Towards a theory of physical computation

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

André Curtis-Trudel, BA, BS

Graduate Program in Philosophy

The Ohio State University

2022

Dissertation Committee:

Richard Samuels, Co-Advisor

Stewart Shapiro, Co-Advisor

Christopher Pincock

John Symons

Abstract

This dissertation is about physical computation, mathematical computation, and their relationship. The primary contribution is the development of a novel account of physical computation, which I call the resemblance account. On this account, physical systems compute to the extent that they resemble mathematically defined computational models in certain contextually-specified respects.

Chapter 1 situates my project with respect to the broader philosophical landscape, tables a few specific questions for later investigation, and considers a range of adequacy criteria that should constrain our answers to these questions.

Chapter 2 clears the ground by looking more closely at the relationship between physical and mathematical computation. Here I argue that we should take a 'mathematics-first' approach, which develops the theory of physical computation in terms of a prior mathematical theory of computation. The primary task for such a theory, as I conceive of it, is to characterize the implementation relating linking physical systems to the mathematical computational structures they realize.

Chapter 3 develops a novel response to a longstanding skeptical worry that physical computation is trivial. I argue that relative to a specific, contextually specified way of regarding a system computationally there is little reason to think that physical computation is trivial. This response is flexible enough to take on board other

suggestions found in the literature, but unlike these accounts avoids positing global constraints on computation generally.

Chapter 4 examines an important class of explanations in computer science. These explanations, which I call 'limitative' explanations, explain why certain problems cannot be solved computationally. I argue that limitative explanations are a kind of non-causal, mathematical explanation that depend on highly idealized computational models such as Turing machines. However, because they are highly idealized, the relationship between Turing machines and the physical systems that implement is not best understood in terms of isomorphism.

Chapter 5 pulls these conclusions together and sketches the resemblance account. On this account, a physical system computes just to the extent that it resembles a computational model — or, as I prefer to call it, a computational architecture — in certain antecedently specified respects. Just what these respects are typically depends on a variety of contextually determined considerations, concerning both objective features of the system under consideration as well as facts about our goals and interests in describing a system computationally. After sketching the account, I close the chapter by noting a few directions for further work.

For my parents, all four of them

Acknowledgments

My interest in computation began sometime in undergrad, when a friend described to me a program they wrote for some data processing task. I remember being both impressed and mystified: impressed that by simply writing a program they could automate what would otherwise be a substantial amount of work, but mystified by how it was that something like a laptop did it. What, exactly, was going on inside the machine? One degree in computer science later, I understood, at least in rough outline, how the laptop worked. But it also seemed to me that I had merely traded one mystery for another. What made what the laptop was doing *computation*? That question, plus or minus a few details, led me to Ohio State.

Happily, I could not have asked for a more welcoming, supportive, or intellectually stimulating environment to pursue this question (plus a few others along the way). For me, at least, it takes a village, and I owe an overwhelming debt of gratitude to those I have had the fortune to work with over the past six years. Let me single out a few for special recognition.

I learned a tremendous amount from my fellow graduate students at Ohio State, past and present, with whom I had the opportunity to discuss my work and philosophy more generally: Soyeong An, Zoe Ashton, Ethan Brauer, Steven Dalglish, Jason Dewitt, Anand Ekbote, Owain Griffin, William Marsolek, Erin Mercurio, Daniel Olson, Giorgio Sbardolini, Damon Stanley, Evan Woods, and Chulmin Yoon.

I owe a special debt of gratitude to my cohort: Scott Harkema, Preston Lennon, and Xiang Yu. I am grateful to have had the opportunity to learn alongside them (and, more often than not, from them) throughout our time in Ohio. Their conversation and company were a welcome fixed point over the last six years.

Many faculty at Ohio State were generous with their time, comments, and advice as well. Special thanks on this front go to Richard Fletcher, Robert Kraut, Alex Petrov, Abe Roth, Lisa Shabel, Declan Smithies, William Taschek, and Neil Tennant. Tristram McPherson, our placement director, deserves special mention for his support leading up to and during my time on the job market. Thanks also to Michelle Brown, Lynaya Elliott, and Sue O'Keeffe for their help and expertise in navigating Ohio State's numerous labyrinthine administrative structures.

Beyond Ohio State, I have benefited from conversations with Neal Anderson, Cruz Davis, Samuel Fletcher, Tyler Millhouse, Philippos Papayannopoulos, Andrew Richmond, Michael Rescorla, Niall Roe, Paul Schweizer, Nick Wiggershaus, and audiences in Alberta, Baltimore, Chicago, Italy, New York, and Toronto.

Some of the material presented here has been published elsewhere. Chapter 2 is a revised version of "Why do we need a theory of implementation?", forthcoming in The British Journal for Philosophy of Science, and Chapter 5 is a revised and much expanded version of "Implementation as resemblance", published in Philosophy of Science 88 (5):1021-1032 (2021). Thanks to the University of Chicago Press for permission to reuse this work. I would also like to thank the battery of anonymous referees at these journals and others who kindly volunteered their time to review my work.

My deepest intellectual debts are to my committee: Stewart Shapiro, Richard Samuels, and Christopher Pincock. Throughout my time at Ohio State they have each gone above and beyond in their support, both professional and personal. I would not be half the philosopher I am today were it not for their conversation, comments, and advice. Special thanks also to John Symons, a late but welcome addition to the team. I am grateful for his enthusiasm and willingness to join on relatively short notice.

Finally, I owe by far the most significant debt of gratitude to my family: to my partner, Olivia, whose love and support have sustained me through the most difficult parts of writing this dissertation, and to my parents and siblings, without whose constant support and encouragement this dissertation would quite literally not exist.

Vita

2016 ..... BA Philosophy

2016 ..... BS Computer Science

2019 ..... Interdisciplinary Specialization,
Cognitive and Brain Science

2016-present ..... Graduate Teaching Associate,
The Ohio State University.

Publications

Research Publications

A. Curtis-Trudel Why do we need a theory of implementation? *The British Journal for Philosophy of Science*, forthcoming.

A. Curtis-Trudel The determinacy of computation. *Synthese*, 200 (1): 1-28. 2022

A. Curtis-Trudel Implementation as resemblance. *Philosophy of Science*, 88 (5): 1021-1032. 2021.

Fields of Study

Major Field: Philosophy

Table of Contents

Page

List of Figures

Figure **Page**

Chapter 1: Introduction

This dissertation is about physical computation, mathematical computation, and the relationship between them. This chapter situates my project with respect to the broader philosophical landscape, tables a few specific questions for later investigation, and considers what adequate answers to them ought to look like.

1.1 The philosophy of physical computation: a brief history

To start, I'll distinguish between two broad projects in the philosophy of physical computation. Philosophical interest in physical computation has a long history, but I'll pick up the thread in the mid-20th century with the emerging computationalist program in the philosophy of mind. At the center of this program is the view that the mind is in some important sense computational. While specific formulations of this view vary considerably, I will refer to them loosely as the computational theory of mind (CTM). For example, perhaps the most well-known version of CTM is the classical computational theory of mind, which holds roughly that the mind performs Turing-style computations over formally defined symbol structures (Rescorla, 2017a).

Although CTM per se makes no claims about how mental computations are performed, underwriting much of this work was the expectation that mental computations are performed by the brain. This expectation relied on a view about what it is

for a physical system such as the brain to compute. This view was almost universally taken for granted and was typically invoked without explicit mention. For instance, here's Putnam:

> A 'machine table' *describes* a machine if the machine has internal states corresponding to the columns of the table, and if it 'obeys' the instruction in the table in the following sense: when it is scanning a square on which a symbol s_1 appears and it is in, say, state B, that it carries out the 'instruction' in the appropriate row and column of the table (in this case, column B and row s_1). Any machine that is described by a machine table of the sort just exemplified is a Turing machine. (Putnam, 1975, 365)

The following remarks by Fodor seem to rely on a similar thought:

> A programming language can be thought of as establishing a mapping of the physical states of a machine onto sentences of English such that the English sentence assigned to a given state expresses the instruction the machine is said to be executing when it is in that state. Conversely, if L is a programming language for machine M, then it is a fact about each (computationally relevant) physical state of M that a certain sentence of English is its image under the mapping effected by L. (Fodor, 1968, 639)

Although neither Putnam nor Fodor explicitly offer a theory of physical computation here, their remarks suggest that they are operating with what is now known as the simple mapping account. Roughly put, this is the view that a physical system computes just in case it stands in an appropriate structural relationship (typically isomorphism or homomorphism) to some formal computational structure, such as a Turing machine or a program. These structures are often referred to generically as 'computations'. When a physical system stands in such a relationship, it is said to 'implement' or 'realize' the computation in question. Here is a fairly typical statement of the view:

> **The Simple Mapping Account**
> A physical system P implements a computation M if and only if:

1. There is grouping of states of P into state-types and a function f (a 'realization' or 'implementation' function) mapping state-types of P to states of M, such that
2. Under f, the state transitions of P are isomorphic to the formal state transitions of M; i.e., whenever P is in state p_1, where $f(p_1) = m_1$, and $m_1 \rightarrow m_2$ is a formal state transition, then P goes into state p_2, where $f(p_2) = m_2$.

One notable feature of this view, which I will discuss at greater length in later chapters, is that it characterizes physical computation in terms of a prior mathematical notion of computation. Thus, even in these early days, philosophical thinking about physical computation recognized a close connections between theories of physical computation on the one hand, and mathematical theories of computation, chiefly computability and computational complexity theory, on the other.

The simple mapping account, or something close to it, was (with a few exceptions) widely accepted for close to thirty years. However, it came under sustained attack in the late eighties and early nineties in the form of so-called 'triviality arguments' leveled by Putnam (1987) (repudiating his earlier view), and Searle (1992). These arguments attempted to show that the notion of physical computation furnished by the simple mapping account was 'trivial' or 'vacuous', on the grounds that every physical system implements every computation.

Putnam and Searle's attacks were prefigured in certain respects by earlier arguments in the philosophy of mind. For instance, Block's (1978) nation of China thought experiment and Hinckfuss's pail (reported in (Lycan, 1981)) put pressure on the idea that having a certain functional organization (computational or otherwise) was sufficient for mentality. Putnam and Searle's arguments, however, departed from

these earlier critiques by focusing more directly on the notion of physical computation underwriting CTM. Their avowed aim was to show that, owing to the nature of physical computation, no version of CTM is viable.

The Putnam-Searle challenge elicited two main responses. One was to give up CTM. This is the response favoured by Putnam and Searle.[1] The other was to give up the theory of physical computation targeted by the challenge — that is, the simple mapping account. This is by far the more popular option. Indeed, it would not be an exaggeration to say that philosophical thinking about physical computation has been motivated, in large part, by the task of crafting a theory of physical computation that avoids the Putnam-Searle challenge.

Perhaps the most well-known response along these lines is due to Chalmers (1994, 1996). Chalmers' overall strategy, in a nutshell, was to impose additional constraints on the simple mapping account to block triviality. This approach accepts isomorphism/homomorphism as a necessary condition on implementation, but requires that the realization map f meets some further condition. For instance, one of Chalmers' proposals was that under f formal state transitions should mirror causal state transitions: if $f(p_1) = m_1$ and $m_1 \rightarrow m_2$, then P's being in state p_1 should cause it to go into a state p_2, where $f(p_2) = m_2$.

Although there was significant disagreement about the success of Chalmers' preferred constraints, the strategy itself was by and large taken to be sound.[2] Chalmers' proposal thus inaugurated a particular research program in the philosophy of physical computation, the primary goal of which was to identify a suite of constraints that

[1] And more recently revived in (Schweizer, 2019b).

[2] See, for instance, Chalmers (2012) and the articles cited therein for critical discussion of Chalmers' proposal.

rendered physical computation non-trivial. This program dominated philosophical work on physical computation through the mid 1990s and early 2000s. Schematically put, the idea was to solve for X in the following account:

The Complex Mapping Account

A physical system P implements a computation M if and only if:

1. There is grouping of states of P into state-types and a function f (a 'realization' or 'implementation' function) mapping state-types of P to states of M, such that
2. Under f, the state transitions of P are isomorphic to the formal state transitions of M; i.e., whenever P is in state p_1, where $f(p_1) = m_1$, and $m_1 \rightarrow m_2$ is a formal state transition, then P goes into state p_2, where $f(p_2) = m_2$.
3. f satisfies condition X.

While all of this transpired, the computational sciences — here I have in mind computer and cognitive science — flourished. In its early days computer scientific research occurred primarily in mathematics and electrical engineering departments, but by the mid 1990s it had largely extracted itself and was established as a field with its own peculiar disciplinary interests and concerns. Similarly, cognitive science had by then emerged as a rich interdisciplinary field with its own distinctive approaches to the mind. Eventually philosophers began to take these developments more seriously in their theorizing about physical computation. Although vindicating CTM remained a central motivation for work on physical computation, accounting for practice in the computational sciences quickly became a prominent item on the agenda.

Motivation for the latter project was not found primarily in the philosophy of mind, but rather in the philosophy of science, where the primary goal is to understand and illuminate scientific practice. Although this might be taken to signal a departure from the earlier project of vindicating CTM, it is perhaps better understood

as a broadening of scope. Whereas before philosophical accounts of physical computation were geared chiefly towards understanding computations in the mind/brain, on the broader project such accounts should understand computation wherever it occurs. Insofar as the computational sciences are the best guide to computation in general, the task for the philosopher becomes to understand those sciences, such as computer and cognitive science, which explicitly describe and explain physical systems computationally. Perhaps the most well-known approach along these lines is due to Gualtiero Piccinini, whose mechanistic view of computation is one of the most thoroughly elaborated accounts of physical computation to date (Piccinini, 2015). Piccinini's mechanistic view was, among other things, explicitly advertised as better capturing computer scientific practice than its competitors. However, despite an avowed interest in actual computer scientific practice, Piccinini's view displays strikingly little engagement with that practice. What engagement there is focuses largely (although not exclusively) on the subfield of computer architecture (see, e.g., (Piccinini, 2015, ch. 8-13)). My own view is that this leads Piccinini to overlook certain important phenomena elsewhere in computer science. Indeed, chapter 4 of this work is dedicated to examining one such phenomenon that arises in theoretical computer science.

Of course, Piccinini's restricted focus is to some extent understandable. Computer science is a large field, and getting larger. One can only do so much. Nonetheless, a satisfactory account of physical computation should engage with a fuller sample of computer scientific practice than one finds in typical philosophical discussions. One goal of this dissertation is to investigate what happens if we focus more directly on practice in the computational sciences. I hope to better understand physical

computation in part by seeing what contemporary computer science, and to a lesser extent cognitive science, have to say about it.

Let me turn next to some of the more specific questions I will address. To date, most philosophical interest in physical computation has focused on broadly semantic questions about the content of central computational concepts, and broadly metaphysical questions about the nature of physical computation. Representative examples of the former include:

> What is the content of the concept of physical computation? What are the relationships between the concepts of an algorithm, of computing, of a computation, of something's being a computer, etc.? Are one or more of these more fundamental than the others, in some appropriate sense of 'fundamental'? To what extent can these concepts be analyzed (given some suitable conception of analysis)?

While representative examples of the latter include:

> What is it for a physical system to compute, or for a system to be a computer? What distinguishes computational states, properties, processes, events, etc. from their non-computational counterparts? What is the relationship between theories of physical computation and the formal, mathematical theories of computation developed in computability and complexity theory?

The traditional focus on such questions will come as no surprise given the semantic and metaphysical interests driving much interest in CTM. While I will have something to say in response to these questions in later chapters, one theme of the dissertation is that overemphasis on such questions has led to a third class of questions receiving less attention than they deserve. Questions in this third class are motivated principally by the goal of understanding the theoretical role played by physical computation in the contemporary computational sciences:

> What, if anything, makes computational descriptions and explanations scientifically valuable? If they are valuable, what makes them that way (e.g., accurate description, predictive utility, theoretically fruitful, etc.) How are these related to or different from non-computational descriptions and explanations? What makes a given computational description descriptively/explanatorily salient in a specific scientific context?

Of course, these lists could undoubtedly be expanded, and the questions themselves are in need substantial clarification and refinement before they can be satisfactorily addressed. Moreover, it is unlikely that these questions can be pursued in isolation. Views about the metaphysics of physical computation bear in complicated ways on how we understand the role of computation in the computational sciences. Similarly, a particular view about the scientific role of computation might require taking a particular stand on the metaphysics. This is of course a familiar philosophical point, but it's worth bearing in mind. Where we start more likely than not will partly determine where we end up.

These caveats notwithstanding, I am inclined to think that an adequate understanding of the computational sciences requires much closer engagement with the third class of questions. This dissertation is intended to be a small step in that direction.

1.2 Adequacy Criteria

It is quite plausible to think that our answers to these questions should meet certain adequacy criteria. Since these criteria play an important role in later chapters, it will be useful to survey and appraise of some of the more prominent ones found in the literature, and to identify those I take to be most important for my purposes.

The goal at this stage is just to get these criteria on the table. Later chapters will discuss them in more detail.

1.2.1 Extensionality

Extensional adequacy criteria are perhaps the most straightforward and widely accepted adequacy criteria in the literature on physical computation. In general, these criteria hold that the range of systems or processes countenanced as computational by a specific theory must match, in a sense to be explained shortly, some predetermined stock of computational systems or processes. Theories which more closely match this stock are preferable to those that do not.

Just what goes into the stock? The standard approach is to identify a set of paradigms on the one hand, and a set of anti-paradigms on the other, so that a theory ought to classify paradigms and anti-paradigms correctly. As Piccinini (2015, 12) puts its, we must ensure both that the right things compute and that the wrong things don't:

> **The right things compute**
>
> A theory should correctly classify paradigm physical computing systems as computational.

> **The wrong things don't compute**
>
> A theory should correctly classify anti-paradigm computing systems as non-computational.

These criteria are schematic, because they depend on a choice of paradigms and anti-paradigms. Different choices may yield quite different adequacy criteria. A theory that succeeds relative to one choice may fail relative to another. Thus, the choice of paradigms and anti-paradigms is itself an important, non-trivial problem.

Given my interest in capturing computational practice, the most obvious approach starts with the computational sciences. These sciences furnish a rough and ready list of paradigms and anti-paradigms, if only tacitly. I will take paradigm computing systems to be those that are routinely treated as computational by computer and cognitive science. To a first approximation, these are the systems that are described as computational or explained in broadly computational terms. Anti-paradigmatic computing systems, by contrast, are those that are not so treated.

By this measure, paradigm physical computing systems comprise both natural and artificial systems. On the natural side are, among other things, brains, certain parts of nervous systems, and human agents working effectively. On the artifactual side are calculators, digital computers, quantum computers, and analog computers. Anti-paradigms include certain middle-sized dry goods such as rocks, walls, and pails of water. Also included are digestive systems, planetary systems, thunderstorms, and tectonic plates.

These lists are obviously not exhaustive. For one thing, there are unclear cases. For instance, do abaci compute, or are they merely instruments used in the course of human calculation? More exotically, do insect swarms, RNA strands, or even the whole universe compute? Although it is sometimes suggested that these systems compute, there is no consensus on the matter. Indeed, for such cases there may be no straightforward answer one way or the other. Some indeterminacy is likely unavoidable, and we should learn to live with it. I do not take it to be a further constraint that a theory should exhaustively classify physical systems into the computational and non-computational.

Note also that the computational sciences now recognize a wider range of computing systems than they did fifty years ago, and we can expect the same in the future. Because of this, our paradigm and anti-paradigm lists are liable to evolve over time. This, of course, is as it should be. Science changes, and our philosophical theories should change with it.

The two extensionality criteria above concern computing systems. They say that a theory must classify systems, as a whole, the right way. But it is also important that we capture not just which physical systems compute, but also what they compute and how they do it. For instance, a theory which entails that a calculator is computing addition when it is actually computing subtraction is arguably unsatisfactory. The foregoing criteria are pitched at too coarse a grain to capture this distinction. This motivates the following additional criterion:

> **The right things compute the right things**
>
> If a physical computing system computes X, then a theory of physical computation should entail that that system computes X.

Here too the notion of a system 'computing X' is in a way schematic. It stands in for a variety of different things that might plausibly regarded as being computed — problems, outputs, functions, etc. Moreover, because there may be different ways to compute the same thing, relative to some choice of what it is that gets computed — different choices of X — a theory should also capture the specific way that a system computes X:

> **The right things compute the right things the right way**
>
> If a physical computing system computes X via procedure P, then a theory of physical computation should entail that it computes X via P.

1.2.2 Explanation

Another main adequacy criterion (or cluster of adequacy criteria) concern computation's explanatory role in science. Here the starting point is the observation that the computational sciences contain many explanations framed in computational terms. A theory of physical computation should account for this central aspect of computational practice.

To bring out my preferred version of this criterion, I'll contrast it with a similar one proposed by Gualtiero Piccinini:

> Computations performed by a system may explain its capacities ... A good account of computing mechanisms should say how appeals to program execution, and more generally to computation, explain the behavior of computing systems. (Piccinini, 2015, 12)

It seem to me that this criterion is too narrow. For one thing, it is framed in terms of Piccinini's proprietary notion of a computing mechanism. But we should not assume that all physical computing systems are computing mechanisms in Piccinini's sense, nor should we restrict our focus to just those systems, if it turns out that there are computing systems which are not Piccinini-style computing mechanisms. Second, Piccinini's' criterion is framed explicitly in terms of the behavior of computing systems. Although the notion of behavior is highly flexible, it seems to me that the computational sciences might aim to explain phenomena which aren't obviously behavioral. One example of this, which I return to at much greater length below, concerns limits on the computational powers of physical devices. Here we wish to explain, not what systems do or can do, but rather what they cannot do.

Consequently, for these reasons I prefer a broader, more neutral criterion:

Explanatory adequacy

> A theory of physical computation should identify explanatory uses of computation in the contemporary computational sciences, and should illuminate how such uses work.

This criterion has two parts. The first is that a theory of physical computation should correctly identify explanatory uses of computation when they occur in the comptuational sciences. This is a sort of extensionality criterion, now directed towards computational explanations. A theory should have the resources to identify paradigm cases of computational explanation, and should be able to distinguish these from, for example, failed explanations or descriptive but non-explanatory uses of computation. As before, this criterion is sensitive to choice choice of paradigms. I will have more to say about this in later chapters.

The second part is that a theory should 'illuminate' computational explanations. This involves, among other things, situating computational explanations with respect to more well-known kinds of scientific explanation and identifying the distinctive explanatory contributions (if any) that they make. For instance, are computational explanations a kind of causal explanation? If so, what distinguishes them from non-computational causal explanations? If not, what kind of explanation are they? A theory of physical computation should say something about questions such as these.

Some philosophers reject explanatory adequacy criteria on the grounds that they assume too much. For instance, Shagrir writes that although he "think[s] that computation has a substantive explanatory role ... this substantivity is not a desideratum of an account of computation" (Shagrir, 2022, 30). This is because our task, as philosophers of science, "is not to require an explanatory role, but to clarify whether the notion of computation has one" (Shagrir, 2022, 31-2).

However I am much less pessimistic than Shagrir that computation plays an explanatory role in science. Frankly, it seems to me to be almost undeniable that it does. Computational scientists routinely and without compunction explain phenomena computationally. Accordingly, I take my job, qua philosopher of science, to be to understand this practice. Of course, in saying this I do not claim that every putative computational explanation succeeds. I allow that computational scientists get things wrong, at least sometimes. A theory of physical computation should account for such cases as well.

1.2.3 Objectivity

A common third criterion holds that it should turn out that computation is in some sense objective. One statement of this comes from Piccinini:

> A account with objectivity is such that whether a system performs a particular computation is a matter of fact. Contrary to objectivity, some authors have suggested that computational descriptions are vacuous — a matter of free interpretation rather than fact... Computer scientists and engineers appeal to empirical facts about the systems they study to determine which computations are performed by which systems ... Unless the prima facie legitimacy of those scientific practices can be explained away, a good account of concrete computation should entail that there is a fact of the matter as to which computations are performed by which systems. (Piccinini, 2015, 11-2)

It seems to me that this statement runs together a couple different ideas. One is that whether a physical system computes should not be 'a matter of free interpretation'. Although Piccinini does not elaborate on the relevant notion of free interpretation, the idea seems to be that computation should be objective in the sense of not being observer-relative — i.e., being observer-independent (cf. Searle (1992, ch. 9)). In particular, here Piccinini seems to want to rule out a very permissive conception

on which an agent merely interpreting a system as computing (whatever exactly that amounts to) suffices for it to be a computing system.

This is not entirely unreasonable, but it is worth noting that computation may be observer-relative even if it is not obviously a matter of 'free interpretation'. Indeed, there are a range of possible notions of observer-relativity, of varying strength, that might be relevant for thinking about physical computation. For instance, it is not implausible to think that sometimes what a system computes depends on facts about the broader linguistic or social environment in which it is embedded. To take an example from Rescorla (2013), one might think that the fact that a system is embedded in an environment where its users interpret it as using base-10 rather than base-13 notation at least partly determines what number-theoretic function it computes. If this is right, then what the calculator computes is in some sense observer-relative even though it is not obviously a matter of 'free interpretation'. A strict ban on some degree of observer-relativity would make our theory deliver the wrong results for cases such as this.

At the same time, an observer-independence requirement is not unreasonable from the point of view of CTM. Historically, CTM was developed in part to rebut homuncular regress and circularity worries in the philosophy of mind Fodor (1968, 1965). Although it depends substantially on the details of the theory, an account of physical computation on which physical computation is observer-relative at least threatens to reintroduce these old worries. However, even if such a constraint is plausible for the project of developing a computational theory of mind, it is less clear that we should endorse it as a global adequacy condition.

For this reason, it seems to me that we should endorse a more attenuated observer-independence criterion. Rather than hold that physical computation is observer-independent across the board, a theory should have the resources to discriminate between observer-relative and observer-independent forms of computation:

> **Observer-relativity**
>
> A theory of physical computation should have the resources to discriminate between physical computing systems, if there are any, whose computational status and/or identity is observer-relative (in some appropriate sense), and those whose status and/or identity is not so relative.

As it stands, this is criterion needs to be filled out with an appropriate notion of observer-relativity. I will say a little bit more about this in chapter 3.[3]

Piccinini also appeals to the idea that empirical facts should be relevant to what a system computes. This is at least partly an epistemological point. Again, although Piccinini does not elaborate, the rough idea seems to be that whether and what a system computes shouldn't turn out to be the kind of thing we can discover entirely from the armchair. This is different from observer-independence per se. True, if computation were a matter of 'free interpretation', empirical facts may not matter very much to how one interprets a system computationally. But it is presumably at least partly an empirical fact that a device, embedded in a certain linguistic environment, computes a certain function, even if this is in some sense a mind-dependent fact about that device. Certainly, what it computes is not the sort of thing one could discover entirely from the armchair.

There are of course a range of well-known epistemic notions of varying strength that could be used to develop an appropriate epistemological criterion for theories of

[3]For a pair of similar criteria, see (Shagrir, 2022, 26-7).

physical computation. The following negative formulation allows us to sidestep this issue for the time being while still recognizing that theories of physical computation should be assessed at least in part in terms of their epistemic commitments:

> **Empirical content**
>
> A theory of physical computation should not entail that a physical system's computational status and/or identity can be discovered wholly a priori.

1.3 What's to come

So much for background and methodology. Let me sketch where we're going. The overall goal of the dissertation is to build towards a novel account of physical computation, which I call the resemblance account. Very roughly, according to this account, a physical system computes just in case it resembles a mathematically characterized computation in certain antecedently specified respects. This account departs from more familiar theories of implementation at two main points. First, on this account implementation is a ternary relation, and computational ascriptions are highly context-sensitive: a system implements a computation relative to a contextually specified way of regarding a system computationally. Second, on this account implementation is not in general a matter of being isomorphic to a mathematical computation, but rather involves resembling it in respects that may outstrip isomorphism (or similar structural relationships). I build up to this view in a few steps.

Chapter 2 clears the ground by looking more closely at the relationship between physical and mathematical computation. As I noted above, philosophers have long accepted close connections between the two, but their exact connection remains obscure. One view, which I call unificationism, holds roughly that a theory of computation should apply uniformly to both physical computing systems such as digital

computers and mathematical computing systems such as Turing machines. Chapter 2 argues that this position is untenable. Specifically, it is very hard to see how to endorse this position without running into the Putnam-Searle triviality worry. In light of this concern, I argue that we should distinguish between the physical and mathematical theories of computation. I call this alternative bifurcationism.

Bifurcationism per se underdetermines the exact relationship between physical and mathematical computation. It says that we cannot assimilate physical and mathematical computation without significant cost, but it does not offer a positive characterization of their relationship. At the end of Chapter 2 I delineate three different options and suggest that we should take a 'mathematics-first' approach, which develops the theory of physical computation in terms of a prior theory of mathematical computation. The primary task for such a theory, as I conceive of it, is to characterize the relationship of realization that relates physical systems to the mathematical computations they implement.

Chapter 3 motivates the idea that implementation is a ternary relation. I argue that treating implementation this way yields a novel and attractive response to the Putnam-Searle triviality worry. Specifically, I argue that relative to a specific, contextually specified way of regarding systems computationally, there is little reason to think that every physical system implements every computation. This response is flexible enough to take on board many other suggestions in the literature but unlike 'complex' mapping accounts avoids positing global constraints on computation generally.

Chapter 4 puts pressure on the idea that isomorphism is a necessary condition on implementation through an examination of an important class of explanations in

computer science. These explanations, which I call 'limitative' explanations, explain why certain problems cannot be solved computationally, either in principle or under certain constraints on computational resources such as time or space. I argue that limitative explanations are a kind of non-causal, mathematical explanation that depend on highly idealized computational models such as Turing machines. However, because they are highly idealized, the relationship between Turing machines and the physical systems that implement is not best understood in terms of isomorphism.

Chapter 5 pulls these conclusions together and sketches the resemblance account. On this account, a physical system computes just to the extent that it resembles a computational model — or, as I prefer to call it, a computational architecture — in certain antecedently specified respects. Just what these respects are typically depends on a variety of contextually determined considerations, concerning both objective features of the system under consideration as well as facts about our goals and interests in describing a system computationally. After sketching the account, I close the chapter by noting a few directions for further work.

Chapter 2: Why do we need a theory of implementation?

