically do not rely on the same logic diagram). Now, the reason why this is worthy of mention is that Physical View's way of discerning

[144] I described the widespread view that (artificial) computing systems are composed of different sorts of levels, forming a computational hierarchy, in Chapter 2.

[145] Some philosophers have previously considered the two machines to discuss the issue of multiple realization for computational systems (Wimsatt 2002; Milkowski 2013, Milkowski 2016).

[146] The newer machine has different specifications than the older one. Due to smaller transistors, it is 50% smaller, requires less ventilation, and consumes 70% less power; transistors also operate faster than tubes (Milkowski 2016).

[147] In a recent paper, Ritchie & Klein (2023) argue that the notion of multiple realizability may prevent successful implementation of interactive programs with specific time requirements (they discuss a video game as an example).

different programs stands in stark contrast with our ordinary practice (where we usually think that many programs can be implemented in multiple machines). It thus raises the urgent question, whether the Physical View is compatible with the intuitive view that programs are multiply realizable. As extensively discussed in Chapter 3, this issue resembles a line of reasoning about the metaphysical nature and identity conditions of so-called repeatable artworks like novels or works of music; I refer the interested reader to the corresponding sections in Ch. 3.

Lastly, although the Physical View is only the first suggested interpretation of the current subject matter, it already underscored the diverse ontological understandings that fall under the term 'program.' The question arises: Should we consider 'program' to encompass both static and dynamic views, or should we make a clear distinction between these interpretations?

A.2 The Mathematical View

Several influential figures in the computing world, such as Dijkstra, Floyd, McCarthy, Naur, and Wirth, believed that taking a mathematical and rigorous approach to program construction could enhance the quality of 'software' and programming. Hoare expressed an extreme stance, suggesting that all of computing could be boiled down to mathematics. According to him, computers function as mathematical machines, computer programs are mathematical expressions, programming languages are mathematical theories, and programming itself is a mathematical activity (Tedre 2015, 59):

> "Computer programs are mathematical expressions. They describe, with unprecedented precision and in the most minute detail, the behavior, intended or unintended, of the computer on which they are executed." Hoare (1985, 1)

Accordingly, there is a widespread view that computing is closely related to mathematics and that programming is a mathematical activity; let us call this the *Mathematical View*. Today, different versions of this standpoint are still frequently embraced (or, at least mentioned) to give an adequate characterization of computing as a discipline (e.g., Denning et al. 1989, Eden 2007, Tedre 2015, Bringsjord 2019, Primiero 2020).

If taken at face value, the Mathematical View commits us to see the entities studied in computing as mathematical in nature. Consequently, entities like programs may become the subject of philosophical considerations similar to other mathematical objects like numbers, proofs, etc. Let me first consider a few of the common tropes that seem to give credence to the Mathematical View (computability theory, and algorithms) before closing with some of the most common philosophical issues pertaining to mathematical objects.

Computability theory

Computability theory is a subdomain of mathematical logic that studies and classifies which mathematical problems are computable and which are not (Davies et al. 1994).[148] Until the beginning of the 20th century, the notion of computation was an informal one and referred to an activity that was carried out by human computors (and their instruments). Notwithstanding, the concept of computation was tightly related to the formal notion of proof and calculations. In principle, formal proofs can be validated, following rules of inference step-by-step. Likewise, calculations were typically executed by human computors by mechanically following rules, simply aided by pencil and paper.

The study of what's formally computable gained considerable traction in the 1930s, when several mathematicians from different parts of the world came up with precise, independent definitions of what it means to be computable: Alonzo Church defined the *Lambda calculus*, Kurt Gödel defined *Recursive functions*, Stephen Kleene defined *Formal systems*, Markov defined what later became known as *Markov algorithms*, and Emil Post and Alan Turing defined abstract machines which are now called *Post machines* and *Turing machines*. What motivated the quest to formally capture the nature of computation was Hilbert's program and to solve the *Entscheidungsproblem*.

Now, the reason why this is relevant concerning the ontological status of computer programs is that one may view programs characterized in terms of one of the formalisms of computability theory. For instance, in his recent monograph *On the Foundations of Computing*, Primiero (2020, Def. 52) described this view for the 'configuration'[149] of Turing Machines:

> "A set of configurations for a given Turing Machine is meant to fully and exhaustively expresses the behaviour of that machine, i.e. to represent a program:
>
> […]. The sequence of configurations of a TM says for each stage of the computation what is on the tape at that stage, what is the state the machine is in at that stage, and which square is being scanned and what the next state is. The full set of configurations for a machine is also called its *program*." Primiero (2020, 46; own emphasis)

[148] I already provided a brief introduction to computability theory in Chapters 3 and 5 and will keep myself brief here with regards to formalisms in order to avoid redundancy (for a more formal introduction I refer the reader to these sections; else another entry to the topic can be found in Primiero's (2020) book).

[149] TMs' formalization doesn't require us to adhere to actual components such as 'tape' or 'read-write head' (even though, when considered with care, it admittedly remains a valuable conceptual aid).

Despite their suggestive name (containing 'machine'), TMs are not actual real-world devices but specific set-theoretic structures (De Mol 2021). Accordingly, programs (qua configurations of abstract machines) also purport to be abstract mathematical objects. (Of course, the same holds true for Post machines, the lambda calculus and so on).

Algorithms

As expounded in the introduction, the usage of the term 'program' has been subject to constant change and may obtain slightly different, yet non co-extensive, meanings in different communities related to computing (it is a polyseme). At the same time, other notions like software or algorithm are often used interchangeably with the term (and may turn out to be hard to define, too).[150] Although, disputes over what counts as 'program' and 'algorithm' may be verbal unless we specify the relevant roles these notions (ought to) play in practice, it is nevertheless instructive to try clarifying to what extent programs and algorithms are related.

A good way to get started is to consult Chabert's (1996) rich historical survey on algorithms. Opening his book, he states:

> "Algorithms have been around since the beginning of time and existed well before a special word had been coined to describe them. Algorithms are simply a set of step by step instructions, to be carried out quite mechanically, so as to achieve some desired result. Given the discovery of a routine method for deriving a solution to a problem, it is not surprising that the 'recipe' was passed on for others to use." Chabert (1996, 1)

At first stab, the notion of algorithms seems to be both historically prior to the development of formal notions of computing (and computer programs).[151] Moreover, it concerns a broader range of non-computing phenomena (sometimes e.g. kitchen recipes are regarded as 'algorithms'). Over the centuries, the term algorithm has come to mean any systematic calculation that could be carried out automatically. Nowadays, due to computing's influence, the idea of finiteness has shaped the meaning of algorithms, and we usually distinguish between algorithms that are deterministic or non-deterministic, parallel, interactive, quantum, etc. A typical view on the

[150] Duncan (2014) provided an overview of 12 different criteria to define the term 'software' in his PhD thesis.

[151] The word 'algorithm' derives from al-Khwarizmi, a 9th-century central Asian mathematician. His influential work on algebra provided an exhaustive account of solving polynomial equations by reducing them to standard forms. In the 12th century, his work and others were translated from Arabic into Latin, and his name became associated with the methods contained in his writings.

relationship between algorithms and programs is expressed by Newell, who states

> "An algorithm is more abstract than a program. Given an algorithm, it is possible to code it up in any programming language. You might think that a program should be something like an algorithm plus implementation details. Thus, you examine the text of a purported algorithm-if you find an implementation detail, you know it is a mere program." (Newell 1986, 1029)

In recent years, there has been some impetus to philosophically scrutinize the nature of algorithms (and their relations to computing) more precisely (e.g., Vardi 2012; Dean 2016; Hill 2016; Angius & Primiero 2019; Primiero 2020, Ch. 6; Papayannopoulos 2023). The takeaway message is that many of the debates' participants noted the existence of multiple notions of algorithms. While Vardi speaks of an "algorithmic duality," Angius and Primiero's ontological analysis suggests a three-fold distinction between algorithms as informal specifications, as (linguistically construed) procedures, and as (implementable) abstract machines.

Similarly, Papayannopoulos (2023) pointed out that the notion of 'algorithm' has been conceptualized and used in contrasting ways. His argument goes that moving from an initial informal idea to a more precise formal concept typically involves moving through different stages of conceptualization (from pre-theoretic to proto-theoretic to fully-theoretic). However, when it came to sharpening algorithms' meaning, the last stage of development culminated in two separate conceptions: On the one hand, the 'abstract view' (according to which algorithms are procedures over abstract objects Moschovakis (2001), Gurevich (2012)), and on the other hand, the 'symbolic view' (algorithms are processes that necessarily hinge on some given alphabet and notational system (Kolmogorov & Uspenski 1963).

Given the complex developments under the umbrellas of both 'algorithm' (even after focusing only on classical sequential algorithms) and 'computer program,' there is no obvious/unique answer to their relationship. The issue is that there are many possibly different relationships due to the combinatorics of the various conceptions of programs and algorithms.

Lastly, is there a philosophical takeaway, particularly concerning our metaphysical understanding of computer programs, that follows from endorsing the Mathematical View? To the best of my knowledge, no study has tackled the issue systematically, i.e., having investigated how the plausibility differing metaphysical frameworks of mathematics shape the views presented here. So, depending on one's general leaning regarding (notorious) metaphysical issues of

mathematical objects, one may need to consider vastly different ontological and epistemological issues. To name but a few problems:

(a) If, for instance, one were a mathematical Platonist (see (Linnebo 2024) for an overview of the different positions under the umbrella 'Platonism') about computer programs, then one would have to face the familiar epistemological puzzle – *Benacerraf's identification problem* – of how we could possibly get to know/discover such mathematical objects (I discuss this issue in more details in Chapter 3).

(b) If, however, one were to be a nominalist about mathematical objects, the Mathematical View of computer programs would be in serious trouble (and may collapse into specific readings of the Physical and/or Notational View). Mathematical nominalism posits that mathematical entities do not exist as abstract objects, lacking location in space-time or causal powers (Bueno 2020).

(c) Philosophers of applied mathematics maintain that there are several so-called application problems of mathematics (Steiner 1998; Fillion 2012). The common theme of these problems is how supposedly abstract mathematical entities relate to the physical world. Consequently, when computational entities such as computer programs are considered mathematical entities, these problems also pertain to computing. (I address one – the so-called Bridging Problem, i.e., the metaphysical problem of how the mathematical relates to the physical– in Appendix B.)

Besides these general themes from the philosophy of mathematics, one can expect additional, more specific issues about computability theory, algorithms, etc. For instance, despite my presentation of algorithms under the 'Mathematical View,' we may need to revise this interpretation. Dean (2016, 26) notes that we typically think of mathematical objects as static. However, our usage of operational terms in the specification of algorithms reflects our understanding that executing an algorithm is to carry out a sequence of operations ordered in time.

A.3 The Notational View

Regarding programs as sorts of texts is parasitic to the widespread use of modern programming languages. According to this view, programs are constituted by a well-formed sequence of symbols written in a programming language. This view raises several questions regarding the nature of programming languages and, consequently, programs qua texts written in such a 'language.' As Lonati et al. (2022, 155) aptly remarked recently

> "A program is a notational artifact, in the same sense way that a manuscript, book, or music score is: it relies on a notation with a particular syntax, according to some formal rules, linguistic notations and conventions."

In the following, I call the view to conceive of programs as some symbolic structure as the *Notational View*. It is instructive to provide some brief (historical) background to better understand the Notational View and its philosophical implications. How exactly became the notational view so pervasive? And what exactly are programming languages?

Until the end of the 19th century, computations were essentially performed by human clerks. Typically, these 'computors' relied on pencil, paper, and possibly some (semi-automatic) calculating tools (see e.g., Campbell-Kelly et al. 2023, Ch. 1-3). Unless wholly carried out in one's head, manual computation involved the manipulation of different symbols according to rules of arithmetic. In parallel, there was a long tradition of considering mathematics in linguistic terms, such as the language of nature, the "grammar of science," or, in the 20th century, as a formal symbolic system (Nofre et al. 2014, 47).

The advent of various mechanical and later electronic (special-purpose) computing machines increasingly enabled practitioners to carry out sequences of computations automatically. Notably, the increasing speed and automatization of electronic processing provided by the technological leaps in the 1940s required program execution to rely less on repeated human intervention. For instance, whereas the functioning of the Harvard Mark I required human operators to manually change program tapes for conditional branching during runtime, ENIAC featured fully automatic conditional branching (Bullynck & De Mol 2010; Priestley 2011, 111-115). Although the ever-increasing automatization of computing machines marked a significant development, it gave rise to a novel, unexpected source of error: human programmers' failure to fully anticipate the effects of the given instructions accurately.

At first, practitioners would usually write programs directly in machine code (i.e., referring to one's device's hardware components in binary notation). As Valdez succinctly writes about the state of the art at the time,

> "There was a very close correspondence between the structure of the program and the structure of the machine itself. Consequently, programmers were required to know every detail of the structure and working of the machine they were programming and inevitably the focus in programming was on the formulation of the problem to fit the structure of the machine; the logic of the program was totally shaped by the structure of the machine." (Valdez 1981, 4)

For successful programming, one had to be familiar with virtually every hardware component and the entire architecture of the machine; this method was tedious, made code difficult to read for humans, and the probability of

accidentally making errors was high. Accordingly, from the late 1940s onwards, programmers created a notation called *assembly code* to simplify the process of writing machine code. Rather than writing down the binary digits for each machine instruction, they used short words or abbreviations like ADD, SUB, or MOVE. These words representing instructions were easier for humans to read and remember than a series of 1s and 0s.

In explaining why it became custom to see programming as a linguistic activity, Nofre et al. (2014) remind us that computing specialists swiftly extended the previously mentioned tradition of viewing mathematics in linguistic terms to the mathematical/computational problems solved by the aid of machines. While initially, the adaptation from assembly to machine code was done by hand, it was soon realized that the process is automatable, too. Resultingly, programs called assemblers emerged to perform the process. In this context, the transition from mathematical problem to code was interpreted as an act of 'translation.'

Importantly, it is possible to apply repeatedly, or 'nest' such translation process to come up with new notational schemes beyond first (machine code) and second-generation programming languages (assembly). Throughout the 1950s, this nesting strategy enabled practitioners to invent third-generation high-level programming languages such as FORTRAN, ALGOL, or Lisp that were more programmer-friendly and typically omitted even more hardware details (Knuth & Pardo 1980; Wexelblat 1981). Slowly, this development separated programmers from the intricate make-up and inner workings of the machine. As such, the computing community increasingly distanced itself from thinking of code as an attribute of individual computing devices and began to draw on linguistics and symbolic logic. The reason for this conceptual borrowing is that new notation schemes are similar to formal languages like first-order logic as they have variables (to which we can assign values), predicates, and functions (White 2004). Interestingly, every new 'language' gives rise to a new model of the machine: Although the hardware remains unchanged, the programmers can now reason in terms of variables rather than memory cells or of algebraic formulas rather than registers and adders. Furthermore, the development introduces the notion of 'machine independence,' meaning that a single program can eventually run on many computers.

In the past seventy years, thousands of programming languages have emerged, utilizing various approaches to writing programs. Some languages, known as imperative languages, specify how a computation should be done, while declarative languages focus on what the computer is supposed to do. There are general-purpose languages as well as those developed for specific application domains. For instance, *C* and *C++* are typical in systems programming, *SQL* for

writing queries for databases, and *PostScript* for describing the layout of printed material. Innovations and applications often lead to the creation of new languages. For example, the development of the Internet led to the creation of *Java* for writing client/server applications and *JavaScript* and Flash for animating web pages.[152]

Summarizing these developments, we can (broadly construed) understand a programming language as an artificial notational formalism in which we can express algorithms/computational problems (Gabbrielli & Martini 2010, 27). Abstracting away from the machine allowed the arrival of notational schemes pertinent to the (human) problems to be solved. As described in Chapter 2, 'Notations – There is no Escape' of the PROGRAMme book,[153] notations are thus frequently regarded as intermediaries between human programmers and computing machines (Fig. A.2).

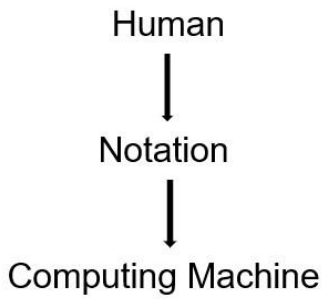

Fig. A.2: Schematic depiction of the relation between computing machines, symbol structures, and humans.

Ultimately the common idea remains to exercise control over the underlying device and use it for problem solving; programmers therefore must formulate or program instructions in a notation that the machine can 'understand.'[154] To turn the human-readable inscription into a machine-readable one, the high-level inscription needs to undergo a translation process and be implemented in a 'lower level.' In practice, we employ interpreters, compilers, and linkers (which are all programs themselves) to go through the different translation stages automatically.[155]

[152] I emulated this paragraph based on (NRC 2004, 76f).
[153] (https://wiki.program-me.org/index.php/Notations:_There_is_no_escape)
[154] N.b. that the term 'understand' is yet another linguistic (and anthropomorphic) metaphor.
[155] These translation stages are frequently discussed under the name 'implementation.' In Chapter 2, I discuss the corresponding philosophical ramifications.

Beyond textual notations

Next to a textual conception, one may employ other forms of notations. Flowcharts represent another prominent notational scheme for computer programming. In his article 'The Multiple Meanings of a Flow Chart,' Ensmenger (2016) illuminates the importance of flowcharts for representing the logical structure of programs beginning in the 1940s. Their genesis can be traced back to a series of reports authored by John von Neumann and Hermann Goldstine. These reports introduced the conventions of the flow diagram notation, intending to capture a program's dynamic unfolding (the program process) by pictorial means. One notable feature of this novel notation was the inclusion of so-called assertion and substitution boxes connected by arrows. These notational devices facilitated the manual conversion of a program to otherwise hardly readable machine code.

Today, programmers still often rely on forms of software visualizations, like Bachman or UML diagrams, to attempt to gain an overview of complex software systems. As such, the purpose of flowcharts diverges from a program as text view in so far as flowcharts are typically not apt to directly exercise control of the machine. Rather, they act as specifications and fill the need to make a program clear to those who want to understand it.

Overall, I have attempted to survey a relatively large area of computer science in this subsection. Accordingly, there is a vast landscape of underlying theoretical and philosophical issues. Let me now sketch some of the implications for the metaphysical nature of computer programs in broad strokes.

The section started with a quote by Lonati et al. (2022), stating that programs are notational artifacts like different kinds of texts or works of music. In fact, there is extensive literature on the ontological status of such repeatable artifacts in the philosophy of art and metaphysics. In so far as one can think of programs as repeatable artifacts, the same conclusions about the ontological status of texts, words and so on should also inform our reasoning in computing. (I extensively discuss the issue and relevant literature in Chapter 3).

At the same time, the nature of programming languages itself raises conceptual issues. Although there are thousands of programming languages nowadays, most share common features when it comes to their semantics. The three most typical styles are operational, denotational, or axiomatic (Jones & Astarte 2018).[156] In some of his work, Turner (2007; 2010; 2014) attempts to

[156] An operational semantics allows us to interpret the meaning of a programming language through an abstract machine. It involves translating expressions in the programming language into instructions or programs for the abstract machine. Denotational semantics formalizes the interpretation of programming languages in terms of mathematical structures (called denotations) like sets or categories that describe the meanings of expressions from the

unscramble how the underlying semantics may furnish potentially different ontological commitments concerning programming languages. One critical problem is that programming languages have an abstract guise we can reason about mathematically but ultimately need to be implemented in a physical medium. Often, this issue is discussed under the label the 'dual nature view,' and we will come back to it shortly in the last section of this appendix.

Another important issue pertains to the relation between programs as static notations and the real-world ongoings in concrete computing devices. As I have argued in Chapters 2, 4, and Appendix B, the relationship between these symbolic structures and material systems is subject to the vexing issue of the *Problem of Implementation*.

A.4 The Artifact View

In our everyday life, we are surrounded and constantly confronted with artifacts. Typically, an artifact is defined as an object intentionally made or produced for a specific purpose (Hilpinen 2017). Intuitively, many computer programs appear to be artifacts because they are 'creations of the mind' (cf. Mahoney's quote a couple of pages ago). In due course, philosophers often distinguish between different types of artifacts. Especially two conceptions turn out to be relevant for classifying computer programs: Technical artifacts and abstract artifacts. In the following, I will provide an entry into these notions (concerning computer programs).

Technical Artifacts

Technical artifacts are considered an important subclass of artifacts in general (I already briefly introduced them in Ch. 2 and Ch. 5). They are taken to include mundane objects like tables, screwdrivers, and toasters to highly sophisticated technologies like particle detectors or spacecraft. All technical artifacts have in common that they are constituted by both *material* and *intentional* features.[157] On the one hand, technical artifacts can be described by their material structure, i.e., in terms of their physical or chemical capacities. On the other hand, they can be couched teleologically with regards to goals and actions. The combination of both structural and teleological (or material and intentional) aspects are required to

languages. The underlying idea is to map a language into some space of such mathematically tractable structures. Lastly, an axiomatic semantics contains axioms and rules of inference that describe computer programs in propositional logic. This approach is beneficial for proving the correctness of programs (*Hoare Logic* is a prominent example).

[157] The materiality delineates technical artifacts from socio-cultural artifacts like constitutions or the law; intentional features render them distinct from naturally occurring material entities like tigers and tornados.

provide a complete picture of them (Kroes 2012). For that reason, technical artifacts are often referred to as having a 'dual nature' (cf. Kroes & Meijers 2006; Baker 2006).

Importantly, the material and intentional features are deemed to stand in a special relationship: It is the artifact's material structure which allows agents to pursue their goals – only if there is the right correspondence between the two may an artifact *function* correctly.[158] Accordingly, philosophers of technology like Houke and Vermaas concluded that "[…] the notion of 'function' is like a bridge connecting the intentional, use-plan description of artefacts and a description of their physicochemical capacities." (Houke & Vermaas 2010, 138). As we will see later on under the discussion of the 'dual nature view of programs', several philosophers have suggested that such teleological function are suited to conceptually connect the different clusters in this appendix.

Abstract Artifact

There is another prominent class of artifacts – *abstract artifacts* (see also the material in Ch. 3). The prefix 'abstract' should be understood in the metaphysical sense of lacking spatial features described previously in the introduction. Despite lacking ordinary spatial features, abstract artifacts are nevertheless characterized as creations of the mind. They are abstract objects that were *created*. Standard examples often mentioned in the relevant literature are fictional characters (Sherlock Holmes, Donald Duck) and other types of so-called repeatable artworks like literature or musical works (Wollheim 1968, Levinson 1980, Thomasson 2006). Even though these entities are typically inscribed in tangible/physical media – a text printed on paper, for instance – many philosophers deem these artifacts abstract. As expounded in Chapter 3, this is the main conclusion of the so-called *Physical Object Hypothesis* about repeatable artworks. Simply put, the idea is that repeatable entities/artworks are abstract because they cannot be identified with a specific copy or token in which they are inscribed. However, in contrast to a platonic conception of abstract objects (which somehow exist independently of us and can be discovered), the artifactual view maintains that (at least some) abstract objects depend on agency and can be created.

Two angles render the abstract artifact view relevant to the study of computer programs. First, and more superficially, one may associate abstract artifacts with programs because the terms 'abstract' and 'artifacts' seem to resemble already

[158] If there is a mismatch between intention and structure, an object may be inadequate for a designated task – a wooden toothpick, for instance, likely won't be an adequate replacement for one's car key.

familiar features. Secondly, and more substantially, one may take the analogy to literary and musical works seriously (CONTU report, Faulkner & Runde 2010; Irmak 2012). As we have already seen, both conceptions have been greatly compared to computer programs (and in Chapter 3, I developed this idea more formally through the so-called Problem of Creation). On the one hand, the analogy of computer programs to literary works was one of the main features of what I referred to as Symbolic View. On the other hand, the analogy of computer programs to musical works has been used extensively by legal practitioners to argue for the copyrightability of software (commissioner Hersey also used both cases to express his dissent with the final CONTU report).

Note that this framing is not merely based on armchair philosophy but is well anchored in the computing community. For instance, in his 1975 book *The Mythical Man-Month* – a seminal text in software development – Frederick Brooks observed that, like the poet, the programmer engages in a creative endeavor and is

> "only slightly removed from pure-thought stuff. He builds his castles in the air, from air, creating by exertion of the imagination […] One types the correct incantation on a keyboard, and a display screen comes to life, showing things that never were nor could be." (Brooks 1975, 7)

And as noted by Tedre (2015, 157), it was a widespread opinion that computing was a craft or art of making programs in the early days of computing. So if programs are indeed pure creations of the mind and if their making is essentially an art or craft, one may reasonably draw the conclusion that they seem to be some kind of abstract artifacts.

Let me wrap up this section by mentioning the key metaphysical implications of the Artifactual View. Given the different philosophical frameworks that underpin the notions of technical and abstract artifacts, it is wise to discuss them one by one. I start by looking at appropriation of technical artifacts into computing.

As already mentioned previously, attempting to extend the technical artifacts view to computing is welcome, for it takes the engineering perspective, as expressed in Denning et al. (1989) or Tedre (2015) seriously. I will discuss some of the details and potential caveats of this approach in the last subsection of this appendix in connection with the work of Lando et al. (2007) and Turner (2014, 2018). Taking stock, we should keep the following points in mind:

(a) Notably, the Artifact View is compatible with many of the other clusters presented so far (technical artifacts are physical objects; depending on one's favored position with regards to the philosophy of mathematics, one may think that mathematical objects are also abstract artifacts; the abstract

artifacts view is complementary to the view that programs are symbol structures).