2.1 The received view of physical computation

The contemporary computational sciences describe and explain certain physical systems in terms of the computations they perform. Philosophical theories of physical computation attempt to codify and illuminate this practice. But what form should such theories take? The received view characterizes physical computation in terms of mathematical computation. Ordinarily this is taken to be a matter of relating mathematical computations to the physical world:

> Determining what conditions a physical system must satisfy in order to compute is the focus of theories of computational implementation, or physical computation . . . An account of implementation aims to specify the conditions under which a physical system performs a computation defined by a mathematical formalism – it is a theory of physical computation. (Ritchie and Piccinini, 2019, 192-3)

Though I will say more about this in due course, the crucial point is that this view begins with a mathematical notion of computation, and applies it to physical systems by way of an implementation relation. Call this 'implementationism' about physical computation.[4]

[4]Particular implementationist views are developed and defended by (Chalmers, 1994, 1996, 2011; Egan, 2010; Matthews and Dresner, 2017; Millhouse, 2019; Rapaport, 1999; Rescorla, 2014b; Schweizer, 2019a), among others. Ritchie and Piccinini (2019) and Sprevak (2019) survey the major issues.

Implementationist theories must negotiate a number of further issues, concerning both the nature of implementation, and what constraints, if any, there are on which physical systems may implement a computation. Perhaps unsurprisingly, there is widespread disagreement about how these issues should be addressed. But this is disagreement over details — how to develop implementationism, and not whether we should. What is the rationale for characterizing physical computation this way in the first place? The standard answer, roughly, is that we already have a mathematical theory of computation, so we might as well use it. Here's David Chalmers:

> The mathematical theory of computation in the abstract is well-understood, but cognitive science and artificial intelligence go beyond the domain of abstract objects to deal with concrete systems in the physical world. The difficult questions about computation are largely questions about the relationship between these domains. How does the abstract theory of computation relate to a science of concrete, causal systems, and how might it help us explain cognition in the real world? To answer these questions, we need a bridge between the abstract and the concrete domains. (Chalmers, 1994, 341-2)

As Chalmers goes on to suggest, a theory of implementation is just what is needed to 'bridge' the gap between mathematical and physical reality.[5]

It seems to me that this 'mathematics-first' approach rests on a more basic attitude towards the relationship between the theories of mathematical computation and physical computation. Implementationist theories say nothing about mathematical computing systems, such as Turing machines, because they are assumed to be dealt with by a prior mathematical theory. Given such a theory, the implementationist's task is to use it to develop an account of physical computation. Thus, by taking a mathematical theory of computation for granted, implementationism divides the

[5]Chalmers' attitude persists today. For instance, Rescorla (2017b, 288) writes that "philosophical discussion of computation should ground itself in the mathematical theory of computation."

theory of computation into two parts, each of which plays a different explanatory role. One part explains computation in the mathematical case, while the other explains computation in the physical case. Call this 'bifurcationism' about the theory of computation. I shall say more about bifurcationism shortly.

Bifurcationism has been tacitly assumed by much of the literature on physical computation. This is striking, for a couple reasons. First, what little argument exists for it rests on questionable assumptions about the nature and character of the mathematical theory of computation. Second, and perhaps more seriously, it seems that considerations from both the general philosophy of science, as well as from the philosophy of computer science in particular, augur against bifurcationism. These considerations point towards a theory which offers the same account of computation in both the physical and mathematical cases. I call this alternative 'unificationism'.

The upshot is that if we are to endorse bifurcationism, much less implementationism, a new argument is needed. The main contribution of this chapter is to provide one. I argue that the prospects for a unificationist theory which does not trivialize physical computation are grim. To the extent that a non-trivial theory of physical computation is desirable — I will assume that it is — and to the extent that bifurcationist accounts stand a better chance of avoiding triviality — I will eventually argue that they do — we ought on balance to prefer bifurcationism to unificationism.

As we will see, however, bifurcationism per se does not mandate implementationism. That is, bifurcationism as such does not require that we take Chalmers' 'mathematics-first' approach to physical computation. In fact, bifurcationism is compatible with at least two other views about the relationship between the theories of physical and mathematical computation. These other views have not received much

attention, and I submit that this partly because bifurcationism itself has not yet been isolated for independent treatment. Thus, a second contribution of this chapter is to clarify the possible relationships between theories of physical and mathematical computation, and to provide new motivation for the mathematics-first approach.

Here is the plan for the rest of the chapter. Section 2 fleshes out the distinction between unificationism and bifurcationism. Section 3 argues that extant motivation for bifurcationism, such as it is, is uncompelling. Section 4 makes a *prima facie* case for unificationism. Sections 5 and 6 argue that, initial attractions aside, unificationism faces serious problems. Section 7 concludes.

2.2 Unificationism and bifurcationism

I take it that any theory of computation must answer two main questions. The first concerns the computational status of computing systems. Here we're interested in the difference between systems that compute and those that don't. The second concerns the computational identity of computing systems. Here we're interested in the conditions under which two computing systems are computationally equivalent or distinct — modulo some appropriate standard for individuating systems computationally, e.g., in terms of the functions they compute, the algorithms they follow, and so forth. I will call these the status and identity questions, respectively:

> **Status** What distinguishes systems that compute from those that don't?
>
> **Identity** Under what conditions are two computing systems computationally identical or distinct?

The contrast between unificationism and bifurcationism can be understood as a disagreement about how these questions should be answered. To a first approximation, the bifurcationist holds that the theory which furnishes answers to the status

and identity questions concerning physical computing systems needn't also furnish answers to these questions as they concern mathematical computing systems. The unificationist, by contrast, holds that one and the same theory which furnishes answers to these questions concerning physical systems must also furnish answers to their mathematical counterparts.

So construed, unificationism and bifurcationism are methodological claims about which questions a theory of computation ought to address. It will be useful to have a 'metaphysical' version of the contrast on the table, too. On this second way of drawing the distinction, a theory is unificationist if it supplies the same kind of answer to the status and identity questions for both sort of computing system, and a theory is bifurcationist if it doesn't.[6]

How are the methodological and metaphysical claims related? For now it's enough to notice that if unificationist or bifurcationist theories are in general untenable, it's likely that the corresponding methodologies are misguided too. That is, the methodological versions of either unificationism or bifurcationism should be abandoned if it can be shown that their metaphysical counterparts are unsustainable. Accordingly, even though I focus on the prospects for unificationist and bifurcationist theories in particular, the methodological versions of these views are implicated by my arguments as well.

[6]Additionally, while I've drawn the distinction at the level of theories, we might also characterize it in terms of a variety of more fine-grained notions, such as the truth-conditions of computational statements, the applicability conditions of computational concepts, or the instantiation conditions of computational properties. For our purposes, however, the theory-centric characterization will suffice.

One might doubt that the unificationism/bifurcationism distinction marks a substantive contrast, on the grounds that it is very easy to turn any putatively bifurcationist theory into a unificationist theory. Suppose that M is a theory which answers the status and identity questions for mathematical systems and that P is a theory which answers these questions for physical systems. Then $M \wedge P$ is a unificationist theory, for one and the same theory, namely $M \wedge P$, answers the status and identity questions for both sorts of system.

However, this sort of construction doesn't get to the heart of the matter. What's at issue is not just the form a theory of computation takes, but how it answers the status and identity questions concerning different sorts of system. Moreover, this sort of construction can be ruled out easily enough: let unificationism be as before, but require that there be no proper sub-theory which supplies answers to the status and identity questions for physical systems alone.

To help bring out the intended contrast, it is instructive to compare some particular theories of physical computation. The first is Chalmers' implementationist theory.[7]

> A physical system P implements a CSA M if there is a decomposition of internal states of P into components $[s^1, s^2, ...]$, and a mapping f from the substates s_j into corresponding substates S_j of M, along with similar decompositions and mappings for inputs and outputs, such that for every state-transition rule $([I^1, ..., I^k], [S^1, S^2, ...]) \rightarrow ([S'^1, S'^2, ...], [O^1, \ldots, O^l])$ of M: if P is in internal state $[s^1, s^2, ...]$ and receiving input $[i^1, ..., i^n]$ which map to formal state and input $[S^1, S^2, ...]$ and $[I^1, ..., I^k]$ respectively, this reliably causes it to enter an internal state and produce an output that map to $[S'^1, S'^2, ...]$ and $[O^1, \ldots, O^l]$ respectively. (Chalmers, 1994, 394)

[7] Chalmers' account is couched in terms of combinatorial state automata (CSAs), a generalization of finite state automata.

Thus on this view whether a physical system computes is a matter of implementing a CSA, and what it computes is determined by the CSA (or CSAs) it implements. This approach relies on a prior mathematical theory of CSAs, specifying what mathematical entities count as CSAs and what computation a given CSA performs, which is then used to define physical computation. The account thus offers different kinds of answers to the status and identity questions as they concern physical and mathematical systems. The latter are addressed by the mathematical theory of CSA computation; the former by the account of implementation just cited. For this reason Chalmers' account seems to me a paradigmatic bifurcationist account of computation. And to the extent that Chalmers' view is representative of implementationist accounts generally, implementationism overall incurs a commitment to bifurcationism too.

Contrast this with the account of function computation found in (Copeland, 1996). According to Copeland's account:

> Entity e is computing function f if and only if there exists a labelling scheme L and a formal specification *SPEC* (of an architecture and an algorithm specific to the architecture that takes arguments of f as inputs and delivers values of f as outputs) such that (e, L) is a model of *SPEC*. (Copeland, 1996, 338)

The relevant notion of modeling, taken from model theory, is Tarskian satisfaction. Importantly, this notion allows that both physical and mathematical structures may model a given architecture–algorithm specification. In order for an entity e to be a model, all that is required is that there be a way to use it to identify a domain of objects and a suite of relations on them. As far the modeling relation is concerned, these entities and relations may be physical, mathematical, or indeed just about anything you like. Moreover, this flexibility with respect to physical and mathematical

models is not just a technical quirk. That both physical and mathematical structures may stand as models is an important part of Copeland's account, a point about which he is quite clear:

> The construction [i.e. the labelling scheme] must be applicable not only to real hardware but also to merely conceptual machines. For example, we wish to say that each action of a Turing machine is the result of its configuration (i.e. the combination of its state and the scanned symbol). Unless we intend to speak metaphorically, the phrase 'is the result of' cannot be replaced here by 'is caused by', for the machine is a purely abstract entity. (Copeland, 1996, 341)

I submit that this account of computation is unificationist, not bifurcationist. On Copeland's view, whether and what a system computes is determined by the architecture-algorithm specifications of which it is a model. Consequently, the status and identity questions for both physical and mathematical computing systems are answered in terms of algorithm-architecture specifications.

To my knowledge, Copeland's view is perhaps the clearest illustration of unificationism in the literature. But at times other philosophers appear to flirt with unificationism, too. For instance, in some moods Piccinini appears to offer the mechanistic account of computation in a unificationist spirit. This account holds that computing systems are a kind of mechanism with teleological functions—or a 'functional mechanism', for short—whose function is computing (Piccinini, 2015). At times, Piccinini appears to suggest that the account applies in equal measure to both physical computing systems, such as digital computers, and mathematical computing systems, such as Turing machines:

> All paradigmatic examples of computing mechanisms, such as digital computers, calculators, Turing machines, and finite state automata ... perform digital computations. Thus, the mechanistic account properly counts

> all paradigmatic examples of computing mechanisms as such. (Piccinini, 2015, 143)
>
> [a] similar notion of functional mechanism [as that which applies to physical computing mechanisms] applies to computing systems that are defined purely mathematically, such as (unimplemented) Turing machines. Turing machines consist of a tape divided into squares and a processing device. The tape and processing device are explicitly defined as spatiotemporal components. They have functions (storing letters; moving along the tape; reading, erasing, and writing letters on the tape) and an organization (the processing device moves along the tape one square at a time, etc.). (Piccinini, 2015, 119-20)
>
> Whether abstract or concrete, Turing machines are mechanisms, subject to mechanistic explanation no more and no less than other mechanisms. (Piccinini, 2010, 290)

Thus on this view, the status and identity questions are answered in mechanistic terms. Whether and what a system computes amounts to being a certain sort of functional mechanism.[8,9]

I think that this is enough to get a feel for the distinction. As I remarked above, it seems to me that much work on physical computation has been guided by a tacit commitment to bifurcationism. Chalmers, for instance, simply assumes bifurcationism without argument. But once we isolate the unificationism/bifurcationism contrast

[8]Is fair to Piccinini? In other moods he seems downright hostile to unificationism (see, for instance, (Piccinini, 2015, 8-9)). Yet, even if this is right, it is not clear how to reconcile this with the outlook expressed in the previously cited passages, which seem to me clearly unificationist. In any event, while I think this points towards an interesting tension in Piccinini's view, I don't propose to get bogged down in exegesis at this stage. I'll return to this point again in Section 6.1, where I argue that Piccinini is committed to unificationism by his own lights.

[9]These views raise another issue worth clarifying. Much of the literature on physical computation takes a realist attitude towards the relevant mathematics: witness Chalmers' talk of computational objects, for instance. But we shouldn't read too much into this. Given the variety and subtlety of extant nominalization strategies, such as those developed in Burgess and Rosen (1997), I see no obvious reason why this literature couldn't be developed in nominalist terms without distorting the major issues. Accordingly, we can set aside questions about fundamental mathematical ontology and focus on issues internal to the philosophies of computer and cognitive science. For convenience, I will continue to adjudicate matters in realist vocabulary.

we can begin to see that matters are not so straightforward. Indeed, as I will argue next, there is a real worry that bifurcationism is misguided.

2.3 Troubles with bifurcationism

If I'm right that bifurcationism has been endorsed only tacitly, it is perhaps unsurprising that we find little explicit argument for it. However, those who appear to endorse it oftentimes emphasize the distinctively 'mathematical' subject matter of computability theory. Sprevak (2010, 262-3) expresses this idea vividly:

> Mathematical computation theory is a branch of pure mathematics and concerns relations between mathematical structures and objects. The 'computers' it studies are mathematical entities not physical systems. According to mathematical computation theory, a Turing machine is not a physical system; it does not 'perform' a computation in the sense that a physical system does.

And a bit later:

> The computations studied in mathematical computation theory are independent of how things are in the physical world. They are independent of empirical ink-marks, they do not 'take place' in time, or depend on the physical possibility of infinitely long tapes. Computers in mathematical computation theory are mathematical entities that bear certain relations, studied by that theory, to other mathematical entities, the functions they compute ... physical entities are not the subject matter of the relevant mathematical claims. Mathematical computaion theory does not say anything about physical systems.[10]

These remarks suggest the following sort of argument: (1) the mathematical theory of computation is concerned solely with mathematical entities, such as Turing machines, recursive functions, and the like. But (2) these are not physical entities

[10]It should be noted that what Sprevak calls 'the received view' is the claim that all physical computation essentially involves representation. This is distinct from how I use the term above, to refer to implementationist theories of physical computation generally. Sprevak's received view is just one kind of implementationist view, that is, just one instance of my 'received view'.

– they are 'independent of how things are in the physical world'. So (3) a theory of such systems is not automatically a theory of physical systems, in which case (4) bifurcationism follows.[11]

However, if this is the master argument for bifurcationism, it leaves much to be desired. One issue is that premise (1) comes perilously close to begging the question. Our topic is the extent to which the mathematical and physical theories of computation share a subject matter. To assume that the mathematical theory is concerned only with mathematical computing systems appears to assume that the status and identity questions concerning physical systems cannot be answered by that theory, which is just what we are wondering about. Another issue is that (2) presupposes a substantial view about the nature of mathematical entities, in effect that they are not physical entities of some sort or another. Yet it is not clear why one must endorse this claim in order to be a bifurcationist, for the unificationism/bifurcationism distinction presumably cuts across issues of fundamental mathematical ontology. That is, it would be rather surprising if endorsing bifurcationism (much less implementationism) required one to take this particular attitude towards the relevant mathematics.

But even setting these worries aside, the crucial move from (2) to (3) is suspect. Suppose it's true that the primary subject matter of computability theory are highly idealized mathematical computing systems. That doesn't show that computability theory can't also be a theory of certain non-idealized physical systems, too. Certainly it is not clear that this reflects how computability theorists conceive of their own subject matter, for at times they seem to be describing actual or possible physical devices. This outlook goes right back to Turing's foundational work on effective

[11]It would be unfair to attribute this argument, at least in this form, to Sprevak. But it does seem to me that this kind of reasoning lies behind bifurcationism.

calculability, for Turing's focus was, in the first instance, on the sorts of problems that could be solved by a human working effectively. In this respect the situation is quite unlike that found in, for example, the more exotic branches of set theory, where there is typically no pretension that the objects of interest are in any sense physical, idealized or not.

Moreover, it seems that computability theory actually has the resources to describe physical systems directly. This can be illustrated by looking at the standard definition of a deterministic finite automaton. On that definition, DFAs are five-tuples $A = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite set of input symbols, $\delta : Q \times \Sigma \to Q \cup F$ is a transition function, $q_0 \in Q$ is a start state, and F is a set of final states. As Rescorla (2014b) observes, this definition imposes no restrictions on the members of Q or F: all that is required that they be sets of states. Now, we might take the states to be purely abstract, but then again we might also take them to be states of a particular physical system. And in the latter case it's unclear why computability theory wouldn't describe the computational characteristics of physical systems. Indeed, it would be surprising if it didn't!

In light of all this, I am skeptical that bifurcationism can be motivated by an appeal to the supposedly 'mathematical' subject matter of computability theory. If we are going to endorse it, we'll need a more sophisticated argument. That's coming up. But first let's what can be said in favour of unificationism.

2.4 Three arguments for unificationism

This section argues that unificationism exhibits certain theoretical virtues, some of which concern philosophical theories generally and some of which are more specific

to theories of physical computation. In light of this, I claim that we shouldn't dismiss unificationism out of hand. Of course, that a theory has certain virtues only counts towards endorsing it if all else is equal. I don't pretend that the considerations presented here are demonstrative. In fact, I'll eventually argue that all else is not equal, and that for other reasons unificationism should ultimately be abandoned. But, taken together, I submit that these considerations make a strong *prima facie* case for unificationism. Strong enough, at any rate, to give the bifurcationist pause.

2.4.1 The value of unification

Many philosophers hold that more unified theories are preferable to less unified theories. This is a familiar point, so I'll be brief.[12] Unificationist theories of computation promise to unify computation, with a twist. Normally unified scientific theories are taken to unify diverse physical phenomena. But unificationism, as a thesis about the theory of computation, would unify both physical and mathematical phenomena. I see no reason why this sort of unification would be any less preferable to the usual sort, in which case a unificationist theory of computation is preferable to a bifurcationist one.

2.4.2 Extensional adequacy

Perhaps the strongest case for unificationism comes from computer scientific practice. If that practice marks no clear distinction between the theory of computation as applied to physical versus mathematical systems, then that is some reason to accept unificationism. In Chapter 1 I noted that this involves capturing both descriptive and

[12]See, for instance, (Kitcher, 1981, 1989; Swoyer, 1999).

explanatory practice. I'll discuss one important aspect of descriptive practice here, and then I'll turn to explanatory practice.

For a theory to capture descriptive practice it ought, at a minimum, to be extensionally adequate. What this comes to depends largely on the choice of paradigms and anti-paradigms, and here scientific practice is our guide. If the ordinary thought and talk of computer scientists marks no clear distinction between computations carried out by mathematical computing systems and physical computing systems, and if computer scientists routinely apply what appear to be the same computational notions to systems of either type, then this is some evidence that both physical and mathematical computing systems should be counted among the paradigms.

Some unificationists seem to interpret computational practice this way. For instance, on Piccinini's (2015, 12) interpretation, computer scientific practice suggests that plausible paradigms include "digital computers, calculators, both universal and non-universal Turing machines," while plausible anti-paradigms are "planetary systems, hurricanes, and digestive systems," among many others. Here both mathematical computing systems, such as Turing machines, and physical computing systems, such as digital computers, are taken to be paradigms. Thus it appears that an extensional adequacy requirement, coupled with *this* choice of paradigms, points towards a single account which captures both mathematical and physical computation. Unificationism naturally fits the bill.

2.4.3 Explanatory practice in computer science

We find similar tendencies in the explanatory practice of computer science. Oftentimes the same explanations are offered in order to explain the computational

Algorithm 1 Mul(x, y)

```
1: if x = 0 then
2:     return 0
3: end if
4: p ← y
5: while x > 1 do
6:     p ← p + y
7:     x ← x − 1
8: end while
9: return p
```

features of both mathematical and physical systems. Since this seems to me the more revealing datum, I'll take a minute to gnaw on it.

To fix ideas, focus on an explanation of a system's behavior in terms of the function it computes. And consider a mathematical computing system, such as an abstract register machine, that computes multiplication by repeated addition.[13] Let's suppose that the explanation why the device outputs $x \times y$ when given x and y is that it follows something like Algorithm 1. This simple explanation is familiar from elementary computability theory, and others like it are readily come by. The point to notice is that this pattern of explanation may equally well be applied to a contemporary digital computer. Just like an abstract register machine, contemporary digital computers manipulate strings of digits in finite memory locations. Given such a device, we can explain that it too outputs $x \times y$ when given x and y because it follows Algorithm 1.

In neither case does the explanation advert to the character of the items manipulated. In the register machine's case the strings are mathematical objects, perhaps defined in terms of pure sets, whereas in the digital computer's case the strings are

[13]See (Cutland, 1980) for such devices.

physical, perhaps realized as voltage levels in silicon. But these differences are irrelevant to the explanation why the devices multiply; what matters, from the perspective of computing multiplication, is that the algorithm takes x and y to $x \times y$. Whether this happens in sets or silicone is simply beside the point.

Of course, computer scientists are not just concerned to explain why a system computes some function. They also wish to explain why certain functions are uncomputable in principle, or are at least uncomputable in a feasible amount of time. Here too we find patterns of explanation applicable to both mathematical and physical systems.

A vivid example comes from work on performance bounds for comparison sorting algorithms.[14] It is known that any comparison sorting algorithm must make approximately $n \lg n$ comparisons to sort a list of length n.[15] A standard explanation notes that comparison sorting is equivalent to the task of guessing which permutation of a list one is given, if one can only 'see' two elements at a time. There are $n!$ permutations of an n-element list, and each guess, or comparison, eliminates half of the remaining permutations from consideration. The guessing procedure can be represented by a binary tree whose internal nodes are comparisons, and whose leaves are permutations.

A path from root to leaf corresponds to a sequence of comparisons—a sorting—in which case a lower bound on the number of comparisons required to sort the list corresponds to the minimum height of such a binary tree, and this is more or less $n \lg n$.

[14]Given an unsorted list $l_1, l_2, ..., l_n$ of items and a linear order $<$ on them, a comparison algorithm sorts by checking whether $l_i < l_j$ holds, and manipulating the list depending on the outcome.

[15]See (Cormen et al., 2001, 166-7) and (Knuth, 1998, 180-2).

Details aside, the important point is again that this explanation can be applied to any comparison sorting system, physical or mathematical. Anything that sorts by comparison must compare list members, and differences in the nature of the items compared are irrelevant as far as the explanation is concerned. That these differences are irrelevant suggest that unificationism, or so the thought goes.

So far, then, I've argued that the case for bifurcationism is uncompelling, and that the alternative, unificationism, is more attractive than one initially might suppose. To the extent that the received, implementationist view tacitly endorses bifurcationism, this argument threatens the received view as well. Nonetheless, as I argue next, unificationist theories face a serious challenge of their own.

2.5 No unificationism without trivialization

Perhaps the most significant problem unificationism is that it falls prey to the Putnam-Searle triviality worry, mentioned in Chapter 1. Triviality follows from two claims. The first is pancomputationalism, the claim that at one and the same time, every (or at least many) physical system performs every (or at least many) computations. The second claim links pancomputationalism to computational explanation, and holds that computational explanation succeeds only if a unique, or at any rate a select few, computations are performed at a time. Although in later chapters I'll offer a more careful appraisal of this argument, for now I'll simply assume that it is valid and that triviality ought to be avoided.[16]

[16]This way of formulating triviality issues ignores a few distinctions sometimes made between different kinds of pancomputationalism (Piccinini, 2015, ch. 4). While it is surely problematic if a given system performs every computation, it is perhaps tolerable if only a select few computations are performed at once. For instance, a given digital circuit may reasonably be said to compute both logical AND and logical OR, depending how one looks at it Dewhurst (2018). However, this

For the most part triviality arguments have been deployed against the received view. And with the received view in their sights, these arguments purport to show that every physical system implements every, or at least many, computations. I mention this only so that I can distinguish it from what I'm up to. The question I'm interested in is whether, and to what extent, triviality worries emerge for unificationism in particular. My claim is that they do, and that they do so in a particularly insidious form.

Here's the problem in brief. Unificationism requires that the account of computation applied to mathematical computing systems also apply to physical computing systems. Let's say that an account that satisfies this requirement 'accommodates' unificationism. Now, it seems clear that a theory which accommodates unificationism must characterize computation only in terms of characteristics shared by both physical and mathematical computing systems. If it didn't, it's hard to see how it would be a genuinely unificationist theory in the first place. But here's the rub: it appears that there is no account of computation which at once appeals only to shared characteristics, but which is also sufficient to block triviality. One can accommodate unificationism, or avoid triviality, but not both.

To argue for this in any detail we must get into the weeds on various accounts of computation. That happens in the next section. But to get a feel for the problem, the rest of this section sketches a triviality result for a simple structuralist theory of computation. With a few exceptions (e.g. (Schweizer, 2019b)), not many philosophers

observation is of no help to the unificationist. Even if we can tolerate some degree of pancomputationalism, my claim will be that unificationism entails an unacceptably strong version of it. Thanks to an anonymous referee for urging me to clarify this point.

endorse this account any more. But it nicely brings out the threat posed by triviality arguments, and the bind in which unificationists find themselves.

On the structuralist view theory, to compute is just to have a certain abstract, formal structure. This account straightforwardly accommodates unificationism, for it is plausible to think that both mathematical and physical systems have structural features of the right sort. But the problem is that structure is cheap, which leads directly to pancomputationalism and hence triviality.

To illustrate, consider Putnam's version of the argument. We'll describe computations in terms of finite state automata (FSAs). Given the structuralist account under consideration, we can think of FSA computations as describing computational structures. For instance, the two state automaton which moves from one state to another and then halts describes a computational structure, instantiation of which is a matter of having two parts (states, for instance) which stand in some sequential relationship.

We wish to establish that every physical system performs every FSA computation. This amounts to showing that every physical system goes through the state-transitions described by every FSA. So consider the particular FSA M with two states, A and B, which proceeds through the following state transitions: $A \rightarrow B \rightarrow A \rightarrow B \rightarrow A$. We define a 'maximal state' of some physical object O at a time to be its total intrinsic state at that time. Next, we consider five sequential maximal states of O: $M_0 \rightarrow M_1 \rightarrow M_2 \rightarrow M_3 \rightarrow M_4$. The idea is to define two new states of O, A_O and B_O, so that these new states evolve in a way that corresponds to the state transitions described by M.

The following will do: $A_O = M_0 \vee M_2 \vee M_4$; $B_O = M_1 \vee M_3$. Thus O proceeds through a series of state transitions capture by M: $A_O \rightarrow B_O \rightarrow A_O \rightarrow B_O \rightarrow A_O$. So O performs M's computation. But since this kind of construction can be had for any FSA and any physical entity, the argument generalizes and triviality follows.

Of course, nothing about this construction is novel. The important point is how it interacts with the background theory of computation: while the structuralist account smoothly accommodates unificationism, it does so at the cost of triviality. The question I take up next is whether some other account can be found which both accommodates unificationism and resists trivialization.

2.6 Four accounts of computation

The implementationist literature contains a variety of strategies for avoiding triviality, and this section explores whether any of them can be appropriated in the service of unificationism. I focus on accounts which hold that computation essentially involves causation, counterfactual dependencies, teleological functions, or representation, respectively. In each case I argue that the proposal fails to avoid the problem outlined in the last section. Some fail to avoid triviality, while others accommodate unificationism only at great cost. The lesson I draw is that the prospects for unificationism seem grim.

I should make a couple more clarificatory points before getting down to business. First, my aim is not to show the impossibility of a unificationist theory of computation. I see no way to establish such a strong result. Rather, and more modestly, my aim is to shift the burden onto the unificationist to articulate a theory that at once accommodates unificationism and avoids triviality. The proposals I investigate below

seem to me to be the most plausible options, but they are not the only ones available. I leave open the question whether some other account can do the job. But to the extent that the options canvassed here stand the best chance of avoiding triviality, unificationism is in trouble.

Second, it is worth highlighting how the responses to triviality investigated below differ from those employed by the received, implementationist view. Because they endorse bifurcationism, proponents of implementationism are under no pressure to employ constraints that apply to mathematical computations. For them, the problem is to just say which physical objects are allowed to figure in the implementation relation. Accordingly, they are free to impose constraints on the physical side which may make no sense in the context of mathematical computation. The unificationist, by contrast, has an altogether different task to pull off. They must impose constraints that apply equally to physical and mathematical computing systems. And this, we will see, is the source of their troubles.

2.6.1 The causal-mechanical account

A natural first response to triviality requires that the computational structure of a physical system track its causal structure.[17] A related suggestion is found in the mechanistic account of computation, which holds that computing systems are mechanisms.[18] Given their close resemblance, we can treat these proposals together. On this view, to have a computational property is just a matter of having a certain causal-mechanical structure. The reason why an arbitrary physical object doesn't

[17]See (Chalmers, 1996; Godfrey-Smith, 2009; Scheutz, 2001) for proposals in this vein.

[18](Milkowski, 2011; Piccinini, 2015). Depending on one's views about causation and mechanism, the mechanistic conception might collapse into the causal conception; see (Glennan, 2017) and (Woodward, 2013) for discussion.

perform every computation is that the computational 'states' cited by triviality arguments are not causally related, or are not genuine mechanistic components of the system. Pancomputationalism (hence triviality) fails because 'pan-causation' and 'pan-mechanism' fail, or so the thought goes.

But the question is whether the causal-mechanical account accommodates unificationism. And it seems not, at least on the face of it. Nowadays Turing machines are defined as consistent sets of 5-tuples $(Q_i, \sigma_1, \sigma_2, m, Q_j)$ where Q_i and Q_j are states, σ_1 and σ_2 are letters in the machine's alphabet, and m is an instruction for moving the read/write head. Sets of 5-tuples fix a set of state-space transitions, which capture the behavior of the machine at every possible stage of operation. Now, whether we identify Turing machines with sets of 5-tuples or with a set of state-space transitions, each of these items is constructed, or at least constructible, out of pure sets. Such pure set theoretic entities are abstract, non-causal, and non-spatiotemporal. But given this view of Turing machines it is hard to see how the causal-mechanical theory accommodates unificationism, for the simple reason that Turing machines, so construed, lack causal-mechanical characteristics altogether.

One response to this is to abandon unificationism. Piccinini takes this tack in his more pessimistic moods, writing that he is "after an account of computation in the physical world" and that "[i]f Turing machines and other mathematically defined computational entities are abstract objects ... they fall outside the scope of my account" (Piccinini, 2015, 9). But it is unclear how to square this with his later claim, in the very same book, that the mechanistic account correctly counts Turing machines as computing mechanisms. Moreover, it seems that this move is unavailable to Piccinini by his own lights, for "[t]he primary aim of the mechanistic account is

doing justice to the practice of computer scientists and engineers'" (Piccinini, 2015, 118). As I argued earlier, that practice supports unificationism, at least *prima facie*. So this sort of response is unavailable to anyone impressed by computer scientific practice, as Piccinini appears to be.

A more radical response denies that Turing machines are purely mathematical entities. Instead, we should regard them as idealized physical entities, which retain some, but not all, of the physical properties of their unidealized counterparts. Copeland and Shagrir (2011) call this view 'Turing-machine realism'. As they explain:

> Turing-machine realism recognizes an ontological level lying between the realization level and the level of pure-mathematical ontology. We term this the level of notional machines. At this level are to be found notional or idealized machines that are rich with spatiotemporality and causality. Copeland and Shagrir (2011, 234)[19]

Such a view would allow the unificationist to endorse a causal-mechanical theory of computation, unifying computation in causal-mechanical terms.

One worry about this maneuver concerns the status of the idealized machines. It is far from clear how we are to make sense of their 'in between' ontological status. Are they abstract objects, or mental entities, or what?[20] This question demands an answer before we can legitimately claim that Turing machines 'have' causal features.

However, even if we can make sense of notional machines, there appears to be a general problem which undercuts any attempt to recharacterize Turing machines, spatiotemporally or otherwise. The trouble is that the computational explanations surveyed above apply not just to physical systems, but also to pure mathematical

[19]It should be noted that Copeland and Shagrir don't endorse Turing–machine realism, but merely flag it as a live option.

[20]See (Thomson-Jones, 2010) for critical discussion of a related proposal found in (Giere, 1988).

computing systems. Those systems, such as the pure set-theoretic counterparts of Turing machines, haven't gone anywhere. And if unificationists take this practice seriously, which they presumably do, they must furnish a theory which accounts for Turing machines in their pure set-theoretic guise too: adding a new kind entity 'in between' physical systems and the level of pure mathematics doesn't discharge that task. For this reason it seems to me that Turing machine realism is a dead end, and that we should explore other options.

2.6.2 The counterfactual account

Copeland (1996) explicitly disavows a causal-mechanical theory of computation in light of the difficulties raised in the last section. He complains that causal-mechanical theories are "intolerably narrow" because they don't capture abstract Turing machine computations Copeland (1996, 353). Despite this, he maintains that causal-mechanical accounts do capture something important about computation, namely that later computational states counterfactually depend upon earlier states. This is true even for Turing machines, for there certainly seems to be some sense in which a Turing machine's later states depend on its earlier states (plus tape contents).

One immediate question is how we should understand the notion of counterfactual dependence Copeland has in mind. The suggestion seems to be that it is a generalization of causal dependence. But whereas causal dependence relations holds only between spatiotemporal particulars, such as physical events, counterfactual dependence relations may hold between either spatiotemporal particulars on the one hand, or between non-spatiotemporal particulars, such as the 'events' in a Turing machine computation, on the other.

Now, this doesn't clear everything up. In what sense do Turing machine operations involve events, for instance? But Copeland thinks we can skirt these sorts of questions. The target notion of counterfactual dependence is logically equivalent to causal dependence, in the sense that both relations underwrite counterfactual assertions about the behavior of computing systems. So, to determine whether the target dependence relation obtains it is enough to determine whether a system satisfies certain 'computational counterfactuals'. Thus, on this account whether a system computes reduces to the question whether the system satisfies certain counterfactual assertions about its behavior.

How does this avoid triviality? We already remarked that most of the physical states appealed to by the triviality argument are not causally related; by similar reasoning it is not hard to see that they will fail to support counterfactuals as well. For example, recalling the construction from above, the following is plausibly false of the physical object O:

1. If it were the case that O was in state A_O, it would transition into state B_O.

In which case O doesn't compute, as desired. And to see that the proposal correctly captures mathematical computations, consider an abstract register machine M with an instruction register whose contents contain a code for the next operation to be performed. M operates by reading the contents of the instruction register and then performing the encoded operation. The device is defined to be sensitive to contents of the instruction register, so that differences in register contents make for differences in the operations performed. Accordingly, M satisfies computational counterfactuals of the form:

2. If it were the case that at some time M's instruction register held such-and-such instruction, M would perform so-and-so operation.

Thus the counterfactual proposal accommodates unificationism while avoiding triviality.

But it seems to me that this proposal runs up against two difficulties. First, Copeland's own version succumbs to a revenge triviality argument. Standard triviality arguments can be tweaked to show that every sufficiently complex physical system computes every computable function while satisfying appropriate computational counterfactuals.[21] So Copeland's account fares no better than the simple structuralist account from before.

It gets worse. There are reasons to be pessimistic that any other counterfactual account will be forthcoming any time soon. The trouble is that an account of computational counterfactuals general enough to apply to the operations of abstract computing systems must also be an account of counterpossibles, and it is not at all clear how to understand counterpossible assertions about the behavior of mathematical computing systems.

To illustrate, consider a Turing machine M described informally by the following two instructions:

- If in state Q_0 and reading a 1, write 0, move the read/write head to the right, and go to state Q_0.

[21] I develop one such argument in (Curtis-Trudel). In brief, the trouble is that Copeland supplies a standard satisfaction-based semantics for his counterfactual conditionals. Absent further constraints on satisfaction—and Copeland doesn't supply any that even remotely do the trick—it's not hard to devise deviant physical models of these counterfactuals. Moreover, on reflection this isn't even all that surprising, since completeness guarantees the existence of models, and it's not hard to manipulate particular physical models in order to come up with arbitrarily many distinct physical interpretations of the counterfactuals.

- If in state Q_1 and reading a 1, write 1, move the read/write head to the right, and go to state Q_1.

This machine erases a contiguous block of 1s and then halt. In line with the counterfactual account, we presumably want M to satisfy counterfactuals such as

3. If M had been in state Q_1 and read 1, it would write 1 and go into state Q_1.

However, if we make the standard assumption that M starts in state Q_0, there is no possible sequence of state transitions which will take it into state Q_1. So the antecedent of (3) is impossible in a strong mathematical sense, making (3) not merely counterfactual, but counterpossible.

This is fine as far as it goes, but how should we understand (3)? It seems inappropriate to endorse the view that all such counterpossibles are vacuously true, as is sometimes suggested (Lewis, 1973; Williamson, 2007). This is because we presumably wish to distinguish (3), which is intuitively true, from (4), which is intuitively false:

4. If M had been in state Q_1 and read 1, it would write 0 and go into state Q_1.

Yet we cannot capture the apparent truth of (3) and falsity of (4) in the usual possible worlds semantics, for the familiar reason that there is no possible world in which M is in state Q_1.

Of course, the unificationist might make various maneuvers at this point. One is to introduce impossible worlds to distinguish between (3) and (4) (Berto and Jago, 2019). Such moves are not unheard of, but it is no understatement to say that impossible worlds are problematic and perplexing. It is far from clear whether a counterfactual conception of computation can be worked out in such a framework.

But I think it is enough for the present, burden-shifting argument, to notice that if this is the road to non-triviality, then friends of unificationism have their work cut out for them. One can be forgiven for exploring other routes.

2.6.3 The teleological account

Some versions of the mechanistic account hold that computing systems are not just mechanisms, but mechanisms with teleological functions. While the causal-mechanical aspects of this view sits poorly with unificationism, perhaps teleological functions alone suffice to avoid triviality. According to the view explored next, possession of certain teleological functions is necessary for computing and sufficient to avoid triviality.[22]

One task is to specify the notion of teleological function at play. I will focus on the account developed in (Piccinini, 2015, ch. 6), since it is apparently designed to apply to both digital computers and Turing machines. A first-pass account is couched in terms of objective goals, which include things such as survival and reproduction:

> A teleological function is a stable contribution to an objective goal of organisms by a trait or an artifact of the organisms.

But even if we can make sense of the thought that Turing machine operations are a 'trait' or 'artifact' of an organism, it is frankly incredible to think that such operations stably contribute to anyone's survival and reproduction, even, I regret to report, those of the professional computer scientist.

Perhaps recognizing this, Piccinini broadens the account to include 'subjective goals' (Piccinini, 2015, 116). These include desires, among other things, so that

[22] An anonymous reviewer asks whether anyone nowadays would really endorse a teleological view of mathematical computing systems. The short answer is apparently 'yes', for Piccinini appears to; see (Piccinini, 2015, ch. 7) and the passages quoted earlier.

something contributes to a subjective goal if it stably contributes to the satisfaction some desire:

> A teleological function (generalized) is a stable contribution to a goal (objective or subjective) of organisms by either a trait or an artifact of the organisms.

So while Turing machine operations might not stably contribute to computer scientists' survival and reproduction, it's not completely unreasonable to think that they might help to satisfy certain of their desires for example, their desires for knowledge about computable sets, Turing degrees, and so on.

But it seems to me that this account faces a number of problems. One is that there are not enough desires to go around. There are only finitely many actual desires, but denumerably many Turing machines. If Turing machines compute only by virtue of satisfying some actual desire, then there will be denumerably many Turing machines which, whatever else they do, don't compute. Surely a bad result!

We might try to get around this by appealing to the desires of possible agents, so that a Turing machine computes if there is some possible agent some of whose desires would be stably satisfied by that machine's operations. Now the trouble is that the suggestion overshoots. There are many possible agents — enough, let us grant, that every Turing machine's operations satisfy some possible agent's desires. If stably satisfying the desire of a possible agent is sufficient to have a teleological function, then given the abundance of possible agents, any object can have a teleological function. In particular, now paradigmatic non-computing systems such as rocks, walls, and pails of water will have teleological functions in this generalized sense, which is just what we want to avoid.

The proper response to this worry, it seems to me, is to grant that generalized teleological functions are in principle universally realizable, but nevertheless deny that this poses a problem for a theory of physical computation in particular. A system computes some mathematical function f only if computing f is one of its teleological functions. As long as the conditions on having computing f as a teleological function are stringent enough, an abundance of generalized teleological functions does not threaten computational explanation. Triviality is avoided not because generalized teleological functions are hard to come by, as it were, but because the particular teleological function of computing f is.

But whether this maneuver succeeds turns crucially on just what it is to compute f. A notion couched in terms of formal string manipulations plausibly applies to Turing machines, but encounters a problem familiar from the structuralist account: how do we determine which parts of a system are strings? There are many ways to arbitrarily identify parts of a physical entity as strings, and it is straightforward to reverse-engineer deviant interpretations according to which computes every function. So we're back to where we started. Moreover, we cannot impose causal-mechanical constraints on string identification, for in this case we're back to the problems encountered with the causal-mechanical account. So, absent some other account of what it is to compute f, it seems the teleological account fares no better than the others.

2.6.4 The representational account

The last account I will consider holds that to compute is, at least in part, a matter of having certain representational features.[23] The version considered here takes this

[23]A number of philosophers have endorsed accounts of computational implementation which are either partly or wholly representational; see (Rescorla, 2014b; Sprevak, 2010; Shagrir, 2020).

to involve representing entities in some external domain, such as a system's distal environment or a set of mathematical objects.

One point in favour of this account is that it is plausible to think that representational notions feature in the mathematical theory of computation (Rescorla, 2015). Turing machines directly compute string-theoretic functions. Computation over non-linguistic domains is characterized relative to a mapping from linguistic entities to non-linguistic entities. When such a mapping is in place, it is natural to regard the linguistic entities as representing the non-linguistic entities. In the case of computation over the naturals, for instance, strings of digits are taken to represent natural numbers. However, the notion of representation that emerges from the mathematical theory of computation is quite thin: all that is required for a Turing machine to represent is that there be an effective mapping from its language to some non-linguistic domain (Shapiro, 1982). Such mappings are abundant; indeed, if the linguistic and non-linguistic domains are denumerable, there will be denumerably many. But plainly pancomputationalism is unavoidable if this is the notion of representation underwriting physical computation, for there are simply too many mappings from physical systems and their states to external entities.

At this point we could cast about for a more stringent account of representation, so that physical systems stand in determinate representational relations to comparably few entities. A causal constraint is natural but unavailable for familiar reasons: a causal notion of representation will not obviously apply to abstract Turing machines, and at any rate there seems to be little sense in which a microprocessor's states stand in causal relations to abstract numbers. Similar points apply to other robust

conceptions of representation too, whether they appeal to functional roles, teleological functions, or whatever.[24]

A different option is to take representational notions as primitive (Burge, 2010; Rescorla, 2013). From this perspective, we shouldn't attempt to characterize representational notions in non-representational terms. Rather, representational notions earn their keep because they play an indispensable explanatory role in our scientific theories. I'll make just one remark about this strategy. If attributions of representational properties are warranted in virtue of the explanatory work they do, then an immediate problem is that there appear to be physical computing systems, such as simple embedded systems in ordinary appliances, whose behavior and character can be exhaustively explained without appeal to representational notions (Rescorla, 2014b; Piccinini, 2008). If this is right, then by the primitivist's lights we have no reason to ascribe representational properties to them. But then it begins to look very difficult to use representational notions to solve the unificationist's woes, for we re-encounter the problem that not every computing system can be brought under a single (primitive representational) rubric. Once again, there seems to be no way to hold on to both unificationism and a substantive theory of physical computation.

2.6.5 Summary

The counterfactual and representational accounts accommodate unificationism but at the cost of triviality. The causal-mechanical and teleological accounts might avoid triviality, but don't accommodate unificationism. Perhaps the unificationist

[24]Note that I am not asking for a naturalization of representation, whatever that amounts to, but a guarantee that representation is non-trivial. Such a guarantee might come on the heels of a naturalistic account of representation, but then again it might not. The worry is that absent an appeal to these other notions, no such guarantee is forthcoming. Naturalization is a separate issue.

has another trick up their sleeve, but I think by now we've seen enough. It is doubtful that a single theory can supply satisfactory answers to the status and identity questions for both physical and mathematical machines. Unificationism has got to go. Instead, we should be bifurcationists about computation.

2.7 Life after bifurcation

Given bifurcationism, how should we approach the status and identity questions concerning physical computing systems? Note that bifurcationism per se does not mandate Chalmers' mathematics-first approach. Indeed, it is neutral between a few different possibilities. One is the received, mathematics-first view. A second reverses this order of things, and addresses the status and identity questions with respect to mathematical computing systems in terms of prior answers to their physical counterparts. We might call this the 'mathematics-last' view. A third possibility denies that either pair of questions ought to be answered in terms of the other — this would be 'mathematics-never'. I'll close with some brief reflections on these alternatives, starting with the last, which seems to me the least promising of the three.[25]

2.7.1 Mathematics-never

On this approach neither the mathematical nor the physical theories of computation should be developed in terms of the other. In its strongest form, this approach holds that these are simply two different theories, with different goals and theoretical

[25]Recall that these are, first and foremost, methodological proposals about how we should address the status and identity questions. However, questions of methodological priority aren't the only ones we might ask when thinking about the relationships between physical and mathematical computation. For instance, we might wonder whether one or the other is 'basic' or 'fundamental' than another, in any of a variety of senses, e.g., conceptually, metaphysically, or epistemologically. I do not have space to investigate these alternatives here.

concerns. Although probably no holds quite such a strong view, some philosophers flirt with more attenuated versions of this approach. For instance, Shagrir rejects what he calls the 'logical dogma', which holds that"there is a strong linkage between the mathematical theories we find in logic and computer science ... and physical computation" (Shagrir, 2022, 5).

However, It's hard to see what can be said for this view. For one thing, it leaves us no better off with respect to theories of physical computation than we were when we started. Indeed, if anything, we're in a worse position, for now we must craft such a theory without the help of a prior mathematical theory. Although I do not have an argument that no such theory is possible, it is an open question at this stage whether one can be developed. Second, and perhaps more seriously, this proposal sits badly with certain aspects of computational practice. Computer scientists *do* characterize physical computing computing systems in broadly mathematical terms. It is hard to see how a mathematics-never approach would capture this aspect of computational practice.

2.7.2 Mathematics-last

According to a mathematics-last approach, questions about the status and identity of mathematical computing systems ought to be addressed by appeal to a prior theory of physical computation. Here we 'apply' the physical theory to the mathematics, rather than the other way around.[26]

What could motivate such a view? Here's one sketchy suggestion. As we know, the task facing Church, Gödel, Turing, and others in the 1930's was to give a precise mathematical statement of the intuitive idea of a worker proceeding effectively.

[26]For proposals in roughly this neighbourhood, see (Joslin, 2006) and (Cleland, 2001, 2002).

Turing's landmark characterization proceeded, in the first instance, by careful reflection on the abilities of actual human workers (Sieg, 2009). Historically, at least, mathematical conceptions of computation emerged from reflection on computation in actual physical systems. This suggests that in order to say what it would be for a Turing machine to compute, we must already grasp *some* notion (however inchoate) of a physical system computing. Thus, on the envisioned theory, we begin with a notion of physical computation—e.g., of a human working effectively —and refine it through a process of idealization and abstraction to arrive at a mathematical notion of computation— e.g., of a Turing machine computation. Crucially, these mathematical objects count as computational only to the extent that they are idealizations or abstractions of physical computing systems.

One advantage of this approach is that it does not obviously run into the Putnam-Searle triviality worry, because our pre-theoretical conceptions of physical computation are plausibly non-trivial. Plainly not every physical system is a human working effectively, for instance. Because, on this approach, a mathematical theory of computation is grounded in a prior physical theory, the question of 'applying' the mathematics to the physical world does not arise. *A fortiori*, neither does the threat of over-application.

However, it seems to me that this approach also faces a few substantial hurdles. For one thing, it runs into the two problems raised in connection with the mathematics-never approach. A third concern is that some mathematical computing systems do not bear a straightforward connection to any physical computing system. If this is right, then there will be mathematical computing systems unaccounted for by this approach. For instance, there are systems for which there is a well-worked

out formalism but whose physical realizations have yet to be constructed. In some cases, such as quantum computers (Nielsen and Chuang, 2010), this is because current technology is not up to the task. In other cases it's because the formalism requires physically unrealizable operations. Infinitary Turing machines are perhaps the most plausible example of this Hamkins and Lewis (2000); Copeland (2002). Again, while I do not have an argument that some mathematics-last approach, suitably elaborated, might account for such cases, it seems to me that the prospects for such an approach are dim.

2.7.3 Mathematics-first

So we're left with the mathematics-first approach. On this approach, we address the status and identity questions by appeal to a prior mathematical theory of computation. The central task of this approach is to delineate a relation of computational implementation or realization linking physical systems to abstract mathematical computations. This approach has certain obvious advantages over the other two bifurcationist proposals. As Chalmers is keen to note, we already have a well defined mathematical notion of computation, and it is a natural starting point for philosophical theorizing about physical computation. However, perhaps the most significant obstacle for this approach is that it faces triviality worries of its own. Thus, the next item on the agenda is to develop a response to triviality on behalf of the implementationist. That is the primary task of the next chapter.

By way of closing, let me return to the title question. Why do we need a theory of implementation? The short answer is: it's the best we've got.

Chapter 3: Computation in context

3.1 The state of play

In the last chapter I argued that unificationist theories of physical computation should be rejected on the grounds that they succumb to a version of the triviality problem identified by Putnam and Searle. I further argued that in light of this we should be bifurcationists, and that in particular we should adopt the mathematics-first implementationist approach advocated by Chalmers and others. However, this may seem to be little progress, given that implementationist theories themselves run up against Putnam and Searle's argument. The task of this chapter is to develop a response to their argument on behalf of the implementationist.

Putnam and Searle's original argument targets the simple mapping account of physical computation. On this account, recall, a physical system implements a computation if the physical structure of the system 'mirrors' the formal, mathematical structure of that computation. 'Mirroring' is typically cashed out in terms of a formal, structural relationship such as isomorphism, so that we have:

The Simple Mapping Account

A physical system P implements a computation M if and only if:

1. There is grouping of states of P into state-types and a function f (a 'realization' or 'implementation' function) mapping state-types of P to states of M, such that

2. Under f, the state transitions of P are isomorphic to the formal state transitions of M; i.e., whenever P is in state p_1, where $f(p_1) = m_1$, and $m_1 \rightarrow m_2$ is a formal state transition, then P goes into state p_2, where $f(p_2) = m_2$.

Putnam and Searle's argument has met fierce resistance, of course, and a variety of defensive maneuvers have been deployed to blunt the force of their attack. Perhaps the most popular maneuver is to beef up the simple mapping account by adding further necessary conditions on implementation (Sprevak, 2019). Such an approach yields a 'complex' mapping account, as mentioned in chapter 1. No complex mapping account enjoys widespread acceptance, however, and even the most plausible versions would, if successful, vindicate the use of computational notions for a limited class of physical systems only. For instance, one well-known response imposes a causal condition on implementation in an effort to carve out a notion of computation appropriate for theorizing about the mind (Chalmers, 1996, 2011). But, even if this response blocks triviality for a restricted class of physical systems, it does not obviously resolve the more general challenge. It is a local solution to a global problem.

This chapter takes a different approach to the Putnam-Searle challenge. Given certain plausible assumptions about the nature of computational ascription, I argue that the Putnam-Searle argument loses much of its force. These assumptions concern both the *relativity* and *context-sensitivity* of computational ascription: specifically, computational notions apply to a physical systems only relative to a contextually determined way of regarding that system computationally — what I shall in this chapter call a 'labelling scheme'. But, relative to such particular labelling scheme, there is little reason to think that computation is trivial. Or so I shall argue.

Here is the plan of the chapter. Sections 2 and 3 reconstruct the Putnam-Searle argument in some detail. Sections 4 and 5 develop the core of my response to their argument. Section 6 elaborates on my response and situates it with respect to others in the literature. Section 7 concludes.

3.2 Pancomputationalism

On my reconstruction, triviality arguments proceeds in two main steps. The first establishes pancomputationalism — the claim that every physical system implements every computational architecture. The second moves from pancomputationalism to the claim that physical computation is trivial. I consider the pancomputationalist step in this section, and the second step in the next.

Pancomputationalism is the claim that every physical system implements every computation.[27] It is straightforward to verify that the simple mapping account entails pancomputationalism. For, according to the simple mapping account, a physical system implements a computation if there is a one-to-one mapping from that system's components, states, or parts to the components, states, or parts of a computation, such that, under that mapping the two are isomorphic. Thus, to show that the simple mapping account entails pancomputationalism, it is enough to show that every physical system is isomorphic with every computation. And this follows under a few very modest additional assumptions.

[27]This is sometimes called 'unlimited' pancomputationalism (Piccinini, 2015, ch. 4). In practice, triviality arguments typically aim at a weaker conclusion than this. For instance, Searle argues that every 'sufficiently complex' system computes every program, while Putnam argues that every 'ordinary open system' realizes every deterministic finite automaton. However, given the way triviality arguments are typically deployed, the differences between these claims are negligible. For this reason I will take the target of the pancomputationalist step of the argument to be the claim that every physical system simultaneously implements every computation.

3.2.1 Push-Through

The first is a basic mathematical technique for defining isomorphic copies of a given structure. Although it goes by various names, it is known in model theory as 'Push-Through' (Walsh and Button, 2018). Structures in the relevant sense are set-theoretic objects comprising a domain of objects plus a suite of functions and relations defined thereon. For simplicity, I'll focus on the case where they contain a single unary function each.

First, recall the definition of isomorphism. Let $\mathcal{M} = \langle M, T_M \rangle$ and $\mathcal{P} = \langle P, T_P \rangle$ be structures, where $T_M : M \to M$ and $T_P : P \to P$ are unary functions. These structures are isomorphic just in case there is a structure preserving bijection $f : M \to P$ between them, so that for any $m \in M$, $f(T_M(m)) = T_P(f(m))$. For illustrative purposes, we can think of these structures as a mathematical computation and a physical system, respectively, where T_M is a transition function on mathematical states and T_P a transition function on physical states.

Now, if a computation $\mathcal{M} = \langle M, T_M \rangle$ plus a set P of physical states are given, Push-Through tells us how to define a new physical structure $\mathcal{P} = \langle P, T_P \rangle$ isomorphic with $\mathcal{M}$. The only requirement for applying Push-Through is that P and M are equinumerous. So suppose for the moment we have established that M and P are equinumerous. Since two sets are equinumerous just in case there is a bijection between them, let $f : P \to M$ be such a bijection. Given f, we define the physical transition function T_P on P as follows: say that $T_P(p_1) = p_2$ just in case $T_M(f(p_1)) = f(p_2)$. That is, the physical transition function is constructed by looking at the image of T_M in M under f. It is trivial to verify that the new structure $\mathcal{P} = \langle P, T_P \rangle$ is isomorphic with the given structure $\mathcal{M}$.

3.2.2 Equinumerosity

Thus, if we can show that the states of some mathematical computation are equinumerous with the states of some physical system the physical states, by Push-Through we can construct a physical state transition function which perfectly mirrors the action of the formal state transition function T_M. At this point we appeal to the empirical assumption, not unreasonable, that physical systems have vastly many components (e.g., microphysical states or parts). These components are the raw material out of which a set of physical state types are defined. The force of this assumption is to ensure that, no matter how many states a given mathematical computation has, we will always be able to find a set of physical state types equinumerous with those of a given computation.[28]

Of course, we must also ensure that the underlying physical states *can* be grouped into state types. Here too different strategies are possible. Searle suggests that any 'assignment' of computational states to microphysical states by an observer implicitly defines a physical state type. Alternatively, Putnam claims that any disjunction of physical states constitutes a physical state type. Other approaches are possible. However we proceed, what ultimately matters is that there is some mechanism for defining state types out of microphysical states.

Thus pancomputationalism follows from the simple mapping account with only basic model theory and a modest empirical assumption. Briefly: let P be any physical

[28]Searle makes this assumption directly. Putnam is more circumspect. He invokes two physical principles, the principle of continuity and the principle of noncyclical behaviour, whose joint effect ensures that, over any interval of time, any ordinary open physical system proceeds through continuously many particular physical states. More recently, Hemmo and Shenker (2019) have argued that this assumption is justified on quite general statistical mechanical grounds. For this reason they suggest that spancomputationalism is a theorem of statistical mechanics. Note additionally that we needn't assume that physical state types be defined out of microphysical states. What ultimately matters is not that they are microphysical, but that there are enough of them.

system and C be any computation. By the empirical assumption, P's components (states, etc.) are equinumerous with those of C. By Push-Through, the two are isomorphic. Hence, by the simple mapping account, P implements C. Pancomputationalism follows, since P and C were arbitrary.

3.3 From pancomputationalism to triviality

Putnam and Searle hold that pancomputationalism renders a theory of implementation 'trivial'. Trivial in what sense? In my estimation, this charge is best understood as a complaint about the theoretical status of the notion of computation, in that any account which entails pancomputationalism suffers from a variety of distinct but related theoretical failures. These failures fall into roughly two camps, concerning an account's descriptive and explanatory ambitions, respectively. I'll take each in turn.

3.3.1 Descriptive Defects

The argument for pancomputationalism relies on extraordinarily weak empirical assumptions. The only empirical facts cited in the argument are facts about number of component states or parts of physical systems. The argument does not rely on facts about the character of these states or parts, nor does it rely on facts about their arrangement, interactions, or activities. Nor, for that matter, does it rely on facts about the gross structure of physical reality, concerning things such as the distribution of matter or which natural laws obtain. In light of this, it is perhaps no surprise that any theory that entails pancomputationalism confronts a battery of problems concerning its descriptive capabilities:

- *Lack of empirical content.* The empirical demands required to establish pancomputationalism are very modest, concerning only the cardinalities of a system's components. Insofar as these demands would be met in almost every physically possible situation, it would thus appear that theories of physical computation lack empirical content almost entirely.

- *No empirical discoveries.* Concerns about empirical content are closely related to the question whether we could empirically discover the computation implemented by a system (cf. (Searle, 1992, 208)). While one might have thought that a great deal of empirical investigation would be required to determine whether a system computes, if pancomputationalism holds such questions are answerable largely from the armchair, by reflecting on the size of the system in question.

- *Extensional inadequacy.* Whereas the computational sciences ascribe computational properties to comparably few physical systems — chiefly engineered devices and cognitive systems — pancomputationalism entails that these notions apply to every physical system. Thus while the account correctly classifies paradigmatic cases, it incorrectly classifies anti-paradigms more or less across the board.

- *Widespread predictive failure.* Knowledge about what a system implements ought to be predictively useful. Knowledge that a robot implements *this* pathfinding algorithm would presumably allow one to predict that it will take *that* path through an obstacle course. Yet it is hard to see how such knowledge can help

one predict anything, if the robot implements everything. For the robot simultaneously implements a wide variety of different pathfinding algorithms, some of which may find quite different paths through the course. It is thus hard to see what would justify the prediction that the robot will take one path instead of another.

- *Unwarranted retrodictive success.* While one might have thought that retrodiction of past pathfinding behavior is the sort of thing that we could in principle get wrong, under pancomputationalism our retrodictions can't help but be true. This is because, no matter what path the robot in fact takes, there will be some pathfinding algorithm it implements under which it would have taken that path.

3.3.2 Explanatory Failures

A second and perhaps more serious concern is that pancomputationalism drains theories of implementation of their explanatory power. Here is how Mark Sprevak describes the problem with respect to explanation in cognitive science:

> Cognitive science explains particular aspects of behavior and mental processing (behavioral or psychological "effects") by appeal to the brain implementing specific computations. Specific effects occur because the brain implements one computation rather than another. This explanatory methodology is threatened by the triviality results. If implementation is trivial, then no distinctive computations are implemented by the brain. The brain, like almost every other physical system, implements almost every computation. Explaining psychological effects by appeal to distinctive computations cannot work because there are no distinctive physical computations. (Sprevak, 2019, 186)

While I am in broad agreement with Sprevak's diagnosis, it can be refined by more sharply distinguishing two distinct explanatory scenarios. Both highlight the

contrastive character of computational explanation. In *intersystemic scenarios*, computations explain differences between different systems or system kinds. For instance, the fact that the visual system computes depth from binocular disparity partly explains why healthy individuals exhibit competent grasping behavior while those with various cortical injuries — not to mention manifestly non-cognitive systems such as rocks, walls, and pails of water — do not. In *intrasystemic scenarios*, computations explain contrasts between the actual and merely possible properties or behavior of a single system or system kind. Here we might explain why a robot takes one path through a maze (rather than another) because it implements this path-finding algorithm (rather than that one).