(b) Accepting the Artifact View entails that programs stand in a particular ontological dependence relation to humans (or other epistemic agents). (see e.g., Duncan 2014)

(c) Depending on which framework of the Artifact View one subscribes to, one may 'inherit' normative features (e.g., due to artifactual functions that come with several conceptions of artifacts) that tell us when an artifact is malfunctioning, and so on. In the realm of computing, normative features are required to make sense of 'miscomputation' (and that the execution of programs can go wrong; see e.g., Fresco & Primiero (2013) and Tucker (2018)).

A.5 The Neural View

There is a long and rich (philosophical) tradition to conceive of the mind as a machine (Boden 2006). With the advent of electronic computing machines, it did not take long until ideas about the computer and the brain became mutually entangled. This shaped both the perception of what kinds of things computers and brains are, thus having implications for the understanding of computer programs. Let me elucidate two main developments.

The Computer as Brain

The first main development is about the public anthropomorphization of the newly emerging electronic computing technology from the 1930s onwards. Increasingly, new computing devices seemed to solve calculations which were previously reserved to only the human mind. Accordingly, the new technology became attributed with human qualities, especially with those of the mind and brain. As Martin explains in her article 'The Myth of the Awesome thinking machine,' it was "[…] the press [who] consistently used exciting imagery and metaphors to describe early computers." (Martin 1993, 130). Analyzing newspapers, she concludes that it was rather mainstream media journalists than the early computing pioneers that spread the use of expressions like 'electronic brain' or 'intelligent machine.'[159]

While using such sensational anthropomorphic labels occurred primarily in the 1950s and 1960s and subsequently influenced large parts of the (American) population, the myth of the awesome thinking machine lost its bite when minicomputers started affecting a more widespread workforce (and users

[159] Von Neumann, Turing, and Vanevar Bush were prominent exceptions. Whereas the Von Neumann wrote of machine elements as neurons, Bush stated that machines performed "repetitive thought" (Martin 1993).

realized that they were not thinking by themselves). However, today, we live through another public hype cycle fueled by recent developments in AI applications. Especially deep neural networks (DNN)[160] – as can already be inferred from their name – are suggestive in resembling the neural structure of the human brain.

Floridi's & Nobre's (2024) recent article casts light on the general anthropomorphization of machines. In particular, they argue that AI's technical vocabulary is deeply entangled with biological and human terms. Cases in point, among others, are 'machine learning,' 'memory,' 'synapses,' and 'hallucinations.' As such, these metaphors may influence our way of thinking about computing machines since whenever technical terms are transferred between disciplines, they carry additional baggage and implications due to their original context.[161]

The Brain as Computer

The second main development however is less superficial and arguably had longer lasting philosophical influence. Roughly put, here the direction of influence worked the other way around – the computational metaphor was applied to the mind and brain. Contrary to the early widespread public conception to see computers as brains, the view to see our neural apparatus as some sort of computing system unfolded less straight forward. One of the first to argue that cognition is the product of computation in the sense of the formal notion of *computability* defined by logicians like Alan Turing were arguably McCulloch & Pitts (1943).[162] In his recent book on neural mechanisms, Piccinini foreshadows some of the potentially far-reaching consequences of this theory for our understanding of computation:

> "McCulloch and Pitts's theory was not the only source of modern computational theories of cognition. But McCulloch and Pitts's use of computation to describe neural functions, together with their proposal to explain cognitive phenomena directly in terms of neural computations, contributed to a large shift in the use of computation that occurred around the middle of the twentieth century. Before 1943, computing was thought of as one human activity among others (e.g., cooking, walking, or talking). After 1943, computing could be thought of as, in a sense, all that humans did. Under McCulloch and Pitts's theory, any neural network could be described as performing a computation. In the sense in which McCulloch–Pitts nets compute, and to the extent that McCulloch–Pitts nets are a

[160] Neural networks are an array of connected signal-processing units known as 'neurons.' Usually, they consist of (i) input units that receive signals from the environment, (ii) output units that send outputs to the environment, and (iii) hidden units that communicate solely with other units within the system.

[161] Colburn & Shute (2008) provide a more in-depth analysis of how metaphors in computer science offer a conceptual framework in which novel concepts can be embedded.

[162] See Piccinini (2004; 2020, Ch.5) for some historico-philosophical assessment of McCulloch and Pitt's theory.

> good model of the brain, every neural activity is a computation. Given that McCulloch and Pitts considered the computations of their nets to be explanations of cognitive processes and human behavior, every cognitive process was turned into a computation, and every behavior into the output of a computation." Piccinini (2020, 124)

While overtly highlighting the impact of 'historical firsts' makes many professional historians chuckle, it is true that the early 1950s saw a surge of interest in formalisms that likened the mind to a computer. Following Penn (2020), one may point out the examples of Herbert Simon's development of complex information processing and heuristic programming, John McCarthy and Marvin Minsky's development of artificial intelligence, and Frank Rosenblatt's development of machine learning. Although, as Penn argues, the approaches to simulate aspects of cognition varied among these computing pioneers, they should nevertheless be regarded as a single intellectual project of reducing "epistemology to code."

Besides, other disciplines picked up on the idea to couch the brain in computational terms. Take, for instance, the seminal textbook *Cognitive Psychology* by Neisser, which massively contributed to putting the same-named field on the map as a discipline in its own right. In the introduction, he draws parallels between humans and computers. He writes that the "task of a psychologist trying to understand human cognition is analogous to that of a man trying to discover how a computer has been programmed. In particular, if the program seems to store and reuse information, he would like to know by what 'routines' or 'procedures' this is done." (Neisser 1967, 6). Today, it is still common practice in cognitive science and adjacent fields to view the internal states in the cognitive system that provide information about the environment and guide behavior as mental representations undergoing computations (i.e., transformations that obey computational rules).

Parallel to the emergence of disciplines like artificial intelligence, cybernetics, and Cognitive Psychology, the idea of viewing the brain as a computer also gained traction in philosophical circles in a family of views nowadays referred to as *Computational Theory of Mind* (CTM).[163] In the 1960s, philosophers like Hilary Putnam (1967) proposed a form of functionalism couched in computational terms that was supposed to supersede its rivals, namely behaviorism and identity theory.

While the first versions of the CTM maintained that minds perform computations similarly to Turing Machines, subsequent proponents of the view, like Fodor (1981), focused on the symbols manipulated under computational

[163] See (Rescorla 2020) for an accessible survey.

rules. (Fodor is sometimes said to have revived the 'language of thought' idea). In brief, the proposal argues that to fully understand cognitive abilities, we need to consider how syntactic operations work on language-like symbolic structures and the digital computational procedures that transform them.

In the 1980s, *connectionism* was revived as an alternative to the classical CTM.[164] This "Golden Age of Connectionism" (Buckner & Garson 2019) was underpinned by a culmination of theoretical refinements, especially the notion of back-propagation in artificial neural networks was paramount (Rummelhart et al. 1986). In a nutshell, connectionism's main difference is that it draws from neurophysiology rather than computability theory. This paradigm relies on computational models and neural networks that differ significantly from abstract machines such as TMs.

Finally, I will shed some light on the philosophical impact of the Neural View on our metaphysical understanding of computer programs. What are the most vital implications?

First, the Neural View paves the way for naturalized conceptions of computation beyond neural activity ('Why stop with the brain?'). Taken to the extreme, this pattern of reasoning opens the floodgates to so-called pancomputationalism (viz., the idea that everything computes). Pancomputationalism is a conception that comes in various flavors[165] and is often considered to trivialize the notion of computation (this phenomenon is frequently referred to as *triviality arguments* (Sprevak 2018)). However, since for proponents of the CTM, it is a *conditio sine qua non* to have a mind-independent notion of physical computation, a lot of ink has been spilled to work out the nature of physical computation precisely. For the sake of brevity, I will not deep-dive into the intricate details. Anyhow, to make a long story short, this development brought forward philosophical debates about the nature of physical computation[166] and gave rise to a host of issues:

(a) Suppose computation is no longer just a human activity but a mind-independent process occurring in various natural systems. In that case, we may ask whether there are also 'natural computer programs.'

---

[164] The connectionist project started in the 1940s (including, e.g., McCulloch and Pitts' work) and attracted considerable attention by the 1960s. However, major concerns about the connectionist modeling techniques led to a decline in research interest and funding.

[165] Pancomputationalist ideas, while often associated with the Neural View, are not limited to it. For instance, computing pioneer Zuse (1969) proposed that all processes in the universe are computational. Today, many philosophers of computing distinguish between limited, unlimited, and ontic pancomputationalism, each with its own unique characteristics. For a comprehensive analysis of these distinctions, see Anderson & Piccinini (2024).

[166] I refer the interested reader to my second and fourth chapters and the following appendix to learn more about the subject.

(b) If one subscribes to some version of the CTM, then physical *computation is no longer based on cognition but the basis for cognition*. This potentially raises issues of circularity in the sense that the sequence of computations carried out by the execution of humanly produced computer programs is dependent on prior neural computation.

Second, to date, the Neural View has consequences on the way in which we conduct our discussions within the various brain sciences, computing, and computationally heavy fields (for instance 'machine *learning*'). Moreover, the 'Computer as Brain' and 'Brain as Computer' views arguably continue to exert a reciprocal influence on each other.

A.6 State of the Art

At first stab, the list of clusters I just presented leaves us with a fragmented picture of computer programs' ontological status – such fragmentation results in conceptual difficulties, mutual misunderstandings, and category mistakes. Tensions arise, such as how qua abstract causally efficacious object programs are simultaneously executable entities that do have real-world effects. Similarly, we may wonder whether we should classify programs as (material) technical artifacts or abstract ones (or whether there might even be naturalized programs). What to make of these rival analyses of the ontological status of computer programs? Are they all of these things, some of them, or a novel *sui generis* entity that requires the revision of our standard metaphysical frameworks?

Some investigators have sketched approaches to answering questions like these in a philosophically satisfying way. In this *State of the Art* section, I briefly yet critically review them chronologically.

The dual nature view & linguistic refinements

In his article *Three Myths of Computer Science*, Moor (1978) provides one of the earliest analyses of the situation by stating that computer programs can be understood on physical and symbolic levels. As we will see, the duality theme will reoccur for many subsequent authors. The way in which Moor tries to explain the duality is by appealing to linguistic confusion. In particular, he writes that extensionally inadequate definitions (being simultaneously too permissive and too narrow) are the root of the problem. To remedy the situation, he proposes a revised definition, viz., a computer program is a set of instructions that a computer can follow to perform an activity. Unfortunately, Moor's attempt at linguistic reform does not completely clarify what kinds of things programs are after all, either.

Everything is a program

In a more radical attempt, Suber (1988) concludes that everything is software:[167] "Hardware, in short, is also software, but only because everything is." Suber (1988, 102). This verdict hinges on equating software with patterns per se and assuming that all kinds of *prima facie* nonsensical patterns could, in principle, retroactively be turned into programming languages. Overall, the idea to reduce software to everything there is flies in the face of our shared beliefs that some things implement computer programs and others don't; Suber's account is, therefore, extensionally inadequate. Notably, his kind of reasoning bears striking similarities to a family of pancomputationalist accounts (e.g., Searle 1990) and ought to be rejected for the same reasons.[168]

A complete metaphysical overhaul

In Smith's (1998) *On the Origin of Objects*, we find yet another far-reaching stratagem on how to tackle the issue. His book is the summit of extensive investigations on the metaphysical foundations of computer science, AI, and cognitive science. As such, the first chapter contains a few pages on the ontological status of programs. Similar to Moor, he first points towards linguistic confusions by claiming that the vocabulary of computing is somewhat vague; since computing as a discipline is relatively young, the field employed metaphors and concepts from other disciplines:

> "Given the intellectual origins of computer science, it is no surprise that much of our present-day computational vocabulary was lifted from the study of logic, mathematics, and formal languages." (Smith 1998, 33)

He believes that the initial conceptual borrowing obscured the "true nature of the computational situation" (ibid.). In his view, programs are best understood through a tripartite distinction of a *program text*, a *program process*, and some (external) *subject matter* the computation is about. (N.b. that Smith's conceptualization of the textual and process aspects essentially aligns with Moor's dual nature view). However, what is truly unique about the account is that it calls for an alternative metaphysical framework beyond the realm of computing ("*it is not just the ontology of computation that is at stake; it is the nature of ontology itself*." (Smith 1998, 42)). While these claims are potentially far reaching, it is well beyond the scope of this survey section to give a fully-fledged critique of Smith's entire attempt (i.e., the 'middle distance' approach) to overhaul the way we ought to conduct metaphysics.

---

[167] Suber uses 'software' and 'computer program' interchangeably (Suber 1988, 94).

[168] As previously mentioned, so-called triviality arguments typically render pancomputationalist positions unplausible; see Anderson & Piccinini (2024) for an in-depth analysis of the topic.

Embracing the Dual Nature View: Concrete Abstractions

At the end of the millennium, Colburn (1999) also picked up the dual nature theme: According to him, programs are comprised of what he calls a *medium of description* (which is usually text) and a *medium of execution* (which is typically a switch setting). However, instead of trying to overcome the issue through linguistic reform (like Moor), he tried to embrace the seemingly opposing features, arguing that together they form *concrete abstractions*.[169]

As such, "the duality inherent in a 'concrete abstraction' crosses metaphysical categories" (Colburn 1999, 10). To make sense of the ontological status of concrete abstractions, he appeals to the mind/body problem as an analogy. Notably, he employs this strategy not because he believes programs are like persons/minds, but for the taxonomy of solutions, this philosophical discourse offers on overcoming an apparent ontological mismatch. Going through the solution space, he contends that

> "[…] the pre-established harmony thesis is well suited for explaining the high correlation between computational processes described abstractly in formal language and machine processes bouncing electrons around in a semiconducting medium." (ibid., 17)

Colburn suggests that, in the absence of a divine entity, programmers can play a role in establishing a harmonic correspondence between the abstract and the concrete.[170] However, his conclusion is rather superficial; what exactly this *relation* between such ontologically distinct categories amounts to remains vague at best.[171]

The dual nature view & linguistic refinement – attempt no. 2

Like their predecessors, Eden & Turner (2007) start from the dual nature assumption about the metaphysical nature of programs. In particular, they identify two sub-categories of the term 'program,' viz., *program-script* and *program-process*. As such, their distinction resembles that of Smith and differs from the one employed by Moor and Colburn. While the latter appeals to the static sub-branch of the Physical View, Eden and Turner mention the dynamic one (i.e., because the program-process is the execution of the program script). Moreover, the authors attempt to unscramble the situation by introducing a

[169] Colburn adopted the notion of 'concrete abstractions' from an undergraduate textbook (Hailperin et al. 1999).

[170] The idea of pre-established harmony goes back to Leibniz, which is, roughly, the thesis that there is no causal mind-body interaction but just a relationship of harmony or parallelism (Kulstad & Carlin 2020).

[171] Throughout my thesis, I argue that we should understand this relation as computational implementation.

refined definition of program script as a well-formed expression based on a Turing-complete programming language.

However, despite these differences, it is paramount to highlight that the authors also appeal to a concretization relation (similar to Colburn's 'pre-established harmony thesis') that is supposed to link the program-script to the corresponding program process. In particular, the authors describe concretization as "a process during which an entity or entities of one category are synthesized (come into being) from entities of a more abstract category." Although this notion is intriguing and already has more substance than Colburn's 'pre-established harmony thesis,' it does not fully clarify the nature of this relationship.

Clarifying definitions and the dual nature through Formal Ontologies

Lando et al. (2007) developed a new domain-specific ontology of programs and software called COPS (Core Ontology of Programs and Software) using high-level formal ontologies as a template.[172] COPS classifies computer programs as abstract documents to enable a computing system to process information. More concretely, computer programs are said to have a dual nature with two elements, branching into static entities and their execution (similar to Eden & Turner (2007)). As previous participants in the debate, the authors shed light on the relation between these elements (of the duality) and suggest further refinements of the term 'program.'

When examining the relation between these elements, the authors identify artefactual functions (cf. Artifact View) as their linkage. Specifically, they define such a function as the ability of an instrument to perform an activity (assigned by agents to endurants) in a perdurant.[173] This conceptualization characterizes an artifact as an endurant with an assigned function and gives rise to the notion of computer programs as *Artefacts of Computation*. I will return to potential issues with this conception when discussing a very similar approach in the work of Turner shortly. When further considering the static side of programs, they suggest distinguishing files (inscribed in a medium), computer language expressions (expressed in a Turing-complete language), data types, and algorithms (conceptualizations representing the semantics of programs). In sum,

[172] The authors' work is underpinned by the DOLCE and I&DA core ontology. DOLCE, a 'foundational' ontology, comprises abstract concepts that generalize ideas in different knowledge domains. Under philosophically grounded principles, DOLCE's domain – that of Particulars – splits into four subdomains.

[173] In context of the here employed framework 'endurants' can be understood as entities that are wholly present at any time at which they exist; 'perdurants' can be understood as entities that happen in time, e.g., events or actions.

programs are thus both computer language expressions and 'artefacts of computation.'

Programs as Abstract Artifacts

In his article *Software is an Abstract Artifact* Irmak (2012) rejects the dual nature view and suggests that programs are abstract artifacts.[174] In a nutshell, he identifies programs as notational entities and argues that they are subject to the same considerations as works of literature and musical works (i.e., so-called repeatable artworks; cf. Notational and Artifact View): Although programs lack spatial properties (as they cannot be identified with any particular implementation), they have temporal properties. As a 'creation of the mind,' programs are artifacts; they are the end product of a laborious process and start to exist at some point in time. Due to their temporality, they can cease to exist when all their copies are destroyed (or nobody is around to remember the underlying source code and algorithms).

While Irmak's innovative idea to draw the analogy to the ontology of art opened up a new avenue to investigate the metaphysical nature, his account is partially incomplete. The crux is that his work remains silent about the way in which programs qua abstract artifacts relate to real-world systems (i.e., it does not tell us anything about computational implementation).

Unscrambling the Software/Hardware distinction through Formal Ontologies

Similarly to the work of Lando et al., Duncan (2014) employed a ready-made formal ontology (the so-called Basic Formal Ontology (BFO)) in his dissertation to elucidate computational notions such as 'software,' 'hardware,' and 'artifacts.' In due course, he argues that we can unscramble the notorious software/hardware dichotomy (i.e., in this case, the dual nature view)[175] by thinking of programs as ontologically dependent entities, while computational hardware is ontologically independent artifact.[176, 177] Regarding software, he states

> "A software program, similar to a novel, is a generically dependent entity. A particular software program does not depend on a particular independent entity

[174] Irmak told me, in private communication, that he used 'software' and 'programs' interchangeably.

[175] It is paramount to note that one should be cautious to simply equate the software/hardware dichotomy with the dual nature view. While in this case, the two overlap, a different understanding of 'software' may change the situation.

[176] Like many authors, Duncan uses 'software' and 'program' interchangeably.

[177] Following the BFO-framework (Arp et al. 2015), Duncan differs between two types of dependent entities: *specifically dependent* and *generically dependent* ones. While the former depend upon a particular bearer to exist (and only as long as that particular entity exists), the latter exist as long as they are borne by some entity (i.e., they do not depend upon a specific bearer).

> (such as a particular DVD or flash drive) in order to exist. Rather, a software program exists as long as it is borne by some independent entity. For example, if you destroy my DVD of Microsoft Word, Microsoft Word (the software program) does not cease to exist." (Duncan 2014, 38)

Engaging in conceptual engineering, one can find, among others, such ambitious novel concepts like *programming language expression*, *computational requirement specification*, *computational information entity*, and *computational plan specification* (where each is rigorously defined in terms of his upper-level ontology). To cut a long story short, a software program is then defined as "a computational plan specification in which the instructions are specified using programming language expressions." (ibid., 133).

When explaining how to relate these entities to hardware, Duncan appeals to a *tool function* (the notion strongly resembles the characterizations of teleological function endorsed by many philosophers of technology; noteworthily, he also introduced the notion of *computational artifact* (ibid., 140)). Accordingly, hardware 'concretizes' programs when the former bears the (teleological) function that is realized in some computational planned process (ibid., 136-140). (Alas, this characterization couches the concretization relation in the unanalyzed terms of 'bearing' and 'realizing').

Ultimately, the success of such linguistic reform and conceptual adaptation in such a heterogeneous field as computing is somewhat doubtful (to the best of my knowledge, the novel concepts have yet to be widely adopted).

A requirement engineering perspective: An abstract information artifact view

Wang et al. (2014a; 2014b)[178] and Wang (2016) aim to refine further the abstract artifact proposal about programs (made by Irmak (2012)). Specifically, their work sheds light on programs' identity criteria against the backdrop of software changes (i.e., code changes). In a nutshell, the question is how specific programs can keep their identity despite code changes. By methodologically relying on the DOLCE ontology (similar to Lando et al. 2007), the authors identify and precisify programs as *abstract information artifacts*. Generally speaking, there is a wide variety of abstract information artifacts; see e.g. (Sanfilippo 2021) for a survey. However, in contrast to other information artifacts, which

> "[…] directly refer to the objects in the world (so that executing a recipe or a law implies a manipulation of objects in the world), software programs refer to virtual variables in a machine, whose manipulation inside the machine affects the outside world in an indirect way" Wang (2016, 63)

[178] Although the two papers have different names and are published in different venues, the content (i.e., the overall argument) is essentially the same.

To then further distill what is peculiar about programs, they draw from software engineering and requirements engineering literature. This strategy allows them to distinguish between software and hardware (i.e., the dual nature view), the underlying hardware, and different 'software artifact' features (code, program, software system, and software product).

Programs are Computational Artifacts

By transposing the conception of 'technical artifact' to computing, Raymond Turner popularized the term *computational artifact*.[179] As arguably one of the most prolific investigators in the philosophy of computer science, the development of 'computational artifacts' is the culmination of a longer lasting research program into various central notions of computing (Turner 2011; Turner 2014; Turner 2018). Arguably, Turner's concept borrowing was supposed to kill multiple birds with one stone:

1. First, the manoeuver accounts for software engineering. As per Turner, programs are artifacts because they are intentionally created objects.
2. Second, the notion is arguably supposed to explain away the longstanding issue of the dual-nature view of programs. In brief, the idea is that we can couch the dual-nature view in terms of the function-structure pair ingrained in the notion of technical artifacts. This move resembles the idea of framing implementation in terms of artifactual functions we have seen in (Lando et al. 2007) and (Duncan 2014).
3. Third, it allows us to account for the normative dimensions of computing. Previously, I already mentioned the verificationist debate and the importance of miscomputation. Having teleological functions at our disposal facilitates normative judgments about computational systems.

Given the scope and different problems Turner's computational artifact proposal is able to address, it is undoubtedly one of the most sophisticated accounts on the market.

However, there is one potentially serious caveat with appropriating technical artifacts to the realm of computer science; the issue is expressed in Kroes's monograph *Technical Artefacts: Creations of Mind and Matter*, where he warns us that

[179] One should not mistake Turner's computational artifacts with Suchman's notion of computational artifacts presented in (Suchman 1987). Remarkably, Tuner and Lando et al. (2007)'s notion of 'Artefacts of Computation' and the idea of using artifactual functions to bridge the gap of the dual nature view seems to have been developed independently. As far as I can tell, Lando et al. only referenced some of Turner's work that preceded the development of his computational artifacts notion. Likewise, I did not find a reference to Lando et al. in Turner's (2014; 2018).

> " […] software programs fall outside the scope of this book. I consider software programs to be 'incomplete' technical artefacts; only in combination with the appropriate hardware that executes software programs are they able to fulfil their technical function." Kroes (2012, fn. 4)

When further developing Kroes' worries, we may pose the question of whether the dual nature view of programs (i.e., the abstract-concrete dichotomy) adequately is accounted for by the function-structure duality of technical artifacts.[180] The crux of the matter is that Turner's account presumes that the notion of physical structure can also be understood in the abstract (like a set-theoretic one).[181]

A Phenomenological Perspective

In his 2019 dissertation, Geisse (2019) takes a phenomenological approach, drawing on Kant's *Critique of Pure Reason* and Husserl's phenomenological perspective to define the term 'program' based on human experience. Subsequently, he examines programs through the properties assigned to them in perception processes. This method results in a multi-dimensional characterization of programs, distinguishing them as (i) physical, (ii) syntactic entities, (iii) semantic entities, and (iv) embedded entities dependent on other entities. Geisse argues that this differentiation of object forms (i)-(iv) allows for greater precision in using the term and elucidating their interrelationships. However, since the focus of his work is primarily on phenomenological aspects of programs -- and not on their metaphysical nature -- dissolving the dual nature view (or, in fact, the occurrence of four categories (i)-(iv)) is not the primary concern. Accordingly, there is no worked-out solution to the problem of implementation.

Programs have a stratified ontology

In several of his works, Primiero picks up the themes of computational artifacts and the corresponding idea of using a function-structure pair to characterize them. However, instead of appealing to the standardly evoked dual ontology of technical/computational artifacts, his work advocates an even more fine-grained classification – a so-called layered or stratified ontology Primiero (2016; 2020). What is the motivation for this move?

---

[180] In an online session of the *Histoire et Philosophie de l'informatique* (HEPIC) seminar (a joint seminar of the University of Lille and University of Paris 1) I attended on October 30, 2022, Maarten Fraanssen (a member of the Dutch research group that developed the notion of 'technical artifacts' in the early 2000s) re-iterated the concern that the dual nature of view of technical artifacts does not transpose to the dual nature view of computer programs.

[181] N.b., similar kinds of worries would apply to other accounts (e.g., by Lando et al. (2007) or Duncan 2014)) that also make use of the notion of teleological function.