Successful explanation in either case presupposes a contrast (Hitchcock, 2013). The fact that healthy visual systems compute some depth-extraction function explains differences in grasping behavior or perceptual capacities only if those same depth-extraction computations are not also implemented by injured visual systems (or rocks, walls, etc.). Similarly, the fact that the robot implements some pathfinding algorithm explains why it takes a certain path only if it doesn't also implement another algorithm which charts a different path. The trouble is that under pancomputationalism no such contrasts obtain. Rocks and injured visual systems implement exactly the same computations as healthy visual systems, and the robot implements every pathfinding algorithm no matter what path it in fact takes. Pancomputationalism thus violates certain plausible necessary conditions on successful contrastive explanation.

These failures in contrast illustrate a more fundamental problem raised by pancomputationalism. Whatever else they might involve, scientific explanations answer

"what if things had been different?" questions (see, e.g., (Woodward, 2003)).[29] In the computational case in particular, such questions ask how the properties or behavior of physical systems would be different, if those systems had been different computationally. Yet under pancomputationalism there is *no* clear sense in which things might have been different computationally, for in every possible world even remotely similar to ours every physical system implements everything. Under pancomputationalism, it thus appears that the computations implemented by physical systems are utterly irrelevant to the explanation of their properties and behavior, for those computations do not appear to make a difference to anything at all.

3.3.3 Panpsychism

Worries about pancomputationalism ramify under certain computationalist views in the philosophy of mind. Although many such views are possible, David Chalmers' 'computational sufficiency thesis' is representative of the genre. This view holds that implementing certain computations is sufficient for possessing a mind and certain mental properties (Chalmers, 2011). Clearly, given pancomputationalism, the computational sufficiency thesis entails panpsychism. Thus to the extent that panpsychism is itself objectionable, some philosophers find pancomputationalism objectionable as well (Schweizer, 2019a; Sprevak, 2019).

But I mention this concern only to set it aside. Triviality arguments cannot be resolved merely by rejecting the computational sufficiency thesis.[30] Indeed, one can

[29]In saying this, I do not mean to endorse any particular view of scientific explanation. The failures discussed here plausibly arise on all the major theories of scientific explanation, albeit in subtly different ways.

[30]This is the tactic pursued in (Schweizer, 2019b).

think that computation plays an important theoretical role in science, including cognitive science, even if one rejects the strong link between computation and cognition expressed by the computational sufficiency thesis. As long as computation is expected to play *some* descriptive or explanatory role in cognitive science, pancomputationalism threatens its ability to play this role.

3.3.4 Taking stock

While I do not claim to have identified all of the problems that might be raised by pancomputationalism, the failures outlined above are arguably the most serious. In light of this, I will take the principle task in responding to triviality arguments to be to meet the following two adequacy criteria:

1. The response should ameliorate all or at any rate most of the foregoing descriptive defects.

2. The response should vindicate computational explanation in both intersystemic and intrasystemic scenarios.

The task of the next two sections is to develop a response that fits this bill.

3.4 Implementation and the applicability of mathematics

My response has two parts. I develop the first part in this section and the second part in the next. Here I point out that when a system implements a computation, it does so relative to a specific assignment of mathematical states or values to physical components. I will call such an assignment a 'labelling scheme'. I argue that, relative to a specific labelling scheme, physical systems typically implement just a single computation.

3.4.1 The relativity of applications

My starting point is the observation that ascribing computations to physical systems is an instance of the more general practice of applying mathematics to physical systems. Important for my purposes is a particular, widely-noted aspect of this more general practice: that physical systems do not wear their mathematical characteristics on their sleeves. Regarded one way, a physical system has a certain mathematical character or structure; regarded a different way, it has another. Perhaps the most well-known example of this involves the application of finite cardinal numbers. Here is Frege in the *Grundlagen*:

> If I give someone a stone with the words: Find the weight of this, I have given him precisely the object he is to investigate. But if I place a pile of playing cards in his hands with the words: Find the Number of these, this does not tell him whether I wish to know the number of cards, or of complete packs of cards, or even say of points in the game of skat ... Number, cannot be said to belong to the pile of playing cards in its own right, but at most to belong to it in view of the way in which we have chosen to regard it ... What we choose to call a complete pack is obviously an arbitrary decision, in which the pile of playing cards has no say. (Frege, 1884, section 21)

Frege observes that different 1ways of regarding' one and the same hunk of physical stuff yield different counts. That is, finite cardinals apply to physical stuff only under a (sortal) concept, which individuate undifferentiated physical matter into objects. Relative to the concept CARD this stuff is fifty-two; relative to DECK, it is one. Physical matter does not wear its cardinality on its sleeve.

Poincaré makes a similar observation with respect to physical geometry:

> All the geometries I considered had thus a common basis, that tridimensional continuum which was the same for all and which differentiated itself only by the figures one drew in it or when one aspired to measure it ...

> In this continuum, primitively amorphous, we may imagine a network of lines and surfaces, we may then convene to regard the meshes of this net as equal to one another, and it is only after this convention that this continuum, become measurable, becomes Euclidean or non-Euclidean space. From this amorphous continuum can therefore arise indifferently one or the other of the two spaces, just as on a blank sheet of paper may be traced indifferently a straight or a circle. (Poincaré, 2015, 235)

Crudely, Poincaré is pointing out that application of geometric notions to physical space depends on a choice of metric. Relative to one metric, space has Euclidean structure; relative to others, non-Euclidean structure. Different ways of regarding the primitive amorphous continuum of space yield different physical geometries.

Frege and Poincaré use their observations to motivate specific views in the philosophies of mathematics and science: Frege argues that number belongs to (Fregean) concepts, not physical objects, while Poincare advocates for a conventionalist view of the geometry of physical space. I take no stand on these further issues here. These conclusions follow only under certain additional assumptions over and above the observation that physical systems do not wear their mathematical characteristics on their sleeves.

The point I wish to emphasize is simpler, and more fundamental. As Frege puts it, mathematical notions apply to physical systems relative to a way of regarding the system in question. While a certain interpretation of this expression would suggest that the mathematical features of physical systems depend on *agents* who regard those systems in certain ways, I wish to distance myself from any such interpretation here. Ways of regarding a physical system mathematically (concepts in the case of arithmetic, metrics in the case of geometry) can be understood as functions assigning mathematical entities or values to physical components (entities, states, properties,

magnitudes, etc.).[31] On this approach, whether a physical system has some mathematical feature thus depends on the existence of a mapping from certain components of that system to some mathematical structure. Thus the first step in my response is the observation that mathematics applies to physical systems *relative to specific assignments of mathematical entities to physical components*.

3.4.2 Labelling schemes

Insofar as the application of computational notions is an instance of the more general practice of applying mathematics, we should expect analogous remarks to hold in the computational case too. Perhaps the simplest illustration of the claim that physical systems do not wear their *computational* characteristics on their sleeves concerns a single digital circuit. The circuit transforms input voltages into output voltages according to a certain pattern. Perhaps it outputs 2V just in case both input voltages are 2V, otherwise it outputs 0V. If we interpret the 2V state as logical 1 and the 0V state as logical 0, the gate computes logical AND; under the reverse assignment, it computes logical OR.[32] Viewed one way, the circuit is an AND gate, while viewed another, it is an OR gate. (Under more recondite interpretations, it presumably computes other logical functions, too.)

In light of this, it would appear that the computational features of physical systems are determined relative to specific ways of regarding systems computationally. Following Copeland (1996) I will refer to these ways as *labelling schemes*. In the case

[31]This way of understanding the applicability of mathematics traces back at least to Quine (1960); see Pincock (2011, ch. 2) for a recent statement. Even philosophers otherwise hostile to mapping accounts of the applicability of mathematics, such as Batterman (2010) and Bueno and Colyvan (2011), recognize that mappings play an important role in applications.

[32]See (Shagrir, 2001; Piccinini, 2008; Sprevak, 2010) for discussion of such cases.

just considered, there are two schemes. The AND-scheme assigns logical 1 to the 2V state and logical 0 to the 0V state, while the OR-scheme assigns 0 to 2V and 1 to 0V. Of course, there may be others as well. As in the general mathematical case, I will take labelling schemes to be functions from physical components (entities, states, properties, events etc.) to mathematical states or values. Thus when a physical system implements a computation, it does so *relative to* a specific labelling scheme.[33]

In light of this relativity, we should thus construe implementation, not as a two place relation between a physical system and a computation, but rather as a three-place relation between a physical system, computation, and labelling scheme:

> **The Simple Mapping Account (three-place)**
>
> A physical system P implements a computation M *relative to a labelling scheme* f if and only if:
>
> 1. f maps state-types of P to states of M, such that
> 2. Under f, the state transitions of P are isomorphic to the formal state transitions of M; i.e., whenever P is in state p_1, where $f(p_1) = m_1$, and $m_1 \rightarrow m_2$ is a formal state transition, then P goes into state p_2, where $f(p_2) = m_2$.

In principle there are as many labelling schemes as there are functions from physical components to states of computations. This in turn depends on the abundance of physical components, where these are understood broadly to include objects and their parts, properties, relations, and states. Although many views on this mark are possible — from very liberal views which countenance a wide array of physical components, to quite sparse parts which do not — I take it that whatever one's views of physical reality, it will turn out that vastly many labelling schemes apply to even a single physical system.

[33]See Blackmon (2013) for a related thought.

Insofar as labelling schemes are functions from physical systems to mathematical structures, a scheme is well-defined only if, under that scheme, a system implements a single computation. While I do not deny that it is in principle possible to define 'schemes' that fail to discriminate between two or more computations, we can safely ignore such possibilities here. This is because the Putnam-Searle argument does not rely on 'schemes' that underdetermine the computation implemented by a system. To see this, consider Putnam's (1987, Appendix) example of a rock that implements every deterministic finite automaton. Relative to one scheme, the rock implements a three-state automaton: ABA. Relative to another scheme, it implements a five-state automaton: $ABABA$. And so on for the rest of the automata implemented by the rock: each is implemented relative to a different labelling scheme.

Putnam's point is not that a single labelling scheme underdetermines the computation implemented by a system. His point, rather, is that many different schemes apply simultaneously, *and* that there are no grounds for preferring any one of them to any other when describing a system computationally. I will return to Putnam's pessimism about singling about a scheme below. For now, I wish to emphasize that, even if for every finite automaton there is some scheme relative to which the rock implements it, there is no single scheme relative to which it implements them all. Relative to a specific labelling scheme, the rock implements just a single finite automaton. That is, relative to a specific labelling scheme, pancomputationalism fails.

Thus if there were grounds for ascribing a computation to a physical system in terms of a *specific* labelling scheme, we would arguably be in a better position to address triviality worries. For, relative to that scheme, the system would implement just a single computation, in which case it would not be unreasonable to expect

that we could frame substantive, non-trivial hypotheses about the descriptive and explanatory adequacy of that computation. We would thus be in a position to meet the two adequacy criteria introduced above. Of course, this observation helps only if such grounds are forthcoming. Happily, as I shall suggest in the next section, such grounds are near at hand.

3.5 Computational ascription is context-sensitive

When computational scientists ask whether a system implements a given computation, they do so in a particular investigative context. That context determines, more or less explicitly, a labelling scheme relative to which they address questions about the computational features of a physical system. They wish to know, given *this* way of regarding a system computationally, what it computes. Relative to *other* schemes, the system will undoubtedly implement other computations. But to the extent that such implementations rely on labelling schemes irrelevant in that specific investigative context, they are irrelevant to questions about the computational characteristics of the system in that context.

In this respect, applying computational notions once again mirrors the more general practice of applying mathematics. By way of illustration, consider the numerical case. We cite finite cardinals to address numerosity questions of the form "how many Fs are there?". Such questions are raised in particular investigative contexts, which fix the relevant sortal F. It is of course true that, had we employed a different concept in that context, we may have arrived at a different count. But as long as a given count is produced by the contextually salient concept, it is hard to see how the existence of different counts, generated by different concepts, trivializes anything. The reason

is straightforward: in that context, those alternative counts answer questions that weren't being asked.

In light of these remarks I can sketch, in rough outline, a response to the Putnam-Searle argument. First, computational ascription does not occur in a vacuum. Rather, it occurs in a specific investigative context. This context, I suggest, fixes a specific, contextually salient labelling scheme — or at least a narrowly circumscribed set of such schemes. But, as I argued in the previous section, relative to a specific labelling scheme a system typically implements just a single computation. Hence, relative to a contextually salient labelling scheme, we are arguably in a position to offer substantive, non-trivial computational descriptions and explanations of physical systems. Put computational ascription in its proper context, and triviality disappears.

However, this sketch leaves two issues unresolved. First: what makes a labelling scheme salient for computational description and explanation in any given case? Second: given a specification of these features, to what extent are the descriptive and explanatory adequacy criteria encountered in section 3 satisfied? I address these questions in the remainder of this section.

3.5.1 What makes a labelling scheme salient?

The bad news is that I somewhat doubt that there is an illuminating general story to be told here, because I somewhat doubt that there is an illuminating general story to be told about the sorts of theoretical goals and interests that drive inquiry in the computational sciences. Of course, computational inquiry is guided by theoretical ideals found throughout science (simplicity, predictive accuracy, explanatoriness, etc.). In light of this, computational scientists will presumably place a premium on

labelling schemes that lead to theories that exemplify familiar scientific ideals. But to say this is not to say very much. The relevant scheme depends on the task at hand, and the tasks of computational science are many and diffuse. I am pessimistic that there are many substantive features common to successful labelling schemes in every computational context.

The good news is that matters are more promising once we attend to more specific theoretical projects. Here it is instructive to treat artificial and natural computing systems separately. For an artificial system, such as a microprocessor, the salient scheme will presumably be determined by the manufacturer or electrical engineers who designed it. In contemporary devices voltages in certain circumscribed ranges are assigned certain logical values. It is standard to assign logical 0 to the low or 'ground' range, and logical 1 to the high range.[34] This convention is not inevitable, of course. Early computers such as the Atanasoff-Berry computer employed the reverse assignment (Burks and Burks, 1988). Here, as elsewhere in science, it is reasonable to defer to expert practitioners, so that the computational features of an artificial ought, all else equal, to be identified by whatever labelling scheme is employed by its designers.[35]

Matters are more complicated for natural computing systems, for here it is often an open empirical question just which scheme is appropriate for theorizing about some system. In practice, there may be a range of incompatible schemes consistent with the available data. Nevertheless, even if it is an open empirical question which scheme is most appropriate for some system, it is not the case that *any* scheme is as

[34]This oversimplifies somewhat. For details see (Harris and Harris, 2013, 22-6).

[35]I do not deny that all else might *not* be equal, in which case there may be grounds for departing from the designer's scheme. This might be warranted if a device systematically behaves in unexpected ways, or if we attempt to repurpose it to perform some new computational task.

good as any other. Acceptable schemes must account for available behavioral data as well as known structural and functional features of the system under consideration. For instance, even if we cannot yet determine which of a variety of schemes is most appropriate for computations in the visual system, any such scheme will presumably be 'neurobiologically plausible', in the sense that it must cohere with knowledge about the structure and function of various components of the visual system. Gruesome schemes that join arbitrary neuron parts, for instance, aren't even on the radar.

The importance of behavioral data deserves special mention. Computations are cited (often if not exclusively) in order to explain some behavioral phenomenon. Absent such a phenomenon, it is unclear what grounds there would be for ascribing *any* computations to a system. This is not to say that we cannot do so. But it is to say that such ascriptions must be motivated on other grounds. Putnam's rock exhibits *no* behavior which might reasonably be explained in computational terms. While Putnam is surely correct that we *can* ascribe all manner of computations to the rock, it is hard to see what theoretical interest there would be in doing so. Moreover, even if we can devise some reason for ascribing computations to the rock, that does not obviously threaten well-motivated computational ascriptions elsewhere.[36]

In general, which physical features are relevant in a given case will depend on both the phenomena of interest, as well as our own explanatory goals in describing systems computationally. Thus whether a scheme is relevant will typically be an empirical question. Elaboration on this point would benefit from detailed case studies which go beyond typical pronouncements about causal-mechanical, representational, etc. features of computational systems. What particular schemes are employed in

[36]Egan (2012) makes a similar point.

computer and cognitive science? What sorts of features do practitioners focus on when applying computational notions? What is it about these features that makes for successful science? I do not have the space to pursue these questions here, and for now they must remain on the agenda for future work.[37]

3.5.2 Meeting the adequacy criteria

The adequacy criteria introduced in section 3 concern both the descriptive and explanatory power of computational implementation. Although the question whether a *particular* labelling scheme meets these criteria is ultimately an empirical question, I will suggest that *in general*, relative to a specific labelling scheme, we are well positioned to offer both descriptively and explanatorily satisfactory computational descriptions of physical systems.

Descriptive criteria

- *Lack of empirical content.* Relative to a fixed labelling scheme, it is an empirical question whether a given system implements a computation, relative to that scheme. Consider the AND-scheme considered earlier. This scheme mapped voltage levels to logical values. We can thus determine empirically whether a given physical system implements an AND-gate, by determining whether it has the right physical properties and transforms them in the right way.

[37]I return to this thought briefly in section 6. One might worry that this approach will simply enumerate different applications without illuminating computational practice.We will be left simply with a motly, unruly collection of ad hoc applications. I think this is a reasonable concern, but it would be premature to press the worry at this stage. Ultimately, whether such principles exist is an empirical question, to be determined by detailed investigation of computational practice.

- *No empirical discoveries.* Similarly, relative to a fixed scheme we may well discover that a physical system implements one computation rather than another, or indeed no computation at all. While this criterion is less important for artificial computing systems, it is more important for natural systems. For instance, relative to the AND-scheme, by examining how a configuration of neurons transforms voltage levels we may discover that they compute logical AND.

- *Extensional inadequacy.* Questions about extensional adequacy should be assessed relative to the sorts of schemes that feature prominently in the computational sciences. For instance, if contemporary hardware engineers typically label states according to specific electromagnetic properties, it would turn out that very few systems compute in the sense relevant for inquiry into designed computing systems. Once we focus attention on the schemes that feature in computational practice, there is little reason to expect widespread extensional inadequacy.

- *Widespread predictive failure.* Relative to a specific scheme, a robot will implement a specific pathfinding algorithm, hence will choose a determinate path through the obstacle course. We can thus make substantive predictions about the robots behavior, at least some of which may succeed.

- *Unwarranted retrodictive success.* Similarly, given the path actually taken, once we fix upon a particular scheme there is a genuine question whether the

pathfinding computation implemented relative to that scheme could have produced that path. Under the salient scheme, we can make substantive retrodictions about the robot's past pathfinding behavior, at least some of which may fail.

Explanatory Criteria

Recall that the explanatory failures revolved around the contrastive character of computational explanation. We must account for two explanatory scenarios in particular: in the intersystemic scenario, we explain differences between systems or system kinds in terms of a computation implemented by one but not the other, while in the intrasystemic scenario, we explain differences in the behavior or properties of a single system or system kind in terms of a specific computation it implements.

The overall point is that there can be genuine computational contrasts relative to a fixed labelling scheme. Start with the intrasystemic case. Relative to a specific labelling scheme, there is a fact of the matter about what computation a system implements. We can thus advert to that specific computation to explain the properties and behavior of that system. For instance, we can explain that the robot takes *this* path because it implements *that* pathfinding algorithm, relative to a scheme appropriate for explaining how the robot moves about its environment.

With respect to the intersystemic case, the point to notice is that schemes appropriate for explaining one kind of system will typically not apply to other kinds of system. A scheme appropriate for explaining computations in healthy visual systems will presumably rely on the specific features of visual systems: the specific voltages employed by neurons in the visual system, spike rates, their organization, and so on.

Not all of these features will be displayed in injured visual systems, which presumably differ in certain of these respects from healthy systems. Moreover, relative to a scheme appropriate for healthy visual systems, it is highly unlikely that manifestly non-cognitive systems such as rocks, walls, or pails of water implement anything, much less a complex depth-from-disparity computation. The reason is straightforward: these systems lack the kinds of features relevant for computational ascription in the visual system.

Of course, rocks presumably implement these computations somehow, relative to some (perhaps highly gerrymandered) labelling schemes. A degree of pancomputationalism is, in this sense, unavoidable. But, from the point of view of cognitive neuroscience, these schemes are utterly inappropriate for explaining physical systems. Cognitive neuroscience countenances schemes relevant for explaining the kinds of systems it investigates, namely brains. Claims about computations in, say, the visual system, must be assessed relative to such schemes. Insofar as rock-computations rely on schemes utterly appropriate for describing brains, there is little reason to think that pancomputationalism threatens the use of computational notions in cognitive neuroscience.

This point is general. For nearly every computation there is *some* (perhaps gruesome) scheme relative to which a physical system will implement it. I am happy to concede that a *context-insensitive* account of computation is trivial. But, in most scientific contexts, these computations will rely on schemes that are explanatorily irrelevant. Relative to a specific, contextually determined labelling scheme, applications of computational notions are as non-trivial as one could reasonably want.

3.6 Elaborations

This completes the main task of the chapter. In the space remaining, I will clarify my position and draw out its relationship to others in the literature.

3.6.1 No special problem for computation

Some readers may be unsatisfied with my response, on the grounds that it is little more than a promissory note. Have I really given us reason to think that computation will be non-trivial, at least in the most important scientific contexts? More worryingly, insofar as I have declined to offer a theory of what makes a given labelling scheme relevant in a given context, how can we be sure that my response doesn't simply *relocate* triviality problems at the level of a contextually determined labelling scheme?

I have a few responses to this worry. On the one hand, it is true that I have not attempted to show that computation is *not* trivial in every theoretical context. This is because, on my view, whether computation is trivial in a specific theoretical context is ultimately an *empirical* question. For this reason I allow that in some contexts computation might turn out to be trivial. But rather than undermine our confidence in computation wholesale, this would constitute a philosophically important discovery, for it would further illuminate which kinds of physical phenomena are fruitfully understood in computational terms and which ones are not.

Nevertheless, there is little reason to think that computation will be non-trivial in most scientific contexts. When applying mathematics to physical systems, there are two kinds of case to consider (Steiner, 1998). In one kind of case there is no non-mathematical description of the target system available. In the other kind, there

is. The former sort of case raises a variety of deep, difficult questions about how mathematical descriptions get a toehold on the world. This sort of case arises frequently in cases of measurement, especially of fundamental magnitudes (Chang, 2007; Van Fraassen, 2008). In the latter sort of case, we can appeal to these pre-existing descriptions when applying new mathematical structures to the target system.

The application of computational notions often resembles the second sort of case more closely than the first.[38] Artificial computing systems are designed in light of detailed scientific knowledge of the materials out of which they are built. And enough is known about basic brain structure and function that computational ascription is fairly tightly constrained in the cognitive case as well. Thus with respect to the applications of computational notions most directly targeted by triviality worries, there will typically be a large stock of prior non-computational descriptions (some mathematical but non-computational, some non-mathematical) on which we can rely when applying computational notions in particular.

What about the worry that my response simply relocates the problem? In reply, I would point out that my response doesn't relocate triviality worries; rather, it *reduces* them to a more familiar general issue. Recall that the original worry was supposed to pose a *special* problem for computational implementation. Computational descriptions were supposed to be unlike other kinds of descriptions (physical, chemical, biological, etc.) description in that the former but not the latter applied to physical systems indiscriminately. However, the question now before us is just the question of

[38]Often but not always. Views on which physical reality is fundamentally computational arguably fall under the first case. For discussion of such views see (Piccinini, 2015, ch. 4) and (Piccinini and Anderson, 2018).

determining when a given scientific description is descriptively or explanatorily adequate, where the instance in question concerns specifically *computational* descriptions of physical systems. To be sure, a problem is no less problematic for being general (or familiar). But to the extent that progress has been made on the more general issue, there is little reason for pessimism about the computational case in particular.

3.6.2 Observer-relativity

While I have focused on the descriptive and explanatory challenges raised by pancomputationalism, one might think that the deeper threat raised by Putnam and Searle is not that computation is trivial, but that it is observer-relative — 'a matter of free interpretation', to use Piccinini's phrase. For if every computational description applies to every physical system, it would seem that there can only be pragmatic, instrumental reasons for working with one description rather than another. Insofar as my account gives pride of place to specific investigative contexts, doesn't my account capitulate this deeper point?

This worry derives some of its force from the suspicion that observer-relative notions are unsuitable for serious scientific or philosophical work (see, e.g., (Searle, 1992, 211-2)). Maybe that's right, maybe not. Ultimately, I think it's a red herring, because my response does not commit me to an observer-relative conception of computation. By the same token, however, neither does it commit me to an observer-independent conception. Indeed, as I will suggest next, the issue of observer-relativity is ultimately orthogonal to the triviality issue. This point is not always appreciated, so I will take a moment to work through it.

Here is a typical statement of the view that computation is observer-relative:[39]

> Computation is not an 'intrinsic' property of physical systems, in the sense that (a) it is founded on an observer-dependent act of ascription, upon an entirely conventional correlation between physical structure and abstract formalism ... there is no deep or metaphysically grounded fact regarding whether or not a physical system 'really' implements a given computational formalism. (Schweizer, 2019b, 31)

If *this* is what is meant, then my account is not committed to the claim that computation is observer-relative. On my view, physical systems implement computations relative to labelling schemes. But whether a given labelling scheme applies to a physical system does not necessarily depend on whether anyone ascribes that scheme to that system. Rather, it depends only on the features of the system under consideration. Labelling schemes are simply functions from physical components (states, parts, properties, etc.) to mathematical entities. Such functions quite plausibly exist whether or not anyone ascribes them to anything. Neither they nor the computational descriptions they sanction depend for their existence or satisfaction on observer-dependent acts of ascription.[40]

Moreover, relative to a specific labelling scheme it would appear that there *can* be 'metaphysically grounded' facts about what computation a system implements. Relative to neurobiologically plausible labelling schemes, the details of which we are only beginning to understand, certain properties and behaviors of the visual system can be described and explained computationally. It is moreover an observer-independent

[39]Schweizer uses 'observer-dependent' rather than 'observer-relative'. See also (Searle, 1992; Matthews and Dresner, 2017; Szangolies, 2020; Fletcher, 2018).

[40]Of course, one *could* think that functions depend for their existence on observer-relative acts of ascription. Such a proposal bears obvious affinities to certain intuitionist programmes in the philosophy of mathematics, for more on which see, e.g., (Iemhoff, 2020). While nothing I've said precludes developing a theory of computational implementation along these lines, it should be clear that my response is not committed to such an approach.

fact about the visual system that it implements certain computations relative to such schemes. Insofar as the computations in question satisfy standard scientific adequacy criteria — are predictively useful, explanatorily fruitful, etc. — there are strong grounds for thinking that the visual system objectively realizes the computational features ascribed to it.

The view that computation is observer-relative is sometimes motivated by the thought that the only reasons we have for preferring one labelling scheme over another derive from facts about our cognitive capacities:

> the crucial requirement [on computational implementation] ... is that of supporting and enabling ... surrogative reasoning and conceptualization ... And this is precisely where the imagined computational implementations proposed by Putnam and Searle fail ... We can neither reason nor conceptualize surrogatively with either. This failure manifests itself in the shortcomings diagnosed in the Putnam/Searle constructions by extant rebuttals to their challenge: These constructions do not support counterfactuals, for example, nor do they rely on groupings of physical states that arise in a natural way from extant physical theory. However, according to the picture presented here *these shortcomings are manifestations of the more basic failure, i.e. lack of surrogativity.* (Matthews and Dresner, 2017, 852, emphasis mine)

Here the suggestion appears to be that whether or not a labelling scheme is relevant in a given theoretical context is ultimately explained, not by the fact that it captures some objective feature of a physical system, but instead by the fact that it supports practical surrogativity. However, nothing I've said commits me to the claim that practical surrogativity (whatever that amounts to) is the only or even the most fundamental criteria for determining whether a labelling scheme is relevant. Computational ascription is guided by a variety of considerations, not all of which are pragmatic. For instance, one plausible adequacy condition is that computational ascriptions be *true*. Yet pragmatically useful theories may be false and true theories

may be practically useless. Practical surrogativity, suitably understood, may be an important theoretical goal, but it is one goal among many.

To be sure, there is an utterly uncontroversial sense in which my account relies on pragmatic, observer-relative considerations. Which scheme is *used* in a given investigative context very obviously depends on the theoretical goals and interests of the researchers in that context. Indeed, how could it not? A scheme that fails to be practically useful, for instance by failing to yield useful predictions about a system, is unlikely to play a role in computational practice. But dependence on practical considerations in *this* sense is found throughout science. On its own, it warrants no strong conclusions about the 'observer-relativity' of computation per se.

It is instructive to consider other applications of mathematics in connection with this point. For instance, relative to the concept DECK, it is an objective fact that there are (say) exactly fifty-two cards on the table. It is also an objective fact that there is one deck on the table. These facts are true simultaneously. But the fact that we must use one concept or another when counting does not entail that number is observer-relative. By the same token, the fact that we must use one labelling scheme or another when ascribing computations does not entail that computation is observer-relative.[41]

3.6.3 An ecumenical approach

So far I have largely avoided discussing specific theories of implementation, because I think that offering such a theory is unnecessary to defuse the Putnam-Searle

[41] Of course, some philosophers *have* concluded on nearby grounds that number is observer-relative; recall Berkeley's argument that number is a 'creature of the mind' (Berkeley, 1999, PHK 12). However, as Frege (1884, sec. 26) points out, Berkeley's inference is too quick. That different mathematical descriptions of a physical system may hold simultaneously does not entail that such descriptions track only mind-dependent features of that system.

challenge. However, most responses assume that offering such a theory *is* required to meet the challenge. Given this, I will take a moment to say something about the relationship between my account and others in the literature. In particular, I will suggest that my response opens a route for unifying extant responses to triviality, in a sense to be explained shortly.

To make matters concrete, consider Chalmers' (1996) well-known account of combinatorial-state automaton (CSA) implementation. On this account, the descriptive content of any computation is expressed in terms of some CSA, and a physical system implements a CSA if it exhibits the 'causal isomorphism-type' described by that CSA. In the language of labelling schemes, the account holds that only certain labelling schemes are adequate for computational theorizing: namely, those which capture the causal isomorphism-type of a physical system (at a certain level of description).

This account imposes a global, context-insensitive condition on computational implementation. Although it is not unreasonable to think that pancomputationalism fails if such a condition is in place, it is questionable whether this tactic yields a fully satisfactory solution to triviality worries. Chalmers provides little reason for us to think that all fruitful applications of the CSA formalism must involve realizing a certain causal isomorphism type. Indeed, insofar as Chalmers' account is motivated on *a priori* philosophical grounds, it would be frankly miraculous if the account captured *every* application of computational notions found in the computational sciences.[42] For this reason, it is doubtful that Chalmers has offered a plausible general solution to the Putnam-Searle challenge.

[42]This point is borne out by more recent work, which suggests that the computational features of some physical systems outstrip their local, causal features. See, e.g., (Shagrir, 2001; Rescorla, 2013).

This is not to say that Chalmers' account should be abandoned outright. The account is primarily designed to vindicate a version of computational functionalism (Chalmers, 2011). Whether or not Chalmers' specific proposal succeeds at this task, the project itself is not unreasonable. Thus I would suggest that we understand Chalmers' suggestion, not as a global condition on implementation per se, but rather as a condition on labelling schemes adequate for the purposes of computational functionalism. Relative to *that* theoretical endeavour, Chalmers' proposal is rather more reasonable.[43]

I would suggest we understand other responses to triviality found in the literature in a similar spirit. These responses appeal variously to counterfactual (Copeland, 1996), causal-mechanical (Chalmers, 1996; Godfrey-Smith, 2009; Piccinini, 2015), dispositional (Klein, 2008), representational (Shagrir, 2001; Sprevak, 2010; Rescorla, 2014b; Peacocke, 1995), teleological (Bontly, 1998; Piccinini, 2015), information-theoretic (Millhouse, 2019), and pragmatic (Matthews and Dresner, 2017) conditions on implementation. Others are surely possible. But rather than posing global, context-insensitive conditions on implementation, these proposals are instead better understood as isolating features of physical systems salient for certain specific theoretical projects.

Thus the current perspective arguably unifies extant responses to triviality arguments by revealing that they need not compete with each other *as accounts of implementation*. Instead, we can regard them as proposals about how computational notions are (or at any rate ought to be) applied to subserve scientific theorizing in different domains. This is not to say that there cannot be disagreement, of course.

[43]Of course, I take no stand here on whether Chalmers' proposal is ultimately successful with respect to this particular project.

But any such disagreement will concern the suitability of a particular kind of labelling scheme for a given theoretical purpose, rather than whether a given kind of scheme captures some fact about implementation as such.

For instance, it is hotly disputed whether a computational account of the mind ought to involve representation — that is, whether when theorizing about the mind we ought to employ labelling schemes in which representational properties play a central role. Without taking a stand on this issue here, I would point out that this not a disagreement about *computation* — it is a disagreement about *cognition*. Computation as such is flexible enough to accommodate both of these views of the mind (and others besides). But which is correct ultimately depends on the nature of the mind, not the nature of computation.

3.7 Conclusion

Throughout this chapter I framed my response in terms of the simple mapping account. This was to show that the simple mapping account, properly understood, has the resources to avoid triviality without the need to develop a complex mapping account, as many philosophers have been tempted to do. Having seen how the response works, however, we can also see that it is not solely the property of the simple mapping account either. The idea that systems implement computations relative to specific ways of regarding systems computationally is, strictly speaking, orthogonal to the question of the formal character of the implementation relation. Indeed, because my response relies only on some straightforward observations about the applicability of mathematics to physical systems, it is compatible with a wide range of views about the relationship between mathematics and physical reality.

In the next chapter I argue that against the simple mapping account, not on the grounds that it is trivial, but on the grounds that in many cases implementation should not be understood in terms of isomorphism. In light of this, I will eventually suggest that we should characterize implementation in terms of the broader notion of resemblance. Nevertheless, the response to triviality developed here is available on a resemblance-based account. I shall return to this point in Chapter 5.

Chapter 4: Limitative Explanations in Computer Science

4.1 Introduction

This chapter investigates a broad class of explanations of central importance to contemporary computer science. These explanations, which I call 'limitative' explanations, explain why certain problems cannot be solved computationally, either in principle or under certain constraints on computational resources such as time or space. Limitative explanations are philosophically rich, but they have not received the attention they deserve. The primary goal of this chapter is isolate limitative explanations as a distinctive kind of computational explanation and providing a preliminary account of what makes them explanatory. In the first half of the chapter I argue that limitative explanations are best understood as a kind of non-causal mathematical explanation which depend crucially on certain highly idealized models of computation.

In the second half of the paper I trace out some upshots of this account for philosophical accounts of computational explanation and computational implementation, respectively. I argue for two negative claims in particular. First, according to the received view of computational explanation, computational explanation is a kind of causal explanation. However, if my account of limitative explanations is correct, then

the received view is wrong: not all computational explanations are causal explanations. Second, as we have seen in previous chapters, many philosophers hold that being isomorphic to a mathematically defined computation is at least a necessary (if not sufficient) condition on implementation. However, I argue that the way limitative explanations apply to physical systems cannot obviously be understood in terms of isomorphism. In particular, there are typically no isomorphisms relating highly idealized computational models to physical computing systems.

Here is the plan. Section 2 surveys a battery of limitative results from across theoretical computer science. Section 3 explains how these results apply to physical systems. Section 4 argues that these results are prima facie explanatory of the computational limits of physical systems. Section 5 develops my positive account of their explanatory power. Sections 6 and 7 trace out the aforementioned upshots of this account for computational explanation and implementation.

4.2 Limitative results in computer science

At the broadest level, computer science addresses two kinds of problems: those which can be solved computationally, and those which cannot. To address the first kind of problem, we must describe a computational procedure, such as an algorithm or program, that solves it. To address the second kind, by contrast, we must show that no such procedure exists. The goal of this section is to survey a representative selection of the latter sort of problem.

As I will use the term, a 'limitative result' is a mathematical theorem that identifies limits or constraints on computational solutions to a certain problem or problem kind. These results vary according to the kind the limit they identify. Non-existence

theorems show that a problem cannot in principle be solved computationally. Other results show that a problem cannot be solved under certain constraints on computational resources such as time or space, or that any solution faces ineliminable tradeoffs between competing adequacy criteria.

In what follows I do not attempt to exhaustively survey or categorize every different kind of limitative result found in contemporary computer science. Nor do I try to sharply demarcate the kinds of limitative results I have discussed. Rather, and more modestly, I survey a representative selection of these results, grouped coarsely for expository purposes according to the kind of limits they identify.

4.2.1 Non-existence Theorems

Perhaps the most well-known limitative results come from computability theory.[44] Limitative results here typically take the form of non-existence theorems, which state that no computational procedure exists for the problem in question. One famous result of this sort is the unsolvability of the halting problem. Informally, this the problem of determining whether a given Turing machine halts on an arbitrary input. Notoriously, however, no Turing machine exists which solves this problem. This captures a limit on the computational powers of Turing machines: even given unlimited time and space, it is a problem which they simply cannot solve.

Non-existence theorems are perhaps the most striking kind of limitative result, and they play a central role in nearly every branch of contemporary computer science, including algorithmic analysis (Sedgewick and Flajolet, 2013; Knuth, 1998), distributed systems theory (Attiya and Ellen, 2014), artificial intelligence (Minsky and Papert,

[44]For introductions to computability theory, see (Rogers, 1987; Cutland, 1980; Davis, 1982; Sipser, 2013; Savage, 2008). (Soare, 1996) is a more advanced treatment.

1988), computer security (Cohen, 1987), computer graphics (Hughes, 2014), and software engineering and compiler design (Appel and Palsberg, 2002). While these results take different forms in different cases, one common technique involves showing how a solution to the new problem would yield a procedure for solving a problem already known to be unsolvable.

To illustrate, consider the following problem from compiler design. A compiler is a program for translating a program written in one language into another. One basic requirement is that a compiler preserve the input/output profile of the program being translated. Very often, however, to improve performance compilers to attempt to optimize the program being translated, for instance by detecting and eliminating needlessly duplicated instructions. We define a *fully optimizing compiler* as one which, given some program produces the smallest possible program with the same input/output profile as the original. The fully optimizing compiler problem is the problem of writing such a compiler.

Unfortunately, however, it can be shown that no fully optimizing compiler exists (Appel and Palsberg, 2002, ch. 17). The reason is that the problem of fully optimizing a program is equivalent to the halting problem. For consider a program which never halts. The smallest possible program with the same input/output profile is:

```
L : goto L
```

Given any input, this program immediately goes into an infinite loop. Thus, to detect whether an arbitrary program is input/output equivalent to this one, a compiler would have to be able to detect infinite loops. However, if a program could detect infinite loops, it could solve the halting problem. Since this is impossible, it is impossible to write a fully optimizing compiler.

This is an informal illustration of a computational reduction.[45] Reductions reveal that one problem is at least as 'hard' as another, in the sense that a solution to one of them could be used to identify a solution to the other. For instance, the problem of writing a fully optimizing compiler is just as hard as the halting problem: if we *could* write a fully optimizing compiler, we could use it to solve the halting problem. Reductions are crucial for extending limitative results to new domains.

Non-existence theorems will be the primary focus of the discussion in what follows. Nevertheless, as intimated above, limitative results take many different forms, and for completeness' sake, I will briefly mention two more families of limitative result.

4.2.2 Resource Constraints

Suppose we restrict attention to problems with can be solved computationally in principle. Even given this restriction, there is a sense in which even some in-principle solvable problems cannot be solved, on the grounds that the computational resources required to solve them may be prohibitively large. Computational complexity theory investigates such problems and classifies them according to their intrinsic difficulty, understood roughly in terms of the amount of time or space required to solve them.[46] Here too limitative results play an important theoretical role, and many problems are known to be unsolvable without using a certain amount of time or space.

[45]Formally, if X and Y are computational problems, a reduction is a computable function $f : X \to Y$ mapping X-instances to Y-instances. Many different reduction relations are known; see, e.g., (Rogers, 1987, ch. 6-9).

[46]For introductions to computational complexity theory, see (Papadimitriou, 1994; Sipser, 2013). For a more advanced treatment, see (Arora and Barak, 2009). (Dean, 2021) is a useful survey.

Central to computational complexity theory is the notion of a 'tractably' solvable problem. Tractability is typically explicated in terms of polynomial time computability. Roughly, a problem is tractably computable just in case it can be computed in time polynomial in the size of its instances, and it is intractable if cannot. That is, a problem is intractable if it can only be solved in time exponential (or more) in the size of its input. An extreme example is the problem of computing the Ackermann function, which grows extraordinarily quickly even for modest inputs.[47] Another is the satisfiability problem for propositional logic. This is the problem of determining whether a given propositional formula has a satisfying truth-value assignment. As is well-known, satisfiability is **NP**-complete (Cook, 1971), thus quite likely intractable.[48] Many practically important problems are known to be **NP**-complete; see for instance the appendix to (Garey and Johnson, 1979).

4.2.3 Tradeoffs

A standard technique for dealing with a problem known or strongly suspected to be intractable is to attempt to find good (albeit suboptimal) solutions through optimization or approximation techniques. Investigation into such techniques has furnished a third family of limitative results, known colloquially as No Free Lunch Theorems.

[47]See, for instance, (Boolos et al., 2007, 84-5).

[48]I say that the satisfiability problem is quite likely intractable because we have not (yet) proved that it is intractable. Whether it is intractable turns on the currently open question whether $\mathbf{P} = \mathbf{NP}$. This is because satisfiability is **NP**-complete. Since it is unknown whether $\mathbf{P} = \mathbf{NP}$, we have at best the conditional result that if $\mathbf{P} \neq \mathbf{NP}$, satisfiability is intractable. Although it is quite plausible that $\mathbf{P} \neq \mathbf{NP}$, , we do not yet have a proof to this effect. See (Arora and Barak, 2009, ch. 2).

These results identify tradeoffs between different approximation techniques.[49] Informally put, these theorems state that there are always tradeoffs between different search or optimization strategies. Improved performance for some range of problems is offset by decreased performance for some other range. There are thus limits to the scope of optimal search or optimization algorithms.

4.3 How limitative results apply to physical computing systems

Limitative results are, in the first instance, mathematical theorems. They are characterized in terms of the members of a class formal, mathematically characterized computational models. The most familiar computational models are Turing machines, although outside of computability and complexity theory it is standard to work with a less cumbersome formalism, typically some generic programming language or pseudocode. Generally speaking, however, the expectation is that a result framed in more informal terms could in principle be expressed in terms of Turing machines, albeit with significantly more labour. Consequently, these results apply first and foremost to the computational powers of more or less mathematically characterized computational models; the halting problem tells us something about what *Turing machines* in particular cannot do.

Nonetheless, these results are widely taken to bear on the computational powers of physical computing systems as well. It is no accident that Turing machines are frequently referred to as computational *models*, for they are taken to be scientific models of ordinary physical computing systems, such as general-purpose stored-program

[49]For classic No Free Lunch theorems see (Wolpert and Macready, 1997; Wolpert, 1996). (Ho and Pepyne, 2002) is a useful introductory survey.

computers (Savage, 2008). As such, they are used to describe (and, as I shall argue, explain) certain aspects of physical computing systems. In particular, to the extent that Turing machines capture the computational capacities of such systems, limits on what can be computed by Turing machines apply to the physical systems that they describe too. In the vocabulary of previous chapters, such limits apply to any system that implements a Turing machine.

Incidentally, although in previous chapters I characterized implementation broadly in terms of isomorphisms, for the moment I wish to remain neutral about whether this is the right way to think about the implementation conditions of Turing machines. Later on I will argue that this is not the right way, but to appreciate this we will first need to examine how limitative results are applied in practice.

For concreteness, let's get a typical application of a limitative result on the table. Since it will likely be familiar, I'll use the halting problem. This is the problem of computing the following function:

$$HALT(M, x) = \begin{cases} 1 & \text{if } M \text{ halts on input } x \\ 0 & \text{otherwise} \end{cases}$$

Where M is (a code for) a Turing machine and x is an arbitrary input string. Now, the halting problem would be computable if some Turing machine computed $HALT$, that is, if there was a machine which, when given M and x as input, either halts and outputs 1 if M halts on x, or halts and outputs 0 otherwise. As is well known, of course, no such Turing machine exists. For this reason we say that $HALT$ is Turing-uncomputable. We know that certain physical systems do not compute $HALT$. But we should distinguish two ways this can happen. One is that a system may fail to compute at all. Deviant Putnamesque interpretations notwithstanding, this is presumably the reason why rocks, walls, tables, and chairs do not compute $HALT$.

But notice that we do not need to appeal to a limitative result to conclude that rocks don't solve the halting problem, because no one realistically expects them to compute anything.

The other way a system can fail to compute $HALT$ is by implementing a computational model that fails to compute $HALT$. This is the sense in which laptops fail to compute $HALT$. Unlike rocks (etc.), digital computers compute. They stood, as it were, a fighting chance at computing $HALT$. Unfortunately, insofar as they implement a computational model such as a Turing machine that cannot compute $HALT$, neither do they.

Of course, that a problem cannot be solved by some computational model does not show that some problem cannot be solved *full stop*. Rather, it shows that it cannot be solved in a certain way, i.e., with the sorts of computational operations and processes furnished by that model. This does not rule out the possibility of solving that problem non-computationally, or even of solving it with a different computational model. For example, certain hypercomputational models, such as accelerating Turing machines, can compute $HALT$. Nonetheless, since laptops (etc.) presumably do not implement accelerating Turing machines, this does not undermine the claim that they do not compute $HALT$ either.

Accordingly, when thinking about how limitative results apply to physical systems, we must bear in mind that what problems a system can or cannot solve are indexed to the specific computational model(s) it implements. In particular, we should allow for the possibility that, qua implementation of model M a system fails to solve some problem, but qua implementation of a different model M^*, it does not. Bearing this distinction in mind will help us to avoid some potential worries later on.

Putting the pieces together, I would suggest that applications of limitative results such as the unsolvability of the halting problem are mediated by the following principle:

> **Uncomputability**
> If a system implements a computational model M, then, qua M-implementation, it cannot compute any M-uncomputable function.

Here, then, is how the unsolvability of $HALT$ applies to physical computing systems. Suppose P is some physical system, such as a laptop, that implements a Turing machine. By the foregoing principle, P does not compute any Turing-uncomputable function. But, of course, $HALT$ is one such function. So P does not compute $HALT$ either.

Incidentally, note that this story about how limitative results applies does not appeal to the Physical Church-Turing thesis. Although there are stronger and weaker versions of the Physical Church-Turing thesis, they all purport to identify a connection between Turing machines and physical computation in general.[50] The principle appealed to here, by contrast, restricts attention to physical implementations of some computational model. It does not purport to tell us whether a problem is physically computable full stop, nor does it attempt to identify some connection between physical computation and Turing computability. Rather, it merely identifies a sufficient condition on a physical system (or class of systems) failing to solve some problem.

4.4 Limitative results are explanatory

Going forward I will take it for granted that limitative results play an important theoretical role in contemporary computer science. But what is that role? In light of

[50]For discussion of physical CT see (Piccinini, 2015, ch 15, 16).

the discussion so far I think it is uncontroversial that limitative results *describe* limits on the computational powers of certain physical computing systems. But this is not all they do. In addition to describing limits on computational powers of physical systems, limitative results explain those limits as well, or so I shall argue next.

To avoid begging questions about the nature of scientific explanation, my argument does not rely on a specific theory of scientific explanation. Instead, I argue that limitative results display certain characteristics prima facie indicative of explanation elsewhere in science: they are regarded as explanatory by the relevant scientific community; they answer 'why?' questions; and they are both deep and broad, in a sense that I will explain below. These considerations are obviously not apodictic, but they do make a strong *prima facie* case for the conclusion that limitative results are explanatory. I'll take them one by one.

4.4.1 Computer Scientific Practice

To begin, limitative results are a primary output of computer scientific theorizing and they are widely regarded by computer scientists as explanatory. For one thing, as will likely be familiar to anyone with a modest amount of computer scientific training, these results are often cited in informal thought and talk with broadly explanatory locutions: "the reason why no laptops solves the halting problem is that it's a Turing machine and no Turing machine solves the halting problem;" "you failed to write a fully optimizing compiler because it's impossible;" and so on.

Moreover, the explanatoriness of limitative results appears to be a primary motivation for much work in theoretical computer science. For instance, a popular

introductory monograph on computational complexity motivates its subject matter with the following parable, which I quote at length:

> Suppose that you, like the authors, are employed in the halls of industry. One day your boss calls you into his office and confides that the company is about to enter the highly competitive "bandersnatch" market. For this reason, a good method is needed for determining whether or not any given set of specifications for a new bandersnatch component can be met and, if so, for constructing a design that meets them. Since you are the company's chief algorithm designer, your charge is to find an efficient algorithm for doing this.
>
> After consulting with your bandersnatch department to determine exactly what the problem is, you eagerly hurry back to your office, pull down your reference books, and plunge into the task with great enthusiasm. Some weeks later, your office filled with mountains of crumpled-up scratch paper, your enthusiasm has lessened considerably. So far you have not been able to come up with any algorithm substantially better than searching through all possible designs. This would not particularly endear you to your boss, since it would involve years of computation time for just one set of specifications, and the bandersnatch department is already 13 components behind schedule. You certainly don't want to return to his office and report: "I can't find an efficient algorithm, I guess I'm just too dumb."
>
> To avoid serious damage to your position within the company, it would be much better if you could prove that the bandersnatch problem is *inherently* intractable, that no algorithm could possibly solve it quickly. You could then stride confidently into the boss's office and proclaim: "I can't find an efficient algorithm because no such algorithm is possible!" (Garey and Johnson, 1979, 1-2)

This 'because' is telling. Suppose we have a limitative results which states that no efficient algorithm for the bandersnatch problem exists. Then it is no mere accident that despite concerted effort you fail to find one. Moreover, this failure cannot merely be chalked up to the fact that you're not clever enough. Rather, it would appear that your failure is in an important sense preordained, and this is explained by the fact that no efficient algorithm for the bandersnatch problem exists.

Although fictitious, this example illustrates one important way limitative results explain. The explanandum is a pattern of failure exhibited by a single system, and the explanans is an appropriate limitative result. Not all limitative explanations take this form however. Sometimes they have a stronger explanandum: that no class of physical computing systems (stored-program computers, e.g.) solves a certain problem, or, even more strongly, that it's impossible for a problem to be solved by members of a certain class of computing system. Sometimes the explanans is merely a limitative result: "why does no implementation of a Turing machine compute $HALT$? Because no Turing machine does." Other times, the explanans is a limitative result plus a contingent claim about the system under consideration: "Why does this laptop keep failing to compute $HALT$? Because it implements a Turing machine, and no Turing machine does."

The fact that limitative results are regarded as explanatory by computer scientists, at least some of the time, is perhaps the most compelling reason to think that they are explanatory. I am happy to take this practice at face value, but let me mention a few further considerations for skeptics in the audience.

4.4.2 Answering 'why?' questions

Whatever else they might do, explanations answer 'why?' questions. As the foregoing discussion already suggests, in some circumstances it seems that we can satisfactorily answer 'why?' questions by citing an appropriate limitative result.

Of course, what counts as a good answer to a 'why?' question is highly context-sensitive. For this reason I do not claim that such questions are in every conversational context adequately answered this way. Sometimes the reason for repeated failure is

more mundane. Suppose one attempts to write a compiler that detects infinite loops. In some contexts, the question 'why does the program keep failing to detect infinite loops?' can be adequately answered by appealing to facts about the compiler or to its programmer's skill (or lack thereof). For instance, perhaps the compiler missed an edge case that it would have detected, were it better written.

Nevertheless, even in such cases there remains a deeper reason for one's failure, namely in that one is trying to solve a mathematically unsolvable problem. The theorem that no fully optimizing compiler exists constrains the possible programs one could have written. Given this constraint, one's failure should come as no surprise. In this respect, limitative results resemble other explanatory uses of mathematical results, in which a mathematical theorem seems to constrain a whole range of possible physical systems — I return to this point below.

4.4.3 Breadth and depth

Limitative results apply broadly, in the sense that they apply to a wide range of different computing systems, and are deep, in that they seem to capture a fundamental computational limit that transcends the peculiarities of individual physical systems. The ideas of breadth and depth figure commonly in discussions of explanation, albeit in different ways in different cases. If these are genuine marks of explanatoriness, then it suggests that limitative results are explanatory too.

Something like a notion of breadth appears in the unificationist tradition (Kitcher, 1981; Friedman, 1974), which emphasizes that explanation is at least partly a matter of capturing what's common to otherwise different phenomena. Although traditional unificationist views characterize unification as a property of theories, and characterize

unification in terms of recycled argument patterns, the fundamental insight is that in explanation is a matter of pulling together what, on the surface, might appear to be different phenomena. Unified theories are explanatory, on this mark, because they show us what is common to a wide range of different physical systems.

Physical computing systems comprise a highly heterogenous class. They include humans working effectively, early mechanical devices such as the pascaline or Babbage's analytical engine, contemporary silicon-based computers with widely ranging architectures (e.g. Stanford architectures, von Neumann architectures), and a motley assortment of unconventional systems based on, to name just two, slime mold (Adamatzky, 2015) and RNA (Akhlaghpour, 2022). Despite their differences, insofar as they can all be seen as implementation of Turing machines, limitative results apply to all of them. They thus reveal a common computational limit shared by many different kinds of physical computing systems.

Similarly, the idea that explanations reveal the 'deeper' sources of a phenomenon is attractive, if somewhat inchoate. Although the notion of depth could be clarified in various ways, one natural thought is that depth involves abstracting away from the peculiarities of system to identify only what is essential to the phenomenon of interest. Something like this seems operative with limitative explanations. Absent an appropriate limitative result, a persistent pattern of failure could potentially be chalked up to any of a variety of disparate causes: programmer inability, software bugs, or even hardware defects. However, if we know that a problem *cannot* be solved, it will come as no surprise that our attempts to do so routinely fail. This story transcends the peculiarities of a given system to reveal a fundamental limit on its computational powers.

4.5 How limitative explanations explain

Going forward I assume that limitative results are explanatory. But where does their explanatory power come from? Because limitative explanations rely on mathematical theorems, a natural starting point is the idea that they are a kind of mathematical scientific explanation. Thus I will start by attempting to characterize them in terms of a popular, well-known account of mathematical scientific explanation due to Marc Lange. As we will see, Lange's account captures some but not all of what seems to make limitative results explanatory.

4.5.1 Limitative explanations as mathematical explanations

According to a familiar line of thought, scientific explanations explain by rendering a phenomenon nomically expectable (Hempel, 1965). Paradigmatic limitative explanations proceed in much the same way. In light of an appropriate limitative result it is unsurprising that a physical computing system or system kind fails to solve certain problems. However, whereas more traditional DN-explanations rely on natural laws or law-like generalizations, limitative explanations rely on mathematical limitative results. Consequently, limitative explanations do not merely reveal that their explanans are nomically expectable. Rather, they seem to show that their explananda are mathematically inevitable, as it were. This is because limitative results hold with whatever degree of necessity is held by pure mathematical facts more generally. If, as is plausible, mathematical facts hold with a very high degree of necessity — perhaps the highest — then at least part of the explanatory power of limitative results derives from the fact that they provide a mathematically necessary sufficient condition for their explanans.

In this respect, limitative explanations resemble more familiar instances of mathematical explanation. Perhaps the most well-developed account of mathematical explanation is due to Marc Lange (2013, 2017, 2018), and it is instructive to think about how we might account for limitative explanations using Lange's theory. Mathematical 'explanations by constraint,' as Lange sometimes calls them, work by "describing how the explanandum involves stronger-than-physical necessity by virtue of certain facts ("constraints") that possess some variety of necessity stronger than ordinary causal laws" (Lange, 2018, 17). Although the explananda of such explanations can themselves come in varying modal strengths, they are in general modally stronger than the explananda of typical causal explanations. Accordingly, their explanans must involve facts which are at least as modally strong as they.[51]

To illustrate with a well-known case, consider the bridges of Königsberg. Crudely, the reason why no one has successfully crossed all of the bridges without crossing at least one of them more than once is that (a) the bridges realize a certain graph-theoretic structure (specifically, they form a non-Eulerian graph), and (b) that as a matter of mathematical necessity any complete circuit of a non-Eulerian circuit has at least one double-crossing (cf. (Pincock, 2007b)). Here the explanans — that no one has successfully crossed the bridges in a certain way — is itself stronger than ordinary causal laws such as the force laws. We could presumably change the laws, and it would still hold that there would be no successful crossing. Thus, to explain this latter fact we must appeal to something modally stronger — in this case, a mathematical necessity.

[51]This brief summary suppresses many of the subtleties of Lange's view, especially concerning the rich hierarchy of necessities he identifies. It would be interesting to investigate in detail how to map the necessities found in theoretical computer science onto Lange's hiearchy, but unfortunately would take us too far afield.

Similarly, it would seem that the reason why my laptop fails to solve the halting problem is that (a) it has a certain *computational* structure, of the sort roughly captured by a Turing machine, and (b) it is mathematically necessary that no object with that structure can solve the halting problem. As with the bridges case, the explandum seems itself to be more necessary than ordinary causal laws. Vary the force laws, and my laptop still wouldn't solve the halting problem. Accordingly, to explain it we need something stronger. Turing's result that the halting problem is Turing-uncomputable explains this in part because it entails that no physical implementation of a Turing machine solves the halting problem. Moreover, it explains why the explanandum has stronger-than-physical necessity: the reason is that it is entailed by a result which itself holds as a matter of mathematical necessity.

Elsewhere, Lange suggests that explanations by constraint concern the 'framework' in which more ordinary causal explanations operate: these explanations work "not by describing the world's actual causal structure, but rather by showing how the explanandum arises from the framework that any *possible* physical system ... must inhabit" (Lange, 2017, 30). This contrasts with ordinary causal explanation, because such explanations take for granted a certain framework — e.g., as captured by the force laws — in which causes operate. Explanations by constraint, by contrast, often arise by consideration of the framework of causal explanation itself and what must (or cannot) be the case, given how it is constituted.

Clearly limitative explanations do not rely on claims about the framework that *any* possible physical system must inhabit. However, it is not implausible to think that they concern general facts about the frameworks that structure *causal* computational explanations in particular. To see this, consider an uncontroversially causal

example of computational explanation, such as explanation by program execution (Piccinini, 2015, ch. 5). Explanations via program execution take for granted a stock of primitive computational operations (typically basic logical, arithmetical, or string-theoretic operations), and explain by citing a causal process composed of an appropriate sequence of basic operations. Which operations may be cited depends the programming language in question, since different languages typically supply a different stock of primitive operations. Different languages thus delimit a broad class of potential causal explanations, which differ primarily with respect to the composition of the causal processes they may describe.

A different kind of computational explanation becomes possible, however, when we focus on a fixed class of computational operations and resources — for example, by focusing on a fixed programming language — and then ask what problems can be solved with respect to that fixed class. This leads us towards limitative explanation. For, by showing that some problem falls outside of that class, we thereby show that no program written in that language can solve that problem, and hence have a story about why attempts to do so systematically fail. Thus, limitative explanations arise from consideration of the framework structuring, not causal explanations in general, but rather causal *computational* explanations in particular.

All of this suggests that limitative explanations can be understood as a kind of non-causal, mathematical explanation by constraint. However, as noted above, I don't think this is the whole story. Part of what seems to make limitative results explanatorily powerful is their breadth: the unsolvability of the halting problem holds not just for physical realizations of Turing machines, but for *any* digital computing system. An adequate account of limitative explanations should explain their wide

applicability. The trouble is that merely noting that limitative explanations cite a mathematically necessary sufficient condition does not address this. Why not think that limitative results framed in terms of Turing machines apply *only* to a specific, restricted class of physical computing systems rather than to digital computing systems quite generally? I take up this question next.

4.5.2 Essential idealization

My account of the wide applicability of limitative results proceeds in two steps. To keep things simple, I start by looking more closely at the implementation relation connecting Turing machines to a single class of physical computing systems, namely human agents working effectively. Then, in the next subsection, I will consider how to extend this basic story to other kinds of physical computing systems.

As I said earlier, I wish to set aside specific theories of implementation for moment. Instead, I will start with the following intuitive idea: a physical system implements a computational model if that model captures the basic computational architecture of that system. This involves capturing, among other things, its basic computational operations and its memory structure.

Because different computational models take different computational operations as primitive and have different memory structures, implementation conditions will typically differ from model to model. Rather than try to tell a completely general story about implementation here, I'll illustrate with a standard deterministic Turing machine (DTM). DTMs are equipped with a read/write head and a one-dimensional tape. They manipulate symbols one at a time on the tape according to a finite, predetermined set of instructions. Thus, at least to a crude first approximation, a

physical system implements a DTM if it manipulates symbols one at a time according to a finite set of determinate instructions on a one-dimensional tape.