As described in the introduction (cf. Ch. 1, especially the section on epistemic abstraction), level-views have been pervasive throughout the history of computer science. In the section on the *Notational View*, for instance, we have seen that abstracting away from the machine licensed programmers to develop new programming environments that omitted cumbersome machine details. As per Primiero, computational artifacts are, therefore, subject to a form of *levelism*:

i. **Intention:** the request of epistemic agents (typically, the customers, users, and other stakeholders involved in software development projects) to define and solve a specific computational problem.
ii. **Specification:** the formal version of the request to solve the computational problem; it provides (formal) constraints on the programs' operations.
iii. **Algorithm:** the procedure providing a solution to the proposed computational problem in line with the specification requirements.
iv. **High-level programming language instructions:** a symbolic implementation of the proposed algorithm in a high-level language (the source code).
v. **Assembly/machine code operations:** typically, the machine cannot directly execute the source code; it is translated (e.g., by a compiler) into assembly code and subsequently assembled in machine code operations**.**
vi. **Execution:** the physical level (the execution LoA) is where the program runs, i.e., where the computer architecture executes the instructions.

While such a level view is certainly not entirely new in computing, what is paramount to note about Primiero's proposal (and what distinguishes it from previous versions) is the claim that it has ontological commitments. Drawing on the notion of *levels of abstraction* (LoA) developed by Floridi (2008; 2011, Ch. 3), Primiero argues that each LoA contains a corresponding pair of epistemological and *ontological* domains (a so-called EO-construct). Accordingly, computational artifacts are ontologically stratified or layered entities in the sense that they are composed of various LoA with different degrees of abstractness. This view, in other words, diverges from the traditional abstract-concrete dichotomy since it requires us to buy into a metaphysical framework that allows for multiple different notions of what it means to be 'abstract.'

Appendix B: Why we should think of computational implementation as a three-place relation

In the main body of the text, I occasionally consider an issue related to implementation which I keep referring to as *Bridging Problem*. So far, the relevant literature on physical computation has discussed the Bridging Problem[182] and the Problem of Implementation [183] separately. This appendix clarifies how the Bridging Problem affects our understanding of the metaphysical assumptions underpinning computational implementation.

While the Bridging Problem deals with the unexpected accurate applicability of mathematics to the physical world, the Problem of Implementation seeks to provide an account of physical computation by establishing an adequate correspondence between abstract logico-mathematical states and physical states. As I will demonstrate, the Problem of Implementation is not just a problem in its own right but an instance of the Bridging Problem. This view has potentially far-reaching implications for our understanding of physical computation because it allows us to apply insights from the philosophy of applied mathematics to computing that were hitherto neglected.

The most significant upshot of this framing is that we should no longer think of implementation as a simple dyadic relation between an abstract model of computation and a physical computing system. The problem with the dyadic-view (i.e., a mind independent one-to-one relation) is that it stands at odds with most approaches to solving the Bridging Problem, which suggest that the math-world relation does not hold by itself but requires a third element – the stipulations and descriptive practices of epistemic agents. Ergo, we should also understand the implementation relation as a three-place relation, where the relata are abstract computational states, physical states, and epistemic agents and their stipulations. This 'three-place conclusion,' which diverges from many of the most prominent accounts of physical computation, redefines our traditional

[182] Wigner (1960), Steiner (1998), French (2000), Wilson (2000), Colyvan (2001), Grattan-Guiness (2008), Pincock 2004, 2009, 2012, Batterman (2010), Fillion (2012), Bueno & Colyvan (2011), Nguyen & Frigg (2021), Vos (2022).

[183] Some key sources that deal with (parts of) the Problem of Implementation are Putnam (1989), Searle (1990), Churchland & Sejnowski (1992), Chalmers (1996), Copeland (1996), Scheutz (1999), Shagrir (2001; 2018; 2022), Klein (2008), Piccinini (2007; 2015), Ladyman (2009), Milkowski (2013), Fresco (2014), Horseman et al. (2014), Fletcher (2018), Dewhurst (2018), Mollo (2018), Sprevak (2010), Rescorla (2013; 2014), Lee (2020), Curtis-Trudel (2022).

understanding of implementation as a completely naturalized phenomenon hinging on a two-place relation.

The structure of the appendix is as follows. In Section 1, I briefly remind us about the main issues underlying the Bridging Problem, which serves as the starting point of our discussion. In Section 2, I shift gears, introducing the Problem of Implementation from the discourse of physical computation and sketch its most prominent solutions. In Section 3, I introduce the Bridging Problem of Applied Mathematics and its candidate solutions in more detail. In Section 4, I adapt the Problem of Implementation to the context of applied mathematics and argue that it is a particular instance of the Bridging Problem. Lastly, in Section 5, I offer some discussion and concluding remarks.

B.1 The Bridging Problem

One of the central questions of (the philosophy of) mathematics has been the seemingly miraculous accurate applicability of mathematics to the empirical sciences. This question, which has captivated scholars for centuries, was perhaps most notably revived by Wigner (1960) when he challenged us to explain the remarkable usefulness of mathematics in science. Considering its long history, the issue is known under many names (e.g., Application Problem) and may comprise several different (albeit related) problems under the same umbrella (Steiner 1998, Fillion 2012).

For instance, some investigators picked up Wigner's theme and tried to demystify the 'unreasonableness' of mathematics' applicability. Grattan-Guiness (2008), for example, suggests that mathematics is so useful for science because many of its formalisms has been motivated by science. In a different manner, Wenmakers (2016) argues that the phenomenon is due to selection effects such as a selection bias that overtly focuses on the rare success of applications but not their ubiquitous failures. Yet others pointed out a semantic problem about 'mixed statements' (Steiner 1998, 13-23, Colyvan 2001, fn.4, Pincock 2004, Fillion 2012). Here, one may wonder about the truth conditions for statements like "there are seven apples on the table" (Steiner 1998, 16) or "the gravitational acceleration is 9.81 m/s^2" (Pincock 2004, 137), where mathematical and physical terminology is mixed (i.e., they occur in the same sentence). While indeed interesting, the semantic problem of applying mathematics will not be the central topic of this appendix.

Instead, I will focus on another issue that has to do with the widespread mathematization of modern science. Given that so many mathematical formalisms accurately describe, explain, and predict empirical phenomena, we suspect an underlying coupling (a math-world relation) that enables such

knowledge gaining. It is widely accepted that here we confront a metaphysical problem of applications stemming "from a gap between mathematics and the world" (Steiner 1998, 19). Intuitively, the mathematical universe shares structural similarities with certain parts of the physical world.

To avoid potential conflations and misunderstandings with the different issues associated with the application of mathematics, I decided to employ the name 'Bridging Problem' (BP) throughout my thesis.[184] This choice, while a matter of taste, highlights the crux of the matter best – the metaphysical problem of bridging the gap between the two fundamentally different ontological domains of abstract mathematical entities and physical objects. At a first stab, we thus may narrow down the problem statement to

BP: *How does the mathematical relate to the physical?*

Solving the 'Bridging Problem' is paramount to understanding the 'model-world relation' featured in the discourse about scientific representation and adjacent fields as the philosophical literature on computer simulations. As I will show, attempting to answer the problem is not only central to many of today's philosophical debates but also besets the physical computation discourse.

B.2 The Problem of Implementation

Let me briefly remind us about the Problem of Implementation (which I already presented more formally in Chapter 2) by describing it in general terms. Subsequently, I introduce the approaches that emerged from trying to solve the issue and argue that all the leading contenders (mapping accounts, semantic accounts, mechanistic accounts) are descendants of the so-called simple mapping account (SMA). This result will be key for framing computational implementation in terms of the BP.

B.2.1 The Main Rationale

Computation is methodologically divided (Curtis-Trudel 2022). On the one hand, we may study computation in the abstract realm of logico-mathematical formalism like Turing Machines (TM), recursive functions, etc. In general, such computational formalisms are definable in a large variety of ways. In Chapter 2, I explained that the computer science literature's two main kinds of computational formalisms are (i) programming languages and (ii) abstract machine models. Following my earlier convention, I use the term 'model of computation' *M*c for both. To put a long story short, such models of computation are logico-mathematical formalisms that encode an abstract sequence of

[184] The particular name 'Bridging Problem' was coined by Contessa (2010b).

computations through a programming language, a machine table, a transition function, and so on.

On the other hand, computations take place in the real world. While the formal theory of computation is a well-established branch of mathematics/theoretical computer science, developing an account that specifies when a physical system implements computations proves challenging. Even though many models of computation allude to a machine metaphor, these theoretical models are divorced from the ongoings in real-world devices, i.e., they do not tell us which real-world systems perform which computations. What makes an abstract M_C stand in relation to genuine physical computing systems – and not to other systems like rocks or hurricanes – is an open question and is commonly referred to as the *Problem of Implementation*.

Roughly put, the common idea of solving the issue is by alluding to a special kind of correspondence or mapping that bridges the gap between abstract computational and physical states. As we have seen, translating this seemingly simple idea into formal terms resulted in the Simple Mapping Account (SMA) (see Chapter 2, sect. §2.3). However, many philosophers have claimed that under the regime of the SMA implementing physical computation would be trivial since virtually any physical computation turns out to implement computations. For most philosophers, such unlimited pancomputationalism is unplausible. Henceforth, the quest began for devising a theory of computational implementation equipped with extensional adequacy concerning paradigmatic computing systems (like computers and brains).

B.2.2 Further Refinements: The Physical Computation Landscape

Providing an answer to the Problem of Implementation – and thus developing an account of physical computation – is paramount for such disciplines as the foundation of computer science, AI, robotics, and cognitive science. Accordingly, solving this issue has received considerable attention and brought forward a host of accounts of physical computation (see Piccinini & Maley (2021) for an overview). Although the resulting options purport to look like a wide range of options on the surface, the physical computation landscape is somewhat deceiving since all accounts are a species of the SMA. In what follows, I will untangle this genealogy by analyzing the three most prominent candidate solutions – extended mapping accounts, semantic accounts, and mechanistic accounts. Mainly, I will show that, despite their different branding, these three leading contenders still adhere to the SMA's strategy of defining physical computation as the relation between two relata (a model of computation and a putative physical computing system).

i. Extended mapping accounts

The threat of unlimited pancomputationalism/triviality arguments has raised concerns about the limitations of the simple mapping account (SMA) in distinguishing genuine from overly permissive mappings. So-called extended mapping accounts (EMAs) seek to address this issue by a common strategy. EMAs aim to filter out spurious mappings by using different forms of counterfactual dependencies to differentiate between the abstract domain of model computation and the physical domain (Chalmers (1995), Copeland (1996), Scheutz (1999), Klein (2008)).[185] For instance, Chalmers states

> "A physical system implements a given computation when there exists a grouping of physical states of the system into state-types and a one-to-one mapping from formal states of the computation to physical state-types, such that formal states related by an abstract state-transition relation are mapped onto physical state-types related by a corresponding *causal* state-transition function." Chalmers (1995, 392; emphasis added)

This approach requires that the formal set-theoretic structure of M_C only maps to the causal/counterfactual structure of a physical system P. Instead of merely considering one execution trace, implementation occurs if counterfactual computations are satisfied. However, importantly EMAs still maintain the fundamental idea of SMA (i.e., a mapping between the abstract domain of the model of computation and the physical domain).

ii. Semantic accounts

Historically, so-called semantic accounts emerged separately from the considerations of the SMA and EMAs. Fodor's slogan "There is no computation without representation." (Fodor 1975; Pylyshyn 1984), captures the essence of semantic accounts. There are two common reasons for embracing semantic accounts: First, the semantic account is consistent with the views of various brain sciences and the Computational Theory of Mind, which suggests that cognition relies on our brains performing computations (Rescorla 2020). Since brain states are believed to have content and process information, computational states must do the same. The semantic view turns this into a doctrine, and accordingly, computational states must have 'aboutness' and carry external content or meaning. Additionally, the computational states of our computing devices often manipulate meaningful symbols.

[185] This view is extrapolated from the causal, dispositional, and counterfactual views. Since on most accounts of causality causal claims support counterfactuals, one may simply lump all these different views together under the label of EMAs. See Piccinini (2015, Ch. 2) for a similar line of reasoning.

It is worth noting that simply having meaning attached to a physical state is not sufficient for a theory of implementation. Otherwise, any random symbol manipulation - like a dog chewing on a newspaper - could be seen as executing computations (Milkowski 2013, 42). Therefore, for semantic accounts to be extensionally adequate, they must incorporate some rule-following that typically boils down to following the the EMAs' specifications. This includes the mapping between physical processes and abstract computational states, as well as the ability of the mapping to support counterfactual state transitions.

The second important reason for embracing the semantic account is of more recent origin. Shagrir (2020; 2022) refers to this as the *master argument* for the semantic account. Accordingly, philosophers of computing like Shagrir (2001), Sprevak (2010) have argued that semantic properties circumvent computational indeterminacy. Philosophers often discuss the issue using the example of logical duals such as AND- or OR-gates (e.g., Papayannopoulos et al. 2022). In cases where more than one mapping between physical and logical structure is possible, the SMA or an EMA alone cannot determine which computation is implemented. Put simply the claim of the master argument goes that an EMA with a semantic condition mounted on top does not suffer from this defect. Notably, contemporary semantic accounts thus also rely on structural mappings.

iii. Mechanistic accounts

Lastly, the mechanistic accounts result from espousing neo-mechanist conceptions and applying them to computation. According to the 'consensus conception,' "a mechanism for a phenomenon consists of entities and activities in such a way that they are responsible for the phenomenon.", (Illari and Williamson 2012, 120). This view has been put into action by philosophers such as Piccinini (2007; 2015), Milkowski (2013), Mollo (2018), and Dewhurst (2018) when characterizing physical computing systems as (functional) mechanisms.

Piccinini's influential account, for instance, states that physical computation is the processing of vehicles by a teleo-functional mechanism according to medium-independent rules (cf. Piccinini 2015, 120-21). The notion of teleological function emphasizes that systems may only execute computations if it is their function. 'Function' here should be understood in the sense of aim or purpose and not in the formal mathematical sense. Specifically, it is a computing system's function to manipulate vehicles following a rule, i.e., computing a mathematical function mapping from inputs (and possibly internal states of *P*) to outputs (Piccinini 2015, 121).[186] The term 'vehicle' denotes a variable or a state that can

[186] Importantly, a rule that is mapping input to outputs should not be conflated with mapping *f* between the abstract computational and physical domain, as depicted in Fig. 1. Rather, the

take different values and change over time. This term is simply another expression for what I previously referred to as grouped-together physical state(s) (cf. quote by Chalmers). Moreover, since computational descriptions/the rules of physical computing systems are abstract, they can define computation independently of the media that implement them – they are 'medium independent.'

Upon first glance, the terminology and concepts used by computational mechanists such as Piccinini to explain physical computation may seem different from the SMA, EMA, and semantic accounts, which could lead to interpretative difficulties. However, once familiar with the terminology, it becomes clear that the mechanistic account also relies on the main idea underlying SMA. So, to better understand how it works, let me break down Piccinini's reasoning into three steps:

1. **Selection**: To show how a concrete mechanism may perform computations (say, of a TM), computational mechanists first need to find concrete counterparts to the formal notions of a finite set of states q that are part of a finite set of symbols Σ (Piccinini 2015, 127). In Piccinini's account, the thus selected concrete counterparts are called 'digits.'
2. **Labeling**: Subsequently, "[o]nce microstates are grouped into digits, they can be given abstract labels such as '0' and '1'." (Piccinini 2015, 128). The purpose of the labeling operation is to prepare the mechanism's concrete components to align with the standard nomenclature of set symbols/alphabet Σ of our M_C.
3. **Imputation to a computational rule:** Lastly, one may generalize the previous step, such that "[g]iven their special functional characteristics, digits can be labeled by letters and strings of digits by strings of letters. As a consequence, the same formal operations and rules that define mathematically defined computations over strings of letters can be used to characterize concrete computations over strings of digits.", (Piccinini 2015, 132). This imputation process to a computational rule is a crucial aspect of understanding physical computation mechanistically, for it glues together the abstract M_C with a concrete counterpart.

Digits (i.e., physical state types p_i) have been labeled with symbols – presumably, letters contained in the alphabet of the model of computation M_C of our choice – so that the mathematical rule/function δ can be used to characterize the concrete computations performed by the system P. The labeling scheme thus establishes

mechanistic framework's notion of rule can be understood as the transitions leading from one computational state to another (the horizontal arrows in the top-span of the picture).

a reference (builds a bridge) to a specific abstract model of computation/computational formalism.

However, together steps 1 to 3 are just a different way of saying that physical states correspond to computational states specified by a computational rule. In other words, the correspondence established by the labeling process still boils down to a mapping relation advocated by the SMA and its descendants. That is why, despite the different terminology/emphasis, mechanistic accounts are descendants of the SMA and also rely on mappings.

Taking stock of this section, we need to keep in mind that although many physical computation accounts are conceptually much richer than the SMA (and sometimes employ different terminology), they still depend on dyadic mappings between a material system and mathematical structures defined by a model of computation. Therefore, all the physical computation accounts discussed above are outgrowths or extensions of the SMA rather than offering a conceptual alternative wholesale. The result that virtually all theories of implementation hinge on mappings is significant because it licenses us to transpose the insights from the BP debate in the philosophy of applied mathematics to how we flesh out computational implementation. In order to turn this conceptual borrowing into a fruitful maneuver, we need to familiarize ourselves with the relevant insights of BP.

B.3 Charting the landscape of the Bridging Problem's Solutions

At the beginning of this Appendix, I briefly introduced the BP by stating that it concerns the relation between mathematics and the physical world. However, I have yet to discuss how philosophers of applied mathematics have tried to make sense of this relation. Hence, this section overviews today's most widely embraced answers. Although they may differ considerably on certain aspects, they all maintain that epistemic agents are necessary for the solution. What is the underlying reason for this commonality? I will start depicting the core idea (§3.1) before canvassing the landscape of more fine-grained solutions (§3.2).

B.3.1 The Core Idea: The Mapping Account

With the advent of structuralism in the 20th century (e.g., Balzer et al. 1987), potential solutions to BP, typically, started to follow a particular strategy: most of the contemporary approaches (re)formulate the issue in terms of structures and the relations between them. On this construal, there is a structural mapping (i.e., a morphism) between the mathematical structure and parts of the physical

world. Due to the central notion of mappings, Pincock (2004) dubbed this the Mapping Account,[187]

> **Mapping Account***: The gap between the mathematical M and the physical P is bridged by a structure-preserving mapping f: $S_P \rightarrow S_M$ between two corresponding structures S_M and S_P.*

The purported advantage of this view is that both 'structure' S and 'mapping' f are understood as precisely definable mathematical objects. To further understand the merits and limitations of the Mapping Account, it is therefore instructive to consider what philosophers of mathematics intend to convey by these terms.

On the one hand, we can understand a mathematical structure as a composite of a family of objects, nodes, or positions (in a domain D) and a set of relations R_i among them. This definition is widely accepted and has, for instance, been discussed in works by Resnik (1997) and Shapiro (1997). Expressed more formally, we can define S as $S = \langle D,\ R_1, R_2, \dots \rangle$. Mappings between structures *f:* $S_P \rightarrow S_M$, on the other hand**,** may come in various flavors and can be understood as an isomorphism (van Fraassen 1980; Suppes 2002), partial isomorphism (Da Costa and French 2003) homomorphism, etc. between mathematical structures. Although it is interesting to ponder the (dis)advantages of each option, the crucial point for this chapter's argument is *how* these mappings come about (and not which one will be the preferred one in this or that scenario). Importantly, all these mapping conceptions require the physical system to display a particular structure to establish any math-world correspondence.

To give an brief example, we may state, for instance that two structures $S = \langle D,\ R_1, R_2, \dots \rangle$ and $S^* = \langle D^*,\ R_1^*, R_2^*, \dots \rangle$ are isomorphic *iff* there is a function f from the domain D of S to the domain D^* of S^* that is total, one-one, and onto and such that for any relation R_i in S and R_i^* in S^* and for all $x_1, \dots, x_n$ in D, $R_i(x_1, \dots, x_n)$ *iff* $R_i^*(f(x_1), \dots, f(x_n))$ (Pincock 2012, 27).

However, there is a fundamental problem with the mapping account: Physical systems must have structures for morphisms to be well defined as "[i]somorphism is a relation that holds between two structures and not between a structure and a piece of the real world per se." Frigg (2006, 55). The issue is that physical systems are concrete entities existing in physical reality, not mathematical structures. Recently, Vos (2022) aptly called this discrepancy the ontological-mismatch problem. Claiming that a set-theoretic structure is

[187] Interestingly, the Mapping Account bears a strikingly similar name to the Simple Mapping Account (SMA) in the physical computation discourse– a happenstance (?) that already seems to hint at a deeper connection.

isomorphic to a physical object $f: P \rightarrow S_M$, thus leads to committing a category mistake. Without accounting for how a physical system P obtains a unique structure S_P, the idea underlying the mapping account remains in jeopardy. What is thus required to solve BP is an account of how physical systems obtain a unique structure.

B.3.2 The Proposals on how Physical Systems obtain structure

In this section, I will review existing accounts and attempt to make sense of the mapping account in light of the call for structure. As we will see, providing a definitive answer is not easy to come by and I won't try to solve this daunting task myself. Instead, I will merely rule out the most implausible candidate solutions and demonstrate that the remaining contenders are unified in their advocacy for a three-place relation. Accordingly, they necessitate the stipulations and descriptive practices to explain how physical systems obtain a structure S_P.

i. The world is fundamentally mathematical

One way to address the ontological mismatch problem is by arguing that it does not actually exist. There are generally two approaches to this.[188] Philosophers who claim that the world is fundamentally mathematical adopt one of them. Tegmark (2008), for instance, maintains his so-called *Mathematical Universe Hypothesis*, according to which our physical worlds turns out to be an abstract mathematical structure. If this were correct, the ontological mismatch would seemingly dissolve.

However, as discussed by Nguyen and Frigg (2021, 5949f), whether or not the world is *fundamentally mathematical* is irrelevant since we are more often than not interested in the math-world relation beyond the fundamental level. So, even if Tegmark's account was correct at the fundamental level, it fails to address how to identify structures at the non-fundamental level of, say, four apples on a table.

For all that, there is a related but less radical-sounding proposal that suggests that the physical world somehow exhibits some unique structure S. The seemingly intuitive idea that physical systems somehow just bear or instantiate a unique structure is highly contentious for two reasons. First, there are multiple

[188] Another way to respond to the ontological mismatch problem lies in the opposite spectrum of the previous proposal: Nominalism. According to nominalists, mathematical objects, relations, and structures either do not exist at all or at least do not exist as abstract objects (Bueno 2020). As a result, there is no ontological mismatch (and hence no problem) because there are no mathematical objects to begin with, or at least not the kind that would require bridging across different ontological domains. As argued by Colyvan (2001) (see also Pincock (2004, 139-140)), BP appears to be independent of any particular philosophy of mathematics. Despite trying to do away with mathematical objects, nominalism faces the challenge of explaining why using mathematics in scientific practice is so effective.

ways of picking out a set of objects to form the domain of a structure of a system. As Bueno & Colyvan (2011, 347) aptly remind us, "the world does not come equipped with a set of objects (or nodes or positions)" that constitute a domain. Second, even if we manage to establish the objects in *D*, the relations *R*i are not fixed, possibly yielding different structures. As Psillo's aptly puts it,

> "[…] the structure of a domain is a relative notion. It depends on, and varies with, the properties and relations that characterize the domain. A domain has no inherent structure unless some properties and relations are imposed on it. Or, two classes A and B may be structured by relations R and R' respectively in such a way that they are isomorphic, but they R may be structured by relations Q and Q' in such a way that they are not isomorphic." (Psillos 2006, 562)

These considerations – that there is not 'the' structure of a system are not new – are often referred to as *Newman's Objection*.[189] According to this objection, the mapping between a set-theoretic structure and a physical object might be trivialized since the latter fails to display a unique or privileged structure. The claim goes that one can always gerrymander the domain *D* and the relations R_i in such a way that they match an arbitrary structure *S*. For given there are enough *n* basic objects $x_1,\ldots, x_n$ in the system (such that the cardinality of the corresponding domain is sufficiently large), then "[…] a system of relations between its members can be found having any assigned structure compatible with the cardinal number of [S]", (Newman 1928, 140), where *S* is an arbitrary structure.[190]

To summarize, mathematical-universe-style proposals – according to which the physical world boils down to mathematical structure – and adjacent realist stances about unique structure-bearing systems do not adequately solve BP. In a nutshell, scientists wishing to link a mathematical formalism with a concrete system encounter significant underdetermination problems. That is why the dyadic view fails. In order to generate a suited structure, we need to engage in two tasks. First, we must specify the domain D of objects x_i; second, we must determine the relations *R*i between these objects.

ii. Inferentialist proposal

Having recognized the challenges associated with a 'pure structuralist' solution, Bueno & Colyvan (2011) put forth a proposal that amends the mapping account. While their approach is still partially structural, it also accounts for practical and context-sensitive factors when utilizing mathematics. Particularly, their inferentialist conception of applied mathematics requires three steps:

[189] See Ainsworth (2009) for a more in-depth problem analysis and potential answers.
[190] Note that this is essentially what the SMA was criticized for by Copeland (1996).

1. **Immersion:** Bueno and Colyvan's first step in their process is called *immersion*. It involves creating a connection between the real-world scenario and a mathematical structure that is convenient to work with.
2. **Derivation**: The second step, *derivation*, involves deriving consequences from the mathematical structure obtained in the immersion step.
3. **Interpretation:** The third step is referred to as *interpretation*. To establish an interpretation, a mapping from the mathematical structure to the initial empirical setup is necessary. N.b., this procedure goes in the opposite direction of the immersion step, but the authors claim that this mapping does not have to be simply the inverse of the mapping used in the immersion step, though it may be in some cases. Using a mapping different from the one used in the immersion steps is unproblematic as long as the mappings in question are definable for appropriate domains.