So construed, however, few physical systems literally implement DTMs. This is for two main reasons. One is that the DTM architecture differs substantially from that found in many contemporary computing systems. For instance, few contemporary systems manipulate symbols one at a time, nor do they manipulate these symbols by scanning back and forth across a one-dimensional memory. A second reason is that DTMs are highly idealized. For instance, physical devices are prone to errors and malfunctions, and accordingly will not always follow their instructions perfectly in the way that DTMs do.

Among physical systems that might reasonably be construed as DTMs, human agents working effectively are perhaps the most plausible example. This is of course unsurprising given that Turing's characterization aimed, in the first instance, to capture the basic elements of effective human calculation (Sieg, 2009). Nevertheless, even here the differences are substantial. Whereas DTMs are assumed to never break down, to follow instructions perfectly, and to have an infinitely long tape (or at least potentially infinite: we can always add more tape if we need it), humans working effectively may fail to follow instructions correctly, only have a finite amount of memory to work on (there is only so much scratch paper in a finite universe), and, alas, will in the fullness of time break down.

These observations suggest that humans working effectively implement DTMs only under significant idealization. This is in some respects a familiar point. Quite often we must idealize physical systems to bring mathematics to bear. However, as I will argue next, what is striking about the computational case is that absent these idealizations

limitative results do not even begin to apply to physical computing systems. These idealizations are thus essential to limitative explanations.[52]

First, notice that actual human calculators have access to only a finite amount of memory. In fact, they are more literally described as Turing machines with only a finite amount of tape, sometimes known as bounded tape Turing machines (BTTMs). Human agents working effectively implement full-strength DTMs only under the idealizing assumption that they have unbounded memory. Now consider an in principle unsolvable problem such as the halting problem. In practical terms, the unsolvability of the halting problem ensures that no human calculator can determine whether a given algorithm will halt on a given input. Strikingly, however, it turns out that this problem is unsolvable only under the idealization that human calculators have unbounded memory resources.

To see this, consider the bounded halting problem:

$$BOUNDED\text{-}HALT(M, x, b) = \begin{cases} 1 & \text{if } M \text{ halts on } x \text{ when restricted} \\ & \text{to } b \text{ bits of memory} \\ 0 & \text{otherwise} \end{cases}$$

This is the problem of determining whether a machine with only a bounded amount of memory halts on a given input. This problem is computable, because a deterministic system with bounded memory has only finitely many possible configurations (i.e., combinations of internal states and memory contents). Thus, we can let the system run until either it produces the desired output, or it goes into a previously seen configuration. In the latter case, because the system is deterministic we know that the system has entered an infinite loop, and so can determine that it will never halt.[53]

[52]There are significant questions about how mathematics applies under idealization, but I propose to set them aside for the time being. See (Pincock, 2007a; Bueno and French, 2018) for discussion.

[53]See Theorem 5.9 of (Sipser, 2013, 222).

Given all of this, we could in principle simulate M on x using only b bits of memory and thereby compute $BOUNDED\text{-}HALT$.

There is thus a sense in which the halting problem doesn't even *concern* ordinary human calculators. If any version of the halting problem applies to them, it's the bounded halting problem. But that problem is computable. If we knew how much memory they had at their disposal, we could determine whether a human calculator following an effective procedure would halt on a given input. But if this is right, then it would seem that limitative results such as the halting problem do not after all apply to physical computing systems. And if these results do not apply, then they quite clearly cannot be used to explain anything about these systems either.

So why, despite this fact, do computer scientists continue to use DTMs rather than, say, BTTMs to investigate the powers of actual physical computing systems? Part of the reason is convenience. Even though the bounded halting problem is computable, it's **NP**-complete (Garey and Johnson, 1979, 175). Thus for all practical purposes it's just as hard to solve the one problem as the other. Because of this, the choice to use the DTM model rather than the BTTM model comes down partly to practical considerations such as the familiarity and mathematical elegance of the DTM model over its bounded counterpart.

However, a much more important point is that we treat human calculators as if they were DTMs rather than BTTMs is that the former allows us to capture deeper facts about the computational powers of humans working effectively. The fact that we only have finite memory to work with is not so much a reflection on our basic computational capacities so much as a contingent fact about the kind of world we happen to find ourselves in. If human calculators *did* have an unbounded

memory resources, Turing machines would much more closely approximate them and the traditional halting problem would more straightforwardly apply. The problem is 'merely' that the world doesn't cooperate, as it were.

Another way to put the point is that idealizations that provision more memory do not require that we change the basic computational operations carried out in effective human calculation (roughly, finite operations on bounded data structures). For this reason, they allow identify deep facts about what kinds of problems can and cannot be solved using such operations. Contrast this with idealizations concerning basic computational operations, for instance by allowing infinitary instructions or architectures in which successive operations are executed twice as quickly. These idealizations seem to depart much more drastically from effective human calculation than do idealizations concerning memory. Although a human working effectively may in some sense, and under significant idealization, implement an infinitary computational model, it is much less clear that characterizing human calculation this way reveals much of interest about the powers of actual human calculation.

Thus it is only under significant idealization that limitative results framed in terms of DTMs apply to human agents working effectively. The required idealizations hold fixed the basic computational operations carried out in effective human calculation, and this idealization is justified on the grounds that basic computational operations are much more intimately tied to a system's computational powers than are facts about how much space or time the system has available. Moreover, these idealizations are essential to limitative explanations in the sense that without them limitative results do not even apply to physical computing systems.

4.5.3 Simulation equivalence

The final part of my account extends this story to explain how these results apply as widely as they do. Here the problems are of a rather different character. Earlier I noted that the DTM architecture differs significantly from the architecture found in many actual computing systems. Idealizations notwithstanding, why should we think that limitative results framed in terms of DTMs apply to systems with widely different computational architectures?

To bring out the problem, consider a contemporary microprocessor. Like a DTM, a microprocessor has memory and processing unit, and manipulates finitely many digits at a time. But the similarities end there (plus or minus a few details). Unlike a Turing machine, the physical system's workspace is broken up into different components (registers, RAM, storage, etc.), it operates directly on 32- or 64-bit words, and its datapath is typically highly parallelized, to name just a few differences.

Indeed, contemporary microprocessors are much more accurately described as register machines. Register machines are an idealized representation of the von Neumann architecture employed in many contemporary computers. Whereas a DTM has a single contiguous block of one-dimensional memory, register machines are equipped with a bank of discrete memory locations ('registers'), each of which holds an integer. And whereas DTMs take certain string-theoretic operations as primitive (e.g. reading, erasing, and writing individual symbols), register machines typically take as primitive basic logical and arithmetical operations. In light of these architectural similarities

it is much more natural to think that microprocessors implement register machines (under appropriate idealizations) than that they implement Turing machines.[54]

In spite of this, many limitative results framed in terms of DTMs are nonetheless taken to apply to contemporary microprocessors. How do we reconcile this with the fact that microprocessors are more accurately described as register machines? The answer is that DTMs are, in a sense to be explained shortly, computationally equally as powerful as register machines. Thus, limits on the computational powers of DTMs carry over to register machines and thereby to their implementations.

The standard technique for showing that two computational models M_1 and M_2 are computationally equally powerful is to show that any given procedure framed in terms of one can be simulated by the other. This involves demonstrating how to systematically transform an M_1-computation into an M_2-computation, and vice versa. In practice, this is facilitated by first proving a universality theorem, which identifies a single machine which can solve any problem solvable by a given model. For example, Turing famously constructed a universal Turing machine, capable of solving any problem solvable by a DTM. With a pair of universality theorems in hand, we need only show how to transform computations carried out by one universal machine in terms of the other. When two computational models are equivalent in this sense, I will say that they are simulation equivalent.

Simulation equivalent models can solve exactly the same computational problems. On the one hand, suppose we know how to solve a problem with an M_1 machine. Then given the simulation equivalence of M_1 and M_2, we can systematically transform the M_1-solution into an M_2-solution. Similarly, if no M_1 machine solves some problem,

[54]Register machines were introduced by (Shepherdson and Sturgis, 1963). See (Cooper, 2004) for a more recent treatment.

then no M_2-solution exists either. For suppose not. By their simulation equivalence, we could transform the M_2-solution into an M_1 solution, a contradiction.

Computational models simulation equivalent to Turing machines are said to be Turing-complete. A wide variety of computational models are known to be Turing complete, including register machines and many different programming languages. Because simulation equivalence is a bona fide equivalence relation, all of these models are computationally equipowerful. Thus a limitative result framed in terms of one member of the class automatically applies to the class as a whole.

This notion of simulation equivalence captures a sense in which two computational models can solve the same problems in principle. A similar but more restricted notion of simulation equivalence can be used to characterize models which can solve the same problems *tractably*. Here the idea is to restrict attention to simulations which induce at most a polynomial slowdown in the simulating system. Two models related by a polynomial time simulation are polynomially equivalent. Since polynomials are closed under addition, anything (not) tractably solvable by one computational model is (not) tractably solvable by any polynomially equivalent model.

I can now explain how limitative results apply so widely. Different kinds of physical computing systems directly implement different kinds of computational models. Which computational model a given system implements depends on its architectural features: its primitive operations, memory organization, and so forth. For instance, humans working effectively implement DTMs, while digital computers machines implement register machines. Limitative results framed in terms of one kind of computational model apply to implementations of a different kind of computational model

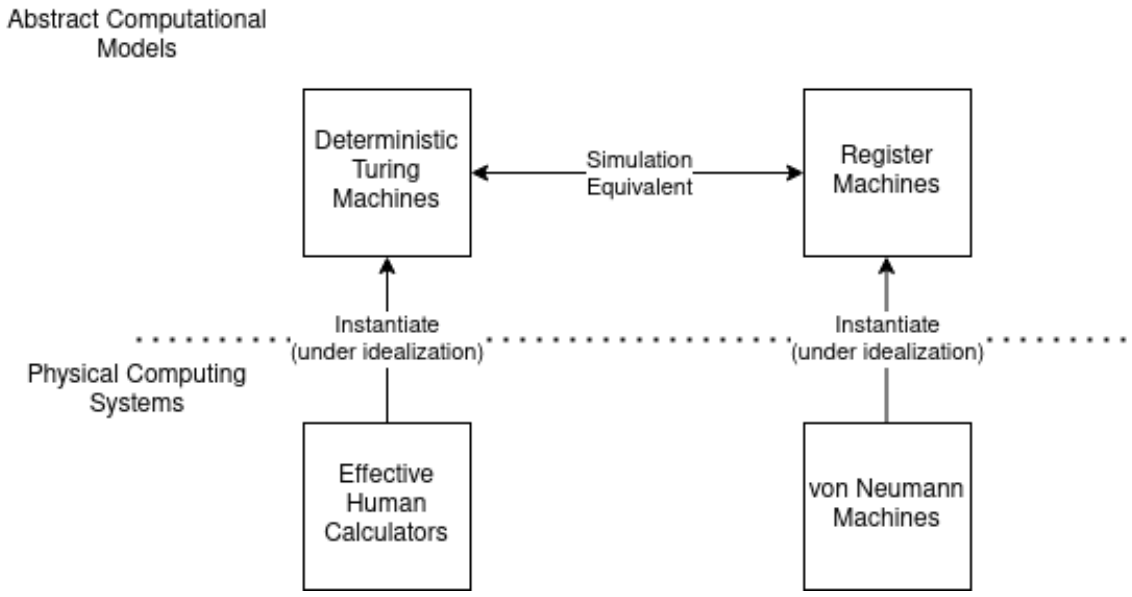

Figure 4.1: How limitative results apply widely.

just in case the two models are simulation equivalent. (See figure 4.1 for a partial sketch of the situation.)

4.6 Upshots for computational explanation

In this section and the next I want to trace out, in a preliminary way, some upshots of my discussion for broader issues in the philosophy of computation.[55] The first of these issues concerns computational explanation. By and large, philosophical accounts of computational explanation treat it as a kind of causal explanation. Indeed, at times computational explanations are held up as causal explanations *par excellence*. For want of a name, I'll call this the received view of computational explanation:

The received view of computational explanation

All computational explanations are causal explanations.

[55]This section and the next are excerpted and highly abridged from two stand-alone papers, currently in progress.

However, if my account of limitative explanations is correct then limitative explanations are a kind of non-causal, mathematical explanation. If, moreover, they are bona fide computational explanations, then the received view is false. That, at least, is the main argument of this section:

Premise 1. Limitative explanations are computational explanations.

Premise 2. Limitative explanations are not causal explanations.

Conclusion. Not all computational explanations are causal explanations.

If this argument is sound, then the received view is false and limitative explanations are a kind of non-causal computational explanation hitherto unaccounted by theories of computational explanation. The goal for the remainder of this section is to elaborate on the received view and then to briefly motivate each premise of the argument.

4.6.1 Background on causal explanation

Since I will argue that limitative explanations are not causal explanations, it will be useful to have a reasonably precise characterization of causal explanation on the table. This will not only provide a standard for discriminating causal from non-causal explanations, but it will also help clarify the received view.

Although there are many different accounts of causal explanation in the literature, the interventionist account (Woodward, 2003; Spirtes et al., 2000) is especially appropriate for my purposes. This is for a couple different reasons. For one thing, the account is well-suited for capturing causal explanation in scientific contexts. Thus, it is a natural framework for understanding extant views of computational explanation

which assimilate it to causal explanation. More importantly for my purposes, the interventionist account is highly liberal among contemporary accounts of causal explanation. Compared, for instance, to mechanistic accounts (e.g., (Glennan, 2017)), it treats a wider range of explanations as causal. Thus, if limitative explanations are causal explanations, we should be able to capture this fact in interventionist terms. And if limitative explanations cannot be so captured, this is strong evidence they are not causal explanations.

Although I assume general familiarity with the interventionist framework, here are a few reminders. Like many contemporary accounts of causal explanation, it rests on the idea that causation should be understood in terms of counterfactual dependence. The guiding idea is that if X is causally relevant to Y (or, as it is sometimes put, if X is a cause of Y), then it is possible to change Y by changing X. On this view, causal explanations reveal patterns of counterfactual dependence between X and Y, and provide what is sometimes called 'what-if-things-had-been-different' information, or w-information (Woodward, 2003, 203). On this broad conception of causal explanation, in the sense that any information about how changes in X make for changes in Y constitutes a causal explanation of Y in terms of X.

On this picture, causation is a relation between variables. Roughly put, variables can be thought of as properties which can take on different values at different times. A variable X is a cause of a variable Y (with respect to a variable set V) iff there is a possible intervention on X that will change the value of Y, when holding fixed the values of all other variables in V (Woodward, 2003, 59). While there is disagreement about how to understand the notion of possibility at play here, interventions are traditionally taken to involve physically possible manipulations to the variables in

question. This is important, because it helps to distinguish causal counterfactual dependencies from other sorts of dependencies, such as those that rely on mathematical, constitutive, or conceptual relationships. Thus, if an explanation works by providing information about how one variable changes under possible interventions to another, it is a causal explanation by the interventionist's lights.

4.6.2 The received view

There are many accounts of computational explanation in the literature, and it would be tedious to survey them all. My goal here is to consider a few of the more popular accounts and to suggest that they all endorse the idea that that computational explanations are causal explanations. I shall focus on accounts that treat computational explanation in formal-syntactic, mechanistic, and representational terms, respectively.

I should note also that not all of the philosophers I discuss would accept the claim that computational explanations are causal explanations. This is because many of them do not endorse interventionism, but instead some more restricted account of causal explanation. My claim that these accounts treat computational explanations as causal explanations is not the claim about how their defenders conceive of them. Rather, it is the claim that on the broad conception of causal explanation furnished by the interventionist account, computational explanations as conceived of by these philosophers turn out to be causal explanations.

The Formal-Syntactic Account

One historically influential conception of computational explanation holds that it is explanation in terms of a system's formal-syntactic features. This view rests on the

the formal-syntactic conception of computation, a classic statement of which is due to Fodor:

> Computational processes are both symbolic and formal. They are symbolic because they are defined over representations, and they are formal because they apply to representations in virtue of (roughly) the syntax of the representations ... being syntactic is a way of not being semantic. Formal operations are the ones that are specified without reference to such semantic properties of representations as, for example, truth, reference, and meaning. (Fodor, 1981, 226-7)

Other proponents of the formal-syntactic conception of computation include Field (1978) and Stich (1983).[56] Although these philosophers does not explicitly offer a theory of computational explanation, it seems reasonably clear that they take computational explanations to be causal explanations which advert to formal-syntactic states or processes of a system. For example, here's how Stich proposes to use of formal-syntactic conception to frame his preferred computational theory of cognition, the syntactic theory of mind (STM):

> The basic idea of the STM is that the cognitive states whose interaction is (in part) responsible for behavior can be systematically mapped to abstract syntactic objects in such a way that causal interactions among cognitive states, as well as causal links with stimuli and behavioral events, can be described in terms of the syntactic properties and relations of the abstract objects to which the cognitive states are mapped ... If this is right, then it will be natural to view cognitive state tokens as tokens of abstract syntactic objects. (Stich, 1983, 149)

Thus, on the view of computational explanation suggested by this passage, a computational explanation of some bit of behavior proceeds, roughly, by citing causally relevant cognitive states described syntactically. To take a slight adaptation of one of Stich's examples, the explanation why a system comes to be in a particular syntactic

[56] A more recent incarnation of the formal-syntactic view is developed by Chalmers (2011).

state B is that (a) the system was in the syntactic states A and $A \rightarrow B$, and (b) it is in general true that when systems are in the states A and $A \rightarrow B$, they go into the syntactic state B. It is not hard to see how to capture this in interventionist terms. Suppose we intervened on a system so that it failed to be in either state A or $A \rightarrow B$. Then, in light of (b), it would not go into state B either (assuming for simplicity that the only way to enter state B is). Thus, this explanation would count as causal according to interventionism.

The Mechanistic Account

A currently popular view identifies computational explanation with mechanistic explanation:

> Computational explanation is a form of mechanistic explanation. As a long philosophical tradition has recognized, mechanistic explanation is explanation in terms of a system's components, functional properties, and organization. Computational explanation is the form taken by mechanistic explanation when the activity of a mechanism can be accurately described as the processing of vehicles in accordance with appropriate rules. (Piccinini, 2015, 142)

On this view, computational explanation is distinctive among mechanistic explanations in that involves transformations of 'medium-independent' vehicles. Transformations vehicles are medium-independent if, roughly, they are "sensitive only to differences between portions (i.e., spatiotemporal parts) of the vehicles along specific dimensions of variation—it is insensitive to any other physical properties of the vehicles" (Piccinini, 2015, 122).

How does this fit the received view? Let me first distinguish between two styles of mechanistic explanation typically distinguish: etiological and constitutive (Craver and Kaplan, 2020; Craver and Tabery, 2019). Etiological mechanistic explanations

explain a phenomenon by appeal to its causally relevant antecedents. Constitutive mechanistic explanations, by contrast, explain a phenomenon in terms of the hierarchical arrangement of a mechanism's components and the way those components interact. One and the same phenomenon can be explained both etiologically and constitutively. For instance, an etiological explanation why a calculator displayed '10' might appeal to the fact that it was at an earlier moment given '5', '5', and '+' as input. A constitutive explanation might appeal (among other things) to the way that the atomic logical or mathematical operations performed by the device (e.g., atomic operations on individual binary digits) constitute the more complex logical or mathematical operations (e.g., operations on binary strings) it performs.

Etiological computational explanations are obviously causal explanations, and it is not implausible to think that they can be captured in interventionist terms.[57] It is more controversial whether constitutive mechanistic explanations can be understood causally, in interventionist terms or otherwise. Some say yes (Craver, 2007), while others say no (Gillett, 2020). I will not attempt to settle that issue here. I constitutive mechanistic explanations are causal explanations, then so much the better for my argument. If they are not, then, if in addition some computational explanations are constitutive mechanistic explanations, we have an alternative challenge to the received view. Nevertheless, since limitative explanations are not plausibly understood as constitutive explanations, the claim that they are a kind of non-causal computational explanation is novel and in my estimation is a more compelling challenge to the received view.

[57]There is some dispute about this, however. Woodward (2011) argues that mechanistic causal relations 'bottom out' in counterfactual dependencies of the sort emphasized by the interventionist. If this is right, it suggests that mechanistic etiological explanations can be understood in interventionist terms. But see (Waskan, 2011) for some discussion.

Representational explanation

Computational explanation is often associated with explanation in terms of a system's representational properties. Although there is longstanding dispute about whether representational properties per se are causally efficacious, and if so, how, it is more widely accepted that explanations in terms of a system's representational properties are in at least some cases causal explanations (Fodor, 1981; Peacocke, 1995, 1999; Rescorla, 2017b; Burge, 2010). It is interesting to note that (Rescorla, 2014a) goes further than this, and develops an account of the causal efficacy of representational properties in broadly interventionist terms. However, such a strong claim isn't strictly speaking required to accept that explanation in terms of a system's representational properties can be a kind of causal explanation.

Limitative explanations are computational explanations

My argument for premise 1 is brief, because I think this premise is highly plausible. Not only are limitative explanations framed in terms of core computational notions like Turing machines, but they also play a central role in contemporary computer science. Insofar as explanatory practice is our best guide to computational explanation, it is hard to imagine a better indicator than this that limitative explanations are computational explanations. To deny premise 1, one would apparently have to reject or at least radically reinterpret large swathes of computational practice. This, to my mind, is a highly unpalatable option.

Limitative explanations are not causal explanations

I have two arguments for premise 2. One 'indirect' argument proceeds by noting the similarities between limitative explanations and mathematical explanations by

constraint, a la Lange. If limitative explanations are a kind of explanation by constraint, then, insofar as explanations by constraint are in general non-causal, so too will be limitative explanations. This argument is indirect in the sense that it doesn't appeal to a specific conception of causal explanation. Instead, it assumes that explanations by constraint are in general non-causal (so says Lange, at any rate), limitative explanations are at least in part explanations by constraint, and concludes on this basis that limitative explanations are non-causal too.

Since some philosophers may balk at the claim that explanations by constraint are non-causal — or, at least, hold that it requires further argument — it will be useful to have a 'direct' argument for premise 2 as well. The direct argument proceeds by arguing that limitative explanations cannot be captured in interventionist terms. I'll sketch this argument next.

For concreteness, I'll focus on a specific limitative explanation. Let the explanandum be the failure of a particular system, such as a contemporary laptop, to solve the halting problem. The explanans is that the laptop implements a certain computational model, such as a universal Turing machine, plus the fact that no universal Turing machines solves the halting problem. It is instructive to consider how we might intervene on the explanans in an effort to reveal counterfactual dependencies between the explanans and explanandum. If such dependencies exist, this is some evidence that the explanation can be captured in interventionist terms.

I think we can immediately rule out 'interventions' on the limitative result that no Turing machine solves the halting problem. For one thing, there is little sense in which we can 'intervene' on a mathematical fact or object. What would that even mean? At least on standard views, Turing machines are abstract objects and limitative results

hold with whatever degree of necessity is had by mathematical facts more generally. These are simply not the kinds of things we can manipulate. Moreover, we if we could make sense of the idea of a mathematical 'intervention', applying it to the case at hand would require us to be able to make sense of countermathematicals like the following:

> If a universal Turing machine could solve the halting problem, then so too could a physical system that implements it.

As I've noted in earlier chapters, there are serious challenges for interpreting countermathematicals such as this. In light of these worries, it is hard to see the explanatory relationship between limitative results and the physical systems they apply to as one of counterfactual dependence. Rather, it would appear to involve the much stronger relation of mathematical necessitation.

The other possibility would be to intervene on the computational model implemented by the laptop. Here we have at least some intuitive sense of what this might involve, since it does seem possible to change what computations a system implements by, for instance, changing its hardware or running a different program on it. Nevertheless, there are no obvious physical interventions we could make on the laptop that would allow it to solve the halting problem. It would seem the only possible changes either leave us with a system equivalent in power to a universal Turing machine or would make the laptop strictly weaker than a universal Turing machine. Either way, there is no way to intervene on the laptop to make it solve the halting problem.

The upshot of these points is that limitative explanations do not seem to provide what-if-things-had-been-different information. For, given the mathematical necessity of limitative theorems, it is quite literally impossible for things to have been different

in the relevant respects. If a physical system implements a Turing machine, then it is impossible for that system (in its capacity as a Turing machine) to solve the halting problem. By appealing to mathematical limits on the computational powers of certain kinds of machines, limitative explanations provide, as it were, how-things-must-have-been information. Given the fact that a system implements a Turing machine (or equivalent model), it couldn't have helped but fail to solve certain kinds of problems.

This line of reasoning could be objected to in various ways, and I do not take the argument presented here, at least as it stands, to be definitive. Nonetheless, it at least suggests that limitative explanations prose a serious prima facie challenge to the received view of computational explanation. A fuller development and appraisal of the ideas presented in this section remains on the agenda for future work.

4.7 Upshots for computational implementation

On the account developed above, limitative results apply to physical computing systems only under significant idealization. As I shall argue next, this poses a problem for many traditional accounts of computational implementation. In particular, it is a problem for accounts which conceive of the relationship between physical systems and computational models as one of isomorphism. For, as I shall argue, for highly idealized computational models the required isomorphisms may not obtain.

To make matters concrete, I'll focus on a particular isomorphism-based account due to Michael Rescorla. In order to provide uniform implementation conditions for different computational models, Rescorla introduces the notion of a canonical state space descriptions $\langle S, I, \Omega, s_0 \rangle$, where S is a set of computational states, I is a set of inputs, $\Omega : S \times I \rightarrow S$ is a transition function, and s_0 is a designated start space.

To obtain the implementation conditions for a given computational model, such as a particular Turing machine, we first redescribe it as a CSSD, and then consult the following definition:

> **CSSD Implementation**
>
> A physical system P implements a CSSD $M = \langle S, I, \Omega, s_0 \rangle$ just in case there is a function f mapping states and inputs of P to states and inputs of M such that:
>
> 1. For each $s \in S$, there is a possible state p of P such that $f(p) = s$.
> 2. For each $i \in I$, there is a possible input k to P such that $f(k) = i$.
> 3. P conforms to the transition function Ω. More precisely: if P were to enter into state p and to receive input k, where $f(p) = s$ and $f(k) = i$, then P would transit to a state p^* at the next stage of computation, where $f(p^*) = \Omega(s, i)$.
> 4. Absent external interference or internal malfunction, P always begins computation in state p_0, where $f(p_0) = s_0$ (assuming that $s_0 \in S$).[58]

Now consider what it would take for, say, a microprocessor to implement some universal Turing machine according to this account. First, we translate the machine into an appropriate CSSD, with corresponding state space S, inputs I, and transition function Ω. Although the details of the translation depend significantly on the specifics of the Turing machine (e.g., its alphabet, number of heads, internal states, and so forth), we can at least note that I, and perhaps also S (depending on the translation), will be infinite. This is because universal TMs are defined over countably infinite sets of inputs.

As I noted in section 3, however, physical computing systems have only finite memory, while Turing machines in general have unbounded memory. So there will be possible inputs $i \in I$ and/or states $s \in S$ with no analogue in any microprocessor — in

[58](Rescorla, 2014b, 1281). This definition modifies Rescorla's slightly in order to make the action of the implementation function f explicit.

which case clause (2) and potentially clause (1) of the above definition fail. Moreover, for those inputs too large for the microprocessor to handle, clause (3) will be true only vacuously if at all, since its antecedent will be typically fail to hold.[59] Indeed, according to the above account it would seem that microprocessors never implement full-strength Turing machines; at best they implement bounded tape Turing machines. Since BTTMs accept only finitely many inputs, it will be possible to find a CSSD characterization of a BTTM that satisfies clauses (1) and (2). However, as I noted in section 3, this conclusion sits awkwardly with computational practice. In particular, it becomes hard to understand why computer scientists take limtiative results such as the unsolvability of the halting problem to describe and explain systems such as microprocessors if these systems do not implement Turing machines.

Perhaps the most obvious response to this worry is to amend the definition of implementation, for instance by dropping or modifying clauses (1) and/or (2). The most straightforward response drops the requirement that each $i \in I$ or $s \in S$ be the value under f of some input or state of P. Instead, we simply require that each input and state are in I and P, respectively, but don't require that this exhaust I or P:

1*. If p is a possible state of P, then for some $s \in S$, $f(p) = s$.

2*. If k is a possible input to P, then for some $i \in I$, $f(k) = i$.

The trouble with this redefinition is that it may turn out that a system implements many different Turing machines the action of whose transition function coincides for those states and/or inputs in the image of f, but which diverges otherwise. This may

[59]This depends to a large extent on how we interpret the counterfactual conditional in (3). On a standard similarity-based semantics (Lewis, 1973), for instance, and for sufficiently large choices of input i, there will be no nearby possible worlds in which P can receive i as input.

be a problem, because those Turing machines may have radically different computational properties owing to the behavior of their transition function on the states and inputs lacking physical counterparts. For example, the machines might differ in their asymptotic running time complexity. This is because results about asymptotic performance are results about the behavior of a machine over all possible inputs. Indeed, in some cases the asymptotic behavior of a machine 'kicks in' only for sufficiently large inputs. Nevertheless, results about the asymptotic performance of Turing machines are taken to bear on the performance capabilities of actual physical devices. But if a system implements many machines which differ in their asymptotic performance, then it is hard to see how to pick just one (or even a select few) of these machines to describe the computational properties of the microprocessor in particular, for the microprocessor would legitimately implement each according to the modified definition.[60]

Briefly, the proponent of the isomorphism-based account faces a dilemma. Either the implementation function f maps onto the states and inputs of a mathematical computation, or it does not. If it does, the physical systems do not implement Turing machines, contra computer scientific practice. If it does — if f merely maps *into* but not *onto* S and I — then it is indeterminate which Turing machine a system implements. And this raises problems for how we use such machines to describe the properties and behavior of physical systems.

Obviously, this this dilemma is sketchy and open to challenge. Nevertheless, it suggests that the implementation conditions of highly idealized computational models such as Turing machines are not best understood in terms of isomorphism. And while

[60]This worry bears certain affinities to the rule-following paradox (Kripke, 2000). For some recent discussion, see (Warren, 2020). I will not pursue this connection here.

I am happy to concede that isomorphism-based accounts may correctly capture the implementation conditions for certain kinds of computational model — especially, if not exclusively, those which require only finite memory — it seems to me they mischaracterize the implementation conditions for others. And this is a problem, if, as I've argued, highly idealized models often underwrite limitative explanations in computer science.

So, if physical computing systems don't implement Turing machines in virtue of being isomorphic to them, then in virtue of what *do* they implement Turing machines? I take up this question in the next chapter.

Chapter 5: Implementation as resemblance

5.1 Introduction

I'll start by summarizing my conclusion so far. In Chapter 2 I argued that we ought to take a 'mathematics-first' approach to theories of physical computation. In particular, I argued that we ought to answer the status and identity questions at least partly in terms of the mathematical computations implemented by physical systems. In Chapter 3, I argued that the most promising way for implementationist theories to avoid the Putnam-Searle triviality worry emphasizes the relativity and context-sensitivity of computational implementation. And in Chapter 4, through an examination of limitative explanations, I argued that implementation does not always involve an isomorphism between a physical system and the computation it implements.

This chapter synthesizes these results and uses them to motivate a new account of computational implementation, which I call the *resemblance account*. On this account, implementation is not always or solely a matter of being isomorphic to a mathematically characterized computation. Rather, implementation is understood in terms of the more general notion of resemblance. Whether a physical system computes, and if so, what it computes, is determined by the mathematical computation(s) it resembles.

The goal for this chapter is to develop the resemblance account in broad outline and to make a prima facie case for it. I should emphasize at the outset, however, that my discussion is programmatic. I do not have the space here to develop some finer points of the account in the amount of detail they deserve. For now they remain on the agenda for future work, and I will flag them as they arise.

Here is the plan. Section 2 introduces the resemblance account and provides some intuitive motivation for it. Sections 3 and 4 develop the account in more detail. Section 5 elaborates on the account and ties off a few loose ends. Section 6 concludes by considering some directions for future work.

5.2 The resemblance account

I propose to begin at the beginning. In his landmark 1936 paper, Turing offers the following description of a Turing machine (what Turing called an 'a-machine'):

> The machine is supplied with a "tape" (the analogue of paper) running through it, and divided into sections (called "squares") each capable of bearing a "symbol" ... the configuration [of the machine] determines the possible behaviour of the machine. In some of the configurations in which the scanned square is blank (i.e. bears no symbol) the machine writes down a new symbol on the scanned square: in other configurations it erases the scanned symbol. The machine may also change the square which is being scanned, but only by shifting it one place to right or left. (Turing, 1936, 231)

As we know, Turing arrived at this conception by carefully considering the activity of human workers proceeding effectively. The restriction that Turing machines may only 'observe' one symbol at a time, for instance, is justified on the grounds that human workers can only distinguish between finitely many different primitive symbol types (Turing, 1936, 249-52).

The importance of Turing's insight is not hard to appreciate. By linking the characterization of a Turing machine directly to the activities of actual human workers, Turing's analysis sheds light on the computational capacities and limitations of humans working effectively. Very roughly, the computational power of Turing machines bears on the computational power of effective human workers because the former resemble the latter in certain important respects: both have certain 'perceptual' limitations, both follow only finitely many instructions one at a time, and so on. Indeed, alternative analyses, such as λ-definability or Herbrand-Gödel general recursivity, were unsatisfactory because they fail to adequately illuminate the basic activities of a human working effectively.

It seems to me that Turing's analysis contains the essentials of the resemblance account. Turing machines bear on the computational powers of human workers because, and to the extent that, the former resemble the latter in certain respects. The resemblance account generalizes Turing's insight:

> **The Resemblance Account**
>
> A physical system implements a computational architecture in certain antecedently specified respects just in case, and to the extent that, it resembles that architecture in those respects.

Subsequent sections unpack the notion of a computational architecture and will explain the notion of resemblance at play. Before that, let's consider how the resemblance account answers the status and identity questions about physical computing systems.

First, on this account, being a physical computing is a matter of resembling what I will call a computational architecture. Computational architectures are objects like Turing machines, microarchitecture specifications, programs, or neural networks.

They are 'blueprints' for physical computing devices in the sense that they specify a range of features a system must have in order to count as a computing device of a particular sort. If a physical system resembles an architecture in the appropriate respects, it thereby counts as a physical computing system. If a system resembles no architecture in the appropriate respects, it does not.

With respect to the identity question, physical computing systems are individuated by what they compute. This is determined by the computational architecture(s) it resembles. Although there are different possibilities for what we might take systems to compute, for illustrative purposes consider the notion of a system computing a function. This is a coarse-grained way of individuating systems computationally, because there are typically different ways of computing the same function. In any event, we know what it is for a Turing machine to compute a function: crudely, M computes f just in case it halts with $f(x)$ on its tape when given x as input. Accordingly, a physical system computes f if it resembles M in the appropriate respects. A physical system that resembles a different machine M^*, which computes a different function h, is accordingly is computationally distinct from a system that implements M.

One other aspect of the resemblance account is worth highlighting up front. Philosophical interest in physical computation has revolved largely around two closely related notions: that of a physical system being a computing system, and that of a physical system carrying out computations. At least to a first approximation, being a physical computing system is a property of a physical system as a whole, whereas being a computation is a property of a physical processes. The resemblance account takes the notion of being a physical computing system to be the more basic of the

two, in the sense that the kinds of computational processes that a system is capable of carrying out depends on the computational architecture(s) that the system implements.

In this respect the resemblance account follows practice in computer science. Much work in theoretical computer science begins by characterizing a computational architecture and then goes on to characterize the kinds of computational processes that can be carried out by that architecture. Similarly, computer engineering and architecture investigate the overall computational structure a system must have if it is to meet certain design standards concerning performance, reliability, and so forth. Thus, initial motivation for the resemblance account derives from this widespread aspect of computational practice.

5.3 Computational architectures

As I will use the term, a computational architecture is any mathematically or semi-mathematically characterized structure used by contemporary computer or cognitive scientists to describe physical systems computationally. So construed, computational architectures comprise a large and heterogenous class of computational objects. Despite their differences, some of which I catalogue below, computational architectures are united by the fact that they attempt to characterize the computational structure of physical systems, albeit in different ways and at different levels of granularity.

A word on terminology. The objects I refer to as 'computational architectures' are sometimes referred to as 'computational models' (see, e.g., (Rescorla, 2014b)), on the grounds that there are important similarities between the use of objects such as Turing machines in computer and cognitive science, and the use of mathematical

models encountered elsewhere in science. That is, the primary rationale for 'model'-talk is that Turing machines (etc.) are genuine scientific models. However, I am unhappy with this choice of terminology, for two reasons.

First, although I agree that computational architectures are scientific models, scientific models in general are used in many of different ways (Frigg and Hartmann, 2020). Computational architectures are unique in that they purport to characterize phenomena which are themselves computational, and they purport do so by describing the computational structure and organization of physical systems. 'Architecture' seems to me to better emphasize this aspect of how objects like Turing machines are used in computer science, at least sometimes. Second, 'computational model' is ambiguous (Piccinini, 2007). On one usage, it means what I mean by 'computational architecture'. On another usage, a computational model is a computational system, such as a program, which models some phenomenon which is not itself computational. Computational models of the weather are a prime example — they model weather patterns, but not by ascribing computational structure to the weather. Again, 'architecture' seems to me to better capture the focus on computational structure.

In any event, whatever we decide to call them, computational architectures are at the center of the resemblance account. This section starts with some illustrative examples before discussing their features more generally.

5.3.1 Some examples

The contemporary scientific literature contains a variety of different kind of computational formalism. These include machine models such Turing machines or finite automata, programs written in some programming language such as C, Java, or

Python, and formal languages such as regular or context-free languages.[61] Machine models and programs are perhaps the most familiar and straightforward examples of computational architectures in my sense.

For example, Turing's original description constitutes a blueprint which specifies the features a physical system must have in order to 'counts as' an Turing machine. Some of these features concern the physical or mechanical features of the device. For instance, on Turing's characterization the machines have a read/write head and a tape divided into squares, and operates by *scanning* the tape, *writing* symbols, and *shifting* left or right. Other features are more abstract, and concern the patterns or regularities the machine or its components exhibit. Others still concern what states of the device represent. The symbols on the tape refer to natural numbers, for instance, and a machine as a whole may be taken to represent, in some sense, the function it computes. The upshot of all of this is that if a physical system exhibits these sorts of features, it thereby counts as a Turing machine.

However, while Turing's characterization is illustrative, it is also highly idealized. Few physical systems compute if any in exactly the same way that Turing machines do. Elsewhere in computer science we find architectures that more closely approximate the structure of actual physical devices. On the more theoretical side are VLSI (very-large-scale-integration) models that attempt to capture the high-level structure and properties of contemporary microprocessors (Savage, 2008, ch. 12). On the more practical end is work in computer architecture and engineering which produces highly specific descriptions of computational architectures. For instance, the blueprint for a contemporary microprocessor is known as a microarchitecture specification Harris

[61] (Savage, 2008) is a nice recent survey of the various formalisms found in contemporary theoretical computer science.

and Harris (2013); Weste and Harris (2011). Microarchitecture specifications detail the features required for a system to count as a certain kind of microarchitecture. Some of these features are described explicitly, for instance that the system have a datapath with a certain pipelining scheme, certain components for sign extension operations, and so on. Others are left tacit, such as the requirement that the system be cast in a silicon wafer, that it have a certain clock rate, and so on.[62]

I also include programs among computational architectures. Unlike machine models, which specify a collection of components and a description of how those components interact, programs are better understood as lists of instructions. They are more abstract than machine models in that they do not always specify the kinds of components that carry out their instructions. However, just how much more abstract a given program is depends substantially on the language in which it is written. Programs written in an assembly or even machine code bear a much closer relationship to hardware than do programs written in high-level languages such as Java or Python. Nevertheless, despite this variability, programs are used to describe physical systems computationally and consequently are computational architectures according to the broad, ecumenical standard adopted here.

Although these are perhaps the clearest examples of computational architectures, I would also include various unconventional and non-standard computing systems in the class of computational architectures as well. Examples here include neural networks (Rumelhart et al., 1987), analog computing systems (Maley, 2021), DNA and RNA

[62]Microarchitecture descriptions are often extraordinarily complicated. For instance, the OpenSPARC specification, a relatively modest microarchitecture developed by Sun Microsystems, runs over five hundred pages. See https://www.oracle.com/servers/technologies/opensparc-t1-page.html.

computing systems (Akhlaghpour, 2022), reservoir computers (Tanaka et al., 2019), and quantum computers (Nielsen and Chuang, 2010), to name just a few.

5.3.2 General features

Next I will step back and consider computational architectures more abstractly. To start, I want to highlight the apparent dual nature of computational objects like Turing machines. Crack open a textbook on computability theory and you will likely find Turing machines described in two main ways. Sometimes they are described broadly physical terms, as devices with certain components (read/write head, tape, etc.) which interact with each other in specific ways (reading, writing, moving left or right, etc.). This is how Turing described them, at least initially. Soon after this however, one will typically encounter a formal, set-theoretic characterization like the following:

> A Turing machine is a 6-tuple $M = (\Gamma, \beta, Q, \delta, s, h)$ where:
>
> 1. Γ is the tape alphabet;
> 2. β is the blank symbol, with $\Gamma \cap \{\beta\} = \emptyset$;
> 3. Q is a finite set of states;
> 4. $\delta : Q \times (\Gamma \cup \{\beta\}) \to (\Gamma \cup \{\beta\}) \times \{L, N, R\}$ is the transition function;
> 5. s is the initial state;
> 6. $h \in Q$ is the accepting halt state.
>
> If M is in state q with letter a under tape head, and $\delta(q, a) = (q^*, a^*, C)$, its control unit enters state q^*, writes a^* under the head, and moves left, right, or not at all as C is L, R, or N, respectively (cf. (Savage, 2008; Sipser, 2013)).

What is the connection between these two ways of thinking about Turing machines? Given computability theory's express goal of proving theorems about what

can or cannot be computed by Turing machines, contemporary standards of mathematical rigor ensure that the set-theoretic characterization is all but inevitable (Burgess, 2015). Nevertheless, on its own the set-theoretic construal bears no obvious connection to physical computing systems. It is only by providing the formalism with a specific physical interpretation that it comes to bear on actual physical devices. This is the role of the physical description — it describes in a rough and ready way the connection between abstract mathematical formalism and the concrete systems that formalism applies to.

I think this dual character of Turing machines is representative of computational architectures in general. On my view, architectures consist of two parts. At the core of a given architecture is a formal, mathematically characterized computational structure. On top of this is a specific physical interpretation of that structure. For ease of reference, I'll call these the 'mathematical core' and the 'physical interpretation', respectively. Let me take each in turn.[63]

Mathematical core

The mathematical core has two main roles: it specifies the basic computational operations carried out by an architecture of a given kind, and it delineates the data structures over which those operations are carried out. For instance, in a classical Turing machine the basic operations are manipulations on single digits, and the memory structure is a single one-dimensional array. Variations on the basic Turing machine tinker with either the basic operations, or the memory structures, or both. For example, K-graph machines generalize Turing machines by taking the memory structure to

[63]The account developed here is indebted to recent work in the philosophy of scientific modeling and the applicability of mathematics, especially (Cartwright, 1983; Giere, 1988; Teller, 2001; Weisberg, 2013; Pincock, 2011).

be an undirected graph, the basic operation on which consists of 'rewiring' large but still finite chunk of graph (Sieg and Byrnes, 1996). More radical departures may involve infinitary instructions (as in (Hamkins and Lewis, 2000)) or more sophisticated data structures.

Just what form the computational operations and data structures take varies from architecture to architecture. Turing machines embody a sharp distinction between operations and memory, in that these two aspects correspond to different machine components: the instruction set and read/write head on the one hand, and the machine tape on the other. This is not always the case. In neural networks, for instance, the weights between nodes play a dual role: they store the learning history of the network, and through the activation function at the various nodes they are part of the computational operation performed by the network too.

Although each computational architecture has a mathematical core, this core is not always described in fully formal, set-theoretic terms. Typically, one finds fully set-theoretic characterizations only in contexts where mathematical rigor is at a premium: chiefly but not exclusively in computability and computational complexity theory. Usually a more informal description suffices. For instance, although it would be possible in principle to describe the mathematical core of a microarchitecture specification in purely set-theoretic terms — e.g., as a specific finite state automaton — there is little reason to do so. For purposes in computer architecture and engineering, a semi-formal description usually suffices. Even if it is not described in fully formal terms, the mathematical core exists nonetheless.

At the most abstract level, we can treat a data structure as a set of internal states S, plus a set of inputs I. The internal states may include output states, and

one of these might be a designated 'start' state. Computational operations can be captured in terms of a transition function $\phi : I \times S \rightarrow S$ (or transition relation, if the architecture is non-deterministic).[64] I should emphasize, however, that the mathematical core is rarely presented in such austere terms, nor should we try to translate a given architecture into some canonical description, such Rescorla's CSSDs. Doing so may obscure features of the architecture relevant to its implementation conditions (Sprevak, 2012). Thus although the mathematical core can for certain purposes be understood in terms of states plus transition function, it would be a mistake to try to frame implementation conditions for architectures the in these terms. I will return to this point below.

Finally, because it imposes no restrictions on I, S, or the action of ϕ, this characterization of the mathematical core casts a very wide net. This is intentional, because we wish to account for a wide range of different architectures. This includes highly idealized architectures, such as those with infinitary instructions, or oracle machines whose oracle contains solutions the halting problem. It seems to me undeniable that these are genuine computational architectures — they are the kinds of things that could be realized, if only physical reality cooperated.

Physical interpretation

The second part of a computational architecture is the physical interpretation. This consists of an assignment of physical components, properties, states, values, etc. to the mathematical core. One and the same mathematical core may be equipped with different physical interpretations. For instance, consider the mathematical core of a standard Turing machine, given above. One possible physical interpretation

[64]For a similar characterization, see, e.g., (Moschovakis, 2001).

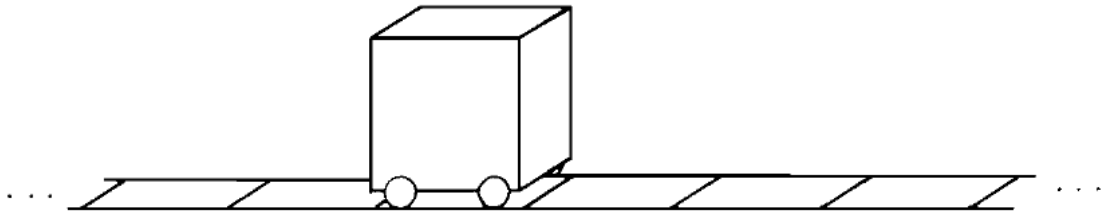

Figure 5.1: One way to be a Turing machine.

hews closely to Turing's original informal description, and takes the data structure to be a one-dimensional, segmented tape, and locates the basic operations in the behavior of a physical read/write head — perhaps something like a trolley cart — that literally moves left and right across the tape. This possibility is sketched in Figure **??** (courtesy of Boolos et al. (2007, 25)). Another possible physical interpretation, not without historical precedent, ties the formalism more closely to the behavior of humans working effectively. Here the states of the machine might correspond to states of a human agent or their brain, and the action of the transition function might be realized in the perceptual and motor apparatus of the agent (e.g., their eyes, hands, and the tip of the pen).

Other physical interpretations are of course possible. Indeed, in principle, there will be as many physical interpretations of a given mathematical core as there are assignments of physical components, properties, or values to the parts of that core. Which is to say: there will be quite a few. Rather than enumerate the possibilities here, I will simply point out that the physical interpretations of interest will often rely on the kinds of features that philosophers have long though important to physical computation, some of which we encountered in previous chapters. These include, but

are not limited to, specific physical properties, abstract causal/functional properties, representational properties, teleological properties, and pragmatic properties concerning how a system is or ought to be used. Of course, I don't take this list to exhaust the features that may be specified by a computational architecture. Indeed, it would be a mistake to try to specify such a list once-and-for-all. Instead, we should regard this list as open to addition or amendment, as computational scientists discover and design new kinds of computational architecture.

The inclusion of specific physical properties deserves special mention. There is a tendency among some philosophers to regard a system's physical composition — whether it is composed of silicon or sillyputty, for instance — as wholly irrelevant to its status as a computational system. In practice, however, there are often subtle relationships between a system's physical composition and its broader computational features. These relationships may be encoded in the description of a computational architecture, so that a physical system counts as a particular kind of architecture not only because it has the right functional, representational, etc. features, but also because it is made out of the right stuff. Let me explain.

One goal in computer architecture is to design systems with a certain level of performance. The most straightforward measure of perforamce is the number of seconds it takes to execute a program:

$$performance = \frac{seconds}{program}$$

We can unroll the right hand side somewhat. Programs are composed of instructions, and instructions take a certain number of (internal) clock cycles to be executed. Clock

cycles in turn take a certain amount of time, so this amounts to:

$$\frac{seconds}{program} = \frac{instructions}{program} \times \frac{clock\ cycles}{instruction} \times \frac{seconds}{clock\ cycle}$$

Now, because the number of instructions per program is typically outside of the computer architect's control, there are two main ways to improve performance: (1) increase the number of clock cycles per second, for instance by increasing the clock rate, which increases power consumption; (2) decrease the number of clock cycles per instruction, for instance by reducing the instruction set (so that individual instructions take fewer cycles), or increasing parallelization.

Quite often these two improvements are complementary, and this has contributed to the massive improvements in microprocessor performance witnessed in the last fifty or so years. In some cases, however, they pull in different directions. For instance, increased CPU miniaturization means that more transistors can be packed into the same amount of space. However, increasing transistor density worsens heat dissipation. This is a problem, because errors become more likely as CPU core temperatures increase. Slower clock rates decrease power consumption, but at the cost of decreased performance. However, this can be offset somewhat by varying the thermal properties of the silicon chip (Sohel Murshed and Nieto de Castro, 2017) or by tinkering with the chip's architecture (Azizi et al., 2010), among other things.

The point is that there can be intimate connections between multiple levels of a system's computational organization, from its physical composition to its datapath and memory organization, and these connections are reflected in certain physical interpretations. Of course, the mathematical core of a given CPU microarchitecture might be given a different physical interpertation — in terms of cogs and gears, for instance. But realizations of *that* interpretation would operate much more slowly.

Only certain kinds of physical substrate will be able to meet high performance demands, and this is reflected in the specific choice of physical interpretation for the microarchitecture core.

As a final note, observe that computational architectures can be that more or less fine-grained. This can be because the mathematical core is specified in more or less detail, or because the physical interpretation is more or less exacting, or both. A bare description of a Turing machine as a set of states plus transition function imposes far fewer conditions on its physical realizations than does a microarchitecture specification that specifies a highly parallelized datapath, multiple cache levels, a certain number of cores, and specific kind of silicon substrate, and so on. As one would expect, more exacting architectures will be realized by fewer system than less exacting ones.

5.4 Resemblance

So much for computational architectures. Next up: resemblance. To a very rough first approximation, the idea pursued here is that a physical system resembles a computational architecture to the extent that it (a) has features 'specified' by the architecture, and (b) lacks features not so specified. A physical system resembles an Turing machine, for instance, to the extent that it has a read/write head, a control unit, tape, and so on. I'll start by motivating the use of resemblance. Then, I'll draw on recent work in the philosophy of science to offer a precise account of resemblance.

5.4.1 Why resemblance?

As we have seen, perhaps the most serious alternative to a resemblance-based account of implementation is an account based on isomorphism. This is the approach

adopted by the simple mapping account and others. There are certain advantages to working with the broader notion resemblance, however. I'll mention three.[65]

Isomorphism can be too demanding

Perhaps the most serious concern is that isomorphism is in some cases too demanding a requirement on implementation. In the previous chapter, I argued that limitative explanations depend upon the implementation of highly idealized computational models such as Turing machines or register machines, but that it is questionable whether the relevant isomorphisms obtain between ordinary physical computing systems and such architectures. If this is right, then whatever makes it the case that these systems implement Turing machines, it isn't the fact that the two are isomorphic.

In fact, I think this conclusion can be motivated on much more mundane grounds. The reason is that physical computing systems suffer from the usual travails of physical existence: among other things, they are subject to environmental noise, defects may occur in the manufacturing or development process, and they accumulate damage over time and decay (Anderson, 2019). Sometimes these defects are catastrophic, but not always; a few dead transistors in a little-used part of a chip likely won't brick a device. To illustrate, consider a microprocessor and the microarchitecture specification it is based on. If we conceive of microarchitecture specifications as highly detailed formal structures, detailing the placement of each transistor, then, owing

[65]This is in many respects an instance of a larger debate about the nature of scientific modeling. Many philosophers have suggested that the relationship between scientific models and physical systems is one of isomorphism or homomorphism (French and Ladyman, 1999; da Costa and French, 2000; Pincock, 2011). Suarez (2003) critically assesses the alternatives. More recently, others have suggested that the relationship is better understood in terms of resemblance or similarity (Giere, 1988; Godfrey-Smith, 2006; Weisberg, 2013). While I do not claim that scientific modeling generally ought to be understood in terms of resemblance, I do think that it is a natural tool for understanding the relationship between computational architectures and the physical systems that implement them.

to damage and defects, there will typically be no one-one, structure-preserving map from the physical transistors to the microarchitecture.[66] Despite this, few computer scientists would deny that the microprocessor implements a certain microarchitecture specification.

In general, there is frequently more slack between physical computing systems and the computational architectures they implement than is recognized by the suggestion that implementation is isomorphism. Working in terms of resemblance allows us to recognize this slack while nevertheless maintaining a connection between systems and the architectures they implement.

Gradability and fine-grained judgments

Isomorphism-based accounts of implementation treat implementation as a yes/no matter: either an appropriate isomorphism obtains, or it does not. Working in terms of resemblance, by contrast, allows us to make finer-grained judgments about what computational architectures a system implements. Because resemblance is always resemblance to a certain degree and in certain respects, we are able to capture the sense in which a system more faithfully implements one architecture instead of another. For instance, although there is a sense in which a microprocessor implements a Turing machine, there is also a sense in which it more faithfully implements a register machine, and a sense in which it even more faithfully implements a microarchitecture specification. It is not clear that an isomorphism-based account has the resources to

[66]In fact, there is typically substantial token-level variation between individual microprocessors. Contemporary microprocessors contain redundant components. If in the course of quality-control testing it is found that one component has failed, it is possible to toggle a backup which achieves the same effect (this can happen either at the hardware level or through software). Indeed, different versions of a single microprocessor family are often built from the same specification, but with certain components disabled. For instance, chips in the intel family (i3, i5, etc.) have the same basic architecture, but less powerful chips have cores disabled, while more powerful chips do not.

capture this, whereas it is straightforward to do so in terms of resemblance. Working with resemblance thus allows us to make a wider range of judgments about what a system implements, and at a finer grain of analysis than otherwise.

Flexibility

A third point is that the notion of resemblance is highly flexible. This is advantageous for a couple different reasons. For one thing, given the wide variety of computational architectures found in contemporary computer and cognitive science, it is unlikely that there will be a straightforward one-size-fits-all approach that captures the implementation conditions for every computational architecture. Rather, different computational architectures will have different implementation conditions, depending on the particular mix of features they specify. While this may in many cases involve isomorphisms, it need not in every case.

Of course, there are no free lunches. Because the notion of resemblance is so flexible, it turns out that the resemblance account does not have a much to say about what implementation consists in generally. Consequently, much of the interesting work happens at the level of individual computational architectures or architecture kinds, where we focus on the particular features required to implement that particular (kind of) architecture. Although some philosophers will be displeased with this, I don't think it's problem. For one thing, in Chapter 3 we saw that something like this is a natural consequence of my response to triviality arguments. I'll come back to this brief below.

Second, because it is highly flexible, nothing is lost by moving to a resemblance-based framework. If we wish, we can require that one of the respects in which physical computing systems resemble architectures is that they are isomorphic. Moreover, as

I will suggest below, it is possible to define a binary yes/no notion of implementation within the resemblance framework. So while the resemblance account goes beyond more traditional, isomorphism-based accounts, it has the resources to account for their insights, too.

5.4.2 The relativity of implementation, redux

At a few points in the discussion I've mentioned that physical systems should not be taken to resemble architectures *tout court*, but rather resemble them in certain predetermined respects. What is the rationale for this, and what fixes the relevant respects? Notice first that, Goodman (1972) points out, everything resembles everything else in *some* respect or other. Absent some restriction on the respects in which a physical system might resemble a computational architecture, we encounter the potentially problematic consequence that every physical system will implement every computational architecture. Restriction makes resemblance substantive.

As noted in Chapter 3, when computational scientists wish to describe or explain a system computationally, they do so in a particular investigative context. This context accomplishes a few different tasks. One is fixing a particular conceptualization of the physical system or systems under consideration. This includes a non-computational description of certain select properties and/or behaviors of the system, typically just those that are relevant, or at least believed to be relevant, to whatever it is about the system that is to be described or explained computationally. The context also delineates a range of possible computational architectures with which to characterize that

system computationally. The choice of both conceptualization and candidate architectures is guided to a large extent by the theoretical goals, interests, and standards of the researchers in question.

Thus an investigative contexts delimits a range of features relative to which computational scientists judge the degree of fit between a system and a computational architecture. That is, it fixes a 'way of regarding' a system computationally (or ways, if there is more than one possibility). In the context of isomorphism-based accounts, such ways were called labelling schemes. For resemblance-based accounts, I'll call the set containing these contextually salient features the 'resemblance base'. The members of the resemblance base will typically be the kinds of features encountered earlier: the properties of the physical system we wish to describe or explain computationally, including its (non-computationally described) structure and organization the descriptively or explanatorily salient properties of the computational architecture(s) used to characterize the system.

On this approach implementation is fundamentally a relative notion: a physical system implements a computational architecture relative to a contextually specified resemblance base. Relative to a different resemblance base, a system may more closely resemble a different architecture, or no architecture at all. This is as it should be. As we have seen, different computational architectures are appropriate for different descriptive and explanatory purposes. An account of physical computation should have the resources to capture this.

5.4.3 Resemblance scores

Some philosophers might be content to rest with an intuitive, or commonsensical, notion of resemblance. This is fine as far as it goes, but there are benefits to working with a more precise account. Different formal accounts have been proposed in the literature, some of which depart more substantially than a more traditional isomorphism-based approach than others. For instance, French and Ladyman (1999) suggest that resemblance can be understood in terms of partial isomorphism. Here I will develop a view of resemblance based on Weisberg's weighted feature-matching account of similarity Weisberg (2012, 2013). However, I am quite open to the idea that we could develop a resemblance-based account of implementation in other terms (e.g., in terms of partial isomorphisms).

As I noted, resemblance always determined relative to a distinguished class of properties in a resemblance base B. If C is a computational architecture and P is a physical system, we'll let $B_C \subseteq B$ be the features specified by C and $B_P \subseteq B$ be the set of features of P. Then we can say that P resembles C, with respect to F, to degree n just in case

$$|B_C \cap B_P| - |B_C - B_P| - |B_P - B_C| = n$$

For convenience, I'll write $Res(B, C, P) = n$. Here $B_C \cap B_P$ are the features shared by C and P. $B_C - B_P$ are the features specified by C which P lacks. And $B_P - B_C$ are the features had by C not specified by C. This equation, in effect, measures the extent to which P 'fits' C's specification.

This account treats resemblance as a graded notion. I think this is the most basic notion of resemblance, but as I mentioned earlier we can define other notions

as the need arises. For instance, we can say that P *perfectly resembles* C just in case $Res(B, C, P) = n$ and $|B_C \cap V_P| = n$. Similarly, we can say that C and P resemble each other simpliciter just in case $Res(B, C, P) \geq m$, for some predetermined 'cutoff' degree of resemblance m. The notion of resemblance simpliciter can be used to capture the kind of binary yes/no judgment furnished by isomorphism-based accounts of implementation.

Exactly what degree of resemblance is required for implementation? On the one hand, perfect resemblance seems too exacting: it is useful to allow that a physical system may implement an architecture even when they don't perfectly resemble each other. On the other, too low a degree threatens to trivialize the notion of physical computation: there are plausibly some simple physical or functional properties shared by paradigmatically non-computing systems and any computational architecture. This is a non-trivial problem, and I do not have a satisfactory answer to it at this point. It remains on the agenda for future work. For the time being, I will simply say that implementation requires a 'sufficiently high' degree of resemblance, noting that this is just a placeholder for what will undoubtedly be a complicated theory of just what a 'sufficiently high' degree amounts to.

5.5 Further issues

This concludes my discussion of the core of the resemblance account. In the space remaining I will fill out the account in certain respects.

5.5.1 Do we need to provide uniform implementation conditions?

The resemblance account declines to give much in the way of an informative general story about implementation. Instead, it holds that we should focus on the implementation conditions of specific architectures or architecture kinds, in the specific scientific contexts in which they are deployed. Some philosophers may be unhappy with this approach. For instance, Rescorla holds that a theory of implementation should furnish a "canonical notation that can uniformly express the descriptive content of any computational model" (Rescorla, 2014b, 1280), and introduces the notion of a canonical state-space description for this purpose. In a similar spirit, Chalmers writes that "the theory of implementation for combinatorial-state automata provides a basis for the theory of implementation in general" (Chalmers, 2011, 331).