The takeaway is that the inferentialist proposal is a mind-dependent notions because it amends the original mapping account by alluding to epistemic agents.

iii. Abstraction based proposal

According to the abstraction-based proposal, the required structure S_P is generated through epistemic abstraction. Nguyen and Frigg (2021) formalized this idea in their 'extensional abstraction account.' Simply put, the idea is this: Given a physical system *P*, one may obtain a unique structure through some structure-generating description, a so-called extensional description. These extensional descriptions can be created by hiding specific physical information about *P* such that it no longer explicitly refers to physical magnitudes.

Expressed formally, epistemic agents must decide on a domain *D* and their respective elements to generate a structure *S* of a physical object. Next, they need to determine the relations between those elements. Once certain choices about the elements in domain *D* and their relations *R*i are agreed upon and held fixed, the extensional description that is thus generated gives rise to a purely set-theoretical structure S_P.

To render the idea less theoretical, it is helpful to consider an example from Frigg (2006, 57-58), where he demonstrates how to generate a structure of a methane molecule (CH4). To obtain a structure that aligns with our previous definition, we need a domain *D* of objects and relations *R*i that are abstracted from the molecule. One way to obtain a structure is to 'abstract away' the physical properties of the atoms and solely focus on the molecule's shape. Since the four hydrogen atoms of the molecule form a regular tetrahedron (with the carbon atom in the center), it would be one choice to pick the edges as objects and the

vertices as relations. This way, one obtains a *unique* structure with four objects (the tetrahedron's edges) and six relations (the connections between the edges).

Although the example is simple, it exemplifies a pattern of reasoning that underlies all abstraction-based structure generation accounts. Worthy of mention in this regard is that structure generation is parasitic on human agents making certain choices about a suitable domain and its relations, typically informed by our scientific practice. (we could have swapped the tetrahedron's vertices and edges with respect to objects and relations and thus obtained a different structure).

In sum, my brief review has shown that there is a genuine philosophical problem associated with the applicability of mathematics. As we have seen, the critical issue of BP is the ontological mismatch between mathematical and physical properties. Most recent accounts adhere to variants of structuralism to bridge the gap between these different domains. However, since more than 'pure' structuralism is needed to solve the problem, various amended schemata have been brought forward. They are all contingent on human activity and as such, they are adherents of mind-dependent three-place views. As far as my research has revealed, the broader implications of this three-place insight on theories of computational implementation have not been previously explored. It is now high time to bring the different results of the previous sections together.

B. 4 Synthesizing the Problems: A new perspective

Although the literature on physical computation has brought forward an impressive number of contributions, the focus on the metaphysical nature of the implementation relation has usually taken the backseat. To change that, I will now propose a way to think about the issue more deeply from a newfangled perspective. In particular, it will become evident that the Problem of Implementation is a species of the BP. Given that employing computational concepts thus falls under the broader practice of applying mathematics, we can anticipate analogous rationales to apply in the realm of computation. [191]

To recap, since the methodology of computing is bifurcated, it faces the issue of an ontological mismatch (the Problem of Implementation). In response, today's most prominent theories of implementation (SMA, EMA, semantic accounts, mechanistic accounts) rest on the assumption that there is a mapping between an empirical setup and some abstract logic-mathematical model of

[191] Admittedly, one can already find traces of this reasoning (i.e., structuralism about physical computation) in the literature (Milkowski 2011, 360), Rescorla (2013; 2014), Doherty & Dewhurst (2022), Curtis-Trudel (2022). However, to the best of my knowledge, the conclusion to view physical computation as a three-placed based on BP has not been drawn before based on this framing; the current proposal thus goes beyond the proposals above.

computation *M*c. Similarly, albeit more globally, the philosophy of applied mathematics also confronts an ontological mismatch – the BP. Here, the most plausible proposals regarding BP likewise rest on the assumption that a mapping constitutes the math-world relation. These parallels are not mere coincidences but the result of a systematic relationship. Uncovering this conjunction requires a closer look into the differences in the frameworks' scopes.

Whereas proposals to solve BP are designed to be *generally* applicable, solutions to the Problem of Implementation are limited to the applicability of computability theory. Based on this comparison, we can deduce that the Problem of Implementation is a specific instance of the Bridging problem. Accepting this conclusion hinges on accepting that the implementation relation is a unique species of the math-world relation. In fact, I have already discussed an instance of this: As we have previously seen in section §2, we can understand TMs as *bona fide* mathematical entities, defined as a four tuple TM = (Q, Σ, m, δ). Importantly, how this mathematical/computational formalism applies to parts of the physical realm is the same metaphysical problem of the ontological mismatch between the mathematical and the physical, investigators like Steiner, Pincock, and others have pointed out (cf. Sect. §B.1).

While both lines of research propose that the ontological mismatch can be overcome by adhering to structure-preserving mappings, most solutions to the Problem of Implementation typically do not further elucidate the metaphysical nature of the mappings they employ.

It is here where my framing of the Problem of Implementation in terms of the solutions of BP becomes a unique selling point because, in contrast, the literature on the application of mathematics *has* thoroughly explored the metaphysical commitments that are needed for a mapping view. We can thus enrich our current understanding of computational implementation by appealing to the insights of the philosophical literature on applied mathematics.

Although no solution to BP has emerged as the definitive one, they pull in the same direction: All corresponding analyses share the idea that the mapping relation is not a brute fact. Instead, the math-world relation necessitates a third relatum – an agent responsible for establishing the mapping f and determining which set-theoretic structures are supposed to be related. On this three-place relation view, physical computation is thus a mind-dependent conception because a system may only be computing due to human activity. The reason is that, strictly speaking, the implementation relation is not reducible to f alone. Despite being a necessary component, f is insufficient, for we also require at least some minimal stipulations by an epistemic agent or community.

While extended investigation is required to determine which solution(s) is/are the correct one(s), I do not need to endorse any particular proposals for enriching our understanding of the metaphysical nature of physical computation. The result that the BP's most promising solutions reject dyadic relations and instead advocate a three-place relation, is a novel and significant contribution to today's physical computation landscape. In sum, my four-step argument thus leads to an *upgraded view* of computational implementation, as shown in Fig. B.2.

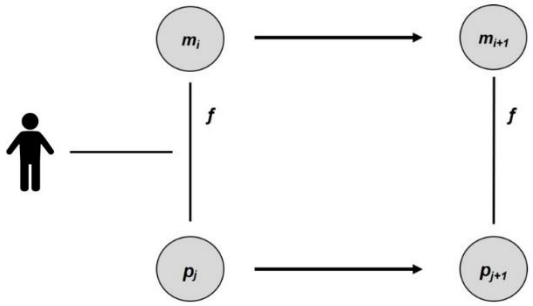

Fig. B.2: The *upgraded view* three-place view of computational implementation (cf. nomenclature to standard view depicted in Fig. 2.1 in Chapter 2). Accordingly, physical computation is not reducible to f as it necessitates an agential component.

As such, the conclusion that computational implementation should be seen as a three-place relation is a considerable advancement of the state of the art that may ignite a hefty research program, inspiring further exploration and discovery in the field.

B.5 Discussion and Conclusion

I argued that computational implementation is an instance of the more general metaphysical problem of how mathematics applies to the physical world (namely, the Bridging Problem). Based on a four-step argument, I have argued that computing systems do not implement computations all by themselves but that implementation should be thought of as a three-place relation. Having introduced the BP as a metaphysical issue of applied mathematics (1), I argued that the Problem of Implementation is of the same species (2). (3) Since most contemporary solutions to the BP advocate for a three-place relation, solutions to the Problem of Implementation should follow suit. Accordingly (4), the designers' or users' intentions and descriptive practices are indispensable for relating physical states to abstract computational states. The takeaway is that we should couch computational implementation as a three-place relation.

Now, why does the here-advocated 'three-place result' matter? Is all of this, not mere pedantry? Several important lessons can be learned from subsuming the Problem of Implementation under BP. First, if my reasoning above is correct, the face of discussion about physical computation will have to change towards focusing on the metaphysical nature of the implementation relation. (One may add this criterion to the list of desiderata for a good theory of computational implementation). Those who presume that computational implementation is a one-to-one correspondence between physical states and the computational states as described by its model must, in defense of their view, explain how this correspondence comes about. Absent such an explanation, the claim that computational implementation is a mind-independent phenomenon remains at odds with today's dominant solutions to BP on the market. The onus is thus on the advocates of naturalizable accounts of physical computation to respond to these tensions. Until such time, it would be wise to assume that computational implementation is a mind-dependent three-place relation. This is a pressing matter because naturalized accounts of computation are deemed essential for the cognitive science project; if computational implementation is a three-place relation, it cannot be the basis for cognition.

Second, I want to clarify that the viewpoint I defended should not be misconstrued as support for overly liberal interpretational theories of implementation. The sticking point has been that some interpretational accounts claim that one can transform arbitrary objects into computers through mere stipulation. However, we do not need to endorse such a conclusion. My argument does not deny or negate that mind-independent requirements are paramount for successful implementation. Here, it is worth noting to similar conclusions in the literature of scientific representation, where philosophers suggest that informational and functional theories are complementary (see e.g., Chakravartty 2010). For instance, computers are useful because of specific causal regularities and counterfactual dependencies. What I contest is the notion that the implementation relation can be wholly naturalized. If correct, this result has potentially far-reaching consequences for our understanding of physical computation.

Lastly (and related to the previous point), this line of research may pave the way to foster synergies between the philosophical discourses on physical computing and scientific models. Recall that philosophers of science have successfully integrated the structuralist insights surrounding BP into characterizations of model-world relations (viz., suggesting that it is a three-place relation). Facing related problems holds the promise of similar solutions. For instance, one promising line of future research could try to frame the

implementation relation in terms of scientific representation accounts (see Chapter 4 for such an account).

Appendix C : Synopsis détaillé en français

Il s'agit d'un résumé d'environ 10 % de ma thèse, « Mind the Gap », qui explore le statut ontologique des programmes informatiques. Les pages suivantes résument la structure argumentative et les conclusions des chapitres de ma thèse.

Chapitre 1 - Introduction

L'introduction (chapitre 1) prépare le terrain en motivant le sujet, en fournissant des préliminaires méthodologiques et en attirant l'attention sur les obstacles conceptuels potentiels. L'affirmation principale est que la notion de « programme informatique » renvoie à de nombreuses choses liées et (historiquement) instables - il s'agit d'un polysème (un mot avec de multiples significations liées). C'est pourquoi, en l'absence d'une caractérisation stable, les études précédentes ont placé les programmes informatiques dans presque toutes les catégories ontologiques disponibles. Afin d'éviter les erreurs précédentes, la thèse poursuit une stratégie indirecte : l'approche méthodologique principale consiste à élucider les relations entre les différents éléments identifiés par le terme « programme » dans le contexte de l'ontologie la plus 'modeste' et la plus répandue sur le marché : la distinction abstrait-concret (un système à deux catégories). Plus concrètement, la thèse vise à élucider ce qui peut être placé dans le système à deux catégories et comment les entités qui s'y trouvent sont liées.

1.1 Prologue

Ma thèse commence par un scénario hypothétique basé sur des événements réels et demande au lecteur d'imaginer qu'il est un jeune avocat spécialisé dans les brevets au début des années 1970. Au cours des années précédentes, les entreprises de logiciels sont apparues comme une industrie de plus en plus puissante. Par la suite, les années 70 ont été le théâtre d'une série de procès novateurs concernant la protection juridique des « logiciels ». Les programmes informatiques devaient-ils être soumis au droit des brevets ou au droit d'auteur ? La question s'est avérée si délicate que le gouvernement américain a même créé une commission spéciale (CONTU) pour trancher le débat (Con Diaz 2019). Cependant, même après quatre ans, les spécialistes ne sont pas parvenus à un consensus. J'ai utilisé cet exemple partiellement fictif comme point de départ, car ces questions juridiques sont sous-tendues par des questions ontologiques/métaphysiques.

1.2 Concrétisation du problème : préliminaires et diagnostic

Dans la section §1.2. j'affine et je précise ma question de recherche initiale. En particulier, je prends trois mesures pour planter le décor et clarifier le problème lié à mon sujet de recherche. Premièrement, je discute de la pertinence globale de cette entreprise pour les philosophes et les informaticiens. Deuxièmement et troisièmement, j'explique les deux éléments constitutifs – l'ontologie et les programmes informatiques - qui définissent la question principale de la recherche.

En ce qui concerne la pertinence, j'affirme que mon sujet de recherche est important pour les raisons suivantes : d'une part, il y a une motivation philosophique/métaphysique. *À première vue*, les programmes sont des entités déroutantes qui semblent échapper aux caractérisations standard et qui peuvent donc soulever des questions métaphysiques intéressantes. D'autre part, la clarification de leur statut ontologique pourrait avoir des conséquences pour l'informatique (les praticiens pourraient éviter des erreurs de catégorie).

En ce qui concerne l'ontologie, je partaique ce que des métaphysiciens contemporains (anglophones) comme Fine (2017) et Hofweber (2016) appellent la « métaphysique traditionnelle ». Notamment, je distingue les questions ontologiques primaires (POQ) et les questions ontologiques secondaires (SOQ), c'est-à-dire,

> **(POQ) :** L'ontologie pose la question de *ce qui existe.*
>
> **(SOQ) :** La métaphysique proprement dite étudie la *nature de ce qui existe.*

À la suite de ces considérations, nous devons spécifier un système de catégories ontologiques (c'est-à-dire un système de classification structuré des types d'êtres qui devrait fournir un inventaire complet de ce qui existe). L'avantage d'un système préconçu est qu'il nous permet de porter des jugements métaphysiques cohérents sur toutes les sortes d'entités sous l'examen de la SOQ. De même, il devient le motif central de la thèse de préciser l'appartenance des programmes informatiques à l'une des catégories du système.

Cependant, il existe plusieurs systèmes ontologiques concurrents. Compte tenu de cette diversité et de l'absence de consensus, nous sommes confrontés au problème suivant

> **Problème I :** *Sélection d'un système de catégories approprié* : *approche fragmentaire ou systématique* ?

Enfin, en ce qui concerne le terme « programme informatique », il est essentiel que nous ayons la même signification que les autres participants au débat (sous

peine de malentendus). Afin d'affiner le type d'entités désignées par le terme « programme informatique », je me penche sur la *Begriffgeschichte* du terme. Le mot « program » (ou « programme » en orthographe britannique) trouve ses racines dans le mot grec προγραφιν, composé de προ (« avant » ou « pré ») et de γραφιν (« écrire ») Grier (1996, 51). En tant que tel, le terme n'est pas né dans un contexte informatique, mais a subi des transformations considérables tout au long de l'histoire. De Mol & Bullynck (2021, 36) expliquent que le mot était employé de manière générique pour désigner une série planifiée d'actions ou d'événements futurs. (Aujourd'hui, nous utilisons encore des expressions telles que programmes de télévision, de théâtre ou de radio).

Comme beaucoup d'autres premières occurrences historiques, les premières occurrences de « programme » dans un contexte informatique sont controversées. Toutefois, à partir des années 1950, l'informatique a évolué vers la fiabilité, la production de masse et la normalisation, et les tentatives se sont multipliées pour déterminer des pratiques normalisées et définir des termes de base tels que « programme » dans des glossaires (De Mol & Bullynck 2022). Un aspect qui allait de pair avec ce développement et cette professionnalisation précoce du domaine était que la configuration des ordinateurs était de plus en plus associée à des langages formels étroitement liés à la logique et à la linguistique (Nofre et al. 2014).

Cependant, pour faire court, la caractérisation ne s'est jamais complètement stabilisée et il serait donc erroné de considérer les programmes comme de simples textes ou entités linguistiques. En raison du pluralisme épistémique de l'informatique, de nombreuses notions centrales de l'informatique présentent une ambiguïté sémantique surprenante. Plus précisément, le terme « programme » est un polysème qui, à l'instar d'une toile, recouvre plusieurs sens différents (bien que liés). Le problème est que nous devons concevoir une stratégie qui empêche la confusion linguistique de s'insinuer dans notre enquête métaphysique, faute de quoi nous pourrions obtenir autant de réponses potentielles sur le statut ontologique des programmes informatiques qu'il y a de significations différentes cachées dans ce complexe polysémique.

Problème II : *démêler l'écheveau polysémique du terme « programme ».*

1.3 Les idées directrices du projet

Ma stratégie pour répondre aux problèmes I et II est de me concentrer explicitement sur les *relations* entre toutes les relata ontologiquement différentes qui se cachent derrière ce réseau polysémique. Pour clarifier, les relations dont je

parle sont celles qui sont apparues dans mon analyse précédente et dans la revue de la littérature, celles qui sont considérées comme responsables de la dualité/du pluralisme présumé des programmes : D'une part, il y a le domaine des objets abstraits, formels et mathématiques. D'autre part, il y a le domaine du physique, des systèmes concrets, des événements et des processus qui se déroulent dans l'espace et le temps.

Plus précisément, je pense que la notion de l'implémentation est essentielle pour comprendre comment ces entités se connectent. Lorsque je parle de « l'implémentation », je me réfère (dans un premier temps) à la relation entre différents domaines informatiques. En outre, ma thèse soutient que les agents jouent un rôle essentiel dans la médiation de l'implémentation. Je développerai ces deux idées en détail dans les chapitres suivants, mais voici un graphique décrivant la situation pour en saisir l'essentiel (Fig. C.1).

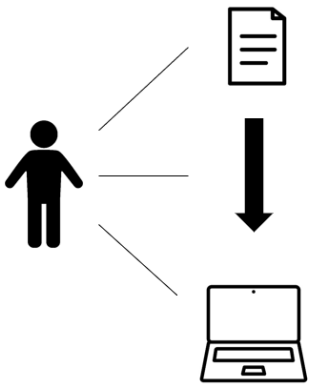

Fig. C.1 : Représentation schématique de l'idée directrice de cette thèse.

Chapitre 2 - Vers une théorie unifiée de l'**implémentation**

Le chapitre 2 identifie « l'implémentation » comme le candidat le plus prometteur pour considérer la relation entre les éléments du programme et s'en sert comme motivation pour étudier les différentes notions disponibles. En résumé, le chapitre esquisse un cadre unificateur qui intègre deux approches jusqu'ici traitées de manière largement indépendante : le type-(A), qui traite de la notion de correction par l'attribution de fonctions téléologiques à des « artefacts informatiques » à différents niveaux d'abstraction (LoA) ; et le type-(B), qui s'attache à combler la dichotomie abstrait/concret pour rendre compte de l'informatique concrète. Bien que je montre que leur champ d'application se chevauche au niveau de l'interface abstrait-physique, peu de recherches

systématiques ont été menées sur leur relation. Compte tenu des développements récents dans les discours respectifs, je soutiens que les deux comptes peuvent s'enrichir mutuellement de manière considérable en s'unifiant. Plus précisément, je soutiens que (A) et (B) peuvent être unifiés en les mettant en relation avec la machinerie conceptuelle des modèles matériels et de la représentation scientifique. Dans cette optique, les agents utilisent des systèmes informatiques putatifs comme outils épistémiques en leur imputant des fonctions mathématiques et en leur attribuant des fonctions téléologiques.

Voici comment cette entreprise se déroule en détail : Après avoir formulé quelques remarques générales sur l'implémentation dans l'introduction du chapitre (2.1.), je donne plus de détails sur l'utilisation de la notion en informatique dans la section 2.2. En particulier, pour avoir une compréhension commune de cette relation d'implémentation, il est instructif de nous rappeler les formalismes informatiques. Bien qu'ils puissent être définis d'une grande variété de façons, la littérature informatique présente généralement deux façons principales de présenter les formalismes informatiques (Turner 2018, 190) :

1. *Langages de programmation*, tels que C, Python, etc.
2. *Modèles de machines*, comme les machines de Turing (TM), les machines à états finis (FSM), etc.

Tout au long de la thèse, j'utilise le terme « modèle de calcul » pour les deux. Les modèles de calcul sont des formalismes logico-mathématiques qui nous permettent d'encoder une séquence abstraite de calculs par le biais d'un langage de programmation, d'une table de machine, d'une fonction de transition, etc. Par exemple, formellement, le concept de machine de Turing peut être caractérisé comme un quadruple $TM = (Q, \Sigma, m, \delta)$, où Q est un ensemble fini d'états q ; Σ est un ensemble fini de symboles ; m est l'état initial $m \in Q$; δ est une fonction de transition qui détermine le prochain mouvement $\delta : (Q \times \Sigma) \rightarrow (\Sigma \times \{L,R\} \times Q)$. La fonction de transition δ de la TM permet de passer d'états de calcul à des états de calcul (De Mol 2021). En d'autres termes, les fonctions de transition comme δ, les programmes informatiques écrits dans un langage de programmation, ou toute autre notion correspondante dans un *M*c théoriquement équivalent, permettent l'encodage d'une séquence de calculs. Pour qu'un système calcule, il doit mettre en oeuvre une séquence de calculs codée dans un programme/une fonction de transition spécifié(e) par un *M*c donné. En pratique, les formalismes de calcul sont souvent intégrés dans une hiérarchie de calcul spéciale composée de ce que l'on appelle des niveaux d'abstraction (« Levels of Abstraction » ; LoA) (Floridi 2008 ; Primiero 2020). En conséquence, l'application de l'implémentation en informatique est très variée. Des exemples sont « l'implémentation d'un

algorithme dans un langage de programmation de haut niveau » ou « l'implémentation d'instructions de code machine dans un ordinateur du monde réel ». Parler de différents « level » est une pratique courante dans les sciences du comptage.

Historiquement, cependant, deux notions de l'implémentation largement séparées ont été développées afin de préciser les exigences relatives à la connexion de ces différents types de niveaux : Pour faciliter la discussion, je les ai appelées l'implémentation de type (A) et l'implémentation de type (B). Il est surprenant de constater que ces deux approches ne sont pas en contact étroit l'une avec l'autre. Dans ce qui suit, je discute des détails de ces deux types d'implémentation.

2.3 Type-A

En ce qui concerne l'implémentation de type (A), on peut identifier trois approches différentes : La première est due à Rapaport (1999, 2005)

> **Implémentation en tant qu'interprétation sémantique :** Un objet est une implémentation d'un domaine syntaxique *A* dans un support *M* s'il est une interprétation sémantique d'un modèle de *A*,

Rapaport présente l'implémentation comme une interprétation sémantique. Son récit a été critiqué parce qu'il tenait la sémantique pour acquise. Turner (2018) a développé la seconde notion et a suggéré de considérer la conception suivante :

> **L'implémentation en tant que relation fonction-structure :** La relation entre la spécification (fonction) et la structure de l'artefact (informatique).

Enfin, Primiero (2020) a lui suggéré l'acception suivante :

> **L'implémentation en tant que relation de LoA :** Une l'implémentation *I* est une relation d'instanciation entre des paires composées d'une construction épistémologique *E* et d'un domaine ontologique *O* d'un artefact informatique**.**

2.4 Type -B

L'implémentation de type (B) est caractérisée par le *problème de l'implémentation*. Des philosophes comme Sprevak (2018) et Ritchie & Piccinini (2018) soutiennent que deux sous-problèmes, à savoir le problème de l'implémentation et le problème de l'application, sont à l'origine de l'implémentation.

> **COMP** Conditions de calcul d'un système physique.
>
> **IDENT** Conditions qui spécifient qu'un système de calcul implément un calcul plutôt qu'un autre.

doivent être abordés pour répondre au problème de l'implémentation. L'une des premières catégories de comptes conçues (par exemple, Putnam 1987) pour résoudre ce problème est aujourd'hui appelée « Simple Mapping Account ».

Simple Mapping Account (SMA)

1. Il existe une correspondance *f* entre les états s_j de S_C et les états mi de M_C, telle que
2. Sous *f*, les transitions d'état physique de S_C sont morphiques aux transitions d'état formel de M_C (spécifiées par δ) , de sorte que si S_C est dans l'état s_1 où $f(s_1)= m_1$, alors S_C évolue dans l'état $s2$ où $f(s2)= m_2$.

L'idée qui sous-tend le SMA est que les transitions d'état d'un *M*C doivent d'une manière ou d'une autre refléter la dynamique (transitions d'état physique) du système matériel.

Bien que le SMA soit apparemment élégant et simple, il est largement admis qu'il entraîne des conséquences indésirables, généralement qualifiées d'*arguments de trivialité*. En conséquence, le calcul physique serait banalisé puisque chaque système implémente toutes sortes de calculs.

En réponse aux arguments de trivialité et d'indétermination computationnelle, la plupart des comptes de l'implémentation de calcul physique/type (B) ont modifié le SMA en introduisant des caractéristiques supplémentaires pour traiter l'un ou l'autre, ou les deux, ainsi que ses descendants (causal/dispositionnel/contrefactuel ; mécaniste).

2.5 Juxtaposition

Après une analyse détaillée de la portée de l'implémentation des types (A) et (B), nous pouvons conclure qu'ils ne s'excluent pas mutuellement. Il existe un point de jonction dans (i) les systèmes informatiques artificiels à (ii) l'interface abstrait-physique (voir Fig. C.2). Du point de vue du type-(A), les idées du type-(B) sont pertinentes pour la mise en œuvre de programmes informatiques à l'interface abstrait-physique. A l'inverse, du point de vue du type-(B), les connaissances contenues dans le discours du type-(A) offrent une image nuancée des systèmes artificiels et des préoccupations et pratiques des informaticiens en la matière.

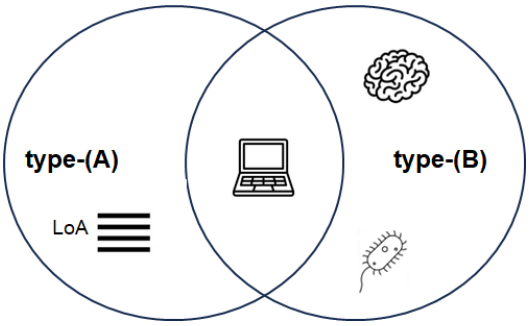

Fig. C.2: Diagramme de Venn schématique de l'intersection l'implémentation des types (A) et (B). Leurs domaines d'application se chevauchent dans (i) les dispositifs informatiques artificiels et (ii) l'interface abstraite-physique.

Cependant, malgré ce chevauchement, il n'y a eu, à ma connaissance, qu'un échange limité entre ces deux domaines de recherche. Pour faire progresser notre compréhension générale l'implémentation et en faire un effort de coopération, juxtaposons les deux différents types des implémentations en ce qui concerne leurs caractéristiques les plus saillantes : (i) la description téléologique des fonctions et (ii) les correspondances entre les niveaux. En résumé, voici ce qu'il faut retenir :

1. Téléologie : Il existe un décalage entre les hypothèses qui sous-tendent l'utilisation des fonctions téléologiques dans l'implémentation des types (A) et (B).
2. Relation de correspondance : Différentes relations de correspondance sont en jeu : d'une part, il existe des correspondances entre structures abstraites et, d'autre part, une correspondance (au niveau de l'interface abstrait-concret) entre une structure abstraite et un système physique.

2.6 UTAI

Pour remédier à cette situation, j'ai esquissé un nouveau cadre vers une théorie unifiée de l'implémentation agentielle, en abrégé UTAI. Une grande partie de ce nouveau cadre est basée sur les idées théoriques de la littérature sur les modèles scientifiques *matériels*. L'un des principaux principes de ce point de vue est qu'il permet de surmonter les différences entre les implémentations de type (A) et (B) en ce qui concerne leurs caractéristiques téléologiques et les relations de mise en correspondance. En effet, le discours sur les modèles matériels a déjà abordé avec succès les questions relatives aux correspondances entre les structures abstraites et les systèmes physiques, ainsi que les cas où les modèles en tant qu'artefacts ne fonctionnent pas correctement. La principale contribution de l'UTAI est le

développement d'une série d'études de cas clarificatrices qui suivent trois relations de dépendance différentes :

1. La *relation de dépendance (a)* met l'accent sur l'implication cruciale des agents humains dans la LoA supérieure. En tant que telle, elle rend compte des désirs, des intentions et des préoccupations pragmatiques des programmeurs et de la manière dont ils formulent leur problème informatique dans un formalisme informatique approprié.
2. La *relation de dépendance (b)* concerne le mappage *f* qui relie l'interface abstraite et physique. L'implémentation peut avoir lieu lorsque les agents proposent une description génératrice de structure (par exemple, par le biais de la « dissimulation d'informations ») et un mappage approprié reliant les domaines abstrait et concret.
3. La *relation de dépendance (c)* caractérise les interactions physiques du ou des agents humains avec le système informatique supposé. Idéalement, un système informatique est non seulement suffisamment fiable pour des exécutions répétées, mais aussi reconfigurable.

Chapitre 3 - Le problème de la création rencontre les programmes informatiques

3.1 Introduction

Selon un point de vue très répandu, les programmes informatiques ne semblent pas être des objets concrets, mais nous les considérons généralement comme des entités créées avec lesquelles il est possible d'interagir. Cela est quelque peu surprenant puisque l'orthodoxie philosophique considère que les objets abstraits ne sont pas intégrés dans la voie causale et ne peuvent pas être créés.[192] Par conséquent, un problème philosophique pressant se profile à l'horizon :

> En quoi ces programmes en tant qu'objets abstraits peuvent-ils être les produits de notre création ?

Malheureusement, cette question n'a pas reçu toute l'attention qu'elle mérite jusqu'à présent. À la lumière de ce problème, mon objectif dans ce chapitre est d'indiquer plus précisément dans quel sens nous pouvons classer les programmes informatiques en tant qu'*objets abstraits*. Pour ce faire, ma stratégie

[192] Le point de vue métaphysique standard stipule que les objets abstraits existent éternellement et ne peuvent pas être créés. La plupart des philosophes conçoivent la création comme une relation de cause à effet entre le créateur et sa création. Cependant, les objets abstraits sont inertes sur le plan causal et ne peuvent donc pas entrer dans une telle relation, de sorte que l'on ne sait pas exactement quel type de processus implique la création d'un objet abstrait (Mag Uidhir 2013, 11).

consiste à adopter ce que l'on appelle le *Problem of Creation* (PoC) de la philosophie de l'art à l'informatique.

3.2 Préparer le terrain

Avant de nous plonger dans les détails philosophiques essentiels du (PoC), je souhaite apporter quelques clarifications initiales sur les programmes. Si les programmes informatiques doivent être le sujet du (PoC), nous devons avoir une idée claire (i) de ce que l'on entend exactement par « programme », (ii) de la manière dont ils sont créés, et (iii) du sens dans lequel ils sont abstraits :

(i) Conformément à mes arguments sur la nature polysémique du terme dans l'introduction de cette thèse (Ch. 1), je ne définirai pas les programmes ici. Au lieu de cela, je m'appuierai uniquement sur ce que je considère comme un exemple paradigmatique de programme informatique écrit dans un langage de programmation de haut niveau, tel qu'illustré à la figure 3.1. Le *code source* est écrit en *C* et, et le programme utilise un *while-loop* pour trouver le plus grand diviseur commun (GCD) de deux entiers. Au lieu de s'appuyer sur une caractérisation rigoureuse de *tous les* programmes informatiques, *cet* exemple me servira de « laboratoire conceptuel » en temps voulu.

(ii) En général, l'histoire de la création des programmes se déroule comme suit : Au départ, l'objectif est de créer un programme spécifique. Cependant, dans le domaine de l'informatique, nous ne pouvons pas simplement transférer nos désirs et nos intentions à nos ordinateurs. Nous devons d'abord les traduire dans un formalisme informatique que la machine peut exécuter. Cela implique généralement d'écrire le code source dans un langage de programmation de haut niveau. Le processus de programmation comporte plusieurs étapes, est sujet aux erreurs et est souvent laborieux. Pour ne pas perdre de vue notre objectif initial, nous créons idéalement une *spécification*. Les spécifications servent de « plan » pour la production de programmes informatiques à grande échelle, qu'ils soient petits ou complexes. En d'autres termes, elles définissent ce que le programme est censé faire (Primiero 2020, 208). Il est donc essentiel de comprendre les spécifications pour comprendre le processus de création des programmes.

(iii) Pourquoi et en quoi les programmes peuvent-ils être conçus comme des objets abstraits ? Pour répondre plus précisément à cette question, je les encadre d'une manière plus formelle. En philosophie de l'art, le problème en jeu est fréquemment discuté sous le nom de *Physical Object Hypothesis* (POH). Adapté de Mag Uidhur (2013, 8, fn. 4), le raisonnement de la (POH) peut être résumé comme suit,

Hypothèse de l'objet physique (POH) :

(POH)$_1$ Il existe des oeuvres d'art.

(POH)$_2$ Les oeuvres d'art peuvent être répétées ou non.

(POH)$_3$ Les oeuvres d'art répétables ne peuvent être interprétées de manière cohérente comme des oeuvres concrètes.

(POH)$_4$ Donc, s'il existe de tels types d'oeuvres d'art, ces oeuvres d'art doivent être des objets abstraits.

(POH)$_5$ Il existe de telles oeuvres d'art.

(POH)$_6$ Il existe donc des oeuvres d'art qui sont des objets abstraits.

Qu'entend-on exactement par « oeuvres d'art reproductibles » ? Comme le décrit Levinson (1980), les philosophes débattent depuis longtemps de l'identité ou de la nature des oeuvres d'art reproductibles (par exemple, la musique et la littérature). Contrairement aux peintures et aux sculptures, ces oeuvres d'art ne peuvent pas être identifiées de manière plausible à une copie spécifique/individuelle. Non seulement elles peuvent survivre à des changements de leur support matériel, mais elles peuvent aussi avoir été réalisées avec des matériaux différents. Les oeuvres d'art répétables sont *modalement flexibles* en ce qui concerne la matière dont elles sont faites.

Par la suite, je soutiens qu'un raisonnement similaire s'applique aux programmes. En philosophie de l'informatique, il est habituel de parler de *multi-réalisabilité*. La « multiréalisabilité » est une notion influente dans la philosophie de l'esprit (Bickle 2020) et est fréquemment employée dans le discours sur l'informatique physique (Milkowski 2016). Dans le contexte du calcul, la multiréalisabilité exprime l'idée que de nombreux systèmes physiques distincts peuvent mettre en oeuvre les *mêmes* séquences de calcul.

3.3 Le problème de la création

Le problème de la création (PoC) est un problème philosophique de la philosophie de l'art qui concerne à l'origine les oeuvres littéraires, les compositions musicales et les personnages de fiction (Deutsch 1991 ; Cameron 2008 ; Irmak 2020). Pour résumer, voici comment cela se présente : Lorsque l'entité examinée est appelée *X*, le modèle de raisonnement du (PoC) prend la forme suivante :

Le problème de la création (PoC) :

X$_1$: Les X sont des objets abstraits (POH).

X$_2$: Des X sont créés.

X$_3$: Les objets abstraits ne peuvent pas être créés.

A première vue, il y a de bonnes raisons d'accepter les propositions (X_1)-(X_3) *individuellement*, car elles semblent toutes parfaitement fondées. Cependant, les trois propositions sont mutuellement *incohérentes*. Ce paradoxe a suscité un débat important pendant de nombreuses années, conduisant à la question de savoir quelle proposition de (X_1)-(X_3) nous sommes prêts à rejeter. En conséquence, trois options majeures peuvent être identifiées :

1. Platonisme
2. Nominalisme
3. Créationnisme

En résumé, chacune de ces trois positions résulte du rejet d'une des trois propositions (X_1)-(X_3) du (PoC). Je présente ci-après chacune d'entre elles en détail. Très schématiquement, les différentes positions se résument à :

(1) Platonisme : Le platonisme est le point de vue qui postule l'existence d'objets abstraits, c'est-à-dire d'entités non physiques et non mentales qui existent en dehors de l'espace et du temps. Selon le platonisme contemporain, les objets abstraits sont immuables et entièrement inertes sur le plan causal, c'est-à-dire qu'ils ne peuvent pas interagir physiquement avec d'autres objets (Balaguer 2016).

(2) Nominalisme : Cette position métaphysique rejette la proposition (X_1) en soutenant qu'un objet abstrait candidat X n'existe pas ou s'avère ne pas être abstrait du tout. En tant que tel, le nominalisme se divise en deux points de vue différents : le matérialisme et l'éliminativisme.

(3) Le créationnisme abstrait : Ce point de vue soutient qu'il est possible de créer des objets abstraits. En d'autres termes, le créationnisme adopte les points de vue (X_1) et (X_2) tout en rejetant le point de vue (X_3).

Le résultat de mon analyse est qu'aucune des trois positions philosophiques décrites précédemment n'est incohérente ou incohérente sur le plan interne - chacune d'entre elles est un point de vue défendable. Cela dit, chaque position se heurte à de sérieuses objections. L'examen de ces objections nécessite une réflexion sur des énigmes métaphysiques plus larges et plus anciennes concernant la causalité, la dichotomie abstrait-concret, la parcimonie ontologique, la paraphrase, etc. La tâche consiste maintenant à déterminer dans quelle mesure ces questions s'appliquent au domaine de l'informatique.

3.4 De l'art à l'informatique

Sur la base de mes premières conclusions, j'applique le (PoC) aux programmes informatiques et j'examine les différentes options (1)-(3) en ce qui concerne les programmes informatiques.

(1) Platonisme sur les programmes informatiques

Selon ce point de vue, les programmes sont des objets abstraits éternels que nous découvrons. Il existe différentes raisons de souscrire à ce point de vue. Selon le « point de vue indirect », on peut en venir à considérer les programmes comme des objets platoniques parce qu'on est un platonicien mathématique et qu'on pense que les programmes sont essentiellement des sortes d'objets mathématiques. Selon le « point de vue direct », on peut penser que les programmes sont des objets platoniques parce que l'on souscrit à des positions similaires concernant des entités linguistiques telles que les romans, etc.

(2) Nominalisme sur les programmes informatiques

Comme dans le cas général évoqué précédemment, on peut présenter le nominalisme sur les programmes sous deux formes principales, l'une éliminativiste et l'autre matérialiste. Cependant, jusqu'à présent, personne ne semble avoir développé ces points de vue. En conséquence, j'esquisse brièvement certains des obstacles potentiels des deux positions.

(3)Le créationnisme en matière de programmes informatiques

Considérer les programmes comme des artefacts (abstraits) a gagné en popularité parmi les chercheurs à tendance philosophique ces dernières années (Lando et al. 2007 ; Faulkner & Runde 2010 ; Irmak 2013 ; Duncan 2014 ; Turner 2011 ; 2014 ; 2018 ; Wang 2016 ; Sanfilippo 2021). Aujourd'hui, ces points de vue dominent sans doute la littérature, bien que dispersée, sur le statut ontologique des programmes informatiques. Deux conceptions populaires se distinguent.

D'une part, il y a le point de vue des artefacts informatiques (Lando et al. 2008 ; Turner 2011 ; 2014 ; 2018). D'autre part, il y a le point de vue selon lequel les programmes sont des abstractions temporelles, c'est-à-dire des artefacts abstraits (Irmak 2013).

3.5 Conclusion

Le principal enseignement de mon application du (POC) aux programmes est double. D'une part, du point de vue de la philosophie de l'informatique, mon approche nous permet de sortir des sentiers battus de la recherche métaphysique en informatique et d'offrir un nouvel angle de vue sur l'ontologie des

programmes. D'autre part, du point de vue de la métaphysique contemporaine, mon approche oriente le débat sur le statut ontologique des programmes informatiques vers un territoire philosophique plus établi. Elle montre notamment que la nature abstraite des programmes ne nécessite pas la postulation de solutions *sui generis* complètes (par exemple, une « double nature »), mais doit être formulée selon l'axe du platonisme, du nominalisme et du créationnisme. Les recherches futures devront montrer si l'un d'entre eux prendra le dessus.

Chapitre 4 - L'implémentation-as : De l'art et de la science à l'informatique

Le chapitre 4 porte sur la relation de dépendance entre les agents humains et l'informatique physique. Ce faisant, ce chapitre présente une nouvelle approche de la compréhension de l'informatique physique, appelée *implementation-as*. D'une manière générale, ma nouvelle approche est redevable à une nouvelle ligne de recherche qui a proposé de formuler l'implémentation en termes de représentation et de modélisation scientifiques. Bien que ce groupe de recherche soit encore relativement dispersé, il diffère des EMA traditionnels parce qu'il soutient que la relation de mise en correspondance *f* doit explicitement être comprise comme une forme de représentation scientifique. Cette perspective repose sur des considérations épistémologiques, métaphysiques et historiques. Plus précisément, mon cadre s'appuie sur le compte DEKI (Frigg & Nguyen 2018), un compte formalisé de la représentation scientifique fondé sur la notion de représentation en tant que de Goodman et Elgin.

4.2 Représentation scientifique et *representation-as*

Pour fournir le contexte nécessaire, je présente d'abord la notion de représentation de Goodman et Elgin - comme dans la philosophie de l'art (sect. §4.2). Dans leurs travaux sur la représentation et la modélisation scientifiques, Frigg et Nguyen se sont approprié les notions de « dénotation », d' « exemplification » et d' « imputation » de Goodman et Elgin et les ont introduites dans l'arène scientifique. En s'appuyant sur la dénotation, l'exemplification et l'imputation, et en ajoutant une quatrième exigence, qu'ils appellent « keying up », leur compte DEKI est né (le nom est un acronyme pour les quatre notions sur lesquelles il s'appuie).

Pour démontrer le bien-fondé du fonctionnement pratique de leur compte, ils utilisent un exemple concret : le MONIAC. Le nom MONIAC signifie « Monetary National Income Analog Computer » (au Royaume-Uni, la machine est également connue sous le nom de « Philips-Newlyn machine »), et il s'agit d'un

ordinateur hydraulique analogique à usage spécial utilisé pour représenter une économie nationale.

Le MONIAC étant un cas limite entre un modèle scientifique matériel et un ordinateur analogique, il constitue une passerelle idéale pour établir un lien entre la représentation scientifique et l'implémentation informatique.

4.3 De la science à l'informatique

Après ces considérations préliminaires, je me concentre sur le calcul physique (sect. §4.3). Ainsi, en utilisant les différents éléments du compte DEKI, je propose une nouvelle approche du calcul physique qui utilise un concept concret de représentation scientifique. Les résultats peuvent être résumés comme suit :

Implementation-as

La paire ordonnée $C=\langle X, \mathrm{I}\rangle$ est un dispositif de calcul, où X est un système matériel et I une interprétation. Soit P le formalisme/programme de calcul. C implémente P en tant que Z_C *si* toutes les conditions suivantes sont remplies :

(1) C désigne P.

(2) C exemplifie les propriétés Z_1 ,...,Z_n sous une interprétation $I : X \rightarrow Z_C$.

(3) C est accompagné d'un codage informatique associant l'ensemble $\{Z_1,\ldots, Z_n\}$ à un ensemble (éventuellement identique) de propriétés $\{P_1,\ldots, P_m\}$. $E\{Z_i\}=\{P_j\}$

(4) C attribue au moins une des propriétés P_1 ,..., P_m à P.

Le cadre qui en résulte est baptisé *implementation-as*, en reconnaissance de l'influence de la représentation-as de la philosophie de l'art et de la science. Cette approche est méthodologiquement différente des précédents récits de calcul physique formulés en termes génériques de représentation scientifique, comme les *L-machines* (Ladyman 2009) ou la théorie A/R (Horsman et al. 2014), parce qu'elle s'appuie sur une proposition de représentation scientifique spécifique.

4.4 Étude de cas : Machine IAS

En discutant des éléments de l'*implementation-as*, j'ai suivi Frigg et Nguyen et j'ai utilisé le MONIAC comme exemple de jouet. Cependant, pour démontrer l'utilité du nouveau compte de l'*implementation-as* au-delà de l'informatique analogique, je vais montrer comment l'appliquer au cas d'une machine informatique numérique : la machine IAS (un ordinateur numérique à programme enregistré qui a été construit entre la fin des années 1940 et le début des années 1950 à Princeton, à l'Institute of Advanced Studies). Si, à première vue, ce dispositif peut sembler un choix arbitraire, deux raisons principales en font une excellente étude de cas : premièrement, l'architecture de la machine a été très influente ; il s'agissait de l'un des premiers ordinateurs à programme stocké binaire, qui

stockait les instructions et les données dans la même mémoire. En tant que telle, elle incarne les principes architecturaux de l'*architecture de von Neumann*, qui est encore couramment utilisée aujourd'hui. L'idée est que ce qui vaut pour cette machine peut aussi valoir pour des machines similaires. Deuxièmement, bien qu'elle possède toutes les caractéristiques principales des ordinateurs numériques modernes, la machine IAS est moins complexe et plus facile à analyser.

4.5 Discussion

Il est important de noter que mon analyse a montré que ce nouveau SRA agentiel/interprétatif répondait aux critères standards évoqués (Piccinini 2015, Duwell 2021) :

Desiderata du calcul physique

(1) L'*objectivité* : La prise en compte du calcul physique doit permettre, au moins en partie, de savoir si un système implémente une fonction de calcul.

(2) *Adéquation extensionnelle* : une description adéquate du calcul devrait permettre de renvoyer correctement aux objets qui calculent sans inclure les objets qui ne calculent pas.

(3) *Explication* : Les calculs effectués par un système matériel doivent, au moins en partie, expliquer son comportement et ses capacités.

(4) *Calculs erronés* : Une conception du calcul doit permettre de rendre compte des cas de calculs erronnés..

(5) *Taxonomie* : Une description de l'informatique doit permettre de démêler les différentes capacités de calcul des différents systèmes.

Pour faire court, l'implémentation est une bonne théorie de l'implémentation informatique parce qu'elle répond aux critères (1)-(6) de manière adéquate.

Comment cette nouvelle théorie de l'implémentation informatique s'accorde-t-elle avec les autres ? Bien que l'implémentation en tant que telle et les EMA « traditionnels » partagent ces similitudes, il existe une différence essentielle entre les deux. Les EMA traditionnelles partent du principe que la relation d'implémentation est une relation à deux places entre des états physiques et des états abstraits de calcul, obtenue de manière naturaliste et indépendante de l'esprit. En revanche, les SRA plaident généralement en faveur d'une interprétation de la mise en correspondance en vertu de la représentation scientifique. Cet engagement est très différent car de nombreuses options de représentation scientifique sont des relations à trois places qui s'obtiennent *si* l'on prend en compte les agents et leurs capacités intentionnelles. C'est la raison pour laquelle de nombreux partisans de l'SRA ont soutenu qu'ils devaient être conçus

comme une théorie agentielle de l'implémentation. Le cadre l'*implémentation-as* rend cela explicite, et je soutiens que son application réussie nécessite les activités de dénotation, d'exemplification, d'encodage et d'imputation qui dépendent de l'esprit.

Le compte sémantique restreint encore les EMA en exigeant que les états de calcul soient toujours porteurs de sens ou de contenu sémantique. Dans une section précédente (3.1), j'ai discuté du lien entre mon approche et les approaches sémantiques. Les SRA et les comptes sémantiques soulignent tous deux l'importance de la représentation dans le calcul. Cependant, il existe des différences notables dans la manière dont la représentation est utilisée et comprise dans les deux cadres.

Dans le cadre de l'implémentation en tant que telle, la représentation scientifique est utilisée pour combler le fossé entre les états de calcul abstraits et les états physiques sans qu'il soit nécessaire de s'engager sur un contenu externe. En général, les SRA n'ont qu'une exigence minimale en matière de contenu : les états physiques doivent simplement être porteurs d'un contenu logico-mathématique (du modèle de calcul implémenté). Tout contenu sémantique supplémentaire ou toute signification des véhicules de calcul n'est pas pertinent pour l'application réussie des accords de reconnaissance mutuelle et donc de l'implémentation. (Toutefois, l'utilisateur du dispositif informatique peut, si nécessaire, attribuer un contenu sémantique ou une signification aux calculs). En revanche, les comptes sémantiques utilisent la représentation dans un sens plus large, où les états informatiques peuvent représenter des états de fait externes. Ce sens de la représentation est plus pertinent pour les sciences cognitives, qui partent du principe que les états du cerveau sont représentatifs.

En ce qui concerne la relation entre les conceptions de l'*implémentation-as* et les conceptions mécanistes, la question de leur lien est nuancée. Selon la version mécaniste que l'on choisit pour la comparaison, il y a différents degrés d'engagements partagés. En général, les approaches mécanistes affirment que les mécanismes fonctionnels sont au coeur de l'informatique ; les véhicules informatiques doivent être des composants d'un mécanisme. Dans sa formulation actuelle, le cadre *implementation-as* ne partage pas spécifiquement cet engagement. Cependant, même si les véhicules informatiques ne doivent pas nécessairement faire partie d'un mécanisme pour une application réussie de l'approche *implementation-as*, rien dans la formulation de mon compte n'exclut que les systèmes informatiques $C=\langle X, \mathrm{I}\rangle$ ne puissent pas être des mécanismes. En fait, les deux cas discutés précédemment - le MONIAC et la machine IAS - sont de *véritables* mécanismes. Les recherches futures devraient élucider si ce fait est

accidentel ou si une combinaison des points de vue pourrait conduire à une théorie encore plus robuste du calcul physique.

Chapitre 5 - Programmabilité physique

Ce chapitre concerne la relation de dépendance entre les programmeurs et les systèmes matériels utilisés pour l'exécution des programmes (relation de dépendance (c)). A ce titre, il présente une nouvelle notion appelée

> **Programmabilité physique** : Le degré auquel l'activité/fonction/opération/phénomène sélectionné(e) sur un automate peut être reconfiguré(e) de manière contrôlée.

Dans l'ensemble, les programmes conçus par des agents humains peuvent consister en un simple séquençage ou en des séquences très complexes d'opérations sur un support physique. Les opérations séquencées vont du son (boîtes à musique) au calcul en passant par le tissage (métiers à tisser Jacquard). Pour exécuter toute séquence d'opérations souhaitée, le système choisi doit être configuré de manière appropriée, ce qui nécessite des interactions (physiques) spécifiques : la machine doit être programmable. Malheureusement, le discours philosophique sur la programmabilité est peu abondant et largement sous-développé.

5.1 Aperçu critique de la programmabilité

Afin de développer une théorie adéquate, je commence (§5.1.) par passer en revue les quelques approches existantes de la programmabilité dues à Conrad (1988), Zenil (2010 ; 2012 ; 2013 ; 2014 ; 2015), Piccinini (2008 ; 2015), et Haigh & Priestley (2018). Bien que je soutienne que chacune d'entre elles a ses propres limites, il existe quelques points communs (voir le tableau C.1).

Sur la base de ces observations, je présente ma nouvelle alternative rigoureuse (programmabilité physique). Ensuite, je me penche sur les détails des variables contenues dans cette définition, à savoir (i) les automates matériels, (ii) les opérations sélectionnées, (iii) la reconfiguration, et (iv) le degré de programmation, et je les relie aux discours fondateurs établis dans la philosophie des sciences.

	Conrad	**Zenil**	**Piccinini**	**Haigh & Priestley**
Type de système physique	Naturel et technique	Naturel et technique	Technique et naturel (?)	Technique
Opérations	Computation	Computation	Opérations séquentielles	Opérations séquentielles
Mode de reconfiguration	Instruction	-	Mécanique, Instructions	-
Système de classement	Efficace et structurel	Mesure quantitative (information algorithmique)	Hard et soft	-

Tableau C.1: Comparaison des différentes caractéristiques des comptes de programmabilité présentés ici.