The expectation underwriting these remarks is that it is possible, at least in principle, to redescribe any given computational model in terms of some designated canonical formalism — canonical state-space descriptions for Rescorla, and combinatorial-state automata for Chalmers. Strikingly, however, both Rescorla and Chalmers say little about the translation procedure. Chalmers claims, but does not argue, that in many cases the translation will be straightforward (Chalmers, 2011, 330). Rescorla calls the translation an 'inducement relation', and declines to say when a given a computational architecture induces a given canonical state-space description. Instead, Rescorla "treat[s] the 'inducement relation' as primitive" on the grounds that he is "not pursuing a reductive analysis," but is instead "trying to offer an illuminating

theory that respects how computational implementation figures within contemporary scientific practice" and so "may legitimately presuppose a primitive inducement relation" (Rescorla, 2014b, 1283).

It seems to me that this is a rather significant lacuna, especially given that there is no a priori guarantee that the translation can be carried out without descriptive or explanatory loss. For instance, Sprevak (2012) argues that in Turing machines there is an distinction between control element (i.e., its transition table) and functional architecture (i.e., read/write head and tape organization), but that this distinction is lost in translation into the CSA formalism. If Sprevak is right, Chalmers' account thus mischaracterizes the implementation conditions for Turing machines.[67]

What of Rescorla's refusal to give a 'reductive' analysis of implementation (whatever exatly that amounts to)? Of course, one is free to pursue whatever philosophical projects one likes. I have no argument that Rescorla should not attempt to provide a 'non-reductive' account, whatever exactly that comes to. Nonetheless, all else equal it seems to me that we should prefer a theory that does not rely on a primitive inducement relation. For, absent such an account, it is little more than an article of faith that a given canonical state-space description in fact captures the descriptive content of a given computational architecture. It is thus far from clear that results and properties framed in terms of one architecture apply to the canonical state-space description as well. Again, there is a threat that something might get lost in translation.

[67]Milkowski (2011) raises a similar worry. For a reply, see (Chalmers, 2012, 11-15).

One way to get around these worries is to provide a substantive account of the translation/inducement relation. Although nothing I've said precludes the development of such a theory, it remains an open issue whether one will be forthcoming any time soon. Another option is to avoid the issue altogether. This is the route taken by the resemblance account. That account concedes that there is much to be said about the implementation conditions of computational architectures generally, and instead suggests that we focus on the implementation conditions for particular architectures or architecture kinds. I take it to be a point in favour of the resemblance account that it avoids the awkward questions about translation confronting Chalmers' account, and that it needn't retreat to the 'non-reductive analysis' of Rescorla's.

5.5.2 Medium-independence and multiple realizability

On the resemblance account, to implement a give computational architecture a system may need to have quite specific physical features. For instance, many contemporary microarchitectures are designed with the expectation that they will be realized in a certain kind of silicon wafer. This may seem to cut against the idea, widely endorsed, that computations are medium-independent (Haugeland, 1985; Piccinini, 2015). I next explain how a notion of medium independence can be developed within the resemblance framework.

To a first approximation, a property or process is medium independent if it can be realized in different physical media. *Cooking lentils* is not medium independent, because it can be realized in only quite specific physical media; *powering a drive-train* is, because it can be accomplished by otherwise quite different physical systems (internal combustion engines, electric motors, etc.).

Medium independence is closely related to multiple realizability. A property or process is multiply realizable if, roughly, it can be realized by different kinds of physical systems. Medium independence entails multiple realizability: if a property or process is medium independent, then it can be realized in different physical media. Note, however, that the converse fails. *Being a corkscrew* is multiply realizable, since many different corkscrew designs might do the trick, but not medium independent. *Being a corkscrew* is a matter of interacting with a specific physical medium (or, at least, a restricted range of physical media), namely cork (both artificial and synthetic).

It seems undeniable that computations are medium independent, hence multiply realizable. Recall Block's (1995) remark an AND-gate might be realized either by transistors, or by mice, string, and cheese. Moreover, the literature on unconventional computation is replete apparent cases in which the same computation (e.g., a sorting task) is performed by quite different physical systems. But it's not clear that the resemblance account can capture this apparent datum. The trouble is that there appears to be no single computational architecture, in the above sense, common to the wide variety of computing systems hypothesized by computer scientists. There is no architecture, for instance, which both silicon and murine AND gates resemble.

What should the resemblance theorist make of this? The solution, I think, becomes clear once we reflect on the role played by medium independence (or multiple realizability) in computational theorizing. In general we require a way to describe physical systems that abstracts away from (some of) their physical details. So abstracted, we can consider whether, e.g., two systems compute the same logical function despite physical dissimilarities. Now, ordinarily this role is played by the alleged medium independence (multiple realizability) of computations. But if the resemblance account

can supply a way to abstract from physical details, it can furnish a way to describe physical systems at the desired level of abstraction. And this, I submit, is just what is needed for computational theorizing. Let me sketch how this might go.

To begin, while above resemblance is characterized in terms of a single class of features, we can also define a notion of resemblance that discriminates between different classes of features. If $B_1, B_2, ..., B_m$ are classes of features, then we can say that P resembles C with respect to the B_i $(1 \leq i \leq m)$, to degree n just in case $\sum_{i=1}^{m} Res(B_i, P, C) = n$. Moreover, by adding coefficients we can discount (or boost) the contribution of a particular class of features to the overall resemblance score. Doing so gives

$$x(|B_C \cap B_P| - |B_C - B_P| - |B_P - B_C|) = n$$

I will write $Res(B, x, P, C) = n$ as shorthand. In general, then, if $x_i \in R$, $(1 \leq i \leq m)$ are coefficients, the generalized resemblance score is given by $\sum_{i=1}^{m} Res(B_i, x_i, P, C) = n$.

By appropriately choosing coefficients we can define a notion of pattern resemblance between systems. For instance, if B_1, B_2, B_3 are classes of physical, functional, and semantic features, respectively, with corresponding coefficients x_1, x_2, x_3, then by setting $x_1 = x_3 = 0$ and setting $x_2 = 1$ we can say that P *pattern resembles* C to degree n just in case $\sum_{u=1}^{3} Res(B_i, x_i, P, C) = n$, that P pattern resembles C simpliciter just in case P pattern resembles C to a high enough degree, and so on.

How does all this help? Medium independence is naturally thought to concern what I've called functional features. We say that silicon and mouse-and-string systems compute AND, when they do, because at a certain abstract level of description they

exhibit the same patterns, regardless of their physical substrate. The resemblance account can accommodate this fact by noting that different AND gates pattern resemble each other. Thus the resemblance account can furnish a level of description appropriate for this part of computational theorizing, without abandoning the insight that inclusion of specific physical features is an important part of computational practice.

5.5.3 Computational explanation

In Chapter 4 I argued that limitative explanations are non-causal explanations, and that dominant theories of computational explanation do not capture their explanatory power. There thus seem to be at least two kinds of computational explanation: causal and non-causal. This final section connects them by sketching a view of computational explanation that naturally emerges from the resemblance account.

To begin, the fact that a system implements a certain computational architecture bears on the causal processes it can go through. If a system implements a Turing machine, then (at one level of description) it is capable of the kinds of causal processes typical of such machines: roughly, step-by-step processes according to finite effective instructions. If, by contrast, a system implements a neural network architecture, then (at the appropriate level of description) it is capable of different sorts of causal processes: roughly, the kind captured by patterns of activation in a neural network. Thus, given that a system implements a certain architecture, it is possible to causally explain some property or behavior of a system by appeal to the specific causal processes sanctioned by that architecture. In this way the notion of a computational architecture underwrites causal computational explanations.

However, reflection on an architecture itself allows us to identify limits on its computational powers, and thereby on the limits of any physical system that implements it. Reflection on the kinds of operations supported by Turing machines, for instance, allows us to reason about what kinds of problems can be solved (in principle or in practice) using only such operations. As I have been at pains to emphasize, identifying such limits is a central part of contemporary computer science, an information about such limits underwrites the wide applicability of limitative results.

Incidentally, although the focus in chapter 4 was on Turing-complete architectures, the view sketched here straightforwardly generalizes, in two directions. On the one hand are architectures strictly weaker than Turing machines, such as finite automata or pushdown automata. On the other are architectures strictly stronger than Turing machines, either in respect of which problems they can solve tractably (this may include quantum computing architectures; see e.g. (Nielsen and Chuang, 2010) or in respect of which problems they can solve in principle [e.g., through various forms of hypercomputation; see (Copeland, 2002). The primary difference between these and more familiar Turing-complete architectures concerns the character of the computational architecture involved (e.g., how much memory is available, or what kinds of basic operations are supported). Nonetheless, in these cases too we can explain systems causally by considering the kinds of causal computational processes they support, and non-causally by considering the limits on those processes imposed by the system's overall architecture.

Thus information about what architecture(s) a system implements serves two distinct but intimately related explanatory goals: it provides information about what

a system can do, computationally speaking, but it also provides information about what it cannot so do.

5.6 What's next?

The goal of this dissertation was to develop a preliminary argument for the resemblance account. One theme of my discussion that philosophical thinking about physical computation often overemphasizes the importance of global, context-insensitive conditions on implementation. Although such conditions are not unimportant, much work remains to be done illuminating how computational scientists use computational architectures in specific investigative contexts for particular descriptive or explanatory purposes. Work on physical computation would benefit from a wider and more systematic survey of the uses of computation in the computational sciences. I'll close by mentioning a few potential avenues for future work.

First, much work remains to be done understanding how computer engineers apply computational notions in the design of artificial computing systems such as microprocessors. Of particular interest are the sorts of tradeoffs in performance, power-consumption, etc. mentioned earlier in this chapter that arguably bear on the implementation conditions of specific architectures. Especially interesting in this connection is work on unconventional computing, which presents a variety of novel computing systems which have received scant philosophical attention. Examining these systems and understanding in what sense, if any, they compute, will further our understanding of physical computation. The resemblance account is a natural framework in which to pursue this work.

Second, my discussion of limitative explanations in Chapter 4 lumped together in-principle uncomputability results on the one hand and intractability results on the other. However, it seems to me that there are important differences between these cases. For instance, what it is for a problem to be solved 'tractably' is philosophically contentious (Dean, 2021), and it seems to me that better understanding the notion of tractability would illuminate how intractability results are applied to physical computing systems, and what such applications show.

Finally, in an effort to more directly engage with computer scientific practice, I have not discussed the computational theory of mind. However, a fuller development of the theory should say something about this. It remains to be seen whether the resemblance account can furnish a notion of physical computation adequate for the computational theory of mind.

Bibliography

A. Adamatzky. Slime mould processors, logic gates and sensors. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 373(2046):20140216, July 2015.

H. Akhlaghpour. An RNA-based theory of natural universal computation. *Journal of Theoretical Biology*, 537:110984, Mar. 2022.

N. G. Anderson. Information Processing Artifacts. *Minds and Machines*, 29(2):193–225, June 2019.

A. W. Appel and J. Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, Cambridge, UK; New York, NY, USA, 2002.

S. Arora and B. Barak. *Computational complexity: A Modern Approach*. Cambridge University Press, New York, 2009.

H. Attiya and F. Ellen. *Impossibility results for distributed computing*. Morgan & Claypool, 2014.

O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz. Energy-Performance Tradeoffs in Processor Architecture and Circuit Design: A Marginal Cost Analysis. *SIGARCH Comput. Archit. News*, 38(3):26–36, June 2010.

R. W. Batterman. On the Explanatory Role of Mathematics in Empirical Science. *The British Journal for the Philosophy of Science*, 61(1):1–25, Mar. 2010.

G. Berkeley. *The Works of George Berkeley*, volume 1. InteLex Corporation, Charlottesville, Va., 1999.

F. Berto and M. Jago. *Impossible Worlds*. Oxford University Press, Oxford, 2019.

J. Blackmon. Searle's Wall. *Erkenntnis*, 78(1):109–117, Feb. 2013.

N. Block. Troubles with functionalism. *Minnesota Studies in the Philosophy of Science*, 9:261–325, 1978.

T. Bontly. Individualism and the Nature of Syntactic States. *The British Journal for the Philosophy of Science*, 49(4):557–574, Dec. 1998.

G. Boolos, J. P. Burgess, and R. C. Jeffrey. *Computability and logic*. Cambridge University Press, Cambridge ; New York, 5th ed edition, 2007.

O. Bueno and M. Colyvan. An Inferential Conception of the Applicability of Mathematics. *Nous*, 45(2):345–374, 2011.

O. Bueno and S. French. *Applying Mathematics: Immersion, Inference, Interpretation*. Oxford University Press, 2018.

T. Burge. *Origins of Objectivity*. Oxford University Press, 2010.

J. P. Burgess. *Rigor and Structure*. Oxford University Press, Feb. 2015.

J. P. Burgess and G. Rosen. *A Subject with No Object: Strategies for Nominalistic Interpretation of Mathematics*. Oxford University Press, 1997.

A. R. Burks and A. W. Burks. *The first electronic computer: the Atanasoff story*. University of Michigan Press, Ann Arbor, 1988.

N. Cartwright. *How the laws of physics lie*. Clarendon Press ; Oxford University Press, Oxford : New York, 1983.

D. Chalmers. Does a Rock Implement Every Finite-State Automaton? *Synthese*, 108 (3):309–33, 1996.

D. Chalmers. A Computational Foundation for the Study of Cognition. *Journal of Cognitive Science*, 12:323–357, 2011.

D. Chalmers. The Varieties of Computation: A Reply. *Journal of Cognitive Science*, 13(3):211–248, Sept. 2012.

D. J. Chalmers. On implementing a computation. *Minds and Machines*, 4(4):391–402, 1994.

H. Chang. *Inventing temperature: measurement and scientific progress*. Oxford studies in philosophy of science. Oxford University Press, Oxford ; New York, 2007.

C. E. Cleland. Recipes, Algorithms, and Programs. *Minds and Machines*, 11:219–237, 2001.

C. E. Cleland. On Effective Procedures. *Minds and Machines*, 12:159–179, 2002.

F. Cohen. Computer viruses. *Computers & Security*, 6(1):22–35, Feb. 1987.

S. A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

S. B. Cooper. *Computability Theory*. Chapman & Hall/CRC, New York, NY, 2004.

B. J. Copeland. What is Computation? *Synthese*, 108(3):335–359, 1996.

B. J. Copeland. Hypercomputation. *Minds and Machines*, 12(4):461–502, 2002.

B. J. Copeland and O. Shagrir. Do Accelerating Turing Machines Compute the Uncomputable? *Minds and Machines*, 21(2):221–239, 2011.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

C. Craver and D. Kaplan. Are More Details Better? On the Norms of Completeness for Mechanistic Explanations. *The British Journal for the Philosophy of Science*, 71(1):287–319, Mar. 2020.

C. Craver and J. Tabery. Mechanisms in Science. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2019 edition, 2019.

C. F. Craver. *Explaining the brain: mechanisms and the mosaic unity of neuroscience*. Clarendon Press, Oxford : New York : Oxford University Press, 2007.

A. Curtis-Trudel. Counterfactuals for Computation.

N. Cutland. *Computability, an Introduction to Recursive Function Theory*. Cambridge University Press, Cambridge, 1980.

N. da Costa and S. French. Models, Theories, and Structures: Thirty Years on. *Philosophy of Science*, 67:S116–S127, Sept. 2000.

M. Davis. *Computability & Unsolvability*. Dover, New York, 1982.

W. Dean. Computational Complexity Theory, 2021.

J. Dewhurst. Individuation without Representation. *The British Journal for the Philosophy of Science*, 69(1):103–116, 2018.

F. Egan. Computational models: a modest role for content. *Studies in History and Philosophy of Science Part A*, 41(3):253–259, Sept. 2010.

F. Egan. Metaphysics and Computational Cognitive Science: Lets Not Let the Tail Wag the Dog. *Journal of Cognitive Science*, 13(1):39–49, Mar. 2012.

H. Field. Mental Representation. *Erkenntnis*, 13:9 – 61, 1978.

S. C. Fletcher. Computers in Abstraction/Representation Theory. *Minds and Machines*, 28(3):445–463, Sept. 2018.

J. Fodor. *Representations*. The MIT Press, 1981.

J. A. Fodor. Explanations in psychology. In M. Black, editor, *Philosophy in America*, pages 161–179. Cornell University Press, 1965.

J. A. Fodor. The Appeal to Tacit Knowledge in Psychological Explanation. *The Journal of Philosophy*, 65(20):627, Oct. 1968.

G. Frege. *The Foundations of Arithmetic*. Northwestern University Press, Evanston, Illinois, 1884.

S. French and J. Ladyman. Reinflating the semantic approach. *International Studies in the Philosophy of Science*, 13(2):103–121, July 1999.

M. Friedman. Explanation and Scientific Understanding. *The Journal of Philosophy*, 71(1):5, Jan. 1974.

R. Frigg and S. Hartmann. Models in Science. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, spring 2020 edition, 2020.

M. R. Garey and D. S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-completeness*. Freeman, 1979.

R. N. Giere. *Explaining Science: A Cognitive Approach*. University of Chicago Press, Chicago, 1988.

C. Gillett. Why constitutive mechanistic explanation cannot be causal. *American Philosophical Quarterly*, 57(1):31–50, Jan. 2020.

S. Glennan. *The New Mechanical Philosophy*. Oxford University Press, Oxford, UK, 2017.

P. Godfrey-Smith. The Strategy of Model-Based Science. *Biology & Philosophy*, 21: 725–740, 2006.

P. Godfrey-Smith. Triviality arguments against functionalism. *Philosophical Studies*, 145(2):273 – 295, 2009.

N. Goodman. *Problems and Projects*. The Bobbs-Merrill Company, Inc., 1972.

J. D. Hamkins and A. Lewis. Infinite time Turing machines. *Journal of Symbolic Logic*, 65(2):567–604, June 2000.

D. M. Harris and S. L. Harris. *Digital Design and Computer Architecture*. Morgan Kaufmann, Waltham, MA, 2nd edition, 2013.

J. Haugeland. *Artificial intelligence: the very idea*. MIT Press, Cambridge, Mass, 1985.

M. Hemmo and O. Shenker. The physics of implementing logic: Landauer's principle and the multiple-computations theorem. *Studies in History and Philosophy of Science Part B: Studies in History and Philosophy of Modern Physics*, 68:90–105, Nov. 2019.

C. G. Hempel. *Aspects of Scientific Explanation and Other Essays in the Philosophy of Science*. The Free Press, 1965.

C. Hitchcock. Contrastive Explanation. In M. Blaauw, editor, *Contrastivism in philosophy*, number 39 in Routledge studies in contemporary philosophy, pages 11–35. Routledge/Taylor & Francis Group, New York, 2013.

Y. Ho and D. Pepyne. Simple Explanation of the No-Free-Lunch Theorem and Its Implications. *Journal of Optimization Theory and Applications*, 115(3):549–570, Dec. 2002.

J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Upper Saddle River, New Jersey, third edition edition, 2014.

R. Iemhoff. Intuitionism in the Philosophy of Mathematics. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, fall 2020 edition, 2020.

D. Joslin. Real realization: Dennett's real patterns versus Putnam's ubiquitous automata. *Minds and Machines*, 16(1):29 – 41, 2006.

P. Kitcher. Explanatory unification. *Philosophy of Science*, 48(4):507–531, 1981.

P. Kitcher. Explanatory unification and the causal structure of the world. In P. Kitcher and W. Salmon, editors, *Scientific Explanation*, volume 8, pages 410–505. Minneapolis: University of Minnesota Press, 1989.

C. Klein. Dispositional Implementation Solves the Superfluous Structure Problem. *Synthese*, 165(1):141 – 153, 2008.

D. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, Reading, Mass., second edition, 1998.

S. A. Kripke. *Wittgenstein on rules and private language: an elementary exposition*. Harvard Univ. Press, Cambridge, Mass, 10. print edition, 2000.

M. Lange. What Makes a Scientific Explanation Distinctively Mathematical? *The British Journal for the Philosophy of Science*, 64(3):485–511, Sept. 2013.

M. Lange. *Because Without Cause: Non-Causal Explanations in Science and Mathematics*. Oxford University Press, 2017.

M. Lange. Because without Cause: Scientific Explanations by Constraint. In A. Reutlinger and J. Saatsi, editors, *Explanation beyond causation: philosophical perspectives on non-causal explanations*, pages 15–38. Oxford University Press, Oxford, United Kingdom, first edition edition, 2018.

D. K. Lewis. *Counterfactuals*. Blackwell, 1973.

W. G. Lycan. Form, Function, and Feel. *The Journal of Philosophy*, 78(1):24, Jan. 1981.

C. Maley. Analog Computation and Representation. *The British Journal for the Philosophy of Science*, page 715031, Apr. 2021.

R. J. Matthews and E. Dresner. Measurement and Computational Skepticism. *Nous*, 51(4):832–854, 2017.

M. Milkowski. Beyond Formal Structure: A Mechanistic Perspective on Computation and Implementation. *Journal of Cognitive Science*, 12(4):361–383, Dec. 2011.

T. Millhouse. A Simplicity Criterion for Physical Computation. *The British Journal for the Philosophy of Science*, 70(1):153–178, 2019.

M. Minsky and S. Papert. *Perceptrons: an Introduction to Computational Geometry*. MIT Press, Cambridge, Mass, 1988.

Y. N. Moschovakis. What is an Algorithm? In B. Engquist and S. Wilfried, editors, *Mathematics Unlimited - 2001 and Beyond*, pages 919–936. Springer-Verlag, Berlin, 2001.

M. A. Nielsen and I. L. Chuang. *Quantum computation and quantum information*. Cambridge University Press, Cambridge ; New York, 10th anniversary ed edition, 2010.

C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, Mass, 1994.

C. Peacocke. Content, Computation and Externalism. *Philosophical Issues*, 6:227–264, 1995.

C. Peacocke. Computation as Involving Content: A Response to Egan. *Mind and Language*, 14(2):195–202, 1999.

G. Piccinini. Computational modelling vs. Computational explanation: Is everything a Turing Machine, and does it matter to the philosophy of mind? *Australasian Journal of Philosophy*, 85(1):93–115, Mar. 2007.

G. Piccinini. Computation without Representation. *Philosophical Studies*, 137(2): 205–241, Jan. 2008.

G. Piccinini. The Mind as Neural Software? Understanding Functionalism, Computationalism, and Computational Functionalism. *Philosophy and Phenomenological Research*, 81(2):269–311, 2010.

G. Piccinini. *Physical Computation: A Mechanistic Account*. Oxford University Press, Oxford, UK, 2015.

G. Piccinini and N. G. Anderson. Ontic Pancomputationalism. In S. C. Fletcher, editor, *Physical Perspectives on Computation, Computational Perspectives on Physics*, pages 23–38. Cambridge University Press, May 2018.

C. Pincock. Mathematical Idealization. *Philosophy of Science*, 74(5):957–967, Dec. 2007a.

C. Pincock. A Role for Mathematics in the Physical Sciences. *Nous*, 41(2):253–275, June 2007b.

C. Pincock. *Mathematics and scientific representation*. Oxford University Press, Oxford ; New York, 2011.

H. Poincaré. *The Foundations of Science*. Cambridge University Press, 2015.

H. Putnam. Minds and Machines. In *Mind, Language, and Reality*, volume 2 of *Philosophical Papers*, pages 362–385. Cambridge University Press, Cambridge, MA, 1975.

H. Putnam. *Representation and Reality*. MIT Press, 1987.

W. V. O. Quine. *Word and object*. MIT Press, Cambridge, Mass, 1960.

W. J. Rapaport. Implementation is Semantic Interpretation. *The Monist*, 82(1): 109–130, 1999.

M. Rescorla. Against Structuralist Theories of Computational Implementation. *The British Journal for the Philosophy of Science*, 64(4):681–707, Dec. 2013.

M. Rescorla. The Causal Relevance of Content to Computation. *Philosophy and Phenomenological Research*, 88(1):173–208, 2014a.

M. Rescorla. A theory of computational implementation. *Synthese*, 191(6):1277–1307, Apr. 2014b.

M. Rescorla. The Representational Foundations of Computation. *Philosophia Mathematica*, 23(3):338–366, Oct. 2015.

M. Rescorla. The Computational Theory of Mind. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, spring 2017 edition, 2017a.

M. Rescorla. From Ockham to Turing – and Back Again. In J. Floyd and A. Bokulich, editors, *Philosophical Explorations of the Legacy of Alan Turing: Turing 100*, pages 279–304. Springer International Publishing, Cham, 2017b.

J. B. Ritchie and G. Piccinini. Computational Implementation. In M. Sprevak and M. Colombo, editors, *Routledge Handbook of the Computational Mind*, pages 192 – 204. Routledge, London, 2019.

H. Rogers. *Theory of recursive functions and effective computability*. MIT Press, Cambridge, Mass, 1st mit press pbk. ed edition, 1987.

D. E. Rumelhart, J. McClelland, and P. R. Group. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Bradford Bks. MIT Press, 1987.

J. E. Savage. *Models of Computation: Exploring the Power of Computing*. 2008.

M. Scheutz. Computational versus Causal Complexity. *Minds and Machines*, 11(4): 543–566, 2001.

P. Schweizer. Computation in Physical Systems: A Normative Mapping Account. In M. V. d'Alfonso, International Association for Computing and Philosophy, Annual Meeting, and D. Berkich, editors, *On the cognitive, ethical, and scientific dimensions of artificial intelligence: themes from IACAP 2016*, pages 27–47. 2019a.

P. Schweizer. Triviality Arguments Reconsidered. *Minds and Machines*, 29(2):287–308, 2019b.

J. R. Searle. *The Rediscovery of the Mind*. MIT Press, 1992.

R. Sedgewick and P. Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley, Upper Saddle River, NJ, 2nd ed edition, 2013.

O. Shagrir. Content, Computation and Externalism. *Mind*, 110(438):369–400, 2001.

O. Shagrir. In defense of the semantic view of computation. *Synthese*, 197:4083–4108, 2020.

O. Shagrir. *The nature of physical computation*. Oxford studies in philosophy of science. Oxford University Press, New York, NY, United States of America, 2022.

S. Shapiro. Acceptable notation. *Notre Dame Journal of Formal Logic*, 23(1):14–20, 1982.

J. C. Shepherdson and H. E. Sturgis. Computability of Recursive Functions. *Journal of the ACM*, 10(2):217–255, Apr. 1963.

W. Sieg. On Computability. In A. Irvine, editor, *Philosophy of Mathematics*, pages 535–630. North-Holland, Burlington, MA, 2009.

W. Sieg and J. Byrnes. K-graph machines: generalizing Turing's machines and arguments. In P. Hájek, editor, *Gödel '96: Logical foundations of mathematics, computer science and physics*, volume Volume 6, pages 98–119. Springer-Verlag, Berlin, 1996.

M. Sipser. *Introduction to the Theory of Computation*. Cengage, 2013.

R. I. Soare. Computability and Recursion. *Bulletin of Symbolic Logic*, 2(03):284–321, 1996.

S. Sohel Murshed and C. Nieto de Castro. A critical review of traditional and emerging techniques and fluids for electronics cooling. *Renewable and Sustainable Energy Reviews*, 78:821–833, Oct. 2017.

P. Spirtes, C. Glymour, S. N, and Richard. *Causation, Prediction, and Search*. Mit Press: Cambridge, 2000.

M. Sprevak. Computation, individuation, and the received view on representation. *Studies in History and Philosophy of Science Part A*, 41(3):260–270, 2010.

M. Sprevak. Three Challenges to Chalmers on Computational Implementation. *Journal of Cognitive Science*, 13(2):107–143, June 2012.

M. Sprevak. Triviality arguments about computational implementation. In M. Sprevak and M. Colombo, editors, *Routledge Handbook of the Computational Mind*, pages 175 – 191. Routledge, London, 2019.

M. Steiner. *The Applicability of Mathematics as a Philosophical Problem*, volume 109. Harvard University Press, 1998.

S. P. Stich. *From Folk Psychology to Cognitive Science: The Case Against Belief*. MIT Press, 1983.

M. Suarez. Scientific representation: against similarity and isomorphism. *International Studies in the Philosophy of Science*, 17(3):225–244, Oct. 2003.

C. Swoyer. How ontology might be possible: Explanation and inference in metaphysics. *Midwest Studies in Philosophy*, 23(1):100–131, 1999.

J. Szangolies. The Abstraction/Representation Account of Computation and Subjective Experience. *Minds and Machines*, 30(2):259–299, June 2020.

G. Tanaka, T. Yamane, J. B. Héroux, R. Nakane, N. Kanazawa, S. Takeda, H. Numata, D. Nakano, and A. Hirose. Recent advances in physical reservoir computing: A review. *Neural Networks*, 115:100–123, July 2019.

P. Teller. Twilight Of The Perfect Model Model. *Erkenntnis*, 55:393–415, 2001.

M. Thomson-Jones. Missing systems and the face value practice. *Synthese*, 172(2): 283–299, Jan. 2010.

A. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(1):230–265, 1936.

B. C. Van Fraassen. *Scientific Representation*. Oxford University Press, Oxford : New York, 2008.

S. Walsh and T. Button. *Philosophy and Model Theory*. Oxford University Press, 2018.

J. Warren. Killing kripkenstein's monster. *Noûs*, 54(2):257–289, 2020.

J. Waskan. Mechanistic explanation at the limit. *Synthese*, 183(3):389–408, Dec. 2011.

M. Weisberg. Getting Serious about Similarity. *Philosophy of Science*, 79(5):785–794, 2012.

M. Weisberg. *Simulation and Similarity: Using Models to Understand the World*. Oxford University Press, 2013.

N. H. E. Weste and D. M. Harris. *CMOS VLSI design: a circuits and systems perspective*. Addison Wesley, Boston, 4th ed edition, 2011.

T. Williamson. *The Philosophy of Philosophy*. Wiley-Blackwell, 2007.

D. Wolpert and W. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, Apr. 1997.

D. H. Wolpert. The Lack of A Priori Distinctions Between Learning Algorithms. *Neural Computation*, 8(7):1341–1390, Oct. 1996.

J. Woodward. *Making Things Happen: A Theory of Causal Explanation*. Oxford University Press, Oxford, UK, 2003.

J. Woodward. Mechanisms revisited. *Synthese*, 183(3):409–427, Dec. 2011.

J. Woodward. Mechanistic Explanation: Its Scope and Limits. *Aristotelian Society Supplementary Volume*, 87(1):39–65, 2013.