5.2 Automate matériel

Je suggère que le fait d'être physiquement programmable se limite aux « automates matériels ». On peut caractériser un automate comme

> **Automate :** Système ayant la capacité d'exécuter une série d'opérations prédéterminées (dans une certaine mesure) de manière autonome.

Il est important de ne pas confondre ces automates avec des entités logico-mathématiques abstraites telles que les machines de Turing. Ces dernières sont de véritables objets mathématiques et ne sont pas sujettes à la programmabilité physique. Les automates matériels sont plutôt des artefacts techniques.

Les artefacts techniques sont des types particuliers d'artefacts qui se caractérisent par leur « double nature » - constituée à la fois de caractéristiques fonctionnelles dépendant de l'esprit et de caractéristiques structurelles indépendantes de l'esprit (cf. Baker 2006 ; Kroes & Meijers 2006 ; Kroes 2012 ; Preston 2018, §2.3). La structure détermine ce qu'un artefact peut faire, tandis que la fonction est ce pour quoi l'artefact est censé être utilisé. En raison de cette normativité, certains chercheurs (Vermaas & Houkes 2003 ; Houkes & Vermaas 2010) ont soutenu que les fonctions techniques nécessitent une intentionnalité. En conséquence, un agent ou une communauté épistémique attribue intentionnellement une fonction à un objet dans un but spécifique.

De ce point de vue théorique, les automates matériels peuvent être considérés comme des artefacts techniques parce qu'ils sont (i) des dispositifs créés intentionnellement avec (ii) la capacité d'exécuter une séquence prédéterminée d'opérations.

5.3 Opération sélectionnée

Le fonctionnement sélectionné de ces automates matériels s'explique mieux par le cadre néo-mécaniste et sa notion de

> **Mécanisme :** « Le mécanisme d'un phénomène est constitué d'entités et d'activités telles qu'elles sont responsables du phénomène. » (Illari et Williamson 2012)

Généralement, le phénomène de niveau supérieur d'un mécanisme/système est appelé Ψ-ing de *S*, où *S* désigne le système, et Ψ-ing le phénomène correspondant. Les entités du mécanisme sont appelées X_i et leurs activités sont désignées par $\{\phi_1, \phi_2, ..., \phi_n\}$ (cf. Craver 2007 ; voir Fig. C.3).

En outre, on peut définir les *mécanismes d' « Input/Output »* (Glennan 2017, 113-116 ; ci-après dénommés « I/O ») comme une sous-classe de la définition générique des mécanismes. Selon Glennan, les mécanismes I/O sont des systèmes dont les actions ou les sorties réagissent aux entrées et peuvent être décrits par une relation fonctionnelle entre les variables d'entrée et de sortie $f(i)=o$, où *i* désigne les entrées, *o* les sorties et *f* leur relation fonctionnelle.

Un système *S* donné peut présenter plusieurs phénomènes à la fois. Lorsque nous jugeons qu'un objet présente un certain degré de programmabilité, nous le faisons généralement en ayant à l'esprit un seul phénomène spécifique *(*Ψ-ing). La programmabilité physique n'a de sens que par rapport à des séries d'opérations spécifiquement sélectionnées. Pour remédier à ce problème, j'ai ajouté la clause de « sélection » dans ma caractérisation de la programmabilité physique. L'idée de la clause de sélection est de nous guider/informer dans le processus de sélection des opérations de l'automate matériel et d'isoler un phénomène spécifique, en fonction de l'intérêt d'un individu ou d'une communauté épistémique. N.b., en tant que telle, la clause de sélection va de pair avec l'idée de restreindre l'applicabilité de la programmabilité physique aux seuls automates matériels conçus.

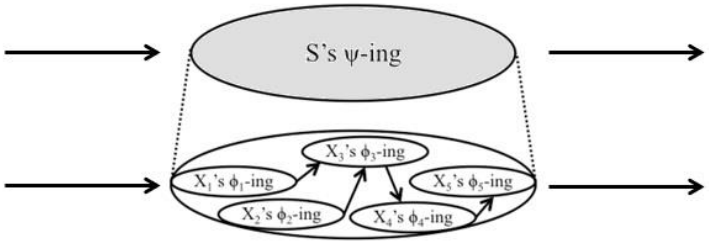

Fig. C.3 : Représentation schématique d'un mécanisme.

5.4 Reconfiguré

Notre compréhension de la « reconfiguration contrôlée » est facilitée par les théories de la causalité dites de manipulabilité ou d'agence, un sous-ensemble de l'interventionnisme causal (cf. Woodward (2023) pour une étude). En bref, les théories de la manipulabilité visent à élucider les structures causales par les moyens suivants

> **« Difference Making »** : *C* est une cause de *E* (l'effet) *si* la manipulation de *C* de la bonne manière affecte (fait une différence sur) *E*.

L'interventionnisme sous sa forme contemporaine (voir, par exemple, (Woodward 2003) et (Pearl 2009) est né de la combinaison de caractéristiques issues de la modélisation causale et des théories de la manipulabilité. La principale réalisation de cette théorie a été de concevoir une notion formelle d' « intervention » basée sur des modèles dits structurels, nous permettant de représenter les relations causales (en science) à travers un cadre formel rigoureux.

En tant que telles, les idées de l'interventionnisme élucident la manière dont nous pouvons reconfigurer et contrôler les automates programmables du monde réel. En nous appuyant sur ce cadre formel, nous pouvons expliquer la programmabilité sans l'anthropomorphisation typiquement évoquée des ordinateurs (« ils comprennent les instructions »). L'application de concepts interventionnistes aux mécanismes d'I/O nous permet de comprendre comment contrôler les automates programmables. En particulier, je suggère que le contrôle humain des machines programmables est rendu plausible par le critère dit de « Manipulabilité Mutuelle » (MM).

5.5 Le degré de programmabilité d'un automate

En examinant divers exemples d'automates, j'ai montré que la programmabilité physique est une notion graduelle et qu'elle existe à différents degrés. Intuitivement, nous pouvons caractériser cette caractéristique comme la variabilité du comportement potentiel du système. Certains chercheurs de tendance historico-philosophique ont tenté de rassembler ce type de considérations sur *toutes sortes d'*automates programmables dans un système de classification théoriquement plus robuste (Brennecke 2000 ; Koetsier 2001 ; D'Udekem Gevers 2013 ; et Copeland & Sommaruga 2021).

Que faut-il retenir de ces schémas de classification des automates programmables du monde réel ? Ils classent tous les machines informatiques à *usage général* comme étant celles qui présentent la plus grande programmabilité. En d'autres termes, les ordinateurs à usage général - parfois appelés machines universelles - sont censés être capables implémente pratiquement toutes les

fonctions *calculables* (telles que définies par la théorie de la calculabilité). En d'autres termes, la classe des fonctions calculables agit comme une *limite supérieure à* la séquence d'opérations qu'une machine informatique peut effectuer. Si nous pouvons configurer une machine pour implémente toutes ces fonctions, elle est entièrement programmable ; les machines qui ne peuvent implémenter qu'un éventail plus restreint de fonctions sont donc moins programmables. En résumé, ces considérations conduisent à la notion suivante

> **Le degré auquel :** La quantité/part de l'ensemble des fonctions possibles (au sens du mécanisme d'I/O $f(i)=o$) que le système peut implémenter**.**

5.6 Remarques finales et questions ouvertes

La conclusion clôt mon chapitre en énonçant les avantages de la « programmabilité physique » et en énumérant les problèmes en suspens. L'un des principaux enseignements de ce nouveau concept est sa capacité à expliquer la programmabilité sans faire appel à la métaphore du langage (« l'ordinateur comprend les instructions ») mais en termes d'interventions sur le mécanisme d'I/O conçu.

Les recherches futures pourraient porter sur deux problèmes en suspens : Premièrement, les questions relatives à la « manipulabilité mutuelle » (MM) persistent (Couch (2011), Leuridan (2012), Romero (2015) et Kästner (2017)). Dans la littérature contemporaine sur les mécanismes, il y a un débat permanent sur la plausibilité de la MM parce que les interventions semblent être *maîtrisées* (c'est-à-dire qu'elles font une différence dans le mécanisme et (au moins certaines) de ses entités agissantes).

Deuxièmement, jusqu'à présent, j'ai simplement discuté de la programmabilité physique en relation avec des automates fonctionnant en séquence et de manière largement autonome. Cependant, en particulier en ce qui concerne l'informatique, il existe d'autres modes d'opérations/paradigmes importants (naturel, analogique et quantique). En outre, il serait intéressant de clarifier la relation entre la programmabilité et les paradigmes informatiques interactifs.

Chapitre 6 - Conclusion

Enfin, le chapitre 6 résume les principales conclusions de la thèse et fournit des indications pour les recherches futures sur le sujet. En résumé, les points à retenir sont les suivants : Le terme « programme » est un polysème. Il désigne donc des choses ontologiquement différentes. Dans cette thèse, j'ai fourni un cadre sur la façon dont ils sont liés : UTAI. Selon cette notion, nous

devons accorder une attention particulière à trois problèmes philosophiques : le problème de la création (qui détermine la nature abstraite des programmes) ; la question de savoir comment déterminer la programmabilité physique (qui détermine l'aspect physique des choses) ; et le problème de l'implémentation. (qui traite de la manière dont les programmes abstraits se rapportent à la physique).

Annexe A - Vue d'ensemble de la chimère des programmes

A1 La vue physique

Je me réfère aux cadres qui préconisent une certaine forme de compréhension physique des programmes en tant que vision physique. Afin de décortiquer les notions regroupées sous l'égide de la vision physique, il est utile de prendre en compte une discussion plus approfondie de la métaphysique : la dualité entre les *continuités* et les *occurrences* (voir, par exemple, Simons 2000). Cette division se reflète également dans les différentes conceptions métaphysiques des programmes en tant qu'entités physiques. En conséquence, je distingue la physicalité des programmes en deux cas différents, à savoir une lecture *statique* et une lecture *dynamique*.

D'une part, les programmes peuvent être considérés comme faisant partie d'une machine. Cette idée était peut-être plus évidente lors de l'utilisation d'ordinateurs de première génération comme l'ENIAC, où les réglages des commutateurs étaient visibles/tangibles. La machine devait être physiquement configurée pour exécuter les opérations requises pour un calcul donné dans la séquence correcte. D'autre part, il existe un point de vue très répandu selon lequel les programmes provoquent ou même *sont des* sortes de processus du monde réel (parfois appelé *processus de programme*). L'accent mis sur l'aspect empirique des choses est, par exemple, largement discuté dans la littérature sur la nature de l'*informatique* en tant que discipline.

A2 Le point de vue mathématique

Plusieurs personnalités influentes du monde de l'informatique, telles que Dijkstra, Floyd, McCarthy, Naur et Wirth, pensaient que l'adoption d'une approche mathématique et rigoureuse de la construction des programmes pouvait améliorer la qualité des « logiciels » et de la programmation. Hoare a exprimé une position extrême, suggérant que toute l'informatique pouvait se résumer aux mathématiques. Selon lui, les ordinateurs fonctionnent comme des machines mathématiques, les programmes informatiques sont des expressions

mathématiques, les langages de programmation sont des théories mathématiques et la programmation elle-même est une activité mathématique.

A3 Le point de vue notationnel

Considérer les programmes comme des sortes de textes est un parasite pour l'utilisation répandue des langages de programmation modernes. Selon ce point de vue, les programmes sont constitués d'une séquence bien formée de symboles écrits dans un langage de programmation. Cette vision soulève plusieurs questions quant à la nature des langages de programmation et, par conséquent, des programmes en tant que textes écrits dans un tel « langage ». Pour mieux cerner la question, un bref aperçu de l'évolution historique des langages de programmation est fourni.

A4 Le point de vue de l'artefact

Dans notre vie quotidienne, nous sommes entourés et constamment confrontés à des artefacts. Généralement, un artefact est défini comme un objet fabriqué ou produit intentionnellement dans un but spécifique (Hilpinen 2017). Intuitivement, de nombreux programmes informatiques semblent être des artefacts parce qu'ils sont des « créations de l'esprit ». En temps voulu, les philosophes font souvent la distinction entre différents types d'artefacts. Deux conceptions en particulier s'avèrent pertinentes pour classer les programmes informatiques : les artefacts techniques et les artefacts abstraits. En termes simples, les artefacts techniques sont des objets matériels conçus intentionnellement et caractérisés par une dualité fonction-structure. Les artefacts abstraits sont des objets abstraits créés intentionnellement qui ne peuvent être identifiés par une instanciation unique.

A5 Le point de vue neuronal

Il existe une longue et riche tradition (philosophique) qui consiste à concevoir l'esprit comme une machine (Boden 2006). Avec l'avènement des machines à calculer électroniques, il n'a pas fallu longtemps pour que les idées sur l'ordinateur et le cerveau s'enchevêtrent mutuellement. Cela a façonné à la fois la perception des types d'objets que sont les ordinateurs et les cerveaux, ce qui a eu des conséquences sur la compréhension des programmes informatiques. En conséquence, j'élucide comment nous pouvons considérer l'ordinateur comme un cerveau et, vice versa, comment nous pouvons considérer le cerveau comme un ordinateur.

A6 « State of the Art »

Dans cette section, je passe en revue les différents points de vue sur le statut ontologique des programmes informatiques sur le marché.

Tout d'abord, je passe en revue l'article classique de Moor (1978) dans lequel il examine le point de vue de la double nature des programmes et propose des raffinements linguistiques. Je procède ensuite à une évaluation critique de la proposition de Suber (1988) selon laquelle tout est un programme. Ensuite, je discute de la proposition de Smith (1998) pour une révision métaphysique complète. Après quoi, je passe au crible les « abstractions concrètes » de Colburn (1999) (dans lesquelles il adopte le point de vue de la double nature). Puis, je fais la lumière sur l'article d'Eden et Turner (2007) dans lequel ils discutent de certaines implications du point de vue de la double nature et proposent d'autres raffinements linguistiques du terme « programme ». Lando et al. (2007) proposent une autre façon de clarifier les définitions et la double nature ; la nouveauté de leur récit est l'appel aux ontologies formelles. Je décris ensuite l'idée d'Irmak (2012) de considérer les programmes comme des artefacts abstraits. Ceci est suivi par une brève analyse de Duncan (2014) dans laquelle il tente de démêler la distinction logiciel/matériel par le biais d'ontologies formelles. Je traite aussi des travaux de Wang et al. (2014a ; 2014b) et de Wang (2016) qui développent une perspective d'ingénierie des exigences selon laquelle les programmes sont des artefacts d'information abstraits. Ensuite, j'examine minutieusement la notion de Turner (2011 ; 2014 ; 2018) selon laquelle les programmes sont des artefacts informatiques. De plus, je mets en lumière la thèse de Geisse (2019) dans laquelle il fournit une perspective phénoménologique sur les programmes. Enfin, je passe en revue la thèse de Primiero (2016 ; 2020) que les programmes ont une ontologie stratifiée.

Annexe B - Pourquoi nous devrions considérer l'implémentation informatique comme une relation à trois places

B.1 Problème de liaison

L'une des questions centrales de la (philosophie des) mathématiques a été l'applicabilité apparemment miraculeuse des mathématiques aux sciences empiriques. Cette question, qui a captivé les chercheurs pendant des siècles, a peut-être été ravivée par Wigner (1960) lorsqu'il nous a mis au défi d'expliquer l'utilité remarquable des mathématiques dans la science. Compte tenu de sa longue histoire, la question est connue sous de nombreux noms (par exemple, le problème de l'application) et peut comprendre plusieurs problèmes différents (bien que liés) sous le même chapeau (Steiner 1998, Fillion 2012). Le problème

particulier sur lequel je me concentre concerne l'inadéquation ontologique entre les mathématiques et le monde (ci-après dénommé « Bridging Problem » (BP)) :

BP : *Quel est le lien entre les mathématiques et la physique ?*

B.2 Le problème de l'implémentation

Le calcul est méthodologiquement divisé (Curtis-Trudel 2022). D'une part, nous pouvons étudier le calcul dans le domaine abstrait du formalisme logico-mathématique comme les machines de Turing (MT), les fonctions récursives, etc. D'autre part, les calculs ont lieu dans le monde réel. Alors que la théorie formelle du calcul est une branche bien établie des mathématiques et de l'informatique théorique, l'élaboration d'un compte rendu précisant quand un système physique implémente des calculs s'avère difficile. En termes simples, la question de savoir comment relier ces deux domaines est appelée le *problème de l'implémentation*.

B.3 Tracer le paysage des solutions au « Bridging Problem »

L'idée centrale de toutes les solutions contemporaines est sans doute influencée par le structuralisme : En raison de la notion centrale de cartographies structurelles, Pincock (2004) a baptisé cette proposition « Mapping Account » (compte de cartographie).

> **Mapping Account** : Le fossé entre le M mathématique et le P physique est comblé par un mappage préservant la structure f: $S_P \rightarrow S_M$ entre deux structures correspondantes S_M et S_P.

Cependant, le compte de correspondance pose un problème fondamental : Les systèmes physiques doivent avoir des structures pour que les morphismes soient bien définis, car « le morphisme est une relation qui existe entre deux structures et non entre une structure et un élément du monde réel en soi ». Frigg (2006, 55). Le problème est que les systèmes physiques sont des entités concrètes existant dans la réalité physique, et non des structures mathématiques. Ce qu'il faut donc pour résoudre le problème de la BP, c'est expliquer comment les systèmes physiques obtiennent une structure unique. Trois solutions principales sont disponibles :

1. Le monde est fondamentalement mathématique (Tegmark 2008)

Cette idée est sujette à l'objection de Newman.

2. Proposition inférentialiste (Bueno & Colyvan 2011)
3. Proposition basée sur l'abstraction. Nguyen et Frigg (2021) ont formalisé cette idée dans leur « compte d'abstraction extensionnelle ».

En résumé, toutes ces propositions dépendent de l'activité humaine et, à ce titre, sont des adeptes de la théorie des trois lieux dépendants de l'esprit.

B.4 Synthèse des problèmes : Une nouvelle perspective

Bien que la littérature sur le calcul physique ait apporté un nombre impressionnant de contributions, l'accent mis sur la nature métaphysique de la relation de l'implémentation a généralement été relégué au second plan. Alors que les propositions de résolution du BP sont conçues pour être généralement applicables, les solutions au problème de l'implémentation sont limitées à l'applicabilité de la théorie de la calculabilité. Sur la base de cette comparaison, nous pouvons déduire que le problème de l'implémentation est une instance spécifique du problème du rapprochement.

Alors que les deux lignes de recherche proposent que le décalage ontologique puisse être surmonté en adhérant à des mappings préservant la structure, la plupart des solutions au problème de la mise en œuvre n'élucident généralement pas davantage la nature métaphysique des mappings qu'elles emploient. Bien qu'aucune solution au problème de la mise en œuvre ne se soit imposée comme définitive, elles vont dans le même sens : Toutes les analyses correspondantes partagent l'idée que la relation de mise en correspondance n'est pas un fait brut. Au lieu de cela, la relation des mathématiques au monde nécessite un troisième relatum - un agent responsable de l'établissement de la correspondance *f* et de la détermination des structures de la théorie des ensembles qui sont censées être reliées. Selon ce point de vue de la relation à trois places, le calcul physique est donc une conception dépendante de l'esprit, car un système ne peut calculer qu'en raison de l'activité humaine.

Bibliography

Abelson, H., Sussman, G. J. and Sussman, J. (1996). *Structure and Interpretation of Computer Programs.* MIT Press.

Ainsworth, P. (2009) 'Newman's Objection.' *The British journal for the Philosophy of Science,* 60 (1): 135–171.

Ambrosetti, N. (2011). *Cultural Roots of Technology: An Interdisciplinary Study of Automated Systems From the Antiquity to the Renaissance.* PhD. diss., University of Milano.

Anderson, N. G. and Piccinini, G. (2024). *The physical signature of computation: A robust mapping account.* Oxford University Press.

Anderson, N. G. 2019. 'Information Processing Artifacts.' *Minds and Machines* 29 (2): 193–225.

Angius, N. (2013). 'Abstraction and Idealization in the Formal Verification of Software Systems.' *Minds and Machines* 23 (2): 211–226.

Angius, N., and Primiero, G. (2018). 'The logic of identity and copy for computational artefacts.' *Journal of Logic and Computation* 28 (6): 1293–1322.

Angius, N. and Primiero, G. (2019). 'Infringing software property rights: Ontological, methodological, and ethical questions.' *Philosophy & Technology* 33, 283-308.

Angius, N., and Primiero, G. (2023). 'Copying safety and liveness properties of computational artefacts.' *Journal of Logic and Computation* 33 (5): 1089–1117.

Arp, R., Smith B., and D. Spear, A. D. (2015) *Building ontologies with basic formal ontology.* MIT Press.

Antigua, M. (2023). 'A Dual-Aspect Theory of Artifact Function.' *Erkenntnis* 88 (4): 1533-1554.

Aspray, W. (1990). *John von Neumann and the origins of modern computing. History of computing.* MIT Press.

Azzouni, J. (2004). *Deflating Existential Consequence: A Case for Nominalism*. Oxford University Press, Oxford, England.

Baker, L. R. (2006). 'On the twofold nature of artefacts.' *Studies in History and Philosophy of Science Part A*, 37(1): 132–136.

Balaguer, M. (1998). *Platonism and anti-Platonism in mathematics*. Oxford University Press.

Balaguer, M. (2016). 'Platonism in Metaphysics.' In *The Stanford Encyclopedia of Philosophy*, edited by Edward N. Zalta, Spring 2016. Metaphysics Research Lab, Stanford University.

Balaguer, M. (2023). 'Fictionalism in the Philosophy of Mathematics.' In *The Stanford Encyclopedia of Philosophy*, edited by Edward N. Zalta, Spring 2023. Metaphysics Research Lab, Stanford University.

Balzer, W., Moulines, C. U., and Sneed, J. D. (1987). *An Architectonic for Science: The Structuralist Program*. Springer.

Barr, N. (1988) 'The Phillips Machine' *LSE Quarterly*, 2(4): 305-337.

Batterman, R. W. (2010). 'On the explanatory role of mathematics in empirical science.' *The British journal for the Philosophy of Science*, 61(1):1–25.

Baumgartner, M. and Casini, L. (2017) 'An Abductive Theory of Constitution.' *Philosophy of Science* 84 (2): 214-233.

Baumgartner, M., and Gebharter, A. (2016). 'Constitutive Relevance, Mutual Manipulability, and Fat-Handedness.' *The British Journal for the Philosophy of Science* 67 (3): 731–756.

Bechtel, W, and Abrahamsen, A. (2005). 'Explanation: A Mechanist Alternative.' *Studies in History and Philosophy of Biological and Biomedical Sciences* 36 (2): 421–441.

Bechtel, W., and Richardson, R. C. (1993). *Discovering complexity: Decomposition and localization as strategies in scientific research*. MIT Press.

Begley, K. (2024). Towards a realist metaphysics of software maintenance. In M. T. Young, and M. Coeckelbergh (eds.), *Maintenance and Philosophy of Technology*, 162-183. Routledge.

Benacerraf, P. (1973). 'Mathematical Truth.' *Journal of Philosophy 70* (19): 661–679.

Bickle, J. (2020). 'Multiple Realizability.' In *The Stanford Encyclopedia of Philosophy*, edited by Edward N. Zalta, Summer 2020. Metaphysics Research Lab, Stanford University.

Bigelow, J. (1980). 'Computer Development for the Institute of Advanced Studies' In G. C. Rota, N. Metropolis, J. Howlett (eds.), *A History of Computing in the Twentieth Century: A Collection of Papers*, 291–310. Academic Press, Inc.

Bissell, C. (2007). 'Historical perspectives – The Moniac A Hydromechanical Analog Computer of the 1950s' *IEEE Control Systems Magazine*, 27(1): 59-64.

Boden, M. A. (2006). *Mind as Machine: A History of Cognitive Science*. Oxford University Press.

Bokulich, A. (2011). 'How Scientific Models Can Explain.' *Synthese* 180 (1): 33–45.

Bournez, O., and Pouly, A. (2021). 'A Survey on Analog Models of Computation.' In V. Brattka and P. Hertling (eds.) *Handbook of Computability and Complexity in Analysis*, 173–226. Springer

Brennecke, A. (2000). 'A Classification Scheme for Program Controlled Calculators.' In R. Rojas and U. Hashagen (eds.) *The First Computers*, 53–68. MIT Press.

Bricker, P. (2016). 'Ontological commitment.' In *The Stanford Encyclopedia of Philosophy*, edited by E. N. Zalta, Winter 2016. Metaphysics Research Lab, Stanford University.

Bringsjord, S. (2019). 'Computer science as immaterial formal logic.' *Philosophy & Technology* 33, 339–347.

Bromley, A. G. (1983). 'What Defines a "General-Purpose" Computer?' *Annals of the History of Computing 5* (3): 303-305.

Brooks, F. P. (1978). *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley Longman Publishing.

Buckner, C. and Garson, J. (2019). 'Connectionism and post-connectionist models.' In M. Sprevak and M. Colombo (eds.) *The Routledge handbook of the computational mind*, 76–90. Routledge.

Bueno, O. (2020a) 'Nominalism in the philosophy of mathematics.' In Zalta, E. N., editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, spring 2020 edition.

Bueno, O. (2020b). 'Contingent Abstract Objects,' In *Abstract Objects*, edited by J. L. Falguera and C. Martínez-Vidal. 91–109 Springer.

Bueno, O. and Colyvan, M. (2011). 'An inferential conception of the application of mathematics.' *Noûs*, 45 (2): 345–374.

Bullynck, M. and De Mol, L. (2010). 'Setting-up early computer programs: D. H. Lehmer's ENIAC computation.' *Archive for Mathematical Logic*, 49 (2): 123–146.

Burks, Arthur W. (1980). 'From ENIAC to the Stored-Program Computer: Two Revolutions in Computers'. In *A History of Computing in the Twentieth Century: A Collection of Papers* (311–344), edited by G.C. Rota, N. Metropolis, J. Howlett. Academic Press, Inc.

Burks, A. W., Goldstine, H. H., and von Neumann, J. (1946). *Preliminary Discussion of the Logical Design of an Electronic Computing Instrument*.

Butterfield, A., Ngondi, G. E and Kerr, A. (2016). *A Dictionary of Computer Science*. Oxford University Press.

Button, G and Sharrock, W. (1995). 'The Mundane Work of Writing and Reading Computer Programs.' In P. T. Have and G. Psathas (eds.), *Situated Order: Studies in the Social Organization of Talk and Embodied Activities*, 231–258. University Press of America.

Callender, C., and Cohen, J. (2006). 'There is no special problem about scientific representation. Theoria.' *Revista de teoría, historia y fundamentos de la ciencia* 21 (1): 67–85.

Cameron, Ross P. (2008). 'There Are No Things That Are Musical Works.' *British Journal of Aesthetics* 48 (3): 295–314.

Campbell-Kelly, M. (1991). 'Punched-Card Machinery' In W. Aspray et al. (eds.) *Computing Before Computers*, 122-155. Iowa State University Press.

Campbell-Kelly, M., Aspray, W. F., Yost, J. R., Tinn, H., and Con Díaz, G. (2023). *Computer: A history of the information machine*. Routledge.

Cardone, F. (2021). 'Games, Full Abstraction and Full Completeness. In E. N. Zalta (ed.) *The Stanford Encyclopedia of Philosophy*, Spring 2021. Metaphysics Research Lab, Stanford University.

Care, C. (2010). *Technology for Modelling. Electrical Analogies, Engineering Practice, and the Development of Analogue Computing.* Springer.

Carnap, R. (1950). 'Empiricism, Semantics and Ontology.' *Revue Internationale de Philosophie* 4 (11): 20–40.

Carston, R. (2021). 'Polysemy: Pragmatics and sense conventions.' *Mind & Language*, 36(1): 108–133.

Cartwright, Nancy. (1989). *Nature's Capacities and Their Measurement*. Oxford University Press.

Chabert, J.-L., editor (1999). *A History of Algorithms: From the Pebble to the Microchip*. Springer.

Chaitin, Gregory. (1966). 'On The Length of Programs for Computing Finite Binary Sequences.' *Journal of the ACM* 13 (4): 547–569.

Chakravartty, A. (2010). 'Informational versus functional theories of scientific representation' *Synthese* 172: 197-213.

Chalmers, David J. (1995). 'On Implementing a Computation.' *Minds and Machines* (4): 391-402.

Chalmers, D. J. (1996). 'Does a rock implement every finite state automaton?' *Synthese*, 108(3): 309–333.

Chalmers, D. (2012). 'The varieties of computation: A reply.' *Journal of Cognitive Science* 13(3): 211–248.

Churchland, P. and T. Sejnowski. (1992) *The Computational Brain*. MIT Press.

Clarke-Doane, J. (2016). 'What is the benacerraf problem?' In *Truth, objects, infinity: New perspectives on the philosophy of Paul Benacerraf*, edited by F. Pataut. 17–43. Springer

Colburn, T. R. (1999). 'Software, abstraction, and ontology.' *The Monist* 82 (1): 3–19.

Colburn, T.R. (2000). *Philosophy and Computer Science.* Routledge.

Colburn, T. R., and Shute, G. (2007). 'Abstraction in computer science.' *Minds and Machines* 17 (2): 169–184.

Colburn, T. R., and Shute, G. M. (2008). 'Metaphor in computer science.' *Journal of Applied Logic* 6 (4): 526-533.

Colburn, T. R., T. L. Rankin, and J. H. Fetzer (eds.). (1993). Program Verification: Fundamental Issues in Computer Science. Springer.

Colyvan, M. (2001a). 'The miracle of applied mathematics.' *Synthese*, 127: 265–278.

Colyvan, M. (2001b). *The indispensability of mathematics*. Oxford University Press.

Colyvan, M. (2024). 'Indispensability Arguments in the Philosophy of Mathematics.' In *The Stanford Encyclopedia of Philosophy,* edited by Edward N. Zalta, Summer 2024. Metaphysics Research Lab, Stanford University.

Con Díaz, G. (2019). *Software Rights*. Yale University Press.

Conrad, M. (1988). 'The price of programmability.' In *A half- century survey on The Universal Turing Machine*. 285–307. Oxford University Press

Contessa, G. (2010). 'Empiricist structuralism, metaphysical realism, and the bridging problem.' *Analysis*, 70(3).

Commission on New Technological Uses of Copyrighted Works (CONTU) (1978). *Final report of the national commission on new technological uses of copyrighted works*.

Copeland, J. B. (1996). 'What is computation?' *Synthese* 108 (3): 335–359.

Copeland, J. B. (2024). The Church-Turing Thesis. In E. N. Zalta (ed.) *The Stanford Encyclopedia of Philosophy*, Spring 2024 Edition. Metaphysics Research Lab, Stanford University.

Copeland, J. B. and Sommaruga, G. (2021). 'The stored-program universal computer: did Zuse anticipate Turing and von Neumann?'. In *Turing's revolution: the impact of his ideas about computability*. G. Sommaruga and T. Strahm (eds.). 43-101. Springer.

Costa, N. C. A. D. and French, S. (2003). *Science and Partial Truth: A Unitary Approach to Models and Scientific Reasoning*. Oxford University Press USA, New York, US.

Couch, Mark B. (2011). 'Mechanisms and constitutive relevance.' *Synthese* 183 (3): 375–388.

Cowling, S. (2017). *Abstract Entities*. Routledge.

Craver, C. F. (2007a). 'Constitutive explanatory relevance.' *Journal of Philosophical Research* 32:3–20.

Craver, C. F. (2007b). *Explaining the Brain.* Oxford University Press.

Craver, C. F. (2015). 'Levels.' In Metzinger T. and J. M. Windt (eds.) *Open MIND*, 1-26. MIND Group.

Cummins, R. (1989). *Meaning and Mental Representation.* MIT Press.

Curtis-Trudel, A. (2022). 'Why do we need a theory of implementation?' *The British Journal for the Philosophy of Science*, 73(4): 1067– 1091.

d'Udekem-Gevers, M. (2013). 'Telling the Long and Beautiful (Hi)Story of Automation!'. In *Making the History of Computing Relevant. HC 2013*. Tatnall, A., Blyth, T., Johnson, R. (eds). 173-195. Springer.

Davis, M., Sigal, R., and Weyuker, E. J. (1994). *Computability, complexity, and languages: fundamentals of theoretical computer science*. Elsevier.

Daylight, E. G., Boute, R. and Fleck, A.C. (2016). *Turing Tales*. Lonely Scholar.

De Mol, L. (2015). 'Some reflections on mathematics and its relation to computer science.' *Automata, Universality, Computation: Tribute to Maurice Margenstern*, 75-101.

De Mol, L. (2021). 'Turing machines.' In Zalta, E. N., editor, *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, winter 2021 edition.

De Mol, L., Bullynck, M. (2021). 'Roots of 'Program' Revisited'. *Commun. ACM*, 64 (4): 35–37.

De Mol, L., and Bullynck, M. (2022). 'What's in a name? Origins, transpositions and transformations of the triptych Algorithm —Code—Program.' In J. Abbate & S. Dick (Eds.), *Abstractions and embodiments: New histories of computing and society*, 146-168. Johns Hopkins University Press.

Dean, W. H. (2007). *What algorithms could not be*. PhD thesis, Rutgers University-Graduate School-New Brunswick.

Dean, W. H. (2016). 'Algorithms and the mathematical foundations of computer science.' In L. Horsten, P. Welch (eds.), *Gödel's disjunction: The scope and limits of mathematical knowledge*, 19–66.

Dennett, D. C. (1990). 'The interpretation of texts, people and other artifacts.' *Philosophy and phenomenological research* 50:177–194.

Denning, P. J., Comer, D. E., Gries, D., Mulder, M. C., Tucker, A., Turner, A. J., and Young, P. R. (1989). 'Computing as a discipline.' *Commun. ACM*, 32 (1): 9–23.

Deutsch, H. (1991). 'The creation problem.' *Topoi*, 10(2): 209–225.

Dewhurst, J. (2018). Computing mechanisms without proper functions. *Minds and Machines*, 28(3): 569–588.

Dijksterhuis, E. J. (1956). *Die Mechanisierung des Weltbildes*. Springer.

Dijkstra, E. W. (1968). 'Letters to the editor: go to statement considered harmful.' *Communications of the ACM* 11(3): 147–148.

Dijkstra, E. W. (1989). 'On the Cruelty of Really Teaching Computing Science.' *Communication of the ACM* 32, 1398–1404.

Dodd, J. (2000). 'Musical Works as Eternal Types.' *British Journal of Aesthetics* 40 (4): 424–440.

Dodd, J. (2002). 'Defending Musical Platonism.' *British Journal of Aesthetics* 42 (4): 380–402.

Dodd, J. (2007). *Works of Music: An Essay in Ontology*. Oxford University Press.

Doherty, F. and Dewhurst, J. (2022). 'Structuralism, indiscernibility, and physical computation.' *Synthese*, 200(3): 189.

Dorr, C. (2005). 'What we disagree about when we disagree about ontology.' In M. E. Kalderon (ed.), *Fictionalism in Metaphysic*s, 234– 286. Oxford University Press.

Dorr, C. (2008). There Are No Abstract Objects. In T. Sider, J. Hawthorne, and D. W. Zimmerman (eds.) *Contemporary Debates in Metaphysics*, 32-64. Blackwell.

Duncan, W. P. (2014). *The Ontology of Computational Artifacts*. PhD thesis, University of Buffalo.

Durán, J. M. (2018). *Computer Simulations in Science and Engineering. Concept, Practices, Perspectives.* Springer.

Duwell, A. (2021). *Physics and Computation*. Cambridge University Press.

Eden, A. H. (2007). 'Three paradigms of computer science.' *Minds and machines*, 17(2):135–167.

Eden, A. H. and Turner, R. (2007). 'Problems in the ontology of computer programs.' *Applied Ontology*, 2(1): 13–36.

Effingham, N. (2013). *An Introduction to Ontology.* Polity.

Egan, F. (2019). 'Defending the Mapping Account of Physical Computation'. *APA Newsletter* 19 (1): 24–25. 2

Elgin, C. Z. (1983). *With Reference to Reference.* Hacket Publishing Company.

Elgin, C. Z. (2010). in 'Telling Instances'. In *Beyond Mimesis and Convention: Representation in Art and Science* (1–17), edited by Mathew Hunter Roman Frigg. Springer.

Elgin, C. Z. 2017. *True Enough*. Cambridge: MIT Press.

Ensmenger, N. (2016). 'The multiple meanings of a flowchart.' *Information & Culture*, 51(3): 321–351.

Eronen, M. I. (2015). 'Levels of Organization: A Deflationary Account.' *Biology and Philosophy* 30 (1): 39–58.

Estrin, G. (1952). A Description of the Electronic Computer at the Institute for Advanced Studies. *Proceedings of the 1952 ACM National Meeting (Toronto)*, ACM '52. New York, NY, USA: Association for Computing Machinery, 95–109.

Falguera, J. L., Martínez-Vidal, C., and Rosen G. (2022). 'Abstract Objects'. In *The Stanford Encyclopedia of Philosophy*, edited by Edward N. Zalta, Summer 2022. Metaphysics Research Lab, Stanford University.

Falkum, I. L. and Augustin, V. (2015). 'Polysemy: Current perspectives and approaches.' *Lingua* 157, 10–16.

Farmer, Henry G. (1931). *The Organ of the Ancients from Eastern Sources From Eastern Sources (Hebrew, Syriac, Arabic).* William Reeves Bookseller.

Faulkner, P. and Runde, J. (2010). 'The social, the material, and the ontology of non-material technological objects.' Draft.

Fetzer, J. H. (1988). 'Program verification: the very idea.' *Communications of the ACM* 31 (9): 1048–1063.

Field, H. H. (1980). *Science Without Numbers: A Defence of Nominalism*. Princeton University Press.

Fillion, N. (2012). *The reasonable effectiveness of mathematics in the natural sciences*. PhD thesis, The University of Western Ontario (Canada).

Fine, Kit. (2017). 'Naive Metaphysics.' *Philosophical Issues* 27: 98–113.

Fletcher, S. C. (2018). 'Computers in Abstraction/ Representation Theory.' *Minds and Machines* 28 (3): 445–463.

Floridi, L. and Nobre, A. C. (2024). 'Anthropomorphising machines and computerising minds: The crosswiring of languages between artificial intelligence and brain & cognitive sciences.' *Minds and Machines*, 34(1): 1–9.

Floridi, L. (2008). 'The method of levels of abstraction.' *Minds and Machines*, 18(3): 303–329.

Floridi, L. (2011). *The Philosophy of Information.* Oxford University Press

Floridi, L, Fresco, N. and Primiero, G. (2015). 'On malfunctioning software.' *Synthese* 192 (4): 1199–1220.

Fodor, J. A. (1975). *The Language of Thought*. Harvard University Press.

Fodor, J. A. (1981). 'The mind-body problem.' *Scientific American*, 244(1): 114–123.

French, S. (2000). 'The reasonable effectiveness of mathematics: Partial structures and the application of group theory to physics.' *Synthese*, 125: 103–120.

French, S. (2020). *There Are No Such Things as Theories*. New York, NY, United States of America: Oxford University Press.

French, S, and Vickers, P. (2011). 'Are There No Things That Are Scientific Theories?' *British Journal for the Philosophy of Science* 62 (4): 771–804.

Fresco, N. (2014). *Physical computation and cognitive science*. Springer.

Fresco, N., and Primiero, G. (2013). 'Miscomputation'. *Philosophy & Technology* 26 (3): 253–272.

Fresco, N., B. Jack Copeland, and Wolf, M. J. (2021). 'The Indeterminacy of Computation.' *Synthese* 199 (5-6): 12753–12775.

Friedell, D. (2021). Creating abstract objects. *Philosophy Compass,* 16(10): e12783.

Frigg, R. (2006). 'Scientific representation and the semantic view of theories.' *Theoria*, 55: 49–65.

Frigg, R., and Nguyen, J. (2017). 'Scientific Representation Is Representation-As.' In H.-K. Chao and J. Reiss (eds.) *Philosophy of Science in Practice: Nancy Cartwright and the Nature of Scientific Reasoning*, 149–179, Springer.

Frigg, R., and Nguyen, J. (2018). 'The turn of the valve: representing with material models.' *European Journal for Philosophy of Science* 8 (2): 205–224.

Frigg, R., and Nguyen, J. (2020). *Modelling nature: An opinionated introduction to scientific representation.* Springer.

Frigg, R., and Nguyen, J. (2021). 'Scientific Representation'. In E. N. Zalta (ed.) *The Stanford Encyclopedia of Philosophy*, Winter 2021. Metaphysics Research Lab, Stanford University.

Frigg, R. (2022). *Models and Theories: A Philosophical Inquiry.* Routledge

M. Gabbrielli and Martini, S. (2010). *Programming Languages: Principles and Paradigms*. Springer.

Gangemi, A., Guarino, N., Masolo, C., Oltramari, A. and Schneider, L. (2002). 'Sweetening ontologies with DOLCE.' In *International conference on knowledge engineering and knowledge management*, 166-181 Springer.

Geisse, J. (2019). *Das Computerprogramm als Erfahrungsgegenstand*. PhD thesis, Technische Universität Darmstadt (Germany).

Gemignani, M. (1981). 'What is a computer program?' *The American Mathematical Monthly*, 88(3): 185–188.

Giere, R. N. (1999). 'Using models to represent reality.' In L. Magnani, N. Nersessian, and P. Thagard (eds.). *Model-based reasoning in scientific discovery,* 41–57. Springer.

Glennan, S. S. (1996). 'Mechanisms and the nature of causation.' *Erkenntnis* 44 (1): 49–71.

Glennan, Stuart. (2017). *The new mechanical philosophy*. Oxford University Press.

Godfrey-Smith, P. (2009). 'Triviality arguments against functionalism'. *Philosophical Studies* 145 (2): 273–295 (Aug).

Goehr, L. (1992). *The imaginary museum of musical works: an essay in the philosophy of music*. Oxford University Press.

Goodman, N. (1976). *Languages of Art: An Approach to a Theory of Symbols*. Hackett Publishing Company.

Grattan-Guinness, I. (2008). 'Solving Wigner's mystery: The reasonable (though perhaps limited) effectiveness of mathematics in the natural sciences.' *The Mathematical Intelligencer*, 30: 7–17.

Grier, D. A. (1996). 'The ENIAC, the Verb "to program" and the Emergence of Digital Computers.' *IEEE Annals of the History of Computing* 18 (1): 51–55.

Grier, A. D. (2013). *When Computers Were Human*. Princeton University Press.

Gurevich, Y. (2012). 'What is an algorithm.' In M. Bieliková et al. (eds.), *SOFSEM 2012: Theory and Practice of Computer Science*, 31–42. Springer.

Haigh, T. (2002). 'Software in the 1960s as concept, service, and product'. *IEEE Annals of the History of Computing* 24 (1): 5–13.

Haigh, T., and Priestley, M. (2016). 'Where Code Comes From: Architecture of Automatic Control from Babbage to Algol.' *Communications of the ACM* 59(1): 39-44.

Haigh, T., and Priestley, M. (2018). 'Colossus and Programmability.' *IEEE Annals of the History of Computing* 40 (4): 5–27.

Hailperin, M. and Kaiser, B. K. K. (1999). *Concrete Abstractions: An Introduction to Computer Science*. PWS publishing.

Hausman, D. M. (2005). 'Causal Relata: Tokens, Types, or Variables?' *Erkenntnis* 63 (1): 33–54.

Hennessy, J. L., and Patterson, D. A. (2012). *Computer architecture: a quantitative approach*. 5. Elsevier.

Hill, R. K. (2016). 'What an algorithm is.' *Philosophy & Technology*, 29(1): 35–59.

Hilpinen, R. (1993). 'X*–Authors and Artifacts.' *Proceedings of the Aristotelian Society*, 93(1): 155–178.

Hilpinen, R. (2017). 'Artifact.' In E. N. Zalta, and U. Nodelman, (eds.), *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Fall 2017 edition.

Hoare, C. (1985). 'The mathematics of programming.' In S.N. Maheshwari, (ed.) *Foundations of Software Technology and Theoretical Computer Science*, 1-18. Springer.

Hofweber, T. (2016). *Ontology and the Ambitions of Metaphysics.* Oxford University Press.

Hollaar, L. A. (2002). *Legal protection of digital information.* Bureau of National Affairs.

Hopcroft, J. E., Motwani, R., and Ullman J. D. (2001) *Introduction to Automata Theory, Languages, and Computation, 2nd Edition.* Addision-Wesley.

Horsman, D. (2015). Abstraction/representation theory for heterotic physical computing. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 373 (2046): 20140224.

Horsman, D. (2017). 'The representation of computation in physical systems.' In M. Massimi, J. Romeijn, and G. Schurz (eds.), *EPSA15 selected papers.*, 191–204. Springer.

Horsman, D., Stepney, S., Wagner, R. C., and Kendon, V. (2014). 'When does a physical system compute?' *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 470 (2169): 20140182.

Horsman, D, Kendon, V., Stepney, S., and Young, J. P. W. (2017). 'Abstraction and representation in living organisms: when does a biological system compute?' In G. Dodig-Crnkovic and R. Goivagnoli (eds.), *Representation and reality in humans, other living organisms and intelligent machines*, 91-116. Springer.

Horsman, D, Kendon, V., and Stepney, S. (2017). 'The Natural Science of Computing.' *Commun. ACM* 60 (8): 31-34 (July).

Horsman, D., Kendon, V., and Stepney, S. (2018). 'Abstraction/ Representation Theory and the Natural Science of Computation.' In M. E. Cuffaro and S. C. Fletcher (eds.) *Physical Perspectives on Computation, Computational Perspectives on Physics*, 127–150. Cambridge University Press.

Houkes, W. and Vermaas, P. E. (2010). *Technical functions: On the use and design of artefacts*. Springer.

Hughes, R. (1997). 'Models and Representation'. *Philosophy of Science* 64 (4): 336.

Hyde, D. and Raffman, D. (2018). 'Sorites Paradox.' In E. N. Zalta (ed.) *The Stanford Encyclopedia of Philosophy*, Summer 2018. Metaphysics Research Lab, Stanford University.

Illari, P. M. and Williamson, J. (2012). 'What is a mechanism? thinking about mechanisms across the sciences.' *European Journal for Philosophy of Science*, 2: 119–135.

Imbert, C. and Ardourel, V. (2023). 'Formal verification, scientific code, and the epistemological heterogeneity of computational science.' *Philosophy of Science*, 90(2): 376–394.

Ingarden, R. (1979). *The Literary Work of Art*. North Western University Press.

Irmak, N. (2012). 'Software is an abstract artifact.' *Grazer Philosophische Studien*, 86(1): 55-72.

Irmak, N. (2020). 'The Problem of Creation and Abstract Artifacts.' *Synthese* 198, 9695–9708.

Jackson, M. (1995). 'The world and the machine.' In *1995 17th International Conference on Software Engineering*, 283–292.

Jones, C. B. and Astarte, T. K. (2018). 'Challenges for semantic description: comparing responses from the main approaches.' In Bowen, J. and Liu, Z. (eds.), *3rd School on Engineering Trustworthy Software Systems*, 176–217. Springer.

Kaiser, M. I., and Krickel, B. (2017). 'The Metaphysics of Constitutive Mechanistic Phenomena.' *The British Journal for the Philosophy of Science* 68 (3): 745-779.

Kania, A. (2013). 'Platonism Vs. Nominalism in Contemporary Musical Ontology.' In C. Mag Uidhir (ed.), *Art and Abstract Objects*, 197-219. Oxford University Press.

Kästner, L. 2017. *Philosophy of Cognitive Neuroscience: Causal Explanations, Mechanisms and Experimental Manipulations*. Boston: De Gruyter.

Kästner, L, and Andersen, L. M. (2018). 'Intervening Into Mechanisms: Prospects and Challenges.' *Philosophy Compass* 13 (11): e12546.

Katz, J. J. (1998). *Realistic Rationalism*. MIT Press.

Kittler, F. (1993). 'Es gibt keine Software.' In *Draculas Vermächtnis. Technische Schriften.* 225-242. Reclam.

Kivy, P. (1983). 'Platonism in Music.' *Grazer Philosophische Studien* 19 (1): 109–129.

Klein, C. (2008). Dispositional implementation solves the superfluous structure problem. *Synthese*, 165(1): 141–153.

Knuth, D. E. and Pardo, L. T. (1980). 'The early development of programming languages.' In N. Metropolis, J. Howlett, and G. Rota (eds.) *A history of computing in the twentieth century*, 197–273. Academic Press.

Knuth, D. (1997). *The Art of Computer Programming.* 3rd ed. Addison Wesley.

Koetsier, T. (2001). 'On the prehistory of programmable machines: musical automata, looms, calculators.' *Mechanism and Machine Theory* 36 (5): 589–603.

Kolmogorov, A. N. and Uspenskii, V. A. (1958 [In Russian; translation Amer. Math. Soc. Transl. (2) 29 217-245 (1963)]). 'On the definition of algorithm.' Uspekhi Mat. Nauk 13 (4). In Russian.

Kolmogorov, A. N. (1965). 'Three approaches to the quantitative definition of information.' *Problemy Peredachi Informatsii* 1 (1): 3–11.

Kramer, J. 2007. 'Is Abstraction the Key to Computing?' *Commun. ACM* 50 (4): 36–42 (April).

Krickel, B. (2018). *The Mechanical World: The Metaphysical Commitments of the New Mechanistic Approach*. Springer

Kripke, S. A. (1982). *Wittgenstein on rules and private language: An elementary exposition.* Harvard University Press.

Kroes, P. and Meijers, A. (2006). 'The dual nature of technical artefacts.' *Studies in History and Philosophy of Science Part A*, 37(1): 1–4.

Kroes, P. (2012). *Technical artefacts: Creations of mind and matter: A philosophy of engineering design*. Springer.

Kulstad, M. and Carlin, L. (2020). 'Leibniz's Philosophy of Mind.' In Zalta, E. N. (ed.), *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Winter 2020 edition.

Ladyman, J. (2009) 'What does it mean to say that a physical system implements a computation?' *Theoretical Computer Science* 410: 376–383.

Lakatos, I. (1976). *Proofs and Refutation.* Cambridge University Press.

Lando, P., Lapujade, A., Kassel, G., and Fürst, F. (2007). 'Towards a general ontology of computer programs.' In J. Filipe, M. Helfert, B. Shishkov (eds.), *Proceedings of the Second International Conference on Software and Data Technologies*, 163-170. Springer.

Lee, J. (2020). 'Mechanisms, wide functions, and content: Towards a computational pluralism.'*The British Journal for the Philosophy of Science* 72 (1): 221-244.

Leuridan, B. (2012). 'Three problems for the mutual manipulability account of constitutive relevance in mechanisms.' *The British Journal for the Philosophy of Science* 62 (2): 399-427.

Levaux, C. (2017). 'The Forgotten History of Repetitive Audio Technologies.' *Organised Sound* 22 (2): 187-194.

Levinson, J. (1980). 'What a musical work is.' *Journal of Philosophy*, 77(1): 5–28.

Lewis, D. K. (1971). 'Analog and Digital'. *Noûs* 5 (3): 321–327.

Lewis, D. K. (1986). *On the Plurality of Worlds.* Malden, Mass.: Wiley-Blackwell.

Liggins, D. (2008). 'Quine, Putnam, and the 'Quine–Putnam? Indispensability Argument.' *Erkenntnis* 68 (1): 113–127.

Linnebo, Ø. (2024). Platonism in the Philosophy of Mathematics. In E. N. Zalta and U. Nodelman (eds.) *The Stanford Encyclopedia of Philosophy*, edited by, Summer 2024. Metaphysics Research Lab, Stanford University.

Livingston, P. (2021). 'History of the Ontology of Art.' In E. N. Zalta (ed.) *The Stanford Encyclopedia of Philosophy*, edited by, Fall 2021. Metaphysics Research Lab, Stanford University.

Lonati, V., Brodnik, A., Bell, T., Csizmadia, A. P., De Mol, L., Hickman, H., Keane, T., Mirolo, C., and Monga, M. (2022). 'What we talk about when we talk about programs.' In *Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education, ITiCSE-WGR '22*, 117–164. Association for Computing Machinery.

Lowe, E. J. (2006). *The Four-Category Ontology: A Metaphysical Foundation for Natural Science*. Clarendon Press.

Machamer, P, Darden, L., and Craver, C. F. (2000). 'Thinking about mechanisms.' *Philosophy of science* 67 (1): 1–25.

MacKenzie, D. A. (2004). *Mechanizing proof: computing, risk, and trust*. MIT Press.

Maddy, P. (1992). 'Indispensability and practice.' *Journal of Philosophy*, 89(6): 275–289.

Mahoney, M. S. (2000). 'Software as science - science as software.' In U. Hashagen, R. Keil-Slawik, and A. L. Norberg (eds.), *History of Computing: Software Issues*, 25–48. Springer.

Maley, C. J. (2011). 'Analog and Digital, Continuous and Discrete.' *Philosophical Studies* 155 (1): 117–131.

Maroney, O. J. E., and Timpson, C. G. (2018). 'How is There a Physics of Information? On Characterizing Physical Evolution as Information Processing.' In *Physical Perspectives on Computation, Computational Perspectives on Physics*, 103-126. Cambridge University Press

Marr, D. (2010). *Vision: A computational investigation into the human representation and processing of visual information.* MIT press.

Martin, C. D. (1993). 'The myth of the awesome thinking machine.' *Commun. ACM*, 36(4): 120–133.

Martin, A., Magnaudet, M. and Conversy, S. (2023). 'Computers as Interactive Machines: Can We Build an Explanatory Abstraction?' *Minds and Machines* 33 (1): 83–112.

McCulloch, W. S. and Pitts, W. (1943). 'A logical calculus of the ideas immanent in nervous activity.' *The bulletin of mathematical biophysics*, 5: 115–133.

Miłkowski, M. (2011). 'Beyond formal structure: A mechanistic perspective on computation and implementation.' *Journal of Cognitive Science* 12(4): 359–379.

Miłkowski, M. (2013). *Explaining the computational mind*. MIT Press.

Miłkowski, M. (2014). 'Computational Mechanisms and Models of Computation.' *Philosophia Scientiæ*, 18 (3): 215-228.

Miłkowski, M. (2016). 'Computation and multiple realizability.' In Müller, V. C., (ed.), *Fundamental Issues of Artificial Intelligence*, 29–41. Springer.

Mollo, D. C. (2017). 'Functional individuation, mechanistic implementation: the proper way of seeing the mechanistic view of concrete computation'. *Synthese* 195: 3477–3497.

Mollo, D. C. (2018). 'Functional individuation, mechanistic implementation: The proper way of seeing the mechanistic view of concrete computation.' *Synthese*, 195(8):3477–3497.

Moor, J. H. (1978). 'Three myths of computer science.' *The British Journal for the Philosophy of Science*, 29(3): 213–222.

Morgan, M. S. (2012) *The World in the Model*. Cambridge University Press.

Moschovakis, Y. N. (2001). 'What is an algorithm?' In B. Engquist & W. Schmid (eds.), *Mathematics unlimited–2001 and beyond*, 919–936. Springer.

Mozgovoy, Maxim. (2009). *Algorithms, Languages, Automata, and Compilers: A Practical Approach*. Jones Bertlett Learning.

Neisser, U. (1967). *Cognitive Psychology*. Appleton-Century- Crofts.

Newell, A. and Simon, H. (1976). 'Computer science as empirical inquiry: symbols and search.' *Communications of the ACM*, 19(3):113–126.

Newell, A. (1986). 'Response: The models are broken, the models are broken.' *University of Pittsburgh Law Review* 47(4): 1023–1031.

Newlyn, W. T. (1950). 'The Phillips/Newlyn Hydraulic Model' *Yorkshire Bulletin of Economics* 17: 282–305.

Newman, M. H. A. (1928). 'Mr. Russell's Causal Theory of Perception.' *Mind* 37 (146): 26–43.

Nguyen, J. and Frigg, R. (2021). 'Mathematics is not the only language in the book of nature.' *Synthese* 198, 5941–5962.

Nofre, D., Priestley, M., and Alberts, G. (2014). 'When technology became language: The origins of the linguistic conception of computer programming, 1950–1960.' *Technology and culture* 55(1): 40–75.

National Research Council (NRC) (2004). *Computer Science: Reflections on the Field, Reflections from the Field*. The National Academies Press, Washington, DC.

Olley, A. (2010). 'Existence Precedes Essence – Meaning of the Stored-Program Concept.' *History of Computing. Learning from the Past*, 169-178. Springer

Papayannopoulos, P. (2020). 'Computing and modelling: Analog vs. Analogue.' *Studies in History and Philosophy of Science Part A* 83:103–120.

Papayannopoulos, P. (2023). 'On algorithms, effective procedures, and their definitions.' *Philosophia Mathematica*, 31(3): 291–329.

Papayannopoulos, P., Fresco, N., and Shagrir, O. (2022). 'On two different kinds of computational indeterminacy.' *The Monist*, 105(2): 229-246.

Patterson, D. A., and Hennessy, J. L. (2014). *Computer Organization and Design - The Hardware /Software Interface* (Revised 5th Edition). The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press.

Pearl, J. 2009. *Causality*. Cambridge University Press.

Penn, J. (2020). *Inventing Intelligence: On the History of Complex Information Processing and Artificial Intelligence in the United States in the Mid-Twentieth Century*. PhD thesis, University of Cambridge (UK).

Phillips, Alban W. H. 1950. 'Mechanical Models in Economy Dynamics' *Economia* 18 (67): 283–305.

Piccinini, G. and C. Maley. (2021) 'Computation in Physical Systems.' *The Stanford Encyclopedia of Philosophy* (Summer 2021 Edition), E. N. Zalta (ed.).

Piccinini, G. (2004). 'The first computational theory of mind and brain: A close look at McCulloch and Pitts' Logical Calculus of Ideas Immanent in Nervous Activity.' *Synthese* 141 (2):175-215.

Piccinini, G. (2007). 'Computing mechanisms.' *Philosophy of Science*, 74(4):501–526.

Piccinini, G. (2008). 'Computers' *Pacific Philosophical Quarterly* 89 (1): 32–73.

Piccinini, G. (2015). *Physical computation: A mechanistic account*. Oxford University Press.

Piccinini, G. (2020). *Neurocognitive Mechanisms: Explaining Biological Cognition*. Oxford University Press.

Pincock, C. (2004). 'A new perspective on the problem of applying mathematics.' *Philosophia Mathematica*, 12(2): 135–161.

Pincock, C. (2012). *Mathematics and scientific representation*. Oxford University Press.

Pincock, C. 2022. 'Concrete Scale Models, Essential Idealization, and Causal Explanation'. *British Journal for the Philosophy of Science* 73 (2): 299–323.

Plebani, M. (2018). 'The Indispensability Argument and the Nature of Mathematical Objects.' *Theoria: An International Journal for Theory, History and Fundations of Scie*nce 33 (2): 249–263.

Popper, K. (1978). 'Three Worlds: The Tanner Lectures on Human Values.' Utah: University Press of Utah.

Pour-El, M. B. (1974). 'Abstract Computability and Its Relation to the General Purpose Analog Computer (Some Connections Between Logic, Differential Equations and Analog Computers)'. *Transactions of the American Mathematical Society* 199:1–28.

Preston, B. (2018). 'Artifact'. In *The Stanford Encyclopedia of Philosophy*, edited by E. N. Zalta, Fall 2018. Metaphysics Research Lab, Stanford University.

Priestley, M. (2011). *A science of operations: machines, logic and the invention of programming*. Springer.

Priestley, M. (2018). *Routines of Substitution: John von Neumann's Work on Software Development, 1945–1948*. Springer.

Primiero, G. (2016). 'Information in the philosophy of computer science.' In L. Floridi (ed.), *The Routledge Handbook of Philosophy of Information.* 90–106. Routledge

Primiero, G. (2020). *On the Foundations of Computing*. Oxford University Press.

PROGRAMme (forthcoming). *What is a Computer Program?*

Psillos, S. (2006). 'The Structure, the Whole Structure, and Nothing but the Structure?' *Philosophy of Science* 73(5): 560–570.

Putnam, H. (1967). 'Psychological predicates.' In Capitan, W. and Merrill, D., editors, Art, Mind, and Religion, pages 37–48. University of Pittsburgh Press.

Putnam, H. (1971). *Philosophy of Logic*. Routledge.

Putnam, H. (1988). *Representation and Reality*. MIT Press.

Pylyshyn, Z. W. (1984). *Computation and Cognition*. MIT Press.

Quine, W. V. O. (1948). 'On what there is.' *Review of Metaphysics*, 2(5): 21–38.

Quine, W. V. O. (1976). 'Carnap and Logical Truth' reprinted in *The Ways of Paradox and Other Essays*, revised edition, 107–132. Harvard University Press

Randell, B. 1994. 'The Origins of Computer Programming.' *IEEE Annals of the History of Computing* 16(4): 6-14.

Rapaport, W. J. (1999). 'Implementation is Semantic Interpretation'. *The Monist* 82 (1): 109–130.

Rapaport, W. J. (2005). 'Implementation is semantic interpretation: further thoughts.' *Journal of Experimental & Theoretical Artificial Intelligence* 17 (4): 385–417.

Reiss, J., and Sprenger, J. (2020). 'Scientific Objectivity.' In E. N. Zalta (ed.), *The Stanford Encyclopedia of Philosophy*, Winter 2020. Metaphysics Research Lab, Stanford University.

Rescorla, M. (2013) 'Against Structuralist Theories of Computational Implementation.' *The British journal for the Philosophy of Science* 64 (4): 681-707.

Rescorla, M. (2014). 'A theory of computational implementation.' *Synthese* 191 (6): 1277–1307.

Rescorla, M. (2020). 'The Computational Theory of Mind.' In Zalta, E. N., editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Fall 2020 edition.

Resnik, M. D. (1997). *Mathematics as a Science of Patterns*. Oxford University Press.

Ritchie, J. B. and Klein, C. (2023). 'Computing in the nick of time.' *Ratio*, 36(3): 169–179.

Ritchie, J. B, and Piccinini, G. (2018). 'Computational implementation'. In *The Routledge Handbook of the Computational Mind.* (192–204), edited by M. Sprevak and M. Colombo. Routledge.

Rojas, R. (1996). 'Conditional Branching is not Necessary for Universal Computation in von Neumann Computers' *Journal of Universal Computer Science* 11 (2): 756-768.

Rojas, R. (1998). 'How to make Zuse's Z3 a universal computer.' *IEEE Annals of the History of Computing* 20(3): 51-54.

Rojas, R. (2023). *Konrad Zuse's Early Computers: The Quest for the Computer in Germany*. Springer.

Romero, F. (2015). 'Why there isn't inter-level causation in mechanisms.' *Synthese* 192 (11): 3731–3755.

Rossberg, M. (2013). 'Destroying Artworks.' In Christy Mag Uidhir (ed.), *Art & Abstract Objects*, 62–83. Oxford University Press.

Rudner, R. (1950). 'The Ontological Status of the Esthetic Object.' *Philosophy and Phenomenological Research* 10 (3): 380–388.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). 'Learning representations by back-propagating errors.' *Nature* 323(6088): 533–536.

Sanfilippo, E. M. (2021). 'Ontologies for information entities: State of the art and open challenges.' *Applied ontology*, 16(2): 111–135.

Scheines, R. (2005). 'The Similarity of Causal Inference in Experimental and Non-Experimental Studies.' *Philosophy of Science* 72 (5): 927–940.

Scheutz, M. (1999). 'When physical systems realize functions...' *Minds and Machines*, 9(2): 161–196.

Schiaffonati, V. and Verdicchio, M. (2014). 'Computing and experiments: A methodological view on the debate on the scientific nature of computing.' *Philosophy and Technology*, 27(3): 359–376.

Schweizer, P. (2019). 'Computation in physical systems: A normative mapping account.' In *On the cognitive, ethical, and scientific dimensions of artificial intelligence*, 27–47. Springer.

Scott, M. L. (2009). *Programming Language Pragmatics*. Morgan Kaufmann Publishers Inc.

Searle, J. R. (1990). 'Is the brain a digital computer?' In *Proceedings and addresses of the American Philosophical Association,* 64 (3): 21–37.

Searle, J. R. (1996). *The Construction of Social Reality*. Pinguin Books Limited.

Sennet, A. (2016). 'Polysemy.' In *The Oxford Handbook of Topics in Philosophy*. Oxford University Press.

Sennet, A. (2023). 'Ambiguity.' In E. N. Zalta and U. Nodelman (eds.), *The Stanford Encyclopedia of Philosophy*, Summer 2023. Metaphysics Research Lab, Stanford University.

Shagrir, O. (2001). 'Content, computation and externalism.' *Mind*, 110 (438): 369–400.

Shagrir, O. (2018). 'In defense of the semantic view of computation.' *Synthese* 197(9): 4083–4108.

Shagrir, O. 2020. 'In Defense of the Semantic account' *Synthese* 197 (9): 4083–4108.

Shagrir, O. (2022). *The Nature of Physical Computation. Oxford studies in philosophy of science*. Oxford University Press.

Shapiro, S. (1997). *Philosophy of Mathematics: Structure and Ontology*. Oxford University Press.

Shapiro, F. A. (2000). 'Origin of the Term Software: Evidence from the JSTOR Electronic Journal Archive.' *IEEE Annals of the History of Computing* 22 (2): 69–71

Sharpe, R. A. (2001). 'Could Beethoven have 'Discovered' the Archduke Trio?' *British Journal of Aesthetics*, 41(3): 325–327.

Simon, H. A. (1996). *The Sciences of the Artificial*. MIT Press.

Simons, P. (2000). 'Continuants and occurrents.' *Aristotelian Society Supplementary* 74(1): 59–75.

Sloman, A. (2002). 'The irrelevance of Turing machines to artificial intelligence.' In *Computationalism: new directions.* Matthias Scheutz (ed.). 87– 127. MIT Press.

Smith, B. C. (1985). 'The limits of correctness.' *ACM SIGCAS Computers and Society*, 14(1): 18–26.

Smith, B. C. (1996). *On the Origin of Objects*. MIT Press.

Solomonoff, R. J. (1964). 'A Formal Theory of Inductive Inference Part I & II.' *Information and Control* 7 (1):1–22, 224–254.

Sorensen, R. (2023). 'Vagueness.' In E. N. Zalta and U. Nodelman (eds.), *The Stanford Encyclopedia of Philosophy*, Winter 2023. Metaphysics Research Lab, Stanford University.

Sprevak, M. (2010). 'Computation, individuation, and the received view on representation.' *Studies in History and Philosophy of Science Part A*, 41 (3): 260–270.

Sprevak, M. (2018). 'Triviality arguments about computational implementation.' In M. Sprevak and M. Colombo (eds.) *The Routledge Handbook of the Computational Mind*, 175–191. Routledge.

Steiner, M. (1998). *The Applicability of Mathematics as a Philosophical Problem*. Harvard University Press.

Sterrett, S. G. (2017). 'Experimentation on analogue models.' In *Springer handbook of model-based science*, 857–878. Springer.

Studtmann, P. (2024). 'Aristotle's Categories.' In E. N. Zalta and U. Nodelman (eds.) *The Stanford Encyclopedia of Philosophy*, Spring 2024. Metaphysics Research Lab, Stanford University.

Suárez, M. (2003). 'Scientific Representation: Against Similarity and Isomorphism.' *International Studies in the Philosophy of Science* 17 (3): 225–244.

Suber, P. (1988). 'What is software?' *The Journal of Speculative Philosophy*, 2(2): 89–119.

Suchman, L. A. (1987). *Plans and situated actions: The problem of human-machine communication.* Cambridge university press.

Suppes, P. (2002). *Representation and Invariance of Scientific Structures*. CSLI Publications (distributed by Chicago University Press).

Swoyer, Chris. (1991). 'Structural Representation and Surrogative Reasoning'. *Synthese* 87 (3): 449–508.

Szangolies, J. 2020. 'The Abstraction/ Representation Account of Computation and Subjective Experience.' *Minds and Machines* 30 (2): 259–299.

Tedre, M. (2011). 'Computing as a science: A survey of competing viewpoints.' *Minds and Machines* (2011) 21: 361–387.

Tedre, M. (2015). *The science of computing: shaping a discipline*. CRC Press.

Tegmark, M. (2007). 'The mathematical universe.' *Foundations of Physics*, 38(2): 101–150.

'The Moniac.' (1952). *Fortune* March 1952, 101.

Thomasson, A. L. (1999). *Fiction and Metaphysics*. Cambridge University Press.

Thomasson, A. L. (2006). 'Debates About the Ontology of Art: What Are We Doing Here?' *Philosophy Compass* 1 (3):245–255.

Thomasson, A. L. (2022). 'Categories.' In E. N. Zalta and U. Nodelman (eds.), *The Stanford Encyclopedia of Philosophy*, edited by, Winter 2022. Metaphysics Research Lab, Stanford University.

Tillman, C. (2011). 'Musical materialism.' *British Journal of Aesthetics*, 51(1): 13–29.

Tucker, C. (2018). 'How to explain miscomputation.' *Philosophers' Imprint*, 18: 1–17.

Turner, R. (2007). 'Understanding programming languages.' *Minds and Machines*, 17(2): 203–216.

Turner, R. (2010). 'Programming languages as mathematical theories.' In J. Vallverdú (ed.) *Thinking Machines and the Philosophy of Computer Science: Concepts and Principles*, 66–82. IGI Global.

Turner, R. (2011). 'Specification.' *Minds and Machines*, 21(2):135–152.

Turner, R. (2012). 'Machines.'In Hector Zenil (ed.), *A Computable Universe*, 63–76. World Scientific.

Turner, R. (2014). 'Programming languages as technical artifacts.' *Philosophy & Technology*, 27(3): 377–397.

Turner, R. (2018). *Computational Artifacts*. Springer.

Turner, R. (2020). 'Computational Inention' *Studies in Logic, Grammar and Rheothoric* 63 (76): 19–30.

Turner, R. (2021). 'Computational Abstraction.' *Entropy* 23 (2): 213

Turner, R., and Angius, N. (2020). 'The Philosophy of Computer Science.' In E. N. Zalta (ed.), *The Stanford Encyclopedia of Philosophy*, Winter 2020. Metaphysics Research Lab, Stanford University.

Uidhir, C. M., (ed.) (2013). *Art & Abstract Objects*. Oxford University Press.

Ulmann, B. (2013). *Analog Computing*. Oldenburg Verlag.

Ulmann, B. (2020). *Analog and hybrid computer programming.* Walter de Gruyter GmbH & Co KG.

Valdez, M. E. P. (1981). *A Gift from Pandora's Box: The Sofware Crisis.* PhD thesis, University of Edinburgh (Germany).

van Eck, D. (2017). 'Mechanisms and engineering science'. In *The Routledge Handbook of Mechanisms and Mechanical Philosophy.* S. Glennan and P. Illari (eds). 447–461. Routledge.

Van Fraassen, B. C. (1980). *The Scientific Image*. Oxford University Press, New York.

van Fraassen, B. C. (2008). *Scientific Representation: Paradoxes of Perspective*. Oxford University Press.

van Inwagen, P. (1977). 'Creatures of Fiction.' *American Philosophical Quarterly* 14 (4): 299–308.

van Inwagen, P., Sullivan, M., and Bernstein, S. (2023). 'Metaphysics.' In *The Stanford Encyclopedia of Philosophy*, edited by E. N. Zalta and U. Nodelman, Summer 2023. Metaphysics Research Lab, Stanford University.

Vardi, M. Y. (2012). 'What is an algorithm?' *Commun. ACM*, 55(3): 5.

Vee, A. (2013). 'Understanding computer programming as a literacy.' *Literacy in Composition Studies*, 1(2): 42–64.

Vermaas, P., and Houkes, W. (2003). 'Ascribing functions to technical artefacts: A challenge to etiological accounts of functions' *British Journal for the Philosophy of Science* 54 (2): 261-289.

von Neumann, J. (1945). *First Draft of a Report on the EDVAC*

Vos, B. (2022). 'Structuralism and the quest for lost reality.' *Journal for General Philosophy of Science / Zeitschrift für Allgemeine Wissenschaftstheorie*, 53 (4): 519–538.

Walter, W. (1973). 'Die gespeicherten Programme des Heron von Alexandria/Heron's of Alexandria stored programs.' *Information Technology* 15 (1-6): 113–118.

Wang, X. (2016). *Towards an Ontology of Software*. PhD thesis, University of Trento

Wang, X., Guarino, N., Guizzardi, G., and Mylopoulos, J. (2014a). 'Software as a social artifact: A management and evolution perspective.' In E. Yu, G. Dobbie, M. Jarke, and Purao, S., (eds.), *Conceptual Modeling*, 321–334. Springer.

Wang, X., Guarino, N., Guizzardi, G., and Mylopoulos, J. (2014b). 'Towards an ontology of software: a requirements engineering perspective.' In O. Kutz and P. Garbacz (eds.), *Formal Ontology in Information Systems*, 317–329. IOS Press.

Ware, W. H. (1953). 'The History and Development of the Electronic Computer Project at the Institute for Advanced Study.' Santa Monica, CA: RAND Corporation.

Wegner, P. 1976. 'Research paradigms in computer science.' In *Proceedings of the 2nd international Conference on Software Engineering*, 322–330.

Weisberg, M. 2013. *Simulation and similarity: Using models to understand the world*. Oxford University Press.

Weizenbaum, J. (1976). *Computer Power and Human Reason: From Judgment to Calculation*. USA: W. H. Freeman Co.

Wenmackers, S. (2016). 'Children of the Cosmos. Presenting a Toy Model of Science with a Supporting Cast of Infinitesimals.' In A. Aguirre, B. Foster and Z. Merali (eds.), *Trick or Truth? The Mysterious Connection Between Physics and Mathematics*, 5–20 Springer.

Wetzel, L. (2009). *Types and tokens: on abstract objects*. MIT Press.

Wexelblatt, R. L. (1981). *History of Programming Languages*. Academic Press.

White, G. (2004). 'The philosophy of computer languages.' In Floridi, L. (ed.), *The Blackwell Guide to the Philosophy of Computing and Information*, 237–247. Blackwell.

Wigner, E. (1960). 'The Unreasonable Effectiveness of Mathematics in the Natural Sciences.' *Communications in Pure and Applied Mathematics* 13: 1–14.

Wilson, M. (2000). 'The unreasonable uncooperativeness of mathematics in the natural sciences.' *The Monist*, 83(2): 296–314.

Wimsatt, W. (2002). 'Functional organization, analogy, and inference. 'In A. Ariew, R. Cummins, and M. Perlman, (eds.), *Functions: New Essays in the Philosophy of Psychology and Biology*, 173–221. Oxford University Press.

Wollheim, R. (1968). *Art and its Objects*. Cambridge University Press.

Wolterstorff, N. 1980. *Works and Worlds of Art*. New York: Oxford University Press

Woodward, J. (2002). What is a mechanism? A counterfactual account. *Proceedings of the Philosophy of Science Association* 2002 (3) S366–S377.

Woodward, J. (2003). *Making things happen: A theory of causal explanation*. Oxford University Press.

Woodward, J. (2008). 'Invariance, modularity, and all that.' In *Nancy Cartwright's philosophy of science*. S. Hartman, C Hoefer, and L. Bovens (eds.). 198-237. Routledge.

Woodward, J. (2023). 'Causation and Manipulability.' In *The Stanford Encyclopedia of Philosophy*, edited by E. N. Zalta, Summer 2023. Metaphysics Research Lab, Stanford University.

Ylikoski, P. (2013). 'Causal and Constitutive Explanation Compared.' *Erkenntnis* 78 (2): 277–297.

Zenil, H. (2010). Compression-based investigation of the dynamical properties of cellular automata and other systems. *Complex Systems*, 19(1), 1–28.

Zenil, H. (2012). 'On the dynamic qualitative behavior of universal computation.' *Complex Systems* 20 (3), 265–278.

Zenil, H. (2013). 'Nature-like computation and a measure of computability.' In G. Dodig-Crnkovic and R. Giovagnoli (eds.), *Natural computing /unconventional computing and its philosophical significance*, 87–113. Springer.

Zenil, H. (2014). 'What is nature-like computation? A behavioural approach and a notion of programmability.' *Philosophy &Technology* 27 (3): 399–421.

Zenil, H. (2015). 'Algorithmicity and programmability in natural computing with the Game of Life as *in silico* case study.' *Journal of Experimental & Theoretical Artificial Intelligence* 27:(1): 109–121.

Zuse, K. (1969). *Rechnender Raum (calculating space)*. Vieweg +Teubner Verlag.